Jakarta EE Tutorial

v10

2026-10-31



This section was last updated for Jakarta EE Version 9.1

Legal

This documentation and the accompanying materials are made available under the terms of the Eclipse Public License v. 2.0, which is available at https://www.eclipse.org/legal/epl-2.0.

SPDX-License-Identifier: EPL-2.0

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

About this Tutorial

This tutorial explains how to use the features of the Jakarta EE Platform to build cloud-native applications, such as microservices, "right sized" services, and server-based web applications.

Prerequisites

Jakarta EE applications use the Java Platform, and are usually written in the Java programming language. All the examples in this tutorial are written in Java. If you're new to Java, spend some time getting up to speed on the language and platform; a good place to start is dev.java/learn.

Each topic in this tutorial provides some background information, but in general, we assume you have a basic understanding of the technologies each Jakarta EE feature works with. For example, in the Jakarta Persistence chapter, we assume you have a basic understanding of relational databases.

Conventions

Throughout this tutorial, we use the following typographic conventions:

Convention	Meaning	Example
Boldface	Boldface type indicates a term defined in text or graphical user interface elements associated with an action.	A **cache** is a copy stored locally. From the **File** menu, choose **Open Project**.
`Monospace`	Monospace type indicates the names of files and directories, commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

Convention	Meaning	Example
Italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.	Read Chapter 6 in the *User's Guide*. Do *not* save the file. The command to remove a file is `rm filename`.

Default Paths and File Names



This section was last updated for Jakarta EE Version 9.1

The following table describes the default paths and file names that are used in this book.

Placeholder	Description	Default Value

`as-install`	Represents the base installation directory for GlassFish Server.	Installations on the Linux, UNIX, or macOS:
		
		`user's-home-directory/glassfish6/glassfish`
		
		Windows, all installations:
		
		`SystemDrive:\glassfish6\glassfish`

`as-install-parent`	Represents the parent of the base installation directory for GlassFish Server.	Installations on UNIX/Linux/macOS:
		
		`user's-home-directory/glassfish6`
		
		Windows:
		
		`SystemDrive:\glassfish6`

`jakartaee-examples`	Represents the base installation directory for the Jakarta EE Tutorial examples project after you download or clone it.	`user's-home-directory/jakartaee-examples`
`domain-dir`	Represents the directory in which a domain's configuration is stored.	`as-install/domains/domain1`

Introduction

Overview

This chapter explains what Jakarta EE is, what features it provides, common terms, and how Jakarta EE applications are structured.

Jakarta EE 10 Platform Highlights

Jakarta EE 10 is the first feature release of the platform since it moved to the Eclipse Foundation in 2017. The key emphasis of this release is better alignment with Java SE and creation of a new lightweight Core profile, which can be easily consumed by other standards such as MicroProfile.

Here are some highlights:

- CDI Alignment
  - `@Asynchronous` in Jakarta Concurrency
  - Better CDI support in Jakarta Batch
- Java SE Alignment
  - Support for Java SE 11 and 17
  - `CompletionStage`, `ForkJoinPool`, parallel streams in Jakarta Concurrency
  - Bootstrap APIs for Jakarta REST
- Closing standardization gaps
  - OpenID Connect support in Jakarta Security
  - UUID as entity keys, more SQL support in Jakarta Persistence queries
  - Multipart/form-data support in Jakarta REST
  - `@ClientWindowScoped` and Facelet pure Java views in Jakarta Faces
  - New Core Profile that includes Jakarta CDI Light to enable next generation cloud native runtimes; used by MicroProfile
  - Jakarta Concurrency has moved to the Web Profile
- Deprecation/removal
  - `@Context` annotation in Jakarta REST
  - EJB Entity Beans
  - Embeddable EJB container
  - Deprecated features in Jakarta Servlet, Jakarta Faces, and Jakarta CDI

What is Jakarta EE?

Jakarta EE is a collection of services that help you write enterprise applications that run on the Java Platform. It provides the infrastructure often required for these applications so that you can focus

on core features and business logic.

Enterprise applications support the high levels of security, scalability, and reliability that large organizations typically require. They can be web services, web applications, batch processes, and so on.

The **Java Platform** consists of the Java Virtual Machine, compiler, standard APIs, and several tools that help you build, deploy, and debug Java applications. Java is the primary programming language, although the Java Platform supports several others, including Kotlin and Scala. Most Jakarta EE applications, however, are written in Java. The **Java Platform** is also called **Java Standard Edition (SE)**.

With Jakarta EE, you can pick and choose which services you need, and you use them with or without other frameworks and libraries. One framework commonly used with Jakarta EE is MicroProfile, which provides additional features commonly used in microservices.

A key benefit of Jakarta EE is that it's a set of open source standards supported by multiple vendors. Each vendor has their own implementation of the standards, and the vendors work together to update the standards over time. This means that you aren't locked into a particular vendor, and each standard is well-thought-out and based on existing patterns and practices.

Jakarta EE is stable and mature, and used in tens of thousands of mission-critical enterprises applications worldwide; yet it has evolved over time to keep up with modern computing trends, such as cloud computing.

Jakarta EE Services

Jakarta EE provides a wide range of services, organized into three main categories. Each category is implemented by one of three profiles. The Core profile contains the foundational services. The Web profile contains the Core profile plus services for writing web applications. The Platform contains the Core and Web profiles, plus additional services for mail, batch processing, and messaging, and more.

Services, Specificiations and Profiles lists each high-level service, the specification that provides it, and the profile that contains it:

Services, Specificiations and Profiles

Service	Specification	Profile
Dependency injection	Jakarta Contexts and Dependency Injection (CDI) Lite	Core
RESTful web services	Jakarta REST (formerly JAX-RS)	Core
JSON processing	Jakarta JSON Binding (JSON-B) and Jakarta JSON Processing (JSON-P)	Core
Advanced dependency injection	Jakarta Contexts and Dependency Injection (CDI) Full	Web

Service	Specification	Profile
Validation	Jakarta Validation (Bean Validation)	Web
Security	Jakarta Security	Web
HTTP request handling	Jakarta Servlet	Web
Server-side web applications	Jakarta Faces (formerly JavaServer Faces, or JSF)	Web
WebSocket request handling	Jakarta WebSocket	Web
Relational data persistence	Jakarta Persistence (formerly JPA)	Web
Application components	Jakarta Enterprise Beans Lite (formerly EJB Lite)	Web
Transactions	Jakarta Transactions	Web
Managed concurrency	Jakarta Concurrency	Web
Email handling	Jakarta Mail	Platform
Messaging	Jakarta Messaging	Platform
Batch processing	Jakarta Batch	Platform

As you work with Jakarta EE, you'll encounter other services that support the ones listed above.

Each service is provided by one or more APIs, which are defined in Jakarta EE specifications. The specifications provide details for users and implementors of the service. You can find a full list of the Jakarta EE specifications on the Jakarta EE site.

Jakarta EE Containers and Servers

The services we discussed in the Jakarta EE Services section are provided by **container**[1]. A **Jakarta EE Server** is software that runs the container, deploys applications into it, and provides administration and additional features. Jakarta EE Servers run stand-alone, but many implementations also support the ability to generate a single "fat", executable Java Archive (JAR) file that includes all the code necessary to run the application.

Jakarta EE Containers vs Docker Containers

Container is a commonly used term in computing, but it basically means "that which holds something". In the case of Jakarta EE, the container "holds," or more accurately, executes your code and provides low-level application services, such as dependency injection, transactions, HTTP request handling, REST support, and so on. In the case of Docker container (which is technically an Open Container Initiative, or OCI container) your application is executed inside it, but the container provides operating system services, such as disk and network I/O.

While the two container types are different, it's entirely possible (and common) to run a

Jakarta EE application inside a Docker container.

Figure 1, “Relationship between Code, Container, and Server” shows how the application code, container, and Jakarta EE Server are related:

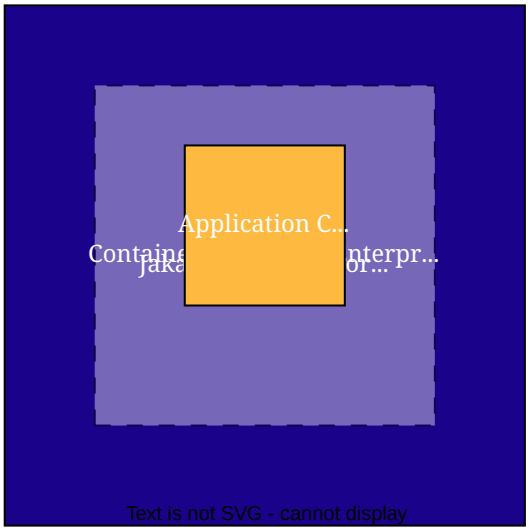


Figure 1. Relationship between Code, Container, and Server

In a simple case, an application might look like this:

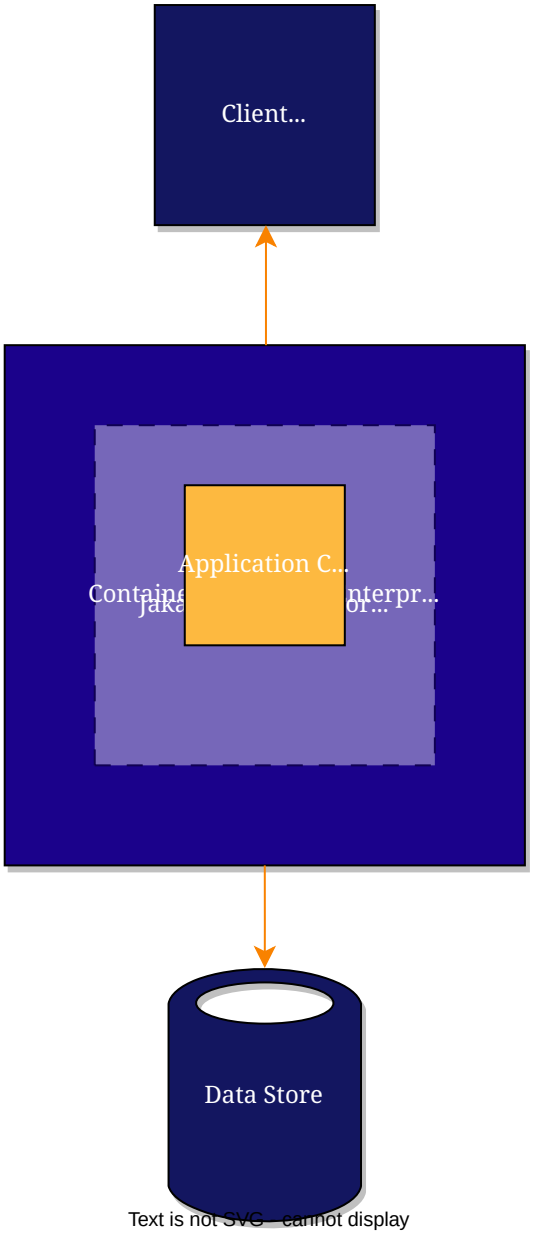


Figure 2. A Simple Jakarta EE Application

However, a single Jakarta EE Server can run multiple applications, and applications can communicate with each other remotely. So you can create more complicated scenarios, as shown in Figure 3, “A More Complex Jakarta EE Application”:

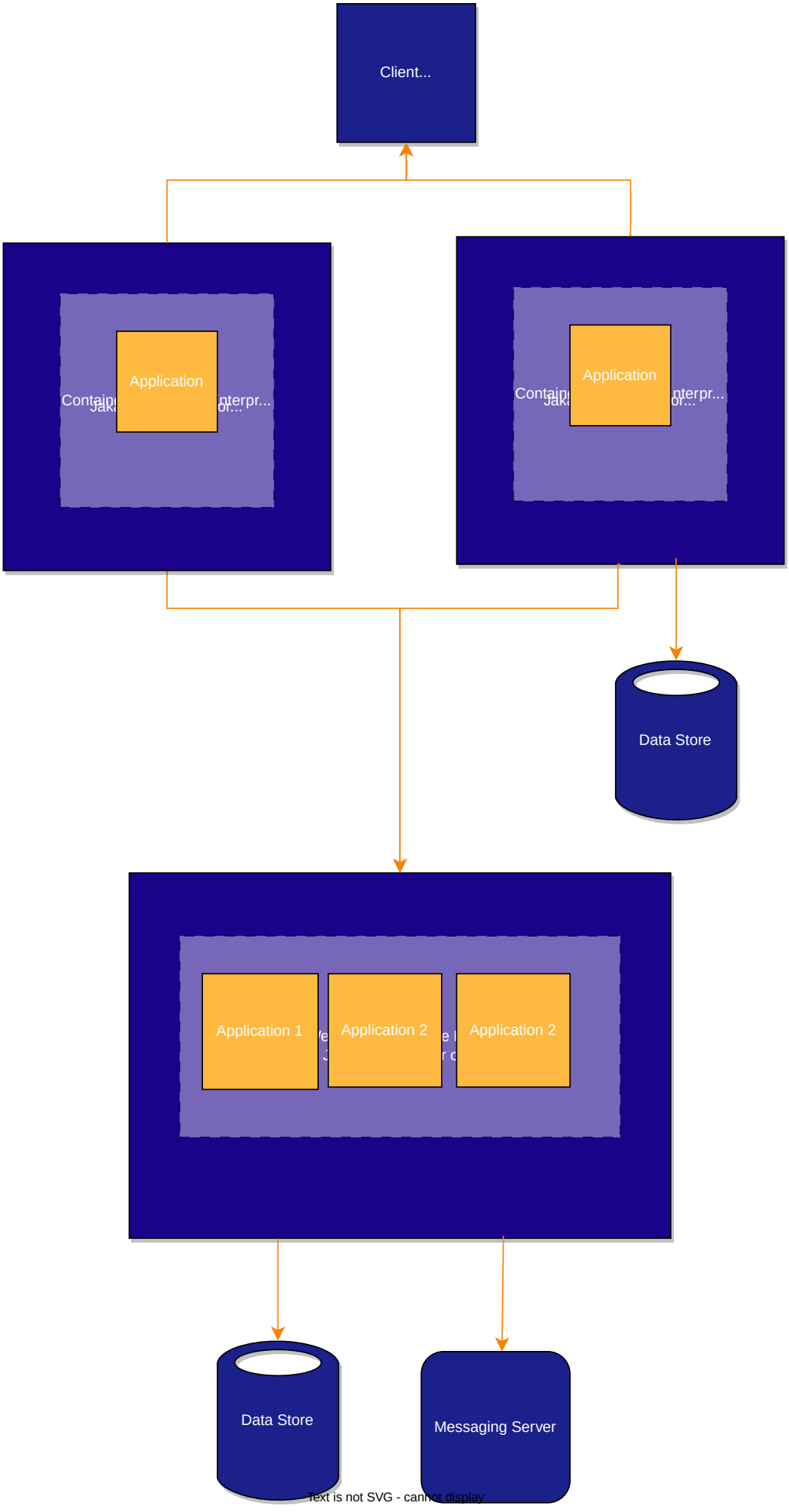


Figure 3. A More Complex Jakarta EE Application



Even though the figures above show communication between multiple Jakarta EE applications, they can and often do communicate with non-Jakarta EE applications via messaging or streaming servers, REST, and so on.

Regardless of how simple or complicated your system architecture is, a key benefit of Jakarta EE is that container services are configurable. This means the same component can behave differently based on where it's deployed. The container also manages non-configurable services, such as the lifecycle of components, database connection resource pooling, data persistence, and access to the Jakarta EE APIs.

Jakarta EE Components

Jakarta EE applications are made up of components. A **Jakarta EE component** is a self-contained functional unit that makes use of one or more container services. That usage could be as simple as injecting a single dependency or responding to REST requests, or as complicated as injecting multiple dependencies, querying a database, firing an event, and participating in distributed transactions.

Jakarta EE supports the following components:

- **Web components**[2] interact with web standards (HTTPS, HTML, WebSocket, and so on) using Jakarta Servlet, Jakarta Faces, Jakarta WebSocket, and related technologies.
- Business components implement logic necessary to support the business domain using either **Enterprise bean components (enterprise beans)**[3] or CDI managed beans.
    - For new applications, business components should be implemented using CDI managed beans, unless you need Enterprise bean-specific features such as transactions, role-based security, messaging driven beans, or remote execution. The two technologies play well together.

All of these components run on the **server**[4], and are ordinary Java classes written with Jakarta EE annotations (or optionally, external configuration files called deployment descriptors).

Usage of Java Standard Edition (SE) APIs

In addition to many of the core Java programming language APIs, there are a couple of key APIs you may encounter when working with Jakarta EE: Java Database Connectivity (JDBC) and Java Naming and Directory Interface (JNDI).

Java Database Connectivity API

The Java Database Connectivity (JDBC) API lets you work with SQL databases. You can use the JDBC API directly from within a Jakarta EE component, but most Jakarta EE applications use Jakarta Persistence to map between objects and database tables. Jakarta EE servers also support managed JDBC data sources, which can be configured for one or more applications.

Java Naming and Directory Interface API

The Java Naming and Directory Interface (JNDI) API allows you to work with multiple naming and

directory services, such as LDAP, DNS, and NIS. The API has methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a Jakarta EE application can store and retrieve any type of named Java object, allowing Jakarta EE applications to coexist with many legacy applications and systems.

Jakarta EE servers provide a naming environment for Jakarta EE components. This environment allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI naming context.

The naming environment provides four logical namespaces: `java:comp`, `java:module`, `java:app`, and `java:global` for objects available to components, modules, or applications or shared by all deployed applications. A Jakarta EE component can access named system-provided and user-defined objects. The names of some system-provided objects, such as a default JDBC `DataSource` object, a default Messaging connection factory, and a Transactions `UserTransaction` object, are stored in the `java:comp` namespace.

You can also create your own objects, such as enterprise beans, environment entries, JDBC `DataSource` objects, and messaging destinations.

How do I get Jakarta EE?

Since Jakarta EE is an open-source industry standard, there are multiple implementations. A good place to start is the Jakarta EE Starter, which lets you quickly generate a sample starter app using one of the supported Jakarta EE servers. You can also find the most recent list of servers on the Jakarta EE website.

If you are working with a Open Container Initiative (OCI)-compliant (i.e., Docker-compatible) containers, most of these vendors also provide images.

If you are using a cloud provider, you may also want to check to see if they have additional support, guides, or managed Jakarta EE servers.

Further Reading

- Jakarta EE Home Page
- Jakarta EE Specification Process
- Jakarta EE Specifications

Using the Tutorial Examples



This section was last updated for Jakarta EE Version 9.1

This chapter tells you everything you need to know to install, build, and run the tutorial examples.

For additional samples, see the GlassFish samples at https://github.com/eclipse-ee4j/glassfish-samples/tree/master/ws/jakartaee9

Required Software

The following software is required to run the examples:

- Java Platform, Standard Edition
- Eclipse Glassfish Server
- Jakarta EE Tutorial Examples
- Apache NetBeans IDE
- Apache Maven

Java Platform, Standard Edition

To build, deploy, and run the examples, you need a copy of the Java Platform, Standard Edition Development Kit (JDK). You must use JDK 8 Update 20 or above. You can download JDK software from https://www.oracle.com/technetwork/java/javase/downloads/index.html.

Eclipse Glassfish Server

GlassFish Server 6.0 is targeted as the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of GlassFish Server and, optionally, NetBeans IDE. You can download GlassFish Server from https://glassfish.org/download.

GlassFish Server Installation Tips

GlassFish Server is installed from a ZIP file. It sets the default administration user name as `admin` with no required password. The Admin Port is set to 4848, and the HTTP Port is set to 8080.

This tutorial refers to `as-install-parent`, the directory where you install GlassFish Server. For example, the default installation directory on Microsoft Windows is `C:\glassfish6`, so `as-install-parent` is `C:\glassfish6`. GlassFish Server itself is installed in `as-install`, the `glassfish` directory under `as-install-parent`. So on Microsoft Windows, `as-install` is `C:\glassfish6\glassfish`.

After you install GlassFish Server, add the following directories to your `PATH` to avoid having to specify the full path when you use commands:

```
as-install-parent/bin
as-install/bin
```

Jakarta EE Tutorial Examples

The tutorial example codes are located at https://github.com/eclipse-ee4j/jakartaee-examples

Clone or download this repository to your preferred location, this path is referenced in the tutorial as the *jakartaee-examples* directory.

Apache NetBeans IDE

The NetBeans integrated development environment (IDE) is a free, open-source IDE for developing

Java applications, including enterprise applications. NetBeans IDE supports the Jakarta EE platform. You can build, package, deploy, and run the tutorial examples from within NetBeans IDE.

To run the tutorial examples, you need the latest version of NetBeans IDE. You can download NetBeans IDE from https://netbeans.apache.org/download/index.html.

To Add GlassFish Server as a Server Using NetBeans IDE

To run the tutorial examples in NetBeans IDE, you must add your GlassFish Server as a server in NetBeans IDE. Follow these instructions to add GlassFish Server to NetBeans IDE.

1. From the **Tools** menu, choose **Servers**.
2. In the Servers wizard, click **Add Server**.
3. Under Choose Server, select **GlassFish Server** and click **Next**.
4. Under Server Location, browse to the GlassFish Server installation and click **Next**.
5. Under Domain Location, select **Register Local Domain**.
6. Click **Finish**.

Apache Maven

Maven is a Java technology-based build tool developed by the Apache Software Foundation and is used to build, package, and deploy the tutorial examples. To run the tutorial examples from the command line, you need Maven 3.0 or higher. If you do not already have Maven, you can install it from:

https://maven.apache.org

Be sure to add the `maven-install/bin` directory to your path.

If you are using NetBeans IDE to build and run the examples, it includes a copy of Maven.

Starting and Stopping GlassFish Server

You can start and stop GlassFish Server using either NetBeans IDE or the command line.

To Start GlassFish Server Using NetBeans IDE

1. Click the **Services** tab.
2. Expand **Servers**.
3. Right-click the **GlassFish Server instance** and select **Start**.

To Stop GlassFish Server Using NetBeans IDE

To stop GlassFish Server using NetBeans IDE, right-click the **GlassFish Server instance** and select **Stop**.

To Start GlassFish Server Using the Command Line

To start GlassFish Server from the command line, open a terminal window or command prompt and execute the following:

```
asadmin start-domain --verbose
```

A domain is a set of one or more GlassFish Server instances managed by one administration server. The following elements are associated with a domain:

- The GlassFish Server port number: The default is 8080.
- The administration server's port number: The default is 4848.
- An administration user name and password: The default user name is `admin`, and by default no password is required.

You specify these values when you install GlassFish Server. The examples in this tutorial assume that you chose the default ports as well as the default user name and lack of password.

With no arguments, the `start-domain` command initiates the default domain, which is `domain1`. The `--verbose` flag causes all logging and debugging output to appear on the terminal window or command prompt. The output also goes into the server log, which is located in `domain-dir/logs/server.log`.

To Stop GlassFish Server Using the Command Line

To stop GlassFish Server, open a terminal window or command prompt and execute:

```
asadmin stop-domain domain1
```

Starting the Administration Console

To administer GlassFish Server and manage users, resources, and Jakarta EE applications, use the Administration Console tool. GlassFish Server must be running before you invoke the Administration Console. To start the Administration Console, open a browser at http://localhost:4848/.

To Start the Administration Console Using NetBeans IDE

1. Click the **Services** tab.
2. Expand **Servers**.
3. Right-click the **GlassFish Server instance** and select **View Domain Admin Console**.



NetBeans IDE uses your default web browser to open the Administration Console.

Starting and Stopping Apache Derby

GlassFish Server includes Apache Derby.

To Start Derby Using Command Line

To start Derby from the command line, open a terminal window or command prompt, change to the `as-install/bin` directory, and execute:

```
asadmin start-database
```

To Stop Derby Using Command Line

To stop Derby from the command line, open a terminal window or command prompt, change to the `as-install/bin` directory, and execute:

```
asadmin stop-database
```

For information about Apache Derby included with GlassFish Server, see the Release Notes that are located in the `as-install/javadb/` directory.

To Start Derby Using NetBeans IDE

When you start GlassFish Server using NetBeans IDE, the database server starts automatically. If you ever need to start the server manually, however, follow these steps.

1. Click the **Services** tab.
2. Expand **Databases**.
3. Right-click **Java DB** and select **Start Server**.

To Stop Derby Using NetBeans IDE

To stop the database using NetBeans IDE, right-click **Java DB** and select **Stop Server**.

Building the Examples

The tutorial examples are distributed with a configuration file for either NetBeans IDE or Maven. Either NetBeans IDE or Maven may be used to build, package, deploy, and run the examples. Directions for building the examples are provided in each chapter.

Tutorial Example Directory Structure

To facilitate iterative development and keep application source files separate from compiled files, the tutorial examples use the Maven application directory structure.

Each application module has the following structure:

- `pom.xml`: Maven build file
- `src/main/java`: Java source files for the module
- `src/main/resources`: configuration files for the module, with the exception of web applications
- `src/main/webapp`: web pages, style sheets, tag files, and images (web applications only)
- `src/main/webapp/WEB-INF`: configuration files for web applications (web applications only)

When an example has multiple application modules packaged into an EAR file, its submodule directories use the following naming conventions:

- *example-name*`-app-client`: application clients
- *example-name*`-ejb`: enterprise bean JAR files
- *example-name*`-war`: web applications
- *example-name*`-ear`: enterprise applications
- *example-name*`-common`: library JAR containing components, classes, and files used by other modules

The Maven build files (`pom.xml`) distributed with the examples contain goals to compile and assemble the application into the `target` directory and deploy the archive to GlassFish Server.

Jakarta EE Maven Archetypes in the Tutorial

Some of the chapters have instructions on how to build an example application using Maven archetypes. Archetypes are templates for generating a particular Maven project. The Tutorial includes several Maven archetypes for generating Jakarta EE projects.

Installing the Tutorial Archetypes

You must install the included Maven archetypes into your local Maven repository before you can create new projects based on the archetypes. You can install the archetypes using NetBeans IDE or Maven.

Installing the Tutorial Archetypes Using NetBeans IDE

1. From the **File** menu, choose **Open Project**.
2. In the Open Project dialog box, navigate to:

```
jakartaee-examples/tutorial
```

1. Select the `archetypes` folder.
2. Click **Open Project**.
3. In the **Projects** tab, right-click the `archetypes` project and select **Build**.

Installing the Tutorial Archetypes Using Maven

1. In a terminal window, go to:

```
jakartaee-examples/tutorial/archetypes/
```

1. Enter the following command:

```
mvn install
```

Debugging Jakarta EE Applications

This section explains how to determine what is causing an error in your application deployment or execution.

Using the Server Log

One way to debug applications is to look at the server log in `domain-dir/logs/server.log`. The log contains output from GlassFish Server and your applications. You can log messages from any Java class in your application with `System.out.println` and the Java Logging APIs (documented at https://docs.oracle.com/javase/8/docs/technotes/guides/logging/index.html) and from web components with the `ServletContext.log` method.

If you use NetBeans IDE, logging output appears in the Output window as well as the server log.

If you start GlassFish Server with the `--verbose` flag, all logging and debugging output will appear on the terminal window or command prompt and the server log. If you start GlassFish Server in the background, debugging information is available only in the log. You can view the server log with a text editor or with the Administration Console log viewer.

To Use the Administration Console Log Viewer

1. Select the **GlassFish Server** node.
2. Click **View Log Files**.
   The log viewer opens and displays the last 40 entries.
3. To display other entries, follow these steps:
   a. Click **Modify Search**.
   b. Specify any constraints on the entries you want to see.
   c. Click Search at the top of the log viewer.

Using a Debugger

GlassFish Server supports the Java Platform Debugger Architecture (JPDA). With JPDA, you can configure GlassFish Server to communicate debugging information using a socket.

To Debug an Application Using a Debugger

1. Follow these steps to enable debugging in GlassFish Server using the Administration Console:
   a. Expand the **Configurations** node, then expand the **server-config** node.
   b. Select the **JVM Settings** node. The default debug options are set to:

      ```
      -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9009
      ```

      As you can see, the default debugger socket port is 9009. You can change it to a port not in use by GlassFish Server or another service.

   c. Select the **Debug Enabled** check box.
   d. Click **Save**.
2. Stop GlassFish Server and then restart it.

[1] Technically, there are multiple containers (Web, CDI, Enterprise Beans), but the distinction about which container is providing the services isn't terribly important. Jakarta EE also supports an Application Client container for Java clients, but this is rarely used.

[2] Not to be confused with the web browser standard, Web Components, or user interface widgets in general, which are often called UI components.

[3] Enterprise Beans previously supported persistence via Entity Beans, but that has been deprecated in favor of Jakarta Persistence

[4] Application Clients are technically supported as well, although they are rarely used.

Platform Basics

Resource Creation



This section was last updated for Jakarta EE Version 9.1

A resource is a program object that provides connections to such systems as database servers and messaging systems. Jakarta EE components can access a wide variety of resources, including databases, mail sessions, Jakarta Messaging objects, and URLs. The Jakarta EE platform provides mechanisms that allow you to access all these resources in a similar manner. This chapter examines several types of resources and explains how to create them.

Resources and JNDI Naming

In a distributed application, components need to access other components and resources, such as databases. For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database. In the Jakarta EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.

A resource is a program object that provides connections to systems, such as database servers and messaging systems. A Java Database Connectivity resource is sometimes referred to as a data source. Each resource object is identified by a unique, people-friendly name, called the JNDI name. For example, the JNDI name of the preconfigured JDBC resource for Apache Derby shipped with GlassFish Server is `java:comp/DefaultDataSource`.

An administrator creates resources in a JNDI namespace. In GlassFish Server, you can use either the Administration Console or the `asadmin` command to create resources. Applications then use annotations to inject the resources. If an application uses resource injection, GlassFish Server invokes the JNDI API, and the application is not required to do so. However, it is also possible for an application to locate resources by making direct calls to the JNDI API.

A resource object and its JNDI name are bound together by the naming and directory service. To create a new resource, a new name/object binding is entered into the JNDI namespace. You inject resources by using the `@Resource` annotation in an application.

You can use a deployment descriptor to override the resource mapping that you specify in an annotation. Using a deployment descriptor allows you to change an application by repackaging it rather than by both recompiling the source files and repackaging. However, for most applications a deployment descriptor is not necessary.

DataSource Objects and Connection Pools

To store, organize, and retrieve data, most applications use a relational database. Jakarta EE components may access relational databases through the JDBC API. For information on this API, see https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

In the JDBC API, databases are accessed by using `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real-world data source that it represents. These properties

include such information as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on. In GlassFish Server, a data source is called a JDBC resource.

Applications access a data source by using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a call to the `getConnection` method returns a connection object that is a physical connection to the data source.

A `DataSource` object may be registered with a JNDI naming service. If so, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

`DataSource` objects that implement connection pooling also produce a connection to the particular data source that the `DataSource` class represents. The connection object that the `getConnection` method returns is a handle to a `PooledConnection` object rather than a physical connection. An application uses the connection object in the same way that it uses a connection. Connection pooling has no effect on application code except that a pooled connection, like all connections, should always be explicitly closed. When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections. The next time `getConnection` is called, a handle to one of these pooled connections will be returned if one is available. Because connection pooling avoids creating a new physical connection every time one is requested, applications can run significantly faster.

A JDBC connection pool is a group of reusable connections for a particular database. Because creating each new physical connection is time consuming, the server maintains a pool of available connections to increase performance. When it requests a connection, an application obtains one from the pool. When an application closes a connection, the connection is returned to the pool.

Applications that use Jakarta Persistence specify the `DataSource` object they are using in the `jta-data-source` element of the `persistence.xml` file:

```
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
```

This is typically the only reference to a JDBC object for a persistence unit. The application code does not refer to any JDBC objects.

Creating Resources Administratively

Before you deploy or run many applications, you may need to create resources for them. An application can include a `glassfish-resources.xml` file that can be used to define resources for that application and others. You can then use the `asadmin` command, specifying as the argument a file named `glassfish-resources.xml`, to create the resources administratively, as shown here:

```
asadmin add-resources glassfish-resources.xml
```

The `glassfish-resources.xml` file can be created in any project using NetBeans IDE or by hand. Some

of the Jakarta Messaging examples use this approach to resource creation. A file for creating the resources needed for the Messaging simple producer example can be found in the `jms/simple/producer/src/main/setup` directory.

You could also use the `asadmin create-jms-resource` command to create the resources for this example. When you are done using the resources, you would use the `asadmin list-jms-resources` command to display their names, and the `asadmin delete-jms-resource` command to remove them, regardless of the way you created the resources.

Injection



This section was last updated for Jakarta EE Version 9.1

This chapter provides an overview of injection in Jakarta EE and describes the two injection mechanisms provided by the platform: resource injection and dependency injection.

Jakarta EE provides injection mechanisms that enable your objects to obtain references to resources and other dependencies without having to instantiate them directly. You declare the required resources and other dependencies in your classes by decorating fields or methods with one of the annotations that mark the field as an injection point. The container then provides the required instances at runtime. Injection simplifies your code and decouples it from the implementations of its dependencies.

Resource Injection

Resource injection enables you to inject any resource available in the JNDI namespace into any container-managed object, such as a servlet, an enterprise bean, or a managed bean. For example, you can use resource injection to inject data sources, connectors, or custom resources available in the JNDI namespace.

The type you use for the reference to the injected instance is usually an interface, which decouples your code from the implementation of the resource.

For example, the following code injects a data source object that provides connections to the default Apache Derby database shipped with Eclipse GlassFish Server:

```
public class MyServlet extends HttpServlet {
    @Resource(name="java:comp/DefaultDataSource")
    private javax.sql.DataSource dsc;
    ...
}
```

In addition to field-based injection as in the preceding example, you can inject resources using method-based injection:

```
public class MyServlet extends HttpServlet {
    private javax.sql.DataSource dsc;
```

```
    ...
    @Resource(name="java:comp/DefaultDataSource")
    public void setDsc(java.sql.DataSource ds) {
        dsc = ds;
    }
}
```

To use method-based injection, the setter method must follow the JavaBeans conventions for property names: The method name must begin with `set`, have a `void` return type, and have only one parameter.

The `@Resource` annotation is in the `jakarta.annotation` package and is defined in the Jakarta Annotations spec. Resource injection resolves by name, so it is not typesafe: the type of the resource object is not known at compile time, so you can get runtime errors if the types of the object and its reference do not match.

Dependency Injection

Dependency injection enables you to turn regular Java classes into managed objects and to inject them into any other managed object. Using dependency injection, your code can declare dependencies on any managed object. The container automatically provides instances of these dependencies at the injection points at runtime, and it also manages the lifecycle of these instances for you.

Dependency injection in Jakarta EE defines scopes, which determine the lifecycle of the objects that the container instantiates and injects. For example, a managed object that is only needed to respond to a single client request (such as a currency converter) has a different scope than a managed object that is needed to process multiple client requests within a session (such as a shopping cart).

You can define managed objects (also called managed beans) that you can later inject by assigning a scope to a regular class:

```
@jakarta.enterprise.context.RequestScoped
public class CurrencyConverter { ... }
```

Use the `jakarta.inject.Inject` annotation to inject managed beans; for example:

```
public class MyServlet extends HttpServlet {
    @Inject CurrencyConverter cc;
    ...
}
```

As opposed to resource injection, dependency injection is typesafe because it resolves by type. To decouple your code from the implementation of the managed bean, you can reference the injected instances using an interface type and have your managed bean implement that interface.

For more information about dependency injection, see Introduction to Jakarta Contexts and Dependency Injection and the Jakarta Contexts and Dependency Injection spec.

The Main Differences between Resource Injection and Dependency Injection

Differences between Resource Injection and Dependency Injection lists the main differences between resource injection and dependency injection.

Differences between Resource Injection and Dependency Injection

Injection Mechanism	Can Inject JNDI Resources Directly	Can Inject Regular Classes Directly	Resolves By	Typesafe
Resource Injection	Yes	No	Resource name	No
Dependency Injection	No	Yes	Type	Yes

Packaging



This section was last updated for Jakarta EE Version 9.1

This chapter describes packaging. A Jakarta EE application is packaged into one or more standard units for deployment to any Jakarta EE platform-compliant system. Each unit contains a functional component or components, such as an enterprise bean, web page, servlet, or applet, and an optional deployment descriptor that describes its content.

Packaging Applications

A Jakarta EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard JAR (`.jar`) file with a `.war` or `.ear` extension. Using JAR, WAR, and EAR files and modules makes it possible to assemble a number of different Jakarta EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various Jakarta EE modules into Jakarta EE JAR, WAR, or EAR files.

An EAR file (see Figure 4, “EAR File Structure”) contains Jakarta EE modules and, optionally, deployment descriptors. A deployment descriptor, an XML document with an `.xml` extension, describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Jakarta EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

Deployment information is most commonly specified in the source code by annotations. Deployment descriptors, if present, override what is specified in the source code.

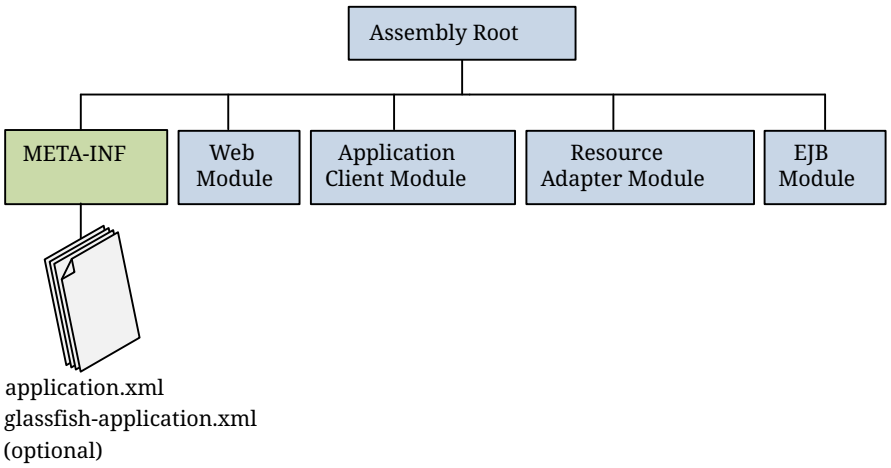


Figure 4. EAR File Structure

The two types of deployment descriptors are Jakarta EE and runtime. A Jakarta EE deployment descriptor is defined by a Jakarta EE specification and can be used to configure deployment settings on any Jakarta EE-compliant implementation. A runtime deployment descriptor is used to configure Jakarta EE implementation-specific parameters. For example, the GlassFish Server runtime deployment descriptor contains such information as the context root of a web application as well as GlassFish Server implementation-specific parameters, such as caching directives. The GlassFish Server runtime deployment descriptors are named `glassfish-moduleType.xml` and are located in the same `META-INF` directory as the Jakarta EE deployment descriptor.

A Jakarta EE module consists of one or more Jakarta EE components for the same container type and, optionally, one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A Jakarta EE module can be deployed as a stand-alone module.

Jakarta EE modules are of the following types:

- Enterprise bean modules, which contain class files for enterprise beans and, optionally, an enterprise bean deployment descriptor. Enterprise bean modules are packaged as JAR files with a `.jar` extension.
- Web modules, which contain servlet class files, web files, supporting class files, GIF and HTML files, and, optionally, a web application deployment descriptor. Web modules are packaged as JAR files with a `.war` (web archive) extension.
- Application client modules, which contain class files and, optionally, an application client deployment descriptor. Application client modules are packaged as JAR files with a `.jar` extension.
- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and, optionally, a resource adapter deployment descriptor. Together, these implement the Connector architecture (see Jakarta Connectors) for a particular EIS. Resource adapter modules are packaged as JAR files with an `.rar` (resource adapter archive) extension.

Packaging Enterprise Beans

This section explains how enterprise beans can be packaged in enterprise bean JAR or WAR modules. It includes the following sections:

- Packaging Enterprise Beans in enterprise bean JAR Modules
- Packaging Enterprise Beans in WAR Modules

Packaging Enterprise Beans in enterprise bean JAR Modules

An enterprise bean JAR file is portable and can be used for various applications.

To assemble a Jakarta EE application, package one or more modules, such as enterprise bean JAR files, into an EAR file, the archive file that holds the application. When deploying the EAR file that contains the enterprise bean's JAR file, you also deploy the enterprise bean to GlassFish Server. You can also deploy an enterprise bean JAR that is not contained in an EAR file. Figure 5, "Structure of an Enterprise Bean JAR" shows the contents of an enterprise bean JAR file.

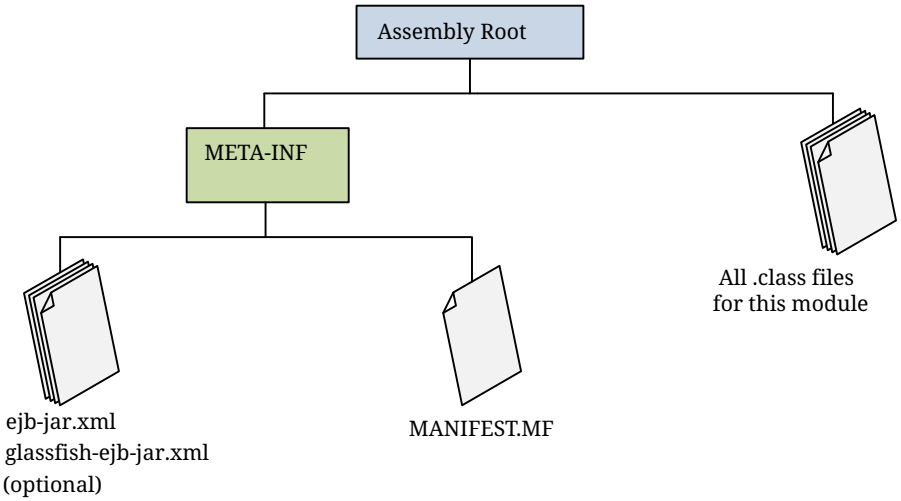


Figure 5. Structure of an Enterprise Bean JAR

Packaging Enterprise Beans in WAR Modules

Enterprise beans often provide the business logic of a web application. In these cases, packaging the enterprise bean within the web application's WAR module simplifies deployment and application organization. Enterprise beans may be packaged within a WAR module as Java programming language class files or within a JAR file that is bundled within the WAR module.

To include enterprise bean class files in a WAR module, the class files should be in the `WEB-INF/classes` directory.

To include a JAR file that contains enterprise beans in a WAR module, add the JAR to the `WEB-INF/lib` directory of the WAR module.

WAR modules that contain enterprise beans do not require an `ejb-jar.xml` deployment descriptor. If the application uses `ejb-jar.xml`, it must be located in the WAR module's `WEB-INF` directory.

JAR files that contain enterprise bean classes packaged within a WAR module are not considered enterprise bean JAR files, even if the bundled JAR file conforms to the format of an enterprise bean

JAR file. The enterprise beans contained within the JAR file are semantically equivalent to enterprise beans located in the WAR module's `WEB-INF/classes` directory, and the environment namespace of all the enterprise beans are scoped to the WAR module.

For example, suppose that a web application consists of a shopping cart enterprise bean, a credit card–processing enterprise bean, and a Java servlet front end. The shopping cart bean exposes a local, no-interface view and is defined as follows:

```
package com.example.cart;

@Stateless
public class CartBean { ... }
```

The credit card–processing bean is packaged within its own JAR file, `cc.jar`, exposes a local, no-interface view, and is defined as follows:

```
package com.example.cc;

@Stateless
public class CreditCardBean { ... }
```

The servlet, `com.example.web.StoreServlet`, handles the web front end and uses both `CartBean` and `CreditCardBean`. The WAR module layout for this application is as follows:

```
WEB-INF/classes/com/example/cart/CartBean.class
WEB-INF/classes/com/example/web/StoreServlet
WEB-INF/lib/cc.jar
WEB-INF/ejb-jar.xml
WEB-INF/web.xml
```

Packaging Web Archives

In the Jakarta EE architecture, a web module is the smallest deployable and usable unit of web resources. A web module contains web components and static web content files, such as images, which are called web resources. A Jakarta EE web module corresponds to a web application as defined in the Jakarta Servlet specification.

In addition to web components and web resources, a web module can contain other files:

- Server-side utility classes, such as shopping carts
- Client-side classes, such as utility classes

A web module has a specific structure. The top-level directory of a web module is the document root of the application. The document root is where XHTML pages, client-side classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named `WEB-INF`, which can contain the following files and directories:

- `classes`, a directory that contains server-side classes: servlets, enterprise bean class files, utility classes, and JavaBeans components
- `lib`, a directory that contains JAR files that contain enterprise beans, and JAR archives of libraries called by server-side classes
- Deployment descriptors, such as `web.xml` (the web application deployment descriptor) and `ejb-jar.xml` (an enterprise bean deployment descriptor)

A web module needs a `web.xml` file if it uses Jakarta Faces technology, if it must specify certain kinds of security information, or if you want to override information specified by web component annotations.

You can also create application-specific subdirectories (that is, package directories) in either the document root or the `WEB-INF/classes/` directory.

A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a Web Archive (WAR) file. Because the contents and use of WAR files differ from those of JAR files, WAR file names use a `.war` extension. The web module just described is portable; you can deploy it into any web container that conforms to the Jakarta Servlet specification.

You can provide a runtime deployment descriptor (DD) when you deploy a WAR on GlassFish Server, but it is not required under most circumstances. The runtime DD is an XML file that may contain such information as the context root of the web application, the mapping of the portable names of an application's resources to GlassFish Server resources, and the mapping of an application's security roles to users, groups, and principals defined in GlassFish Server. The GlassFish Server web application runtime DD, if used, is named `glassfish-web.xml` and is located in the `WEB-INF` directory. The structure of a web module that can be deployed on GlassFish Server is shown in Figure 6, "Web Module Structure".

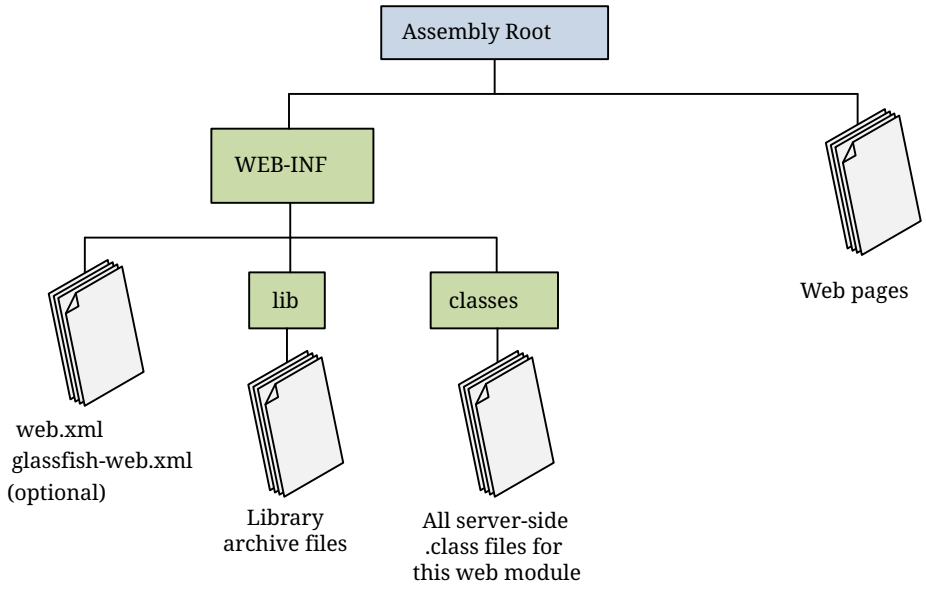


Figure 6. Web Module Structure

Packaging Resource Adapter Archives

A Resource Adapter Archive (RAR) file stores XML files, Java classes, and other objects for Jakarta EE Connector applications. A resource adapter can be deployed on any Jakarta EE server, much like a Jakarta EE application. A RAR file can be contained in an Enterprise Archive (EAR) file, or it can exist as a separate file.

The RAR file contains

- A JAR file with the implementation classes of the resource adapter
- An optional `META-INF/` directory that can store an `ra.xml` file and/or an application server–specific deployment descriptor used for configuration purposes

A RAR file can be deployed on the application server as a standalone component or as part of a larger application. In both cases, the adapter is available to all applications using a lookup procedure.

Jakarta EE Core Profile

Jakarta CDI Lite

Introduction to Jakarta Contexts and Dependency Injection



This section was last updated for Jakarta EE Version 9.1

This chapter describes Jakarta Contexts and Dependency Injection (CDI) which is one of several Jakarta EE features that help to knit together the web tier and the transactional tier of the Jakarta EE platform.

Getting Started

Contexts and Dependency Injection (CDI) enables your objects to have their dependencies provided to them automatically, instead of creating them or receiving them as parameters. CDI also manages the lifecycle of those dependencies for you.

For example, consider the following servlet:

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
    private Message message;

    @Override
    public void init() {
        message = new MessageB();
    }

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                  throws IOException {
        response.getWriter().write(message.get());
    }
}
```

This servlet needs an instance of an object that implements the `Message` interface:

```
public interface Message {
    public String get();
}
```

The servlet creates itself an instance of the following object:

```
public class MessageB implements Message {
    public MessageB() { }
```

```
    @Override
    public String get() {
        return "message B";
    }
}
```

Using CDI, this servlet can declare its dependency on a `Message` instance and have it injected automatically by the CDI runtime. The new servlet code is the following:

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
    @Inject private Message message;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                  throws IOException {
        response.getWriter().write(message.get());
    }
}
```

The CDI runtime looks for classes that implement the `Message` interface, finds the `MessageB` class, creates a new instance of it, and injects it into the servlet at runtime. To manage the lifecycle of the new instance, the CDI runtime needs to know what the scope of the instance should be. In this example, the servlet only needs the instance to process an HTTP request; the instance can then be garbage collected. This is specified using the `jakarta.enterprise.context.RequestScoped` annotation:

```
@RequestScoped
public class MessageB implements Message { ... }
```

For more information on scopes, see Using Scopes.

The `MessageB` class is a CDI bean. CDI beans are classes that CDI can instantiate, manage, and inject automatically to satisfy the dependencies of other objects. Almost any Java class can be managed and injected by CDI. For more information on beans, see About Beans. A JAR or WAR file that contains a CDI bean is a bean archive. For more information on packaging bean archives, see Configuring a CDI Application in this chapter and Packaging CDI Applications in [cdi:cdi-adv::cdi-adv:::_jakarta_contexts_and_dependency_injection_advanced_topics].

In this example, `MessageB` is the only class that implements the `Message` interface. If an application has more than one implementation of an interface, CDI provides mechanisms that you can use to select which implementation to inject. For more information, see Using Qualifiers in this chapter and Using Alternatives in CDI Applications in [cdi:cdi-adv::cdi-adv:::_jakarta_contexts_and_dependency_injection_advanced_topics].

Overview of CDI

CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with Jakarta Faces technology in web applications. Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but typesafe way.

CDI 3.0 is specified in a Jakarta EE specification. Related specifications that CDI uses include the following:

- Jakarta Dependency Injection
- The Managed Beans specification, an offshoot of the Jakarta EE platform specification

The most fundamental services provided by CDI are as follows.

- Contexts: This service enables you to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts.
- Dependency injection: This service enables you to inject components into an application in a typesafe way and to choose at deployment time which implementation of a particular interface to inject.

In addition, CDI provides the following services:

- Integration with the Expression Language (EL), which allows any component to be used directly within a Jakarta Faces page or a Jakarta Server Pages page
- The ability to decorate injected components
- The ability to associate interceptors with components using typesafe interceptor bindings
- An event-notification model
- A web conversation scope in addition to the three standard scopes (request, session, and application) defined by the Jakarta Servlet specification
- A complete Service Provider Interface (SPI) that allows third-party frameworks to integrate cleanly in the Jakarta EE environment

A major theme of CDI is loose coupling. CDI does the following:

- Decouples the server and the client by means of well-defined types and qualifiers, so that the server implementation may vary
- Decouples the lifecycles of collaborating components by
    - Making components contextual, with automatic lifecycle management
    - Allowing stateful components to interact like services, purely by message passing
- Completely decouples message producers from consumers, by means of events
- Decouples orthogonal concerns by means of Jakarta EE interceptors

Along with loose coupling, CDI provides strong typing by

- Eliminating lookup using string-based names for wiring and correlations so that the compiler

will detect typing errors

- Allowing the use of declarative Java annotations to specify everything, largely eliminating the need for XML deployment descriptors, and making it easy to provide tools that introspect the code and understand the dependency structure at development time

CDI Lite vs CDI Full

As of CDI version 4.0 in Jakarta EE version 10, the Core CDI functionality has been split into two parts: CDI Lite and CDI Full.

CDI Lite provides a subset of the CDI Full functionality with an emphasis on build-time implementations. By leaving out the additional components from CDI Full such as those dealing with runtime reflection, CDI Lite is able to execute in lighter and more restricted environments.

Jakarta EE-compliant application servers will still implement the CDI Full functionality so this change will benefit those developers working in alternate (e.g. cloud-based) environments without affecting those working in a standard Jakarta EE environment.

Functionality available only in CDI Full includes the following:

- Binding interceptors using @Interceptors
- Explicit bean archives
- Declaring interceptors on classes using @AroundInvoke
- Decorator classes
- Portable extensions
- Serialization via passivation/activation
- Session scope, conversation scope
- Specialization using @Alternative and @Specializes

Please see the Further Information about CDI section of this chapter for links to the latest specification.



The remainder of this chapter deals with the CDI Lite profile. The tutorial chapter on CDI Full can be found here.

About Beans

CDI redefines the concept of a bean beyond its use in other Java technologies, such as the JavaBeans and Jakarta Enterprise Beans technologies. In CDI, a bean is a source of contextual objects that define application state or logic. A Jakarta EE component is a bean if the lifecycle of its instances may be managed by the container according to the lifecycle context model defined in the CDI specification.

More specifically, a bean has the following attributes:

- A (nonempty) set of bean types

- A (nonempty) set of qualifiers (see Using Qualifiers)
- A scope (see Using Scopes)
- Optionally, a bean EL name (see Giving Beans EL Names)
- A set of interceptor bindings
- A bean implementation

A bean type defines a client-visible type of the bean. Almost any Java type may be a bean type of a bean.

- A bean type may be an interface, a concrete class, or an abstract class and may be declared final or have final methods.
- A bean type may be a parameterized type with type parameters and type variables.
- A bean type may be an array type. Two array types are considered identical only if the element type is identical.
- A bean type may be a primitive type. Primitive types are considered to be identical to their corresponding wrapper types in `java.lang`.
- A bean type may be a raw type.

About CDI Managed Beans

A managed bean is implemented by a Java class, which is called its bean class. A top-level Java class is a managed bean if it is defined to be a managed bean by any other Jakarta EE technology specification, such as the Jakarta Faces technology specification, or if it meets all the following conditions.

- It is not a nonstatic inner class.
- It is a concrete class or is annotated `@Decorator`.
- It is not annotated with an enterprise bean component-defining annotation or declared as an enterprise bean class in `ejb-jar.xml`.
- It has an appropriate constructor. That is, one of the following is the case.
  - The class has a constructor with no parameters.
  - The class declares a constructor annotated `@Inject`.

No special declaration, such as an annotation, is required to define a managed bean.

Beans as Injectable Objects

The concept of injection has been part of Java technology for some time. Since the Java EE 5 platform was introduced, annotations have made it possible to inject resources and some other kinds of objects into container-managed objects. CDI makes it possible to inject more kinds of objects and to inject them into objects that are not container-managed.

The following kinds of objects can be injected:

- Almost any Java class

- Session beans
- Jakarta EE resources: data sources, Messaging topics, queues, connection factories, and the like
- Persistence contexts (Jakarta Persistence `EntityManager` objects)
- Producer fields
- Objects returned by producer methods
- Web service references
- Remote enterprise bean references

For example, suppose that you create a simple Java class with a method that returns a string:

```
package greetings;

public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

This class becomes a bean that you can then inject into another class. This bean is not exposed to the EL in this form. Giving Beans EL Names explains how you can make a bean accessible to the EL.

Using Qualifiers

You can use qualifiers to provide various implementations of a particular bean type. A qualifier is an annotation that you apply to a bean. A qualifier type is a Java annotation defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})` and `@Retention(RUNTIME)`.

For example, you could declare an `@Informal` qualifier type and apply it to another class that extends the `Greeting` class. To declare this qualifier type, use the following code:

```
package greetings;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;

import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
```

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

You can then define a bean class that extends the `Greeting` class and uses this qualifier:

```
package greetings;

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "Hi, " + name + "!";
    }
}
```

Both implementations of the bean can now be used in the application.

If you define a bean with no qualifier, then the bean automatically has the qualifier `@Default`. The unannotated `Greeting` class could be declared as follows:

```
package greetings;

import jakarta.enterprise.inject.Default;

@Default
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

Injecting Beans

To use the beans you create, you inject them into yet another bean that can then be used by an application, such as a Jakarta Faces application. For example, you might create a bean called `Printer` into which you would inject one of the `Greeting` beans:

```
import jakarta.inject.Inject;

public class Printer {

    @Inject Greeting greeting;
    ...
}
```

This code injects the `@Default Greeting` implementation into the bean. The following code injects the `@Informal` implementation:

```
import jakarta.inject.Inject;

public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

More is needed for the complete picture of this bean. Its use of scope needs to be understood. In addition, for a Jakarta Faces application, the bean needs to be accessible through the EL.

Now that you can identify the target of the injection, it is important to understand what can be injected and in what context. Faces 2.3 and above provides producers that enable most important Faces artifacts to be injected. For detailed information, see the package javadoc for `jakarta.faces.annotation`.

Using Scopes

For a web application to use a bean that injects another bean class, the bean needs to be able to hold state over the duration of the user's interaction with the application. The way to define this state is to give the bean a scope. You can give an object any of the scopes described in Scopes, depending on how you are using it.

Scopes

Scope	Annotation	Duration
Request	`@RequestScoped`	A user's interaction with a web application in a single HTTP request.
Session	`@SessionScoped`	A user's interaction with a web application across multiple HTTP requests.
Application	`@ApplicationScoped`	Shared state across all users' interactions with a web application.
Dependent	`@Dependent`	The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).
Conversation	`@ConversationScoped`	A user's interaction with a servlet, including Jakarta Faces applications. The conversation scope exists within developer-controlled boundaries that extend it across multiple requests for long-running conversations. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

The first three scopes are defined by both Jakarta Context and Dependency Injection and the

Jakarta Faces specification. The last two are defined by Jakarta Context and Dependency Injection.

All predefined scopes except `@Dependent` are contextual scopes. CDI places beans of contextual scope in the context whose lifecycle is defined by the Jakarta EE specifications. For example, a session context and its beans exist during the lifetime of an HTTP session. Injected references to the beans are contextually aware. The references always apply to the bean that is associated with the context for the thread that is making the reference. The CDI container ensures that the objects are created and injected at the correct time as determined by the scope that is specified for these objects.

You can also define and implement custom scopes, but that is an advanced topic. Custom scopes are likely to be used by those who implement and extend the CDI specification.

A scope gives an object a well-defined lifecycle context. A scoped object can be automatically created when it is needed and automatically destroyed when the context in which it was created ends. Moreover, its state is automatically shared by any clients that execute in the same context.

Jakarta EE components, such as servlets and enterprise beans, and JavaBeans components do not by definition have a well-defined scope. These components are one of the following:

- Singletons, such as enterprise singleton beans, whose state is shared among all clients
- Stateless objects, such as servlets and stateless session beans, which do not contain client-visible state
- Objects that must be explicitly created and destroyed by their client, such as JavaBeans components and stateful session beans, whose state is shared by explicit reference passing between clients

However, if you create a Jakarta EE component that is a managed bean, then it becomes a scoped object, which exists in a well-defined lifecycle context.

The web application for the `Printer` bean will use a simple request and response mechanism, so the managed bean can be annotated as follows:

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;

@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

Beans that use session, application, or conversation scope must be serializable, but beans that use request scope do not have to be serializable.

Giving Beans EL Names

To make a bean accessible through the EL, use the `@Named` built-in qualifier:

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

The `@Named` qualifier allows you to access the bean by using the bean name, with the first letter in lowercase. For example, a Facelets page would refer to the bean as `printer`.

You can specify an argument to the `@Named` qualifier to use a nondefault name:

```
@Named("MyPrinter")
```

With this annotation, the Facelets page would refer to the bean as `MyPrinter`.

Adding Setter and Getter Methods

To make the state of the managed bean accessible, add setter and getter methods for that state. The `createSalutation` method calls the bean's `greet` method, and the `getSalutation` method retrieves the result.

Once the setter and getter methods have been added, the bean is complete. The final code looks like this:

```
package greetings;

import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;

    private String name;
    private String salutation;

    public void createSalutation() {
        this.salutation = greeting.greet(name);
    }
```

```
    public String getSalutation() {
        return salutation;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Using a Managed Bean in a Facelets Page

To use the managed bean in a Facelets page, create a form that uses user interface elements to call its methods and to display their results. The following example provides a button that asks the user to type a name, retrieves the salutation, and then displays the text in a paragraph below the button:

```
<h:form id="greetme">
   <p><h:outputLabel value="Enter your name: " for="name"/>
      <h:inputText id="name" value="#{printer.name}"/></p>
   <p><h:commandButton value="Say Hello"
                       action="#{printer.createSalutation}"/></p>
   <p><h:outputText value="#{printer.salutation}"/></p>
</h:form>
```

Injecting Objects by Using Producer Methods

Producer methods provide a way to inject objects that are not beans, objects whose values may vary at runtime, and objects that require custom initialization. For example, if you want to initialize a numeric value defined by a qualifier named `@MaxNumber`, then you can define the value in a managed bean and then define a producer method, `getMaxNumber`, for it:

```
private int maxNumber = 100;
...
@Produces @MaxNumber int getMaxNumber() {
    return maxNumber;
}
```

When you inject the object in another managed bean, the container automatically invokes the producer method, initializing the value to 100:

```
@Inject @MaxNumber private int maxNumber;
```

If the value can vary at runtime, then the process is slightly different. For example, the following code defines a producer method that generates a random number defined by a qualifier called `@Random`:

```
private java.util.Random random =
    new java.util.Random( System.currentTimeMillis() );

java.util.Random getRandom() {
        return random;
}

@Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
}
```

When you inject this object in another managed bean, you declare a contextual instance of the object:

```
@Inject @Random Instance<Integer> randomInt;
```

You then call the `get` method of the `Instance`:

```
this.number = randomInt.get();
```

Configuring a CDI Application

When your beans are annotated with a scope type, the server recognizes the application as a bean archive and no additional configuration is required. The possible scope types for CDI beans are listed in Using Scopes.

CDI uses an optional deployment descriptor named `beans.xml`. Like other Jakarta EE deployment descriptors, the configuration settings in `beans.xml` are used in addition to annotation settings in CDI classes. The settings in `beans.xml` override the annotation settings if there is a conflict. An archive must contain the `beans.xml` deployment descriptor only in certain limited situations, described in [cdi:cdi-adv::cdi-adv:::_jakarta_contexts_and_dependency_injection_advanced_topics].

For a web application, the `beans.xml` deployment descriptor, if present, must be in the `WEB-INF` directory. For EJB modules or JAR files, the `beans.xml` deployment descriptor, if present, must be in the `META-INF` directory.

Using the @PostConstruct and @PreDestroy Annotations with CDI Managed Bean Classes

CDI managed bean classes and their superclasses support the annotations for initializing and for preparing for the destruction of a bean. These annotations are defined in Jakarta Annotations (https://jakarta.ee/specifications/annotations/2.0/).

To Initialize a Managed Bean Using the @PostConstruct Annotation

Initializing a managed bean specifies the lifecycle callback method that the CDI framework should call after dependency injection but before the class is put into service.

1. In the managed bean class or any of its superclasses, define a method that performs the initialization that you require.
2. Annotate the declaration of the method with the `jakarta.annotation.PostConstruct` annotation.

When the managed bean is injected into a component, CDI calls the method after all injection has occurred and after all initializers have been called.



As mandated in Jakarta Annotations, if the annotated method is declared in a superclass, the method is called unless a subclass of the declaring class overrides the method.

The `UserNumberBean` managed bean in The guessnumber-cdi CDI Example uses `@PostConstruct` to annotate a method that resets all bean fields:

```
@PostConstruct
public void reset () {
    this.minimum = 0;
    this.userNumber = 0;
    this.remainingGuesses = 0;
    this.maximum = maxNumber;
    this.number = randomInt.get();
}
```

To Prepare for the Destruction of a Managed Bean Using the @PreDestroy Annotation

Preparing for the destruction of a managed bean specifies the lifecycle call back method that signals that an application component is about to be destroyed by the container.

1. In the managed bean class or any of its superclasses, prepare for the destruction of the managed bean.

   In this method, perform any cleanup that is required before the bean is destroyed, such as releasing a resource that the bean has been holding.

2. Annotate the declaration of the method with the `jakarta.annotation.PreDestroy` annotation.

CDI calls this method before starting to destroy the bean.

Further Information about CDI

For more information about CDI, see

- Jakarta Contexts and Dependency Injection specification: https://jakarta.ee/specifications/cdi/4.0/

- Weld - CDI Implementation: https://docs.jboss.org/weld/reference/latest/en-US/html/
- Jakarta Dependency Injection specification: https://jakarta.ee/specifications/dependency-injection/2.0/

Running the Basic Contexts and Dependency Injection Examples



This section was last updated for Jakarta EE Version 9.1

This chapter describes in detail how to build and run simple examples that use CDI.

Building and Running the CDI Samples

The examples are in the `jakartaee-examples/tutorial/cdi/` directory.

To build and run the examples, you will do the following:

1. Use NetBeans IDE or the Maven tool to compile and package the example.
2. Use NetBeans IDE or the Maven tool to deploy the example.
3. Run the example in a web browser.

See [intro:usingexamples::usingexamples:::_using_the_tutorial_examples], for basic information on installing, building, and running the examples.

The simplegreeting CDI Example

The `simplegreeting` example illustrates some of the most basic features of CDI: scopes, qualifiers, bean injection, and accessing a managed bean in a Jakarta Faces application. When you run the example, you click a button that presents either a formal or an informal greeting, depending on how you edited one of the classes. The example includes four source files, a Facelets page and template, and configuration files.

The simplegreeting Source Files

The four source files for the `simplegreeting` example are:

- The default `Greeting` class, shown in Beans as Injectable Objects
- The `@Informal` qualifier interface definition and the `InformalGreeting` class that implements the interface, both shown in Using Qualifiers
- The `Printer` managed bean class, which injects one of the two interfaces, shown in full in Adding Setter and Getter Methods

The source files are located in the `jakartaee-examples/tutorial/cdi/simplegreeting/src/main/java/jakarta/tutorial/simplegreeting` directory.

The Facelets Template and Page

To use the managed bean in a simple Facelets application:

1. Use a very simple template file and `index.xhtml` page.

   The template page, `/WEB-INF/templates/template.xhtml`, looks like this:

```
<!DOCTYPE html>
<html lang="en"
      xmlns:h="jakarta.faces.html"
      xmlns:ui="jakarta.faces.facelets">
    <h:head>
        <title><ui:insert name="title">Default Title</ui:insert></title>
        <h:outputStylesheet name="css/default.css" />
    </h:head>
    <h:body>
        <main>
            <header>
                <h1>
                    <ui:insert name="head">
                        <ui:insert name="title">Default Header</ui:insert>
                    </ui:insert>
                </h1>
            </header>

            <article>
                <ui:insert name="content" />
            </article>
        </main>
    </h:body>
</html>
```

2. To create the Facelets page, redefine the title and head, then add a small form to the content:

```
<ui:composition template="/WEB-INF/templates/template.xhtml"
                xmlns:ui="jakarta.faces.facelets"
                xmlns:h="jakarta.faces.html">
    <ui:define name="title">Simple Greeting</ui:define>
    <ui:define name="content">
        <h:form id="simpleGreetingForm">
            <div class="input">
                <h:outputLabel for="name" value="Enter your name" />
                <h:inputText id="name" value="#{printer.name}"/>
            </div>
            <div class="actions">
                <h:commandButton id="createSalutation"
                                 value="Say Hello"
                                 action="#{printer.createSalutation}">
                    <f:ajax execute="@form" render="salutation" />
                </h:commandButton>
            </div>
            <div class="output">
```

```
                <p>
                    <h:outputText id="salutation"
                                  value="#{printer.salutation}" />
                </p>
            </div>
        </h:form>
    </ui:define>
</ui:composition>
```

The form asks the user to enter a name. The button is labeled **Say Hello**, and the action defined for it is to call the `createSalutation` method of the `Printer` managed bean. This method in turn calls the `greet` method of the defined `Greeting` class.

The output text for the form is the value of the greeting returned by the setter method. Depending on whether the default or the `@Informal` version of the greeting is injected, this is one of the following, where `name` is the name entered by the user:

```
Hello, name.

Hi, name!
```

The Facelets page and template are located in the `jakartaee-examples/tutorial/cdi/simplegreeting/src/main/webapp/` directory.

The simple CSS file that is used by the Facelets page is in the following location:

```
jakartaee-
examples/tutorial/cdi/simplegreeting/src/main/webapp/resources/css/default.css
```

Running the simplegreeting Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `simplegreeting` application.

To Build, Package, and Run the simplegreeting Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

```
jakartaee-examples/tutorial/cdi
```

4. Select the `simplegreeting` folder.
5. Click **Open Project**.
6. To modify the `Printer.java` file, perform these steps:

a. Expand the **Source Packages** node.

b. Expand the `greetings` node.

c. Double-click the `Printer.java` file.

d. In the editor, comment out the `@Informal` annotation:

```
@Inject
//@Informal
Greeting greeting;
```

e. Save the file.

7. In the **Projects** tab, right-click the `simplegreeting` project and select **Build**.

   This command builds and packages the application into a WAR file, `simplegreeting.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Build, Package, and Deploy the simplegreeting Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/cdi/simplegreeting/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `simplegreeting.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the simplegreeting Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/simplegreeting
   ```

   The **Simple Greeting** page opens.

2. Enter a name in the field.

   For example, suppose that you enter `Duke`.

3. Click **Say Hello**.

   If you did not modify the `Printer.java` file, then the following text string appears below the button:

```
Hi, Duke!
```

If you commented out the `@Informal` annotation in the `Printer.java` file, then the following text string appears below the button:

```
Hello, Duke.
```

The guessnumber-cdi CDI Example

The `guessnumber-cdi` example, somewhat more complex than the `simplegreeting` example, illustrates the use of producer methods and of session and application scope. The example is a game in which you try to guess a number in fewer than ten attempts. It is similar to the `guessnumber-faces` example described in [web:faces-facelets::faces-facelets:::_introduction_to_facelets], except that you can keep guessing until you get the right answer or until you use up your ten attempts.

The example includes four source files, a Facelets page and template, and configuration files. The configuration files and the template are the same as those used for the `simplegreeting` example.

The guessnumber-cdi Source Files

The four source files for the `guessnumber-cdi` example are:

- The `@MaxNumber` qualifier interface
- The `@Random` qualifier interface
- The `Generator` managed bean, which defines producer methods
- The `UserNumberBean` managed bean

The source files are located in the `jakartaee-examples/tutorial/cdi/guessnumber-cdi/src/main/java/jakarta/tutorial/guessnumber` directory.

The @MaxNumber and @Random Qualifier Interfaces

The `@MaxNumber` qualifier interface is defined as follows:

```
package guessnumber;

import java.lang.annotation.Documented;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;
import jakarta.inject.Qualifier;

@Target({TYPE, METHOD, PARAMETER, FIELD})
```

```
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface MaxNumber {
}
```

The `@Random` qualifier interface is defined as follows:

```
package guessnumber;

import java.lang.annotation.Documented;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;
import jakarta.inject.Qualifier;

@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface Random {
}
```

The Generator Managed Bean

The `Generator` managed bean contains the two producer methods for the application. The bean has the `@ApplicationScoped` annotation to specify that its context extends for the duration of the user's interaction with the application:

```
package guessnumber;

import java.io.Serializable;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.inject.Produces;

@ApplicationScoped
public class Generator implements Serializable {

    private static final long serialVersionUID = 1L;

    private final java.util.Random random =
        new java.util.Random( System.currentTimeMillis() );

    private final int maxNumber = 100;
```

```
    java.util.Random getRandom() {
        return random;
    }

    @Produces @Random int next() {
        return getRandom().nextInt(maxNumber + 1);
    }

    @Produces @MaxNumber int getMaxNumber() {
        return maxNumber;
    }

}
```

The UserNumberBean Managed Bean

The `UserNumberBean` managed bean, the managed bean for the Jakarta Faces application, provides the basic logic for the game. This bean does the following:

- Implements setter and getter methods for the bean fields
- Injects the two qualifier objects
- Provides a `reset` method that allows you to begin a new game after you complete one
- Provides a `check` method that determines whether the user has guessed the number
- Provides a `validateNumberRange` method that determines whether the user's input is correct

The bean is defined as follows:

```
package guessnumber;

import java.io.Serializable;
import jakarta.annotation.PostConstruct;
import jakarta.faces.view.ViewScoped;
import jakarta.enterprise.inject.Instance;
import jakarta.faces.application.FacesMessage;
import jakarta.faces.component.UIComponent;
import jakarta.faces.component.UIInput;
import jakarta.faces.context.FacesContext;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@ViewScoped
public class UserNumberBean implements Serializable {

    private static final long serialVersionUID = 1L;

    private int number;
    private Integer userNumber;
```

```
    private int minimum;
    private int remainingGuesses;

    @MaxNumber
    @Inject
    private int maxNumber;

    private int maximum;

    @Random
    @Inject
    Instance<Integer> randomInt;

    @PostConstruct
    public void init() {
        minimum = 0;
        userNumber = 0;
        remainingGuesses = 10;
        maximum = maxNumber;
        number = randomInt.get();
    }

    public void check() {
        if (userNumber > number) {
            maximum = userNumber - 1;
        }
        if (userNumber < number) {
            minimum = userNumber + 1;
        }
        if (userNumber == number) {
            FacesContext.getCurrentInstance().addMessage(null,
                new FacesMessage("Correct!"));
        }
        remainingGuesses--;
    }

    public void reset() {
        init();
    }

    public void validateNumberRange(FacesContext context,
                                    UIComponent toValidate,
                                    Object value) {
        int input = (Integer) value;

        if (input < minimum || input > maximum) {
            throw new ValidatorException(new FacesMessage("Invalid guess"));
        }
    }

    public void setUserNumber(Integer userNumber) {
```

```
        this.userNumber = userNumber;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public int getMaximum() {
        return maximum;
    }

    public int getMinimum() {
        return minimum;
    }

    public int getRemainingGuesses() {
        return remainingGuesses;
    }

}
```

The Facelets Page

This example uses the same template that the `simplegreeting` example uses. The `index.xhtml` file, however, is more complex.

```
<ui:composition template="/WEB-INF/templates/template.xhtml"
                xmlns:ui="jakarta.faces.facelets"
                xmlns:h="jakarta.faces.html">
    <ui:define name="title">Guess My Number</ui:define>
    <ui:define name="content">
        <h:form id="guessMyNumberForm">
            <p>
                I'm thinking of a number from #{userNumberBean.minimum}
                to #{userNumberBean.maximum}. You have
                <h:outputText id="remainingGuesses"
                              value="#{userNumberBean.remainingGuesses}" />
                guesses.
            </p>
            <div class="input">
                <h:outputLabel for="userNumber" value="Number" />
                <h:inputText id="userNumber"
                             value="#{userNumberBean.userNumber}"
                             required="true" size="3"
                             disabled="#{
                                userNumberBean.remainingGuesses le 0
                                    or
                                userNumberBean.number eq userNumberBean.userNumber
                             }"
                             validator="#{userNumberBean.validateNumberRange}" />
```

```
            </div>
            <div class="actions">
                <h:commandButton id="check" value="Guess"
                                 action="#{userNumberBean.check}"
                                 disabled="#{
                                    userNumberBean.remainingGuesses le 0
                                        or
                                    userNumberBean.number eq userNumberBean.userNumber
                                 }">
                    <f:ajax execute="@form"
                            render="@this remainingGuesses userNumber output messages"
/>
                </h:commandButton>
                <h:commandButton id="reset" value="Reset"
                                 action="#{userNumberBean.reset}">
                    <f:ajax execute="@this"
                            render="@this remainingGuesses userNumber output messages"
/>
                </h:commandButton>
            </div>
            <h:panelGroup id="output" layout="block">
                <h:outputText value="Higher!" rendered="#{
                    userNumberBean.userNumber ne 0
                        and
                    userNumberBean.number gt userNumberBean.userNumber
                }" />
                <h:outputText value="Lower!" rendered="#{
                    userNumberBean.userNumber ne 0
                        and
                    userNumberBean.number lt userNumberBean.userNumber
                }" />
            </h:panelGroup>
            <h:messages id="messages" />
        </h:form>
    </ui:define>
</ui:composition>
```

The Facelets page presents the user with the minimum and maximum values and the number of guesses remaining. The user's interaction with the game takes place within the `panelGrid` table, which contains an input field, **Guess** and **Reset** buttons, and a field that appears if the guess is higher or lower than the correct number. Every time the user clicks **Guess**, the `userNumberBean.check` method is called to reset the maximum or minimum value or, if the guess is correct, to generate a `FacesMessage` to that effect. The method that determines whether each guess is valid is `userNumberBean.validateNumberRange`.

Running the guessnumber-cdi Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `guessnumber-cdi` application.

To Build, Package, and Deploy the guessnumber-cdi Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/cdi
   ```

4. Select the `guessnumber-cdi` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `guessnumber-cdi` project and select **Build**.

   This command builds and packages the application into a WAR file, `guessnumber-cdi.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Build, Package, and Deploy the guessnumber-cdi Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, change to the following directory:

   ```
   jakartaee-examples/tutorial/cdi/guessnumber-cdi/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `guessnumber-cdi.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the guessnumber Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/guessnumber-cdi
   ```

   The **Guess My Number** page opens.

2. On the **Guess My Number** page, enter a number in the **Number** field and click **Guess**.

   The minimum and maximum values are modified, along with the remaining number of guesses.

3. Keep guessing numbers until you get the right answer or run out of guesses.

   If you get the right answer or run out of guesses, the input field and **Guess** button are grayed out.

4. Click **Reset** to play the game again with a new random number.

Using Jakarta EE Interceptors



This section was last updated for Jakarta EE Version 9.1

This chapter discusses how to create interceptor classes and methods that interpose on method invocations or lifecycle events on a target class.

Overview of Interceptors

Interceptors are used in conjunction with Jakarta EE managed classes to allow developers to invoke interceptor methods on an associated target class, in conjunction with method invocations or lifecycle events. Common uses of interceptors are logging, auditing, and profiling.

You can use interceptors with session beans, message-driven beans, and CDI managed beans. In all of these cases, the interceptor target class is the bean class.

An interceptor can be defined within a target class as an interceptor method, or in an associated class called an interceptor class. Interceptor classes contain methods that are invoked in conjunction with the methods or lifecycle events of the target class.

Interceptor classes and methods are defined using metadata annotations, or in the deployment descriptor of the application that contains the interceptors and target classes.



Applications that use the deployment descriptor to define interceptors are not portable across Jakarta EE servers.

Interceptor methods within the target class or in an interceptor class are annotated with one of the metadata annotations defined in Interceptor Metadata Annotations.

Interceptor Metadata Annotations

Interceptor Metadata Annotation	Description
`jakarta.interceptor.AroundConstruct`	Designates the method as an interceptor method that receives a callback after the target class is constructed
`jakarta.interceptor.AroundInvoke`	Designates the method as an interceptor method
`jakarta.interceptor.AroundTimeout`	Designates the method as a timeout interceptor for interposing on timeout methods for enterprise bean timers
`jakarta.annotation.PostConstruct`	Designates the method as an interceptor method for post-construct lifecycle events
`jakarta.annotation.PreDestroy`	Designates the method as an interceptor method for pre-destroy lifecycle events

Interceptor Classes

Interceptor classes may be designated with the optional `jakarta.interceptor.Interceptor` annotation, but interceptor classes are not required to be so annotated. An interceptor class must have a public, no-argument constructor.

The target class can have any number of interceptor classes associated with it. The order in which the interceptor classes are invoked is determined by the order in which the interceptor classes are defined in the `jakarta.interceptor.Interceptors` annotation. However, this order can be overridden in the deployment descriptor.

Interceptor classes may be targets of dependency injection. Dependency injection occurs when the interceptor class instance is created, using the naming context of the associated target class, and before any `@PostConstruct` callbacks are invoked.

Interceptor Lifecycle

Interceptor classes have the same lifecycle as their associated target class. When a target class instance is created, an interceptor class instance is also created for each declared interceptor class in the target class. That is, if the target class declares multiple interceptor classes, an instance of each class is created when the target class instance is created. The target class instance and all interceptor class instances are fully instantiated before any `@PostConstruct` callbacks are invoked, and any `@PreDestroy` callbacks are invoked before the target class and interceptor class instances are destroyed.

Interceptors and CDI

Jakarta Contexts and Dependency Injection (CDI) builds on the basic functionality of Jakarta EE interceptors. For information on CDI interceptors, including a discussion of interceptor binding types, see Using Interceptors in CDI Applications.

Using Interceptors

To define an interceptor, use one of the interceptor metadata annotations listed in Interceptor Metadata Annotations within the target class, or in a separate interceptor class. The following code declares an `@AroundTimeout` interceptor method within a target class:

```
@Stateless
public class TimerBean {
    ...
    @Schedule(minute="*/1", hour="*")
    public void automaticTimerMethod() { ... }

    @AroundTimeout
    public void timeoutInterceptorMethod(InvocationContext ctx) { ... }
    ...
}
```

If you are using interceptor classes, use the `jakarta.interceptor.Interceptors` annotation to declare one or more interceptors at the class or method level of the target class. The following code declares

interceptors at the class level:

```
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean { ... }
```

The following code declares a method-level interceptor class:

```
@Stateless
public class OrderBean {
    ...
    @Interceptors(OrderInterceptor.class)
    public void placeOrder(Order order) { ... }
    ...
}
```

Intercepting Method Invocations

Use the `@AroundInvoke` annotation to designate interceptor methods for managed object methods. Only one around-invoke interceptor method per class is allowed. Around-invoke interceptor methods have the following form:

```
@AroundInvoke
visibility Object method-name(InvocationContext) throws Exception { ... }
```

For example:

```
@AroundInvoke
public void interceptOrder(InvocationContext ctx) { ... }
```

Around-invoke interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

An around-invoke interceptor can call any component or resource that is callable by the target method on which it interposes, can have the same security and transaction context as the target method, and can run in the same Java virtual machine call stack as the target method.

Around-invoke interceptors can throw runtime exceptions and any exception allowed by the `throws` clause of the target method. They may catch and suppress exceptions, and then recover by calling the `InvocationContext.proceed` method.

Using Multiple Method Interceptors

Use the `@Interceptors` annotation to declare multiple interceptors for a target method or class:

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
```

```
        LastInterceptor.class})
public void updateInfo(String info) { ... }
```

The order of the interceptors in the `@Interceptors` annotation is the order in which the interceptors are invoked.

You can also define multiple interceptors in the deployment descriptor. The order of the interceptors in the deployment descriptor is the order in which the interceptors will be invoked:

```
...
<interceptor-binding>
    <target-name>myapp.OrderBean</target-name>
    <interceptor-class>myapp.PrimaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.SecondaryInterceptor.class</interceptor-class>
    <interceptor-class>myapp.LastInterceptor.class</interceptor-class>
    <method-name>updateInfo</method-name>
</interceptor-binding>
...
```

To explicitly pass control to the next interceptor in the chain, call the `InvocationContext.proceed` method.

Data can be shared across interceptors.

- The same `InvocationContext` instance is passed as an input parameter to each interceptor method in the interceptor chain for a particular target method. The `InvocationContext` instance's `contextData` property is used to pass data across interceptor methods. The `contextData` property is a `java.util.Map<String, Object>` object. Data stored in `contextData` is accessible to interceptor methods further down the interceptor chain.
- The data stored in `contextData` is not sharable across separate target class method invocations. That is, a different `InvocationContext` object is created for each invocation of the method in the target class.

Accessing Target Method Parameters from an Interceptor Class

You can use the `InvocationContext` instance passed to each around-invoke method to access and modify the parameters of the target method. The `parameters` property of `InvocationContext` is an array of `Object` instances that corresponds to the parameter order of the target method. For example, for the following target method, the `parameters` property, in the `InvocationContext` instance passed to the around-invoke interceptor method in `PrimaryInterceptor`, is an `Object` array containing two `String` objects (`firstName` and `lastName`) and a `Date` object (`date`):

```
@Interceptors(PrimaryInterceptor.class)
public void updateInfo(String firstName, String lastName, Date date) { ... }
```

You can access and modify the parameters by using the `InvocationContext.getParameters` and `InvocationContext.setParameters` methods, respectively.

Intercepting Lifecycle Callback Events

Interceptors for lifecycle callback events (around-construct, post-construct, and pre-destroy) may be defined in the target class or in interceptor classes. The `jakarta.interceptor.AroundConstruct` annotation designates the method as an interceptor method that interposes on the invocation of the target class's constructor. The `jakarta.annotation.PostConstruct` annotation is used to designate a method as a post-construct lifecycle event interceptor. The `jakarta.annotation.PreDestroy` annotation is used to designate a method as a pre-destroy lifecycle event interceptor.

Lifecycle event interceptors defined within the target class have the following form:

```
void method-name() { ... }
```

For example:

```
@PostConstruct
void initialize() { ... }
```

Lifecycle event interceptors defined in an interceptor class have the following form:

```
void method-name(InvocationContext) { ... }
```

For example:

```
@PreDestroy
void cleanup(InvocationContext ctx) { ... }
```

Lifecycle interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final. Lifecycle interceptors may throw runtime exceptions but cannot throw checked exceptions.

Lifecycle interceptor methods are called in an unspecified security and transaction context. That is, portable Jakarta EE applications should not assume the lifecycle event interceptor method has access to a security or transaction context. Only one interceptor method for each lifecycle event (post-create and pre-destroy) is allowed per class.

Using AroundConstruct Interceptor Methods

`@AroundConstruct` methods are interposed on the invocation of the target class's constructor. Methods decorated with `@AroundConstruct` may only be defined within interceptor classes or superclasses of interceptor classes. You may not use `@AroundConstruct` methods within the target class.

The `@AroundConstruct` method is called after dependency injection has been completed for all interceptors associated with the target class. The target class is created and the target class's constructor injection is performed after all associated `@AroundConstruct` methods have called the

`Invocation.proceed` method. At that point, dependency injection for the target class is completed, and then any `@PostConstruct` callback methods are invoked.

`@AroundConstruct` methods can access the constructed target instance after calling `Invocation.proceed` by calling the `InvocationContext.getTarget` method.



Calling methods on the target instance from an `@AroundConstruct` method is dangerous because dependency injection may not have completed on the target instance.

`@AroundConstruct` methods must call `Invocation.proceed` in order to create the target instance. If an `@AroundConstruct` method does not call `Invocation.proceed`, the target instance will not be created.

Using Multiple Lifecycle Callback Interceptors

You can define multiple lifecycle interceptors for a target class by specifying the interceptor classes in the `@Interceptors` annotation:

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
@Stateless
public class OrderBean { ... }
```

Data stored in the `contextData` property of `InvocationContext` is not sharable across different lifecycle events.

Intercepting Timeout Events

You can define interceptors for Enterprise Bean timer service timeout methods by using the `@AroundTimeout` annotation on methods in the target class or in an interceptor class. Only one `@AroundTimeout` method per class is allowed.

Timeout interceptors have the following form:

```
Object method-name(InvocationContext) throws Exception { ... }
```

For example:

```
@AroundTimeout
protected void timeoutInterceptorMethod(InvocationContext ctx) { ... }
```

Timeout interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Timeout interceptors can call any component or resource callable by the target timeout method, and are invoked in the same transaction and security context as the target method.

Timeout interceptors may access the timer object associated with the target timeout method through the `InvocationContext` instance's `getTimer` method.

Using Multiple Timeout Interceptors

You can define multiple timeout interceptors for a given target class by specifying the interceptor classes containing `@AroundTimeout` interceptor methods in an `@Interceptors` annotation at the class level.

If a target class specifies timeout interceptors in an interceptor class, and also has an `@AroundTimeout` interceptor method within the target class itself, the timeout interceptors in the interceptor classes are called first, followed by the timeout interceptors defined in the target class. For example, in the following example, assume that both the `PrimaryInterceptor` and `SecondaryInterceptor` classes have timeout interceptor methods:

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
@Stateful
public class OrderBean {
    ...
    @AroundTimeout
    private void last(InvocationContext ctx) { ... }
    ...
}
```

The timeout interceptor in `PrimaryInterceptor` will be called first, followed by the timeout interceptor in `SecondaryInterceptor`, and finally the `last` method defined in the target class.

Binding Interceptors to Components

Interceptor binding types are annotations that may be applied to components to associate them with a particular interceptor. Interceptor binding types are typically custom runtime annotation types that specify the interceptor target. Use the `jakarta.interceptor.InterceptorBinding` annotation on the custom annotation definition and specify the target by using `@Target`, setting one or more of `TYPE` (class-level interceptors), `METHOD` (method-level interceptors), `CONSTRUCTOR` (around-construct interceptors), or any other valid target:

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
pubic @interface Logged { ... }
```

Interceptor binding types may also be applied to other interceptor binding types:

```
@Logged
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
```

```
@Inherited
public @interface Secured { ... }
```

Declaring the Interceptor Bindings on an Interceptor Class

Annotate the interceptor class with the interceptor binding type and `@Interceptor` to associate the interceptor binding with the interceptor class:

```
@Logged
@Interceptor
public class LoggingInterceptor {
    @AroundInvoke
    public Object logInvocation(InvocationContext ctx) throws Exception { ... }
    ...
}
```

An interceptor class may declare multiple interceptor binding types, and more than one interceptor class may declare an interceptor binding type.

If the interceptor class intercepts lifecycle callbacks, it can only declare interceptor binding types with `Target(TYPE)`, or in the case of `@AroundConstruct` lifecycle callbacks, `Target(CONSTRUCTOR)`.

Binding a Component to an Interceptor

Add the interceptor binding type annotation to the target component's class, method, or constructor. Interceptor binding types are applied using the same rules as `@Interceptor` annotations:

```
@Logged
public class Message {
    ...
    @Secured
    public void getConfidentialMessage() { ... }
    ...
}
```

If the component has a class-level interceptor binding, it must not be `final` or have any non-`static`, non-`private` `final` methods. If a non-`static`, non-`private` method has an interceptor binding applied to it, it must not be `final`, and the component class cannot be `final`.

Ordering Interceptors

The order in which multiple interceptors are invoked is determined by the following rules.

- Default interceptors are defined in a deployment descriptor, and are invoked first. They may specify the invocation order or override the order specified using annotations. Default interceptors are invoked in the order in which they are defined in the deployment descriptor.
- The order in which the interceptor classes are listed in the `@Interceptors` annotation defines the

order in which the interceptors are invoked. Any `@Priority` settings for interceptors listed within an `@Interceptors` annotation are ignored.

- If the interceptor class has superclasses, the interceptors defined on the superclasses are invoked first, starting with the most general superclass.
- Interceptor classes may set the priority of the interceptor methods by setting a value within a `jakarta.annotation.Priority` annotation.
- After the interceptors defined within interceptor classes have been invoked, the target class's constructor, around-invoke, or around-timeout interceptors are invoked in the same order as the interceptors within the `@Interceptors` annotation.
- If the target class has superclasses, any interceptors defined on the superclasses are invoked first, starting with the most general superclass.

The `@Priority` annotation requires an `int` value as an element. The lower the number, the higher the priority of the associated interceptor.



The invocation order of interceptors with the same priority value is implementation-specific.

The `jakarta.interceptor.Interceptor.Priority` class defines the priority constants listed in Interceptor Priority Constants.

Interceptor Priority Constants

Priority Constant	Value	Description
`PLATFORM_BEFORE`	0	Interceptors defined by the Jakarta EE Platform and intended to be invoked early in the invocation chain should use the range between `PLATFORM_BEFORE` and `LIBRARY_BEFORE`. These interceptors have the highest priority.
`LIBRARY_BEFORE`	1000	Interceptors defined by extension libraries that should be invoked early in the interceptor chain should use the range between `LIBRARY_BEFORE` and `APPLICATION`.
`APPLICATION`	2000	Interceptors defined by applications should use the range between `APPLICATION` and `LIBRARY_AFTER`.
`LIBRARY_AFTER`	3000	Low priority interceptors defined by extension libraries should use the range between `LIBRARY_AFTER` and `PLATFORM_AFTER`.
`PLATFORM_AFTER`	4000	Low priority interceptors defined by the Jakarta EE Platform should have values higher than `PLATFORM_AFTER`.



Negative priority values are reserved by the Interceptors specification for future use, and should not be used.

The following code snippet shows how to use the priority constants in an application-defined interceptor:

```
@Interceptor
@Priority(Interceptor.Priority.APPLICATION+200)
public class MyInterceptor { ... }
```

The interceptor Example Application

The `interceptor` example demonstrates how to use an interceptor class, containing an `@AroundInvoke` interceptor method, with a stateless session bean.

The `HelloBean` stateless session bean is a simple enterprise bean with two business methods, `getName` and `setName`, to retrieve and modify a string. The `setName` business method has an `@Interceptors` annotation that specifies an interceptor class, `HelloInterceptor`, for that method:

```
@Interceptors(HelloInterceptor.class)
public void setName(String name) {
    this.name = name;
}
```

The `HelloInterceptor` class defines an `@AroundInvoke` interceptor method, `modifyGreeting`, that converts the string passed to `HelloBean.setName` to lowercase:

```
@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}
```

The parameters to `HelloBean.setName` are retrieved and stored in an `Object` array by calling the `InvocationContext.getParameters` method. Because `setName` only has one parameter, it is the first and only element in the array. The string is set to lowercase and stored in the `parameters` array, then passed to `InvocationContext.setParameters`. To return control to the session bean, `InvocationContext.proceed` is called.

The user interface of `interceptor` is a JavaServer Faces web application that consists of two Facelets views: `index.xhtml`, which contains a form for entering the name, and `response.xhtml`, which displays the final name.

Running the interceptor Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `interceptor` example.

To Run the interceptor Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/ejb
   ```

4. Select the `interceptor` folder and click Open Project.
5. In the Projects tab, right-click the `interceptor` project and select Run.

   This will compile, deploy, and run the `interceptor` example, opening a web browser to the following URL:

   ```
   http://localhost:8080/interceptor/
   ```

6. Enter a name into the form and click Submit.

   The name will be converted to lowercase by the method interceptor defined in the `HelloInterceptor` class.

To Run the interceptor Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. Go to the following directory:

   ```
   jakartaee-examples/tutorial/ejb/interceptor/
   ```

3. To compile the source files and package the application, use the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `interceptor.war`, located in the `target` directory. The WAR file is then deployed to GlassFish Server.

4. Open the following URL in a web browser:

   ```
   http://localhost:8080/interceptor/
   ```

5. Enter a name into the form and click Submit.

   The name will be converted to lowercase by the method interceptor defined in the `HelloInterceptor` class.

Jakarta REST

Building RESTful Web Services with Jakarta REST

This chapter describes the REST architecture, RESTful web services, and the Jakarta RESTful Web Services.

Jakarta REST makes it easy for developers to build RESTful web services using the Java programming language.

What Are RESTful Web Services?

RESTful web services are loosely coupled, lightweight web services that are particularly well suited for creating APIs for clients spread out across the internet. Representational State Transfer (REST) is an architectural style of client-server application centered around the transfer of representations of resources through requests and responses. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are represented by documents and are acted upon by using a set of simple, well-defined operations.

For example, a REST resource might be the current weather conditions for a city. The representation of that resource might be an XML document, an image file, or an HTML page. A client might retrieve a particular representation, modify the resource by updating its data, or delete the resource entirely.

The REST architectural style is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

- Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See The @Path Annotation and URI Path Templates for more information.
- Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.
- Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and other document formats. Metadata about the resource is available and used, for example, to

control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

- Stateful interactions through links: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and Extracting Request Parameters in the Jakarta REST Overview document for more information.

Creating a RESTful Root Resource Class

Root resource classes are "plain old Java objects" (POJOs) that are either annotated with `@Path` or have at least one method annotated with `@Path` or a request method designator, such as `@GET`, `@PUT`, `@POST`, or `@DELETE`. Resource methods are methods of a resource class annotated with a request method designator. This section explains how to use Jakarta REST to annotate Java classes to create RESTful web services.

Developing RESTful Web Services with Jakarta REST

Jakarta REST is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The Jakarta REST API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with Jakarta REST annotations to define resources and the actions that can be performed on those resources. Jakarta REST annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource. A Jakarta EE application archive containing Jakarta REST resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Jakarta EE server.

Summary of Jakarta REST Annotations lists some of the Java programming annotations that are defined by Jakarta REST, with a brief description of how each is used. Further information on the Jakarta REST APIs can be viewed at https://jakarta.ee/specifications/platform/9/apidocs/.

Summary of Jakarta REST Annotations

Annotation	Description
`@Path`	The `@Path` annotation's value is a relative URI path indicating where the Java class will be hosted: for example, `/helloworld`. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: `/helloworld/{username}`.
`@GET`	The `@GET` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.

Annotation	Description
`@POST`	The `@POST` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PUT`	The `@PUT` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@DELETE`	The `@DELETE` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@HEAD`	The `@HEAD` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@OPTIONS`	The `@OPTIONS` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP OPTIONS requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PATCH`	The `@PATCH` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PATCH requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PathParam`	The `@PathParam` annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.
`@QueryParam`	The `@QueryParam` annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
`@Consumes`	The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
`@Produces`	The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, `"text/plain"`.
`@Provider`	The `@Provider` annotation is used for anything that is of interest to the Jakarta REST runtime, such as `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built using `Response.ResponseBuilder`.

Annotation	Description
`@ApplicationPath`	The `@ApplicationPath` annotation is used to define the URL mapping for the application. The path specified by `@ApplicationPath` is the base URI for all resource URIs specified by `@Path` annotations in the resource class. You may only apply `@ApplicationPath` to a subclass of `jakarta.ws.rs.core.Application`.

Overview of a Jakarta REST Application

The following code sample is a very simple example of a root resource class that uses Jakarta REST annotations:

```
package ee.jakarta.tutorial.hello;

import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.PUT;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.UriInfo;

/**
 * Root resource (exposed at "helloworld" path)
 */
@Path("helloworld")
public class HelloWorld {
    @Context
    private UriInfo context;

    /** Creates a new instance of HelloWorld */
    public HelloWorld() {
    }

    /**
     * Retrieves representation of an instance of helloWorld.HelloWorld
     * @return an instance of java.lang.String
     */
    @GET
    @Produces("text/html")
    public String getHtml() {
        return "<html lang=\"en\"><body><h1>Hello, World!!</h1></body></html>";
    }
}
```

The following sections describe the annotations used in this example.

- The `@Path` annotation's value is a relative URI path. In the preceding example, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the `@Path` annotation,

with a static URI path. Variables can be embedded in the URIs. URI path templates are URIs with variables embedded within the URI syntax.

- The `@GET` annotation is a request method designator, along with `@POST`, `@PUT`, `@DELETE`, and `@HEAD`, defined by Jakarta REST and corresponding to the similarly named HTTP methods. In the example, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- The `@Produces` annotation is used to specify the MIME media types a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type `"text/html"`.
- The `@Consumes` annotation is used to specify the MIME media types a resource can consume that were sent by the client. The example could be modified to set the message returned by the `getHtml` method, as shown in this code example:

```
@POST
@Consumes("text/plain")
public void postHtml(String message) {
    // Store the message
}
```

The @Path Annotation and URI Path Templates

The `@Path` annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource. The `@Path` annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the Jakarta REST runtime responds.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces (`{` and `}`). For example, look at the following `@Path` annotation:

```
@Path("/users/{username}")
```

In this kind of example, a user is prompted to type his or her name, and then a Jakarta REST web service configured to respond to requests to this URI path template responds. For example, if the user types the user name "Galileo," the web service responds to the following URL:

```
http://example.com/users/Galileo
```

To obtain the value of the user name, the `@PathParam` annotation may be used on the method parameter of a request method, as shown in the following code example:

```
@Path("/users/{username}")
public class UserResource {
```

```
    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

By default, the URI variable must match the regular expression `"[^/]+?"`. This variable may be customized by specifying a different regular expression after the variable name. For example, if a user name must consist only of lowercase and uppercase alphanumeric characters, override the default regular expression in the variable definition:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

In this example, the `username` variable will match only user names that begin with one uppercase or lowercase letter and zero or more alphanumeric characters and the underscore character. If a user name does not match that template, a 404 (Not Found) response will be sent to the client.

A `@Path` value isn't required to have leading or trailing slashes (/). The Jakarta REST runtime parses URI path templates the same way, whether or not they have leading or trailing slashes.

A URI path template has one or more variables, with each variable name surrounded by braces: `{` to begin the variable name and `}` to end it. In the preceding example, `username` is the variable name. At runtime, a resource configured to respond to the preceding URI path template will attempt to process the URI data that corresponds to the location of `{username}` in the URI as the variable data for `username`.

For example, if you want to deploy a resource that responds to the URI path template `http://example.com/myContextRoot/resources/{name1}/{name2}/`, you must first deploy the application to a Jakarta EE server that responds to requests to the `http://example.com/myContextRoot` URI and then decorate your resource with the following `@Path` annotation:

```
@Path("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

In this example, the URL pattern for the Jakarta REST helper servlet, specified in `web.xml`, is the default:

```
<servlet-mapping>
      <servlet-name>jakarta.ws.rs.core.Application</servlet-name>
      <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

A variable name can be used more than once in the URI path template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with `%20`.

When defining URI path templates, be careful that the resulting URI after substitution is valid.

Examples of URI Path Templates lists some examples of URI path template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

- `name1`: `james`
- `name2`: `gatz`
- `name3`:
- `location`: `Main%20Street`
- `question`: `why`



The value of the `name3` variable is an empty string.

Examples of URI Path Templates

URI Path Template	URI After Substitution
http://example.com/{name1}/{name2}/	http://example.com/james/gatz/
http://example.com/{question}/{question}/{question}/	http://example.com/why/why/why/
http://example.com/maps/{location}	http://example.com/maps/Main%20Street
http://example.com/{name3}/home/	http://example.com//home/

Responding to HTTP Methods and Requests

The behavior of a resource is determined by the HTTP methods (typically, GET, POST, PUT, or DELETE) to which the resource is responding.

The Request Method Designator Annotations

Request method designator annotations are runtime annotations, defined by Jakarta REST, that correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods by using the request method designator annotations. The behavior of a resource is determined by which HTTP method the resource is responding to. Jakarta REST defines a set of request method designators for the common HTTP methods GET, POST, PUT, DELETE, and HEAD; you can also create your own custom request method designators. Creating custom request method designators is outside the scope of this document.

The following example shows the use of the PUT method to create or update a storage container:

```
@PUT
public Response putContainer() {
```

```
    System.out.println("PUT CONTAINER " + container);

    URI uri =  uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

By default, the Jakarta REST runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented. For HEAD, the runtime will invoke the implemented GET method, if present, and ignore the response entity, if set. For OPTIONS, the `Allow` response header will be set to the set of HTTP methods supported by the resource. In addition, the Jakarta REST runtime will return a Web Application Definition Language (WADL) document describing the resource; see https://www.w3.org/Submission/wadl/ for more information.

Methods decorated with request method designators must return `void`, a Java programming language type, or a `jakarta.ws.rs.core.Response` object. Multiple parameters may be extracted from the URI by using the `@PathParam` or `@QueryParam` annotations, as described in Extracting Request Parameters. Conversion between Java types and an entity body is the responsibility of an entity provider, such as `MessageBodyReader` or `MessageBodyWriter`. Methods that need to provide additional metadata with a response should return an instance of the `Response` class. The `ResponseBuilder` class provides a convenient way to create a `Response` instance using a builder pattern. The HTTP PUT and POST methods expect an HTTP request body, so you should use a `MessageBodyReader` for methods that respond to PUT and POST requests.

Both `@PUT` and `@POST` can be used to create or update a resource. POST can mean anything, so when using POST, it is up to the application to define the semantics. PUT has well-defined semantics. When using PUT for creation, the client declares the URI for the newly created resource.

PUT has very clear semantics for creating and updating a resource. The representation the client sends must be the same representation that is received using a GET, given the same media type. PUT does not allow a resource to be partially updated, a common mistake when attempting to use the PUT method. A common application pattern is to use POST to create a resource and return a `201` response with a location header whose value is the URI to the newly created resource. In this pattern, the web service declares the URI for the newly created resource.

Using Entity Providers to Map HTTP Response and Request Entity Bodies

Entity providers supply mapping services between representations and their associated Java types. The two types of entity providers are `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the

response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built by using `Response.ResponseBuilder`.

Types Supported for HTTP Request and Response Entity Bodies shows the standard types that are supported automatically for HTTP request and response entity bodies. You need to write an entity provider only if you are not choosing one of these standard types.

Types Supported for HTTP Request and Response Entity Bodies

Java Type	Supported Media Types
`byte[]`	All media types (`*/*`)
`java.lang.String`	All text media types (`text/*`)
`java.io.InputStream`	All media types (`*/*`)
`java.io.Reader`	All media types (`*/*`)
`java.io.File`	All media types (`*/*`)
`jakarta.activation.DataSource`	All media types (`*/*`)
`javax.xml.transform.Source`	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`jakarta.xml.bind.JAXBElement` and application-supplied Jakarta XML Binding classes	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`MultivaluedMap<String, String>`	Form content (`application/x-www-form-urlencoded`)
`StreamingOutput`	All media types (`/`), `MessageBodyWriter` only

The following example shows how to use `MessageBodyReader` with the `@Consumes` and `@Provider` annotations:

```
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> { }
```

The following example shows how to use `MessageBodyWriter` with the `@Produces` and `@Provider` annotations:

```
@Produces("text/html")
@Provider
public class FormWriter implements
        MessageBodyWriter<Hashtable<String, String>> { }
```

The following example shows how to use `ResponseBuilder`:

```
@GET
public Response getItem() {
    System.out.println("GET ITEM " + container + " " + item);

    Item i = MemoryStore.MS.getItem(container, item);
    if (i == null)
        throw new NotFoundException("Item not found");
    Date lastModified = i.getLastModified().getTime();
    EntityTag et = new EntityTag(i.getDigest());
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);
    if (rb != null)
        return rb.build();

    byte[] b = MemoryStore.MS.getItemData(container, item);
    return Response.ok(b, i.getMimeType()).
            lastModified(lastModified).tag(et).build();
}
```

Using @Consumes and @Produces to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the following annotations:

- `jakarta.ws.rs.Consumes`
- `jakarta.ws.rs.Produces`

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

The @Produces Annotation

The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If `@Produces` is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If applied at the method level, the annotation overrides any `@Produces` annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the Jakarta REST runtime sends back an HTTP "406 Not Acceptable" error.

The value of `@Produces` is an array of `String` of MIME types or a comma-separated list of `MediaType`

constants. For example:

```
@Produces({"image/jpeg,image/png"})
```

The following example shows how to apply `@Produces` at both the class and method levels:

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

The `doGetAsPlainText` method defaults to the MIME media type of the `@Produces` annotation at the class level. The `doGetAsHtml` method's `@Produces` annotation overrides the class-level `@Produces` setting and specifies that the method can produce HTML rather than plain text.

`@Produces` can also use the constants defined in the `jakarta.ws.rs.core.MediaType` class to specify the media type. For example, specifying `MediaType.APPLICATION_XML` is equivalent to specifying `"application/xml"`.

```
@Produces(MediaType.APPLICATION_XML)
@GET
public Customer getCustomer() { ... }
```

If a resource class is capable of producing more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically, the `Accept` header of the HTTP request declares what is most acceptable. For example, if the `Accept` header is `Accept: text/plain`, the `doGetAsPlainText` method will be invoked. Alternatively, if the `Accept` header is `Accept: text/plain;q=0.9, text/html`, which declares that the client can accept media types of `text/plain` and `text/html` but prefers the latter, the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same `@Produces` declaration. The following code example shows how this is done:

```
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
```

```
    ...
}
```

The `doGetAsXmlOrJson` method will get invoked if either of the media types `application/xml` or `application/json` is acceptable. If both are equally acceptable, the former will be chosen because it occurs first. The preceding examples refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors. For more information, see the API documentation for the constant field values of `jakarta.ws.rs.core.MediaType`.

The @Consumes Annotation

The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If `@Consumes` is applied at the class level, all the response methods accept the specified MIME types by default. If applied at the method level, `@Consumes` overrides any `@Consumes` annotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the Jakarta REST runtime sends back an HTTP 415 ("Unsupported Media Type") error.

The value of `@Consumes` is an array of `String` of acceptable MIME types, or a comma-separated list of `MediaType` constants. For example:

```
@Consumes({"text/plain,text/html"})
```

This is the equivalent of:

```
@Consumes({MediaType.TEXT_PLAIN,MediaType.TEXT_HTML})
```

The following example shows how to apply `@Consumes` at both the class and method levels:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

The `doPost` method defaults to the MIME media type of the `@Consumes` annotation at the class level.

The `doPost2` method overrides the class level `@Consumes` annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 ("Unsupported Media Type") error is returned to the client.

The `HelloWorld` example discussed previously in this section can be modified to set the message by using `@Consumes`, as shown in the following code example:

```
@POST
@Consumes("text/html")
public void postHtml(String message) {
    // Store the message
}
```

In this example, the Java method will consume representations identified by the MIME media type `text/plain`. Note that the resource method returns `void`. This means that no representation is returned and that a response with a status code of HTTP 204 ("No Content") will be returned.

Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the `@PathParam` parameter to extract a path parameter from the path component of the request URL that matched the path declared in `@Path`.

You can extract the following types of parameters for use in your resource class:

- Query
- URI path
- Form
- Cookie
- Header
- Matrix

Query parameters are extracted from the request URI query parameters and are specified by using the `jakarta.ws.rs.QueryParam` annotation in the method parameter arguments. The following example demonstrates using `@QueryParam` to extract query parameters from the `Query` component of the request URL:

```
@Path("smooth")
@GET
public Response smooth(
        @DefaultValue("2") @QueryParam("step") int step,
        @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
        @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
        @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
```

```
        @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
        @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
        @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
        ) { ... }
```

If the query parameter `step` exists in the query component of the request URI, the value of `step` will be extracted and parsed as a 32-bit signed integer and assigned to the `step` method parameter. If `step` does not exist, a default value of 2, as declared in the `@DefaultValue` annotation, will be assigned to the `step` method parameter. If the `step` value cannot be parsed as a 32-bit signed integer, an HTTP 400 ("Client Error") response is returned.

User-defined Java programming language types may be used as query parameters. The following code example shows the `ColorParam` class used in the preceding query parameter example:

```
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
                return ((Color)f.get(null)).getRGB();
            } catch (Exception e) {
                throw new WebApplicationException(400);
            }
        }
    }
}
```

The constructor for `ColorParam` takes a single `String` parameter.

Both `@QueryParam` and `@PathParam` can be used only on the following Java types.

- All primitive types except `char`.
- All wrapper classes of primitive types except `Character`.
- Any class with a constructor that accepts a single `String` argument.
- Any class with the static method named `valueOf(String)` that accepts a single `String` argument.
- `List<T>`, `Set<T>`, or `SortedSet<T>`, where T matches the already listed criteria. Sometimes,

parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values.

If `@DefaultValue` is not used in conjunction with `@QueryParam`, and the query parameter is not present in the request, the value will be an empty collection for `List`, `Set`, or `SortedSet`; null for other object types; and the default for primitive types.

URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation. URI parameters are specified using the `jakarta.ws.rs.PathParam` annotation in the method parameter arguments. The following example shows how to use `@Path` variables and the `@PathParam` annotation in a method:

```
@Path("/{username}")
public class MyResourceBean {
    ...
    @GET
    public String printUsername(@PathParam("username") String userId) {
        ...
    }
}
```

In the preceding snippet, the URI path template variable name `username` is specified as a parameter to the `printUsername` method. The `@PathParam` annotation is set to the variable name `username`. At runtime, before `printUsername` is called, the value of `username` is extracted from the URI and cast to a `String`. The resulting `String` is then available to the method as the `userId` variable.

If the URI path template variable cannot be cast to the specified type, the Jakarta REST runtime returns an HTTP 400 ("Bad Request") error to the client. If the `@PathParam` annotation cannot be cast to the specified type, the Jakarta REST runtime returns an HTTP 404 ("Not Found") error to the client.

The `@PathParam` parameter and the other parameter-based annotations (`@MatrixParam`, `@HeaderParam`, `@CookieParam`, and `@FormParam`) obey the same rules as `@QueryParam`.

Cookie parameters, indicated by decorating the parameter with `jakarta.ws.rs.CookieParam`, extract information from the cookies declared in cookie-related HTTP headers. Header parameters, indicated by decorating the parameter with `jakarta.ws.rs.HeaderParam`, extract information from the HTTP headers. Matrix parameters, indicated by decorating the parameter with `jakarta.ws.rs.MatrixParam`, extract information from URL path segments.

Form parameters, indicated by decorating the parameter with `jakarta.ws.rs.FormParam`, extract information from a request representation that is of the MIME media type `application/x-www-form-urlencoded` and conforms to the encoding specified by HTML forms, as described in https://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1. This parameter is very useful for extracting information sent by POST in HTML forms.

The following example extracts the `name` form parameter from the POST form data:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}
```

To obtain a general map of parameter names and values for query and path parameters, use the following code:

```
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

The following method extracts header and cookie parameter names and values into a map:

```
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

In general, `@Context` can be used to obtain contextual Java types related to the request or response.

For form parameters, it is possible to do the following:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```

Configuring Jakarta REST Applications

A Jakarta REST application consists of at least one resource class packaged within a WAR file. The base URI from which an application's resources respond to requests can be set one of two ways:

- Using the `@ApplicationPath` annotation in a subclass of `jakarta.ws.rs.core.Application` packaged within the WAR
- Using the `servlet-mapping` tag within the WAR's `web.xml` deployment descriptor

Configuring a Jakarta REST Application Using a Subclass of Application

Create a subclass of `jakarta.ws.rs.core.Application` to manually configure the environment in which the REST resources defined in your resource classes are run, including the base URI. Add a

class-level `@ApplicationPath` annotation to set the base URI.

```
@ApplicationPath("/webapi")
public class MyApplication extends Application { ... }
```

In the preceding example, the base URI is set to `/webapi`, which means that all resources defined within the application are relative to `/webapi`.

By default, all the resources in an archive will be processed for resources. Override the `getClasses` method to manually register the resource classes in the application with the Jakarta REST runtime.

```
@Override
public Set<Class<?>> getClasses() {
    final Set<Class<?>> classes = new HashSet<>();
    // register root resource
    classes.add(MyResource.class);
    return classes;
}
```

Configuring the Base URI in web.xml

The base URI for a Jakarta REST application can be set using a `servlet-mapping` tag in the `web.xml` deployment descriptor, using the `Application` class name as the servlet.

```
<servlet-mapping>
    <servlet-name>jakarta.ws.rs.core.Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
```

This setting will also override the path set by `@ApplicationPath` when using an `Application` subclass.

```
<servlet-mapping>
   <servlet-name>com.example.rest.MyApplication</servlet-name>
   <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

Example Applications for Jakarta REST

This section provides an introduction to creating, deploying, and running your own Jakarta REST applications. This section demonstrates the steps that are needed to create, build, deploy, and test a very simple web application that uses Jakarta REST annotations.

Creating a Simple RESTful Web Service

This section explains how to use NetBeans IDE to create a RESTful web service using a Maven archetype. The archetype generates a skeleton for the application, and you simply need to

implement the appropriate method.

You can find a version of this application at `jakartaee-examples/tutorial/rest/hello/`.

To Create a RESTful Web Service Using NetBeans IDE

1. Ensure you have installed the tutorial archetypes as described in Installing the Tutorial Archetypes.
2. In NetBeans IDE, create a simple web application using the `jaxrs-service-archetype` Maven archetype. This archetype creates a very simple "Hello, World" web application.
   a. From the File menu, choose New Project.
   b. From Categories, select Maven. From Projects, select Project From Archetype. Click Next.
   c. Under Search enter `rest-service`, select the `rest-service-archetype`, and click Next.
   d. Under Project Name enter `HelloWorldApplication`, set the Project Location, and set the Package name to `ee.jakarta.tutorial.hello`, and click Finish.

   The project is created.
3. In `HelloWorld.java`, find the `getHtml()` method. Replace the `//TODO` comment with the following text, so that the finished product resembles the following method:

```
@GET
@Produces("text/html")
public String getHtml() {
    return "<html lang=\"en\"><body><h1>Hello, World!!</body></h1></html>";
}
```



Because the MIME type produced is HTML, you can use HTML tags in your return statement.

4. Right-click the `HelloWorldApplication` project in the Projects pane and select Run.

   This will build and deploy the application to GlassFish Server.
5. In a browser, open the following URL:

```
http://localhost:8080/HelloWorldApplication/HelloWorldApplication
```

   A browser window opens and displays the return value of `Hello, World!!`

For other sample applications that demonstrate deploying and running Jakarta REST applications using NetBeans IDE, see The rsvp Example Application and Your First Cup: An Introduction to the Jakarta EE Platform at https://eclipse-ee4j.github.io/jakartaee-firstcup/toc.html. You may also look at the tutorials on the NetBeans IDE tutorial site, such as the one titled "Getting Started with RESTful Web Services" at https://netbeans.apache.org/kb/docs/websvc/rest.html. This tutorial includes a section on creating a CRUD application from a database. Create, read, update, and delete (CRUD) are

the four basic functions of persistent storage and relational databases.

The rsvp Example Application

The `rsvp` example application, located in the `jakartaee-examples/tutorial/rest/rsvp/` directory, allows invitees to an event to indicate whether they will attend. The events, people invited to the event, and the responses to the invite are stored in Apache Derby using Jakarta Persistence. The Jakarta REST resources in `rsvp` are exposed in a stateless session enterprise bean.

Components of the rsvp Example Application

The three enterprise beans in the `rsvp` example application are `rsvp.ejb.ConfigBean`, `rsvp.ejb.StatusBean`, and `rsvp.ejb.ResponseBean`.

`ConfigBean` is a singleton session bean that initializes the data in the database.

`StatusBean` exposes a Jakarta REST resource for displaying the current status of all invitees to an event. The URI path template is declared first on the class and then on the `getEvent` method:

```
@Stateless
@Named
@Path("/status")
public class StatusBean {
    ...
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Path("{eventId}/")
    public Event getEvent(@PathParam("eventId") Long eventId) {
        ...
    }
}
```

The combination of the two `@Path` annotations results in the following URI path template:

```
@Path("/status/{eventId}/")
```

The URI path variable `eventId` is a `@PathParam` variable in the `getEvent` method, which responds to HTTP GET requests and has been annotated with `@GET`. The `eventId` variable is used to look up all the current responses in the database for that particular event.

`ResponseBean` exposes a Jakarta REST resource for setting an invitee's response to a particular event. The URI path template for `ResponseBean` is declared as follows:

```
@Path("/{eventId}/{inviteId}")
```

Two URI path variables are declared in the path template: `eventId` and `inviteId`. As in `StatusBean`, `eventId` is the unique ID for a particular event. Each invitee to that event has a unique ID for the

invitation, and that is the `inviteId`. Both of these path variables are used in two Jakarta REST methods in `ResponseBean`: `getResponse` and `putResponse`. The `getResponse` method responds to HTTP GET requests and displays the invitee's current response and a form to change the response.

The `ee.jakarta.tutorial.rsvp.rest.RsvpApplication` class defines the root application path for the resources by applying the `jakarta.ws.rs.ApplicationPath` annotation at the class level.

```
@ApplicationPath("/webapi")
public class RsvpApplication extends Application {
}
```

An invitee who wants to change his or her response selects the new response and submits the form data, which is processed as an HTTP POST request by the `putResponse` method. The new response is extracted from the HTTP POST request and stored as the `userResponse` string. The `putResponse` method uses `userResponse`, `eventId`, and `inviteId` to update the invitee's response in the database.

The events, people, and responses in `rsvp` are encapsulated in Jakarta Persistence entities. The `rsvp.entity.Event`, `rsvp.entity.Person`, and `rsvp.entity.Response` entities respectively represent events, invitees, and responses to an event.

The `rsvp.util.ResponseEnum` class declares an enumerated type that represents all the possible response statuses an invitee may have.

The web application also includes two CDI managed beans, `StatusManager` and `EventManager`, which use the Jakarta REST Client API to call the resources exposed in `StatusBean` and `ResponseBean`. For information on how the Client API is used in `rsvp`, see The Client API in the rsvp Example Application.

Running the rsvp Example Application

Both NetBeans IDE and Maven can be used to deploy and run the `rsvp` example application.

To Run the rsvp Example Application Using NetBeans IDE

1. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
2. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
3. From the **File** menu, choose **Open Project**.
4. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/rest
   ```

5. Select the `rsvp` folder.
6. Click **Open Project**.
7. In the **Projects** tab, right-click the `rsvp` project and select **Run**.

The project will be compiled, assembled, and deployed to GlassFish Server. A web browser window will open to the following URL:

```
http://localhost:8080/rsvp/index.xhtml
```

8. In the web browser window, click the Event status link for the Duke's Birthday event.

   You'll see the current invitees and their responses.

9. Click the current response of one of the invitees in the Status column of the table, select a new response, and click Update your status.

   The invitee's new status should now be displayed in the table of invitees and their response statuses.

To Run the rsvp Example Application Using Maven

1. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
2. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
3. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/rest/rsvp/
   ```

4. Enter the following command:

   ```
   mvn install
   ```

   This command builds, assembles, and deploys `rsvp` to GlassFish Server.

5. Open a web browser window to the following URL:

   ```
   http://localhost:8080/rsvp/
   ```

6. In the web browser window, click the Event status link for the Duke's Birthday event.

   You'll see the current invitees and their responses.

7. Click the current response of one of the invitees in the Status column of the table, select a new response, and click Update your status.

   The invitee's new status should now be displayed in the table of invitees and their response statuses.

Real-World Examples

Most blog sites use RESTful web services. These sites involve downloading XML files, in RSS or Atom format, that contain lists of links to other resources. Other websites and web applications that use REST-like developer interfaces to data include Twitter and Amazon S3 (Simple Storage Service). With Amazon S3, buckets and objects can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface. The examples that ship with Jersey include a storage service example with a RESTful interface.

Further Information about Jakarta REST

For more information about RESTful web services and Jakarta REST, see

- "Fielding Dissertation: Chapter 5: Representational State Transfer (REST)": https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RESTful Web Services, by Leonard Richardson and Sam Ruby, available from O'Reilly Media at https://www.oreilly.com/library/view/restful-web-services/9780596529260/
- Jakarta RESTful Web Services 3.0 specification: https://jakarta.ee/specifications/restful-ws/3.0/
- Jersey project: https://eclipse-ee4j.github.io/jersey/

Accessing REST Resources with the Jakarta REST Client API

This chapter describes the Jakarta REST Client API and includes examples of how to access REST resources using the Java programming language.

Jakarta REST provides a client API for accessing REST resources from other Java applications.

Overview of the Client API

The Jakarta REST Client API provides a high-level API for accessing any REST resources, not just Jakarta REST services. The Client API is defined in the `jakarta.ws.rs.client` package.

Creating a Basic Client Request Using the Client API

The following steps are needed to access a REST resource using the Client API.

1. Obtain an instance of the `jakarta.ws.rs.client.Client` interface.
2. Configure the `Client` instance with a target.
3. Create a request based on the target.
4. Invoke the request.

The Client API is designed to be fluent, with method invocations chained together to configure and submit a request to a REST resource in only a few lines of code.

```
Client client = ClientBuilder.newClient();
String name = client.target("http://example.com/webapi/hello")
```

```
        .request(MediaType.TEXT_PLAIN)
        .get(String.class);
```

In this example, the client instance is first created by calling the `jakarta.ws.rs.client.ClientBuilder.newClient` method. Then, the request is configured and invoked by chaining method calls together in one line of code. The `Client.target` method sets the target based on a URI. The `jakarta.ws.rs.client.WebTarget.request` method sets the media type for the returned entity. The `jakarta.ws.rs.client.Invocation.Builder.get` method invokes the service using an HTTP `GET` request, setting the type of the returned entity to `String`.

Obtaining the Client Instance

The `Client` interface defines the actions and infrastructure a REST client requires to consume a RESTful web service. Instances of `Client` are obtained by calling the `ClientBuilder.newClient` method.

```
Client client = ClientBuilder.newClient();
```

Use the `close` method to close `Client` instances after all the invocations for the target resource have been performed:

```
Client client = ClientBuilder.newClient();
...
client.close();
```

`Client` instances are heavyweight objects. For performance reasons, limit the number of `Client` instances in your application, as the initialization and destruction of these instances may be expensive in your runtime environment.

Setting the Client Target

The target of a client, the REST resource at a particular URI, is represented by an instance of the `jakarta.ws.rs.client.WebTarget` interface. You obtain a `WebTarget` instance by calling the `Client.target` method and passing in the URI of the target REST resource.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi");
```

For complex REST resources, it may be beneficial to create several instances of `WebTarget`. In the following example, a base target is used to construct several other targets that represent different services provided by a REST resource.

```
Client client = ClientBuilder.newClient();
WebTarget base = client.target("http://example.com/webapi");
// WebTarget at http://example.com/webapi/read
WebTarget read = base.path("read");
```

```
// WebTarget at http://example.com/webapi/write
WebTarget write = base.path("write");
```

The `WebTarget.path` method creates a new `WebTarget` instance by appending the current target URI with the path that was passed in.

Setting Path Parameters in Targets

Path parameters in client requests can be specified as URI template parameters, similar to the template parameters used when defining a resource URI in a Jakarta REST service. Template parameters are specified by surrounding the template variable with braces (`{}`). Call the `resolveTemplate` method to substitute the `{username}`, and then call the `queryParam` method to add another variable to pass.

```
WebTarget myResource = client.target("http://example.com/webapi/read")
        .path("{userName}")
        .resolveTemplate("userName", "janedoe")
        .queryParam("chapter", "1");
// http://example.com/webapi/read/janedoe?chapter=1
Response response = myResource.request(...).get();
```

Invoking the Request

After setting and applying any configuration options to the target, call one of the `WebTarget.request` methods to begin creating the request. This is usually accomplished by passing to `WebTarget.request` the accepted media response type for the request either as a string of the MIME type or using one of the constants in `jakarta.ws.rs.core.MediaType`. The `WebTarget.request` method returns an instance of `jakarta.ws.rs.client.Invocation.Builder`, a helper object that provides methods for preparing the client request.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Invocation.Builder builder = myResource.request(MediaType.TEXT_PLAIN);
```

Using a `MediaType` constant is equivalent to using the string defining the MIME type.

```
Invocation.Builder builder = myResource.request("text/plain");
```

After setting the media type, invoke the request by calling one of the methods of the `Invocation.Builder` instance that corresponds to the type of HTTP request the target REST resource expects. These methods are:

- `get()`
- `post()`
- `delete()`

- `put()`
- `head()`
- `options()`

For example, if the target REST resource is for an HTTP GET request, call the `Invocation.Builder.get` method. The return type should correspond to the entity returned by the target REST resource.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .get(String.class);
```

If the target REST resource is expecting an HTTP POST request, call the `Invocation.Builder.post` method.

```
Client client = ClientBuilder.newClient();
StoreOrder order = new StoreOrder(...);
WebTarget myResource = client.target("http://example.com/webapi/write");
TrackingNumber trackingNumber = myResource.request(MediaType.APPLICATION_XML)
                                    .post(Entity.xml(order), TrackingNumber.class);
```

In the preceding example, the return type is a custom class and is retrieved by setting the type in the `Invocation.Builder.post(Entity<?> entity, Class<T> responseType)` method as a parameter.

If the return type is a collection, use `jakarta.ws.rs.core.GenericType<T>` as the response type parameter, where `T` is the collection type:

```
List<StoreOrder> orders = client.target("http://example.com/webapi/read")
        .path("allOrders")
        .request(MediaType.APPLICATION_XML)
        .get(new GenericType<List<StoreOrder>>() {});
```

This preceding example shows how methods are chained together in the Client API to simplify how requests are configured and invoked.

Using the Client API in the Jakarta REST Example Applications

The `rsvp` and `customer` examples use the Client API to call Jakarta REST services. This section describes how each example application uses the Client API.

The Client API in the rsvp Example Application

The `rsvp` application allows users to respond to event invitations using Jakarta REST resources, as explained in The rsvp Example Application. The web application uses the Client API in CDI backing beans to interact with the service resources, and the Facelets web interface displays the results.

The `StatusManager` CDI backing bean retrieves all the current events in the system. The client instance used in the backing bean is obtained in the constructor:

```
public StatusManager() {
    this.client = ClientBuilder.newClient();
}
```

The `StatusManager.getEvents` method returns a collection of all the current events in the system by calling the resource at http://localhost:8080/rsvp/webapi/status/all, which returns an XML document with entries for each event. The Client API automatically unmarshals the XML and creates a `List<Event>` instance.

```
public List<Event> getEvents() {
    List<Event> returnedEvents = null;
    try {
        returnedEvents = client.target(baseUri)
                .path("all")
                .request(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Event>>() {
        });
        if (returnedEvents == null) {
            logger.log(Level.SEVERE, "Returned events null.");
        } else {
            logger.log(Level.INFO, "Events have been returned.");
        }
    } catch (WebApplicationException ex) {
        throw new WebApplicationException(Response.Status.NOT_FOUND);
    }
    ...
    return returnedEvents;
}
```

The `StatusManager.changeStatus` method is used to update the attendee's response. It creates an HTTP `POST` request to the service with the new response. The body of the request is an XML document.

```
public String changeStatus(ResponseEnum userResponse,
        Person person, Event event) {
    String navigation;
    try {
        logger.log(Level.INFO,
                "changing status to {0} for {1} {2} for event ID {3}.",
                new Object[]{userResponse,
                    person.getFirstName(),
                    person.getLastName(),
                    event.getId().toString()});
            client.target(baseUri)
                    .path(event.getId().toString())
```

```
                    .path(person.getId().toString())
                    .request(MediaType.APPLICATION_XML)
                    .post(Entity.xml(userResponse.getLabel()));
        navigation = "changedStatus";
    } catch (ResponseProcessingException ex) {
        logger.log(Level.WARNING, "couldn''t change status for {0} {1}",
                new Object[]{person.getFirstName(),
                    person.getLastName()});
        logger.log(Level.WARNING, ex.getMessage());
        navigation = "error";
    }
    return navigation;
}
```

The Client API in the customer Example Application

The `customer` example application stores customer data in a database and exposes the resource as XML, as explained in The customer Example Application. The service resource exposes methods that create customers and retrieve all the customers. A Facelets web application acts as a client for the service resource, with a form for creating customers and displaying the list of customers in a table.

The `CustomerBean` stateless session bean uses the Jakarta REST Client API to interface with the service resource. The `CustomerBean.createCustomer` method takes the `Customer` entity instance created by the Facelets form and makes a POST call to the service URI.

```
public String createCustomer(Customer customer) {
    if (customer == null) {
        logger.log(Level.WARNING, "customer is null.");
        return "customerError";
    }
    String navigation;
    Response response =
            client.target("http://localhost:8080/customer/webapi/Customer")
            .request(MediaType.APPLICATION_XML)
            .post(Entity.entity(customer, MediaType.APPLICATION_XML),
                    Response.class);
    if (response.getStatus() == Status.CREATED.getStatusCode()) {
        navigation = "customerCreated";
    } else {
        logger.log(Level.WARNING,
                "couldn''t create customer with id {0}. Status returned was {1}",
                new Object[]{customer.getId(), response.getStatus()});
        FacesContext context = FacesContext.getCurrentInstance();
        context.addMessage(null,
                new FacesMessage("Could not create customer."));
        navigation = "customerError";
    }
    return navigation;
```

```
}
```

The XML request entity is created by calling the `Invocation.Builder.post` method, passing in a new `Entity` instance from the `Customer` instance, and specifying the media type as `MediaType.APPLICATION_XML`.

The `CustomerBean.retrieveCustomer` method retrieves a `Customer` entity instance from the service by appending the customer's ID to the service URI.

```
public String retrieveCustomer(String id) {
    String navigation;
    Customer customer =
            client.target("http://localhost:8080/customer/webapi/Customer")
            .path(id)
            .request(MediaType.APPLICATION_XML)
            .get(Customer.class);
    if (customer == null) {
        navigation = "customerError";
    } else {
        navigation = "customerRetrieved";
    }
    return navigation;
}
```

The `CustomerBean.retrieveAllCustomers` method retrieves a collection of customers as a `List<Customer>` instance. This list is then displayed as a table in the Facelets web application.

```
public List<Customer> retrieveAllCustomers() {
    List<Customer> customers =
            client.target("http://localhost:8080/customer/webapi/Customer")
            .path("all")
            .request(MediaType.APPLICATION_XML)
            .get(new GenericType<List<Customer>>() {
            });
    return customers;
}
```

Because the response type is a collection, the `Invocation.Builder.get` method is called by passing in a new instance of `GenericType<List<Customer>>`.

Advanced Features of the Client API

This section describes some of the advanced features of the Jakarta REST Client API.

Configuring the Client Request

Additional configuration options may be added to the client request after it is created but before it is invoked.

Setting Message Headers in the Client Request

You can set HTTP headers on the request by calling the `Invocation.Builder.header` method.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .header("myHeader", "The header value")
        .get(String.class);
```

If you need to set multiple headers on the request, call the `Invocation.Builder.headers` method and pass in a `jakarta.ws.rs.core.MultivaluedMap` instance with the name-value pairs of the HTTP headers. Calling the `headers` method replaces all the existing headers with the headers supplied in the `MultivaluedMap` instance.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
MultivaluedMap<String, Object> myHeaders =
    new MultivaluedMap<>("myHeader", "The header value");
myHeaders.add(...);
String response = myResource.request(MediaType.TEXT_PLAIN)
        .headers(myHeaders)
        .get(String.class);
```

The `MultivaluedMap` interface allows you to specify multiple values for a given key.

```
MultivaluedMap<String, Object> myHeaders =
    new MultivaluedMap<String, Object>();
List<String> values = new ArrayList<>();
values.add(...);
myHeaders.add("myHeader", values);
```

Setting Cookies in the Client Request

You can add HTTP cookies to the request by calling the `Invocation.Builder.cookie` method, which takes a name-value pair as parameters.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .cookie("myCookie", "The cookie value")
        .get(String.class);
```

The `jakarta.ws.rs.core.Cookie` class encapsulates the attributes of an HTTP cookie, including the name, value, path, domain, and RFC specification version of the cookie. In the following example, the `Cookie` object is configured with a name-value pair, a path, and a domain.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Cookie myCookie = new Cookie("myCookie", "The cookie value",
    "/webapi/read", "example.com");
String response = myResource.request(MediaType.TEXT_PLAIN)
        .cookie(myCookie)
        .get(String.class);
```

Adding Filters to the Client

You can register custom filters with the client request or the response received from the target resource. To register filter classes when the `Client` instance is created, call the `Client.register` method.

```
Client client = ClientBuilder.newClient().register(MyLoggingFilter.class);
```

In the preceding example, all invocations that use this `Client` instance have the `MyLoggingFilter` filter registered with them.

You can also register the filter classes on the target by calling `WebTarget.register`.

```
Client client = ClientBuilder.newClient().register(MyLoggingFilter.class);
WebTarget target = client.target("http://example.com/webapi/secure")
        .register(MyAuthenticationFilter.class);
```

In the preceding example, both the `MyLoggingFilter` and `MyAuthenticationFilter` filters are attached to the invocation.

Request and response filter classes implement the `jakarta.ws.rs.client.ClientRequestFilter` and `jakarta.ws.rs.client.ClientResponseFilter` interfaces, respectively. Both of these interfaces define a single method, `filter`. All filters must be annotated with `jakarta.ws.rs.ext.Provider`.

The following class is a logging filter for both client requests and client responses.

```
@Provider
public class MyLoggingFilter implements ClientRequestFilter,
        ClientResponseFilter {
    static final Logger logger = Logger.getLogger(...);

    // implement the ClientRequestFilter.filter method
    @Override
    public void filter(ClientRequestContext requestContext)
            throws IOException {
        logger.log(...);
        ...
    }
```

```
    // implement the ClientResponseFilter.filter method
    @Override
    public void filter(ClientRequestContext requestContext,
           ClientResponseContext responseContext) throws IOException {
        logger.log(...);
        ...
    }
}
```

If the invocation must be stopped while the filter is active, call the context object's `abortWith` method, and pass in a `jakarta.ws.rs.core.Response` instance from within the filter.

```
@Override
public void filter(ClientRequestContext requestContext) throws IOException {
    ...
    Response response = new Response();
    response.status(500);
    requestContext.abortWith(response);
}
```

Asynchronous Invocations in the Client API

In networked applications, network issues can affect the perceived performance of the application, particularly in long-running or complicated network calls. Asynchronous processing helps prevent blocking and makes better use of an application's resources.

In the Jakarta REST Client API, the `Invocation.Builder.async` method is used when constructing a client request to indicate that the call to the service should be performed asynchronously. An asynchronous invocation returns control to the caller immediately, with a return type of `java.util.concurrent.Future<T>` (part of the Java SE concurrency API) and with the type set to the return type of the service call. `Future<T>` objects have methods to check if the asynchronous call has been completed, to retrieve the final result, to cancel the invocation, and to check if the invocation has been cancelled.

The following example shows how to invoke an asynchronous request on a resource.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Future<String> response = myResource.request(MediaType.TEXT_PLAIN)
        .async()
        .get(String.class);
```

Using Custom Callbacks in Asynchronous Invocations

The `InvocationCallback` interface defines two methods, `completed` and `failed`, that are called when an asynchronous invocation either completes successfully or fails, respectively. You may register an `InvocationCallback` instance on your request by creating a new instance when specifying the request method.

The following example shows how to register a callback object on an asynchronous invocation.

```
Client client = ClientBuilder.newClient();
WebTarget myResource = client.target("http://example.com/webapi/read");
Future<Customer> fCustomer = myResource.request(MediaType.TEXT_PLAIN)
        .async()
        .get(new InvocationCallback<Customer>() {
            @Override
            public void completed(Customer customer) {
            // Do something with the customer object
            }
            @Override
             public void failed(Throwable throwable) {
            // handle the error
            }
        });
```

Using Reactive Approach in Asynchronous Invocations

Using custom callbacks in asynchronous invocations is easy in simple cases and when there are many independent calls to make. In nested calls, using custom callbacks becomes very difficult to implement, debug, and maintain.

Jakarta REST defines a new type of invoker called as `RxInvoker` and a default implementation of this type is `CompletionStageRxInvoker`. The new `rx` method is used as in the following example:

```
CompletionStage<String> csf = client.target("forecast/{destination}")
    .resolveTemplate("destination","mars")
    .request().rx().get(String.class);
csf.thenAccept(System.out::println);
```

In the example, an asynchronous processing of the interface `CompletionStage<String>` is created and waits till it is completed and the result is displayed. The `CompletionStage` that is returned can then be used only to retrieve the result as shown in the above example or can be combined with other completion stages to ease and improve the processing of asynchronous tasks.

Using Server-Sent Events

Server-sent Events (SSE) technology is used to asynchronously push notifications to the client over standard HTTP or HTTPS protocol. Clients can subscribe to event notifications that originate on a server. Server generates events and sends these events back to the clients that are subscribed to receive the notifications. The one-way communication channel connection is established by the client. Once the connection is established, the server sends events to the client whenever new data is available.

The communication channel established by the client lasts till the client closes the connection and it is also re-used by the server to send multiple events from the server.

Overview of the SSE API

The SSE API is defined in the `jakarta.ws.rs.sse` package that includes the interfaces `SseEventSink`, `SseEvent`, `Sse`, and `SseEventSource`. To accept connections and send events to one or more clients, inject an `SseEventSink` in the resource method that produces the media type `text/event-stream`.

The following example shows how to accept the SSE connections and to send events to the clients:

```
@GET
@Path("eventStream")
@Produces(MediaType.SERVER_SENT_EVENTS)
public void eventStream(@Context SseEventSink eventSink, @Context Sse sse) {
    executor.execute(() -> {
        try (SseEventSink sink = eventSink) {
            eventSink.send(sse.newEvent("event1"));
            eventSink.send(sse.newEvent("event2"));
            eventSink.send(sse.newEvent("event3"));
        }
    });
}
```

The `SseEventsink` is injected into the resource method and the underlying client connection is kept open and used to send events. The connection persists until the client disconnects from the server. The method `send` returns an instance of `CompletionStage<T>` which indicates the action of asynchronously sending a message to a client is enabled.

The events that are streamed to the clients can be defined with the details such as `event`, `data`, `id`, `retry`, and `comment`.

Broadcasting Using SSE

Broadcasting is the action of sending events to multiple clients simultaneously. Jakarta REST SSE API provides `SseBroadcaster` to register all `SseEventSink` instances and send events to all registered event outputs. The life-cycle and scope of an `SseBroadcaster` is fully controlled by applications and not the Jakarta REST runtime. The following example show the use of broadcasters:

```
@Path("/")
@Singleton
public class SseResource {
    @Context
    private Sse sse;

    private volatile SseBroadcaster sseBroadcaster;

    @PostConstruct
    public init() {
        this.sseBroadcaster = sse.newBroadcaster();
    }
```

```
    @GET
    @Path("register")
    @Produces(MediaType.SERVER_SENT_EVENTS)
        public void register(@Context SseEventSink eventSink) {
        eventSink.send(sse.newEvent("welcome!"));
        sseBroadcaster.register(eventSink);
    }

    @POST
    @Path("broadcast")
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    public void broadcast(@FormParam("event") String event) {
        sseBroadcaster.broadcast(sse.newEvent(event));
    }
}
```

`@Singleton` annotation is defined for the resource class restricting the creation of multiple instances of the class. The `register` method on a broadcaster is used to add a new `SseEventSink`; the `broadcast` method is used to send an SSE event to all registered clients.

Listening and Receiving Events

Jakarta REST SSE provides the `SseEventSource` interface for the client to subscribe to notifications. The client can get asynchronously notified about incoming events by invoking one of the `subscribe` methods in `jakarta.ws.rs.sse.SseEventSource`.

The following example shows how to use the `SseEventSource` API to open an SSE connection and read some of the messages for a period:

```
WebTarget target = client.target("http://...");
try (SseEventSource source = SseEventSource.target(target).build()) {
    source.register(System.out::println);
    source.open();
    Thread.sleep(500); // Consume events for just 500 ms
    source.close();
} catch (InterruptedException e) {
    // falls through
}
```

Jakarta REST: Advanced Topics and an Example

Jakarta RESTful Web Services (Jakarta REST) is designed to make it easy to develop applications that use the REST architecture. This chapter describes advanced features of Jakarta REST. If you are new to Jakarta REST, see [websvcs:rest::rest:::_building_restful_web_services_with_jakarta_rest] before you proceed with this chapter.

Jakarta REST is integrated with Jakarta Contexts and Dependency Injection (CDI), Jakarta Enterprise Beans technology, and Jakarta Servlet technology.

Annotations for Field and Bean Properties of Resource Classes

Jakarta REST annotations for resource classes let you extract specific parts or values from a Uniform Resource Identifier (URI) or request header.

Jakarta REST provides the annotations listed in Advanced Jakarta REST Annotations.

Advanced Jakarta REST Annotations

Annotation	Description
`@Context`	Injects information into a class field, bean property, or method parameter
`@CookieParam`	Extracts information from cookies declared in the cookie request header
`@FormParam`	Extracts information from a request representation whose content type is `application/x-www-form-urlencoded`
`@HeaderParam`	Extracts the value of a header
`@MatrixParam`	Extracts the value of a URI matrix parameter
`@PathParam`	Extracts the value of a URI template parameter
`@QueryParam`	Extracts the value of a URI query parameter

Extracting Path Parameters

URI path templates are URIs with variables embedded within the URI syntax. The `@PathParam` annotation lets you use variable URI path fragments when you call a method.

The following code snippet shows how to extract the last name of an employee when the employee's email address is provided:

```
@Path("/employees/{firstname}.{lastname}@{domain}.com")
public class EmpResource {
```

```
    @GET
    @Produces("text/xml")
    public String getEmployeelastname(@PathParam("lastname") String lastName) {
      ...
    }
}
```

In this example, the `@Path` annotation defines the URI variables (or path parameters) `{firstname}`, `{lastname}`, and `{domain}`. The `@PathParam` in the method parameter of the request method extracts the last name from the email address.

If your HTTP request is `GET /employees/john.doe@example.com`, the value "doe" is injected into `{lastname}`.

You can specify several path parameters in one URI.

You can declare a regular expression with a URI variable. For example, if it is required that the last name must consist only of lowercase and uppercase characters, you can declare the following regular expression:

```
@Path("/employees/{firstname}.{lastname[a-zA-Z]*}@{domain}.com")
```

If the last name does not match the regular expression, a 404 response is returned.

Extracting Query Parameters

Use the `@QueryParam` annotation to extract query parameters from the query component of the request URI.

For instance, to query all employees who have joined within a specific range of years, use a method signature like the following:

```
@Path("/employees/")
@GET
public Response getEmployees(
        @DefaultValue("2003") @QueryParam("minyear") int minyear,
        @DefaultValue("2013") @QueryParam("maxyear") int maxyear)
    {...}
```

This code snippet defines two query parameters, `minyear` and `maxyear`. The following HTTP request would query for all employees who have joined between 2003 and 2013:

```
GET /employees?maxyear=2013&minyear=2003
```

The `@DefaultValue` annotation defines a default value, which is to be used if no values are provided for the query parameters. By default, Jakarta REST assigns a null value for `Object` values and zero for primitive data types. You can use the `@DefaultValue` annotation to eliminate null or zero values

and define your own default values for a parameter.

Extracting Form Data

Use the `@FormParam` annotation to extract form parameters from HTML forms. For example, the following form accepts the name, address, and manager's name of an employee:

```
<FORM action="http://example.com/employees/" method="post">
  <p>
    <fieldset>
      Employee name: <INPUT type="text" name="empname" tabindex="1">
      Employee address: <INPUT type="text" name="empaddress" tabindex="2">
      Manager name: <INPUT type="text" name="managername" tabindex="3">
    </fieldset>
  </p>
</FORM>
```

Use the following code snippet to extract the manager name from this HTML form:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("managername") String managername) {
    // Store the value
    ...
}
```

To obtain a map of form parameter names to values, use a code snippet like the following:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```

Extracting the Java Type of a Request or Response

The `jakarta.ws.rs.core.Context` annotation retrieves the Java types related to a request or response.

The `jakarta.ws.rs.core.UriInfo` interface provides information about the components of a request URI. The following code snippet shows how to obtain a map of query and path parameter names to values:

```
@GET
public String getParams(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
```

```
}
```

The `jakarta.ws.rs.core.HttpHeaders` interface provides information about request headers and cookies. The following code snippet shows how to obtain a map of header and cookie parameter names to values:

```
@GET
public String getHeaders(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    MultivaluedMap<String, Cookie> pathParams = hh.getCookies();
}
```

Validating Resource Data with Bean Validation

Jakarta REST supports Bean Validation to verify Jakarta REST resource classes. This support consists of:

- Adding constraint annotations to resource method parameters
- Ensuring entity data is valid when the entity is passed in as a parameter

Using Constraint Annotations on Resource Methods

Bean Validation constraint annotations may be applied to parameters for a resource. The server will validate the parameters and either pass or throw a `jakarta.validation.ValidationException`.

```
@POST
@Path("/createUser")
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public void createUser(@NotNull @FormParam("username") String username,
                       @NotNull @FormParam("firstName") String firstName,
                       @NotNull @FormParam("lastName") String lastName,
                       @Email @FormParam("email") String email) {
    ...
}
```

In the preceding example, the built-in constraint `@NotNull` is applied to the `username`, `firstName`, and `lastName` form fields. Another built-in constraint `@Email` validates that the email address supplied by the `email` form field is correctly formatted.

The constraints may also be applied to fields within a resource class.

```
@Path("/createUser")
public class CreateUserResource {
  @NotNull
  @FormParam("username")
  private String username;
```

```
  @NotNull
  @FormParam("firstName")
  private String firstName;

  @NotNull
  @FormParam("lastName")
  private String lastName;

  @Email
  @FormParam("email")
  private String email;

  ...
}
```

In the preceding example, the same constraints that were applied to the method parameters in the previous example are applied to the class fields. The behavior is the same in both examples.

Constraints may also be applied to a resource class's JavaBeans properties by adding the constraint annotations to the getter method.

```
@Path("/createuser")
public class CreateUserResource {
  private String username;

  @FormParam("username")
  public void setUsername(String username) {
    this.username = username;
  }

  @NotNull
  public String getUsername() {
    return username;
  }
  ...
}
```

Constraints may also be applied at the resource class level. In the following example, `@PhoneRequired` is a user-defined constraint that ensures that a user enters at least one phone number. That is, either `homePhone` or `mobilePhone` can be null, but not both.

```
@Path("/createUser")
@PhoneRequired
public class CreateUserResource {
  @FormParam("homePhone")
  private Phone homePhone;

  @FormParam("mobilePhone")
```

```
  private Phone mobilePhone;
  ...
}
```

Validating Entity Data

Classes that contain validation constraint annotations may be used in method parameters in a resource class. To validate these entity classes, use the `@Valid` annotation on the method parameter. For example, the following class is a user-defined class containing both standard and user-defined validation constraints.

```
@PhoneRequired
public class User {
  @NotNull
  private String username;

  private Phone homePhone;

  private Phone mobilePhone;
  ...
}
```

This entity class is used as a parameter to a resource method.

```
@Path("/createUser")
public class CreateUserResource {
  ...
  @POST
  @Consumers(MediaType.APPLICATION_XML)
  public void createUser(@Valid User user) {
    ...
  }
  ...
}
```

The `@Valid` annotation ensures that the entity class is validated at runtime. Additional user-defined constraints can also trigger validation of an entity.

```
@Path("/createUser")
public class CreateUserResource {
  ...
  @POST
  @Consumers(MediaType.APPLICATION_XML)
  public void createUser(@ActiveUser User user) {
    ...
  }
  ...
```

```
}
```

In the preceding example, the user-defined `@ActiveUser` constraint is applied to the `User` class in addition to the `@PhoneRequired` and `@NotNull` constraints defined within the entity class.

If a resource method returns an entity class, validation may be triggered by applying the `@Valid` or any other user-defined constraint annotation to the resource method.

```
@Path("/getUser")
public class GetUserResource {
  ...
  @GET
  @Path("{username}")
  @Produces(MediaType.APPLICATION_XML)
  @ActiveUser
  @Valid
  public User getUser(@PathParam("username") String username) {
    // find the User
    return user;
  }
  ...
}
```

As in the previous example, the `@ActiveUser` constraint is applied to the returned entity class as well as the `@PhoneRequired` and `@NotNull` constraints defined within the entity class.

Validation Exception Handling and Response Codes

If a `jakarta.validation.ValidationException` or any subclass of `ValidationException` except `ConstraintValidationException` is thrown, the Jakarta REST runtime will respond to the client request with a 500 (Internal Server Error) HTTP status code.

If a `ConstraintValidationException` is thrown, the Jakarta REST runtime will respond to the client with one of the following HTTP status codes:

- `500` (Internal Server Error) if the exception was thrown while validating a method return type
- `400` (Bad Request) in all other cases

Subresources and Runtime Resource Resolution

You can use a resource class to process only a part of the URI request. A root resource can then implement subresources that can process the remainder of the URI path.

A resource class method that is annotated with `@Path` is either a subresource method or a subresource locator.

- A subresource method is used to handle requests on a subresource of the corresponding resource.

- A subresource locator is used to locate subresources of the corresponding resource.

Subresource Methods

A subresource method handles an HTTP request directly. The method must be annotated with a request method designator, such as `@GET` or `@POST`, in addition to `@Path`. The method is invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method.

The following code snippet shows how a subresource method can be used to extract the last name of an employee when the employee's email address is provided:

```
@Path("/employeeinfo")
public class EmployeeInfo {

    public employeeinfo() {}

    @GET
    @Path("/employees/{firstname}.{lastname}@{domain}.com")
    @Produces("text/xml")
    public String getEmployeeLastName(@PathParam("lastname") String lastName) {
       ...
    }
}
```

The `getEmployeeLastName` method returns `doe` for the following `GET` request:

```
GET /employeeinfo/employees/john.doe@example.com
```

Subresource Locators

A subresource locator returns an object that will handle an HTTP request. The method must not be annotated with a request method designator. You must declare a subresource locator within a subresource class, and only subresource locators are used for runtime resource resolution.

The following code snippet shows a subresource locator:

```
// Root resource class
@Path("/employeeinfo")
public class EmployeeInfo {

    // Subresource locator: obtains the subresource Employee
    // from the path /employeeinfo/employees/{empid}
    @Path("/employees/{empid}")
    public Employee getEmployee(@PathParam("empid") String id) {
        // Find the Employee based on the id path parameter
        Employee emp = ...;
        ...
```

```
        return emp;
    }
}

// Subresource class
public class Employee {

    // Subresource method: returns the employee's last name
    @GET
    @Path("/lastname")
    public String getEmployeeLastName() {
        ...
        return lastName;
    }
}
```

In this code snippet, the `getEmployee` method is the subresource locator that provides the `Employee` object, which services requests for `lastname`.

If your HTTP request is `GET /employeeinfo/employees/as209/`, the `getEmployee` method returns an `Employee` object whose id is `as209`. At runtime, Jakarta REST sends a `GET /employeeinfo/employees/as209/lastname` request to the `getEmployeeLastName` method. The `getEmployeeLastName` method retrieves and returns the last name of the employee whose id is `as209`.

Integrating Jakarta REST with Jakarta Enterprise Beans Technology and CDI

Jakarta REST works with Jakarta Enterprise Beans technology and Jakarta Contexts and Dependency Injection (CDI).

In general, for Jakarta REST to work with enterprise beans, you need to annotate the class of a bean with `@Path` to convert it to a root resource class. You can use the `@Path` annotation with stateless session beans and singleton POJO beans.

The following code snippet shows a stateless session bean and a singleton bean that have been converted to Jakarta REST root resource classes.

```
@Stateless
@Path("stateless-bean")
public class StatelessResource {...}

@Singleton
@Path("singleton-bean")
public class SingletonResource {...}
```

Session beans can also be used for subresources.

Jakarta REST and CDI have slightly different component models. By default, Jakarta REST root resource classes are managed in the request scope, and no annotations are required for specifying the scope. CDI managed beans annotated with `@RequestScoped` or `@ApplicationScoped` can be

converted to Jakarta REST resource classes.

The following code snippet shows a Jakarta REST resource class.

```
@Path("/employee/{id}")
public class Employee {
    public Employee(@PathParam("id") String id) {...}
}

@Path("{lastname}")
public final class EmpDetails {...}
```

The following code snippet shows this Jakarta REST resource class converted to a CDI bean. The beans must be proxyable, so the `Employee` class requires a nonprivate constructor with no parameters, and the `EmpDetails` class must not be `final`.

```
@Path("/employee/{id}")
@RequestScoped
public class Employee {
    public Employee() {...}

    @Inject
    public Employee(@PathParam("id") String id) {...}
}

@Path("{lastname}")
@RequestScoped
public class EmpDetails {...}
```

Conditional HTTP Requests

Jakarta REST provides support for conditional `GET` and `PUT` HTTP requests. Conditional `GET` requests help save bandwidth by improving the efficiency of client processing.

A `GET` request can return a Not Modified (304) response if the representation has not changed since the previous request. For example, a website can return 304 responses for all its static images that have not changed since the previous request.

A `PUT` request can return a Precondition Failed (412) response if the representation has been modified since the last request. The conditional `PUT` can help avoid the lost update problem.

Conditional HTTP requests can be used with the `Last-Modified` and `ETag` headers. The `Last-Modified` header can represent dates with granularity of one second.

```
@Path("/employee/{joiningdate}")
public class Employee {

    Date joiningdate;
```

```
    @GET
    @Produces("application/xml")
    public Employee(@PathParam("joiningdate") Date joiningdate,
                    @Context Request req,
                    @Context UriInfo ui) {

        this.joiningdate = joiningdate;
        ...
        this.tag = computeEntityTag(ui.getRequestUri());
        if (req.getMethod().equals("GET")) {
            Response.ResponseBuilder rb = req.evaluatePreconditions(tag);
            if (rb != null) {
                throw new WebApplicationException(rb.build());
            }
        }
    }
}
```

In this code snippet, the constructor of the `Employee` class computes the entity tag from the request URI and calls the `request.evaluatePreconditions` method with that tag. If a client request returns an `If-none-match` header with a value that has the same entity tag that was computed, `evaluate.Preconditions` returns a pre-filled-out response with a 304 status code and an entity tag set that may be built and returned.

Runtime Content Negotiation

The `@Produces` and `@Consumes` annotations handle static content negotiation in Jakarta REST. These annotations specify the content preferences of the server. HTTP headers such as `Accept`, `Content-Type`, and `Accept-Language` define the content negotiation preferences of the client.

For more details on the HTTP headers for content negotiation, see HTTP/1.1 - Content Negotiation (https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html).

The following code snippet shows the server content preferences:

```
@Produces("text/plain")
@Path("/employee")
public class Employee {

    @GET
    public String getEmployeeAddressText(String address) {...}

    @Produces("text/xml")
    @GET
    public String getEmployeeAddressXml(Address address) {...}
}
```

The `getEmployeeAddressText` method is called for an HTTP request that looks like the following:

```
GET /employee
Accept: text/plain
```

This will produce the following response:

```
500 Oracle Parkway, Redwood Shores, CA
```

The `getEmployeeAddressXml` method is called for an HTTP request that looks like the following:

```
GET /employee
Accept: text/xml
```

This will produce the following response:

```
<address street="500 Oracle Parkway, Redwood Shores, CA" country="USA"/>
```

With static content negotiation, you can also define multiple content and media types for the client and server.

```
@Produces("text/plain", "text/xml")
```

In addition to supporting static content negotiation, Jakarta REST also supports runtime content negotiation using the `jakarta.ws.rs.core.Variant` class and `Request` objects. The `Variant` class specifies the resource representation of content negotiation. Each instance of the `Variant` class may contain a media type, a language, and an encoding. The `Variant` object defines the resource representation that is supported by the server. The `Variant.VariantListBuilder` class is used to build a list of representation variants.

The following code snippet shows how to create a list of resource representation variants:

```
List<Variant> vs = Variant.mediatypes("application/xml", "application/json")
        .languages("en", "fr").build();
```

This code snippet calls the `build` method of the `VariantListBuilder` class. The `VariantListBuilder` class is invoked when you call the `mediatypes`, `languages`, or `encodings` methods. The `build` method builds a series of resource representations. The `Variant` list created by the `build` method has all possible combinations of items specified in the `mediatypes`, `languages`, and `encodings` methods.

In this example, the size of the `vs` object as defined in this code snippet is 4, and the contents are as follows:

```
[["application/xml","en"], ["application/json","en"],
```

```
    ["application/xml","fr"],["application/json","fr"]]
```

The `jakarta.ws.rs.core.Request.selectVariant` method accepts a list of `Variant` objects and chooses the `Variant` object that matches the HTTP request. This method compares its list of `Variant` objects with the `Accept`, `Accept-Encoding`, `Accept-Language`, and `Accept-Charset` headers of the HTTP request.

The following code snippet shows how to use the `selectVariant` method to select the most acceptable `Variant` from the values in the client request:

```
@GET
public Response get(@Context Request r) {
    List<Variant> vs = ...;
    Variant v = r.selectVariant(vs);
    if (v == null) {
        return Response.notAcceptable(vs).build();
    } else {
        Object rep = selectRepresentation(v);
        return Response.ok(rep, v);
    }
}
```

The `selectVariant` method returns the `Variant` object that matches the request or null if no matches are found. In this code snippet, if the method returns null, a `Response` object for a nonacceptable response is built. Otherwise, a `Response` object with an OK status and containing a representation in the form of an `Object` entity and a `Variant` is returned.

Using Jakarta REST with Jakarta XML Binding

Jakarta XML Binding is an XML-to-Java binding technology that simplifies the development of web services by enabling transformations between schema and Java objects and between XML instance documents and Java object instances. An XML schema defines the data elements and structure of an XML document. You can use Jakarta XML Binding APIs and tools to establish mappings between Java classes and XML schema. Jakarta XML Binding technology provides the tools that enable you to convert your XML documents to and from Java objects.

By using Jakarta XML Binding, you can manipulate data objects in the following ways.

- You can start with an XML schema definition (XSD) and use `xjc`, the Jakarta XML Binding schema compiler tool, to create a set of Jakarta XML Binding annotated Java classes that map to the elements and types defined in the XSD schema.
- You can start with a set of Java classes and use `schemagen`, the Jakarta XML Binding schema generator tool, to generate an XML schema.
- Once a mapping between the XML schema and the Java classes exists, you can use the Jakarta XML Binding runtime to marshal and unmarshal your XML documents to and from Java objects and use the resulting Java classes to assemble a web services application.

XML is a common media format that RESTful services consume and produce. To deserialize and serialize XML, you can represent requests and responses by Jakarta XML Binding annotated objects.

Your Jakarta REST application can use the Jakarta XML Binding objects to manipulate XML data. Jakarta XML Binding objects can be used as request entity parameters and response entities. The Jakarta REST runtime environment includes standard `MessageBodyReader` and `MessageBodyWriter` provider interfaces for reading and writing Jakarta XML Binding objects as entities.

With Jakarta REST, you enable access to your services by publishing resources. Resources are just simple Java classes with some additional Jakarta REST annotations. These annotations express the following:

- The path of the resource (the URL you use to access it)
- The HTTP method you use to call a certain method (for example, the `GET` or `POST` method)
- The MIME type with which a method accepts or responds

As you define the resources for your application, consider the type of data you want to expose. You may already have a relational database that contains information you want to expose to users, or you may have static content that does not reside in a database but does need to be distributed as resources. Using Jakarta REST, you can distribute content from multiple sources. RESTful web services can use various types of input/output formats for request and response. The `customer` example, described in The customer Example Application, uses XML.

Resources have representations. A resource representation is the content in the HTTP message that is sent to, or returned from, the resource using the URI. Each representation a resource supports has a corresponding media type. For example, if a resource is going to return content formatted as XML, you can use `application/xml` as the associated media type in the HTTP message.Depending on the requirements of your application, resources can return representations in a preferred single format or in multiple formats. Jakarta REST provides `@Consumes` and `@Produces` annotations to declare the media types that are acceptable for a resource method to read and write.

Jakarta REST also maps Java types to and from resource representations using entity providers. A `MessageBodyReader` entity provider reads a request entity and deserializes the request entity into a Java type. A `MessageBodyWriter` entity provider serializes from a Java type into a response entity. For example, if a `String` value is used as the request entity parameter, the `MessageBodyReader` entity provider deserializes the request body into a new `String`. If a Jakarta XML Binding type is used as the return type on a resource method, the `MessageBodyWriter` serializes the Jakarta XML Binding object into a response body.

By default, the Jakarta REST runtime environment attempts to create and use a default `JAXBContext` class for Jakarta XML Binding classes. However, if the default `JAXBContext` class is not suitable, then you can supply a `JAXBContext` class for the application using a Jakarta REST `ContextResolver` provider interface.

The following sections explain how to use Jakarta XML Binding with Jakarta REST resource methods.

Using Java Objects to Model Your Data

If you do not have an XML schema definition for the data you want to expose, you can model your data as Java classes, add Jakarta XML Binding annotations to these classes, and use Jakarta XML Binding to generate an XML schema for your data. For example, if the data you want to expose is a

collection of products and each product has an ID, a name, a description, and a price, you can model it as a Java class as follows:

```
@XmlRootElement(name="product")
@XmlAccessorType(XmlAccessType.FIELD)
public class Product {

    @XmlElement(required=true)
    protected int id;
    @XmlElement(required=true)
    protected String name;
    @XmlElement(required=true)
    protected String description;
    @XmlElement(required=true)
    protected int price;

    public Product() {}

    // Getter and setter methods
    // ...
}
```

Run the Jakarta XML Binding schema generator on the command line to generate the corresponding XML schema definition:

```
schemagen Product.java
```

This command produces the XML schema as an `.xsd` file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="product" type="product"/>

    <xs:complexType name="product">
      <xs:sequence>
        <xs:element name="id" type="xs:int"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="price" type="xs:int"/>
      </xs:sequence>
    <xs:complexType>
</xs:schema>
```

Once you have this mapping, you can create `Product` objects in your application, return them, and use them as parameters in Jakarta REST resource methods. The Jakarta REST runtime uses Jakarta XML Binding to convert the XML data from the request into a `Product` object and to convert a

`Product` object into XML data for the response. The following resource class provides a simple example:

```
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public Product getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return prod;
    }

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Process or store the product and return a response
        // ...
    }
}
```

Some IDEs, such as NetBeans IDE, will run the schema generator tool automatically during the build process if you add Java classes that have Jakarta XML Binding annotations to your project. For a detailed example, see The customer Example Application. The `customer` example contains a more complex relationship between the Java classes that model the data, which results in a more hierarchical XML representation.

Starting from an Existing XML Schema Definition

If you already have an XML schema definition in an `.xsd` file for the data you want to expose, use the Jakarta XML Binding schema compiler tool. Consider this simple example of an `.xsd` file:

```
<xs:schema targetNamespace="http://xml.product"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           xmlns:myco="http://xml.product">
  <xs:element name="product" type="myco:Product"/>
  <xs:complexType name="Product">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:int"/>
    </xs:sequence>
```

```
  </xs:complexType>
</xs:schema>
```

Run the schema compiler tool on the command line as follows:

```
xjc Product.xsd
```

This command generates the source code for Java classes that correspond to the types defined in the `.xsd` file. The schema compiler tool generates a Java class for each `complexType` defined in the `.xsd` file. The fields of each generated Java class are the same as the elements inside the corresponding `complexType`, and the class contains getter and setter methods for these fields.

In this case, the schema compiler tool generates the classes `product.xml.Product` and `product.xml.ObjectFactory`. The `Product` class contains Jakarta XML Binding annotations, and its fields correspond to those in the `.xsd` definition:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Product", propOrder = {
    "id",
    "name",
    "description",
    "price"
})
public class Product {
    protected int id;
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected String description;
    protected int price;

    // Setter and getter methods
    // ...
}
```

You can create instances of the `Product` class from your application (for example, from a database). The generated class `product.xml.ObjectFactory` contains a method that allows you to convert these objects to Jakarta XML Binding elements that can be returned as XML inside Jakarta REST resource methods:

```
@XmlElementDecl(namespace = "http://xml.product", name = "product")
public JAXBElement<Product> createProduct(Product value) {
    return new JAXBElement<Product>(_Product_QNAME, Product.class, null, value);
}
```

The following code shows how to use the generated classes to return a Jakarta XML Binding

element as XML in a Jakarta REST resource method:

```
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public JAXBElement<Product> getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return new ObjectFactory().createProduct(prod);
    }
}
```

For `@POST` and `@PUT` resource methods, you can use a `Product` object directly as a parameter. Jakarta REST maps the XML data from the request into a `Product` object.

```
@Path("/product")
public class ProductService {
    @GET
    // ...

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Process or store the product and return a response
        // ...
    }
}
```

Using JSON with Jakarta REST and Jakarta XML Binding

Jakarta REST can automatically read and write XML using Jakarta XML Binding, but it can also work with JSON data. JSON is a simple text-based format for data exchange derived from JavaScript. For the preceding examples, the XML representation of a product is

```
<?xml version="1.0" encoding="UTF-8"?>
<product>
  <id>1</id>
  <name>Mattress</name>
  <description>Queen size mattress</description>
  <price>500</price>
</product>
```

The equivalent JSON representation is

```
{
    "id":"1",
    "name":"Mattress",
    "description":"Queen size mattress",
    "price":500
}
```

You can add the format `application/json` or `MediaType.APPLICATION_JSON` to the `@Produces` annotation in resource methods to produce responses with JSON data:

```
@GET
@Path("/get")
@Produces({"application/xml","application/json"})
public Product getProduct() { ... }
```

In this example, the default response is XML, but the response is a JSON object if the client makes a `GET` request that includes this header:

```
Accept: application/json
```

The resource methods can also accept JSON data for Jakarta XML Binding annotated classes:

```
@POST
@Path("/create")
@Consumes({"application/xml","application/json"})
public Response createProduct(Product prod) { ... }
```

The client should include the following header when submitting JSON data with a `POST` request:

```
Content-Type: application/json
```

The customer Example Application

This section describes how to build and run the `customer` example application. This application is a RESTful web service that uses Jakarta XML Binding to perform the create, read, update, delete (CRUD) operations for a specific entity.

The `customer` sample application is in the `jakartaee-examples/tutorial/rest/customer/` directory. See [intro:usingexamples::usingexamples:::_using_the_tutorial_examples], for basic information on building and running sample applications.

Overview of the customer Example Application

The source files of this application are at `jakartaee-examples/tutorial/rest/customer/src/main/java/`. The application has three parts.

- The `Customer` and `Address` entity classes. These classes model the data of the application and contain Jakarta XML Binding annotations.
- The `CustomerService` resource class. This class contains Jakarta REST resource methods that perform operations on `Customer` instances represented as XML or JSON data using Jakarta XML Binding. See The CustomerService Class for details.
- The `CustomerBean` session bean that acts as a backing bean for the web client. `CustomerBean` uses the Jakarta REST client API to call the methods of `CustomerService`.

The `customer` example application shows you how to model your data entities as Java classes with Jakarta XML Binding annotations.

The Customer and Address Entity Classes

The following class represents a customer's address:

```
@Entity
@Table(name="CUSTOMER_ADDRESS")
@XmlRootElement(name="address")
@XmlAccessorType(XmlAccessType.FIELD)
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @XmlElement(required=true)
    protected int number;

    @XmlElement(required=true)
    protected String street;

    @XmlElement(required=true)
    protected String city;

    @XmlElement(required=true)
    protected String province;

    @XmlElement(required=true)
    protected String zip;

    @XmlElement(required=true)
    protected String country;

    public Address() { }

    // Getter and setter methods
```

```
    // ...
}
```

The `@XmlRootElement(name="address")` annotation maps this class to the `address` XML element. The `@XmlAccessorType(XmlAccessType.FIELD)` annotation specifies that all the fields of this class are bound to XML by default. The `@XmlElement(required=true)` annotation specifies that an element must be present in the XML representation.

The following class represents a customer:

```
@Entity
@Table(name="CUSTOMER_CUSTOMER")
@NamedQuery(
    name="findAllCustomers",
    query="SELECT c FROM Customer c " +
          "ORDER BY c.id"
)
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @XmlAttribute(required=true)
    protected int id;

    @XmlElement(required=true)
    protected String firstname;

    @XmlElement(required=true)
    protected String lastname;

    @XmlElement(required=true)
    @OneToOne
    protected Address address;

    @XmlElement(required=true)
    protected String email;

    @XmlElement (required=true)
    protected String phone;

    public Customer() {...}

    // Getter and setter methods
    // ...
}
```

The `Customer` class contains the same Jakarta XML Binding annotations as the previous class, except for the `@XmlAttribute(required=true)` annotation, which maps a property to an attribute of the XML

element representing the class.

The `Customer` class contains a property whose type is another entity, the `Address` class. This mechanism allows you to define in Java code the hierarchical relationships between entities without having to write an `.xsd` file yourself.

Jakarta XML Binding generates the following XML schema definition for the two preceding classes:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="address" type="address"/>
  <xs:element name="customer" type="customer"/>

  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="id" type="xs:long" minOccurs="0"/>
      <xs:element name="number" type="xs:int"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="province" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element ref="address"/>
      <xs:element name="email" type="xs:string"/>
      <xs:element name="phone" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>
</xs:schema>
```

The CustomerService Class

The `CustomerService` class has a `createCustomer` method that creates a customer resource based on the `Customer` class and returns a URI for the new resource.

```
@Stateless
@Path("/Customer")
public class CustomerService {
    public static final Logger logger =
            Logger.getLogger(CustomerService.class.getCanonicalName());
    @PersistenceContext
```

```
    private EntityManager em;
    private CriteriaBuilder cb;

    @PostConstruct
    private void init() {
        cb = em.getCriteriaBuilder();
    }
    ...
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response createCustomer(Customer customer) {

        try {
            long customerId = persist(customer);
            return Response.created(URI.create("/" + customerId)).build();
        } catch (Exception e) {
            logger.log(Level.SEVERE,
                    "Error creating customer for customerId {0}. {1}",
                    new Object[]{customer.getId(), e.getMessage()});
            throw new WebApplicationException(e,
                    Response.Status.INTERNAL_SERVER_ERROR);
        }
    }
    ...
    private long persist(Customer customer) {
        try {
            Address address = customer.getAddress();
            em.persist(address);
            em.persist(customer);
        } catch (Exception ex) {
            logger.warning("Something went wrong when persisting the customer");
        }
        return customer.getId();
    }
}
```

The response returned to the client has a URI to the newly created resource. The return type is an entity body mapped from the property of the response with the status code specified by the status property of the response. The `WebApplicationException` is a `RuntimeException` that is used to wrap the appropriate HTTP error status code, such as 404, 406, 415, or 500.

The `@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})` and `@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})` annotations set the request and response media types to use the appropriate MIME client. These annotations can be applied to a resource method, a resource class, or even an entity provider. If you do not use these annotations, Jakarta REST allows the use of any media type (`"*/*"`).

The following code snippet shows the implementation of the `getCustomer` and `findbyId` methods. The `getCustomer` method uses the `@Produces` annotation and returns a `Customer` object, which is converted to an XML or JSON representation depending on the `Accept:` header specified by the client.

```
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Customer getCustomer(@PathParam("id") String customerId) {
        Customer customer = null;

        try {
            customer = findById(customerId);
        } catch (Exception ex) {
            logger.log(Level.SEVERE,
                    "Error calling findCustomer() for customerId {0}. {1}",
                    new Object[]{customerId, ex.getMessage()});
        }
        return customer;
    }
    ...
    private Customer findById(String customerId) {
        Customer customer = null;
        try {
            customer = em.find(Customer.class, customerId);
            return customer;
        } catch (Exception ex) {
            logger.log(Level.WARNING,
                    "Couldn't find customer with ID of {0}", customerId);
        }
        return customer;
    }
```

Using the Jakarta REST Client in the CustomerBean Classes

Use the Jakarta REST Client API to write a client for the `customer` example application.

The `CustomerBean` enterprise bean class calls the Jakarta REST Client API to test the `CustomerService` web service:

```
@Named
@Stateless
public class CustomerBean {
    protected Client client;
    private static final Logger logger =
            Logger.getLogger(CustomerBean.class.getName());

    @PostConstruct
    private void init() {
        client = ClientBuilder.newClient();
    }

    @PreDestroy
    private void clean() {
```

```
        client.close();
    }

    public String createCustomer(Customer customer) {
        if (customer == null) {
            logger.log(Level.WARNING, "customer is null.");
            return "customerError";
        }
        String navigation;
        Response response =
                client.target("http://localhost:8080/customer/webapi/Customer")
                .request(MediaType.APPLICATION_XML)
                .post(Entity.entity(customer, MediaType.APPLICATION_XML),
                        Response.class);
        if (response.getStatus() == Status.CREATED.getStatusCode()) {
            navigation = "customerCreated";
        } else {
            logger.log(Level.WARNING, "couldn''t create customer with " +
                    "id {0}. Status returned was {1}",
                    new Object[]{customer.getId(), response.getStatus()});
            navigation = "customerError";
        }
        return navigation;
    }

    public String retrieveCustomer(String id) {
        String navigation;
        Customer customer =
                client.target("http://localhost:8080/customer/webapi/Customer")
                .path(id)
                .request(MediaType.APPLICATION_XML)
                .get(Customer.class);
        if (customer == null) {
            navigation = "customerError";
        } else {
            navigation = "customerRetrieved";
        }
        return navigation;
    }

    public List<Customer> retrieveAllCustomers() {
        List<Customer> customers =
                client.target("http://localhost:8080/customer/webapi/Customer")
                .path("all")
                .request(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Customer>>() {});
        return customers;
    }
}
```

This client uses the `POST` and `GET` methods.

All of these HTTP status codes indicate success: 201 for `POST`, 200 for `GET`, and 204 for `DELETE`. For details about the meanings of HTTP status codes, see https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Running the customer Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `customer` application.

To Build, Package, and Deploy the customer Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/rest
   ```

4. Select the `customer` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `customer` project and select **Build**.

   This command builds and packages the application into a WAR file, `customer.war`, located in the `target` directory. Then, the WAR file is deployed to GlassFish Server.

7. Open the web client in a browser at the following URL:

   ```
   http://localhost:8080/customer/
   ```

   The web client allows you to create and view customers.

To Build, Package, and Deploy the customer Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/rest/customer/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `customer.war`, located in the

`target` directory. Then, the WAR file is deployed to GlassFish Server.

4. Open the web client in a browser at the following URL:

```
http://localhost:8080/customer/
```

   The web client allows you to create and view customers.

Jakarta JSON

JSON Binding



This section was last updated for Jakarta EE Version 9.1

This chapter describes the Jakarta JSON Binding. JSON is a data exchange format widely used in web services and other connected applications. For a brief overview of JSON, see Introduction to JSON.

The Jakarta JSON Binding specification provides a standard binding layer (metadata and runtime) between Java classes and JSON documents. One Jakarta JSON Binding reference implementation is Yasson, which is developed through Eclipse.org and is included as part of GlassFish Server.

You can learn more about Yasson at https://projects.eclipse.org/projects/ee4j.yasson.

JSON Binding in the Jakarta EE Platform

Jakarta EE includes support for the Jakarta JSON Binding spec, which provides an API that can serialize Java objects to JSON documents and deserialize JSON documents to Java objects. Jakarta JSON Binding contains the following packages:

- The `jakarta.json.bind` package contains the binding interface, the builder interface, and a configuration class. Main Classes and Interfaces in jakarta.json.bind lists the main classes and interfaces in this package.
- The `jakarta.json.bind.adapter` package contains the `JsonbAdapter` interface, which provides methods for binding custom Java types by converting them to known types.
- The `jakarta.json.bind.annotation` package defines annotations that can be used to customize default binding behavior. Annotations can be used for field, JavaBean property, type, or package elements.
- The `jakarta.json.bind.config` package interfaces and classes for customizing default binding behavior. Main Classes and Interfaces in jakarta.json.bind.config lists the main classes and interfaces in this package.
- The `jakarta.json.bind.serializer` package contains interfaces that are used to create serialization and deserialization routines for custom types that cannot be easily mapped using the `JsonbAdapter` methods. Main Classes and Interfaces in jakarta.json.bind.serializer lists the main interfaces in this package.

- The `jakarta.json.bind.spi` package contains a Service Provider Interface (SPI) for creating JSON Binding implementations. This package contains the `JsonbProvider` class, which contains the methods that a service provider implements.

Main Classes and Interfaces in jakarta.json.bind

Class or Interface	Description
`Jsonb`	Contains the JSON binding methods for serializing Java objects to JSON and deserailizing JSON to Java objects.
`JsonBuilder`	Used by clients to create `Jsonb` instances.
`JsonbConfig`	Used to set configuration properties on `Jsonb` instances. Properties include binding strategies and properties for configuring custom serializers and deserializers.
`JsonbException`	Indicates that a problem occurred during JSON binding.

Main Classes and Interfaces in jakarta.json.bind.config

Class or Interface	Description
`PropertyNamingStrategy`	Used to set how property names are translated.
`PropertyVisibilityStrategy`	Used to set whether fields and methods should be considered properties overriding the default scope and field access behavior.
`BinaryDataStrategy`	Used to set binary encoding.
`PropertyOrderStrategy`	Used to set how properties are ordered during serialization.

Main Classes and Interfaces in jakarta.json.bind.serializer

Class or Interface	Description
`JsonbDeserializer`	Used to create a deserialization routine for a custom type.
`JsonbSerializer`	Used to create a serialization routine for a custom type.

Overview of the JSON Binding API

This section provides basic instructions for using the Jakarta JSON Binding client API. The instructions provide a basis for understanding the Running the jsonbbasics Example Application. Refer to the Jakarta JSON Binding project page for API documentation and a more detailed User Guide.

Creating a jasonb Instance

A `jsonb` instance provides access to methods for binding objects to JSON. A single `jsonb` instance is required for most applications. A `jsonb` instance is created using the `JsonbBuilder` interface, which is a client's entry point to the JSON Binding API. For example:

```
Jsonb jsonb = JsonbBuilder.create();
```

Using the Default Mapping

Jakarta JSON Binding provides default mappings for serializing and deserializing basic Java and Java SE types as well Java date and time classes. To use the default mappings and mapping behavior, create a josnb instance and use the `toJson` method to serialize to JSON and the `fromJson` method to deserialize back to an object. The following example binds a simple `Person` object that contains a single `name` field.

```
Jsonb jsonb = JsonbBuilder.create();

Person person = new Person();
person.name = "Fred";

Jsonb jsonb = JsonbBuilder.create();

// serialize to JSON
String result = jsonb.toJson(person);

// deserialize from JSON
person = jsonb.fromJson("{name:\"joe\"}", Person.class);
```

Using Customizations

Jakarta JSON Binding supports many ways to customize the default mapping behavior. For runtime customizations, a `JsonbConfig` configuration object is used when creating the `jsonbinstance`. The `JsonbConfig` class supports many configuration options and also includes advanced options for binding custom types. For advanced options, see the `JsonbAdapter` interface and the `JsonbSerializer` and `JsonbDeserializer` interfaces.

The following example creates a configuration object that sets the `FORMATTING` property to specify whether or not the serialized JSON data is formatted with linefeeds and indentation.

```
JsonbConfig config = new JsonbConfig()
    .withFormatting(true);

Jsonb jsonb = JsonbBuilder.create(config);
```

Using Annotations

Jakarta JSON Binding includes many annotations that can be used at compile time to customize the default mapping behavior. The following example uses the `@JsonbProperty` annotation to change the `name` field to `person-name` when the object is serialized to JSON.

```
public class Person {
    @JsonbProperty("person-name")
    private String name;
}
```

The resulting JSON document is written as:

```
{
    "person-name": "Fred",
}
```

Running the jsonbbasics Example Application

This section describes how to build and run the `jsonbbasics` example application. This example is a web application that demonstrates how to serialize an object to JSON and how to deserialize JSON to an object.

The `jsonbbasics` example application is in the `jakartaee-examples/tutorial/web/jsonb/jsonbbasics` directory.

Components of the jsonbbasics Example Application

The `jsonbbasics` example application contains the following files.

- Two Jakarta Faces pages.
  - The `index.xhtml` page contains a form to collect data that is used to create a `Person` object.
  - The `jsongenerated.xhtml` page contains a text area that displays the data in JSON format.
- The `JsonbBean.java` managed bean, which is a session-scoped managed bean that stores the data from the form and directs the navigation between the Facelets pages. This file contains code that uses the JSON Binding API.

Running the jsonbbasics Example Application

This section describes how to run the `jsonbbasics` example application from the command line using Maven.

To run the jsonbbasics example application using Maven:

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/jsonb/jsonbbasics
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

4. Open a web browser window and enter the following address:

   ```
   http://localhost:8080/jsonbbasics/
   ```

5. Enter data on form and click Serialize to JSON to submit the form. The following page shows the JSON format of the object data.
6. Click Deserialize JSON. The index page displays and contains the fields populated from the object data.

Further Information about the Jakarta JSON Binding

For more information on Jakarta JSON Binding, see:

- Jakarta JSON Binding 3.0 spec:
  https://jakarta.ee/specifications/jsonb/3.0/
- Specification project:
  https://github.com/eclipse-ee4j/jsonb-api
- Yasson (Implementation):
  https://projects.eclipse.org/projects/ee4j.yasson

JSON Processing



This section was last updated for Jakarta EE Version 9.1

This chapter describes Jakarta JSON Processing. JSON is a data exchange format widely used in web services and other connected applications. Jakarta JSON Processing provides an API to parse, transform, and query JSON data using the object model or the streaming model.

Introduction to JSON

JSON is a text-based data exchange format derived from JavaScript that is used in web services and other connected applications. The following sections provide an introduction to JSON syntax, an overview of JSON uses, and a description of the most common approaches to generate and parse JSON.

JSON Syntax

JSON defines only two data structures: objects and arrays. An object is a set of name-value pairs, and an array is a list of values. JSON defines seven value types: string, number, object, array, true, false, and null.

The following example shows JSON data for a sample object that contains name-value pairs. The value for the name `"phoneNumbers"` is an array whose elements are two objects.

```
{
   "firstName": "Duke",
   "lastName": "Java",
   "age": 18,
   "streetAddress": "100 Internet Dr",
   "city": "JavaTown",
   "state": "JA",
   "postalCode": "12345",
```

```
   "phoneNumbers": [
      { "Mobile": "111-111-1111" },
      { "Home": "222-222-2222" }
   ]
}
```

JSON has the following syntax.

- Objects are enclosed in braces ({}), their name-value pairs are separated by a comma (,), and the name and value in a pair are separated by a colon (:). Names in an object are strings, whereas values may be of any of the seven value types, including another object or an array.
- Arrays are enclosed in brackets ([]), and their values are separated by a comma (,). Each value in an array may be of a different type, including another array or an object.
- When objects and arrays contain other objects or arrays, the data has a tree-like structure.

Uses of JSON

JSON is often used as a common format to serialize and deserialize data in applications that communicate with each other over the Internet. These applications are created using different programming languages and run in very different environments. JSON is suited to this scenario because it is an open standard, it is easy to read and write, and it is more compact than other representations.

RESTful web services use JSON extensively as the format for the data inside requests and responses. The HTTP header used to indicate that the content of a request or a response is JSON data is

```
Content-Type: application/json
```

JSON representations are usually more compact than XML representations because JSON does not have closing tags. Unlike XML, JSON does not have a widely accepted schema for defining and validating the structure of JSON data.

Generating and Parsing JSON Data

For generating and parsing JSON data, there are two programming models, which are similar to those used for XML documents.

- The object model creates a tree that represents the JSON data in memory. The tree can then be navigated, analyzed, or modified. This approach is the most flexible and allows for processing that requires access to the complete contents of the tree. However, it is often slower than the streaming model and requires more memory. The object model generates JSON output by navigating the entire tree at once.
- The streaming model uses an event-based parser that reads JSON data one element at a time. The parser generates events and stops for processing when an object or an array begins or ends, when it finds a key, or when it finds a value. Each element can be processed or discarded by the application code, and then the parser proceeds to the next event. This approach is adequate for local processing, in which the processing of an element does not require information from the

rest of the data. The streaming model generates JSON output to a given stream by making a function call with one element at a time.

There are many JSON generators and parsers available for different programming languages and environments. JSON Processing in the Jakarta EE Platform describes the functionality provided by Jakarta JSON Processing.

JSON Processing in the Jakarta EE Platform

Jakarta EE includes support for the Jakarta JSON Processing spec, which provides an API to parse, transform, and query JSON data using the object model or the streaming model described in Generating and Parsing JSON Data. Jakarta JSON Processing contains the following packages:

- The `jakarta.json` package contains a reader interface, a writer interface, a model builder interface for the object model, and utility classes and Java types for JSON elements. This package also includes several classes that implement other JSON-related standards: JSON Pointer, JSON Patch, and JSON Merge Patch. These standards are used to retrieve, transform or manipulate values in an object model. [web:jsonp::jsonp:::_main_classes_and_interfaces_in_jakarta.json] lists the main classes and interfaces in this package.
- The `jakarta.json.stream` package contains a parser interface and a generator interface for the streaming model. [web:jsonp::jsonp:::_main_classes_and_interfaces_in_jakarta.json.stream] lists the main classes and interfaces in this package.
- The `jakarta.json.spi` package contains a Service Provider Interface (SPI) to plug in implementations for JSON processing objects. This package contains the `JsonProvider` class, which contains the methods that a service provider implements.

Main Classes and Interfaces in jakarta.json

Class or Interface	Description
`Json`	Contains static methods to create instances of JSON parsers, builders, and generators. This class also contains methods to create parser, builder, and generator factory objects.
`JsonReader`	Reads JSON data from a stream and creates an object model in memory.
`JsonObjectBuilder`, `JsonArrayBuilder`	Create an object model or an array model in memory by adding elements from application code.
`JsonWriter`	Writes an object model from memory to a stream.
`JsonValue`	Represents an element (such as an object, an array, or a value) in JSON data.
`JsonStructure`	Represents an object or an array in JSON data. This interface is a subtype of `JsonValue`.
`JsonObject`, `JsonArray`	Represent an object or an array in JSON data. These two interfaces are subtypes of `JsonStructure`.

Class or Interface	Description
`JsonPointer`	Contains methods for operating on specific targets within JSON documents. The targets can be `JsonValue`, `JsonObject`, or `JsonArray` objects.
`JsonPatch`	An interface for supporting a sequence of operations to be applied to a target JSON resource. The operations are defined within a JSON patch document.
`JsonMergePatch`	An interface for supporting updates to target JSON resources. A JSON patch document is compared with the target resource to determine the specific set of change operations to be applied.
`JsonString`, `JsonNumber`	Represent data types for elements in JSON data. These two interfaces are subtypes of `JsonValue`.
`JsonException`	Indicates that a problem occurred during JSON processing.

Main Classes and Interfaces in jakarta.json.stream

Class or Interface	Description
`JsonParser`	Represents an event-based parser that can read JSON data from a stream or from an object model.
`JsonGenerator`	Writes JSON data to a stream one element at a time.

Using the Object Model API

This section describes four use cases of the object model API: creating an object model from JSON data, creating an object model from application code, navigating an object model, and writing an object model to a stream.

Creating an Object Model from JSON Data

The following code demonstrates how to create an object model from JSON data in a text file:

```
import java.io.FileReader;
import jakarta.json.Json;
import jakarta.json.JsonReader;
import jakarta.json.JsonStructure;
...
JsonReader reader = Json.createReader(new FileReader("jsondata.txt"));
JsonStructure jsonst = reader.read();
```

The object reference `jsonst` can be either of type `JsonObject` or of type `JsonArray`, depending on the contents of the file. `JsonObject` and `JsonArray` are subtypes of `JsonStructure`. This reference represents the top of the tree and can be used to navigate the tree or to write it to a stream as JSON data.

Creating an Object Model from Application Code

The following code demonstrates how to create an object model from application code:

```
import jakarta.json.Json;
import jakarta.json.JsonObject;
...
JsonObject model = Json.createObjectBuilder()
   .add("firstName", "Duke")
   .add("lastName", "Java")
   .add("age", 18)
   .add("streetAddress", "100 Internet Dr")
   .add("city", "JavaTown")
   .add("state", "JA")
   .add("postalCode", "12345")
   .add("phoneNumbers", Json.createArrayBuilder()
      .add(Json.createObjectBuilder()
         .add("type", "mobile")
         .add("number", "111-111-1111"))
      .add(Json.createObjectBuilder()
         .add("type", "home")
         .add("number", "222-222-2222")))
   .build();
```

The object reference `model` represents the top of the tree, which is created by nesting calls to the `add` methods and built by calling the `build` method. The `JsonObjectBuilder` class contains the following `add` methods:

```
JsonObjectBuilder add(String name, BigDecimal value)
JsonObjectBuilder add(String name, BigInteger value)
JsonObjectBuilder add(String name, boolean value)
JsonObjectBuilder add(String name, double value)
JsonObjectBuilder add(String name, int value)
JsonObjectBuilder add(String name, JsonArrayBuilder builder)
JsonObjectBuilder add(String name, JsonObjectBuilder builder)
JsonObjectBuilder add(String name, JsonValue value)
JsonObjectBuilder add(String name, long value)
JsonObjectBuilder add(String name, String value)
JsonObjectBuilder addNull(String name)
```

The `JsonArrayBuilder` class contains similar `add` methods that do not have a name (key) parameter. You can nest arrays and objects by passing a new `JsonArrayBuilder` object or a new `JsonObjectBuilder` object to the corresponding `add` method, as shown in this example.

The resulting tree represents the JSON data from JSON Syntax.

Navigating an Object Model

The following code demonstrates a simple approach to navigating an object model:

```
import jakarta.json.JsonValue;
import jakarta.json.JsonObject;
import jakarta.json.JsonArray;
import jakarta.json.JsonNumber;
import jakarta.json.JsonString;
...
public static void navigateTree(JsonValue tree, String key) {
   if (key != null)
      System.out.print("Key " + key + ": ");
   switch(tree.getValueType()) {
      case OBJECT:
         System.out.println("OBJECT");
         JsonObject object = (JsonObject) tree;
         for (String name : object.keySet())
            navigateTree(object.get(name), name);
         break;
      case ARRAY:
         System.out.println("ARRAY");
         JsonArray array = (JsonArray) tree;
         for (JsonValue val : array)
            navigateTree(val, null);
         break;
      case STRING:
         JsonString st = (JsonString) tree;
         System.out.println("STRING " + st.getString());
         break;
      case NUMBER:
         JsonNumber num = (JsonNumber) tree;
         System.out.println("NUMBER " + num.toString());
         break;
      case TRUE:
      case FALSE:
      case NULL:
         System.out.println(tree.getValueType().toString());
         break;
   }
}
```

The method `navigateTree` can be used with the models built in Creating an Object Model from JSON Data and Creating an Object Model from Application Code as follows:

```
navigateTree(model, null);
```

The `navigateTree` method takes two arguments: a JSON element and a key. The key is used only to help print the key-value pairs inside objects. Elements in a tree are represented by the `JsonValue` type. If the element is an object or an array, a new call to this method is made for every element contained in the object or array. If the element is a value, it is printed to the standard output.

The `JsonValue.getValueType` method identifies the element as an object, an array, or a value. For objects, the `JsonObject.keySet` method returns a set of strings that contains the keys in the object, and the `JsonObject.get(String name)` method returns the value of the element whose key is `name`. For arrays, `JsonArray` implements the `List<JsonValue>` interface. You can use enhanced `for` loops with the `Set<String>` instance returned by `JsonObject.keySet` and with instances of `JsonArray`, as shown in this example.

The `navigateTree` method for the model built in Creating an Object Model from Application Code produces the following output:

```
OBJECT
   Key firstName: STRING Duke
   Key lastName: STRING Java
   Key age: NUMBER 18
   Key streetAddress: STRING 100 Internet Dr
   Key city: STRING JavaTown
   Key state: STRING JA
   Key postalCode: STRING 12345
   Key phoneNumbers: ARRAY
      OBJECT
         Key type: STRING mobile
         Key number: STRING 111-111-1111
      OBJECT
         Key type: STRING home
         Key number: STRING 222-222-2222
```

Writing an Object Model to a Stream

The object models created in Creating an Object Model from JSON Data and Creating an Object Model from Application Code can be written to a stream using the `JsonWriter` class as follows:

```
import java.io.StringWriter;
import jakarta.json.JsonWriter;
...
StringWriter stWriter = new StringWriter();
JsonWriter jsonWriter = Json.createWriter(stWriter);
jsonWriter.writeObject(model);
jsonWriter.close();

String jsonData = stWriter.toString();
System.out.println(jsonData);
```

The `Json.createWriter` method takes an output stream as a parameter. The `JsonWriter.writeObject` method writes the object to the stream. The `JsonWriter.close` method closes the underlying output stream.

The following example uses `try-with-resources` to close the JSON writer automatically:

```
StringWriter stWriter = new StringWriter();
try (JsonWriter jsonWriter = Json.createWriter(stWriter)) {
   jsonWriter.writeObject(model);
}

String jsonData = stWriter.toString();
System.out.println(jsonData);
```

Using the Streaming API

This section describes two use cases of the streaming API.

Reading JSON Data Using a Parser

The streaming API is the most efficient approach for parsing JSON text. The following code demonstrates how to create a `JsonParser` object and how to parse JSON data using events:

```
import jakarta.json.Json;
import jakarta.json.stream.JsonParser;
...
JsonParser parser = Json.createParser(new StringReader(jsonData));
while (parser.hasNext()) {
   JsonParser.Event event = parser.next();
   switch(event) {
      case START_ARRAY:
      case END_ARRAY:
      case START_OBJECT:
      case END_OBJECT:
      case VALUE_FALSE:
      case VALUE_NULL:
      case VALUE_TRUE:
         System.out.println(event.toString());
         break;
      case KEY_NAME:
         System.out.print(event.toString() + " " +
                          parser.getString() + " - ");
         break;
      case VALUE_STRING:
      case VALUE_NUMBER:
         System.out.println(event.toString() + " " +
                            parser.getString());
         break;
   }
}
```

This example consists of three steps.

1. Obtain a parser instance by calling the `Json.createParser` static method.

2. Iterate over the parser events with the `JsonParser.hasNext` and the `JsonParser.next` methods.
3. Perform local processing for each element.

The example shows the ten possible event types from the parser. The parser's `next` method advances it to the next event. For the event types `KEY_NAME`, `VALUE_STRING`, and `VALUE_NUMBER`, you can obtain the content of the element by calling the method `JsonParser.getString`. For `VALUE_NUMBER` events, you can also use the following methods:

- `JsonParser.isIntegralNumber`
- `JsonParser.getInt`
- `JsonParser.getLong`
- `JsonParser.getBigDecimal`

See the Jakarta EE API reference for the `jakarta.json.stream.JsonParser` interface for more information.

The output of this example is the following:

```
START_OBJECT
KEY_NAME firstName - VALUE_STRING Duke
KEY_NAME lastName - VALUE_STRING Java
KEY_NAME age - VALUE_NUMBER 18
KEY_NAME streetAddress - VALUE_STRING 100 Internet Dr
KEY_NAME city - VALUE_STRING JavaTown
KEY_NAME state - VALUE_STRING JA
KEY_NAME postalCode - VALUE_STRING 12345
KEY_NAME phoneNumbers - START_ARRAY
START_OBJECT
KEY_NAME type - VALUE_STRING mobile
KEY_NAME number - VALUE_STRING 111-111-1111
END_OBJECT
START_OBJECT
KEY_NAME type - VALUE_STRING home
KEY_NAME number - VALUE_STRING 222-222-2222
END_OBJECT
END_ARRAY
END_OBJECT
```

Writing JSON Data Using a Generator

The following code demonstrates how to write JSON data to a file using the streaming API:

```
FileWriter writer = new FileWriter("test.txt");
JsonGenerator gen = Json.createGenerator(writer);
gen.writeStartObject()
   .write("firstName", "Duke")
   .write("lastName", "Java")
   .write("age", 18)
```

```
    .write("streetAddress", "100 Internet Dr")
    .write("city", "JavaTown")
    .write("state", "JA")
    .write("postalCode", "12345")
    .writeStartArray("phoneNumbers")
       .writeStartObject()
          .write("type", "mobile")
          .write("number", "111-111-1111")
       .writeEnd()
       .writeStartObject()
          .write("type", "home")
          .write("number", "222-222-2222")
       .writeEnd()
    .writeEnd()
 .writeEnd();
 gen.close();
```

This example obtains a JSON generator by calling the `Json.createGenerator` static method, which takes a writer or an output stream as a parameter. The example writes JSON data to the `test.txt` file by nesting calls to the `write`, `writeStartArray`, `writeStartObject`, and `writeEnd` methods. The `JsonGenerator.close` method closes the underlying writer or output stream.

JSON in Jakarta EE RESTful Web Services

This section explains how the Jakarta JSON Processing is related to other Jakarta EE packages that provide JSON support for RESTful web services. See [websvcs:rest::rest:::_building_restful_web_services_with_jakarta_rest] for more information on RESTful web services.

Jersey, the Jakarta RESTful Web Services implementation included in GlassFish Server, provides support for binding JSON data from RESTful resource methods to Java objects using Jakarta XML Binding, as described in Using Jakarta REST with Jakarta XML Binding in [websvcs:rest-advanced::rest-advanced:::_jakarta_rest_advanced_topics_and_an_example]. However, JSON support is not part of Jakarta RESTful Web Services or Jakarta XML Binding, so that procedure may not work for Jakarta EE implementations other than GlassFish Server.

You can still use the Jakarta JSON Processing with Jakarta RESTful Web Services resource methods. For more information, see the sample code for JSON Processing included with the Jakarta EE tutorial examples.

The jsonpmodel Example Application

This section describes how to build and run the `jsonpmodel` example application. This example is a web application that demonstrates how to create an object model from form data, how to parse JSON data, and how write JSON data using the object model API.

The `jsonpmodel` example application is in the `jakartaee-examples/tutorial/web/jsonp/jsonpmodel` directory.

Components of the jsonpmodel Example Application

The `jsonpmodel` example application contains the following files.

- Three Jakarta Faces pages.
  - The `index.xhtml` page contains a form to collect information.
  - The `modelcreated.xhtml` page contains a text area that displays JSON data.
  - The `parsejson.xhtml` page contains a table that shows the elements of the object model.
- The `ObjectModelBean.java` managed bean, which is a session-scoped managed bean that stores the data from the form and directs the navigation between the Facelets pages. This file also contains code that uses the JSON object model API.

The code used in `ObjectModelBean.java` to create an object model from the data in the form is similar to the example in Creating an Object Model from Application Code. The code to write JSON output from the model is similar to the example in Writing an Object Model to a Stream. The code to navigate the object model tree is similar to the example in Navigating an Object Model.

Running the jsonpmodel Example Application

This section describes how to run the `jsonpmodel` example application using NetBeans IDE and from the command line.

To Run the jsonpmodel Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/jsonp
   ```

4. Select the `jsonpmodel` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `jsonpmodel` project and select **Run**.

   This command builds and packages the application into a WAR file (`jsonpmodel.war`) located in the `target` directory, deploys it to the server, and opens a web browser window with the following URL:

   ```
   http://localhost:8080/jsonpmodel/
   ```

7. Edit the data on the page and click Create a JSON Object to submit the form. The following page shows a JSON object that contains the data from the form.
8. Click Parse JSON. The following page contains a table that lists the nodes of the object model tree.

To Run the jsonpmodel Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/jsonp/jsonpmodel
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

4. Open a web browser window and enter the following address:

   ```
   http://localhost:8080/jsonpmodel/
   ```

5. Edit the data on the page and click Create a JSON Object to submit the form. The following page shows a JSON object that contains the data from the form.
6. Click Parse JSON. The following page contains a table that lists the nodes of the object model tree.

The jsonpstreaming Example Application

This section describes how to build and run the `jsonpstreaming` example application. This example is a web application that demonstrates how to create JSON data from form data, how to parse JSON data, and how to write JSON output using the streaming API.

The `jsonpstreaming` example application is in the `jakartaee-examples/tutorial/web/jsonp/jsonpstreaming` directory.

Components of the jsonpstreaming Example Application

The `jsonpstreaming` example application contains the following files.

- Three Jakarta Faces pages.
  - The `index.xhtml` page contains a form to collect information.
  - The `filewritten.xhtml` page contains a text area that displays JSON data.
  - The `parsed.xhtml` page contains a table that lists the events from the parser.
- The `StreamingBean.java` managed bean, a session-scoped managed bean that stores the data from the form and directs the navigation between the Facelets pages. This file also contains code that uses the JSON streaming API.

The code used in `StreamingBean.java` to write JSON data to a file is similar to the example in Writing JSON Data Using a Generator. The code to parse JSON data from a file is similar to the example in Reading JSON Data Using a Parser.

Running the jsonpstreaming Example Application

This section describes how to run the `jsonpstreaming` example application using NetBeans IDE and from the command line.

To Run the jsonpstreaming Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/jsonp
   ```

4. Select the `jsonpstreaming` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `jsonpstreaming` project and select **Run**.

   This command builds and packages the application into a WAR file (`jsonpstreaming.war`) located in the `target` directory, deploys it to the server, and opens a web browser window with the following URL:

   ```
   http://localhost:8080/jsonpstreaming/
   ```

7. Edit the data on the page and click Write a JSON Object to a File to submit the form and write a JSON object to a text file. The following page shows the contents of the text file.
8. Click Parse JSON from File. The following page contains a table that lists the parser events for the JSON data in the text file.

To Run the jsonpstreaming Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/jsonp/jsonpstreaming/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

4. Open a web browser window and enter the following URL:

   ```
   http://localhost:8080/jsonpstreaming/
   ```

5. Edit the data on the page and click Write a JSON Object to a File to submit the form and write a JSON object to a text file. The following page shows the contents of the text file.
6. Click Parse JSON from File. The following page contains a table that lists the parser events for the JSON data in the text file.

Further Information about the Jakarta JSON Processing

For more information on JSON processing in Jakarta EE, see the Jakarta JSON Processing specification:

https://jakarta.ee/specifications/jsonp/2.1/

Jakarta EE Web Profile

Jakarta CDI Full

Jakarta Contexts and Dependency Injection: Advanced Topics



This section was last updated for Jakarta EE Version 9.1

This chapter describes more advanced features of Jakarta Contexts and Dependency Injection. Specifically, it covers additional features CDI provides to enable loose coupling of components with strong typing, in addition to those described in Overview of CDI.

Packaging CDI Applications

When you deploy a Jakarta EE application, CDI looks for beans inside bean archives. A bean archive is any module that contains beans that the CDI runtime can manage and inject. There are two kinds of bean archives: explicit bean archives and implicit bean archives.

An explicit bean archive is an archive that contains a `beans.xml` deployment descriptor, which can be an empty file, contain no version number, or contain the version number 3.0 with the `bean-discovery-mode` attribute set to `all`. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                           https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
       version="3.0" bean-discovery-mode="all">
    ...
</beans>
```

CDI can manage and inject any bean in an explicit archive, except those annotated with `@Vetoed`.

An implicit bean archive is an archive that contains some beans annotated with a scope type, contains no `beans.xml` deployment descriptor, or contains a `beans.xml` deployment descriptor with the `bean-discovery-mode` attribute set to `annotated`.

In an implicit archive, CDI can only manage and inject beans annotated with a scope type.

For a web application, the `beans.xml` deployment descriptor, if present, must be in the `WEB-INF` directory. For enterprise bean modules or JAR files, the `beans.xml` deployment descriptor, if present, must be in the `META-INF` directory.

Using Alternatives in CDI Applications

When you have more than one version of a bean that you use for different purposes, you can choose between them during the development phase by injecting one qualifier or another, as

shown in The simplegreeting CDI Example.

Instead of having to change the source code of your application, however, you can make the choice at deployment time by using alternatives.

Alternatives are commonly used for purposes such as the following:

- To handle client-specific business logic that is determined at runtime
- To specify beans that are valid for a particular deployment scenario (for example, when country-specific sales tax laws require country-specific sales tax business logic)
- To create dummy (mock) versions of beans to be used for testing

To make a bean available for lookup, injection, or EL resolution using this mechanism, give it a `jakarta.enterprise.inject.Alternative` annotation and then use the `alternatives` element to specify it in the `beans.xml` file.

For example, you might want to create a full version of a bean and also a simpler version that you use only for certain kinds of testing. The example described in The encoder Example: Using Alternatives contains two such beans, `CoderImpl` and `TestCoderImpl`. The test bean is annotated as follows:

```
@Alternative
public class TestCoderImpl implements Coder { ... }
```

The full version is not annotated:

```
public class CoderImpl implements Coder { ... }
```

The managed bean injects an instance of the `Coder` interface:

```
@Inject
Coder coder;
```

The alternative version of the bean is used by the application only if that version is declared as follows in the `beans.xml` file:

```
<beans ...>
    <alternatives>
        <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
    </alternatives>
</beans>
```

If the `alternatives` element is commented out in the `beans.xml` file, the `CoderImpl` class is used.

You can also have several beans that implement the same interface, all annotated `@Alternative`. In

this case, you must specify in the `beans.xml` file which of these alternative beans you want to use. If `CoderImpl` were also annotated `@Alternative`, one of the two beans would always have to be specified in the `beans.xml` file.

The alternatives that you specify in the `beans.xml` file apply only to classes in the same archive. Use the `@Priority` annotation to specify alternatives globally for an application that consists of multiple modules, as in the following example:

```
@Alternative
@Priority(Interceptor.Priority.APPLICATION+10)
public class TestCoderImpl implements Coder { ... }
```

The alternative with higher priority value is selected if several alternative beans that implement the same interface are annotated with `@Priority`. You do not need to specify the alternative in the `beans.xml` file when you use the `@Priority` annotation.

Using Specialization

Specialization has a function similar to that of alternatives in that it allows you to substitute one bean for another. However, you might want to make one bean override the other in all cases. Suppose you defined the following two beans:

```
@Default @Asynchronous
public class AsynchronousService implements Service { ... }

@Alternative
public class MockAsynchronousService extends AsynchronousService { ... }
```

If you then declared `MockAsynchronousService` as an alternative in your `beans.xml` file, the following injection point would resolve to `MockAsynchronousService`:

```
@Inject Service service;
```

The following, however, would resolve to `AsynchronousService` rather than `MockAsynchronousService`, because `MockAsynchronousService` does not have the `@Asynchronous` qualifier:

```
@Inject @Asynchronous Service service;
```

To make sure that `MockAsynchronousService` was always injected, you would have to implement all bean types and bean qualifiers of `AsynchronousService`. However, if `AsynchronousService` declared a producer method or observer method, even this cumbersome mechanism would not ensure that the other bean was never invoked. Specialization provides a simpler mechanism.

Specialization happens at development time as well as at runtime. If you declare that one bean specializes another, it extends the other bean class, and at runtime the specialized bean completely replaces the other bean. If the first bean is produced by means of a producer method, you must also

override the producer method.

You specialize a bean by giving it the `jakarta.enterprise.inject.Specializes` annotation. For example, you might declare a bean as follows:

```
@Specializes
public class MockAsynchronousService extends AsynchronousService { ... }
```

In this case, the `MockAsynchronousService` class will always be invoked instead of the `AsynchronousService` class.

Usually, a bean marked with the `@Specializes` annotation is also an alternative and is declared as an alternative in the `beans.xml` file. Such a bean is meant to stand in as a replacement for the default implementation, and the alternative implementation automatically inherits all qualifiers of the default implementation as well as its EL name, if it has one.

Using Producer Methods, Producer Fields, and Disposer Methods in CDI Applications

A producer method generates an object that can then be injected.Typically, you use producer methods in the following situations:

- When you want to inject an object that is not itself a bean
- When the concrete type of the object to be injected may vary at runtime
- When the object requires some custom initialization that the bean constructor does not perform

For more information on producer methods, see Injecting Objects by Using Producer Methods.

A producer field is a simpler alternative to a producer method; it is a field of a bean that generates an object. It can be used instead of a simple getter method. Producer fields are particularly useful for declaring Jakarta EE resources such as data sources, JMS resources, and web service references.

A producer method or field is annotated with the `jakarta.enterprise.inject.Produces` annotation.

Using Producer Methods

A producer method can allow you to select a bean implementation at runtime instead of at development time or deployment time. For example, in the example described in The producermethods Example: Using a Producer Method to Choose a Bean Implementation, the managed bean defines the following producer method:

```
@Produces
@Chosen
@RequestScoped
public Coder getCoder() {

    switch (coderType) {
        case TEST:
            return new TestCoderImpl();
```

```
        case SHIFT:
            return new CoderImpl();
        default:
            return null;
    }
}
```

Here, `getCoder` becomes in effect a getter method, and when the `coder` property is injected with the same qualifier and other annotations as the method, the selected version of the interface is used.

```
@Inject
@Chosen
@RequestScoped
Coder coder;
```

Specifying the qualifier is essential: It tells CDI which `Coder` to inject. Without it, the CDI implementation would not be able to choose between `CoderImpl`, `TestCoderImpl`, and the one returned by `getCoder` and would cancel deployment, informing the user of the ambiguous dependency.

Using Producer Fields to Generate Resources

A common use of a producer field is to generate an object such as a JDBC `DataSource` or a Jakarta Persistence `EntityManager` (see [persist:persistence-intro::persistence-intro:::_introduction_to_jakarta_persistence], for more information). The object can then be managed by the container. For example, you could create a `@UserDatabase` qualifier and then declare a producer field for an entity manager as follows:

```
@Produces
@UserDatabase
@PersistenceContext
private EntityManager em;
```

The `@UserDatabase` qualifier can be used when you inject the object into another bean, `RequestBean`, elsewhere in the application:

```
    @Inject
    @UserDatabase
    EntityManager em;
    ...
```

The producerfields Example: Using Producer Fields to Generate Resources shows how to use producer fields to generate an entity manager. You can use a similar mechanism to inject `@Resource`, `@EJB`, or `@WebServiceRef` objects.

To minimize the reliance on resource injection, specify the producer field for the resource in one

place in the application, and then inject the object wherever in the application you need it.

Using a Disposer Method

You can use a producer method or a producer field to generate an object that needs to be removed when its work is completed. If you do, you need a corresponding disposer method, annotated with a `@Disposes` annotation. For example, you can close the entity manager as follows:

```
public void close(@Disposes @UserDatabase EntityManager em) {
    em.close();
}
```

The disposer method is called automatically when the context ends (in this case, at the end of the conversation, because `RequestBean` has conversation scope), and the parameter in the `close` method receives the object produced by the producer field.

Using Predefined Beans in CDI Applications

Jakarta EE provides predefined beans that implement the following interfaces.

- `jakarta.transaction.UserTransaction`: A Jakarta Transactions user transaction.
- `java.security.Principal`: The abstract notion of a principal, which represents any entity, such as an individual, a corporation, or a login ID. Whenever the injected principal is accessed, it always represents the identity of the current caller. For example, a principal is injected into a field at initialization. Later, a method that uses the injected principal is called on the object into which the principal was injected. In this situation, the injected principal represents the identity of the current caller when the method is run.
- `jakarta.validation.Validator`: A validator for bean instances. The bean that implements this interface enables a `Validator` object for the default bean validation object `ValidatorFactory` to be injected.
- `jakarta.validation.ValidatorFactory`: A factory class for returning initialized `Validator` instances. The bean that implements this interface enables the default bean validation `ValidatorFactory` object to be injected.
- `jakarta.servlet.http.HttpServletRequest`: An HTTP request from a client. The bean that implements this interface enables a servlet to obtain all the details of a request.
- `jakarta.servlet.http.HttpSession`: An HTTP session between a client and a server. The bean that implements this interface enables a servlet to access information about a session and to bind objects to a session.
- `jakarta.servlet.ServletContext`: A context object that servlets can use to communicate with the servlet container.

To inject a predefined bean, create an injection point to obtain an instance of the bean by using the `jakarta.annotation.Resource` annotation for resources or the `jakarta.inject.Inject` annotation for CDI beans. For the bean type, specify the class name of the interface the bean implements.

Injection of Predefined Beans

Predefined Bean	Resource or CDI Bean	Injection Example
`UserTransaction`	Resource	`@Resource UserTransaction transaction;`
`Principal`	Resource	`@Resource Principal principal;`
`Validator`	Resource	`@Resource Validator validator;`
`ValidatorFactory`	Resource	`@Resource ValidatorFactory factory;`
`HttpServletRequest`	CDI bean	`@Inject HttpServletRequest req;`
`HttpSession`	CDI bean	`@Inject HttpSession session;`
`ServletContext`	CDI bean	`@Inject ServletContext context;`

Predefined beans are injected with dependent scope and the predefined default qualifier `@Default`.

For more information about injecting resources, see Resource Injection.

The following code snippet shows how to use the `@Resource` and `@Inject` annotations to inject predefined beans. This code snippet injects a user transaction and a context object into the servlet class `TransactionServlet`. The user transaction is an instance of the predefined bean that implements the `jakarta.transaction.UserTransaction` interface. The context object is an instance of the predefined bean that implements the `jakarta.servlet.ServletContext` interface.

```
import jakarta.annotation.Resource;
import jakarta.inject.Inject;
import jakarta.servlet.http.HttpServlet;
import jakarta.transaction.UserTransaction;
...
public class TransactionServlet extends HttpServlet {
    @Resource UserTransaction transaction;
    @Inject ServletContext context;
    ...
}
```

Using Events in CDI Applications

Events allow beans to communicate without any compile-time dependency. One bean can define an event, another bean can fire the event, and yet another bean can handle the event. In addition, events can be fired asynchronously. The beans can be in separate packages and even in separate tiers of the application.

Defining Events

An event consists of the following:

- The event object, a Java object
- Zero or more qualifier types, the event qualifiers

For example, in the `billpayment` example described in The billpayment Example: Using Events and Interceptors, a `PaymentEvent` bean defines an event using three properties, which have setter and getter methods:

```
public String paymentType;
public BigDecimal value;
public Date datetime;

public PaymentEvent() {
}
```

The example also defines qualifiers that distinguish between two kinds of `PaymentEvent`. Every event also has the default qualifier `@Any`.

Using Observer Methods to Handle Events

An event handler uses an observer method to consume events.

Each observer method takes as a parameter an event of a specific event type that is annotated with the `@Observes` annotation and with any qualifiers for that event type. The observer method is notified of an event if the event object matches the event type and if all the qualifiers of the event match the observer method event qualifiers.

The observer method can take other parameters in addition to the event parameter. The additional parameters are injection points and can declare qualifiers.

The event handler for the `billpayment` example, `PaymentHandler`, defines two observer methods, one for each type of `PaymentEvent`:

```
public void creditPayment(@Observes @Credit PaymentEvent event) {
    ...
}

public void debitPayment(@Observes @Debit PaymentEvent event) {
    ...
}
```

Conditional and Transactional Observer Methods

Observer methods can also be conditional or transactional:

- A conditional observer method is notified of an event only if an instance of the bean that defines the observer method already exists in the current context. To declare a conditional observer method, specify `notifyObserver=IF_EXISTS` as an argument to `@Observes`:

  ```
  @Observes(notifyObserver=IF_EXISTS)
  ```

To obtain the default unconditional behavior, you can specify `@Observes(notifyObserver=ALWAYS)`.

- A transactional observer method is notified of an event during the before-completion or after-completion phase of the transaction in which the event was fired. You can also specify that the notification is to occur only after the transaction has completed successfully or unsuccessfully. To specify a transactional observer method, use any of the following arguments to `@Observes`:

```
@Observes(during=BEFORE_COMPLETION)

@Observes(during=AFTER_COMPLETION)

@Observes(during=AFTER_SUCCESS)

@Observes(during=AFTER_FAILURE)
```

  To obtain the default nontransactional behavior, specify `@Observes(during=IN_PROGRESS)`.

  An observer method that is called before completion of a transaction may call the `setRollbackOnly` method on the transaction instance to force a transaction rollback.

Observer methods may throw exceptions. If a transactional observer method throws an exception, the exception is caught by the container. If the observer method is nontransactional, the exception terminates processing of the event, and no other observer methods for the event are called.

Observer Method Ordering

Before a certain observer event notification is generated, the container determines the order in which observer methods for that event are invoked. Observer method order is established through the declaration of the `@Priority` annotation on an event parameter of an observer method, as in the following example:

```
void afterLogin(@Observes
@Priority(jakarta.interceptor.Interceptor.Priority.APPLICATION) LoggedInEvent event) {
... }
```

Note the following:

- If the `@Priority` annotation is not specified, the default value is `jakarta.interceptor.Interceptor.Priority.APPLICATION + 500`.
- If two or more observer methods are assigned the same priority, the order in which they are invoked is undefined and is therefore unpredictable.

Firing Events

Beans fire events by implementing an instance of the `jakarta.enterprise.event.Event` interface. Events can be fired synchronously or asynchronously.

Firing Events Synchronously

To activate an event synchronously, call the `jakarta.enterprise.event.Event.fire` method. This method fires an event and notifies any observer methods.

In the `billpayment` example, a managed bean called `PaymentBean` fires the appropriate event by using information it receives from the user interface. There are actually four event beans, two for the event object and two for the payload. The managed bean injects the two event beans. The `pay` method uses a `switch` statement to choose which event to fire, using `new` to create the payload.

```
@Inject
@Credit
Event<PaymentEvent> creditEvent;

@Inject
@Debit
Event<PaymentEvent> debitEvent;

private static final int DEBIT = 1;
private static final int CREDIT = 2;
private int paymentOption = DEBIT;
...

@Logged
public String pay() {
    ...
    switch (paymentOption) {
        case DEBIT:
            PaymentEvent debitPayload = new PaymentEvent();
            // populate payload ...
            debitEvent.fire(debitPayload);
            break;
        case CREDIT:
            PaymentEvent creditPayload = new PaymentEvent();
            // populate payload ...
            creditEvent.fire(creditPayload);
            break;
        default:
            logger.severe("Invalid payment option!");
    }
    ...
}
```

The argument to the `fire` method is a `PaymentEvent` that contains the payload. The fired event is then consumed by the observer methods.

Firing Events Asynchronously

To activate an event asynchronously, call the `jakarta.enterprise.event.Event.fireAsync` method. This method calls all resolved asynchronous observers in one or more different threads.

```
@Inject Event<LoggedInEvent> loggedInEvent;

public void login() {
    ...
    loggedInEvent.fireAsync( new LoggedInEvent(user) );
}
```

The invocation of the `fireAsync()` method returns immediately.

When events are fired asynchronously, observer methods are notified asynchronously. Consequently, observer method ordering cannot be guaranteed, because observer method invocation and the firing of asynchronous events occur on separate threads.

Using Interceptors in CDI Applications

An interceptor is a class used to interpose in method invocations or lifecycle events that occur in an associated target class. The interceptor performs tasks, such as logging or auditing, that are separate from the business logic of the application and are repeated often within an application. Such tasks are often called cross-cutting tasks. Interceptors allow you to specify the code for these tasks in one place for easy maintenance. When interceptors were first introduced to the Jakarta EE platform, they were specific to enterprise beans. On the Jakarta EE platform, you can use them with Jakarta EE managed objects of all kinds, including managed beans.

For information on Jakarta EE interceptors, see [supporttechs:interceptors::interceptors:::_using_jakarta_ee_interceptors].

An interceptor class often contains a method annotated `@AroundInvoke`, which specifies the tasks the interceptor will perform when intercepted methods are invoked. It can also contain a method annotated `@PostConstruct`, `@PreDestroy`, `@PrePassivate`, or `@PostActivate`, to specify lifecycle callback interceptors, and a method annotated `@AroundTimeout`, to specify enterprise bean timeout interceptors. An interceptor class can contain more than one interceptor method, but it must have no more than one method of each type.

Along with an interceptor, an application defines one or more interceptor binding types, which are annotations that associate an interceptor with target beans or methods. For example, the `billpayment` example contains an interceptor binding type named `@Logged` and an interceptor named `LoggedInterceptor`. The interceptor binding type declaration looks something like a qualifier declaration, but it is annotated with `jakarta.interceptor.InterceptorBinding`:

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {
}
```

An interceptor binding also has the `java.lang.annotation.Inherited` annotation, to specify that the annotation can be inherited from superclasses. The `@Inherited` annotation also applies to custom

scopes (not discussed in this tutorial) but does not apply to qualifiers.

An interceptor binding type may declare other interceptor bindings.

The interceptor class is annotated with the interceptor binding as well as with the `@Interceptor` annotation. For an example, see The LoggedInterceptor Interceptor Class.

Every `@AroundInvoke` method takes a `jakarta.interceptor.InvocationContext` argument, returns a `java.lang.Object`, and throws an `Exception`. It can call `InvocationContext` methods. The `@AroundInvoke` method must call the `proceed` method, which causes the target class method to be invoked.

Once an interceptor and binding type are defined, you can annotate beans and individual methods with the binding type to specify that the interceptor is to be invoked either on all methods of the bean or on specific methods. For example, in the `billpayment` example, the `PaymentHandler` bean is annotated `@Logged`, which means that any invocation of its business methods will cause the interceptor's `@AroundInvoke` method to be invoked:

```
@Logged
@SessionScoped
public class PaymentHandler implements Serializable {...}
```

However, in the `PaymentBean` bean, only the `pay` and `reset` methods have the `@Logged` annotation, so the interceptor is invoked only when these methods are invoked:

```
@Logged
public String pay() {...}

@Logged
public void reset() {...}
```

In order for an interceptor to be invoked in a CDI application, it must, like an alternative, be specified in the `beans.xml` file. For example, the `LoggedInterceptor` class is specified as follows:

```
<interceptors>
    <class>ee.jakarta.tutorial.billpayment.interceptors.LoggedInterceptor</class>
</interceptors>
```

If an application uses more than one interceptor, the interceptors are invoked in the order specified in the `beans.xml` file.

The interceptors that you specify in the `beans.xml` file apply only to classes in the same archive. Use the `@Priority` annotation to specify interceptors globally for an application that consists of multiple modules, as in the following example:

```
@Logged
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
```

```
public class LoggedInterceptor implements Serializable { ... }
```

Interceptors with lower priority values are called first. You do not need to specify the interceptor in the `beans.xml` file when you use the `@Priority` annotation.

Using Decorators in CDI Applications

A decorator is a Java class that is annotated `jakarta.decorator.Decorator` and that has a corresponding `decorators` element in the `beans.xml` file.

A decorator bean class must also have a delegate injection point, which is annotated `jakarta.decorator.Delegate`. This injection point can be a field, a constructor parameter, or an initializer method parameter of the decorator class.

Decorators are outwardly similar to interceptors. However, they actually perform tasks complementary to those performed by interceptors. Interceptors perform cross-cutting tasks associated with method invocation and with the lifecycles of beans, but cannot perform any business logic. Decorators, on the other hand, do perform business logic by intercepting business methods of beans. This means that instead of being reusable for different kinds of applications, as are interceptors, their logic is specific to a particular application.

For example, instead of using an alternative `TestCoderImpl` class for the `encoder` example, you could create a decorator as follows:

```
@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject
    @Delegate
    @Any
    Coder coder;

    public String codeString(String s, int tval) {
        int len = s.length();

        return "\"" + s + "\" becomes " + "\"" + coder.codeString(s, tval)
                + "\", " + len + " characters in length";
    }
}
```

See The decorators Example: Decorating a Bean for an example that uses this decorator.

This simple decorator returns more detailed output than the encoded string returned by the `CoderImpl.codeString` method. A more complex decorator could store information in a database or perform some other business logic.

A decorator can be declared as an abstract class so that it does not have to implement all the business methods of the interface.

In order for a decorator to be invoked in a CDI application, it must, like an interceptor or an alternative, be specified in the `beans.xml` file. For example, the `CoderDecorator` class is specified as follows:

```
<decorators>
    <class>ee.jakarta.tutorial.decorators.CoderDecorator</class>
</decorators>
```

If an application uses more than one decorator, the decorators are invoked in the order in which they are specified in the `beans.xml` file.

If an application has both interceptors and decorators, the interceptors are invoked first. This means, in effect, that you cannot intercept a decorator.

The decorators that you specify in the `beans.xml` file apply only to classes in the same archive. Use the `@Priority` annotation to specify decorators globally for an application that consists of multiple modules, as in the following example:

```
@Decorator
@Priority(Interceptor.Priority.APPLICATION)
public abstract class CoderDecorator implements Coder { ... }
```

Decorators with lower priority values are called first. You do not need to specify the decorator in the `beans.xml` when you use the `@Priority` annotation.

Using Stereotypes in CDI Applications

A stereotype is a kind of annotation, applied to a bean, that incorporates other annotations. Stereotypes can be particularly useful in large applications in which you have a number of beans that perform similar functions. A stereotype is a kind of annotation that specifies the following:

- A default scope
- Zero or more interceptor bindings
- Optionally, a `@Named` annotation, guaranteeing default EL naming
- Optionally, an `@Alternative` annotation, specifying that all beans with this stereotype are alternatives

A bean annotated with a particular stereotype will always use the specified annotations, so you do not have to apply the same annotations to many beans.

For example, you might create a stereotype named `Action`, using the `jakarta.enterprise.inject.Stereotype` annotation:

```
@RequestScoped
@Secure
@Transactional
```

```
@Named
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

All beans annotated `@Action` will have request scope, use default EL naming, and have the interceptor bindings `@Transactional` and `@Secure`.

You could also create a stereotype named `Mock`:

```
@Alternative
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {}
```

All beans with this annotation are alternatives.

It is possible to apply multiple stereotypes to the same bean, so you can annotate a bean as follows:

```
@Action
@Mock
public class MockLoginAction extends LoginAction { ... }
```

It is also possible to override the scope specified by a stereotype, simply by specifying a different scope for the bean. The following declaration gives the `MockLoginAction` bean session scope instead of request scope:

```
@SessionScoped
@Action
@Mock
public class MockLoginAction extends LoginAction { ... }
```

CDI makes available a built-in stereotype called `Model`, which is intended for use with beans that define the model layer of a model-view-controller application architecture. This stereotype specifies that a bean is both `@Named` and `@RequestScoped`:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```

Using the Built-In Annotation Literals

The following built-in annotations define a `Literal` static nested class, which can be used as a convenience feature for creating instances of annotations:

- `jakarta.enterprise.inject.Any`
- `jakarta.enterprise.inject.Default`
- `jakarta.enterprise.inject.New`
- `jakarta.enterprise.inject.Specializes`
- `jakarta.enterprise.inject.Vetoed`
- `jakarta.enterprise.util.Nonbinding`
- `jakarta.enterprise.context.Initialized`
- `jakarta.enterprise.context.Destroyed`
- `jakarta.enterprise.context.RequestScoped`
- `jakarta.enterprise.context.SessionScoped`
- `jakarta.enterprise.context.ApplicationScoped`
- `jakarta.enterprise.context.Dependent`
- `jakarta.enterprise.context.ConversationScoped`
- `jakarta.enterprise.inject.Alternative`
- `jakarta.enterprise.inject.Typed`

For example:

```
Default defaultLiteral = new Default.Literal();

RequestScoped requestScopedLiteral = RequestScoped.Literal.INSTANCE;

Initialized initializedForApplicationScoped = new
Initialized.Literal(ApplicationScoped.class);

Initialized initializedForRequestScoped = Initialized.Literal.of(RequestScoped.class);
```

Using the Configurators Interfaces

The CDI 2.0 specification defines the following Configurators interfaces, which are used for dynamically defining and modifying CDI objects:

Interface	Description

`AnnotatedTypeConfigurator` SPI	Helps create and configure the following type metadata:
	
	`AnnotatedType`
	
	`AnnotatedField`
	
	`AnnotatedConstructor`
	
	`AnnotatedMethod`
	
	`AnnotatedParameter`

`InjectionPointConfigurator` interface	Helps configure an existing `InjectionPoint` instance
`BeanAttributesConfigurator` interface	Helps configure a new `BeanAttributes` instance
`BeanConfigurator` interface	Helps configure a new `Bean` instance
`ObserverMethodConfigurator` interface	Helps configure an `ObserverMethod` instance
`ProducerConfigurator` interface	Helps configure a `Producer` instance

Bootstrapping a CDI Container in Java SE

This chapter explains how to use the API for bootstrapping a CDI container in Java SE. This capability allows you to run CDI applications on Java SE and obtain beans, independently of an application server or any Jakarta EE APIs.

For more information about bootstrapping a CDI container in Java SE, see the *Weld Reference Guide* at https://weld.cdi-spec.org/documentation/.

The Bootstrap API

The API for bootstrapping a CDI container in Java SE consists of the following entities:

- `jakarta.enterprise.inject.se.SeContainerInitializer` class – Allows you to configure and bootstrap the CDI container. This class includes the following key methods:
  - `newInstance()` obtains a `SeContainerInitializer` instance, which allows you to configure the container prior to bootstrapping it.
  - `initialize()` bootstraps the container.
- `jakarta.enterprise.inject.se.SeContainer` interface – Provides access to the `BeanManager` instance for programmatic lookup, as defined in the `SeContainer` interface, which is described at https://jakarta.ee/specifications/cdi/3.0/jakarta-cdi-spec-3.0.html#se_container.

Configuring the CDI Container

The configuration of the `SeContainerInitializer` instance allows the explicit addition of elements into an internal **synthetic bean archive**. The synthetic bean archive represents the set of beans that have been loaded while initializing the container. The contents of the synthetic bean archive depend on whether discovery is enabled:

- If discovery is enabled, the synthetic bean archive is created using standard bean discovery rules and contains a superset of all JAR files on the classpath. Archives that do not include a `beans.xml` file are excluded.
- If discovery is disabled, and beans are added programmatically, the synthetic bean archive contains only the beans that have been programmatically added.

Running the Advanced Contexts and Dependency Injection Examples



This section was last updated for Jakarta EE Version 9.1

This chapter describes in detail how to build and run several advanced examples that use CDI.

Building and Running the CDI Advanced Examples

The examples are in the `jakartaee-examples/tutorial/cdi/` directory. To build and run the examples, you will do the following.

1. Use NetBeans IDE or the Maven tool to compile, package, and deploy the example.
2. Run the example in a web browser.

See [intro:usingexamples::usingexamples:::_using_the_tutorial_examples], for basic information on installing, building, and running the examples.

The encoder Example: Using Alternatives

The `encoder` example shows how to use alternatives to choose between two beans at deployment time, as described in Using Alternatives in CDI Applications. The example includes an interface and two implementations of it, a managed bean, a Facelets page, and configuration files.

The source files are located in the `jakartaee-examples/tutorial/cdi/encoder/src/main/java/jakarta/tutorial/encoder/` directory.

The Coder Interface and Implementations

The `Coder` interface contains just one method, `codeString`, that takes two arguments: a string, and an integer value that specifies how the letters in the string should be transposed.

```
public interface Coder {

    public String codeString(String s, int tval);
}
```

The interface has two implementation classes, `CoderImpl` and `TestCoderImpl`. The implementation of `codeString` in `CoderImpl` shifts the string argument forward in the alphabet by the number of letters specified in the second argument; any characters that are not letters are left unchanged. (This simple shift code is known as a Caesar cipher because Julius Caesar reportedly used it to communicate with his generals.) The implementation in `TestCoderImpl` merely displays the values of the arguments. The `TestCoderImpl` implementation is annotated `@Alternative`:

```
import jakarta.enterprise.inject.Alternative;

@Alternative
public class TestCoderImpl implements Coder {

    @Override
    public String codeString(String s, int tval) {
        return ("input string is " + s + ", shift value is " + tval);
    }
}
```

The `beans.xml` file for the `encoder` example contains an `alternatives` element for the `TestCoderImpl` class, but by default the element is commented out:

```
<beans ...>
    <!--<alternatives>
        <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
    </alternatives>-->
</beans>
```

This means that by default, the `TestCoderImpl` class, annotated `@Alternative`, will not be used. Instead, the `CoderImpl` class will be used.

The encoder Facelets Page and Managed Bean

The simple Facelets page for the `encoder` example, `index.xhtml`, asks the user to enter the string and integer values and passes them to the managed bean, `CoderBean`, as `coderBean.inputString` and `coderBean.transVal`:

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html">
    <h:head>
        <h:outputStylesheet library="css" name="default.css"/>
        <title>String Encoder</title>
    </h:head>
    <h:body>
        <h2>String Encoder</h2>
        <p>Type a string and an integer, then click Encode.</p>
        <p>Depending on which alternative is enabled, the coder bean
            will either display the argument values or return a string that
```

```
            shifts the letters in the original string by the value you
            specify. The value must be between 0 and 26.</p>
        <h:form id="encodeit">
            <p><h:outputLabel value="Enter a string: " for="inputString"/>
                <h:inputText id="inputString"
                             value="#{coderBean.inputString}"/>
                <h:outputLabel value="Enter the number of letters to shift by: "
                               for="transVal"/>
                <h:inputText id="transVal" value="#{coderBean.transVal}"/></p>
            <p><h:commandButton value="Encode"
                                action="#{coderBean.encodeString()}"/></p>
            <p><h:outputLabel value="Result: " for="outputString"/>
                <h:outputText id="outputString"
                              value="#{coderBean.codedString}"
                              style="color:blue"/></p>
            <p><h:commandButton value="Reset"
                                action="#{coderBean.reset}"/></p>
        </h:form>
        ...
    </h:body>
</html>
```

When the user clicks the Encode button, the page invokes the managed bean's `encodeString` method and displays the result, `coderBean.codedString`, in blue. The page also has a Reset button that clears the fields.

The managed bean, `CoderBean`, is a `@RequestScoped` bean that declares its input and output properties. The `transVal` property has three Bean Validation constraints that enforce limits on the integer value, so that if the user enters an invalid value, a default error message appears on the Facelets page. The bean also injects an instance of the `Coder` interface:

```
@Named
@RequestScoped
public class CoderBean {

    private String inputString;
    private String codedString;
    @Max(26)
    @Min(0)
    @NotNull
    private int transVal;

    @Inject
    Coder coder;
    ...
}
```

In addition to simple getter and setter methods for the three properties, the bean defines the `encodeString` action method called by the Facelets page. This method sets the `codedString` property

to the value returned by a call to the `codeString` method of the `Coder` implementation:

```
public void encodeString() {
    setCodedString(coder.codeString(inputString, transVal));
}
```

Finally, the bean defines the `reset` method to empty the fields of the Facelets page:

```
public void reset() {
    setInputString("");
    setTransVal(0);
}
```

Running the encoder Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `encoder` application.

To Build, Package, and Deploy the encoder Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/cdi
   ```

4. Select the `encoder` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `encoder` project and select **Build**.

   This command builds and packages the application into a WAR file, `encoder.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the encoder Example Using NetBeans IDE

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/encoder
   ```

2. On the String Encoder page, enter a string and the number of letters to shift by, and then click Encode.

   The encoded string appears in blue on the Result line. For example, if you enter `Java` and `4`, the result is `Neze`.

3. Now, edit the `beans.xml` file to enable the alternative implementation of `Coder`.
    a. In the Projects tab, under the `encoder` project, expand the Web Pages node, then expand the WEB-INF node.
    b. Double-click the `beans.xml` file to open it.
    c. Remove the comment characters that surround the `alternatives` element, so that it looks like this:

    ```
    <alternatives>
        <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
    </alternatives>
    ```

    d. Save the file.
4. Right-click the `encoder` project and select Clean and Build.
5. In the web browser, reenter the URL to show the String Encoder page for the redeployed project:

    ```
    http://localhost:8080/encoder/
    ```

6. Enter a string and the number of letters to shift by, and then click Encode.

    This time, the Result line displays your arguments. For example, if you enter `Java` and `4`, the result is:

    ```
    Result: input string is Java, shift value is 4
    ```

To Build, Package, and Deploy the encoder Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

    ```
    jakartaee-examples/tutorial/cdi/encoder/
    ```

3. Enter the following command to deploy the application:

    ```
    mvn install
    ```

    This command builds and packages the application into a WAR file, `encoder.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the encoder Example Using Maven

1. In a web browser, enter the following URL:

```
http://localhost:8080/encoder/
```

The String Encoder page opens.

2. Enter a string and the number of letters to shift by, and then click Encode.

   The encoded string appears in blue on the Result line. For example, if you enter `Java` and `4`, the result is `Neze`.

3. Now, edit the `beans.xml` file to enable the alternative implementation of `Coder`.

   a. In a text editor, open the following file:

   ```
   jakartaee-examples/tutorial/cdi/encoder/src/main/webapp/WEB-INF/beans.xml
   ```

   b. Remove the comment characters that surround the `alternatives` element, so that it looks like this:

   ```
   <alternatives>
       <class>ee.jakarta.tutorial.encoder.TestCoderImpl</class>
   </alternatives>
   ```

   c. Save and close the file.

4. Enter the following command:

   ```
   mvn clean install
   ```

5. In the web browser, reenter the URL to show the String Encoder page for the redeployed project:

   ```
   http://localhost:8080/encoder
   ```

6. Enter a string and the number of letters to shift by, and then click Encode.

   This time, the Result line displays your arguments. For example, if you enter `Java` and `4`, the result is:

   ```
   Result: input string is Java, shift value is 4
   ```

The producermethods Example: Using a Producer Method to Choose a Bean Implementation

The `producermethods` example shows how to use a producer method to choose between two beans at runtime, as described in Using Producer Methods, Producer Fields, and Disposer Methods in CDI Applications. It is very similar to the `encoder` example described in The encoder Example: Using

Alternatives. The example includes the same interface and two implementations of it, a managed bean, a Facelets page, and configuration files. It also contains a qualifier type. When you run it, you do not need to edit the `beans.xml` file and redeploy the application to change its behavior.

Components of the producermethods Example

The components of `producermethods` are very much like those for `encoder`, with some significant differences.

Neither implementation of the `Coder` bean is annotated `@Alternative`, and there is no `beans.xml` file, because it is not needed.

The Facelets page and the managed bean, `CoderBean`, have an additional property, `coderType`, that allows the user to specify at runtime which implementation to use. In addition, the managed bean has a producer method that selects the implementation using a qualifier type, `@Chosen`.

The bean declares two constants that specify whether the coder type is the test implementation or the implementation that actually shifts letters:

```
private final static int TEST = 1;
private final static int SHIFT = 2;
private int coderType = SHIFT; // default value
```

The producer method, annotated with `@Produces` and `@Chosen` as well as `@RequestScoped` (so that it lasts only for the duration of a single request and response), returns one of the two implementations based on the `coderType` supplied by the user.

```
@Produces
@Chosen
@RequestScoped
public Coder getCoder() {

    switch (coderType) {
        case TEST:
            return new TestCoderImpl();
        case SHIFT:
            return new CoderImpl();
        default:
            return null;
    }
}
```

Finally, the managed bean injects the chosen implementation, specifying the same qualifier as that returned by the producer method to resolve ambiguities:

```
@Inject
@Chosen
@RequestScoped
```

```
Coder coder;
```

The Facelets page contains modified instructions and a pair of options whose selected value is assigned to the property `coderBean.coderType`:

```
    <h2>String Encoder</h2>
        <p>Select Test or Shift, type a string and an integer, then click
            Encode.</p>
        <p>If you select Test, the TestCoderImpl bean will display the
            argument values.</p>
        <p>If you select Shift, the CoderImpl bean will return a string that
            shifts the letters in the original string by the value you specify.
            The value must be between 0 and 26.</p>
        <h:form id="encodeit">
            <h:selectOneRadio id="coderType"
                              required="true"
                              value="#{coderBean.coderType}">
                <f:selectItem
                    itemValue="1"
                    itemLabel="Test"/>
                <f:selectItem
                    itemValue="2"
                    itemLabel="Shift Letters"/>
            </h:selectOneRadio>
            ...
```

Running the producermethods Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `producermethods` application.

To Build, Package, and Deploy the producermethods Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/cdi
   ```

4. Select the `producermethods` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `producermethods` project and select **Build**.

   This command builds and packages the application into a WAR file, `producermethods.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Build, Package, and Deploy the producermethods Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/cdi/producermethods/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `producermethods.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the producermethods Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/producermethods
   ```

2. On the String Encoder page, select either the Test or Shift Letters option, enter a string and the number of letters to shift by, and then click Encode.

   Depending on your selection, the Result line displays either the encoded string or the input values you specified.

The producerfields Example: Using Producer Fields to Generate Resources

The `producerfields` example, which allows you to create a to-do list, shows how to use a producer field to generate objects that can then be managed by the container. This example generates an `EntityManager` object, but resources such as JDBC connections and datasources can also be generated this way.

The `producerfields` example is the simplest possible entity example. It also contains a qualifier and a class that generates the entity manager. It also contains a single entity, a stateful session bean, a Facelets page, and a managed bean.

The source files are located in the `jakartaee-examples/tutorial/cdi/producerfields/src/main/java/jakarta/tutorial/producerfields/` directory.

The Producer Field for the producerfields Example

The most important component of the `producerfields` example is the smallest, the `db.UserDatabaseEntityManager` class, which isolates the generation of the `EntityManager` object so it can easily be used by other components in the application. The class uses a producer field to inject an `EntityManager` annotated with the `@UserDatabase` qualifier, also defined in the `db` package:

```
@Singleton
public class UserDatabaseEntityManager {

    @Produces
    @PersistenceContext
    @UserDatabase
    private EntityManager em;
    ...
}
```

The class does not explicitly produce a persistence unit field, but the application has a `persistence.xml` file that specifies a persistence unit. The class is annotated `jakarta.inject.Singleton` to specify that the injector should instantiate it only once.

The `db.UserDatabaseEntityManager` class also contains commented-out code that uses `create` and `close` methods to generate and remove the producer field:

```
/*
@PersistenceContext
private EntityManager em;

@Produces
@UserDatabase
public EntityManager create() {
    return em;
}
*/

public void close(@Disposes @UserDatabase EntityManager em) {
    em.close();
}
```

You can remove the comment indicators from this code and place them around the field declaration to test how the methods work. The behavior of the application is the same with either mechanism.

The advantage of producing the `EntityManager` in a separate class rather than simply injecting it into an enterprise bean is that the object can easily be reused in a typesafe way. Also, a more complex application can create multiple entity managers using multiple persistence units, and this mechanism isolates this code for easy maintenance, as in the following example:

```
@Singleton
public class JPAResourceProducer {
    @Produces
    @PersistenceUnit(unitName="pu3")
    @TestDatabase
    EntityManagerFactory customerDatabasePersistenceUnit;
```

```
    @Produces
    @PersistenceContext(unitName="pu3")
    @TestDatabase
    EntityManager customerDatabasePersistenceContext;

    @Produces
    @PersistenceUnit(unitName="pu4")
    @Documents
    EntityManagerFactory customerDatabasePersistenceUnit;

    @Produces
    @PersistenceContext(unitName="pu4")
    @Documents
    EntityManager docDatabaseEntityManager;
}
```

The `EntityManagerFactory` declarations also allow applications to use an application-managed entity manager.

The producerfields Entity and Session Bean

The `producerfields` example contains a simple entity class, `entity.ToDo`, and a stateful session bean, `ejb.RequestBean`, that uses it.

The entity class contains three fields: an autogenerated `id` field, a string specifying the task, and a timestamp.

```
@Entity
public class ToDo implements Serializable {

    ...
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    protected String taskText;
    protected LocalDateTime timeCreated;

    public ToDo() {
    }

    public ToDo(Long id, String taskText, LocalDateTime timeCreated) {
        this.id = id;
        this.taskText = taskText;
        this.timeCreated = timeCreated;
    }
    ...
}
```

The remainder of the `ToDo` class contains the usual getters, setters, and other entity methods.

The `RequestBean` class injects the `EntityManager` generated by the producer method, annotated with the `@UserDatabase` qualifier:

```
@ConversationScoped
@Stateful
public class RequestBean {

    @Inject
    @UserDatabase
    EntityManager em;
}
```

It then defines two methods, one that creates and persists a single `ToDo` list item, and another that retrieves all the `ToDo` items created so far by creating a query:

```
public ToDo createToDo(String inputString) {
    try {
        ToDo toDo = new ToDo();
        toDo.setTaskText(inputString);
        toDo.setTimeCreated(LocalDateTime.now());
        em.persist(toDo);
        return toDo;
    } catch (Exception e) {
        throw new EJBException(e);
    }
}

public List<ToDo> getToDos() {
    try {
            List<ToDo> toDos =
                (List<ToDo>) em.createQuery(
                "SELECT t FROM ToDo t ORDER BY t.timeCreated")
                .getResultList();
        return toDos;
    } catch (Exception e) {
        throw new EJBException(e.getMessage());
    }
}
```

The producerfields Facelets Pages and Managed Bean

The `producerfields` example has two Facelets pages, `index.xhtml` and `todolist.xhtml`. The simple form on the `index.xhtml` page asks the user only for the task. When the user clicks the Submit button, the `listBean.createTask` method is called. When the user clicks the Show Items button, the action specifies that the `todolist.xhtml` file should be displayed:

```
    <h:body>
        <h2>To Do List</h2>
```

```
        <p>Enter a task to be completed.</p>
        <h:form id="todolist">
            <p><h:outputLabel value="Enter a string: " for="inputString"/>
                <h:inputText id="inputString"
                             value="#{listBean.inputString}"/></p>
            <p><h:commandButton value="Submit"
                                action="#{listBean.createTask()}"/></p>
            <p><h:commandButton value="Show Items"
                                action="todolist"/></p>
        </h:form>
        ...
    </h:body>
```

The managed bean, `web.ListBean`, injects the `ejb.RequestBean` session bean. It declares the `entity.ToDo` entity and a list of the entity along with the input string that it passes to the session bean. The `inputString` is annotated with the `@NotNull` Bean Validation constraint, so an attempt to submit an empty string results in an error.

```
@Named
@ConversationScoped
public class ListBean implements Serializable {

    ...
    @EJB
    private RequestBean request;
    @NotNull
    private String inputString;
    private ToDo toDo;
    private List<ToDo> toDos;
    ...
}
```

The `createTask` method called by the Submit button calls the `createToDo` method of `RequestBean`:

```
public void createTask() {
    this.toDo = request.createToDo(inputString);
}
```

The `getToDos` method, which is called by the `todolist.xhtml` page, calls the `getToDos` method of `RequestBean`:

```
public List<ToDo> getToDos() {
    return request.getToDos();
}
```

To force the Facelets page to recognize an empty string as a null value and return an error, the

`web.xml` file sets the context parameter `jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to `true`:

```
<context-param>
  <param-name>jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-
name>
  <param-value>true</param-value>
</context-param>
```

The `todolist.xhtml` page is a little more complicated than the `index.html` page. It contains a `dataTable` element that displays the contents of the `ToDo` list. The body of the page looks like this:

```
    <body>
        <h2>To Do List</h2>
        <h:form id="showlist">
            <h:dataTable var="toDo"
                         value="#{listBean.toDos}"
                         rules="all"
                         border="1"
                         cellpadding="5">
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Time Stamp" />
                    </f:facet>
                    <h:outputText value="#{toDo.timeCreated}" />
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Task" />
                    </f:facet>
                    <h:outputText value="#{toDo.taskText}" />
                </h:column>
            </h:dataTable>
            <p><h:commandButton id="back" value="Back" action="index" /></p>
        </h:form>
    </body>
```

The value of the `dataTable` is `listBean.toDos`, the list returned by the managed bean's `getToDos` method, which in turn calls the session bean's `getToDos` method. Each row of the table displays the `timeCreated` and `taskText` fields of the individual task. Finally, a Back button returns the user to the `index.xhtml` page.

Running the producerfields Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `producerfields` application.

To Build, Package, and Deploy the producerfields Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. From the **File** menu, choose **Open Project**.
4. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/cdi
   ```

5. Select the `producerfields` folder.
6. Click **Open Project**.
7. In the **Projects** tab, right-click the `producerfields` project and select **Build**.

   This command builds and packages the application into a WAR file, `producerfields.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Build, Package, and Deploy the producerfields Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/cdi/producerfields/
   ```

4. Enter the following command to deploy the application:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `producerfields.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the producerfields Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/producerfields
   ```

2. On the Create To Do List page, enter a string in the field and click Submit.

   You can enter additional strings and click Submit to create a task list with multiple items.

3. Click Show Items.

The To Do List page opens, showing the timestamp and text for each item you created.

4. Click Back to return to the Create To Do List page.

   On this page, you can enter more items in the list.

The billpayment Example: Using Events and Interceptors

The `billpayment` example shows how to use both events and interceptors.

The source files are located in the `jakartaee-examples/tutorial/cdi/billpayment/src/main/java/jakarta/tutorial/billpayment/` directory.

Overview of the billpayment Example

The example simulates paying an amount using a debit card or credit card. When the user chooses a payment method, the managed bean creates an appropriate event, supplies its payload, and fires it. A simple event listener handles the event using observer methods.

The example also defines an interceptor that is set on a class and on two methods of another class.

The PaymentEvent Event Class

The event class, `event.PaymentEvent`, is a simple bean class that contains a no-argument constructor. It also has a `toString` method and getter and setter methods for the payload components: a `String` for the payment type, a `BigDecimal` for the payment amount, and a `Date` for the timestamp.

```
public class PaymentEvent implements Serializable {

    ...
    public String paymentType;
    public BigDecimal value;
    public Date datetime;

    public PaymentEvent() {
    }

    @Override
    public String toString() {
        return this.paymentType
                + " = $" + this.value.toString()
                + " at " + this.datetime.toString();
    }
    ...
}
```

The event class is a simple bean that is instantiated by the managed bean using `new` and then populated. For this reason, the CDI container cannot intercept the creation of the bean, and hence it cannot allow interception of its getter and setter methods.

The PaymentHandler Event Listener

The event listener, `listener.PaymentHandler`, contains two observer methods, one for each of the two event types:

```
@Logged
@SessionScoped
public class PaymentHandler implements Serializable {

    ...
    public void creditPayment(@Observes @Credit PaymentEvent event) {
        logger.log(Level.INFO, "PaymentHandler - Credit Handler: {0}",
                event.toString());

        // call a specific Credit handler class...
    }

    public void debitPayment(@Observes @Debit PaymentEvent event) {
        logger.log(Level.INFO, "PaymentHandler - Debit Handler: {0}",
                event.toString());

        // call a specific Debit handler class...
    }
}
```

Each observer method takes as an argument the event, annotated with `@Observes` and with the qualifier for the type of payment. In a real application, the observer methods would pass the event information on to another component that would perform business logic on the payment.

The qualifiers are defined in the `payment` package, described in The billpayment Facelets Pages and Managed Bean.

The `PaymentHandler` bean is annotated `@Logged` so that all its methods can be intercepted.

The billpayment Facelets Pages and Managed Bean

The `billpayment` example contains two Facelets pages, `index.xhtml` and the very simple `response.xhtml`. The body of `index.xhtml` looks like this:

```
    <h:body>
        <h3>Bill Payment Options</h3>
        <p>Enter an amount, select Debit Card or Credit Card,
            then click Pay.</p>
        <h:form>
            <p>
            <h:outputLabel value="Amount: $" for="amt"/>
            <h:inputText id="amt" value="#{paymentBean.value}"
                         required="true"
                         requiredMessage="An amount is required."
                         maxlength="15" />
```

```
            </p>
            <h:outputLabel value="Options:" for="opt"/>
            <h:selectOneRadio id="opt" value="#{paymentBean.paymentOption}">
                <f:selectItem id="debit" itemLabel="Debit Card"
                              itemValue="1"/>
                <f:selectItem id="credit" itemLabel="Credit Card"
                              itemValue="2" />
            </h:selectOneRadio>
            <p><h:commandButton id="submit" value="Pay"
                                action="#{paymentBean.pay}" /></p>
            <p><h:commandButton value="Reset"
                                action="#{paymentBean.reset}" /></p>
        </h:form>
        ...
    </h:body>
```

The input field takes a payment amount, passed to `paymentBean.value`. Two options ask the user to select a Debit Card or Credit Card payment, passing the integer value to `paymentBean.paymentOption`. Finally, the Pay command button's action is set to the method `paymentBean.pay`, and the Reset button's action is set to the `paymentBean.reset` method.

The `payment.PaymentBean` managed bean uses qualifiers to differentiate between the two kinds of payment event:

```
@Named
@SessionScoped
public class PaymentBean implements Serializable {

   ...
    @Inject
    @Credit
    Event<PaymentEvent> creditEvent;

    @Inject
    @Debit
    Event<PaymentEvent> debitEvent;
    ...
}
```

The qualifiers, `@Credit` and `@Debit`, are defined in the `payment` package along with `PaymentBean`.

Next, the `PaymentBean` defines the properties it obtains from the Facelets page and will pass on to the event:

```
public static final int DEBIT = 1;
public static final int CREDIT = 2;
private int paymentOption = DEBIT;
```

```
@Digits(integer = 10, fraction = 2, message = "Invalid value")
private BigDecimal value;

private Date datetime;
```

The `paymentOption` value is an integer passed in from the option component; the default value is `DEBIT`. The `value` is a `BigDecimal` with a Bean Validation constraint that enforces a currency value with a maximum number of digits. The timestamp for the event, `datetime`, is a `Date` object initialized when the `pay` method is called.

The `pay` method of the bean first sets the timestamp for this payment event. It then creates and populates the event payload, using the constructor for the `PaymentEvent` and calling the event's setter methods, using the bean properties as arguments. It then fires the event.

```
@Logged
public String pay() {
    this.setDatetime(LocalDateTime.now());
    switch (paymentOption) {
        case DEBIT:
            PaymentEvent debitPayload = new PaymentEvent();
            debitPayload.setPaymentType("Debit");
            debitPayload.setValue(value);
            debitPayload.setDatetime(datetime);
            debitEvent.fire(debitPayload);
            break;
        case CREDIT:
            PaymentEvent creditPayload = new PaymentEvent();
            creditPayload.setPaymentType("Credit");
            creditPayload.setValue(value);
            creditPayload.setDatetime(datetime);
            creditEvent.fire(creditPayload);
            break;
        default:
            logger.severe("Invalid payment option!");
    }
    return "response";
}
```

The `pay` method returns the page to which the action is redirected, `response.xhtml`.

The `PaymentBean` class also contains a `reset` method that empties the value field on the `index.xhtml` page and sets the payment option to the default:

```
@Logged
public void reset() {
    setPaymentOption(DEBIT);
    setValue(BigDecimal.ZERO);
}
```

In this bean, only the `pay` and `reset` methods are intercepted.

The `response.xhtml` page displays the amount paid. It uses a `rendered` expression to display the payment method:

```
    <h:body>
        <h:form>
            <h2>Bill Payment: Result</h2>
            <h3>Amount Paid with
                <h:outputText id="debit" value="Debit Card: "
                              rendered="#{paymentBean.paymentOption eq 1}" />
                <h:outputText id="credit" value="Credit Card: "
                              rendered="#{paymentBean.paymentOption eq 2}" />
                <h:outputText id="result" value="#{paymentBean.value}">
                    <f:convertNumber type="currency"/>
                </h:outputText>
            </h3>
            <p><h:commandButton id="back" value="Back" action="index" /></p>
        </h:form>
    </h:body>
```

The LoggedInterceptor Interceptor Class

The interceptor class, `LoggedInterceptor`, and its interceptor binding, `Logged`, are both defined in the `interceptor` package. The `Logged` interceptor binding is defined as follows:

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {
}
```

The `LoggedInterceptor` class looks like this:

```
@Logged
@Interceptor
public class LoggedInterceptor implements Serializable {

    ...

    public LoggedInterceptor() {
    }

    @AroundInvoke
    public Object logMethodEntry(InvocationContext invocationContext)
            throws Exception {
        System.out.println("Entering method: "
```

```
                + invocationContext.getMethod().getName() + " in class "
                + invocationContext.getMethod().getDeclaringClass().getName());

        return invocationContext.proceed();
    }
}
```

The class is annotated with both the `@Logged` and the `@Interceptor` annotations. The `@AroundInvoke` method, `logMethodEntry`, takes the required `InvocationContext` argument and calls the required `proceed` method. When a method is intercepted, `logMethodEntry` displays the name of the method being invoked as well as its class.

To enable the interceptor, the `beans.xml` file defines it as follows:

```
<interceptors>
    <class>ee.jakarta.tutorial.billpayment.interceptor.LoggedInterceptor</class>
</interceptors>
```

In this application, the `PaymentEvent` and `PaymentHandler` classes are annotated `@Logged`, so all their methods are intercepted. In `PaymentBean`, only the `pay` and `reset` methods are annotated `@Logged`, so only those methods are intercepted.

Running the billpayment Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `billpayment` application.

To Build, Package, and Deploy the billpayment Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/cdi
   ```

4. Select the `billpayment` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `billpayment` project and select **Build**.

   This command builds and packages the application into a WAR file, `billpayment.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Build, Package, and Deploy the billpayment Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

```
jakartaee-examples/tutorial/cdi/billpayment/
```

3. Enter the following command to deploy the application:

```
mvn install
```

This command builds and packages the application into a WAR file, `billpayment.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Run the billpayment Example

1. In a web browser, enter the following URL:

```
http://localhost:8080/billpayment
```

2. On the Bill Payment Options page, enter a value in the Amount field.

The amount can contain up to 10 digits and include up to two decimal places. For example:

```
9876.54
```

3. Select Debit Card or Credit Card and click Pay.

The Bill Payment: Result page opens, displaying the amount paid and the method of payment:

```
Amount Paid with Credit Card: $9,876.34
```

4. Click Back to return to the Bill Payment Options page.

You can also click Reset to return to the initial page values.

5. Examine the server log output.

In NetBeans IDE, the output is visible in the GlassFish Server Output tab. Otherwise, view `domain-dir/logs/server.log`.

The output from each interceptor appears in the log, followed by the additional logger output defined by the constructor and methods:

```
INFO: Entering method: pay in class billpayment.payment.PaymentBean
INFO: PaymentHandler created.
INFO: Entering method: debitPayment in class billpayment.listener.PaymentHandler
INFO: PaymentHandler - Debit Handler: Debit = $1234.56 at Tue Dec 14 14:50:28 EST
2010
```

The decorators Example: Decorating a Bean

The `decorators` example, which is yet another variation on the `encoder` example, shows how to use a decorator to implement additional business logic for a bean.

The source files are located in the `jakartaee-examples/tutorial/cdi/decorators/src/main/java/jakarta/tutorial/decorators/` directory.

Overview of the decorators Example

Instead of having the user choose between two alternative implementations of an interface at deployment time or runtime, a decorator adds some additional logic to a single implementation of the interface.

The example includes an interface, an implementation of it, a decorator, an interceptor, a managed bean, a Facelets page, and configuration files.

Components of the decorators Example

The `decorators` example is very similar to the `encoder` example described in The encoder Example: Using Alternatives. Instead of providing two implementations of the `Coder` interface, however, this example provides only the `CoderImpl` class. The decorator class, `CoderDecorator`, rather than simply return the coded string, displays the input and output strings' values and length.

The `CoderDecorator` class, like `CoderImpl`, implements the business method of the `Coder` interface, `codeString`:

```
@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject
    @Delegate
    @Any
    Coder coder;

    public String codeString(String s, int tval) {
        int len = s.length();

        return "\"" + s + "\" becomes " + "\"" + coder.codeString(s, tval)
                + "\", " + len + " characters in length";
    }
}
```

The decorator's `codeString` method calls the delegate object's `codeString` method to perform the actual encoding.

The `decorators` example includes the `Logged` interceptor binding and `LoggedInterceptor` class from the `billpayment` example. For this example, the interceptor is set on the `CoderBean.encodeString` method and the `CoderImpl.codeString` method. The interceptor code is unchanged; interceptors are usually reusable for different applications.

Except for the interceptor annotations, the `CoderBean` and `CoderImpl` classes are identical to the versions in the `encoder` example.

The `beans.xml` file specifies both the decorator and the interceptor:

```
<decorators>
    <class>ee.jakarta.tutorial.decorators.CoderDecorator</class>
</decorators>
<interceptors>
    <class>ee.jakarta.tutorial.decorators.LoggedInterceptor</class>
</interceptors>
```

Running the decorators Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `decorators` application.

To Build, Package, and Deploy the decorators Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/cdi
   ```

4. Select the `decorators` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `decorators` project and select **Build**.

   This command builds and packages the application into a WAR file, `decorators.war`, located in the `target` directory, and then deploys it to GlassFish Server.

To Build, Package, and Deploy the decorators Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/cdi/decorators/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `decorators.war`, located in

the `target` directory, and then deploys it to GlassFish Server.

To Run the decorators Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/decorators
   ```

2. On the Decorated String Encoder page, enter a string and the number of letters to shift by, and then click Encode.

   The output from the decorator method appears in blue on the Result line. For example, if you entered `Java` and `4`, you would see the following:

   ```
   "Java" becomes "Neze", 4 characters in length
   ```

3. Examine the server log output.

   In NetBeans IDE, the output is visible in the GlassFish Server Output tab. Otherwise, view `domain-dir/logs/server.log`.

   The output from the interceptors appears:

   ```
   INFO: Entering method: encodeString in class
   ee.jakarta.tutorial.decorators.CoderBean
   INFO: Entering method: codeString in class ee.jakarta.tutorial.decorators.CoderImpl
   ```

Jakarta Validation

Introduction to Jakarta Bean Validation



This section was last updated for Jakarta EE Version 9.1

This chapter describes Jakarta Bean Validation available as part of the Jakarta EE platform and the facility for validating objects, object members, methods, and constructors.

Overview of Jakarta Bean Validation

Validating input received from the user to maintain data integrity is an important part of application logic. Validation of data can take place at different layers in even the simplest of applications, as shown in Developing a Simple Facelets Application: The guessnumber-jsf Example Application. The `guessnumber-faces` example application validates the user input (in the `h:inputText` tag) for numerical data at the presentation layer and for a valid range of numbers at the business layer.

Jakarta Bean Validation provides a facility for validating objects, object members, methods, and

constructors. In Jakarta EE environments, Jakarta Bean Validation integrates with Jakarta EE containers and services to allow developers to easily define and enforce validation constraints. Jakarta Bean Validation is available as part of the Jakarta EE platform.

Using Jakarta Bean Validation Constraints

The Jakarta Bean Validation model is supported by constraints in the form of annotations placed on a field, method, or class of a JavaBeans component, such as a managed bean.

Constraints can be built in or user defined. User-defined constraints are called custom constraints. Several built-in constraints are available in the `jakarta.validation.constraints` package. Built-In Jakarta Bean Validation Constraints lists all the built-in constraints. See Creating Custom Constraints for information on creating custom constraints.

Built-In Jakarta Bean Validation Constraints

Constraint	Description	Example
`@AssertFalse`	The value of the field or property must be `false`.	`@AssertFalse boolean isUnsupported;`
`@AssertTrue`	The value of the field or property must be `true`.	`@AssertTrue boolean isActive;`
`@DecimalMax`	The value of the field or property must be a decimal value lower than or equal to the number in the value element.	`@DecimalMax("30.00") BigDecimal discount;`
`@DecimalMin`	The value of the field or property must be a decimal value greater than or equal to the number in the value element.	`@DecimalMin("5.00") BigDecimal discount;`

Constraint	Description	Example
`@Digits`	The value of the field or property must be a number within a specified range. The `integer` element specifies the maximum integral digits for the number, and the `fraction` element specifies the maximum fractional digits for the number.	`@Digits(integer=6, fraction=2) BigDecimal price;`
`@Email`	The value of the field or property must be a valid email address.	`@Email String emailaddress;`
`@Future`	The value of the field or property must be a date in the future.	`@Future Date eventDate;`
`@FutureOrPresent`	The value of the field or property must be a date or time in present or future.	`@FutureOrPresent Time travelTime;`
`@Max`	The value of the field or property must be an integer value lower than or equal to the number in the value element.	`@Max(10) int quantity;`
`@Min`	The value of the field or property must be an integer value greater than or equal to the number in the value element.	`@Min(5) int quantity;`
`@Negative`	The value of the field or property must be a negative number.	`@Negative int basementFloor;`

Constraint	Description	Example
`@NegativeOrZero`	The value of the field or property must be negative or zero.	`@NegativeOrZero int debtValue;`
`@NotBlank`	The value of the field or property must contain atleast one non-white space character.	`@NotBlank String message;`
`@NotEmpty`	The value of the field or property must not be empty. The length of the characters or array, and the size of a collection or map are evaluated.	`@NotEmpty String message;`
`@NotNull`	The value of the field or property must not be null.	`@NotNull String username;`
`@Null`	The value of the field or property must be null.	`@Null String unusedString;`
`@Past`	The value of the field or property must be a date in the past.	`@Past Date birthday;`
`@PastOrPresent`	The value of the field or property must be a date or time in the past or present.	`@PastOrPresent Date travelDate;`

Constraint	Description	Example
`@Pattern`	The value of the field or property must match the regular expression defined in the `regexp` element.	`@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;`
`@Positive`	The value of the field or property must be a positive number.	`@Positive BigDecimal area;`
`@PositiveOrZero`	The value of the field or property must be a positive number or zero.	`@PositiveOrZero int totalGoals;`
`@Size`	The size of the field or property is evaluated and must match the specified boundaries. If the field or property is a `String`, the size of the string is evaluated. If the field or property is a `Collection`, the size of the `Collection` is evaluated. If the field or property is a `Map`, the size of the `Map` is evaluated. If the field or property is an array, the size of the array is evaluated. Use one of the optional `max` or `min` elements to specify the boundaries.	`@Size(min=2, max=240) String briefMessage;`

In the following example, a constraint is placed on a field using the built-in `@NotNull` constraint:

```
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
    ...
}
```

You can also place more than one constraint on a single JavaBeans component object. For example, you can place an additional constraint for size of field on the `firstname` and the `lastname` fields:

```
public class Name {
```

```
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
    ...
}
```

The following example shows a method with a user-defined constraint that checks user-defined constraint that checks for a predefined phone number pattern, such as a country specific phone number:

```
@USPhoneNumber
public String getPhone() {
    return phone;
}
```

For a built-in constraint, a default implementation is available. A user-defined or custom constraint needs a validation implementation. In the preceding example, the `@USPhoneNumber` custom constraint needs an implementation class.

Repeating Annotations

From Bean Validation 2.0 onwards, you can specify the same constraint several times on a validation target using repeating annotation:

```
public class Account {

    @Max (value = 2000, groups = Default.class, message = "max.value")
    @Max (value = 5000, groups = GoldCustomer.class, message = "max.value")
    private long withdrawalAmount;
}
```

All in-built constraints from `jakarta.validation.constraints` package support repeatable annotations. Similarly, custom constraints can use `@Repeatable` annotation. In the following sample, depending on whether the group is `PeakHour` or `NonPeakHour`, the car instance is validated as either two passengers or three passengers based car, and then listed as eligible in the car pool lane:

```
/**
 * Validate whether a car is eligible for car pool lane
 */
@Documented
@Constraint(validatedBy = CarPoolValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
```

```
@Repeatable(List.class)
public @interface CarPool {

    String message() default "{CarPool.message}";

    Class<?>[] groups() default {};

    int value();

    Class<? extends Payload>[] payload() default {};

    /**
     * Defines several @CarPool annotations on the same element
     * @see (@link CarPool}
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        CarPool[] value();
    }
}
public class Car{

  private String registrationNumber;

  @CarPool(value = 2, group = NonPeakHour.class)
  @CarPool(value = 3, group = {Default.class, PeakHour.class})
  private int totalPassengers;
}
```

Any validation failures are gracefully handled and can be displayed by the `h:messages` tag.

Any managed bean that contains Bean Validation annotations automatically gets validation constraints placed on the fields on a Jakarta Faces application's web pages.

For more information on using validation constraints, see the following:

- [beanvalidation:bean-validation-advanced::bean-validation-advanced:::_bean_validation_advanced_topics]
- Validating Resource Data with Bean Validation
- Validating Persistent Fields and Properties

Validating Null and Empty Strings

The Java programming language distinguishes between null and empty strings. An empty string is a string instance of zero length, whereas a null string has no value at all.

An empty string is represented as `""`. It is a character sequence of zero characters. A null string is represented by `null`. It can be described as the absence of a string instance.

Managed bean elements represented as a Jakarta Faces text component such as `inputText` are initialized with the value of the empty string by the Jakarta Faces implementation. Validating these strings can be an issue when user input for such fields is not required. Consider the following example, in which the string `testString` is a bean variable that will be set using input entered by the user. In this case, the user input for the field is not required.

```
if (testString==null) {
    doSomething();
} else {
    doAnotherThing();
}
```

By default, the `doAnotherThing` method is called even when the user enters no data, because the `testString` element has been initialized with the value of an empty string.

In order for the Bean Validation model to work as intended, you must set the context parameter `jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to `true` in the web deployment descriptor file, `web.xml`:

```
<context-param>
    <param-name>jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-
name>
    <param-value>true</param-value>
</context-param>
```

This parameter enables the Jakarta Faces implementation to treat empty strings as null.

Suppose, on the other hand, that you have a `@NotNull` constraint on an element, meaning that input is required. In this case, an empty string will pass this validation constraint. However, if you set the context parameter `jakarta.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to `true`, the value of the managed bean attribute is passed to the Jakarta Bean Validation runtime as a null value, causing the `@NotNull` constraint to fail.

Validating Constructors and Methods

Jakarta Bean Validation constraints may be placed on the parameters of nonstatic methods and constructors and on the return values of nonstatic methods. Static methods and constructors will not be validated.

```
public class Employee {
...
  public Employee (@NotNull String name) { ... }

  public void setSalary(
      @NotNull
      @Digits(integer=6, fraction=2) BigDecimal salary,
      @NotNull
```

```
      @ValidCurrency
      String currencyType) {
    ...
  }
...
}
```

In this example, the `Employee` class has a constructor constraint requiring a name and has two sets of method parameter constraints. The amount of the salary for the employee must not be null, cannot be greater than six digits to the left of the decimal point, and cannot have more than two digits to the right of the decimal place. The currency type must not be null and is validated using a custom constraint.

If you add method constraints to classes in an object hierarchy, special care must be taken to avoid unintended behavior by subtypes. See Using Method Constraints in Type Hierarchies for more information.

Cross-Parameter Constraints

Constraints that apply to multiple parameters are called cross-parameter constraints, and may be applied at the method or constructor level.

```
@ConsistentPhoneParameters
@NotNull
public Employee (String name, String officePhone, String mobilePhone) {
  ...
}
```

In this example, a custom cross-parameter constraint, `@ConsistentPhoneParameters`, validates that the format of the phone numbers passed into the constructor match. The `@NotNull` constraint applies to all the parameters in the constructor.



Cross-parameter constraint annotations are applied directly to the method or constructor. Return value constraints are also applied directly to the method or constructor. To avoid confusion as to where the constraint applies, parameter or return value, choose a name for any custom constraints that identifies where the constraint applies. For instance, the preceding example applies a custom constraint, `@ConsistentPhoneParameters`, that indicates that it applies to the parameters of the method or constructor.

When you create a custom constraint that applies to both method parameters and return values, the `validationAppliesTo` element of the constraint annotation may be set to `ConstraintTarget.RETURN_VALUE` or `ConstraintTarget.PARAMETERS` to explicitly set the target of the validation constraint.

Validating Type Arguments of Parameterized Types

From Bean Validation 2.0 onwards, you can apply constraints to the type arguments of

parameterized types. For example: `List<@NotNull Long> numbers;` Constraints can be applied to elements of container types such as `List`, `Map`, `Optional`, and others.

```
List<@Email String> emails;
public Map<@NotNull String, @USPhoneNumber String> getAddressesByType() { }
```

In this sample, `@Email` is an in-built constraint supported by Bean Validation, and `@USPhoneNumber` is a user-defined constraint. See Using the Built-In Constraints to Make a New Constraint.

`@USPhoneNumber` has `ElementType.TYPE_USE` as one of its `@Target`, and therefore it is possible to use `@USPhoneNumber` constraint for validating type arguments of parameterized types.

Identifying Parameter Constraint Violations

If a `ConstraintViolationException` occurs during a method call, the Bean Validation runtime returns a parameter index to identify which parameter caused the constraint violation. The parameter index is in the form `arg`*`PARAMETER_INDEX`*, where *PARAMETER_INDEX* is an integer that starts at 0 for the first parameter of the method or constructor.

Adding Constraints to Method Return Values

To validate the return value for a method, you can apply constraints directly to the method or constructor declaration.

```
@NotNull
public Employee getEmployee() { ... }
```

Cross-parameter constraints are also applied at the method level. Custom constraints that could be applied to both the return value and the method parameters have an ambiguous constraint target. To avoid this ambiguity, add a `validationAppliesTo` element to the constraint annotation definition with the default set to either `ConstraintTarget.RETURN_VALUE` or `ConstraintTarget.PARAMETERS` to explicitly set the target of the validation constraint.

```
@Manager(validationAppliesTo=ConstraintTarget.RETURN_VALUE)
public Employee getManager(Employee employee) { ... }
```

See Removing Ambiguity in Constraint Targets for more information.

Further Information about Jakarta Bean Validation

For more information on Jakarta Bean Validation, see

- Jakarta Bean Validation 3.0 Specification:
  https://jakarta.ee/specifications/bean-validation/3.0/
- Bean Validation Specification website:
  https://beanvalidation.org/

Bean Validation: Advanced Topics



This section was last updated for Jakarta EE Version 9.1

This chapter describes how to create custom constraints, custom validator messages, and constraint groups using the Jakarta Bean Validation (Bean Validation).

Creating Custom Constraints

Jakarta Bean Validation defines annotations, interfaces, and classes to allow developers to create custom constraints.

Using the Built-In Constraints to Make a New Constraint

Jakarta Bean Validation includes several built-in constraints that can be combined to create new, reusable constraints. This can simplify constraint definition by allowing developers to define a custom constraint made up of several built-in constraints that may then be applied to component attributes with a single annotation.

```
@Pattern.List({
  /* A number of format +1-NNN-NNN-NNNN */
  @Pattern(regexp = "\\+1-\\d{3}-\\d{3}-\\d{4}")
})
@Constraint(validatedBy = {})
@Documented
@Target({ElementType.METHOD,
    ElementType.FIELD,
    ElementType.ANNOTATION_TYPE,
    ElementType.CONSTRUCTOR,
    ElementType.PARAMETER,
    ElementType.TYPE_USE})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(List.class)
public @interface USPhoneNumber {

  String message() default "Not a valid US Phone Number";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

  @Target({ElementType.METHOD,
     ElementType.FIELD,
     ElementType.ANNOTATION_TYPE,
     ElementType.CONSTRUCTOR,
     ElementType.PARAMETER,
     ElementType.TYPE_USE })
  @Retention(RetentionPolicy.RUNTIME)
  @Documented
  @interface List {
```

```
  USPhoneNumber[] value();

  }
}
```

You can also implement a `Constraint Validator` to validate the constraint `@USPhoneNumber`. For more information about using `Constraint Validator`, see `jakarta.validation.ConstraintValidator`.

```
@USPhoneNumber
protected String phone;
```

Removing Ambiguity in Constraint Targets

Custom constraints that can be applied to both return values and method parameters require a `validationAppliesTo` element to identify the target of the constraint.

```
@Constraint(validatedBy=MyConstraintValidator.class)
@Target({ METHOD, FIELD, TYPE, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface MyConstraint {
  String message() default "{com.example.constraint.MyConstraint.message}";
  Class<?>[] groups() default {};
  ConstraintTarget validationAppliesTo() default ConstraintTarget.PARAMETERS;
...
}
```

This constraint sets the `validationAppliesTo` target by default to the method parameters.

```
@MyConstraint(validationAppliesTo=ConstraintTarget.RETURN_TYPE)
public String doSomething(String param1, String param2) { ... }
```

In the preceding example, the target is set to the return value of the method.

Implementing Temporal Constraints Using ClockProvider

From Bean Validation 2.0 onwards, a `Clock` instance is available for validator implementations to validate any temporal date or time based constraints.

```
ValidatorFactory validatorFactory = Validation.buildDefaultValidatorFactory();
ClockProvider clockProvider = validatorFactory.getClockProvider();
java.time.Clock Clock = clockProvider.getClock();
```

You can also register a custom `ClockProvider` with a `ValidatorFactory`:

```
//Register a custom clock provider implementation with validator factory
```

```
ValidatorFactory factory = Validation
       .byDefaultProvider().configure()
          .clockProvider( new CustomClockProvider() )
          .buildValidatorFactory();

//Retrieve and use the custom Clock Provider and Clock in the Validator implementation
public class CustomConstraintValidator implements
ConstraintValidator<CustomConstraint, Object> {

    public boolean isValid(Object value, ConstraintValidatorContext context){
        java.time.Clock clock = context.getClockProvider().getClock();
        ...
        ...

    }
}
```

See ClockProvider in https://jakarta.ee/specifications/platform/9/apidocs/.

Custom Constraints

Consider an employee in a firm located in U.S.A. When you register the phone number of an employee or modify the phone number, the phone number needs to be validated to ensure that the phone number conforms to US phone number pattern.

```
public class Employee extends Person {

  @USPhoneNumber
  protected String phone;

  public Employee(String name, String phone, int age){
    super(name, age);
    this.phone = phone;
  }

  public String getPhone() {
    return phone;
  }

  public void setPhone(String phone) {
    this.phone = phone;
  }
}
```

The constraint definition `@USPhoneNumber` is define in the sample listed under Using the Built-In Constraints to Make a New Constraint. In the sample, another constraint `@Pattern` is used to validate the phone number.

Using In-Built Value Extractors in Custom Containers

Cascading validation

Bean Validation supports cascading validation for various entities. You can specify `@Valid` on a member of the object that is validated to ensure that the member is also validated in a cascading fashion. You can validate type arguments, for example, parameterized types and its members if the members have the specified `@Valid` annotation.

```
public class Department {
    private List<@Valid Employee> employeesList;
}
```

By specifying `@Valid` on a parameterized type, when an instance of `Department` is validated, all elements such as `Employee` in the `employeesList` are also validated. In this example, each employee's "phone" is validated against the constraint `@USPhoneNumber`.

For more information see https://jakarta.ee/specifications/platform/9/apidocs/

Value Extractor

While validating the object or the object graph, it may be necessary to validate the constraints in the parameterized types of a container as well. To validate the elements of the container, the validator must extract the values of these elements in the container. For example, in order to validate the element values of `List` against one or more constraints such as `List<@NotOnVacation Employee>` or to apply cascading validation to `List<@Valid Employee>`, you need a value extractor for the container `List`.

Jakarta Bean validation provides in-built value extractors for most commonly used container types such as List, Iterable, and others. However, it is also possible to implement and register value-extractor implementations for custom container types or override the in-built value-extractor implementations.

Consider a Statistics Calculator for a group of `Person` entity and `Employee` is one of the sub-type of the entity `Person`.

```
public class StatsCalculator<T extends Person> {

  /* Cascading validation as well as @NotNull constraint */
  private List<@NotNull @Valid T> members = new ArrayList<T>();


  public void addMember(T member) {
    members.add(member);
  }

  public boolean removeMember(T member) {
    return members.remove(member);
  }
```

```
  public int getAverageAge() {

    if (members.size() == 0)
      return 0;

    short sum = 0;
    for (T member : members) {
      if(member != null) {
        sum += member.getAge();
      }
    }
    return sum / members.size();
  }

  public int getOldest() {
    int oldest = -1;

    for (T member : members) {
      if(member != null) {
        if (member.getAge() > oldest) {
          oldest = member.getAge();
        }
      }
    }
    return oldest;
  }
}
```

When the `StatsCalculator` is validated, the "members" field is also validated. The in-built value extractor for `List` is used to extract the values of `List` to validate the elements in `List`. In the case of an employee based List, each "Employee" element is validated. For example, an employee's "phone" is validated using the `@USPhoneNumber` constraint.

In the following example, let us consider a `StatisticsPrinter` that prints the statistics or displays the statistics on screen.

```
public class StatisticsPrinter {
  private StatsCalculator<@Valid Employee> calculator;

  public StatisticsPrinter(StatsCalculator<Employee> statsCalculator){
    this.calculator = statsCalculator;
  }

  public void displayStatistics(){
    //Use StatsCalculator, get stats, format and display them.
  }

  public void printStatistics(){
    //Use StatsCalculator, get stats, format and print them.
  }
```

```
}
```

The container `StatisticsPrinter` uses `StatisticsCalculator`. When `StatisticsPrinter` is validated, the `StatisticsCalculator` is also validated by using the cascading validation such as `@Valid` annotation. However, in order to retrieve the values of `StatsCalculator` container type, a value extractor is required. An implementation of `ValueExtractor` for `StatsCalculator` is as follows:

```
public class ExtractorForStatsCalculator implements
ValueExtractor<StatsCalculator<@ExtractedValue ?>> {

  @Override
  public void extractValues(StatsCalculator<@ExtractedValue ?> statsCalculator,
      ValueReceiver valueReceiver) {
      /* Simple value retrieval is done here.
          It is possible to adapt or unwrap the value if required.*/
    valueReceiver.value("<extracted value>", statsCalculator);
  }
}
```

There are multiple mechanisms to register the `ValueExtractor` with Jakarta Bean Validation. See, “Registering ValueExtractor” implementations section in the Jakarta Bean Validation specification https://jakarta.ee/specifications/bean-validation/3.0/. One of the mechanisms is to register the value extractor with Jakarta Bean Validation Context.

```
ValidatorFactory validatorFactory = Validation
        .buildDefaultValidatorFactory();

    ValidatorContext context = validatorFactory.
        usingContext()
        .addValueExtractor(new ExtractorForStatsCalculator());


    Validator validator = context.getValidator();
```

Using this validator, `StatsisticsPrinter` is validated in the following sequence of operations:

1. `StatisticsPrinter` is validated.
    a. The members of `StatisticsPrinter` that need cascading validation are validated.
    b. For container types, value extractor is determined. In the case of `StatsCalculator`, `ExtractorForStatsCalculator` is found and then values are retrieved for validation.
    c. `StatsCalculator` and its members such as `List` are validated.
    d. In-built `ValueExtractor` for `java.util.List` is used to retrieve the values of elements of the list and the validated. In this case, Employee and the field "phone" that is annotated with `@USPhoneNumber` constraint is validated.

Customizing Validator Messages

Jakarta Bean Validation includes a resource bundle of default messages for the built-in constraints. These messages can be customized and can be localized for non-English-speaking locales.

The ValidationMessages Resource Bundle

The `ValidationMessages` resource bundle and the locale variants of this resource bundle contain strings that override the default validation messages. The `ValidationMessages` resource bundle is typically a properties file, `ValidationMessages.properties`, in the default package of an application.

Localizing Validation Messages

Locale variants of `ValidationMessages.properties` are added by appending an underscore and the locale prefix to the base name of the file. For example, the Spanish locale variant resource bundle would be `ValidationMessages_es.properties`.

Grouping Constraints

Constraints may be added to one or more groups. Constraint groups are used to create subsets of constraints so that only certain constraints will be validated for a particular object. By default, all constraints are included in the `Default` constraint group.

Constraint groups are represented by interfaces.

```
public interface Employee {}

public interface Contractor {}
```

Constraint groups can inherit from other groups.

```
public interface Manager extends Employee {}
```

When a constraint is added to an element, the constraint declares the groups to which that constraint belongs by specifying the class name of the group interface name in the `groups` element of the constraint.

```
@NotNull(groups=Employee.class)
Phone workPhone;
```

Multiple groups can be declared by surrounding the groups with braces (`{` and `}`) and separating the groups' class names with commas.

```
@NotNull(groups={ Employee.class, Contractor.class })
Phone workPhone;
```

If a group inherits from another group, validating that group results in validating all constraints declared as part of the supergroup. For example, validating the `Manager` group results in the `workPhone` field being validated, because `Employee` is a superinterface of `Manager`.

Customizing Group Validation Order

By default, constraint groups are validated in no particular order. There are cases in which some groups should be validated before others. For example, in a particular class, basic data should be validated before more advanced data.

To set the validation order for a group, add a `jakarta.validation.GroupSequence` annotation to the interface definition, listing the order in which the validation should occur.

```
@GroupSequence({Default.class, ExpensiveValidationGroup.class})
public interface FullValidationGroup {}
```

When validating `FullValidationGroup`, first the `Default` group is validated. If all the data passes validation, then the `ExpensiveValidationGroup` group is validated. If a constraint is part of both the `Default` and the `ExpensiveValidationGroup` groups, the constraint is validated as part of the `Default` group and will not be validated on the subsequent `ExpensiveValidationGroup` pass.

Using Method Constraints in Type Hierarchies

If you add validation constraints to objects in an inheritance hierarchy, take special care to avoid unintended errors when using subtypes.

For a given type, subtypes should be able to be substituted without encountering errors. For example, if you have a `Person` class and an `Employee` subclass that extends `Person`, you should be able to use `Employee` instances wherever you might use `Person` instances. If `Employee` overrides a method in `Person` by adding method parameter constraints, code that works correctly with `Person` objects may throw validation exceptions with `Employee` objects.

The following code shows an incorrect use of method parameter constraints within a class hierarchy:

```
public class Person {
...
  public void setPhone(String phone) { ... }
}

public class Employee extends Person {
...
  @Override
  public void setPhone(@Verified String phone) { ... }
}
```

By adding the `@Verified` constraint to `Employee.setPhone`, parameters that were valid with `Person.setPhone` will not be valid with `Employee.setPhone`. This is called strengthening the

preconditions (that is, the method parameters) of a subtype's method. You may not strengthen the preconditions of subtype method calls.

Similarly, the return values from method calls should not be weakened in subtypes. The following code shows an incorrect use of constraints on method return values in a class hierarchy:

```
public class Person {
...
  @Verified
  public USPhoneNumber getPhone() { ... }
}

public class Employee extends Person {
...
  @Override
  public USPhoneNumber getPhone() { ... }
}
```

In this example, the `Employee.getPhone` method removes the `@Verified` constraint on the return value. Return values that would be not pass validation when calling `Person.getEmail` are allowed when calling `Employee.getPhone`. This is called weakening the postconditions (that is, return values) of a subtype. You may not weaken the postconditions of a subtype method call.

If your type hierarchy strengthens the preconditions or weakens the postconditions of subtype method calls, a `jakarta.validation.ConstraintDeclarationException` will be thrown by the Jakarta Bean Validation runtime.

Classes that implement several interfaces that each have the same method signature, known as parallel types, need to be aware of the constraints applied to the interfaces that they implement to avoid strengthening the preconditions. For example:

```
public interface PaymentService {
  void processOrder(Order order, double amount);
...
}

public interface CreditCardPaymentService {
  void processOrder(@NotNull Order order, @NotNull double amount);
...
}

public class MyPaymentService implements PaymentService,
        CreditCardPaymentService {

  @Override
  public void processOrder(Order order, double amount) { ... }
...
}
```

In this case, `MyPaymentService` has the constraints from the `processOrder` method in `CreditCardPaymentService`, but client code that calls `PaymentService.processOrder` doesn't expect these constraints. This is another example of strengthening the preconditions of a subtype and will result in a `ConstraintDeclarationException`.

Rules for Using Method Constraints in Type Hierarchies

The following rules define how method validation constraints should be used in type hierarchies.

- Do not add method parameter constraints to overridden or implemented methods in a subtype.
- Do not add method parameter constraints to overridden or implemented methods in a subtype that was originally declared in several parallel types.
- You may add return value constraints to an overridden or implemented method in a subtype.

Jakarta Security

Jakarta Security is the overarching security API in Jakarta EE. Overarching here means that it strives to address the security needs of all other APIs in Jakarta EE in a holistic way.

Due to historical and political reasons, a number of security features are still distributed among several other APIs in Jakarta EE. Sometimes they overlap, and sometimes such features are only accessible from these other APIs. In this chapter, we'll focus primarily on explaining Jakarta Security, but we'll mention when other APIs are needed to accomplish a certain task.

Overview

Before we look at some practical examples, let's quickly go through some basics.

Some of the guiding principles in Jakarta Security are:

1. It should work directly out of the box, without requiring vendor-specific configuration.
2. It leverages Jakarta CDI as much as possible. Most artifacts are CDI beans, and many features are done via CDI interceptors.
3. The difference between framework-provided artifacts and custom (user provided) artifacts is minimal or non-existent.
4. It fully integrates with security features from other Jakarta EE APIs and proprietary (vendor-specific) artifacts.

Jakarta Security defines several distinct artifacts that play an important role in the security process:

1. Authentication Mechanism
2. Identity Store
3. Permission Store

The first two of these are used in the authentication process:

An *authentication mechanism* is somewhat like a controller in the well-known MVC pattern; it is the

entity that interacts with the caller (typically a human), via some kind of view to collect credentials, and with the model (business logic) to validate these credentials. An authentication mechanism knows about the environment this caller uses to communicate with the server. An authentication mechanism for HTTP knows about URLs to redirect or forward to, or about response headers to send to the client. It also knows about the data coming back, such as cookies, request headers, and post data. Examples of authentication mechanisms are Form authentication and Basic authentication.

An *identity store* is more like the model in the MVC pattern. This entity strictly performs a business / data operation where credentials go in, and an identity comes out. The identity contains logic to validate said credentials, and embeds or contacts a database. This "database" contains usernames, along with their credentials and (typically) roles. An identity store therefore knows nothing about the environment that this caller uses to communicate with the server; for example, it doesn't know about HTTP or headers and more. Some examples of identity stores are services that contact SQL or NoSQL databases, LDAP servers, files on the file-system, and more.

Figure 7, "Mechanism Store in MVC" shows the *authentication mechanism* and *identity store* in an MVC-like structure.

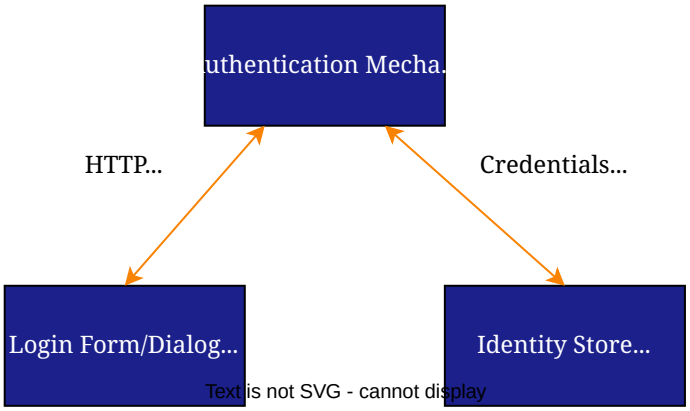


Figure 7. Mechanism Store in MVC

The third one is used for the authorization process:

A *permission store* is another kind of model that stores permissions, typically either globally, or per role (role-based permissions). This entity then performs a business / data operation where a query and an identity go in, and a yes/no answer goes out. For instance, a query such as "can access /foo/bar?" along with the identity for user "John" with roles "bar" and "kaz" would return "yes" if that identity is authorized to access "/foo/bar", and "no" if not authorized. Examples of permission stores are the Jakarta Authorization usage of the Policy class, or the internal data structure where a Servlet Container such as Tomcat or Jetty stores the security constraints an application defined.

Provided authentication mechanisms and identity stores

Jakarta Security provides a number of built-in authentication mechanisms and identity stores. We'll enumerate them here first, and will look at them in more detail below.

Authentication mechanisms:

1. Basic

2. Form
3. Custom Form
4. Open ID Connect (OIDC)

Identity stores:

1. Database
2. LDAP

Custom authentication mechanisms and identity stores

When the provided authentication mechanisms and identity stores aren't sufficient, we can easily define custom ones. Both provided and custom ones use the same interfaces, and the system doesn't distinguish between them.

Authentication mechanisms and identity stores from other APIs

The Servlet specification defines the exact same Form and Basic authentication mechanisms. Authenticating with them will have the same result as authenticating with a Jakarta Security authentication mechanism. (Role checks will work the same independent on which API was used to authenticate.)

A Servlet authentication mechanism, however, will not necessarily consult a Jakarta Security identity store. This is server dependent. The identity store that is called is server dependent as well. Calling this server-dependent identity store is possible from Jakarta Security, but as an advanced feature.

Likewise, programmatic role checks can be done from various APIs, including Jakarta Security, Jakarta REST, and Jakarta Servlet. These all return the same outcome, independent of whether authentication took place with a Jakarta Security Authentication Mechanism or a Servlet Authentication Mechanism. Within a Jakarta EE environment the usage of Jakarta Security for this is encouraged, and the usage of those other APIs is discouraged.



Programmatic role checks in Jakarta REST, Jakarta Servlet and various other APIs are not being deprecated for the time being, as those APIs are also used stand-alone (outside Jakarta EE). Future versions of those APIs may contain warnings about their usage within Jakarta EE.

Securing an endpoint with Basic authentication

In the following example, we'll be securing a REST endpoint using Basic authentication.

You'll learn how to:

1. Define security constraints
2. Set a provided authentication mechanism
3. Define (and implicitly set) a custom identity store

4. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the security constraints

Next we'll define the security constraints in `web.xml`, which tell the security system that access to a given URL or URL pattern is protected, and hence authentication is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>protected</web-resource-name>
```

```
            <url-pattern>/rest/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

This XML essentially says that to access any URL that starts with "/rest" requires the caller to have the role "user". Roles are opaque strings; merely identifiers. It's fully up to the application how broad or fine-grained they are.



In Jakarta EE, internally these XML constraints are transformed into `Permission` instances and made available via a specific type of permission store. Knowledge about this transformation is only needed for very advanced use cases.



The observant reader may wonder if XML is really the only option here, given the strong feelings that exist in parts of the community around XML. The answer is yes and no. Jakarta EE does define the @RolesAllowed annotation that could be used to replace the XML shown above, but only the legacy Enterprise Beans has specified a behaviour for this when put on an Enterprise Bean. Jakarta REST has done no such thing, although the JWT API in MicroProfile has defined this for REST resources. In Jakarta EE, however, this remains a vendor-specific extension. There are also a number of annotations and APIs in Jakarta EE to set these kinds of constraints for individual Servlets, but those won't help us much either here.

Declare the authentication mechanism

```
@ApplicationScoped
@BasicAuthenticationMechanismDefinition(realmName = "basicAuth")
@DeclareRoles({ "user", "caller" })
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

}
```

To declare the usage of a specific authentication mechanism, Jakarta EE provides `[XYZ]MechanismDefinition` annotations. Such an annotation is picked up by the security system, and in response to it a CDI bean that implements the HttpAuthenticationMechanism is enabled for it.

The annotation can be put on any bean, but in a REST application it fits particularly well on the `Application` subclass because it also declares the path for REST resources.

Define the identity store

Finally, let's define a simple identity store that the security system can use to validate provided credentials for Basic authentication:

```
@ApplicationScoped
public class TestIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
    }

}
```

This identity store only validates the single identity (user) "john", with password "secret1" and roles "user" and "caller". Defining this kind of identity store is often the simplest way to get started.



Jakarta Security doesn't provide a simple identity store out of the box. The reason is that everything in Jakarta Security promotes best practices, and it's not clear if a simple identity store fits in with those best practices.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restBasicAuthCustomStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.414 s - in
jakartaee.examples.focused.security.restbasicauthcustomstore.RestBasicAuthCustomStoreI
T
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
```

```
@RunAsClient
public class RestBasicAuthCustomStoreIT extends ITBase {

    /**
     * Stores the base URL.
     */
    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        DefaultCredentialsProvider credentialsProvider = new
DefaultCredentialsProvider();
        credentialsProvider.addCredentials("john", "secret1");

        webClient.setCredentialsProvider(credentialsProvider);

        TextPage page = webClient.getPage(baseUrl + "/rest/resource");
        String content = page.getContent();

        System.out.println(content);
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restBasicAuthCustomStore/target/restBasicAuthCustomStore.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

The `DefaultCredentialsProvider` used here makes sure that the headers for Basic authentication are added to the request. The Basic authentication mechanism that we defined for our applications reads those headers, extracts the username and password from them, and consults our identity store with them.

Securing an Endpoint with Basic Authentication and a Database Identity Store

In the following example, we'll secure a REST endpoint using Basic authentication and the database identity store that is provided by Jakarta Security.

You'll learn how to:

1. Define security constraints
2. Use the provided BasicAuthenticationMechanismDefinition
3. Use the provided DatabaseIdentityStoreDefinition
4. Populate and configure the identity store
5. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the authentication mechanism and identity store

```
@ApplicationScoped
@BasicAuthenticationMechanismDefinition(
    realmName = "basicAuth"
)
@DatabaseIdentityStoreDefinition(
```

```
    callerQuery = "select password from basic_auth_user where username = ?",
    groupsQuery = "select name from basic_auth_group where username = ?",
    hashAlgorithmParameters = {
        "Pbkdf2PasswordHash.Iterations=3072",
        "Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512",
        "Pbkdf2PasswordHash.SaltSizeBytes=64"
    }
)
@DeclareRoles("user")
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {
```

To declare the usage of a specific authentication mechanism, Jakarta EE provides `[XYZ]MechanismDefinition` annotations. Such an annotation is picked up by the security system, and in response to it a CDI bean that implements the HttpAuthenticationMechanism is enabled for it.

The annotation can be put on any bean, but in a REST application it fits particularly well on the `Application` subclass because it also declares the path for REST resources.

Likewise, to declare the usage of a specific identity store, Jakarta EE provides `[XYZ]StoreDefinition` annotations.

The annotations can be put on any bean, but in a REST application it fits particularly well on the `Application` subclass that also declares the path for REST resources.

You can use the provided `DatabaseIdentityStoreDefinition` with any authentication mechanism that validates username/password credentials. It requires at least two SQL queries:

1. A query that returns a password for the username part of credentials. The returned password is compared with the password part of those credentials. If they match (of more typically, their hashes match) the credential is considered valid.
2. A query that returns a number of roles given that same username part of the credentials

Although not required, it's a good practice to provide some parameters for the hash algorithm. Passwords should never be stored in plain-text in a database.

Populating the identity store

In order to use the identity store, we need to put some data in a database. The following code shows one way how to do that:

```
@ApplicationScoped
@BasicAuthenticationMechanismDefinition(
    realmName = "basicAuth"
)
@DatabaseIdentityStoreDefinition(
    callerQuery = "select password from basic_auth_user where username = ?",
    groupsQuery = "select name from basic_auth_group where username = ?",
    hashAlgorithmParameters = {
        "Pbkdf2PasswordHash.Iterations=3072",
```

```
        "Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512",
        "Pbkdf2PasswordHash.SaltSizeBytes=64"
    }
)
@DeclareRoles("user")
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

    /**
     * Id of the one and only user we populate in out DB.
     */
    private static final BigInteger USER_ID = ONE;

    /**
     * Id of the one and only group we populate in out DB.
     */
    private static final BigInteger GROUP_ID = ONE;

    @PersistenceContext
    private EntityManager entityManager;

    @Inject
    private Pbkdf2PasswordHash passwordHash;

    @Transactional
    public void onStart(@Observes @Initialized(ApplicationScoped.class) Object
applicationContext) {
        passwordHash.initialize(Map.of(
            "Pbkdf2PasswordHash.Iterations", "3072",
            "Pbkdf2PasswordHash.Algorithm", "PBKDF2WithHmacSHA512",
            "Pbkdf2PasswordHash.SaltSizeBytes", "64"));

        if (entityManager.find(User.class, USER_ID) == null) {
            var user = new User();
            user.id  = USER_ID;
            user.username = "john";
            user.password = passwordHash.generate("secret1".toCharArray());
            entityManager.persist(user);
        }

        if (entityManager.find(Group.class, GROUP_ID) == null) {
            var group = new Group();
            group.id = GROUP_ID;
            group.name = "user";
            group.username = "john";
            entityManager.persist(group);
        }
    }

}
```

```
@Entity
@Table(name = "basic_auth_user")
class User {
    @Id
    BigInteger id;

    @Column(name = "password")
    String password;

    @Column(name = "username", unique = true)
    String username;
}

@Entity
@Table(name = "basic_auth_group")
class Group {
    @Column(name = "id")
    @Id
    BigInteger id;

    @Column(name = "name")
    String name;

    @Column(name = "username")
    String username;
}
```

The code above uses Jakarta Persistence, which generates SQL from Java types. Jakarta Persistence is discussed in detail in its own chapter. Since we haven't specified a datasource, the `@DatabaseIdentityStoreDefinition` annotation will use the default datasource defined in Jakarta EE, so you don't have to explicitly install and configure an external database such as Postgres or MySQL. However, if necessary, you can configure a different one using the `dataSourceLookup` attribute.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restBasicAuthDBStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.307 s - in
jakartaee.examples.focused.security.restbasicauthdbstore.RestBasicAuthDBStoreIT
```

The test itself is basically the same as that for the Securing an endpoint with Basic authentication example.

Securing an endpoint with Basic authentication and multiple identity stores

In the following example, we'll be securing a REST endpoint using Basic authentication and two identity stores: the database identity store that is provided by Jakarta Security and a custom identity store.

You'll learn how to:

1. Define security constraints
2. Use the provided BasicAuthenticationMechanismDefinition
3. Use the provided DatabaseIdentityStoreDefinition
4. Create a custom identity store
5. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is

> a discouraged type and the Jakarta Security version is to be preferred.

Declare the authentication mechanism and identity store

```
@ApplicationScoped
@BasicAuthenticationMechanismDefinition(
    realmName = "basicAuth"
)
@DatabaseIdentityStoreDefinition(
    callerQuery = "select password from basic_auth_user where username = ?",
    groupsQuery = "select name from basic_auth_group where username = ?",
    hashAlgorithmParameters = {
        "Pbkdf2PasswordHash.Iterations=3072",
        "Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512",
        "Pbkdf2PasswordHash.SaltSizeBytes=64"
    }
)
@DeclareRoles("user")
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {
```

```
@ApplicationScoped
public class CustomIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("pete", "secret2")) {
            return new CredentialValidationResult("pete", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
    }

}
```

In this example we have two enabled CDI beans implementing the `IdentityStore` interface. One of them will be implicitly enabled via the `@DatabaseIdentityStoreDefinition` annotation, while the other one is defined explicitly via the `CustomIdentityStore` class. As with a single identity store, it doesn't matter how or where the CDI beans are defined, only that multiple enabled ones exist.

When multiple identity stores are present, the security system will try them in order of their priority. We didn't set a priority here, so the order will be undefined. If the default validation algorithm is used, a successful validation wins over a failed validation. For example, let's say we have multiple identity stores that know about the user "pete". If "pete" fails validation in one store, but passes validation in another store, the end result is still that validation passed.

In the two stores above, however only one store knows about "pete" and that's the `CustomIdentityStore`. The store created from `@DatabaseIdentityStoreDefinition` doesn't know about

"pete" at all, and will simply not validate it.

Populating the identity store

In order to use the identity store, we need to put some data in a database. This is done in the same as in Securing an Endpoint with Basic Authentication and a Database Identity Store.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restBasicAuthDBStoreAndCustomStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
pete : true
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 9.239 s - in
jakartaee.examples.focused.security.restbasicauthdbstoreandcustomstore.RestBasicAuthDB
StoreAndCustomStoreIT
```

Let's take a quick look at the actual test again:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestBasicAuthDBStoreAndCustomStoreIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * <p>
     * This will use the "john" credentials, which should be validated by the DB store
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall1() throws Exception {
        DefaultCredentialsProvider credentialsProvider = new
DefaultCredentialsProvider();
        credentialsProvider.addCredentials("john", "secret1");
```

```
        webClient.setCredentialsProvider(credentialsProvider);

        TextPage page = webClient.getPage(baseUrl + "/rest/resource");
        String content = page.getContent();

        System.out.println(content);
    }

    /**
     * Test the call to a protected REST service
     *
     * <p>
     * This will use the "pete" credentials, which should be validated by the custom
store
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall2() throws Exception {
        DefaultCredentialsProvider credentialsProvider = new
DefaultCredentialsProvider();
        credentialsProvider.addCredentials("pete", "secret2");

        webClient.setCredentialsProvider(credentialsProvider);

        TextPage page = webClient.getPage(baseUrl + "/rest/resource");
        String content = page.getContent();

        System.out.println(content);
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restBasicAuthDBStoreAndCustomStore/target/restBasicAuthDBStoreAndCustomStore.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

We have two tests here: in one test we try to authenticate as "john", in the other test as "pete". As we've seen, each identity store only validates one of them. The fact that both tests pass demonstrates that each store will validate the right user, and that not recognizing a username by any of them will not fail the overall validation.

Securing an endpoint with Form authentication

In the following example, we'll secure a REST endpoint using Form authentication.

You'll learn how to:

1. Define security constraints
2. Use the Form authentication mechanism
3. How to define (and implicitly set) a custom identity store
4. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the security constraints

Next we'll define the security constraints in `web.xml`, which tell the security system that access to a given URL or URL pattern is protected, and hence authentication is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>protected</web-resource-name>
            <url-pattern>/rest/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

This XML essentially says that to access any URL that starts with "/rest" requires the caller to have the role "user". Roles are opaque strings; merely identifiers. It's fully up to the application how broad or fine-grained they are.



In Jakarta EE, internally these XML constraints are transformed into `Permission` instances and made available via a specific type of permission store. Knowledge about this transformation is only needed for very advanced use cases.



The observant reader may wonder if XML is really the only option here, given the strong feelings that exist in parts of the community around XML. The answer is yes and no. Jakarta EE does define the `@RolesAllowed` annotation that could be used to replace the XML shown above, but only the legacy Enterprise Beans has specified a behaviour for this when put on an Enterprise Bean. Jakarta REST has done no such thing, although the JWT API in MicroProfile has defined this for REST resources. In Jakarta EE, however, this remains a vendor-specific extension. There are also a number of annotations and APIs in Jakarta EE to set these kinds of constraints for individual Servlets, but those won't help us much either here.

Declare the authentication mechanism

```
@ApplicationScoped
@FormAuthenticationMechanismDefinition(
    loginToContinue = @LoginToContinue(
        loginPage="/login.html",
        errorPage="/login-error.html"
    )
)
@DeclareRoles({ "user", "caller" })
```

```
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

}
```

To declare the usage of a specific authentication mechanism, Jakarta EE provides `[XYZ]MechanismDefinition` annotations. Such an annotation is picked up by the security system, and in response to it a CDI bean that implements the HttpAuthenticationMechanism is enabled for it.

The annotation can be put on any bean, but in a REST application it fits particularly well on the `Application` subclass because it also declares the path for REST resources.

Contrary to the Basic HTTP authentication mechanism, the Form authentication mechanism allows us to customize the login dialog (the process between the caller and the authentication mechanism) and to keep track of the authenticated session on the server (using a cookie). This also allows us to logout, something that for unknown reasons has never been specified for Basic HTTP authentication.

To use this authentication method, we need to designate two paths to resources that are relative to our application. One path is for the login page, which the user will be directed to when attempting to access a protected resource. The other path is for when login fails, such as when the user enters incorrect login credentials. If the paths are the same, a request parameter can be used to distinguish between them. Paths can point to anything our server can respond to; a static HTML file, a REST resource, or anything else. For simplicity, we'll use two static HTML files here:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Login to continue</title></head>
    <body>
        <h1>Login to continue</h1>
        <form method="post" action="j_security_check">
            <div>
                <label>Username: <input type="text" name="j_username"></label>
            </div>
            <div>
                <label>Password: <input type="password" name="j_password"></label>
            </div>
            <div>
                <input type="submit" value="Submit">
            </div>
        </form>
    </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Login failed!</title></head>
    <body>
```

```
        <h1>Login failed!</h1>
        <div>
            <a href="login.html">Try again</a>
        </div>
    </body>
</html>
```

Define the identity store

Finally, let's define a basic identity store that the security system can use to validate provided credentials for Form authentication:

```
@ApplicationScoped
public class CustomIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
    }

}
```

This identity store only validates the single identity (user) "john", with password "secret1" and roles "user" and "caller". Defining this kind of identity store is often the simplest way to get started. Note that Jakarta Security doesn't define a simple identity store out of the box, because there are questions about whether that would promote security best practices.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restBasicAuthCustomStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.24 s - in
jakartaee.examples.focused.security.restformauthcustomstore.RestFormAuthCustomStoreIT
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestFormAuthCustomStoreIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        HtmlPage loginPage = webClient.getPage(baseUrl + "/rest/resource");
        System.out.println(loginPage.asXml());

        HtmlForm form = loginPage.getForms()
                                 .get(0);

        form.getInputByName("j_username")
            .setValueAttribute("john");

        form.getInputByName("j_password")
            .setValueAttribute("secret1");

        TextPage page = form.getInputByValue("Submit")
                    .click();

        System.out.println(page.getContent());
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restBasicAuthCustomStore/target/restBasicAuthCustomStore.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

The test first sends a request here to the protected resource, and the server responds with the HTML form we defined above. Using the `HtmlUnit` API, it's easy to navigate the HTML DOM, fill out the username and password in the form, and programmatically click the Submit button. The form posts back to a special "j_security_check" URL, where the authentication mechanism receives the request and retrieves the username and password from the POST data, much like the Basic authentication mechanism retrieves them from the HTTP headers.

Securing an endpoint with Basic authentication and a custom algorithm for handling multiple identity stores

In the following example, we'll be securing a REST endpoint using Basic authentication and two identity stores: the database identity store that is provided by Jakarta Security and a custom identity store. Instead of relying on the default algorithm provided by Jakarta Security to handle multiple identity stores we'll be using a custom algoritm.

You'll learn how to:

1. Define security constraints
2. Use the provided BasicAuthenticationMechanismDefinition
3. Use the provided DatabaseIdentityStoreDefinition
4. Create a custom identity store
5. Create a custom identity store handler
6. Use the Jakarta Security SecurityContext

Write the application

We'll use a slightly modified resource and security constraints compared to the ones we used for the Securing an endpoint with Basic authentication example.

The REST resource is now as follows:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user") + "," +
            securityContext.isCallerInRole("caller1") + "," +
            securityContext.isCallerInRole("caller2");
    }
```

```
}
```

As can be seen, the difference is quite small; we're now printing out the results of two extra role checks.

`web.xml` on its turn looks as follows now:

```
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>protected</web-resource-name>
            <url-pattern>/rest/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
            <role-name>caller2</role-name>
        </auth-constraint>
    </security-constraint>
    <security-role>
        <role-name>caller1</role-name>
    </security-role>

</web-app>
```

Compared to the example in Securing an endpoint with Basic authentication we have now added an extra role to the `<auth-constraint>` section. The semantics of that are that a caller needs to have both of these roles in order to be authorised to access the resource under `/rest/*`.

Although it's customary to explicitly declare all roles in the application using `<security-role>`, it's technically not needed. As long as the role name appears in some XML fragment or annotation attribute the Jakarta EE requirement to declare all roles upfront is satisfied. As we can see in the fragment above, the role names "user" and "caller2" already appear in the `<auth-constraint>` section, so they don't **have** to be repeated.



The reason it's deemed good practice to list all roles in the `<security-role>` element in web.xml (or alternatively in an `@DeclareRoles` annotation) even when not really needed is to have a single place where all roles are listed, instead of them being scattered throughout the application.

Declare the authentication mechanism and identity store

```
@ApplicationScoped
```

```
@BasicAuthenticationMechanismDefinition(
    realmName = "basicAuth"
)
@DatabaseIdentityStoreDefinition(
    callerQuery = "select password from basic_auth_user where username = ?",
    groupsQuery = "select name from basic_auth_group where username = ?",
    hashAlgorithmParameters = {
        "Pbkdf2PasswordHash.Iterations=3072",
        "Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512",
        "Pbkdf2PasswordHash.SaltSizeBytes=64"
    }
)
@DeclareRoles("user")
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {
```

```
@ApplicationScoped
public class CustomIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("caller1",
"caller2"));
        }

        return INVALID_RESULT;
    }

}
```

In this example we have two enabled CDI beans implementing the `IdentityStore` interface. One of them will be implicitly enabled via the `@DatabaseIdentityStoreDefinition` annotation, while the other one is defined explicitly via the `CustomIdentityStore` class. As with a single identity store, it doesn't matter how or where the CDI beans are defined, only that multiple enabled ones exist.

When multiple identity stores are present, an identity store handler (of type `IdentityStoreHandler`) is consulted. Jakara Security provides a default one as explained in Securing an endpoint with Basic authentication and multiple identity stores. This default handler can be overridden however to provide custom semantics. We'll use a custom handler to enforce a caller authenticates with both identity stores, and we'll combine the roles returned by both in the final result.

Populating the identity store

In order to use the identity store, we need to put some data in a database. This is done in the same as in Securing an Endpoint with Basic Authentication and a Database Identity Store.



In the custom identity store defined above and in the database identity store here

> we both use name "john' and password "secret1".

Writing the identity store handler

We'll now write the identity store handler:

```
@Alternative ①
@Priority(APPLICATION) ②
@ApplicationScoped
public class CustomIdentityStoreHandler implements IdentityStoreHandler {

    @Inject
    Instance<IdentityStore> identityStores; ③

    @Override
    public CredentialValidationResult validate(Credential credential) {
        CredentialValidationResult result = null;
        Set<String> groups = new HashSet<>();

        for (IdentityStore identityStore : identityStores) {
            result = identityStore.validate(credential);
            if (result.getStatus() == NOT_VALIDATED) {
                // Identity store probably doesn't handle our credential type
                continue;
            }

            if (result.getStatus() == INVALID) {
                // Identity store handled our credential type and determined its
                // invalid. End the loop.
                return INVALID_RESULT;
            }

            groups.addAll(result.getCallerGroups());
        }

        return new CredentialValidationResult(
            result.getCallerPrincipal(), groups);
    }
}
```

① Since we're overriding an existing CDI bean (the default `IdentityStoreHandler` provided by Jakarta Security), we have to annotate our custom `IdentityStoreHandler` with `@Alternative`.

② To make `@Alternative` actually work, we additionally have to annotate with `@Priority(APPLICATION)`

③ With `@Inject Instance<IdentityStore> identityStores` CDI will give us a collection of all identity stores in the application. In the case of this example that will be the store behind `@DatabaseIdentityStoreDefinition` and our `CustomIdentityStore`. We can the iterate over those stores in our code, and offer the credentials (the username and password in this example) to each of them.

There are various result outcomes possible.

`NOT_VALIDATED` means the store did not try to validate the credentials at all. In most situations that status is set when the store in question doesnt't handle a given credential. I.e. it only handles say `JWTCredentials` and not `UsernamePasswordCredential`.

`INVALID` means the store tried to validate the credentials, and validation failed. For example the username and password were wrong.

In our custom handler code here we return an `INVALID_RESULT` for the first store that fails, as we want all stores to validate successfully here. If validation does succeed (the outcome is `VALID` then) we grab the groups it returned and store in a set.



Identity stores also have a capability to query it for roles directly, without validating credentials. We haven't used that feature here.

Eventually we return a result based on the `CallerPrincipal` from the last successful validation result, and all the collected groups.



In our example it doesn't matter from which validation result we grab the `CallerPrincipal` as it's all the one with name "pete" here. In general identity stores may transform the name from the input credential (for example "pete") to something else (for example "Pete Anderson").

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restBasicAuthCustomStoreHandler
```

This will run a test associated with the project, printing something like the following:

```
john : true,true,true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 7.634 s - in
jakartaee.examples.focused.security.restbasicauthcustomstorehandler.RestBasicAuthCusto
mStoreHandlerIT
```

The resource that we defined above required only two roles to access it (`user` and `caller2`), but our custom identity store also returned `caller1`. The resource we created tests for this, and as it appears, we indeed had this role.



If we hadn't declared `caller1` in `web.xml` (or via an annotation), the test for `caller1` might have returned false. This is however server dependent.

Securing an endpoint with a custom authentication mechanism and a custom identity store

In the following example, we'll be securing a REST endpoint using a custom authentication mechanism. A custom authentication mechanism is one we provide ourselves, instead of using one provided by Jakarta Security (such as the Basic HTTP authentication mechanism).

You'll learn how to:

1. Define security constraints
2. Define (and implicitly set) a custom authentication mechanism
3. Define (and implicitly set) a custom identity store
4. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Define the authentication mechanism

Let's now define a simple authentication mechanism that the security system can use to interact with the caller who tries to access a resource:

```
@ApplicationScoped
public class CustomAuthenticationMechanism implements HttpAuthenticationMechanism {

    @Inject
    private IdentityStoreHandler identityStoreHandler;

    @Override
    public AuthenticationStatus validateRequest(
        HttpServletRequest request,
        HttpServletResponse response,
        HttpMessageContext httpMessageContext) throws AuthenticationException {

        var callerName = request.getHeader("callername"); ①
        var password = request.getHeader("callerpassword");

        if (callerName == null || password == null) { ②
            return httpMessageContext.doNothing();
        }

        var result = identityStoreHandler.validate( ④
            new UsernamePasswordCredential(callerName, password)); ③

        if (result.getStatus() != VALID) {
            return httpMessageContext.responseUnauthorized();
        }

        return httpMessageContext.notifyContainerAboutLogin( ⑤
            result.getCallerPrincipal(),
            result.getCallerGroups());
    }

}
```

This custom authentication mechanism interacts with the caller by grabbing two headers from the request: `callername` and `callerpassword`. (1) In case any of them are `null`, we return a special status; the "do nothing" status. (2) This means there has been no request or attempt to do authentication. If the resource the caller is trying to access is not protected, the caller can access it anonymously. If it is proteced, the caller will not be able to access it.

When the two required headers are provided by the caller, we create a `UsernamePasswordCredential` out of their values (3) and pass that into the injected `IdentityStoreHandler`. (4) We saw how this type of handler worked in the example Securing an endpoint with Basic authentication and a custom algorithm for handling multiple identity stores.



An authentication mechanism in Jakarta Security is not strictly required to

delegate the credential validation to the identity store handler. However not doing so is considered bad practice, as it would restrict developers from things like inserting extra identity stores into the chain that can do things like adding extra groups.

If the credentials validated correctly, we use the `HttpMessageContext` to communicate the details of the authenticated caller to the container. (5)



In Jakarta Security the two basic items that make up an "authenticated identity" are just a caller principal (of type `Principal`) and a set of groups (of type `String`). Via a Service Provider Interface a specific Jakarta EE product (such as WildFly or GlassFish) is able to receive these two items and then stores it internally in some way.

The authentication mechanism is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement HttpAuthenticationMechanism. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Define the identity store

Finally, let's define a simple identity store that the security system can use to validate provided credentials for Basic authentication:

```
@ApplicationScoped
public class TestIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
    }

}
```

This identity store only validates the single identity (user) "john", with password "secret1" and roles "user" and "caller". Defining this kind of identity store is often the simplest way to get started.



Jakarta Security doesn't provide a simple identity store out of the box. The reason is that everything in Jakarta Security promotes best practices, and it's not clear if a simple identity store fits in with those best practices.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the

security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restCustomAuthCustomStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.591 s - in
jakartaee.examples.focused.security.restcustomauthcustomstore.RestCustomAuthCustomStor
eIT
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestCustomAuthCustomStoreIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        webClient.addRequestHeader("callername", "john");
        webClient.addRequestHeader("callerpassword", "secret1");

        TextPage page = webClient.getPage(baseUrl + "rest/resource");
        String content = page.getContent();

        System.out.println(content);
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restCustomAuthCustomStore/target/restCustomAuthCustomStore.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

The `webClient.addRequestHeader()` calls used here make sure that the headers for our custom authentication mechanism are added to the request. The authentication mechanism that we defined for our applications reads those headers, extracts the username and password from them, and consults our identity store with them.

Securing an endpoint with Form authentication and remember-me

In the following example, we'll secure a REST endpoint using Form authentication and remember-me.

Remember-me is a facility where an authenticated identity can be remembered beyond the scope of an HTTP session. This happens via a separate cookie that has a longer life-time than the cookie used for the HTTP session (and the session itself on the server).

You'll learn how to:

1. Define security constraints
2. Use the Form authentication mechanism
3. Enable the remember-me feature
4. How to define (and implicitly set) a custom remember-me identity store
5. How to define (and implicitly set) a custom identity store
6. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
```

```
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the security constraints

Next we'll define the security constraints in `web.xml`, which tell the security system that access to a given URL or URL pattern is protected, and hence authentication is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>protected</web-resource-name>
            <url-pattern>/rest/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

This XML essentially says that to access any URL that starts with "/rest" requires the caller to have the role "user". Roles are opaque strings; merely identifiers. It's fully up to the application how broad or fine-grained they are.



In Jakarta EE, internally these XML constraints are transformed into `Permission` instances and made available via a specific type of permission store. Knowledge about this transformation is only needed for very advanced use cases.



The observant reader may wonder if XML is really the only option here, given the strong feelings that exist in parts of the community around XML. The answer is yes and no. Jakarta EE does define the @RolesAllowed annotation that could be used to replace the XML shown above, but only the legacy Enterprise Beans has specified a behaviour for this when put on an Enterprise Bean. Jakarta REST has done no such thing, although the JWT API in MicroProfile has defined this for REST resources. In Jakarta EE, however, this remains a vendor-specific extension. There are also a number of annotations and APIs in Jakarta EE to set these kinds of constraints for individual Servlets, but those won't help us much either here.

Declare the authentication mechanism

We'll use the same authentication mechanism declaration as we used for Securing an endpoint with Form authentication

Enable remember-me

In Jakarta Security, there are several services available through CDI Interceptors [1], one of which is the remember-me service. Remember-me can be transparently applied to basically every authentication mechanism. In CDI, it's trivial to add Interceptors to beans that we define ourselves, but a little less trivial to add to provided beans. In this section we explain how to do this via a CDI extension.

For this example, we'll add the CDI extension interface (1) to our application config class:

```
@ApplicationScoped
@FormAuthenticationMechanismDefinition(
    loginToContinue = @LoginToContinue(
        loginPage="/login.html",
        errorPage="/login-error.html"
    )
)
@ApplicationPath("/rest")
public class ApplicationConfig extends Application
    implements BuildCompatibleExtension { ①

    @Enhancement(
        types = HttpAuthenticationMechanism.class,
        withSubtypes = true) ②
    public void addRememberMe(ClassConfig httpAuthenticationMechanism) {
        httpAuthenticationMechanism.addAnnotation(
            RememberMe.Literal.INSTANCE); ③
    }
}
```

CDI allows us to enhance classes using a method annotated with the `@Enhancement` annotation and as attribute the class we're seeking to enhance. For our example that will be a sub-type of the `HttpAuthenticationMechanism` interface (we know the bean enabled by `FormAuthenticationMechanismDefinition` will implement the `HttpAuthenticationMechanism` interface),

hence we set the `withSubtypes` attribute to `true`. (2)

Within the method we can then programmatically add the `@RememberMe` annotation used to bind the remember-me interceptor to a class. In the example here we use the default instance (which has all attributes set to their defaults). There are attributes for setting various aspects of the cookie, such as its name, whether it should be secure and http only, and perhaps most importantly the max age of the cookie (default is one day).

Define the remember-me identity store

For remember-me to work a token has to be created that is used as a credential to authenticate right away instead of invoking the authentication mechanism that is being intercepted. Jakarta Security uses a special identity store for this; the `RememberMeIdentityStore`. This type of identity store is exclusively used by the remember-me feature, hence it's a different type from `IdentityStore`.

Jakarta Security does not ship with any provided remember-me identity store, but for demonstration purposes we can easily create one ourselves.

The following shows an example:

```
@ApplicationScoped
public class CustomRememberMeIdentityStore implements RememberMeIdentityStore {

    private final Map<String, CredentialValidationResult> tokenToIdentityMap =
        new ConcurrentHashMap<>();

    @Override
    public String generateLoginToken(
        CallerPrincipal callerPrincipal, Set<String> groups) { ①
        var token = UUID.randomUUID().toString();

        tokenToIdentityMap.put(
            token,
            new CredentialValidationResult(callerPrincipal, groups));

        return token;
    }

    @Override
    public CredentialValidationResult validate(
        RememberMeCredential credential) { ②
        if (tokenToIdentityMap.containsKey(credential.getToken())) {
            return tokenToIdentityMap.get(credential.getToken());
        }

        return INVALID_RESULT;
    }

    @Override
    public void removeLoginToken(String token) { ③
```

```
        tokenToIdentityMap.remove(token);
    }

}
```

The `RememberMeIdentityStore` needs to perform 3 tasks.

It first needs to generate a token representing a caller principal and a set of groups. The caller principal and the set of groups are the ones set by the authentication mechanism right after the caller successfully authenticated. In our example (1) here we're generating a random UUID that's used as a key in an application scoped map.



Storing the authenticated identity (principal and groups) in an application scoped map is just an example. Other options could be storing it in a database or key-value store, encrypting the principal and groups, or generating some kind of JSON Web Token (JWT).



When storing the Principal, care must be taken that the Principal could be an elaborate custom Principal containing many more fields than just `name`.

The next thing that must be done is essentially similar to what a normal identity store does: validating a `Credential`. For a `RememberMeIdentityStore` this will always be of type `RememberMeCredential` with `getToken()` returning a token of the kind that was generated in `generateLoginToken()`. In our example (2) we're just using the token as key in our map.

Finally we can provide behaviour to remove the login token (and essentially invalidate it) via the `removeLoginToken` method. This method is called when a caller explicitly logs out. In our example (3) we just remove the token from our map.



When storing the principal and groups in a token that we send to the client we can't always easily invalidate it when the caller logs out; the caller can always keep the token and send it again.

Define the identity store

Finally, let's define a simple identity store that the security system can use to validate provided credentials for Basic authentication:

```
@ApplicationScoped
public class TestIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
```

```
    }

}
```

This identity store only validates the single identity (user) "john", with password "secret1" and roles "user" and "caller". Defining this kind of identity store is often the simplest way to get started.



Jakarta Security doesn't provide a simple identity store out of the box. The reason is that everything in Jakarta Security promotes best practices, and it's not clear if a simple identity store fits in with those best practices.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restFormAuthCustomStoreRememberMe
```

This will run a test associated with the project, printing something like the following:

```
john : true
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.702 s - in
jakartaee.examples.focused.security.restformauthcustomatorerememberme.RestFormAuthCust
omStoreIT
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestFormAuthCustomStoreRememberMeIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
```

```
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        // Initial request
        HtmlPage loginPage = webClient.getPage(baseUrl + "/rest/resource");
        System.out.println(loginPage.asXml());

        // Response is login form, so we can authenticate
        HtmlForm form = loginPage.getForms()
                                 .get(0);

        form.getInputByName("j_username")
            .setValueAttribute("john");

        form.getInputByName("j_password")
            .setValueAttribute("secret1");

        // After logging in, we should get the actual resource response
        TextPage page = form.getInputByValue("Submit")
                    .click();

        System.out.println(page.getContent());

        // Remove all cookies (specially the JSESSONID), except for the
        // JREMEMBERMEID cookie which carries the token to login again
        for (Cookie cookie : webClient.getCookieManager().getCookies()) {
            if (!"JREMEMBERMEID".equals(cookie.getName())) {
                webClient.getCookieManager().removeCookie(cookie);
            }
        }

        // Should get the resource response, and not the login form
        TextPage pageAgain = webClient.getPage(baseUrl + "/rest/resource");

        System.out.println(pageAgain.getContent());
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restFormAuthCustomStoreRememberMe/target/restFormAuthCustomStoreRememberMe.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

The test first sends a request here to the protected resource, and the server responds with the

HTML form we defined above. Using the `HtmlUnit` API, it's easy to navigate the HTML DOM, fill out the username and password in the form, and programmatically click the Submit button. The form posts back to a special "j_security_check" URL, where the authentication mechanism receives the request and retrieves the username and password from the POST data, much like the Basic authentication mechanism retrieves them from the HTTP headers.

Then we delete all cookies, specifically the `JSESSIONID` cookie that keeps the session that the form authentication mechanism uses to remember the authenticated identity. The test then does another request, and this time the value from the `JREMEMBERMEID` cookie is used to login.

Securing an endpoint with a custom authentication mechanism, a custom identity store and remember-me

In the following example, we'll secure a REST endpoint using custom authentication and remember-me.

Remember-me is a facility where an authenticated identity can be remembered beyond the scope of an HTTP session. This happens via a separate cookie that has a longer life-time than the cookie used for the HTTP session (and the session itself on the server).

You'll learn how to:

1. Define security constraints
2. Define (and implicitly set) a custom authentication mechanism with remember-me
3. How to define (and implicitly set) a custom remember-me identity store
4. Define (and implicitly set) a custom identity store
5. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }
```

```
}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Define the authentication mechanism

Let's now define a simple authentication mechanism that the security system can use to interact with the caller who tries to access a resource and specifically make sure the RememberMe feature is used:

```
@RememberMe ⑥
@ApplicationScoped
public class CustomAuthenticationMechanism implements HttpAuthenticationMechanism {

    @Inject
    private IdentityStoreHandler identityStoreHandler;

    @Override
    public AuthenticationStatus validateRequest(
        HttpServletRequest request,
        HttpServletResponse response,
        HttpMessageContext httpMessageContext) throws AuthenticationException {

        var callerName = request.getHeader("callername"); ①
        var password = request.getHeader("callerpassword");

        if (callerName == null || password == null) { ②
            return httpMessageContext.doNothing();
        }

        var result = identityStoreHandler.validate( ④
            new UsernamePasswordCredential(callerName, password)); ③

        if (result.getStatus() != VALID) {
            return httpMessageContext.responseUnauthorized();
        }

        return httpMessageContext.notifyContainerAboutLogin( ⑤
            result.getCallerPrincipal(),
            result.getCallerGroups());
```

```
    }

}
```



This is the same custom authentication mechanism that was used in Securing an endpoint with a custom authentication mechanism and a custom identity store, but with the `@RememberMe` annotation added.

This custom authentication mechanism interacts with the caller by grabbing two headers from the request: `callername` and `callerpassword`. (1) In case any of them are `null`, we return a special status; the "do nothing" status. (2) This means there has been no request or attempt to do authentication. If the resource the caller is trying to access is not protected, the caller can access it anonymously. If it is proteced, the caller will not be able to access it.

When the two required headers are provided by the caller, we create a `UsernamePasswordCredential` out of their values (3) and pass that into the injected `IdentityStoreHandler`. (4) We've seen how such handler worked in the example Securing an endpoint with Basic authentication and a custom algorithm for handling multiple identity stores.



An authentication mechanism in Jakarta Security is not strictly required to delegate the credential validation to the identity store handler. However not doing so is considered bad practice, as it would restrict developers from things like inserting extra identity stores into the chain that can do things like adding extra groups.

If the credentials validated correctly, we use the `HttpMessageContext` to communicate the details of the authenticated caller to the container. (5)



In Jakarta Security the two basic items that make up an "authenticated identity" are just a caller principal (of type `Principal`) and a set of groups (of type `String`). Via a Service Provider Interface a specific Jakarta EE product (such as WildFly or GlassFish) is able to receive these two items and then stores it internally in some way.

We annotate our custom authentication mechanism with the `@RememberMe` annotation to enable the remember-me feature for use with this authentication mechanism. In the example here we don't set any attributes (all of them have default values). There are attributes for setting various aspects of the cookie used for remember-me, such as its name, whether it should be secure and http only, and perhaps most importantly the max age of the cookie (default is one day).



Instead of using the `@RememberMe` annotation here, we could also have used the same extension that was used in Securing an endpoint with Form authentication and remember-me to enable the remember-me feature. The annotation however is a little bit easier to use.

The authentication mechanism is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement HttpAuthenticationMechanism. Such

beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Define the identity store

Finally, let's define a simple identity store that the security system can use to validate provided credentials for Basic authentication:

```
@ApplicationScoped
public class TestIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
    }

}
```

This identity store only validates the single identity (user) "john", with password "secret1" and roles "user" and "caller". Defining this kind of identity store is often the simplest way to get started.



Jakarta Security doesn't provide a simple identity store out of the box. The reason is that everything in Jakarta Security promotes best practices, and it's not clear if a simple identity store fits in with those best practices.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Define the remember-me identity store

We'll use the same remember-me identity store as we used for the Securing an endpoint with Form authentication and remember-me example.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restCustomAuthCustomStoreRememberMe
```

This will run a test associated with the project, printing something like the following:

```
john : true
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.287 s - in
jakartaee.examples.focused.security.restcustomauthcustomstorerememberme.RestCustomAuth
CustomStoreRememberMeIT
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestFormAuthCustomStoreRememberMeIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        // Initial request
        HtmlPage loginPage = webClient.getPage(baseUrl + "/rest/resource");
        System.out.println(loginPage.asXml());

        // Response is login form, so we can authenticate
        HtmlForm form = loginPage.getForms()
                                 .get(0);

        form.getInputByName("j_username")
            .setValueAttribute("john");

        form.getInputByName("j_password")
            .setValueAttribute("secret1");

        // After logging in, we should get the actual resource response
        TextPage page = form.getInputByValue("Submit")
                    .click();

        System.out.println(page.getContent());

        // Remove all cookies (specially the JSESSONID), except for the
```

```
        // JREMEMBERMEID cookie which carries the token to login again
        for (Cookie cookie : webClient.getCookieManager().getCookies()) {
            if (!"JREMEMBERMEID".equals(cookie.getName())) {
                webClient.getCookieManager().removeCookie(cookie);
            }
        }

        // Should get the resource response, and not the login form
        TextPage pageAgain = webClient.getPage(baseUrl + "/rest/resource");

        System.out.println(pageAgain.getContent());
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restCustomAuthCustomStoreRememberMe/target/restCustomAuthCustomStoreRememberMe.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

The `webClient.addRequestHeader()` calls used here make sure that the headers for our custom authentication mechanism are added to the request. The authentication mechanism that we defined for our applications reads those headers, extracts the username and password from them, and consults our identity store with them.

The test sends a request here to the protected resource along with the headers we mentioned above, and the server responds with the right content.

Then we delete all cookies, except for the `JREMEMBERMEID` cookie, and we unset all headers that we used before. The test then does another request, and this time the value from the `JREMEMBERMEID` cookie is used to login.

Securing an endpoint with OpenID Connect authentication

In the following example, we'll be securing a REST endpoint using OpenID Connect authentication.

With OpenID Connect authentication a caller is redirected to a third party server, typically a public one such as Google, Facebook, Linkedin, Apple, and more, but it can be a private one as well. The caller authenticates with that third party server, and is then redirected back along with a token. Our server than validates that token, and if it's valid the caller is considered authenticated.

You'll learn how to:

1. Define security constraints
2. Use the OpenID Connect authentication mechanism

3. Define (and implicitly set) a custom identity store used for authorization only
4. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the security constraints

Next we'll define the security constraints in `web.xml`, which tell the security system that access to a given URL or URL pattern is protected, and hence authentication is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <security-constraint>
```

```
        <web-resource-collection>
            <web-resource-name>protected</web-resource-name>
            <url-pattern>/rest/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

This XML essentially says that to access any URL that starts with "/rest" requires the caller to have the role "user". Roles are opaque strings; merely identifiers. It's fully up to the application how broad or fine-grained they are.



In Jakarta EE, internally these XML constraints are transformed into `Permission` instances and made available via a specific type of permission store. Knowledge about this transformation is only needed for very advanced use cases.



The observant reader may wonder if XML is really the only option here, given the strong feelings that exist in parts of the community around XML. The answer is yes and no. Jakarta EE does define the @RolesAllowed annotation that could be used to replace the XML shown above, but only the legacy Enterprise Beans has specified a behaviour for this when put on an Enterprise Bean. Jakarta REST has done no such thing, although the JWT API in MicroProfile has defined this for REST resources. In Jakarta EE, however, this remains a vendor-specific extension. There are also a number of annotations and APIs in Jakarta EE to set these kinds of constraints for individual Servlets, but those won't help us much either here.

Declare the authentication mechanism

```
@OpenIdAuthenticationMechanismDefinition(
    providerURI = "https://localhost:8443/openid-connect-server-webapp", ①
    clientId = "client", ②
    clientSecret = "secret", ③
    redirectToOriginalResource = true ④
)
@ApplicationScoped
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

}
```

To declare the usage of a specific authentication mechanism, Jakarta EE provides `[XYZ]MechanismDefinition` annotations. Such an annotation is picked up by the security system, and in response to it a CDI bean that implements the HttpAuthenticationMechanism is enabled for it.

The annotation can be put on any bean, but in a REST application it fits particularly well on the

`Application` subclass because it also declares the path for REST resources.

Contrary to the Basic HTTP authentication mechanism and the Form authentication mechanism, the OpenID Connect authentication mechanism requires a third party server that performs the actual authentication. Such third party server is called the OpenID Connect Provider (OIDC provider or OpenID Provider are also used). After authentication this provider handles user consent and and issues a token. The client requesting a user's authentication is called a Relying Party. In the case of Jakarta EE and Jakarta Security, the Jakarta EE server running the OpenID Connect authentication mechanism is a Relying Party.

To use this authentication mechanism, Jakarta Security provides the `@OpenIdAuthenticationMechanismDefinition` annotation, for which we typically need 3 mandatory configuration items as shown in the example code above.

The first is the `providerURI` (1), which points to the third party OpenID Connect Provider. In this example we use `https://localhost:8443/openid-connect-server-webapp`, which is the URL on which the example code has installed and started a local OpenID Connect provider called "Mitre". Whenever a caller accesses a protected resource, that caller is redirected to that OpenID Connect Provider.

The OpenId authentication mechanism needs to identify itself to the OpenID Connect Provider via a username/password (called `clientId` (2) and `clientSecret` (3)). We use "client" respectively "secret" here for those, which are the credentials for a default client that is available in Mitre.

After a caller successfully authenticates with the OpenID Connect Provider, that caller is redirected back to a URL on the Relying Party (our Jakarta EE server). This is called the "callback URL" and can be set via the `redirectURI` attribute. The default value is `${baseURL}/Callback`, where `${baseURL}` expands to the context-root of the application that uses Jakarta Security, for example `https://localhost:8080/openid-client` in our example. This exact URI must be known to Mitre. Mitre (and any OpenID Connect Provider in general) never redirects to unknown URIs.

By default, after the caller is redirected back to the Relying Party (our Jakarta EE server), the resource behind `/Callback` is invoked. When the attribute `redirectToOriginalResource` (4) is set to `true` however, the caller is once again redirected to the URL originally requested and which triggered the authentication process.



When `redirectToOriginalResource` is set to to `true` it's not necessary to actually map anything to the callback URL (for example a Servlet or a REST resource). The authentication mechanism is invoked before the resource mapped to the callback URL is invoked, so if the authentication mechanism always redirects it never invokes this resource and the resource therefore doesn't need to actually exist.

Define the identity store

In many cases the OpenID Connect Provider has no knowledge of the application for which it authenticates the caller, and therefore does not normally provide any logical groups for the authenticated user. Those groups are application specific after all. We therefore define an additional identity store that does provide those groups for a caller.



Despite not being typical, Jakarta Security supports getting the groups via the claimsDefinition attribute of the `@OpenIdAuthenticationMechanismDefinition` annotation. This can be used to set a claim name (default is "groups"). Jakarta Security then tries to find this name in the AccessToken, IdentityToken, or in the info returned by the `/userinfo` endpoint of the OpenId Connect Provider. Providers often need special configuration to return group claims.

```
@ApplicationScoped
public class AuthorizationIdentityStore implements IdentityStore {

    private Map<String, Set<String>> groupsPerCaller =
        Map.of("user", Set.of("user")); ②

    @Override
    public Set<ValidationType> validationTypes() {
        return EnumSet.of(PROVIDE_GROUPS); ①
    }

    @Override
    public Set<String> getCallerGroups(
        CredentialValidationResult validationResult) { ③
        return groupsPerCaller.get(validationResult.getCallerPrincipal().getName());
    }

}
```

This identity store is set to `PROVIDE_GROUPS` (1) only, which means the default `IdentityStoreHandler` will consult this identity store for groups after another identity store has successfully validated the credentials. For our example here we create a `Map` (2) with as key the caller principal name, and as value the set of groups. When the `IdentityStoreHandler` comes asking for the groups (3) of caller "user", a set with just the group "user" is returned.



As validation of the `IdentityToken` that's returned by the OpenID Connect Provider is integral to the OpenID Connect flow and not application specific, developers don't have to provide or define an identity store explicitly for this. Such a store is provided by Jakarta Security as an implementation detail, and automatically activated when the OpenID Connect authentication mechanism is activated.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Install and configure Mitre

Installing and configuring the OpenID Connect provider Mitre is outside the scope of Jakarta Security itself, but for completeness sake we'll briefly discuss it by illustration of Maven pom

fragments.

```
<plugin>

    <!--
        Unpack and install Tomcat + Mitre

        Mitre is a Spring based OpenID Connect Server that best runs on a javax based
Tomcat.
    -->
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>org.apache.tomcat</groupId>
                        <artifactId>tomcat</artifactId>
                        <version>9.0.76</version>
                        <type>zip</type>
                        <outputDirectory>${tomcat.root}</outputDirectory>
                    </artifactItem>
                    <artifactItem>
                        <groupId>org.mitre</groupId>
                        <artifactId>openid-connect-server-webapp</artifactId>
                        <version>1.3.4</version>
                        <type>war</type>
                        <outputDirectory>${tomcat.dir}/webapps/openid-connect-server-
webapp</outputDirectory>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
```

Mitre is a Spring application that uses the `javax.*` namespace. We therefore need a Tomcat from the 9.x series, which is available as a zip file from the Maven coordinates `org.apache.tomcat:tomcat:9.0.76`. Likewise, Mitre is available from `org.mitre:openid-connect-server-webapp:1.3.4`. We simply need to unzip Tomcat, and unzip Mitre into its `webapps` folder. We also need to update Tomcat with the JAXB standalone libraries (see full example code).

```
<!--
    Configure Tomcat to use HTTPS, as Open ID Connect strictly speaking requires this.
    Some servers may refuse to use Open ID Connect if not running on a secure
connection.

    Also configure Mitre to use the callback of our client.
```

```
    Then start Tomcat and with it Mitre.
-->
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <target>
                    <echo level="info">Replacing in ${tomcat.dir}</echo>

                    <!-- Configure Mitre to let it know its running on HTTPS  -->
                    <replace token="http://localhost:8080"
value="https://localhost:8443" dir="${tomcat.dir}/webapps/openid-connect-server-
webapp/WEB-INF" summary="yes">
                        <include name="server-config.xml" />
                    </replace>

                    <!-- Configure Mitre to let it know where the Open ID callback
needs to go to -->
                    <replace token="http://localhost/"
value="http://localhost:8080/openid-client/Callback"
dir="${tomcat.dir}/webapps/openid-connect-server-webapp/WEB-INF/classes/db/hsql"
summary="yes">
                        <include name="clients.sql" />
                    </replace>

                    <!-- Configure Tomcat using our pre-configured server.xml (which
sets https) -->
                    <copy file="src/test/resources/server.xml"
todir="${tomcat.dir}/conf"/>
                    <copy file="src/test/resources/localhost-rsa.jks"
todir="${tomcat.dir}/conf"/>

                    <chmod dir="${tomcat.dir}/bin" perm="ugo+rx" includes="*" />

                    <!-- Start Tomcat and Mitre -->
                    <exec executable="${tomcat.dir}/bin/startup.sh"
dir="${tomcat.dir}" >
                        <env key="CATALINA_PID" value="${tomcat.pidfile}" />
                    </exec>
                </target>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Out of the box Mitre runs on HTTP, but since we're using HTTPS instead we need to configure it to run on HTTPS using the `server-config.xml` file. We also need to tell it about the exact callback URL that we discussed above (`http://localhost:8080/openid-client/Callback`), which can be done in `clients.sql`. Tomcat has to be configured to run on HTTPS as well, which requires updating `server.xml` and providing it with a keystore.

Tomcat, and with it Mitre, can be started by executing `[tomcat dir]/bin/startup.sh`.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restOpenIdConnectAuth
```

This will run a test associated with the project, printing something like the following:

```
user : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 22.324 s - in
jakartaee.examples.focused.security.restopenidconnectauth.RestOpenIdConnectAuthIT
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestOpenIdConnectAuthIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        HtmlPage page = webClient.getPage(baseUrl + "/rest/resource"); ①

        // Authenticate with the OpenId Provider using the
        // username and password for a default user
        page.getElementById("j_username")
            .setAttribute("value", "user");

        page.getElementById("j_password")
```

```
            .setAttribute("value", "password"); ②

        // Submit
        HtmlPage confirmationPage =
            page.getElementByName("submit")
                .click(); ③

        // Confirm
        TextPage originalResource =
            confirmationPage.getElementByName("authorize")
                            .click(); ④

        System.out.println(originalResource.getContent());
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restOpenIdConnectAuth/target/restOpenIdConnectAuth.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

After the test requests our protected resource at "/rest/resource" (1), the OpenID Connect authentication mechanism redirects to Mitre, which will respond with a login page. The test programmically sets the fields `j_username` and `j_password`, (2) and then clicks submits (3). After confirming (4), Mitre will redirect the test code back to the `/Callback` URL, which will redirect back to the original resource at "/rest/resource".

Securing an endpoint with Basic authentication and an LDAP identity store

In the following example, we'll secure a REST endpoint using Basic authentication and the LDAP identity store that is provided by Jakarta Security.

You'll learn how to:

1. Define security constraints
2. Use the provided BasicAuthenticationMechanismDefinition
3. Use the provided LdapIdentityStoreDefinition
4. Populate and configure the identity store
5. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the authentication mechanism and identity store

```
@ApplicationScoped
@BasicAuthenticationMechanismDefinition(
    realmName = "basicAuth"
)
@LdapIdentityStoreDefinition(
    url = "ldap://localhost:40000",
    callerBaseDn = "ou=caller,dc=jakartaee",
    groupSearchBase = "ou=group,dc=jakartaee",
    groupSearchFilter = "(&(member=%s)(objectclass=groupofnames))"
)
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {
```

To declare the usage of a specific authentication mechanism, Jakarta EE provides `[XYZ]MechanismDefinition` annotations. Such an annotation is picked up by the security system, and in response to it a CDI bean that implements the HttpAuthenticationMechanism is enabled for it.

The annotation can be put on any bean, but in a REST application it fits particularly well on the

`Application` subclass because it also declares the path for REST resources.

Likewise, to declare the usage of a specific identity store, Jakarta EE provides `[XYZ]StoreDefinition` annotations.

The annotations can be put on any bean, but in a REST application it fits particularly well on the `Application` subclass that also declares the path for REST resources.

You can use the provided `@LdapIdentityStoreDefinition` with any authentication mechanism that validates username/password credentials. LDAP structures are very open-ended, and there's a lot of possible ways to model callers, their passwords, and their groups. We'll present one way here, where we'll define a `caller.jakartaee` object, that contains the caller name and password, and a `group.jakartaee` object that contains the group name and a list of all callers in that group.

For this structure we need 3 attributes to be defined:

1. `callerBaseDn` - Used for credential validation using "direct binding", with the default caller name being "uid".
2. `groupSearchBase` - The object root used to search for groups of the caller
3. `groupSearchFilter` - The subtree to search for groups

Populating the identity store

In order to use the identity store, we need to put some data in an LDAP server. The following code shows one way how to do that:

```
@ApplicationScoped
@BasicAuthenticationMechanismDefinition(
    realmName = "basicAuth"
)
@LdapIdentityStoreDefinition(
    url = "ldap://localhost:40000",
    callerBaseDn = "ou=caller,dc=jakartaee",
    groupSearchBase = "ou=group,dc=jakartaee",
    groupSearchFilter = "(&(member=%s)(objectclass=groupofnames))"
)
@DeclareRoles("user")
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

    private InMemoryDirectoryServer directoryServer;

    public void onStart(@Observes @Initialized(ApplicationScoped.class) Object
applicationContext) {
        try {
            InMemoryDirectoryServerConfig config = new
InMemoryDirectoryServerConfig("dc=jakartaee");
            config.setListenerConfigs(
                new InMemoryListenerConfig("myListener", null, 40000, null, null,
null));
```

```
            directoryServer = new InMemoryDirectoryServer(config);

            directoryServer.importFromLDIF(true,
                new LDIFReader(new ByteArrayInputStream("""

                        # Define caller.jakartaee and group.jakartaee structure

                        dn: dc=jakartaee
                        objectclass: top
                        objectclass: dcObject
                        objectclass: organization
                        dc: jakartaee
                        o: jakartaee

                        dn: ou=caller,dc=jakartaee
                        objectclass: top
                        objectclass: organizationalUnit
                        ou: caller

                        dn: ou=group,dc=jakartaee
                        objectclass: top
                        objectclass: organizationalUnit
                        ou: group


                        # Add caller john:secret1 and group user with member john

                        dn: uid=john,ou=caller,dc=jakartaee
                        objectclass: top
                        objectclass: uidObject
                        objectclass: person
                        uid: john
                        cn: John Smith
                        sn: John
                        userPassword: secret1

                        dn: cn=user,ou=group,dc=jakartaee
                        objectclass: top
                        objectclass: groupOfNames
                        cn: user
                        member: uid=john,ou=caller,dc=jakartaee

                        """.getBytes())));

            directoryServer.startListening();
        } catch (LDAPException e) {
            throw new IllegalStateException(e);
        }
    }
```

```
}
```

The code above uses an in-memory LDAP server called Unboundid that we start on port 40000. We populate it using embedded LDIF, which is a popular format to configure LDAP servers.

In-depth explanation of LDAP itself is beyond the scope of this tutorial, but we'll briefly explain the process here. When a caller authenticates with username "john" and password "secret1", our LDAP identity store will construct the full name "uid=john,ou=caller,dc=jakartaee" using `callerNameAttribute` ("uid" as a default), and `callerBaseDn` ("ou=caller,dc=jakartaee" here). The store then uses the LDAP 'Bind' operation and directly attempts to "log in" as that user to the LDAP server. Unlike the Database identity store, the LDAP store doesn't look up and compare the passwords. If the aforementioned login succeeds, the credentials are assumed to be correct.

The LDAP store will then search for groups using the `javax.naming.directory.DirContext.search()` method, with the `groupSearchBase` value ("ou=group,dc=jakartaee") and the formatted value from `groupSearchFilter` ("(&(member=uid=john,ou=caller,dc=jakartaee)(objectclass=groupofnames))") as parameters. The LDAP server will subsequently return "cn=user,ou=group,dc=jakartaee" for our example. From this the value of `groupNameAttribute` (defaults to "cn") is taken, which resolves to "user" here.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restBasicAuthLdapStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.247 s - in
jakartaee.examples.focused.security.restBasicAuthLdapStore.RestBasicAuthLdapStoreIT
```

The test itself is basically the same as that for the Securing an endpoint with Basic authentication example.

Securing an endpoint with Custom Form authentication

In the following example, we'll secure a REST endpoint using Custom Form authentication.

You'll learn how to:

1. Define security constraints
2. Use the Custom Form authentication mechanism
3. Use Jakarta Faces and Jakarta Validation to customize the Form used for Form authentication

4. How to define (and implicitly set) a custom identity store
5. Use the Jakarta Security SecurityContext

Write the application

Let's start with defining a simple REST resource class for a `/rest/resource` endpoint:

```
@Path("/resource")
@RequestScoped
public class Resource {

    @Inject
    private SecurityContext securityContext;

    @GET
    @Produces(TEXT_PLAIN)
    public String getCallerAndRole() {
        return
            securityContext.getCallerPrincipal().getName() + " : " +
            securityContext.isCallerInRole("user");
    }

}
```

This resource uses the injected Jakarta EE SecurityContext to obtain access to the current authenticated caller, which is represented by a `Principal` instance.

If this resource were available to unauthenticated callers, `getCallerPrincipal()` would return `null` for unauthenticated requests, so we'd have to check for `null`. Our example, however, requires authentication for this resource, so we can skip that check.



There is a Jakarta REST-specific type that is also named SecurityContext and has similar methods as the ones we used here. From the Jakarta EE perspective, that is a discouraged type and the Jakarta Security version is to be preferred.

Declare the security constraints

Next we'll define the security constraints in `web.xml`, which tell the security system that access to a given URL or URL pattern is protected, and hence authentication is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <security-constraint>
```

```
        <web-resource-collection>
            <web-resource-name>protected</web-resource-name>
            <url-pattern>/rest/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

</web-app>
```

This XML essentially says that to access any URL that starts with "/rest" requires the caller to have the role "user". Roles are opaque strings; merely identifiers. It's fully up to the application how broad or fine-grained they are.



In Jakarta EE, internally these XML constraints are transformed into `Permission` instances and made available via a specific type of permission store. Knowledge about this transformation is only needed for very advanced use cases.



The observant reader may wonder if XML is really the only option here, given the strong feelings that exist in parts of the community around XML. The answer is yes and no. Jakarta EE does define the @RolesAllowed annotation that could be used to replace the XML shown above, but only the legacy Enterprise Beans has specified a behaviour for this when put on an Enterprise Bean. Jakarta REST has done no such thing, although the JWT API in MicroProfile has defined this for REST resources. In Jakarta EE, however, this remains a vendor-specific extension. There are also a number of annotations and APIs in Jakarta EE to set these kinds of constraints for individual Servlets, but those won't help us much either here.

Declare the authentication mechanism

```
@ApplicationScoped
@CustomFormAuthenticationMechanismDefinition(
    loginToContinue = @LoginToContinue(
        loginPage="/login.xhtml",
        errorPage=""
    )
)
@DeclareRoles({ "user", "caller" })
@FacesConfig
@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

}
```

To declare the usage of a specific authentication mechanism, Jakarta EE provides `[XYZ]MechanismDefinition` annotations. Such an annotation is picked up by the security system, and

in response to it a CDI bean that implements the HttpAuthenticationMechanism is enabled for it.

The annotation can be put on any bean, but in a REST application it fits particularly well on the `Application` subclass because it also declares the path for REST resources.

Contrary to the Basic HTTP authentication mechanism, the Form authentication mechanism allows us to customize the login dialog (the process between the caller and the authentication mechanism) and to keep track of the authenticated session on the server (using a cookie). This also allows us to logout, something that for unknown reasons has never been specified for Basic HTTP authentication.

Contrary to the regular Form authentication mechanism, the Custom Form authentication mechanism lets us customize the login dialog even more by having the ability to execute custom code between the postback of a login form and the form authentication mechanism taking the provided credentials.

To use this authentication method, we need to designate a path to a resource that is relative to our application. The authentication mechanism will redirect the caller to this resource when authentication is required. The resource can be anything, but a postback should eventually lead to some code being executed that continues the authentication dialog. For example a plain .html file or .jsp file combined with a Filter, or a Faces view with a backing bean.

Define the authentication mechanism's view and backing code

For this example we'll use a Faces view with a backing bean:

```
<!DOCTYPE html>
<html lang="en" xmlns:h="jakarta.faces.html">
    <h:head>
        <title>Login to continue</title>
    </h:head>
    <h:body>
        <h1>Login to continue</h1>

        <h:messages />

        <h:form id="form">
            <div>
                <h:outputLabel for="username" value="Username" />
                <h:inputText id="username" value="#{loginBacking.username}"/>
            </div>
            <div>
                <h:outputLabel for="password" value="Password" />
                <h:inputSecret id="password" value="#{loginBacking.password}"/>
            </div>
            <div>
                <h:commandButton value="Login" type="submit"
action="#{loginBacking.login}" />
            </div>
        </h:form>
```

```
    </h:body>
</html>
```

```
@Named
@RequestScoped
public class LoginBacking {

    @Inject
    private SecurityContext securityContext;

    @Inject
    private FacesContext facesContext;

    @NotNull
    @Size(min = 3, max = 15, message="Username must be between 3 and 15 characters")
    private String username;

    @NotNull
    @Size(min = 5, max = 50, message="Password must be between 5 and 50 characters")
    private String password;

    public void login() {
        switch (
            // Continue the authentication dialog manually by invoking the
authenticate()
            // method. The form authentication picks this up, just like a post to
j_security does.
            securityContext.authenticate(
                getRequest(),
                getResponse(),
                withParams()
                    .credential(new UsernamePasswordCredential(username, new
Password(password))))) {

            case SEND_CONTINUE:

                // Authentication mechanism has send a redirect, should not
                // send anything to response from Faces now.
                facesContext.responseComplete();
                return;

            case SEND_FAILURE:

                addError("Login failed");
                return;

            default:
        }
    }
```

```
    // getters/setters + utility methods omitted
}
```

The view itself is quite similar to the HTML page we used for the form in Securing an endpoint with Form authentication. The main difference is bindings of the form fields to a (CDI) backing bean. The bean side of the binding has Jakarta Validation constraints applied to it; this allows for fine-grained validation of some general requirements of the credentials without actually attempting authentication.

If this initial validation passes, we arrive in the `login()` method. For our example the only thing we need to do here is signaling that we want to continue the authentication dialog (the process or interaction between the caller and the authentication mechanism), and when doing so provide the credentials that we earlier obtained in a custom way.

We have two important outcomes to handle. `SEND_CONTINUE` effectively means the credentials were validated successfully, and the caller is therefore directed to the resource that was originally requested. `SEND_FAILURE` means the opposite; the credentials were not validated successfully. By just returning from our callback method in the last case the form will be redisplayed, albeit with the error message added.



`NOT_DONE` and `SUCCESS` are two other outcomes, but we don't have to handle them here. `NOT_DONE` only applies to pre-emptive authentication, but we're doing explicit (forced, mandatory) authentication here. `SUCCESS` means we can go ahead and render the page, which is exactly what happens if we don't do anything.

See [web:webapp::webapp:::_getting_started_with_web_applications] for more information on running `.xhtml` files and their backing beans.

Define the identity store

Finally, let's define a basic identity store that the security system can use to validate provided credentials for Form authentication:

```
@ApplicationScoped
public class CustomIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
usernamePasswordCredential) {
        if (usernamePasswordCredential.compareTo("john", "secret1")) {
            return new CredentialValidationResult("john", Set.of("user", "caller"));
        }

        return INVALID_RESULT;
    }

}
```

This identity store only validates the single identity (user) "john", with password "secret1" and roles

"user" and "caller". Defining this kind of identity store is often the simplest way to get started. Note that Jakarta Security doesn't define a simple identity store out of the box, because there are questions about whether that would promote security best practices.

The identity store is installed and used by the security system just by the virtue of being there; it picks up all enabled CDI beans that implement IdentityStore. Such beans can be enabled by the security system itself (following some configuration annotation), or can be programmatically added using the appropriate CDI APIs. Where the bean comes from doesn't matter for Jakarta Security, only the fact that it's there.

Test the application

It's now time to test our application. A ready-to-test version is available from the Jakarta EE Examples project at https://github.com/eclipse-ee4j/jakartaee-examples.

Download or clone this repo, then cd into the `focused` folder and execute:

```
mvn clean install -pl :restCustomFormAuthCustomStore
```

This will run a test associated with the project, printing something like the following:

```
john : true
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 9.272 s - in
jakartaee.examples.focused.security.restCustomFormAuthCustomStore.RestCustomFormAuthCu
stomStoreIT
```

Let's take a quick look at the actual test:

```
@RunWith(Arquillian.class)
@RunAsClient
public class RestCustomFormAuthCustomStoreIT extends ITBase {

    @ArquillianResource
    private URL baseUrl;

    /**
     * Test the call to a protected REST service
     *
     * @throws Exception when a serious error occurs.
     */
    @RunAsClient
    @Test
    public void testRestCall() throws Exception {
        HtmlPage loginPage = webClient.getPage(baseUrl + "/rest/resource");
        System.out.println(loginPage.asXml());

        HtmlForm form = loginPage.getForms()
                                 .get(0);
```

```
        form.getInputByName("form:username")
            .setValueAttribute("john");

        form.getInputByName("form:password")
            .setValueAttribute("secret1");

        TextPage page = form.getInputByValue("Login")
                            .click();

        System.out.println(page.getContent());
    }
}
```

The test starts a server and deploys the output of the build process (a .war file) to it. The test runs in the integration test phase, rather than the unit test phase, to make sure this build output is available when it runs. The test then sends a request to the server using the provided HtmlUnit `webClient`. Note that the `webClient` can be used for any other HTTP requests your test requires.

If you want to inspect the app yourself, you can manually deploy the WAR file (`security/restCustomFormAuthCustomStore/target/restCustomFormAuthCustomStore.war`) to the server of your choice (e.g. GlassFish 7), and request the URL via a browser or a commandline util such as `curl`.

The test first sends a request here to the protected resource, and the server responds with the rendered version of the Faces view form we defined above. Using the `HtmlUnit` API, it's easy to navigate the HTML DOM, fill out the username and password in the form, and programmatically click the `Login` button. The form posts back to the same URL it was requested from. Faces will detect this postback and will orchestrate the validation using Jakarta Validation and invoking the CDI based backing bean.

Jakarta Servlets

Getting Started with Web Applications

The Web Profile allows you to get started developing Java web applications, which typically use Jakarta Servlet API as corner stone.

Web Applications

A web application, also known as a web app, is a software application that runs on one or more web servers. It is typically accessed through a web browser over a network, such as the Internet. The advantage over desktop and mobile applications is being platform-independent, as it can be accessed and used on different devices. Web applications are of the following types:

Presentation-oriented

A presentation-oriented web application (also called a "website") generates dynamic web pages in response to HTTP requests. The response is usually represented as a HTML document along

with assets, such as Cascading Style Sheets (CSS), JavaScript (JS) and images. Presentation-oriented applications are often directly used by humans. Development of presentation-oriented web applications is covered in "Building Web Services with Jakarta XML Web Services" (available in a previous version of the tutorial) through Jakarta Servlet Technology

Service-oriented

A service-oriented web application (also called a "service") generates dynamic data structures in response to HTTP requests. The response is usually represented as a JSON object or as a XML document or even as plain text. Service-oriented applications are often directly used by presentation-oriented web applications or other service-oriented applications. Development of service-oriented web applications is covered in "Building Web Services with Jakarta XML Web Services" (available in a previous version of the tutorial)

In the Jakarta EE platform, web applications are represented by web components as seen in Jakarta Web Application Request Handling. A web component can be represented by Jakarta Servlet, Jakarta Faces or Jakarta REST.

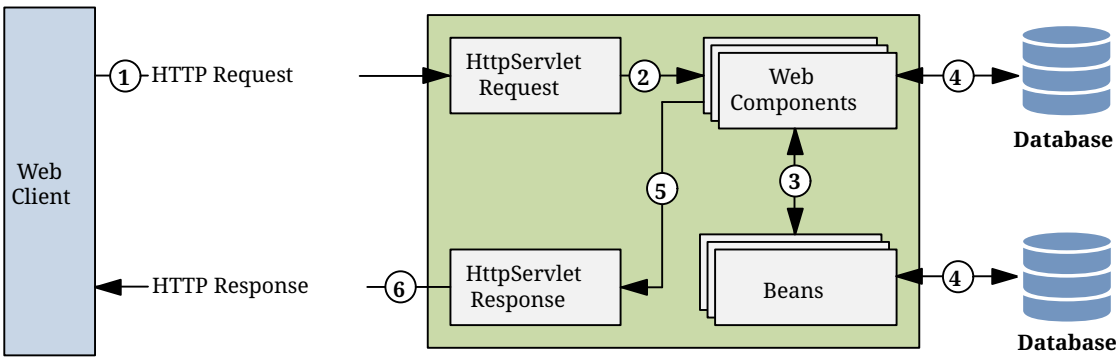


Figure 8. Jakarta Web Application Request Handling

1. The client sends an HTTP request to the web server.
2. The web server module that supports Jakarta Servlet-based web components is called a servlet container.
3. The servlet container converts the HTTP request into an `HttpServletRequest` object and prepares the `HttpServletResponse` object.
4. These objects are delivered to a web component, which can interact with beans or a database to generate dynamic content.
5. The web component can fill the `HttpServletResponse` object with the generated dynamic content or can pass the object to another web component to fill it.
6. The servlet container ultimately converts the `HttpServletResponse` object to an HTTP response and the web server returns it to the client.

Jakarta Servlet can be used to build both presentation and service-oriented web applications. It intends to reduce the boilerplate code needed to convert the HTTP request into a Java object and to offer a Java object as an HTTP response, and to manage all the lifecycle around them.

Jakarta Faces is a component-based MVC framework that can be executed on a servlet container and act as a presentation-oriented web application. It intends to reduce the boilerplate code needed to collect the request parameters, convert and validate them, update bean properties with them,

invoke bean methods, and generate HTML output as response. It is designed to run in servlet and non-servlet environments such as a portlet container.[2]

Jakarta REST is a RESTful web service framework that can be executed on a servlet container and act as a service-oriented web application. It intends to reduce the boilerplate code needed to convert the request parameters to a bean and further convert it to the desired response such as JSON or XML. It is designed to run in servlet and non-servlet environments such as a microservice.[3]

A servlet container can be part of a Jakarta runtime such as an application server. Certain aspects of web component behavior can be configured when the web application is installed, or deployed, to a servlet container. The configuration information can be specified using annotations or can be maintained in a XML file called a deployment descriptor. A deployment descriptor file must comply to the schema described in the specification associated with the web component. When the same configuration is specified using annotations and in a deployment descriptor file, then the configuration in the deployment descriptor file will always have precedence.

This chapter gives a brief overview of the activities involved in developing Jakarta Servlet-based web applications. It explains how to compile, package, deploy, and run Jakarta Servlet-based web applications in a servlet container.

The Web Application Archive

A Jakarta Servlet-based web application can contain one or more of the following parts:

- One or more web components, which can be represented by Jakarta Servlet, Jakarta Faces or Jakarta REST.
- Assets (also called static resource files), such as Cascading Style Sheets (CSS), JavaScript (JS) and images.
- Dependencies (also called helper libraries, third party libraries, "JARs").
- Deployment descriptor files.

The process for creating, deploying, and executing a Jakarta Servlet-based web application is different from that of Java classes which are packaged and executed as a Java application archive (JAR). It can be summarized as follows:

1. Develop the web component code.
2. Develop the deployment descriptor files, if necessary.
3. Compile the web component code against the libraries of the servlet container and the helper libraries, if any.
4. Package the compiled code along with helper libraries, assets and deployment descriptor files, if any, into a deployable unit, called a web application archive (WAR).
5. Deploy the WAR into a servlet container.
6. Run the web application by accessing a URL that references the web component.

Developing the web component code and deployment descriptor files is covered in the later chapters. Steps 3 through 6 are expanded on in the following sections and illustrated with two web

applications, in Hello World–style. The web applications take a name as an HTTP request parameter, generate the greeting and return it as an HTTP response. This chapter discusses the following simple web applications:

- `hello-servlet`, a Jakarta Servlet-based service-oriented web application
- `hello-faces`, a Jakarta Faces-based presentation-oriented web application

They are used to illustrate tasks involved in compiling, packaging, deploying, and running a Jakarta Servlet-based web application that contains web component code.

Building, Deploying and Running The Example Projects

This describes steps 3 through 6. You need Maven and a servlet container.

Building The Example Projects

This describes steps 3 and 4. Maven is used to build the example projects. Download and install it as per instructions in maven.apache.org. A Maven WAR project has the following basic structure; not all mentioned resources are required, only the `pom.xml` is required and the rest is optional:

```
|-- src
|    `-- main
|         |-- java
|         |    `-- com
|         |         `-- example
|         |              |-- controller
|         |              |    `-- Servlet.java
|         |              `-- model
|         |                   `-- Bean.java
|         |-- resources
|         |    `-- com
|         |         `-- example
|         |              `-- i18n
|         |                   |-- text.properties
|         |                   |-- text_en.properties
|         |                   `-- text_es.properties
|         `-- webapp
|              |-- META-INF
|              |    `-- MANIFEST.MF
|              |-- resources
|              |    |-- css
|              |    |    `-- style.css
|              |    |-- img
|              |    |    `-- logo.svg
|              |    `-- js
|              |         `-- script.js
|              |-- WEB-INF
|              |    |-- beans.xml
|              |    |-- faces-config.xml
|              |    `-- web.xml
```

```
|             |-- favicon.ico
|             `-- index.xhtml
`-- pom.xml
```

In a terminal window, go to the folder containing the `pom.xml` and execute the following command:

```
mvn install
```

This command compiles and packages the code as described in steps 3 and 4. After the build, the resulting WAR file will be present in the automatically created `target` folder. The WAR file is recognizable by having the `.war` extension.

Deploying The Example Projects

This describes step 5. You need at least a servlet container in order to deploy a WAR file. The Jakarta EE Compatible Products page lists several Jakarta runtimes having a servlet container. At least those compatible with "Web Profile" and "Jakarta EE Platform" have a servlet container. Examples are Eclipse GlassFish, IBM Open Liberty and Red Hat WildFly. They are available in both "Web Profile" and "Jakarta EE Platform" flavors.

There are also partial Jakarta runtimes having a servlet container. Partial as in, they do not fully implement "Web Profile". One example is Apache Tomcat. It does implement Jakarta Servlet, Jakarta Pages, Jakarta Expression Language, Jakarta WebSocket, and Jakarta Security. But it does not implement Jakarta REST, Jakarta CDI, Jakarta Faces nor Jakarta Tags.

Deploying a WAR file is generally done by placing the WAR file in the runtime-specific folder dedicated for (automatic) deployments. The exact location depends on the runtime used:

- In case of Eclipse GlassFish that's the `glassfish/domains/domain1/autodeploy` folder.
- In case of IBM Open Liberty that's the `config/dropins` folder.
- In case of Red Hat WildFly that's the `standalone/deployments` folder.
- In case of Apache Tomcat that's the `webapps` folder.

Refer to the documentation for your runtime for the most recent information.

Undeploying a WAR file is generally done by removing the WAR file from the folder.

Some runtimes offer a graphical user interface (GUI) for administration tasks such as selecting, uploading and a deploying a WAR file. For this you'll need to start the server first and then open a specific URL in your web browser representing the location of the admin console:

- In case of Eclipse GlassFish that's reachable via `http://localhost:4848`
- In case of IBM Open Liberty that's reachable via `https://localhost:9443`
- In case of Red Hat WildFly that's reachable via `http://localhost:9990`
- In case of Apache Tomcat that's reachable via `http://localhost:8080/manager`

Refer to the documentation for your runtime for the most recent information.

You can also use an integrated development environment (IDE) to manage runtimes and deployments. Examples are Eclipse IDE, IntelliJ IDEA and Apache NetBeans. They can also automatically build the projects for you.

Running The Example Projects

This describes step 6. You'll need to start the server first and then open a specific URL in your web browser representing the location of the WAR deployment. By default, the URL has the following form:

```
http(s)://host:port/context-path
```

By default, the `context-path` is represented by the base file name of the WAR file, without the extension. If there is no web component listening on the root of the context path, then you could face an HTTP 404 'Not Found' error page. In that case you would need to use a more specific URL, depending on the configuration of the desired web component. This will be detailed in Mapping URLs to Web Components.

Hello World Web Application Using Jakarta Servlet

The `hello-servlet` application is a web application that uses Jakarta Servlet take a name as an HTTP request parameter, generate the greeting and return it as an HTTP response. As a service-oriented web application, the response is in this minimal example represented as plain text.

The source code for this application is in the `jakartaee-examples/tutorial/web/servlet/hello-servlet/` directory.

The Servlet

In a typical Jakarta Servlet-based web application, the class must extend `jakarta.servlet.http.HttpServlet` and override one of the `doXxx()` methods where `Xxx` represents the HTTP method of interest. Such a class is in the Jakarta Servlet world called a *Servlet*.

For this Hello World web application, the `src/main/java/jakarta/tutorial/web/servlet/Greeting.java` servlet listens on HTTP GET requests and extracts the `name` request parameter as input and creates the `greeting` as output.

```
package jakarta.tutorial.web.servlet;

import java.io.IOException;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet("/greeting")
public class Greeting extends HttpServlet {
```

```
    @Override
    public void doGet
        (HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        var name = request.getParameter("name");

        if (name == null || name.isBlank()) {
            response.sendError(HttpServletResponse.SC_BAD_REQUEST);
            return;
        }

        var greeting = "Hello, " + name + "!";

        response.setContentType("text/plain");
        response.getWriter().write(greeting);
    }
}
```

Mapping URLs to Web Components

When the servlet container receives a request, it must determine which web component should handle the request. The servlet container does so by mapping the URL contained in the request to a web component. A URL contains the context path and, optionally, a URL pattern:

```
http(s)://host:port/context-path[/url-pattern]
```

You can set the URL pattern for a servlet by using the `@WebServlet` annotation in the servlet source file or by using `<servlet-mapping>` entry in the Jakarta Servlet deployment descriptor file, the `src/main/webapp/WEB-INF/web.xml`. In the `Greeting` servlet example the `@WebServlet` annotation indicates that the URL pattern is `/greeting`. Therefore, when the servlet is deployed to a local server listening on `http://localhost:8080` with the context path set to `hello-servlet`, it is accessed with the following URL:

```
http://localhost:8080/hello-servlet/greeting
```

The Hello World example will return an HTTP 400 error as response indicating a 'Bad Request'. When specifying the name as a request parameter in the following URL:

```
http://localhost:8080/hello-servlet/greeting?name=Duke
```

Then it will return the response `Hello, Duke!`.

Running the `hello-servlet` example application

Build and deploy as instructed in Building, Deploying and Running The Example Projects.

In a web browser, open the following URL:

```
http://localhost:8080/hello-servlet/greeting?name=Duke
```

It will return the response `Hello, Duke!`. You can edit the `name` parameter into something else to get a different response.

```
http://localhost:8080/hello-servlet/greeting?name=Joe
```

This will return the response `Hello, Joe!`.

Hello World Web Application Using Jakarta Faces

The `hello-faces` application is a web application that uses Jakarta Faces take a name as an HTTP request parameter, generate the greeting and return it as an HTTP response. As a presentation-oriented web application, the response is represented as a HTML document.

The source code for this application is in the `jakartaee-examples/tutorial/web/faces/hello-faces/` directory.

Jakarta Faces is a component-based MVC framework that provides its own servlet as controller (the "C" part of MVC), the `FacesServlet`. The model and the view are usually provided by you, the web developer.

The Model

In a typical Jakarta Faces application, the model (the "M" part of MVC) is represented by a bean class. Such a bean class is in the Jakarta Faces world called a *Backing Bean*.

For this Hello World web application, the `src/main/java/jakarta/tutorial/web/faces/Hello.java` backing bean defines a `name` property along with a getter and setter, a `greeting` property along with a getter, and a `submit` action method which creates the `greeting` as output-based on `name` as input.

```
package jakarta.tutorial.web.faces;

import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Hello {

    private String name;
    private String greeting;
```

```
    public void submit() {
        greeting = "Hello, " + name + "!";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getGreeting() {
        return greeting;
    }
}
```

Note that a getter is required for output and that a setter is only required for input.

In a typical Jakarta Faces application, CDI is used to manage the backing beans. We'll briefly go through the CDI annotations that are used here:

- `@Named` — gives the backing bean a managed bean name, which is primarily used to reference it in Expression Language (EL). Without any attributes this defaults to the simple class name with the first letter in lowercase. This backing bean will thus be available as a managed bean via `#{hello}` in EL.
- `@RequestScoped` — gives the backing bean a managed bean scope, which essentially represents its lifespan. In this case that lifespan is the duration of an HTTP request. When the HTTP request ends, then the managed bean instance is destroyed.

The View

In a typical Jakarta Faces application, the view (the "V" part of MVC) is represented by a XHTML file. Such a XHTML file is in the Jakarta Faces world called a *Facelets File* or just *Facelet*. In Jakarta Faces, the view technology is pluggable and Facelets is used as the default view technology. XHTML markup is being used because it allows the framework to easily use a XML parser to find Jakarta Faces tags of interest and to generate HTML output.

For this Hello World web application, the `src/main/webapp/hello.xhtml` Facelet defines a form with an input field, a command button and an output field.

```
<!DOCTYPE html>
<html lang="en"
      xmlns:f="jakarta.faces.core"
      xmlns:h="jakarta.faces.html">
    <h:head>
        <title>Jakarta Faces Hello World</title>
    </h:head>
    <h:body>
```

```
        <h1>Hello, what's your name?</h1>
        <h:form>
            <h:inputText value="#{hello.name}" required="true" />
            <h:commandButton value="Submit" action="#{hello.submit}">
                <f:ajax execute="@form" render=":greeting" />
            </h:commandButton>
            <h:messages />
        </h:form>
        <h:outputText id="greeting" value="#{hello.greeting}" />
    </h:body>
</html>
```

We'll briefly go through the Jakarta Faces-specific XHTML tags that are used here.

- `<h:head>` — generates the HTML `<head>`. It gives Jakarta Faces the opportunity to automatically include any necessary JS and CSS files in the generated HTML head.
- `<h:body>` — generates the HTML `<body>`. It gives Jakarta Faces the opportunity to automatically include any necessary JS files in the end of the generated HTML body.
- `<h:form>` — generates the HTML `<form>`. It gives Jakarta Faces the opportunity to automatically include a hidden field representing the view state.
- `<h:inputText>` — generates the HTML `<input>`. It gives Jakarta Faces the opportunity to automatically get and set the value as a managed bean property specified in the `value` attribute, as well as to perform any conversion and validation on it.
- `<h:commandButton>` — generates the HTML `<input type="submit">`. It gives Jakarta Faces the opportunity to automatically invoke the backing bean method specified in the `action` attribute.
- `<h:messages>` — generates the HTML `<ul>` or `<div>` or `<table>` depending on state and configuration. It gives you the opportunity to declare the place where any conversion and validation messages will be displayed.
- `<f:ajax>` — generates the necessary JavaScript code for Ajax behavior. If gives you the opportunity to configure the form submit to be performed asynchronously using Ajax.
- `<h:outputText>` — generates the HTML `<span>`. This is the one being updated on completion of the Ajax submit and it will display the current value of the managed bean property specified in the `value` attribute.

The `<h:···>` tags define standard HTML components. The `<f:···>` tags define behavior.

The input value is via EL value expression `#{hello.name}` connected to the `name` property of the `Hello` backing bean via its getter and setter. The output value is via EL value expression `#{hello.greeting}` connected to the `greeting` property of the `Hello` backing bean via its getter. The command button is via EL method expression `#{hello.submit}` connected to the `submit` method of the `Hello` backing bean.

Note that for EL value expressions the physical field representing the property doesn't need to have exactly the same name nor that it needs to exist at all. EL only looks for the getter and setter methods, not for the field.

The Controller

In a typical Jakarta Faces application, the controller (the "C" part of MVC) is registered in the Jakarta Servlet deployment descriptor file, the `src/main/webapp/WEB-INF/web.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">
    <servlet>
        <servlet-name>facesServlet</servlet-name>
        <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>facesServlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
</web-app>
```

It basically instructs the servlet container to create an instance of `jakarta.faces.webapp.FacesServlet` as `facesServlet` during startup and to execute it when the URL pattern of the HTTP request matches `*.xhtml`.

Running the `hello-faces` example application

Build and deploy as instructed in Building, Deploying and Running The Example Projects.

In a web browser, open the following URL:

```
http://localhost:8080/hello-faces/hello.xhtml
```

It will show a web page with a form asking for your name. Enter your name in the input field of the form, for example 'Duke', and click Submit. This will show the text `Hello, Duke!` below the form.

Jakarta Servlet Deployment Descriptor

This section describes the following tasks involved with configuring Jakarta Servlet-based web applications:

- Preparing deployment descriptor
- Setting context parameters
- Declaring welcome files
- Mapping errors to error screens
- Declaring resource references

Preparing Deployment Descriptor

The Jakarta Servlet deployment descriptor is represented by the `WEB-INF/web.xml` file which is in a Maven-based project placed in `src/main/webapp` folder. It must be a XML file with a `<web-app>` root element conform the `web-app_X_Y.xsd` XML schema of the https://jakarta.ee/xml/ns/jakartaee namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">

    <!-- Put configuration here -->

</web-app>
```

Setting Context Parameters

The web components in a web application share an object that represents their application context, the `jakarta.servlet.ServletContext`. You define context-wide initialization parameters in the `web.xml` file.

```
<context-param>
    <param-name>com.example.THEME</param-name>
    <param-value>blue</param-value>
</context-param>
```

You can obtain them in application code via `getInitParameter(String name)` method of the `ServletContext` instance.

```
String theme = servletContext.getInitParameter("com.example.THEME");
```

These context parameters can be used to configure (default) application-wide variables, such as the web application's project stage, the path to save uploaded files, the path to a more specific configuration file, the maximum size of a cache, the expiration time of a cacheable resource, the name of the tenant/theme/scheme, or even helper library specific parameters, etcetera, but never passwords. It are those variables which you'd normally define as constants in a Java class, but then without the need to recompile Java classes whenever you'd like to edit them.

Declaring Welcome Files

The welcome files mechanism allows you to specify a list of files that the servlet container can return as a default resource when any folder is requested which does not map to an existing web component. For example, suppose that you define two welcome files `index.xhtml` and `index.html`. So, if for example the `/` folder is requested, and it does not map to a web component, then it'll search

for `/index.xhtml` and `/index.html` files and return the first found one. Or if there is none, then the servlet container will continue to perform the default behavior, which is usually displaying an HTTP 404 error page. Note that this also applies to sub folders, so if for example the `/foo` folder is requested, then it'll search for `/foo/index.xhtml` and `/foo/index.html` files and return the first found one. You specify welcome files in the `web.xml` file.

```
<welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

A specified welcome file must not have a leading slash (/) as it is interpreted independently of the requested folder.

Mapping Errors to Error Pages

When an error occurs during execution of a web application, you can have the application display a specific error page according to the type of error. In particular, you can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any web component and any type of error page. You specify error pages in the `web.xml` file.

```
<error-page>
    <error-code>401</error-code>
    <location>/WEB-INF/error-pages/unauthorized.xhtml</location>
</error-page>
<error-page>
    <error-code>403</error-code>
    <location>/WEB-INF/error-pages/unauthenticated.xhtml</location>
</error-page>
<error-page>
    <error-code>404</error-code>
    <location>/WEB-INF/error-pages/not-found.xhtml</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/WEB-INF/error-pages/general-error.html</location>
</error-page>
<error-page>
    <exception-type>jakarta.faces.application.ViewExpiredException</exception-type>
    <location>/WEB-INF/error-pages/view-expired.html</location>
</error-page>
<error-page>
    <!-- No error-code or exception-type, i.e. this will match any other HTTP status
than defined above -->
    <location>/WEB-INF/error-pages/unknown-error.html</location>
</error-page>
```

A specified location must have a leading slash (/) as it is interpreted as an absolute path within the context of the web application deployment. The error page being placed in `/WEB-INF` folder is not technically required, but it is the best practice as it will prevent the client from being able to request or even bookmark them individually.

Declaring Resource References

If your web application uses external resources, such as data sources or mail sessions, then you can specify it in `web.xml` and use the `@javax.annotation.Resource` annotation to inject it into a container-managed object of your web application.

The `@Resource` annotation can be specified on a class, a method, or a field. The container is responsible for instantiating and injecting references to resources declared by the `@Resource` annotation in container-managed objects and mapping it to the proper JNDI resources.

Container-managed objects are primarily those objects which you do not explicitly instantiate yourself as in `new ContainerManagedObject()` but basically let the container do. For example CDI managed beans in case of a CDI container, and web components in case of a servlet container. CDI managed beans are recognizable as classes having a CDI scope annotation such as `@RequestScoped` or `@Dependent`. Web components are recognizable as classes that extend or implement a class or interface of the `jakarta.servlet.` package, and are registered via an annotation of the `jakarta.servlet.annotation.` package or an entry in the `web.xml` file.

Declaring Data Source References

If your web application uses data sources, then you can specify it in `web.xml` and use the `@Resource` annotation to inject it into a container-managed object of your web application as a `javax.sql.DataSource` instance. The below example specifies a H2 data source listening on JNDI resource name of `java:global/YourDataSourceName`, using the `org.h2.jdbcx.JdbcDataSource` as `javax.sql.DataSource` implementation.

```
<data-source>
    <name>java:global/YourDataSourceName</name>
    <class-name>org.h2.jdbcx.JdbcDataSource</class-name>
    <url>jdbc:h2:mem:test</url>
</data-source>
```

If you have only one data source reference specified, then you can inject it as a `@Resource` of `javax.sql.DataSource` type without an explicit JNDI resource name.

```
@Resource
private DataSource dataSource;

public Connection getConnection() {
    return dataSource.getConnection();
}
```

If you have more than one data source reference specified, then you need to explicitly specify the

JNDI resource name.

```
@Resource(name="java:global/YourDataSourceName")
private DataSource dataSource;

public Connection getConnection() {
    return dataSource.getConnection();
}
```

Do note that `javax.sql.DataSource` is not part of Jakarta EE but of Java SE and hence it has still the `javax` as root package.

Declaring Mail Session References

If your web application uses mail sessions, then you can specify it in `web.xml` and use the `@Resource` annotation to inject it into a container-managed object of your web application as a `jakarta.mail.Session` instance. The below example specifies a SMTP mail session listening on JNDI name of `java:global/YourMailSessionName`, using the `smtp.example.com` host to create `jakarta.mail.Session` for.

```
<mail-session>
    <name>java:global/YourMailSessionName</name>
    <host>smtp.example.com</host>
    <user>user@example.com</user>
</mail-session>
```

If you have only one mail session reference specified, then you can inject it as a `@Resource` of `jakarta.mail.Session` type without an explicit JNDI resource name.

```
@Resource
private Session session;

public void sendMail(YourMail mail) throws MessagingException {
    Message message = new MimeMessage(session);
    // ...
}
```

If you have more than one mail session reference specified, then you need to explicitly specify the JNDI resource name.

```
@Resource(name="java:global/YourMailSessionName")
private Session session;

public void sendMail(YourMail mail) throws MessagingException {
    Message message = new MimeMessage(session);
    // ...
```

```
}
```

Further Information about Web Applications

For more information on web applications, see

- Jakarta Servlet 6.0 specification: https://jakarta.ee/specifications/servlet/6.0/
- Jakarta Faces 4.0 specification: https://jakarta.ee/specifications/faces/4.0/

Jakarta Servlet

Jakarta Servlet is a corner stone web framework that can act as a presentation-oriented as well as a service-oriented web application. Jakarta Servlet intends to reduce the boilerplate code needed to convert the HTTP request into a Java object and to offer a Java object as an HTTP response, and to manage all the lifecycle around them.

What Is a Servlet?

A servlet is a Java programming language class that directly or indirectly implements the `jakarta.servlet.Servlet` interface. The `jakarta.servlet` and `jakarta.servlet.http` packages provide interfaces and classes for writing servlets. All servlets must implement the `jakarta.servlet.Servlet` interface, which defines lifecycle methods such as `init`, `service`, and `destroy`. When implementing a generic service, you can extend the `jakarta.servlet.GenericServlet` class which already implements the `Servlet` interface. When implementing an HTTP service, you can extend the `jakarta.servlet.http.HttpServlet` class which already extends the `GenericServlet` class.

In a typical Jakarta Servlet based web application, the class must extend `jakarta.servlet.http.HttpServlet` and override one of the `doXxx` methods where `Xxx` represents the HTTP method of interest.

Servlet Lifecycle

The lifecycle of a servlet is controlled by the servlet container in which the servlet has been deployed. When a request is mapped to a servlet, the servlet container performs the following steps:

1. If an instance of the servlet does not exist, the servlet container:
    a. Loads the servlet class
    b. Creates an instance of the servlet class
    c. Initializes the servlet instance by calling the `init` method
2. The servlet container invokes the `service` method, passing request and response objects.

Initialization is covered in Creating and Initializing a Servlet. Service methods are discussed in Writing Service Methods.

If it needs to remove the servlet, the servlet container finalizes the servlet by calling the servlet's `destroy` method. For more information, see Finalizing a Servlet.

Sharing Information

Web components, like most objects, usually work with data to accomplish their tasks. Web components can store this information in a data store or in a scoped bean, among others.

Using CDI Managed Beans

CDI can be used to define scoped beans which can be injected in any container-managed objects such as web components. These scoped beans can then be used to store and transfer information.

CDI defines three basic scopes which can be used for this:

`@jakarta.enterprise.context.RequestScoped`
: In a servlet container, this is mapped to the `jakarta.servlet.ServletRequest`. It lives from the time that the client has sent the request until it has retrieved the corresponding response. It is not shared elsewhere.

`@jakarta.enterprise.context.SessionScoped`
: In a servlet container, this is mapped to the `jakarta.servlet.http.HttpSession`. It lives for as long as the client is interacting with the web application with the same browser instance, and the session hasn't timed out at the server side. It is shared among all requests in the same session.

`@jakarta.enterprise.context.ApplicationScoped`
: In a servlet container, this is mapped to the `jakarta.servlet.ServletContext`. It lives for as long as the web application lives. It is shared among all requests in all sessions.

The below example illustrates how CDI `@SessionScoped` can be used to define a session scoped bean which stores the user preferences.

```
@Named
@SessionScoped
public class UserPreferences implements Serializable {

    private Locale language;
    private ZoneId timeZone;
    private boolean darkMode;

    // ...
}
```

It can be injected in any web component. The below example illustrates how a servlet can be used to take a time zone offset in minutes and update the user preferences with it.

```
@WebServlet("/timeZoneHandler")
public class TimeZoneHandler extends HttpServlet {
```

```
    @Inject
    private UserPreferences userPreferences;

    @Override
    public void doPost
        (HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        var offsetInMinutes = request.getParameter("offsetInMinutes");

        if (offsetInMinutes == null || !offsetInMinutes.matches("\\-?[0-9]{1,3}")) {
            response.sendError(HttpServletResponse.SC_BAD_REQUEST);
            return;
        }

        var offsetInSeconds =
TimeUnit.MINUTES.toSeconds(Long.valueOf(offsetInMinutes));
        ZoneId timeZone = ZoneOffset.ofTotalSeconds((int) offsetInSeconds);
        userPreferences.setTimeZone(timeZone);
        response.setStatus(HttpServletResponse.SC_NO_CONTENT);
    }
}
```

For example JavaScript's `new Date().getTimeZoneOffset()` returns the local (negative) time zone offset in minutes. This servlet can then be invoked as follows in JavaScript in order to inform the server about the client's time zone:

```
fetch("/context-path/timeZoneHandler", {
    method: "POST",
    body: new URLSearchParams({
        offsetInMinutes: -new Date().getTimezoneOffset()
    })
});
```

Using Scope Objects Directly

If CDI is not available, an alternative is to use Jakarta Servlet's own scope objects directly. You can use `getAttribute` and `setAttribute` methods of the Jakarta Servlet class representing the scope. Scope Objects lists the scope objects.

Scope Objects

Scope Object	Class	Accessible From
Application	`jakarta.servlet.ServletContext`	Web components within the web application. See Accessing the Web Context.
Session	`jakarta.servlet.http.HttpSession`	Web components handling a request that belongs to the session. See Maintaining Client State.

Request	jakarta.servlet.ServletRequest	Web components handling the request.

The below example illustrates how the previously shown servlet needs to be adjusted to manually manage the `UserPreferences` bean.

```
@WebServlet("/timeZoneHandler")
public class TimeZoneHandler extends HttpServlet {

    @Override
    public void doPost
        (HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        var offsetInMinutes = request.getParameter("offsetInMinutes");

        if (offsetInMinutes == null || !offsetInMinutes.matches("\\-?[0-9]{1,3}")) {
            response.sendError(HttpServletResponse.SC_BAD_REQUEST);
            return;
        }

        var offsetInSeconds =
TimeUnit.MINUTES.toSeconds(Long.valueOf(offsetInMinutes));
        ZoneId timeZone = ZoneOffset.ofTotalSeconds((int) offsetInSeconds);
        HttpSession session = request.getSession();
        UserPreferenes userPreferences = session.getAttribute("userPreferences");

        if (userPreferences == null) {
            userPreferences = new UserPreferences();
            session.setAttribute("userPreferences", userPreferences);
        }

        userPreferences.setTimeZone(timeZone);
        response.setStatus(HttpServletResponse.SC_NO_CONTENT);
    }
}
```

Controlling Concurrent Access to Shared Resources

In a multithreaded server, shared resources can be accessed concurrently. In addition to scope object attributes, shared resources include in-memory data, such as instance or class variables, and external objects, such as files, database connections, and network connections.

Concurrent access can arise in several situations.

- Multiple web components accessing objects stored in the application scope.
- Multiple web components accessing objects stored in the session scope.
- Multiple threads within a web component accessing instance variables.

A web container will typically create a thread to handle each request. When resources can be accessed concurrently, they can be used in an inconsistent fashion. First step is to ensure that the variable representing the resource has the correct scope and use a as narrow as possible scope. For example, request scoped information should not be stored in a session scoped bean nor be assigned as an instance variable of a servlet, and session scoped information should not be stored in an application scoped bean.

If concurrent access is inevitable, then you prevent this by using synchronized or atomic objects such as wrapping a `Map` in `Collections.synchronizedMap()` before assigning it to a property of a session scoped bean.

Creating and Initializing a Servlet



This section was last updated for Jakarta EE Version 9.1

Use the `@WebServlet` annotation to define a servlet component in a web application. This annotation is specified on a class and contains metadata about the servlet being declared. The annotated servlet must specify at least one URL pattern. This is done by using the `urlPatterns` or `value` attribute on the annotation. All other attributes are optional, with default settings. Use the `value` attribute when the only attribute on the annotation is the URL pattern; otherwise, use the `urlPatterns` attribute when other attributes are also used.

Classes annotated with `@WebServlet` must extend the `jakarta.servlet.http.HttpServlet` class. For example, the following code snippet defines a servlet with the URL pattern `/report`:

```
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;

@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
}
```

The web container initializes a servlet after loading and instantiating the servlet class and before delivering requests from clients. To customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities, you can either override the `init` method of the `Servlet` interface or specify the `initParams` attribute of the `@WebServlet` annotation. The `initParams` attribute contains a `@WebInitParam` annotation. If it cannot complete its initialization process, a servlet throws an `UnavailableException`.

Use an initialization parameter to provide data needed by a particular servlet. By contrast, a context parameter provides data that is available to all components of a web application.

Writing Service Methods

The service provided by a servlet is implemented in the `service` method of a `GenericServlet`, in the `doMethod` methods (where *Method* can take the value `Get`, `Delete`, `Options`, `Post`, `Put`, or `Trace`) of an `HttpServlet` object, or in any other protocol-specific methods defined by a class that implements the

`Servlet` interface. The term service method is used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information. For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.
2. Fill in the response headers.
3. Write any body content to the output stream.

Response headers must always be set before the response has been committed. The web container will ignore any attempt to set or add headers after the response has been committed. The next two sections describe how to get information from requests and generate responses.

Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the `ServletRequest` interface. This interface defines methods for accessing the following information:

- Parameters, which are typically used to convey information between clients and servlets
- Object-valued attributes, which are typically used to pass information between the web container and a servlet or between collaborating servlets
- Information about the protocol used to communicate the request and about the client and server involved in the request
- Information relevant to localization

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the `BufferedReader` object returned by the request's `getReader` method. To read binary data, use the `ServletInputStream` returned by `getInputStream`.

HTTP servlets are passed an HTTP request object, `HttpServletRequest`, which contains the request URL, HTTP headers, query string, and so on. An HTTP request URL contains the following parts:

```
http://[host]:[port][request-path]?[query-string]
```

The request path is further composed of the following elements.

- Context path: A concatenation of a forward slash (`/`) with the context root of the servlet's web application.
- Servlet path: The path section that corresponds to the component alias that activated this request. This path starts with a forward slash (`/`).
- Path info: The part of the request path that is not part of the context path or the servlet path.

You can use the `getContextPath`, `getServletPath`, and `getPathInfo` methods of the `HttpServletRequest` interface to access this information. Except for URL encoding differences between the request URI

and the path parts, the request URI is always comprised of the context path plus the servlet path plus the path info.

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request by using the `getParameter` method. There are two ways to generate query strings.

- A query string can explicitly appear in a web page.
- A query string is appended to a URL when a form with a `GET` HTTP method is submitted.

Constructing Responses

A response contains data passed between a server and the client. All responses implement the `ServletResponse` interface. This interface defines methods that allow you to do the following.

- Retrieve an output stream to use to send data to the client. To send character data, use the `PrintWriter` returned by the response's `getWriter` method. To send binary data in a Multipurpose Internet Mail Extensions (MIME) body response, use the `ServletOutputStream` returned by `getOutputStream`. To mix binary and text data, as in a multipart response, use a `ServletOutputStream` and manage the character sections manually.
- Indicate the content type (for example, `text/html`) being returned by the response with the `setContentType(String)` method. This method must be called before the response is committed. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at https://www.iana.org/assignments/media-types/.
- Indicate whether to buffer output with the `setBufferSize(int)` method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.
- Set localization information, such as locale and character encoding. See [web:webi18n::webi18n:::_internationalizing_and_localizing_web_applications] for details.

HTTP response objects, `jakarta.servlet.http.HttpServletResponse`, have fields representing HTTP headers, such as the following.

- Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.
- Cookies, which are used to store application-specific information at the client. Sometimes, cookies are used to maintain an identifier for tracking a user's session (see Session Tracking).

Handling Servlet Lifecycle Events

You can monitor and react to events in a servlet's lifecycle by defining listener objects whose methods get invoked when lifecycle events occur. To use these listener objects, you must define and specify the listener class.

Defining the Listener Class

You define a listener class as an implementation of a listener interface. Servlet Lifecycle Events lists

the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

Servlet Lifecycle Events

Object	Event	Listener Interface and Event Class
Web context	Initialization and destruction	`jakarta.servlet.ServletContextListener` and `ServletContextEvent`
Web context	Attribute added, removed, or replaced	`jakarta.servlet.ServletContextAttributeListener` and `ServletContextAttributeEvent`
Session	Creation, invalidation, activation, passivation, and timeout	`jakarta.servlet.http.HttpSessionListener`, `jakarta.servlet.http.HttpSessionActivationListener`, and `HttpSessionEvent`
Session	Attribute added, removed, or replaced	`jakarta.servlet.http.HttpSessionAttributeListener` and `HttpSessionBindingEvent`
Request	A servlet request has started being processed by web components	`jakarta.servlet.ServletRequestListener` and `ServletRequestEvent`
Request	Attribute added, removed, or replaced	`jakarta.servlet.ServletRequestAttributeListener` and `ServletRequestAttributeEvent`

Use the `@WebListener` annotation to define a listener to get events for various operations on the particular web application context. Classes annotated with `@WebListener` must implement one of the following interfaces:

```
jakarta.servlet.ServletContextListener
jakarta.servlet.ServletContextAttributeListener
jakarta.servlet.ServletRequestListener
jakarta.servlet.ServletRequestAttributeListener
jakarta.servlet..http.HttpSessionListener
jakarta.servlet..http.HttpSessionAttributeListener
```

For example, the following code snippet defines a listener that implements two of these interfaces:

```
import jakarta.servlet.ServletContextAttributeListener;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;

@WebListener()
public class SimpleServletListener implements ServletContextListener,
        ServletContextAttributeListener {
    ...
}
```

Handling Servlet Errors

Any number of exceptions can occur when a servlet executes. When an exception occurs, the web container generates a default page containing the following message:

```
A Servlet Exception Has Occurred
```

But you can also specify that the container should return a specific error page for a given exception.

Filtering Requests and Responses

A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be "attached" to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows.

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

Programming Filters

The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `jakarta.servlet` package. You define a filter by implementing the `Filter` interface.

Use the `@WebFilter` annotation to define a filter in a web application. This annotation is specified on a class and contains metadata about the filter being declared. The annotated filter must specify at least one URL pattern. This is done by using the `urlPatterns` or `value` attribute on the annotation. All other attributes are optional, with default settings. Use the `value` attribute when the only attribute on the annotation is the URL pattern; use the `urlPatterns` attribute when other attributes are also used.

Classes annotated with the `@WebFilter` annotation must implement the `jakarta.servlet.Filter`

interface.

To add configuration data to the filter, specify the `initParams` attribute of the `@WebFilter` annotation. The `initParams` attribute contains a `@WebInitParam` annotation. The following code snippet defines a filter, specifying an initialization parameter:

```
import jakarta.servlet.Filter;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter", urlPatterns = {"/*"},
initParams = {@WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
}
```

The most important method in the `Filter` interface is `doFilter`, which is passed request, response, and filter chain objects. This method can perform the following actions.

- Examine the request headers.
- Customize the request object if the filter wishes to modify request headers or data.
- Customize the response object if the filter wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the `doFilter` method on the chain object, passing in the request and response it was called with or the wrapped versions it may have created. Alternatively, the filter can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
- Examine response headers after invoking the next filter in the chain.
- Throw an exception to indicate an error in processing.

In addition to `doFilter`, you must implement the `init` and `destroy` methods. The `init` method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the `FilterConfig` object passed to `init`.

Programming Customized Requests and Responses

There are many ways for a filter to modify a request or a response. For example, a filter can add an attribute to the request or can insert data in the response.

A filter that modifies a response must usually capture the response before it is returned to the client. To do this, you pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the

`getWriter` or `getOutputStream` method to return this stand-in stream. The wrapper is passed to the `doFilter` method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object.

To override request methods, you wrap the request in an object that extends either `ServletRequestWrapper` or `HttpServletRequestWrapper`. To override response methods, you wrap the response in an object that extends either `ServletResponseWrapper` or `HttpServletResponseWrapper`.

Specifying Filter Mappings

A web container uses filter mappings to decide how to apply filters to web resources. A filter mapping matches a filter to a web component by name or to web resources by URL pattern. The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR. You specify a filter mapping list for a WAR in its deployment descriptor by either using NetBeans IDE or coding the list by hand with XML.

If you want to log every request to a web application, you map the hit counter filter to the URL pattern `/*`.

You can map a filter to one or more web resources, and you can map more than one filter to a web resource. This is illustrated in Figure 9, "Filter-to-Servlet Mapping", in which filter F1 is mapped to servlets S1, S2, and S3; filter F2 is mapped to servlet S2; and filter F3 is mapped to servlets S1 and S2.

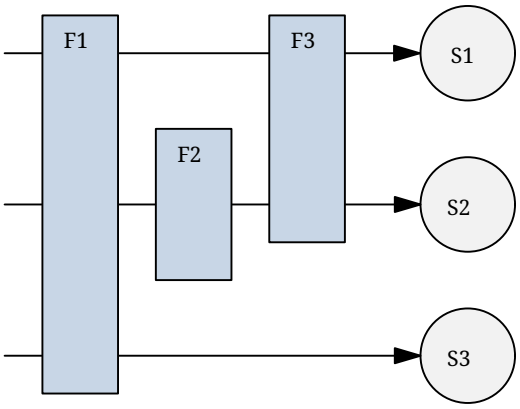


Figure 9. Filter-to-Servlet Mapping

Recall that a filter chain is one of the objects passed to the `doFilter` method of a filter. This chain is formed indirectly by means of filter mappings. The order of the filters in the chain is the same as the order in which filter mappings appear in the web application deployment descriptor.

When a filter is mapped to servlet S1, the web container invokes the `doFilter` method of F1. The `doFilter` method of each filter in S1's filter chain is invoked by the preceding filter in the chain by means of the `chain.doFilter` method. Because S1's filter chain contains filters F1 and F3, F1's call to `chain.doFilter` invokes the `doFilter` method of filter F3. When F3's `doFilter` method completes, control returns to F1's `doFilter` method.

To Specify Filter Mappings Using NetBeans IDE

1. Expand the application's project node in the **Project** tab.
2. Expand the **Web Pages** and **WEB-INF** nodes under the project node.

3. Double-click `web.xml`.
4. Click **Filters** at the top of the editor window.
5. Expand the **Servlet Filters** node in the editor window.
6. Click **Add Filter Element** to map the filter to a web resource by name or by URL pattern.
7. In the Add Servlet Filter dialog box, enter the name of the filter in the **Filter Name** field.
8. Click **Browse** to locate the servlet class to which the filter applies.

   You can include wildcard characters so that you can apply the filter to more than one servlet.
9. Click **OK**.
10. To constrain how the filter is applied to requests, follow these steps:

    a. Expand the **Filter Mappings** node.

    b. Select the filter from the list of filters.

    c. Click **Add**.

    d. In the Add Filter Mapping dialog box, select one of the following dispatcher types:

REQUEST	Only when the request comes directly from the client
ASYNC	Only when the asynchronous request comes from the client
FORWARD	Only when the request has been forwarded to a component (see Transferring Control to Another Web Component)
INCLUDE	Only when the request is being processed by a component that has been included (see Including Other Resources in the Response)
ERROR	Only when the request is being processed with the error page mechanism (see Handling Servlet Errors)

    You can direct the filter to be applied to any combination of the preceding situations by selecting multiple dispatcher types. If no types are specified, the default option is **REQUEST**.

Invoking Other Web Resources

Web components can invoke other web resources both indirectly and directly. A web component indirectly invokes another web resource by embedding a URL that points to another web component in content returned to a client. While it is executing, a web component directly invokes another resource by either including the content of another resource or forwarding a request to another resource.

To invoke a resource available on the server that is running a web component, you must first obtain a `RequestDispatcher` object by using the `getRequestDispatcher("URL")` method. You can get a `RequestDispatcher` object from either a request or the web context; however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a `/`), but the web context requires an absolute path. If the resource is not available or if the server has not implemented a

`RequestDispatcher` object for that type of resource, `getRequestDispatcher` will return null. Your servlet should be prepared to deal with this condition.

Including Other Resources in the Response

It is often useful to include another web resource, such as banner content or copyright information, in the response returned from a web component. To include another resource, invoke the `include` method of a `RequestDispatcher` object:

```
include(request, response);
```

If the resource is static, the `include` method enables programmatic server-side includes. If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet. An included web component has access to the request object but is limited in what it can do with the response object.

- It can write to the body of the response and commit a response.
- It cannot set headers or call any method, such as `setCookie`, that affects the headers of the response.

Transferring Control to Another Web Component

In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component, depending on the nature of the request.

To transfer control to another web component, you invoke the `forward` method of a `RequestDispatcher`. When a request is forwarded, the request URL is set to the path of the forwarded page. The original URI and its constituent parts are saved as the following request attributes:

```
jakarta.servlet.forward.request_uri
jakarta.servlet.forward.context_path
jakarta.servlet.forward.servlet_path
jakarta.servlet.forward.path_info
jakarta.servlet.forward.query_string
```

The `forward` method should be used to give another resource responsibility for replying to the user. If you have already accessed a `ServletOutputStream` or `PrintWriter` object within the servlet, you cannot use this method; doing so throws an `IllegalStateException`.

Accessing the Web Context

The context in which web components execute is an object that implements the `ServletContext` interface. You retrieve the web context by using the `getServletContext` method. The web context provides methods for accessing

- Initialization parameters
- Resources associated with the web context
- Object-valued attributes
- Logging capabilities

The counter's access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object by using the context's `getAttribute` method. The incremented value of the counter is recorded in the log.

Maintaining Client State

Many applications require that a series of requests from a client be associated with one another. For example, a web application can save the state of a user's shopping cart across requests. Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless. To support applications that need to maintain state, Jakarta Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

Accessing a Session

Sessions are represented by an `HttpSession` object. You access a session by calling the `getSession` method of a request object. This method returns the current session associated with this request; or, if the request does not have a session, this method creates one.

Associating Objects with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any web component that belongs to the same web context and is handling a request that is part of the same session.

Recall that your application can notify web context and session listener objects of servlet lifecycle events (Handling Servlet Lifecycle Events). You can also notify objects of certain events related to their association with a session, such as the following.

- When the object is added to or removed from a session. To receive this notification, your object must implement the `jakarta.servlet.http.HttpSessionBindingListener` interface.
- When the session to which the object is attached will be passivated or activated. A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage. To receive this notification, your object must implement the `jakarta.servlet.http.HttpSessionActivationListener` interface.

Session Management

Because an HTTP client has no way to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed by using a session's `getMaxInactiveInterval` and `setMaxInactiveInterval` methods.

- To ensure that an active session is not timed out, you should periodically access the session by using service methods because this resets the session's time-to-live counter.

- When a particular client interaction is finished, you use the session's `invalidate` method to invalidate a session on the server side and remove any session data.

To Set the Timeout Period Using NetBeans IDE

To set the timeout period in the deployment descriptor using NetBeans IDE, follow these steps.

1. Open the project if you haven't already.
2. Expand the node of your project in the **Projects** tab.
3. Expand the **Web Pages** and **WEB-INF** nodes that are under the project node.
4. Double-click `web.xml`.
5. Click **General** at the top of the editor.
6. In the **Session Timeout** field, enter an integer value.

   The integer value represents the number of minutes of inactivity that must pass before the session times out.

Session Tracking

To associate a session with a user, a web container can use several methods, all of which involve passing an identifier between the client and the server. The identifier can be maintained on the client as a cookie, or the web component can include the identifier in every URL that is returned to the client.

If your application uses session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies. You do this by calling the response's `encodeURL(URL)` method on all URLs returned by a servlet. This method includes the session ID in the URL only if cookies are disabled; otherwise, the method returns the URL unchanged.

Servlet Request Connection Information

The session tracking as described above is sufficient for most applications but debugging and certain advanced use cases may require tracking individual low level connections.

New in the Servlet 6.0 API release is the `jakarta.servlet.ServletConnection` interface which gives fine-grained information about the connection associated with a given servlet request. Most notably `getConnectionId()` returns a unique ID for the network connection and `getRequestId()` returns a unique ID for the given servlet request.

Here is an example of retrieving the connection ID inside a servlet method:

```
@Override
public void doGet
    (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
{
    ServletConnection connection = request.getServletConnection();
```

```
    String id = connection.getConnectionId();
    ...
}
```

Finalizing a Servlet

The web container may determine that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shut down). In such a case, the container calls the `destroy` method of the `Servlet` interface. In this method, you release any resources the servlet is using and save any persistent state. The `destroy` method releases the database object created in the `init` method.

A servlet's service methods should all be complete when a servlet is removed. The server tries to ensure this by calling the `destroy` method only after all service requests have returned or after a server-specific grace period, whichever comes first. If your servlet has operations that may run longer than the server's grace period, the operations could still be running when `destroy` is called. You must make sure that any threads still handling client requests complete.

The remainder of this section explains how to do the following.

- Keep track of how many threads are currently running the `service` method.
- Provide a clean shutdown by having the `destroy` method notify long-running threads of the shutdown and wait for them to complete.
- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return.

Tracking Service Requests

To track service requests:

1. Include a field in your servlet class that counts the number of service methods that are running.

   The field should have synchronized access methods to increment, decrement, and return its value:

   ```
   public class ShutdownExample extends HttpServlet {
       private int serviceCounter = 0;
       ...
       // Access methods for serviceCounter
       protected synchronized void enteringServiceMethod() {
           serviceCounter++;
       }
       protected synchronized void leavingServiceMethod() {
           serviceCounter--;
       }
       protected synchronized int numServices() {
           return serviceCounter;
       }
   ```

```
}
```

The `service` method should increment the service counter each time the method is entered and should decrement the counter each time the method returns. This is one of the few times that your `HttpServlet` subclass should override the `service` method. The new method should call `super.service` to preserve the functionality of the original `service` method:

```
protected void service(HttpServletRequest req,
                       HttpServletResponse resp)
                       throws ServletException,IOException {
    enteringServiceMethod();
    try {
        super.service(req, resp);
    } finally {
        leavingServiceMethod();
    }
}
```

Notifying Methods to Shut Down

To ensure a clean shutdown, your `destroy` method should not release any shared resources until all the service requests have completed:

1. Check the service counter.
2. Notify long-running methods that it is time to shut down.

   For this notification, another field is required. The field should have the usual access methods:

```
public class ShutdownExample extends HttpServlet {
    private boolean shuttingDown;
    ...
    //Access methods for shuttingDown
    protected synchronized void setShuttingDown(boolean flag) {
        shuttingDown = flag;
    }
    protected synchronized boolean isShuttingDown() {
        return shuttingDown;
    }
}
```

   Here is an example of the `destroy` method using these fields to provide a clean shutdown:

```
public void destroy() {
    /* Check to see whether there are still service methods /*
    /* running, and if there are, tell them to stop. */
    if (numServices() > 0) {
        setShuttingDown(true);
```

```
    }

    /* Wait for the service methods to stop. */
    while (numServices() > 0) {
        try {
            Thread.sleep(interval);
        } catch (InterruptedException e) {
        }
    }
}
```

Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary:

```
public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) &&
         !isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        } catch (InterruptedException e) {
            ...
        }
    }
}
```

Uploading Files with Jakarta Servlet Technology

Supporting file uploads is a very basic and common requirement for many web applications. In prior versions of the Servlet specification, implementing file upload required the use of external libraries or complex input processing. The Jakarta Servlet specification now helps to provide a viable solution to the problem in a generic and portable way. Jakarta Servlet technology now supports file upload out of the box, so any web container that implements the specification can parse multipart requests and make mime attachments available through the `HttpServletRequest` object.

A new annotation, `jakarta.servlet.annotation.MultipartConfig`, is used to indicate that the servlet on which it is declared expects requests to be made using the `multipart/form-data` MIME type. Servlets that are annotated with `@MultipartConfig` can retrieve the `Part` components of a given `multipart/form-data` request by calling the `request.getPart(String name)` or `request.getParts()` method.

The @MultipartConfig Annotation

The `@MultipartConfig` annotation supports the following optional attributes.

- `location`: An absolute path to a directory on the file system. The `location` attribute does not support a path relative to the application context. This location is used to store files temporarily while the parts are processed or when the size of the file exceeds the specified `fileSizeThreshold` setting. The default location is `""`.
- `fileSizeThreshold`: The file size in bytes after which the file will be temporarily stored on disk. The default size is 0 bytes.
- `maxFileSize`: The maximum size allowed for uploaded files, in bytes. If the size of any uploaded file is greater than this size, the web container will throw an exception (`IllegalStateException`). The default size is unlimited.
- `maxRequestSize`: The maximum size allowed for a `multipart/form-data` request, in bytes. The web container will throw an exception if the overall size of all uploaded files exceeds this threshold. The default size is unlimited.

For, example, the `@MultipartConfig` annotation could be constructed as follows:

```
@MultipartConfig(location="/tmp", fileSizeThreshold=1024*1024,
    maxFileSize=1024*1024*5, maxRequestSize=1024*1024*5*5)
```

Instead of using the `@MultipartConfig` annotation to hard-code these attributes in your file upload servlet, you could add the following as a child element of the servlet configuration element in the `web.xml` file:

```
<multipart-config>
    <location>/tmp</location>
    <max-file-size>20848820</max-file-size>
    <max-request-size>418018841</max-request-size>
    <file-size-threshold>1048576</file-size-threshold>
</multipart-config>
```

The getParts and getPart Methods

The Servlet specification supports two additional `HttpServletRequest` methods:

- `Collection<Part> getParts()`
- `Part getPart(String name)`

The `request.getParts()` method returns collections of all `Part` objects. If you have more than one input of type file, multiple `Part` objects are returned. Because `Part` objects are named, the `getPart(String name)` method can be used to access a particular `Part`. Alternatively, the `getParts()` method, which returns an `Iterable<Part>`, can be used to get an `Iterator` over all the `Part` objects.

The `jakarta.servlet.http.Part` interface is a simple one, providing methods that allow introspection of each `Part`. The methods do the following:

- Retrieve the name, size, and content-type of the `Part`
- Query the headers submitted with a `Part`

- Delete a `Part`
- Write a `Part` out to disk

For example, the `Part` interface provides the `write(String filename)` method to write the file with the specified name. The file can then be saved in the directory that is specified with the `location` attribute of the `@MultipartConfig` annotation or, in the case of the `fileupload` example, in the location specified by the Destination field in the form.

Asynchronous Processing

Web containers in application servers normally use a server thread per client request. Under heavy load conditions, containers need a large amount of threads to serve all the client requests. Scalability limitations include running out of memory or exhausting the pool of container threads. To create scalable web applications, you must ensure that no threads associated with a request are sitting idle, so the container can use them to process new requests.

There are two common scenarios in which a thread associated with a request can be sitting idle.

- The thread needs to wait for a resource to become available or process data before building the response. For example, an application may need to query a database or access data from a remote web service before generating the response.
- The thread needs to wait for an event before generating the response. For example, an application may have to wait for a Jakarta Messaging message, new information from another client, or new data available in a queue before generating the response.

These scenarios represent blocking operations that limit the scalability of web applications. Asynchronous processing refers to assigning these blocking operations to a new thread and retuning the thread associated with the request immediately to the container.

Asynchronous Processing in Servlets

Jakarta EE provides asynchronous processing support for servlets and filters. If a servlet or a filter reaches a potentially blocking operation when processing a request, it can assign the operation to an asynchronous execution context and return the thread associated with the request immediately to the container without generating a response. The blocking operation completes in the asynchronous execution context in a different thread, which can generate a response or dispatch the request to another servlet.

To enable asynchronous processing on a servlet, set the parameter `asyncSupported` to `true` on the `@WebServlet` annotation as follows:

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet { ... }
```

The `jakarta.servlet.AsyncContext` class provides the functionality that you need to perform asynchronous processing inside service methods. To obtain an instance of `AsyncContext`, call the `startAsync()` method on the request object of your service method; for example:

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
   ...
   AsyncContext acontext = req.startAsync();
   ...
}
```

This call puts the request into asynchronous mode and ensures that the response is not committed after exiting the service method. You have to generate the response in the asynchronous context after the blocking operation completes or dispatch the request to another servlet.

Functionality Provided by the AsyncContext Class describes the basic functionality provided by the `AsyncContext` class.

Functionality Provided by the AsyncContext Class

Method Signature	Description
`void start(Runnable run)`	The container provides a different thread in which the blocking operation can be processed. You provide code for the blocking operation as a class that implements the `Runnable` interface. You can provide this class as an inner class when calling the `start` method or use another mechanism to pass the `AsyncContext` instance to your class.
`ServletRequest getRequest()`	Returns the request used to initialize this asynchronous context. In the example above, the request is the same as in the service method. You can use this method inside the asynchronous context to obtain parameters from the request.
`ServletResponse getResponse()`	Returns the response used to initialize this asynchronous context. In the example above, the response is the same as in the service method. You can use this method inside the asynchronous context to write to the response with the results of the blocking operation.
`void complete()`	Completes the asynchronous operation and closes the response associated with this asynchronous context. You call this method after writing to the response object inside the asynchronous context.
`void dispatch(String path)`	Dispatches the request and response objects to the given path. You use this method to have another servlet write to the response after the blocking operation completes.

Waiting for a Resource

This section demonstrates how to use the functionality provided by the `AsyncContext` class for the following use case:

1. A servlet receives a parameter from a GET request.
2. The servlet uses a resource, such as a database or a web service, to retrieve information based on the value of the parameter. The resource can be slow at times, so this may be a blocking operation.
3. The servlet generates a response using the result from the resource.

The following code shows a basic servlet that does not use asynchronous processing:

```
@WebServlet(urlPatterns={"/syncservlet"})
public class SyncServlet extends HttpServlet {
   private MyRemoteResource resource;
   @Override
   public void init(ServletConfig config) {
      resource = MyRemoteResource.create("config1=x,config2=y");
   }

   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html;charset=UTF-8");
      String param = request.getParameter("param");
      String result = resource.process(param);
      /* ... print to the response ... */
   }
}
```

The following code shows the same servlet using asynchronous processing:

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet {
   /* ... Same variables and init method as in SyncServlet ... */
   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      response.setContentType("text/html;charset=UTF-8");
      final AsyncContext acontext = request.startAsync();
      acontext.start(new Runnable() {
         public void run() {
            String param = acontext.getRequest().getParameter("param");
            String result = resource.process(param);
            HttpServletResponse response = acontext.getResponse();
            /* ... print to the response ... */
            acontext.complete();
         }
```

```
        });
    }
}
```

`AsyncServlet` adds `asyncSupported=true` to the `@WebServlet` annotation. The rest of the differences are inside the service method.

- `request.startAsync()` causes the request to be processed asynchronously; the response is not sent to the client at the end of the service method.
- `acontext.start(new Runnable() {⋯})` gets a new thread from the container.
- The code inside the `run()` method of the inner class executes in the new thread. The inner class has access to the asynchronous context to read parameters from the request and write to the response. Calling the `complete()` method of the asynchronous context commits the response and sends it to the client.

The service method of `AsyncServlet` returns immediately, and the request is processed in the asynchronous context.

Nonblocking I/O

Web containers in application servers normally use a server thread per client request. To develop scalable web applications, you must ensure that threads associated with client requests are never sitting idle waiting for a blocking operation to complete. Asynchronous Processing provides a mechanism to execute application-specific blocking operations in a new thread, returning the thread associated with the request immediately to the container. Even if you use asynchronous processing for all the application-specific blocking operations inside your service methods, threads associated with client requests can be momentarily sitting idle because of input/output considerations.

For example, if a client is submitting a large HTTP POST request over a slow network connection, the server can read the request faster than the client can provide it. Using traditional I/O, the container thread associated with this request would be sometimes sitting idle waiting for the rest of the request.

Jakarta EE provides nonblocking I/O support for servlets and filters when processing requests in asynchronous mode. The following steps summarize how to use nonblocking I/O to process requests and write responses inside service methods.

1. Put the request in asynchronous mode as described in Asynchronous Processing.
2. Obtain an input stream and/or an output stream from the request and response objects in the service method.
3. Assign a read listener to the input stream and/or a write listener to the output stream.
4. Process the request and the response inside the listener's callback methods.

Nonblocking I/O Support in jakarta.servlet.ServletOutputStream describe the methods available in the servlet input and output streams for nonblocking I/O support. Listener Interfaces for Nonblocking I/O Support describes the interfaces for read listeners and write listeners.

Nonblocking I/O Support in jakarta.servlet.ServletInputStream

Method	Description
`void setReadListener(ReadListener rl)`	Associates this input stream with a listener object that contains callback methods to read data asynchronously. You provide the listener object as an anonymous class or use another mechanism to pass the input stream to the read listener object.
`boolean isReady()`	Returns true if data can be read without blocking.
`boolean isFinished()`	Returns true when all the data has been read.

Nonblocking I/O Support in jakarta.servlet.ServletOutputStream

Method	Description
`void setWriteListener(WriteListener wl)`	Associates this output stream with a listener object that contains callback methods to write data asynchronously. You provide the write listener object as an anonymous class or use another mechanism to pass the output stream to the write listener object.
`boolean isReady()`	Returns true if data can be written without blocking.

Listener Interfaces for Nonblocking I/O Support

Interface	Methods	Description

`ReadListener`	`void onDataAvailable()`
	`void onAllDataRead()`
	`void onError(Throwable t)`
		A `ServletInputStream` instance calls these methods on its listener when there is data available to read, when all the data has been read, or when there is an error.

`WriteListener`	`void onWritePossible()` `void onError(Throwable t)`	A `ServletOutputStream` instance calls these methods on its listener when it is possible to write data without blocking or when there is an error.

Reading a Large HTTP POST Request Using Nonblocking I/O

The code in this section shows how to read a large HTTP POST request inside a servlet by putting the request in asynchronous mode (as described in Asynchronous Processing) and using the nonblocking I/O functionality from Nonblocking I/O Support in jakarta.servlet.ServletInputStream and Listener Interfaces for Nonblocking I/O Support.

```
@WebServlet(urlPatterns={"/asyncioservlet"}, asyncSupported=true)
public class AsyncIOServlet extends HttpServlet {
```

```
    @Override
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws IOException {
       final AsyncContext acontext = request.startAsync();
       final ServletInputStream input = request.getInputStream();

       input.setReadListener(new ReadListener() {
          byte buffer[] = new byte[4*1024];
          StringBuilder sbuilder = new StringBuilder();
          @Override
          public void onDataAvailable() {
             try {
                do {
                   int length = input.read(buffer);
                   sbuilder.append(new String(buffer, 0, length));
                } while(input.isReady());
             } catch (IOException ex) { ... }
          }
          @Override
          public void onAllDataRead() {
             try {
                acontext.getResponse().getWriter()
                                      .write("...the response...");
             } catch (IOException ex) { ... }
             acontext.complete();
          }
          @Override
          public void onError(Throwable t) { ... }
       });
    }
}
```

This example declares the web servlet with asynchronous support using the `@WebServlet` annotation parameter `asyncSupported=true`. The service method first puts the request in asynchronous mode by calling the `startAsync()` method of the request object, which is required in order to use nonblocking I/O. Then, the service method obtains an input stream associated with the request and assigns a read listener defined as an inner class. The listener reads parts of the request as they become available and then writes some response to the client when it finishes reading the request.

Server Push

Server push is the ability of the server to anticipate what will be needed by the client in advance of the client's request. It lets the server pre-populate the browser's cache in advance of the browser asking for the resource to put in the cache.

Server push is the most visible of the improvements in HTTP/2 to appear in the servlet API. All of the new features in HTTP/2, including server push, are aimed at improving the performance of the web browsing experience.

Server push derives its contribution to improved browser performance from the fact that servers know what additional assets (such as images, stylesheets, and scripts) go along with initial requests. For example, servers might know that whenever a browser requests `index.html`, it will shortly thereafter request `header.gif`, `footer.gif`, and `style.css`. Servers can preemptively start sending the bytes of these assets along with the bytes of the `index.html`.

To use server push, obtain a reference to a `PushBuilder` from an `HttpServletRequest`, edit the builder as desired, then call `push()`. See the javadoc for the class `jakarta.servlet.http.PushBuilder` and the method `jakarta.servlet.http.HttpServletRequest.newPushBuilder()`.

Protocol Upgrade Processing

In HTTP/1.1, clients can request to switch to a different protocol on the current connection by using the `Upgrade` header field. If the server accepts the request to switch to the protocol indicated by the client, it generates an HTTP response with status 101 (switching protocols). After this exchange, the client and the server communicate using the new protocol.

For example, a client can make an HTTP request to switch to the XYZP protocol as follows:

```
GET /xyzpresource HTTP/1.1
Host: localhost:8080
Accept: text/html
Upgrade: XYZP
Connection: Upgrade
OtherHeaderA: Value
```

The client can specify parameters for the new protocol using HTTP headers. The server can accept the request and generate a response as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: XYZP
Connection: Upgrade
OtherHeaderB: Value

(XYZP data)
```

Jakarta EE supports the HTTP protocol upgrade functionality in servlets, as described in Protocol Upgrade Support.

Protocol Upgrade Support

Class or Interface	Method
`HttpServletRequest`	`HttpUpgradeHandler upgrade(Class handler)` The upgrade method starts the protocol upgrade processing. This method instantiates a class that implements the `HttpUpgradeHandler` interface and delegates the connection to it. You call the `upgrade` method inside a service method when accepting a request from a client to switch protocols.
`HttpUpgradeHandler`	`void init(WebConnection wc)` The `init` method is called when the servlet accepts the request to switch protocols. You implement this method and obtain input and output streams from the `WebConnection` object to implement the new protocol.
`HttpUpgradeHandler`	`void destroy()` The `destroy` method is called when the client disconnects. You implement this method and free any resources associated with processing the new protocol.
`WebConnection`	`ServletInputStream getInputStream()` The `getInputStream` method provides access to the input stream of the connection. You can use Nonblocking I/O with the returned stream to implement the new protocol.
`WebConnection`	`ServletOutputStream getOutputStream()` The `getOutputStream` method provides access to the output stream of the connection. You can use Nonblocking I/O with the returned stream to implement the new protocol.

The following code demonstrates how to accept an HTTP protocol upgrade request from a client:

```
@WebServlet(urlPatterns={"/xyzpresource"})
public class XYZPUpgradeServlet extends HttpServlet {
   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
      if ("XYZP".equals(request.getHeader("Upgrade"))) {
         /* Accept upgrade request */
         response.setStatus(101);
         response.setHeader("Upgrade", "XYZP");
         response.setHeader("Connection", "Upgrade");
         response.setHeader("OtherHeaderB", "Value");
         /* Delegate the connection to the upgrade handler */
```

```
            XYZPUpgradeHandler = request.upgrade(XYZPUpgradeHandler.class);
            /* (the service method returns immedately) */
        } else {
            /* ... write error response ... */
        }
    }
}
```

The `XYZPUpgradeHandler` class handles the connection:

```
public class XYZPUpgradeHandler implements HttpUpgradeHandler {
    @Override
    public void init(WebConnection wc) {
        ServletInputStream input = wc.getInputStream();
        ServletOutputStream output = wc.getOutputStream();
        /* ... implement XYZP using these streams (protocol-specific) ... */
    }
    @Override
    public void destroy() { ... }
}
```

The class that implements `HttpUpgradeHandler` uses the streams from the current connection to communicate with the client using the new protocol. See the Servlet 5.0 specification at https://jakarta.ee/specifications/servlet/5.0 for details on HTTP protocol upgrade support.

HTTP Trailer

HTTP trailer is a collection of a special type of HTTP headers that comes after the response body. The trailer response header allows the sender to include additional fields at the end of chunked messages in order to supply metadata that might be dynamically generated while the message body is sent, such as a message integrity check, digital signature, or post-processing status.

If trailer headers are ready for reading, `isTrailerFieldsReady()` will return `true`. Then a servlet can read trailer headers of the HTTP request using the `getTrailerFields` method of the `HttpServletRequest` interface. If trailer headers are not ready for reading, `isTrailerFieldsReady()` returns `false` and will cause an `IllegalStateException`.

A servlet can write trailer headers to the response by providing a supplier to the `setTrailerFields()` method of the `HttpServletResponse` interface. The following headers and types of headers must *not* be included in the set of keys in the map passed to `setTrailerFields()`: `Transfer-Encoding`, `Content-Length`, `Host`, controls and conditional headers, authentication headers, `Content-Encoding`, `Content-Type`, `Content-Range`, and `Trailer`. When sending response trailers, you must include a regular header, called `Trailer`, whose value is a comma-separated list of all the keys in the map that is supplied to the `setTrailerFields()` method. The value of the `Trailer` header lets the client know what trailers to expect.

The supplier of the trailer headers can be obtained by accessing the `getTrailerFields()` method of the `HttpServletResponse` interface.

See the javadoc for `getTrailerFields()` and `isTrailerFieldsReady()` in `HttpServletRequest`, and `getTrailerFields()` and `setTrailerFields()` in `HttpServletResponse`.

The mood Example Application

The `mood` example application, located in the `jakartaee-examples/tutorial/web/servlet/mood/` directory, is a simple example that displays Duke's moods at different times during the day. The example shows how to develop a simple application by using the `@WebServlet`, `@WebFilter`, and `@WebListener` annotations to create a servlet, a listener, and a filter.

Components of the mood Example Application

The `mood` example application is comprised of three components: `mood.web.MoodServlet`, `mood.web.TimeOfDayFilter`, and `mood.web.SimpleServletListener`.

`MoodServlet`, the presentation layer of the application, displays Duke's mood in a graphic, based on the time of day. The `@WebServlet` annotation specifies the URL pattern:

```
@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
}
```

`TimeOfDayFilter` sets an initialization parameter indicating that Duke is awake:

```
@WebFilter(filterName = "TimeOfDayFilter",
urlPatterns = {"/*"},
initParams = {
    @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...
}
```

The filter calls the `doFilter` method, which contains a `switch` statement that sets Duke's mood based on the current time.

`SimpleServletListener` logs changes in the servlet's lifecycle. The log entries appear in the server log.

Running the mood Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `mood` example.

To Run the mood Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

```
jakartaee-examples/tutorial/web/servlet
```

4. Select the `mood` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `mood` project and select **Build**.
7. In a web browser, enter the following URL:

   ```
   http://localhost:8080/mood/report
   ```

   The URL specifies the context root, followed by the URL pattern.

   A web page appears with the title "Servlet MoodServlet at /mood", a text string describing Duke's mood, and an illustrative graphic.

To Run the mood Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/servlet/mood/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

4. In a web browser, enter the following URL:

   ```
   http://localhost:8080/mood/report
   ```

   The URL specifies the context root, followed by the URL pattern.

   A web page appears with the title "Servlet MoodServlet at /mood", a text string describing Duke's mood, and an illustrative graphic.

The fileupload Example Application

The `fileupload` example, located in the `jakartaee-examples/tutorial/web/servlet/fileupload/` directory, illustrates how to implement and use the file upload feature.

The Duke's Forest case study provides a more complex example that uploads an image file and stores its content in a database.



Except where expressly provided otherwise, the site, and all content provided on

> or through the site, are provided on an "as is" and "as available" basis. Oracle expressly disclaims all warranties of any kind, whether express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose and non-infringement with respect to the site and all content provided on or through the site. Oracle makes no warranty that: (a) the site or content will meet your requirements; (b) the site will be available on an uninterrupted, timely, secure, or error-free basis; (c) the results that may be obtained from the use of the site or any content provided on or through the site will be accurate or reliable; or (d) the quality of any content purchased or obtained by you on or through the site will meet your expectations.
>
> Any content accessed, downloaded or otherwise obtained on or through the use of the site is used at your own discretion and risk. Oracle shall have no responsibility for any damage to your computer system or loss of data that results from the download or use of content.

Architecture of the fileupload Example Application

The `fileupload` example application consists of a single servlet and an HTML form that makes a file upload request to the servlet.

This example includes a very simple HTML form with two fields, File and Destination. The input type, `file`, enables a user to browse the local file system to select the file. When the file is selected, it is sent to the server as a part of a POST request. During this process, two mandatory restrictions are applied to the form with input type `file`.

- The `enctype` attribute must be set to a value of `multipart/form-data`.
- Its method must be POST.

When the form is specified in this manner, the entire request is sent to the server in encoded form. The servlet then uses its own means to handle the request to process the incoming file data and extract a file from the stream. The destination is the path to the location where the file will be saved on your computer. Pressing the Upload button at the bottom of the form posts the data to the servlet, which saves the file in the specified destination.

The HTML form in `index.html` is as follows:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>File Upload</title>
    </head>
    <body>
        <form method="post" action="upload" enctype="multipart/form-data">
            <div>
                <label>File: <input type="file" name="file" /></label>
            </div>
            <div>
                <label>Destination: <input name="destination" value="/tmp" /></label>
```

```
            </div>
            <div>
                <input type="submit" name="upload" value="Upload" />
            </div>
        </form>
    </body>
</html>
```

A POST request method is used when the client needs to send data to the server as part of the request, such as when uploading a file or submitting a completed form. In contrast, a GET request method sends a URL and headers only to the server, whereas POST requests also include a message body. This allows arbitrary length data of any type to be sent to the server. A header field in the POST request usually indicates the message body's Internet media type.

When submitting a form, the browser streams the content in, combining all parts, with each part representing a field of a form. Parts are named after the `input` elements and are separated from each other with string delimiters named `boundary`.

This is what submitted data from the `fileupload` form looks like, after selecting `sample.txt` as the file that will be uploaded to the `tmp` directory on the local file system:

```
POST /fileupload/upload HTTP/1.1
Host: localhost:8080
Content-Type: multipart/form-data;
boundary=---------------------------263081694432439 Content-Length: 441
-----------------------------263081694432439
Content-Disposition: form-data; name="file"; filename="sample.txt"
Content-Type: text/plain
 Data from sample file
-----------------------------263081694432439
Content-Disposition: form-data; name="destination"
 /tmp
-----------------------------263081694432439
Content-Disposition: form-data; name="upload"
 Upload
-----------------------------263081694432439--
```

The servlet `FileUploadServlet.java` begins as follows:

```
@WebServlet(name = "FileUploadServlet", urlPatterns = {"/upload"})
@MultipartConfig
public class FileUploadServlet extends HttpServlet {
    private final static Logger LOGGER =
            Logger.getLogger(FileUploadServlet.class.getCanonicalName());
}
```

The `@WebServlet` annotation uses the `urlPatterns` property to define servlet mappings.

The `@MultipartConfig` annotation indicates that the servlet expects requests to be made using the `multipart/form-data` MIME type.

The `processRequest` method retrieves the destination and file part from the request, then calls the `getFileName` method to retrieve the file name from the file part. The method then creates a `FileOutputStream` and copies the file to the specified destination. The error-handling section of the method catches and handles some of the most common reasons why a file would not be found. The `processRequest` and `getFileName` methods look like this:

```
protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");

    // Create path components to save the file
    final String path = request.getParameter("destination");
    final Part filePart = request.getPart("file");
    final String fileName = getFileName(filePart);

    OutputStream out = null;
    InputStream filecontent = null;
    final PrintWriter writer = response.getWriter();

    try {
        out = new FileOutputStream(new File(path + File.separator
                + fileName));
        filecontent = filePart.getInputStream();

        int read = 0;
        final byte[] bytes = new byte[1024];

        while ((read = filecontent.read(bytes)) != -1) {
            out.write(bytes, 0, read);
        }
        writer.println("New file " + fileName + " created at " + path);
        LOGGER.log(Level.INFO, "File{0}being uploaded to {1}",
                new Object[]{fileName, path});
    } catch (FileNotFoundException fne) {
        writer.println("You either did not specify a file to upload or are "
                + "trying to upload a file to a protected or nonexistent "
                + "location.");
        writer.println("<br/> ERROR: " + fne.getMessage());

        LOGGER.log(Level.SEVERE, "Problems during file upload. Error: {0}",
                new Object[]{fne.getMessage()});
    } finally {
        if (out != null) {
            out.close();
        }
        if (filecontent != null) {
```

```
            filecontent.close();
        }
        if (writer != null) {
            writer.close();
        }
    }
}

private String getFileName(final Part part) {
    final String partHeader = part.getHeader("content-disposition");
    LOGGER.log(Level.INFO, "Part Header = {0}", partHeader);
    for (String content : part.getHeader("content-disposition").split(";")) {
        if (content.trim().startsWith("filename")) {
            return content.substring(
                    content.indexOf('=') + 1).trim().replace("\"", "");
        }
    }
    return null;
}
```

Running the fileupload Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `fileupload` example.

To Build, Package, and Deploy the fileupload Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/servlet
   ```

4. Select the `fileupload` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `fileupload` project and select **Build**.

To Build, Package, and Deploy the fileupload Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/servlet/fileupload/
   ```

3. Enter the following command to deploy the application:

```
mvn install
```

To Run the fileupload Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/fileupload/
   ```

2. On the File Upload page, click Choose File to display a file browser window.
3. Select a file to upload and click Open.

   The name of the file you selected is displayed in the File field. If you do not select a file, an exception will be thrown.

4. In the Destination field, type a directory name.

   The directory must have already been created and must also be writable. If you do not enter a directory name or if you enter the name of a nonexistent or protected directory, an exception will be thrown.

5. Click Upload to upload the file that you selected to the directory that you specified in the Destination field.

   A message reports that the file was created in the directory that you specified.

6. Go to the directory that you specified in the Destination field and verify that the uploaded file is present.

The dukeetf Example Application

The `dukeetf` example application, located in the `jakartaee-examples/tutorial/web/dukeetf/` directory, demonstrates how to use asynchronous processing in a servlet to provide data updates to web clients. The example resembles a service that provides periodic updates on the price and trading volume of an electronically traded fund (ETF).

Architecture of the dukeetf Example Application

The `dukeetf` example application consists of a servlet, an enterprise bean, and an HTML page.

- The servlet puts requests in asynchronous mode, stores them in a queue, and writes the responses when new data for price and trading volume becomes available.
- The enterprise bean updates the price and volume information once every second.
- The HTML page uses JavaScript code to make requests to the servlet for new data, parse the response from the servlet, and update the price and volume information without reloading the page.

The `dukeetf` example application uses a programming model known as long polling. In the

traditional HTTP request and response model, the user must make an explicit request (such as clicking a link or submitting a form) to get any new information from the server, and the page has to be reloaded. Long polling provides a mechanism for web applications to push updates to clients using HTTP without the user making an explicit request. The server handles connections asynchronously, and the client uses JavaScript to make new connections. In this model, clients make a new request immediately after receiving new data, and the server keeps the connection open until new data becomes available.

The Servlet

The `DukeETFServlet` class uses asynchronous processing:

```
@WebServlet(urlPatterns={"/dukeetf"}, asyncSupported=true)
public class DukeETFServlet extends HttpServlet {
...
}
```

In the following code, the `init` method initializes a queue to hold client requests and registers the servlet with the enterprise bean that provides the price and volume updates. The `send` method gets called once per second by the `PriceVolumeBean` to send updates and close the connection:

```
@Override
public void init(ServletConfig config) {
   /* Queue for requests */
   requestQueue = new ConcurrentLinkedQueue<>();
   /* Register with the enterprise bean that provides price/volume updates */
   pvbean.registerServlet(this);
}

/* PriceVolumeBean calls this method every second to send updates */
public void send(double price, int volume) {
   /* Send update to all connected clients */
   for (AsyncContext acontext : requestQueue) {
      try {
         String msg = String.format("%.2f / %d", price, volume);
         PrintWriter writer = acontext.getResponse().getWriter();
         writer.write(msg);
         logger.log(Level.INFO, "Sent: {0}", msg);
         /* Close the connection
          * The client (JavaScript) makes a new one instantly */
         acontext.complete();
      } catch (IOException ex) {
         logger.log(Level.INFO, ex.toString());
      }
   }
}
```

The service method puts client requests in asynchronous mode and adds a listener to each request.

The listener is implemented as an anonymous class that removes the request from the queue when the servlet finishes writing a response or when there is an error. Finally, the service method adds the request to the request queue created in the `init` method. The service method is the following:

```
@Override
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
   response.setContentType("text/html");
   /* Put request in async mode */
   final AsyncContext acontext = request.startAsync();
   /* Remove from the queue when done */
   acontext.addListener(new AsyncListener() {
      public void onComplete(AsyncEvent ae) throws IOException {
         requestQueue.remove(acontext);
      }
      public void onTimeout(AsyncEvent ae) throws IOException {
         requestQueue.remove(acontext);
      }
      public void onError(AsyncEvent ae) throws IOException {
         requestQueue.remove(acontext);
      }
      public void onStartAsync(AsyncEvent ae) throws IOException {}
   });
   /* Add to the queue */
   requestQueue.add(acontext);
}
```

The Enterprise Bean

The `PriceVolumeBean` class is an enterprise bean that uses the timer service from the container to update the price and volume information and call the servlet's `send` method once every second:

```
@Startup
@Singleton
public class PriceVolumeBean {
    /* Use the container's timer service */
    @Resource TimerService tservice;
    private DukeETFServlet servlet;
    ...

    @PostConstruct
    public void init() {
        /* Initialize the EJB and create a timer */
        random = new Random();
        servlet = null;
        tservice.createIntervalTimer(1000, 1000, new TimerConfig());
    }

    public void registerServlet(DukeETFServlet servlet) {
```

```
        /* Associate a servlet to send updates to */
        this.servlet = servlet;
    }

    @Timeout
    public void timeout() {
        /* Adjust price and volume and send updates */
        price += 1.0*(random.nextInt(100)-50)/100.0;
        volume += random.nextInt(5000) - 2500;
        if (servlet != null)
            servlet.send(price, volume);
    }
}
```

See Using the Timer Service in [entbeans:ejb-basicexamples::ejb-basicexamples:::_running_the_enterprise_bean_examples] for more information on the timer service.

The HTML Page

The HTML page consists of a table and some JavaScript code. The table contains two fields referenced from JavaScript code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>...</head>
<body onload="makeAjaxRequest();">
  ...
  <table>
    ...
    <td id="price">--.--</td>
    ...
    <td id="volume">--</td>
    ...
  </table>
</body>
</html>
```

The JavaScript code uses the `XMLHttpRequest` API, which provides functionality for transferring data between a client and a server. The script makes an asynchronous request to the servlet and designates a callback method. When the server provides a response, the callback method updates the fields in the table and makes a new request. The JavaScript code is the following:

```
var ajaxRequest;
function updatePage() {
   if (ajaxRequest.readyState === 4) {
      var arraypv = ajaxRequest.responseText.split("/");
      document.getElementById("price").innerHTML = arraypv[0];
      document.getElementById("volume").innerHTML = arraypv[1];
```

```
        makeAjaxRequest();
    }
}
function makeAjaxRequest() {
    ajaxRequest = new XMLHttpRequest();
    ajaxRequest.onreadystatechange = updatePage;
    ajaxRequest.open("GET", "http://localhost:8080/dukeetf/dukeetf",
                     true);
    ajaxRequest.send(null);
}
```

The `XMLHttpRequest` API is supported by most modern browsers, and it is widely used in Ajax web client development (Asynchronous JavaScript and XML).

See The dukeetf2 Example Application in [web:websocket::websocket:::_jakarta_websocket] for an equivalent version of this example implemented using a WebSocket endpoint.

Running the dukeetf Example Application

This section describes how to run the `dukeetf` example application using NetBeans IDE and from the command line.

To Run the dukeetf Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/servlet
   ```

4. Select the `dukeetf` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `dukeetf` project and select **Run**.

   This command builds and packages the application into a WAR file (`dukeetf.war`) located in the `target` directory, deploys it to the server, and launches a web browser window with the following URL:

   ```
   http://localhost:8080/dukeetf/
   ```

   Open the same URL in a different web browser to see how both pages get price and volume updates simultaneously.

To Run the dukeetf Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).

2. In a terminal window, go to:

```
jakartaee-examples/tutorial/web/servlet/dukeetf/
```

3. Enter the following command to deploy the application:

```
mvn install
```

4. Open a web browser window and type the following address:

```
http://localhost:8080/dukeetf/
```

   Open the same URL in a different web browser to see how both pages get price and volume updates simultaneously.

Further Information about Jakarta Servlet Technology

For more information on Jakarta Servlet technology, see the Jakarta Servlet 6.0 specification at https://jakarta.ee/specifications/servlet/6.0/.

The Jakarta Servlet 6.0 specification was released with Jakarta EE 10. For more about the changes in this release please see this article.



The Servlet 6.0 specification permanently removes several methods and classes that had been deprecated in the Servlet 5.0 specification, please see the list here: https://jakarta.ee/specifications/servlet/5.0/apidocs/deprecated-list.html.

Jakarta Faces

Jakarta Faces Technology



This section was last updated for Jakarta EE Version 9.1

Jakarta Faces technology is a server-side component framework for building Java technology–based web applications.

Introduction to Jakarta Faces Technology

Jakarta Faces technology consists of the following:

- An API for representing components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features
- Tag libraries for adding components to web pages and for connecting components to server-side objects

Jakarta Faces technology provides a well-defined programming model and various tag libraries. The tag libraries contain tag handlers that implement the component tags. These features significantly ease the burden of building and maintaining web applications with server-side user interfaces (UIs). With minimal effort, you can complete the following tasks.

- Create a web page.
- Drop components onto a web page by adding component tags.
- Bind components on a page to server-side data.
- Wire component-generated events to server-side application code.
- Save and restore application state beyond the life of server requests.
- Reuse and extend components through customization.

This chapter provides an overview of Jakarta Faces technology. After explaining what a Jakarta Faces application is and reviewing some of the primary benefits of using Jakarta Faces technology, this chapter describes the process of creating a simple Jakarta Faces application. This chapter also introduces the Jakarta Faces lifecycle by describing the example Jakarta Faces application and its progression through the lifecycle stages.

What Is a Jakarta Faces Application?

The functionality provided by a Jakarta Faces application is similar to that of any other Java web application. A typical Jakarta Faces application includes the following parts.

- A set of web pages in which components are laid out.
- A set of tags to add components to the web page.
- A set of managed beans, which are lightweight, container-managed objects (POJOs). In a Jakarta Faces application, managed beans serve as backing beans, which define properties and functions for UI components on a page.
- A web deployment descriptor (`web.xml` file).
- Optionally, one or more application configuration resource files, such as a `faces-config.xml` file, which can be used to define page navigation rules and configure beans and other custom objects, such as custom components.
- Optionally, a set of custom objects, which can include custom components, validators, converters, or listeners, created by the application developer.
- Optionally, a set of custom tags for representing custom objects on the page.

Figure 10, “Responding to a Client Request for a Jakarta Faces Page” shows the interaction between client and server in a typical Jakarta Faces application. In response to a client request, a web page is rendered by the web container that implements Jakarta Faces technology.

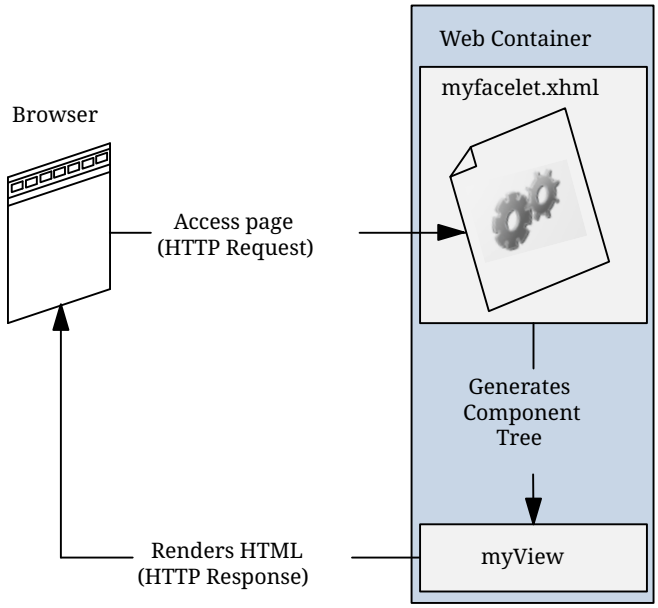


Figure 10. Responding to a Client Request for a Jakarta Faces Page

The web page, `myfacelet.xhtml`, is built using Jakarta Faces component tags. Component tags are used to add components to the `view` (represented by `myView` in the diagram), which is the server-side representation of the page. In addition to components, the web page can also reference objects, such as the following:

- Any event listeners, validators, and converters that are registered on the components
- The JavaBeans components that capture the data and process the application-specific functionality of the components

On request from the client, the view is rendered as a response. Rendering is the process whereby, based on the server-side view, the web container generates output, such as HTML or XHTML, that can be read by the client, such as a browser.

Jakarta Faces Technology Benefits

One of the greatest advantages of Jakarta Faces technology is that it offers a clean separation between behavior and presentation for web applications. A Jakarta Faces application can map HTTP requests to component-specific event handling and manage components as stateful objects on the server. Jakarta Faces technology allows you to build web applications that implement the finer-grained separation of behavior and presentation that is traditionally offered by client-side UI architectures.

The separation of logic from presentation also allows each member of a web application development team to focus on a single piece of the development process and provides a simple programming model to link the pieces. For example, page authors with no programming expertise can use Jakarta Faces technology tags in a web page to link to server-side objects without writing any scripts.

Another important goal of Jakarta Faces technology is to leverage familiar component and web-tier concepts without limiting you to a particular scripting technology or markup language. Jakarta Faces technology APIs are layered directly on top of the Servlet API, as shown in Figure 11, “Web Application Technologies”.



Figure 11. Web Application Technologies

This layering of APIs enables several important application use cases, such as using different presentation technologies, creating your own custom components directly from the component classes, and generating output for various client devices.

Facelets technology, available as part of Jakarta Faces technology, is the preferred presentation technology for building Jakarta Faces technology–based web applications. For more information on Facelets technology features, see [web:faces-facelets::faces-facelets:::_introduction_to_facelets].

Facelets technology offers several advantages.

- Code can be reused and extended for components through the templating and composite component features.
- You can use annotations to automatically register the managed bean as a resource available for Jakarta Faces applications. In addition, implicit navigation rules allow developers to quickly configure page navigation (see Navigation Model for details). These features reduce the manual configuration process for applications.
- Most important, Jakarta Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.

A Simple Jakarta Faces Application

Jakarta Faces technology provides an easy and user-friendly process for creating web applications. Developing a simple Jakarta Faces application typically requires the following tasks, which have already been described in A Web Module That Uses Jakarta Faces Technology: The hello1 Example:

- Creating web pages using component tags
- Developing managed beans
- Mapping the `FacesServlet` instance

The `hello1` example includes a managed bean and two Facelets web pages. When accessed by a client, the first web page asks the user for his or her name, and the second page responds by providing a greeting.

For details on Facelets technology, see [web:faces-facelets::faces-facelets:::_introduction_to_facelets]. For details on using EL expressions, see Expression Language. For details on the Jakarta Faces programming model and building web pages using Jakarta Faces technology, see [web:faces-page::faces-page:::_using_jakarta_faces_technology_in_web_pages].

Every web application has a lifecycle. Common tasks, such as handling incoming requests, decoding parameters, modifying and saving state, and rendering web pages to the browser, are all performed during a web application lifecycle. Some web application frameworks hide the details of

the lifecycle from you, whereas others require you to manage them manually.

By default, Jakarta Faces automatically handles most of the lifecycle actions for you. However, it also exposes the various stages of the request lifecycle so that you can modify or perform different actions if your application requirements warrant it.

The lifecycle of a Jakarta Faces application starts and ends with the following activity: The client makes a request for the web page, and the server responds with the page. The lifecycle consists of two main phases: Execute and Render.

During the Execute phase, several actions can take place.

- The application view is built or restored.
- The request parameter values are applied.
- Conversions and validations are performed for component values.
- Managed beans are updated with component values.
- Application logic is invoked.

For a first (initial) request, only the view is built. For subsequent (postback) requests, some or all of the other actions can take place.

In the Render phase, the requested view is rendered as a response to the client. Rendering is typically the process of generating output, such as HTML or XHTML, that can be read by the client, usually a browser.

The following short description of the example Jakarta Faces application passing through its lifecycle summarizes the activity that takes place behind the scenes.

The `hello1` example application goes through the following stages when it is deployed on GlassFish Server.

1. When the `hello1` application is built and deployed on GlassFish Server, the application is in an uninitiated state.
2. When a client makes an initial request for the `index.xhtml` web page, the `hello1` Facelets application is compiled.
3. The compiled Facelets application is executed, and a new component tree is constructed for the `hello1` application and placed in a `FacesContext`.
4. The component tree is populated with the component and the managed bean property associated with it, represented by the EL expression `hello.name`.
5. A new view is built, based on the component tree.
6. The view is rendered to the requesting client as a response.
7. The component tree is destroyed automatically.
8. On subsequent (postback) requests, the component tree is rebuilt, and the saved state is applied.

For full details on the lifecycle, see The Lifecycle of a Jakarta Faces Application.

User Interface Component Model

In addition to the lifecycle description, an overview of Jakarta Faces architecture provides better understanding of the technology.

Jakarta Faces components are the building blocks of a Jakarta Faces view. A component can be a user interface (UI) component or a non-UI component.

Jakarta Faces UI components are configurable, reusable elements that compose the user interfaces of Jakarta Faces applications. A component can be simple, such as a button, or can be compound, such as a table composed of multiple components.

Jakarta Faces technology provides a rich, flexible component architecture that includes the following:

- A set of `jakarta.faces.component.UIComponent` classes for specifying the state and behavior of UI components
- A rendering model that defines how to render the components in various ways
- A conversion model that defines how to register data converters onto a component
- An event and listener model that defines how to handle component events
- A validation model that defines how to register validators onto a component

This section briefly describes each of these pieces of the component architecture.

User Interface Component Classes

Jakarta Faces technology provides a set of UI component classes and associated behavioral interfaces that specify all the UI component functionality, such as holding component state, maintaining a reference to objects, and driving event handling and rendering for a set of standard components.

The component classes are completely extensible, allowing component writers to create their own custom components. See [web:faces-custom::faces-custom:::_creating_custom_ui_components_and_other_custom_objects] for more information.

The abstract base class for all components is `jakarta.faces.component.UIComponent`. Jakarta Faces UI component classes extend the `UIComponentBase` class (a subclass of `UIComponent`), which defines the default state and behavior of a component. The following set of component classes is included with Jakarta Faces technology.

- `UIColumn`: Represents a single column of data in a `UIData` component.
- `UICommand`: Represents a control that fires actions when activated.
- `UIData`: Represents a data binding to a collection of data represented by a `jakarta.faces.model.DataModel` instance.
- `UIForm`: Represents an input form to be presented to the user. Its child components represent (among other things) the input fields to be included when the form is submitted. This component is analogous to the `form` tag in HTML.

- `UIGraphic`: Displays an image.
- `UIInput`: Takes data input from a user. This class is a subclass of `UIOutput`.
- `UIMessage`: Displays a localized error message.
- `UIMessages`: Displays a set of localized error messages.
- `UIOutcomeTarget`: Displays a link in the form of a link or a button.
- `UIOutput`: Displays data output on a page.
- `UIPanel`: Manages the layout of its child components.
- `UIParameter`: Represents substitution parameters.
- `UISelectBoolean`: Allows a user to set a `boolean` value on a control by selecting or deselecting it. This class is a subclass of the `UIInput` class.
- `UISelectItem`: Represents a single item in a set of items.
- `UISelectItems`: Represents an entire set of items.
- `UISelectMany`: Allows a user to select multiple items from a group of items. This class is a subclass of the `UIInput` class.
- `UISelectOne`: Allows a user to select one item from a group of items. This class is a subclass of the `UIInput` class.
- `UIViewParameter`: Represents the query parameters in a request. This class is a subclass of the `UIInput` class.
- `UIViewRoot`: Represents the root of the component tree.

In addition to extending `UIComponentBase`, the component classes also implement one or more behavioral interfaces, each of which defines certain behavior for a set of components whose classes implement the interface.

These behavioral interfaces, all defined in the `jakarta.faces.component` package unless otherwise stated, are as follows.

- `ActionSource`: Indicates that the component can fire an action event. This interface is intended for use with components based on JavaServer Faces technology 1.1_01 and earlier versions. This interface is deprecated in JavaServer Faces 2.
- `ActionSource2`: Extends `ActionSource` and therefore provides the same functionality. However, it allows components to use the Expression Language (EL) when they are referencing methods that handle action events.
- `EditableValueHolder`: Extends `ValueHolder` and specifies additional features for editable components, such as validation and emitting value-change events.
- `NamingContainer`: Mandates that each component rooted at this component have a unique ID.
- `StateHolder`: Denotes that a component has state that must be saved between requests.
- `ValueHolder`: Indicates that the component maintains a local value as well as the option of accessing data in the model tier.
- `jakarta.faces.event.SystemEventListenerHolder`: Maintains a list of `jakarta.faces.event.SystemEventListener` instances for each type of

`jakarta.faces.event.SystemEvent` defined by that class.

- `jakarta.faces.component.behavior.ClientBehaviorHolder`: Adds the ability to attach `jakarta.faces.component.behavior.ClientBehavior` instances, such as a reusable script.

`UICommand` implements `ActionSource2` and `StateHolder`. `UIOutput` and component classes that extend `UIOutput` implement `StateHolder` and `ValueHolder`. `UIInput` and component classes that extend `UIInput` implement `EditableValueHolder`, `StateHolder`, and `ValueHolder`. `UIComponentBase` implements `StateHolder`.

Only component writers will need to use the component classes and behavioral interfaces directly. Page authors and application developers will use a standard component by including a tag that represents it on a page. Most of the components can be rendered in different ways on a page. For example, a `UICommand` component can be rendered as a button or a link.

The next section explains how the rendering model works and how page authors can choose to render the components by selecting the appropriate tags.

Component Rendering Model

The Jakarta Faces component architecture is designed such that the functionality of the components is defined by the component classes, whereas the component rendering can be defined by a separate renderer class. This design has several benefits, including the following.

- Component writers can define the behavior of a component once but create multiple renderers, each of which defines a different way to render the component to the same client or to different clients.
- Page authors and application developers can change the appearance of a component on the page by selecting the tag that represents the appropriate combination of component and renderer.

A render kit defines how component classes map to component tags that are appropriate for a particular client. The Jakarta Faces implementation includes a standard HTML render kit for rendering to an HTML client.

The render kit defines a set of `jakarta.faces.render.Renderer` classes for each component that it supports. Each `Renderer` class defines a different way to render the particular component to the output defined by the render kit. For example, a `UISelectOne` component has three different renderers. One of them renders the component as a group of options. Another renders the component as a combo box. The third one renders the component as a list box. Similarly, a `UICommand` component can be rendered as a button or a link, using the `h:commandButton` or `h:commandLink` tag. The `command` part of each tag corresponds to the `UICommand` class, specifying the functionality, which is to fire an action. The `Button` or `Link` part of each tag corresponds to a separate `Renderer` class that defines how the component appears on the page.

Each custom tag defined in the standard HTML render kit is composed of the component functionality (defined in the `UIComponent` class) and the rendering attributes (defined by the `Renderer` class).

The section Adding Components to a Page Using HTML Tag Library Tags lists all supported

component tags and illustrates how to use the tags in an example.

The Jakarta Faces implementation provides a custom tag library for rendering components in HTML.

Conversion Model

A Jakarta Faces application can optionally associate a component with server-side object data. This object is a JavaBeans component, such as a managed bean. An application gets and sets the object data for a component by calling the appropriate object properties for that component.

When a component is bound to an object, the application has two views of the component's data.

- The model view, in which data is represented as data types, such as `int` or `long`.
- The presentation view, in which data is represented in a manner that can be read or modified by the user. For example, a `java.util.Date` might be represented as a text string in the format `mm/dd/yy` or as a set of three text strings.

The Jakarta Faces implementation automatically converts component data between these two views when the bean property associated with the component is of one of the types supported by the component's data. For example, if a `UISelectBoolean` component is associated with a bean property of type `java.lang.Boolean`, the Jakarta Faces implementation will automatically convert the component's data from `String` to `Boolean`. In addition, some component data must be bound to properties of a particular type. For example, a `UISelectBoolean` component must be bound to a property of type `boolean` or `java.lang.Boolean`.

Sometimes you might want to convert a component's data to a type other than a standard type, or you might want to convert the format of the data. To facilitate this, Jakarta Faces technology allows you to register a `jakarta.faces.convert.Converter` implementation on `UIOutput` components and components whose classes subclass `UIOutput`. If you register the `Converter` implementation on a component, the `Converter` implementation converts the component's data between the two views.

You can either use the standard converters supplied with the Jakarta Faces implementation or create your own custom converter. Custom converter creation is covered in [web:faces-custom::faces-custom:::_creating_custom_ui_components_and_other_custom_objects].

Event and Listener Model

The Jakarta Faces event and listener model is similar to the JavaBeans event model in that it has strongly typed event classes and listener interfaces that an application can use to handle events generated by components.

The Jakarta Faces specification defines three types of events: application events, system events, and data-model events.

Application events are tied to a particular application and are generated by a `UIComponent`. They represent the standard events available in previous versions of Jakarta Faces technology.

An event object identifies the component that generated the event and stores information about the event. To be notified of an event, an application must provide an implementation of the listener

class and must register it on the component that generates the event. When the user activates a component, such as by clicking a button, an event is fired. This causes the Jakarta Faces implementation to invoke the listener method that processes the event.

Jakarta Faces supports two kinds of application events: action events and value-change events.

An action event (class `jakarta.faces.event.ActionEvent`) occurs when the user activates a component that implements `ActionSource`. These components include buttons and links.

A value-change event (class `jakarta.faces.event.ValueChangeEvent`) occurs when the user changes the value of a component represented by `UIInput` or one of its subclasses. An example is selecting a check box, an action that results in the component's value changing to `true`. The component types that can generate these types of events are the `UIInput`, `UISelectOne`, `UISelectMany`, and `UISelectBoolean` components. Value-change events are fired only if no validation errors are detected.

Depending on the value of the `immediate` property (see The immediate Attribute) of the component emitting the event, action events can be processed during the Invoke Application phase or the Apply Request Values phase, and value-change events can be processed during the Process Validations phase or the Apply Request Values phase.

System events are generated by an `Object` rather than a `UIComponent`. They are generated during the execution of an application at predefined times. They are applicable to the entire application rather than to a specific component.

A data-model event occurs when a new row of a `UIData` component is selected.

There are two ways to cause your application to react to action events or value-change events that are emitted by a standard component:

- Implement an event listener class to handle the event, and register the listener on the component by nesting either an `f:valueChangeListener` tag or an `f:actionListener` tag inside the component tag.
- Implement a method of a managed bean to handle the event, and refer to the method with a method expression from the appropriate attribute of the component's tag.

See Implementing an Event Listener for information on how to implement an event listener. See Registering Listeners on Components for information on how to register the listener on a component.

See Writing a Method to Handle an Action Event and Writing a Method to Handle a Value-Change Event for information on how to implement managed bean methods that handle these events.

See Referencing a Managed Bean Method for information on how to refer to the managed bean method from the component tag.

When emitting events from custom components, you must implement the appropriate event class and manually queue the event on the component in addition to implementing an event listener class or a managed bean method that handles the event. Handling Events for Custom Components explains how to do this.

Validation Model

Jakarta Faces technology supports a mechanism for validating the local data of editable components (such as text fields). This validation occurs before the corresponding model data is updated to match the local value.

Like the conversion model, the validation model defines a set of standard classes for performing common data validation checks. The Jakarta Faces core tag library also defines a set of tags that correspond to the standard `jakarta.faces.validator.Validator` implementations. See Using the Standard Validators for a list of all the standard validation classes and corresponding tags.

Most of the tags have a set of attributes for configuring the validator's properties, such as the minimum and maximum allowable values for the component's data. The page author registers the validator on a component by nesting the validator's tag within the component's tag.

In addition to validators that are registered on the component, you can declare a default validator that is registered on all `UIInput` components in the application. For more information on default validators, see Using Default Validators.

The validation model also allows you to create your own custom validator and corresponding tag to perform custom validation. The validation model provides two ways to implement custom validation.

- Implement a `Validator` interface that performs the validation.
- Implement a managed bean method that performs the validation.

If you are implementing a `Validator` interface, you must also do the following.

- Register the `Validator` implementation with the application.
- Create a custom tag or use an `f:validator` tag to register the validator on the component.

In the previously described standard validation model, the validator is defined for each input component on a page. The Bean Validation model allows the validator to be applied to all fields in a page. See [beanvalidation:bean-validation::bean-validation:::_introduction_to_jakarta_bean_validation] and [beanvalidation:bean-validation-advanced::bean-validation-advanced:::_bean_validation_advanced_topics] for more information on Bean Validation.

Navigation Model

The Jakarta Faces navigation model makes it easy to define page navigation and to handle any additional processing that is needed to choose the sequence in which pages are loaded.

In Jakarta Faces technology, navigation is a set of rules for choosing the next page or view to be displayed after an application action, such as when a button or link is clicked.

Navigation can be implicit or user-defined. Implicit navigation comes into play when user-defined navigation rules are not configured in the application configuration resource files.

When you add a component such as a `commandButton` to a Facelets page, and assign another page as

the value for its `action` property, the default navigation handler will try to match a suitable page within the application implicitly. In the following example, the default navigation handler will try to locate a page named `response.xhtml` within the application and navigate to it:

```
<h:commandButton value="submit" action="response">
```

User-defined navigation rules are declared in zero or more application configuration resource files, such as `faces-config.xml`, by using a set of XML elements. The default structure of a navigation rule is as follows:

```
<navigation-rule>
    <description></description>
    <from-view-id></from-view-id>
    <navigation-case>
        <from-action></from-action>
        <from-outcome></from-outcome>
        <if></if>
        <to-view-id></to-view-id>
    </navigation-case>
</navigation-rule>
```

User-defined navigation is handled as follows.

- Define the rules in the application configuration resource file.
- Refer to an outcome `String` from the button or link component's `action` attribute. This outcome `String` is used by the Jakarta Faces implementation to select the navigation rule.

Here is an example navigation rule:

```
<navigation-rule>
    <from-view-id>/greeting.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/response.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

This rule states that when a command component (such as an `h:commandButton` or an `h:commandLink`) on `greeting.xhtml` is activated, the application will navigate from the `greeting.xhtml` page to the `response.xhtml` page if the outcome referenced by the button component's tag is `success`. Here is an `h:commandButton` tag from `greeting.xhtml` that would specify a logical outcome of `success`:

```
<h:commandButton id="submit" value="Submit" action="success"/>
```

As the example demonstrates, each `navigation-rule` element defines how to get from one page

(specified in the `from-view-id` element) to the other pages of the application. The `navigation-rule` elements can contain any number of `navigation-case` elements, each of which defines the page to open next (defined by `to-view-id`) based on a logical outcome (defined by `from-outcome`).

In more complicated applications, the logical outcome can also come from the return value of an action method in a managed bean. This method performs some processing to determine the outcome. For example, the method can check whether the password the user entered on the page matches the one on file. If it does, the method might return `success`; otherwise, it might return `failure`. An outcome of `failure` might result in the logon page being reloaded. An outcome of `success` might cause the page displaying the user's credit card activity to open. If you want the outcome to be returned by a method on a bean, you must refer to the method using a method expression with the `action` attribute, as shown by this example:

```
<h:commandButton id="submit" value="Submit"
                 action="#{cashierBean.submit}" />
```

When the user clicks the button represented by this tag, the corresponding component generates an action event. This event is handled by the default `jakarta.faces.event.ActionListener` instance, which calls the action method referenced by the component that triggered the event. The action method returns a logical outcome to the action listener.

The listener passes the logical outcome and a reference to the action method that produced the outcome to the default `jakarta.faces.application.NavigationHandler`. The `NavigationHandler` selects the page to display next by matching the outcome or the action method reference against the navigation rules in the application configuration resource file by the following process.

1. The `NavigationHandler` selects the navigation rule that matches the page currently displayed.
2. It matches the outcome or the action method reference that it received from the default `jakarta.faces.event.ActionListener` with those defined by the navigation cases.
3. It tries to match both the method reference and the outcome against the same navigation case.
4. If the previous step fails, the navigation handler attempts to match the outcome.
5. Finally, the navigation handler attempts to match the action method reference if the previous two attempts failed.
6. If no navigation case is matched, it displays the same view again.

When the `NavigationHandler` achieves a match, the Render Response phase begins. During this phase, the page selected by the `NavigationHandler` will be rendered.

The Duke's Tutoring case study example application uses navigation rules in the business methods that handle creating, editing, and deleting the users of the application. For example, the form for creating a student has the following `h:commandButton` tag:

```
<h:commandButton id="submit"
        action="#{adminBean.createStudent(studentManager.newStudent)}"
        value="#{bundle['action.submit']}"/>
```

The action event calls the `dukestutoring.ejb.AdminBean.createStudent` method:

```
public String createStudent(Student student) {
    em.persist(student);
    return "createdStudent";
}
```

The return value of `createdStudent` has a corresponding navigation case in the `faces-config.xml` configuration file:

```
<navigation-rule>
    <from-view-id>/admin/student/createStudent.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>createdStudent</from-outcome>
        <to-view-id>/admin/index.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

After the student is created, the user is returned to the Administration index page.

For more information on how to define navigation rules, see Configuring Navigation Rules.

For more information on how to implement action methods to handle navigation, see Writing a Method to Handle an Action Event.

For more information on how to reference outcomes or action methods from component tags, see Referencing a Method That Performs Navigation.

The Lifecycle of a Jakarta Faces Application

The lifecycle of an application refers to the various stages of processing of that application, from its initiation to its conclusion. All applications have lifecycles. During a web application lifecycle, common tasks are performed, including the following.

- Handling incoming requests
- Decoding parameters
- Modifying and saving state
- Rendering web pages to the browser

The Jakarta Faces web application framework manages lifecycle phases automatically for simple applications or allows you to manage them manually for more complex applications as required.

Jakarta Faces applications that use advanced features may require interaction with the lifecycle at certain phases. For example, Ajax applications use partial processing features of the lifecycle (see Partial Processing and Partial Rendering). A clearer understanding of the lifecycle phases is key to creating well-designed components.

A simplified view of the Jakarta Faces lifecycle, consisting of the two main phases of a Jakarta Faces web application, is introduced in A Simple Jakarta Faces Application. This section examines the Jakarta Faces lifecycle in more detail.

Overview of the Jakarta Faces Lifecycle

The lifecycle of a Jakarta Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page, translated to HTML.

The lifecycle can be divided into two main phases: Execute and Render. The Execute phase is further divided into subphases to support the sophisticated component tree. This structure requires that component data be converted and validated, component events be handled, and component data be propagated to beans in an orderly fashion.

A Jakarta Faces page is represented by a tree of components, called a view. During the lifecycle, the Jakarta Faces implementation must build the view while considering the state saved from a previous submission of the page. When the client requests a page, the Jakarta Faces implementation performs several tasks, such as validating the data input of components in the view and converting input data to types specified on the server side.

The Jakarta Faces implementation performs all these tasks as a series of steps in the Jakarta Faces request-response lifecycle. Figure 12, “Jakarta Faces Standard Request-Response Lifecycle” illustrates these steps.

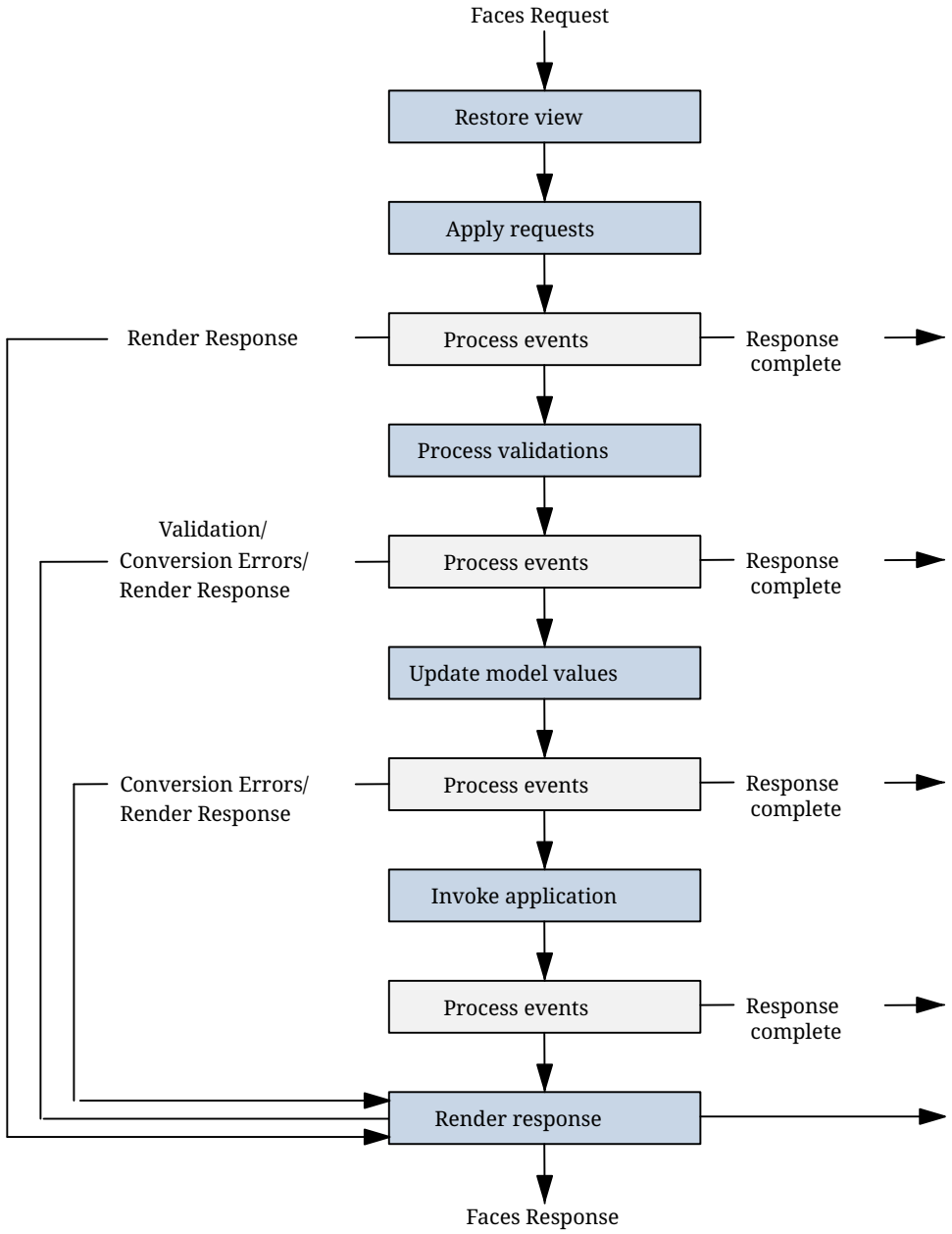


Figure 12. Jakarta Faces Standard Request-Response Lifecycle

The request-response lifecycle handles two kinds of requests: initial requests and postbacks. An initial request occurs when a user makes a request for a page for the first time. A postback request occurs when a user submits the form contained on a page that was previously loaded into the browser as a result of executing an initial request.

When the lifecycle handles an initial request, it executes only the Restore View and Render Response phases, because there is no user input or action to process. Conversely, when the lifecycle handles a postback, it executes all of the phases.

Usually, the first request for a Jakarta Faces page comes in from a client, as a result of clicking a link or button component on a Jakarta Faces page. To render a response that is another Jakarta Faces page, the application creates a new view and stores it in the `jakarta.faces.context.FacesContext` instance, which represents all of the information associated with processing an incoming request and creating a response. The application then acquires object references needed by the view and calls the `FacesContext.renderResponse` method, which forces immediate rendering of the view by

skipping to the Render Response Phase of the lifecycle, as is shown by the arrows labelled Render Response in Figure 12, “Jakarta Faces Standard Request-Response Lifecycle”.

Sometimes, an application might need to redirect to a different web application resource, such as a web service, or generate a response that does not contain Jakarta Faces components. In these situations, the developer must skip the Render Response phase by calling the `FacesContext.responseComplete` method. This situation is also shown in , with the arrows labelled Response Complete.

The most common situation is that a Jakarta Faces component submits a request for another Jakarta Faces page. In this case, the Jakarta Faces implementation handles the request and automatically goes through the phases in the lifecycle to perform any necessary conversions, validations, and model updates and to generate the response.

There is one exception to the lifecycle described in this section. When a component’s `immediate` attribute is set to `true`, the validation, conversion, and events associated with these components are processed during the Apply Request Values Phase rather than in a later phase.

The details of the lifecycle explained in the following sections are primarily intended for developers who need to know information such as when validations, conversions, and events are usually handled and ways to change how and when they are handled. For more information on each of the lifecycle phases, download the latest Jakarta Faces Specification documentation from https://jakarta.ee/specifications/faces/.

The Jakarta Faces application lifecycle Execute phase contains the following subphases:

- Restore View Phase
- Apply Request Values Phase
- Process Validations Phase
- Update Model Values Phase
- Invoke Application Phase
- Render Response Phase

Restore View Phase

When a request for a Jakarta Faces page is made, usually by an action, such as when a link or a button component is clicked, the Jakarta Faces implementation begins the Restore View phase.

During this phase, the Jakarta Faces implementation builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the `FacesContext` instance, which contains all the information needed to process a single request. All the application’s components, event handlers, converters, and validators have access to the `FacesContext` instance.

If the request for the page is an initial request, the Jakarta Faces implementation creates an empty view during this phase and the lifecycle advances to the Render Response phase, during which the empty view is populated with the components referenced by the tags in the page.

If the request for the page is a postback, a view corresponding to this page already exists in the

`FacesContext` instance. During this phase, the Jakarta Faces implementation restores the view by using the state information saved on the client or the server.

Apply Request Values Phase

After the component tree is restored during a postback request, each component in the tree extracts its new value from the request parameters by using its `decode` (`processDecodes()`) method. The value is then stored locally on each component.

If any `decode` methods or event listeners have called the `renderResponse` method on the current `FacesContext` instance, the Jakarta Faces implementation skips to the Render Response phase.

If any events have been queued during this phase, the Jakarta Faces implementation broadcasts the events to interested listeners.

If some components on the page have their `immediate` attributes (see The immediate Attribute) set to `true`, then the validations, conversions, and events associated with these components will be processed during this phase. If any conversion fails, an error message associated with the component is generated and queued on `FacesContext`. This message will be displayed during the Render Response phase, along with any validation errors resulting from the Process Validations phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any Jakarta Faces components, it can call the `FacesContext.responseComplete` method.

At the end of this phase, the components are set to their new values, and messages and events have been queued.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

Process Validations Phase

During this phase, the Jakarta Faces implementation processes all validators registered on the components in the tree by using its `validate` (`processValidators`) method. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component. The Jakarta Faces implementation also completes conversions for input components that do not have the `immediate` attribute set to true.

If the local value is invalid, or if any conversion fails, the Jakarta Faces implementation adds an error message to the `FacesContext` instance, and the lifecycle advances directly to the Render Response phase so that the page is rendered again with the error messages displayed. If there were conversion errors from the Apply Request Values phase, the messages for these errors are also displayed.

If any `validate` methods or event listeners have called the `renderResponse` method on the current `FacesContext`, the Jakarta Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any Jakarta Faces components, it can call the

`FacesContext.responseComplete` method.

If events have been queued during this phase, the Jakarta Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

Update Model Values Phase

After the Jakarta Faces implementation determines that the data is valid, it traverses the component tree and sets the corresponding server-side object properties to the components' local values. The Jakarta Faces implementation updates only the bean properties pointed at by an input component's `value` attribute. If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to the Render Response phase so that the page is re-rendered with errors displayed. This is similar to what happens with validation errors.

If any `updateModels` methods or any listeners have called the `renderResponse` method on the current `FacesContext` instance, the Jakarta Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any Jakarta Faces components, it can call the `FacesContext.responseComplete` method.

If any events have been queued during this phase, the Jakarta Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

Invoke Application Phase

During this phase, the Jakarta Faces implementation handles any application-level events, such as submitting a form or linking to another page.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any Jakarta Faces components, it can call the `FacesContext.responseComplete` method.

If the view being processed was reconstructed from state information from a previous request and if a component has fired an event, these events are broadcast to interested listeners.

Finally, the Jakarta Faces implementation transfers control to the Render Response phase.

Render Response Phase

During this phase, Jakarta Faces builds the view and delegates authority to the appropriate resource for rendering the pages.

If this is an initial request, the components that are represented on the page will be added to the component tree. If this is not an initial request, the components are already added to the tree and

need not be added again.

If the request is a postback and errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered again during this phase. If the pages contain `h:message` or `h:messages` tags, any queued error messages are displayed on the page.

After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it. The saved state is available to the Restore View phase.

Partial Processing and Partial Rendering

The Jakarta Faces lifecycle spans all of the execute and render processes of an application. It is also possible to process and render only parts of an application, such as a single component. For example, the Jakarta Faces Ajax framework can generate requests containing information on which particular component may be processed and which particular component may be rendered back to the client.

Once such a partial request enters the Jakarta Faces lifecycle, the information is identified and processed by a `jakarta.faces.context.PartialViewContext` object. The Jakarta Faces lifecycle is still aware of such Ajax requests and modifies the component tree accordingly.

The `execute` and `render` attributes of the `f:ajax` tag are used to identify which components may be executed and rendered. For more information on these attributes, see [web:faces-ajax::faces-ajax:::_using_ajax_with_jakarta_faces_technology].

Further Information about Jakarta Faces Technology

For more information on Jakarta Faces technology, see

- Jakarta Faces 4.0 specification:
  https://jakarta.ee/specifications/faces/4.0/
- Mojarra website:
  https://eclipse-ee4j.github.io/mojarra/

For additional samples, see the GlassFish samples at https://github.com/eclipse-ee4j/glassfish-samples/tree/master/ws/jakartaee9.

Introduction to Facelets



This section was last updated for Jakarta EE Version 9.1

The term Facelets refers to the view declaration language for Jakarta Faces technology. Facelets is a part of the Jakarta Faces specification and also the preferred presentation technology for building Jakarta Faces technology–based applications. Jakarta Server Pages technology, previously used as the presentation technology for Jakarta Faces, does not support all the new features available in Jakarta Faces in the Jakarta EE platform. Jakarta Server Pages technology is considered to be a deprecated presentation technology for Jakarta Faces.

What Is Facelets?

Facelets is a powerful but lightweight page declaration language that is used to build Jakarta Faces views using HTML style templates and to build component trees. Facelets features include the following:

- Use of XHTML for creating web pages
- Support for Facelets tag libraries in addition to Jakarta Faces and JSTL tag libraries
- Support for the Expression Language (EL)
- Templating for components and pages

The advantages of Facelets for large-scale development projects include the following:

- Support for code reuse through templating and composite components
- Functional extensibility of components and other server-side objects through customization
- Faster compilation time
- Compile-time EL validation
- High-performance rendering

In short, the use of Facelets reduces the time and effort that needs to be spent on development and deployment.

Facelets views are usually created as XHTML pages. Jakarta Faces implementations support XHTML pages created in conformance with the XHTML Transitional Document Type Definition (DTD), as listed at https://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional. By convention, web pages built with XHTML have an `.xhtml` extension.

Jakarta Faces technology supports various tag libraries to add components to a web page. To support the Jakarta Faces tag library mechanism, Facelets uses XML namespace declarations. Tag Libraries Supported by Facelets lists the tag libraries supported by Facelets.

Tag Libraries Supported by Facelets

Tag Library	URI	Prefix	Example	Contents

Jakarta Faces Facelets Tag Library	jakarta.faces.facelets	`ui:`	`ui:component`
			
			`ui:insert`
				Tags for templating

Jakarta Faces HTML Tag Library	jakarta.faces.html	`h:`	`h:head`
			
			`h:body`
			
			`h:outputText`
			
			`h:inputText`
				Jakarta Faces component tags for all `UIComponent` objects

Tag Library	URI	Prefix	Example	Contents
Jakarta Faces Core Tag Library	jakarta.faces.core	`f:`	`f:actionListener` `f:attribute`	Tags for Jakarta Faces custom actions that are independent of any particular render kit
Pass-through Elements Tag Library	jakarta.faces	`faces:`	`faces:id`	Tags to support HTML5-friendly markup
Pass-through Attributes Tag Library	jakarta.faces.passthrough	`p:`	`p:type`	Tags to support HTML5-friendly markup
Composite Component Tag Library	jakarta.faces.composite	`cc:`	`cc:interface`	Tags to support composite components
JSTL Core Tag Library	jakarta.tags.core	`c:`	`c:forEach` `c:catch`	JSTL 1.2 Core Tags
JSTL Functions Tag Library	jakarta.tags.functions	`fn:`	`fn:toUpperCase` `fn:toLowerCase`	JSTL 1.2 Functions Tags

Facelets provides two namespaces to support HTML5-friendly markup. For details, see HTML5-Friendly Markup.

Facelets supports tags for composite components, for which you can declare custom prefixes. For more information on composite components, see Composite Components.

The namespace prefixes shown in the table are conventional, not mandatory. As is always the case when you declare an XML namespace, you can specify any prefix in your Facelets page. For example, you can declare the prefix for the composite component tag library as

```
xmlns:composite="jakarta.faces.composite"
```

instead of as

```
xmlns:cc="jakarta.faces.composite"
```

Based on the Jakarta Faces support for Expression Language (EL) syntax, Facelets uses EL expressions to reference properties and methods of managed beans. EL expressions can be used to bind component objects or values to methods or properties of managed beans that are used as backing beans. For more information on using EL expressions, see Using the EL to Reference

Managed Beans.

The Lifecycle of a Facelets Application

The Jakarta Faces specification defines the lifecycle of a Jakarta Faces application. For more information on this lifecycle, see The Lifecycle of a Jakarta Faces Application. The following steps describe that process as applied to a Facelets-based application.

1. When a client, such as a browser, makes a new request to a page that is created using Facelets, a new component tree or `jakarta.faces.component.UIViewRoot` is created and placed in the `FacesContext`.
2. The `UIViewRoot` is applied to the Facelets, and the view is populated with components for rendering.
3. The newly built view is rendered back as a response to the client.
4. On rendering, the state of this view is stored for the next request. The state of input components and form data is stored.
5. The client may interact with the view and request another view or change from the Jakarta Faces application. At this time, the saved view is restored from the stored state.
6. The restored view is once again passed through the Jakarta Faces lifecycle, which eventually will either generate a new view or re-render the current view if there were no validation problems and no action was triggered.
7. If the same view is requested, the stored view is rendered once again.
8. If a new view is requested, then the process described in Step 2 is continued.
9. The new view is then rendered back as a response to the client.

Developing a Simple Facelets Application: The guessnumber-faces Example Application

This section describes the general steps involved in developing a Jakarta Faces application. The following tasks are usually required:

- Developing the managed beans
- Creating the pages using the component tags
- Defining page navigation
- Mapping the `FacesServlet` instance
- Adding managed bean declarations

Creating a Facelets Application

The example used in this tutorial is the `guessnumber-faces` application. The application presents you with a page that asks you to guess a number from 0 to 10, validates your input against a random number, and responds with another page that informs you whether you guessed the number correctly or incorrectly.

The source code for this application is in the `jakartaee-examples/tutorial/web/faces/guessnumber-faces/` directory.

Developing a Managed Bean

In a typical Jakarta Faces application, each page of the application connects to a managed bean that serves as a backing bean. The backing bean defines the methods and properties that are associated with the components. In this example, both pages use the same backing bean.

The following managed bean class, `UserNumberBean.java`, generates a random number from 0 to 10 inclusive:

```
package ee.jakarta.tutorial.guessnumber;

import java.io.Serializable;
import java.util.Random;

import jakarta.annotation.PostConstruct;
import jakarta.faces.view.ViewScoped;
import jakarta.inject.Named;

@Named
@ViewScoped
public class UserNumberBean implements Serializable {

    private static final long serialVersionUID = 1L;

    private static final int MINIMUM = 0;
    private static final int MAXIMUM = 10;

    private int randomInt;
    private Integer userNumber;
    private String response;

    @PostConstruct
    public void init() {
        randomInt = new Random().nextInt(maximum + 1);
        // Print number to server log.
        System.out.println("Duke's number: " + randomInt);
    }

    public void guess() {
        if (userNumber.compareTo(randomInt) != 0) {
            response = "Sorry, " + userNumber + " is incorrect.";
        }
        else {
            response = "Yay! You got it!";
        }
    }

    public Integer getUserNumber() {
        return userNumber;
    }
```

```
    public void setUserNumber(Integer userNumber) {
        this.userNumber = userNumber;
    }

    public String getResponse() {
        return response;
    }

    public int getMinimum() {
        return MINIMUM;
    }

    public int getMaximum() {
        return MAXIMUM;
    }
}
```

Note the use of the `@Named` annotation, which makes the managed bean accessible through the EL. The `@ViewScoped` annotation registers the bean scope as `view` to enable you to reuse the same bean instance as long as you interact with the same web page without navigating away.

Creating Facelets Views

To create a page or view, you add components to the pages, wire the components to backing bean values and properties, and register converters, validators, or listeners on the components.

For the example application, XHTML web pages serve as the front end. The page of the example application is a page called `greeting.xhtml`. A closer look at various sections of this web page provides more information.

The first section of the web page declares the document type of the generated HTML output, which is HTML5:

```
<!DOCTYPE html>
```

The next section specifies the language of the text embedded in the generated HTML output and then declares the XML namespace for the tag libraries that are used in the web page:

```
<html lang="en"
      xmlns:h="jakarta.faces.html"
      xmlns:f="jakarta.faces.core">
```

The next section uses various tags to insert components into the web page:

```
    <h:head>
        <title>Guess Number Facelets Application</title>
        <h:outputStylesheet name="css/default.css"/>
```

```
    </h:head>
    <h:body>
        <h:form>
            <h:graphicImage name="images/wave.svg"
                            alt="Duke waving his hand"
                            width="100" height="100" />
            <h1>
                Hi, my name is Duke. I am thinking of a number from
                #{userNumberBean.minimum} to #{userNumberBean.maximum}.
                Can you guess it?
            </h1>
            <div class="input">
                <h:outputLabel id="userNumberLabel" for="userNumber"
                               value="Enter a number from 0 to 10:" />
                <h:inputText id="userNumber" required="true"
                             value="#{userNumberBean.userNumber}">
                    <f:validateLongRange minimum="#{userNumberBean.minimum}"
                                         maximum="#{userNumberBean.maximum}" />
                </h:inputText>
                <h:message id="userNumberMessage" for="userNumber" />
            </div>
            <div class="actions">
                <h:commandButton id="guess" value="Guess"
                                 action="#{userNumberBean.guess}">
                    <f:ajax execute="@form"
                            render="userNumberMessage response" />
                </h:commandButton>
            </div>
            <div class="output">
                <h:outputText id="response" value="#{userNumberBean.response}" />
            </div>
        </h:form>
    </h:body>
```

Note the use of the following tags:

- Facelets HTML tags (those beginning with `h:`) to add components
- The Facelets core tag `f:validateLongRange` to validate the user input

An `h:inputText` tag accepts user input and sets the value of the managed bean property `userNumber` through the EL expression `#{userNumberBean.userNumber}`. The input value is validated for value range by the Jakarta Faces standard validator tag `f:validateLongRange`.

The image file, `wave.svg`, is added to the page as a resource, as is the style sheet. For more details about the resources facility, see Web Resources.

An `h:commandButton` tag with the ID `guess` starts validation of the input data when a user clicks the button. Using `f:ajax`, the tag updates the `h:message` associated with the `h:inputText` so that it can display any validation error messages. Also the `h:outputText` is being updated which shows the response to your input.

Configuring the Application

Configuring a Jakarta Faces application involves mapping the Faces Servlet in the web deployment descriptor file, such as a `web.xml` file, and possibly adding managed bean declarations, navigation rules, and resource bundle declarations to the application configuration resource file, `faces-config.xml`.

If you are using NetBeans IDE, a web deployment descriptor file is automatically created for you. In such an IDE-created `web.xml` file, change the default greeting page, which is `index.xhtml`, to `greeting.xhtml`. Here is an example `web.xml` file, showing this change in bold.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">
    <context-param>
        <param-name>jakarta.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>greeting.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Note the use of the context parameter `PROJECT_STAGE`. This parameter identifies the status of a Jakarta Faces application in the software lifecycle.

The stage of an application can affect the behavior of the application. For example, if the project stage is defined as `Development`, debugging information is automatically generated for the user. If not defined by the user, the default project stage is `Production`.

Running the guessnumber-faces Facelets Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `guessnumber-faces` example.

To Build, Package, and Deploy the guessnumber-faces Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).

2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

```
jakartaee-examples/tutorial/web/faces
```

4. Select the `guessnumber-faces` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `guessnumber-faces` project and select **Build**.

   This option builds the example application and deploys it to your GlassFish Server instance.

To Build, Package, and Deploy the guessnumber-faces Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

```
jakartaee-examples/tutorial/web/faces/guessnumber-faces/
```

3. Enter the following command:

```
mvn install
```

   This command builds and packages the application into a WAR file, `guessnumber-faces.war`, that is located in the `target` directory. It then deploys it to the server.

To Run the guessnumber-faces Example

1. Open a web browser.
2. Enter the following URL in your web browser:

```
http://localhost:8080/guessnumber-faces
```

3. In the field, enter a number from 0 to 10 and click Submit.

   Another page appears, reporting whether your guess is correct or incorrect.

4. If you guessed incorrectly, click Back to return to the main page.

   You can continue to guess until you get the correct answer, or you can look in the server log, where the `UserNumberBean` constructor displays the correct answer.

Using Facelets Templates

Jakarta Faces technology provides the tools to implement user interfaces that are easy to extend and reuse. Templating is a useful Facelets feature that allows you to create a page that will act as

the base, or template, for the other pages in an application. By using templates, you can reuse code and avoid recreating similarly constructed pages. Templating also helps in maintaining a standard look and feel in an application with a large number of pages.

Facelets Templating Tags lists Facelets tags that are used for templating and their respective functionality.

Facelets Templating Tags

Tag	Function
`ui:component`	Defines a component that is created and added to the component tree.
`ui:composition`	Defines a page composition that optionally uses a template. Content outside of this tag is ignored.
`ui:debug`	Defines a debug component that is created and added to the component tree.
`ui:decorate`	Similar to the composition tag but does not disregard content outside this tag.
`ui:define`	Defines content that is inserted into a page by a template.
`ui:fragment`	Similar to the component tag but does not disregard content outside this tag.
`ui:include`	Encapsulates and reuses content for multiple pages.
`ui:insert`	Inserts content into a template.
`ui:param`	Used to pass parameters to an included file.
`ui:repeat`	Used as an alternative for loop tags, such as `c:forEach` or `h:dataTable.`
`ui:remove`	Removes content from a page.

For more information on Facelets templating tags, see the Jakarta Faces Facelets Tag Library documentation.

The Facelets tag library includes the main templating tag `ui:insert`. A template page that is created with this tag allows you to define a default structure for a page. A template page is used as a template for other pages, usually referred to as client pages.

Here is an example of a template saved as `/WEB-INF/templates/template.xhtml`:

```
<!DOCTYPE html>
<html xmlns:ui="jakarta.faces.facelets"
      xmlns:h="jakarta.faces.html">

    <h:head>
        <title>Facelets Template</title>
        <h:outputStylesheet name="css/default.css"/>
        <h:outputStylesheet name="css/layout.css"/>
    </h:head>

    <h:body>
        <div id="top">
            <ui:insert name="top">Top Section</ui:insert>
```

```
        </div>
        <div>
            <div id="left">
                <ui:insert name="left">Left Section</ui:insert>
            </div>
            <div id="content">
                <ui:insert name="content">Main Content</ui:insert>
            </div>
        </div>
    </h:body>
</html>
```

The example page defines an XHTML page that is divided into three sections: a top section, a left section, and a main section. The sections have style sheets associated with them. The same structure can be reused for the other pages of the application.

The client page invokes the template by using the `ui:composition` tag. In the following example, a client page named `templateclient.xhtml` invokes the template page named `template.xhtml` from the preceding example. A client page allows content to be inserted with the help of the `ui:define` tag.

```
<ui:composition template="/WEB-INF/templates/template.xhtml">
                xmlns:ui="jakarta.faces.facelets"
                xmlns:h="jakarta.faces.html">
    <ui:define name="top">
        <h1>Welcome to Template Client Page</h1>
    </ui:define>

    <ui:define name="left">
        <p>You are in the Left Section.</p>
    </ui:define>

    <ui:define name="content">
        <header>
            <h:graphicImage name="images/wave.svg"
                            alt="Duke waving his hand"
                            width="100" height="100" />
        </header>
        <p>You are in the Main Content Section.</p>
    </ui:define>
</ui:composition>
```

You can use NetBeans IDE to create Facelets template and client pages. For more information on creating these pages, see https://netbeans.org/kb/docs/web/jsf20-intro.html.

Composite Components

Jakarta Faces technology offers the concept of composite components with Facelets. A composite component is a special type of template that acts as a component.

Any component is essentially a piece of reusable code that behaves in a particular way. For example, an input component accepts user input. A component can also have validators, converters, and listeners attached to it to perform certain defined actions.

A composite component consists of a collection of markup tags and other existing components. This reusable, user-created component has a customized, defined functionality and can have validators, converters, and listeners attached to it like any other component.

With Facelets, any XHTML page that contains markup tags and other components can be converted into a composite component. Using the resources facility, the composite component can be stored in a library that is available to the application from the defined resources location.

Composite Component Tags lists the most commonly used composite tags and their functions.

Composite Component Tags

Tag	Function
`composite:interface`	Declares the usage contract for a composite component. The composite component can be used as a single component whose feature set is the union of the features declared in the usage contract.
`composite:implementation`	Defines the implementation of the composite component. If a `composite:interface` element appears, there must be a corresponding `composite:implementation`.
`composite:attribute`	Declares an attribute that may be given to an instance of the composite component in which this tag is declared.
`composite:insertChildren`	Any child components or template text within the composite component tag in the using page will be reparented into the composite component at the point indicated by this tag's placement within the `composite:implementation` section.
`composite:valueHolder`	Declares that the composite component whose contract is declared by the `composite:interface` in which this element is nested exposes an implementation of `ValueHolder` suitable for use as the target of attached objects in the using page.
`composite:editableValueHolder`	Declares that the composite component whose contract is declared by the `composite:interface` in which this element is nested exposes an implementation of `EditableValueHolder` suitable for use as the target of attached objects in the using page.
`composite:actionSource`	Declares that the composite component whose contract is declared by the `composite:interface` in which this element is nested exposes an implementation of `ActionSource2` suitable for use as the target of attached objects in the using page.

For more information and a complete list of Facelets composite tags, see the Jakarta Faces Facelets Tag Library documentation.

The following example shows a composite component that renders an form field with a label, input and message component:

```
<ui:component xmlns:ui="jakarta.faces.facelets"
              xmlns:cc="jakarta.faces.composite"
              xmlns:h="jakarta.faces.html">
    <cc:interface>
        <cc:attribute name="label" required="true" />
        <cc:attribute name="value" required="true" />
    </cc:interface>

    <cc:implementation>
        <div id="#{cc.clientId}" class="field">
            <h:outputLabel for="input" value="#{cc.attrs.label}" />
            <h:inputText id="input" value="#{cc.attrs.value}" />
            <h:message for="input" />
        </div>
    </cc:implementation>
</ui:component>
```

Note the use of `cc.attrs.value` when defining the value of the `outputLabel` and `inputText` components. The word `cc` in Jakarta Faces is a reserved word for composite components. The `#{cc.attrs.attribute-name}` expression is used to access the attributes defined for the composite component's interface, which in this case happens to be `value`.

The preceding example content is stored as a file named `field.xhtml` in a folder named `resources/mycomponents`, under the application web root directory. This directory is considered a library by Jakarta Faces, and a component can be accessed from such a library. The `mycomponents` folder name is free to your choice. For more information on resources, see Web Resources.

The web page that uses this composite component is generally called a using page. The using page includes a reference to the composite component, in the `xml` namespace declarations:

```
<!DOCTYPE html>
<html xmlns:h="jakarta.faces.html"
      xmlns:my="jakarta.faces.composite/mycomponents">

    <h:head>
        <title>Using a sample composite component</title>
    </h:head>

    <h:body>
        <h:form>
            <my:field id="email"
                      label="Enter your email address"
                      value="#{bean.email}" />
        </h:form>
    </h:body>
</html>
```

The local composite component library is defined in the `xmlns` namespace with the declaration

`xmlns:my="jakarta.faces.composite/mycomponents"`. The `my` XML namespace are free to your choice. The `/mycomponents` part must represent the folder name where the composite component files are located. The component itself is accessed through the `my:field` tag. The preceding example content can be stored as a web page named `userpage.xhtml` under the web root directory. When compiled and deployed on a server, it can be accessed with the following URL:

```
http://localhost:8080/application-name/userpage.xhtml
```

See [web:faces-advanced-cc::faces-advanced-cc:::_composite_components_advanced_topics_and_an_example] for more information and an example.

Web Resources

Web resources are any software artifacts that the web application requires for proper rendering, including images, script files, and any user-created component libraries. Resources must be collected in a standard location, which can be one of the following.

- A resource packaged in the web application root must be in a subdirectory of a `resources` directory at the web application root: `resources/resource-identifier`.
- A resource packaged in the web application's classpath must be in a subdirectory of the `META-INF/resources` directory within a web application: `META-INF/resources/resource-identifier`. You can use this file structure to package resources in a JAR file bundled in the web application.

The Jakarta Faces runtime will look for the resources in the preceding listed locations, in that order.

Resource identifiers are unique strings that conform to the following format (all on one line):

```
[locale-prefix/][library-name/][library-version/]resource-name[/resource-version]
```

Elements of the resource identifier in brackets (`[]`) are optional, indicating that only a resource-name, which is usually a file name, is a required element. For example, the most common way to specify a style sheet, image, or script is to use the `library` and `name` attributes, as in the following tag from the `guessnumber-faces` example:s

```
<h:outputStylesheet library="css" name="default.css"/>
```

This tag specifies that the `default.css` style sheet is in the directory `web/resources/css`.

You can also specify the location of an image using the following syntax, also from the `guessnumber-faces` example:

```
<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
```

This tag specifies that the image named `wave.med.gif` is in the directory `web/resources/images`.

Resources can be considered as a library location. Any artifact, such as a composite component or a template that is stored in the `resources` directory, becomes accessible to the other application components, which can use it to create a resource instance.

Relocatable Resources

You can place a resource tag in one part of a page and specify that it be rendered in another part of the page. To do this, you use the `target` attribute of a tag that specifies a resource. Acceptable values for this attribute are as follows.

- "head" renders the resource in the `head` element.
- "body" renders the resource in the `body` element.
- "form" renders the resource in the `form` element.

For example, the following `h:outputScript` tag is placed within an `h:form` element, but it renders the JavaScript in the `head` element:

```
<h:form>
    <h:outputScript name="myscript.js" library="mylibrary" target="head"/>
</h:form>
```

The `h:outputStylesheet` tag also supports resource relocation, in a similar way.

Relocatable resources are essential for composite components that use stylesheets and can also be useful for composite components that use JavaScript. See The compositecomponentexample Example Application for an example.

Resource Library Contracts

Resource library contracts allow you to define a different look and feel for different parts of one or more applications, instead of either having to use the same look and feel for all or having to specify a different look on a page-by-page basis.

To do this, you create a `contracts` section of your web application. Within the `contracts` section, you can specify any number of named areas, each of which is called a contract. Within each contract you can specify resources such as template files, stylesheets, JavaScript files, and images.

For example, you could specify two contracts named `c1` and `c2`, each of which uses a template and other files:

```
src/main/webapp
    WEB-INF/
    contracts
        c1
            template.xhtml
            style.css
            myImg.gif
            myJS.js
```

```
        c2
            template.xhtml
            style2.css
            img2.gif
            JS2.js
    index.xhtml
    ...
```

One part of the application can use `c1`, while another can use `c2`.

Another way to use contracts is to specify a single contract that contains multiple templates:

```
src/main/webapp
    contracts
        myContract
            template1.xhtml
            template2.xhtml
            style.css
            img.png
            img2.png
```

You can package a resource library contract in a JAR file for reuse in different applications. If you do so, the contracts must be located under `META-INF/contracts`. You can then place the JAR file in the `WEB-INF/lib` directory of an application. This means that the application would be organized as follows:

```
src/main/webapp/
    WEB-INF/lib/myContract.jar
    ...
```

You can specify the contract usage within an application's `faces-config.xml` file, under the `resource-library-contracts` element. You need to use this element only if your application uses more than one contract, however.

The hello1-rlc Example Application

The `hello1-rlc` example modifies the simple `hello1` example from A Web Module That Uses Jakarta Faces Technology: The hello1 Example to use two resource library contracts. Each of the two pages in the application uses a different contract.

The managed bean for `hello1-rlc`, `Hello.java`, is identical to the one for `hello1` (except that it replaces the `@Named` and `@RequestScoped` annotations with `@Model`).

The source code for this application is in the `jakartaee-examples/tutorial/web/faces/hello1-rlc/` directory.

Configuring the hello1-rlc Example

The `faces-config.xml` file for the `hello1-rlc` example contains the following elements:

```
<resource-library-contracts>
    <contract-mapping>
        <url-pattern>/reply/*</url-pattern>
        <contracts>reply</contracts>
    </contract-mapping>
    <contract-mapping>
        <url-pattern>*</url-pattern>
        <contracts>hello</contracts>
    </contract-mapping>
</resource-library-contracts>
```

The `contract-mapping` elements within the `resource-library-contracts` element map each contract to a different set of pages within the application. One contract, named `reply`, is used for all pages under the `reply` area of the application (`/reply/*`). The other contract, `hello`, is used for all other pages in the application (`*`).

The application is organized as follows:

```
hello1-rlc
    pom.xml
    src/main/java/jakarta/tutorial/hello1rlc/Hello.java
    src/main/webapp
        WEB-INF
            faces-config.xml
            web.xml
        contracts
            hello
                default.css
                duke.handsOnHips.gif
                template.xhtml
            reply
                default.css
                duke.thumbsup.gif
                template.xhtml
        reply
            response.xhtml
        greeting.xhtml
```

The `web.xml` file specifies the `welcome-file` as `greeting.xhtml`. Because it is not located under `src/main/webapp/reply`, this Facelets page uses the `hello` contract, whereas `src/main/webapp/reply/response.xhtml` uses the `reply` contract.

The Facelets Pages for the hello1-rlc Example

The `greeting.xhtml` and `response.xhtml` pages have identical code calling in their templates:

```
<ui:composition template="/template.xhtml">
```

The `template.xhtml` files in the `hello` and `reply` contracts differ only in two respects: the placeholder text for the `title` element ("Hello Template" and "Reply Template") and the graphic that each specifies.

The `default.css` stylesheets in the two contracts differ in only one respect: the background color specified for the `body` element.

To Build, Package, and Deploy the hello1-rlc Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `hello1-rlc` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `hello1-rlc` project and select **Build**.

   This option builds the example application and deploys it to your GlassFish Server instance.

To Build, Package, and Deploy the hello1-rlc Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/hello1-rlc/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `hello1-rlc.war`, that is located in the `target` directory. It then deploys it to your GlassFish Server instance.

To Run the hello1-rlc Example

1. Enter the following URL in your web browser:

   ```
   http://localhost:8080/hello1-rlc
   ```

2. The `greeting.xhtml` page looks just like the one from `hello1` except for its background color and graphic.
3. In the text field, enter your name and click Submit.
4. The response page also looks just like the one from `hello1` except for its background color and graphic.

   The page displays the name you submitted. Click Back to return to the `greeting.xhtml` page.

HTML5-Friendly Markup

When you want to produce user interface features for which HTML does not have its own elements, you can create a custom Jakarta Faces component and insert it in your Facelets page. This mechanism can cause a simple element to create complex web code. However, creating such a component is a significant task (see [web:faces-custom::faces-custom:::_creating_custom_ui_components_and_other_custom_objects]).

HTML5 offers new elements and attributes that can make it unnecessary to write your own components. It also provides many new capabilities for existing components. Jakarta Faces technology supports HTML5 not by introducing new UI components that imitate HTML5 ones but by allowing you to use HTML5 markup directly. It also allows you to use Jakarta Faces attributes within HTML5 elements. Jakarta Faces technology support for HTML5 falls into two categories:

- Pass-through elements
- Pass-through attributes

The effect of the HTML5-friendly markup feature is to offer the Facelets page author almost complete control over the rendered page output, rather than having to pass this control off to component authors. You can mix and match Jakarta Faces and HTML5 components and elements as you see fit.

Using Pass-Through Elements

Pass-through elements allow you to use HTML5 tags and attributes but to treat them as equivalent to Jakarta Faces components associated with a server-side `UIComponent` instance.

To make an element that is not a Jakarta Faces element a pass-through element, specify at least one of its attributes using the `jakarta.faces` namespace. For example, the following code declares the namespace with the short name `faces`:

```
<html ... xmlns:faces="jakarta.faces"
...
    <input type="email" faces:id="email" name="email"
           value="#{reservationBean.email}" required="required"/>
```

Here, the `faces` prefix is placed on the `id` attribute so that the HTML5 input tag's attributes are treated as part of the Facelets page. This means that, for example, you can use EL expressions to retrieve managed bean properties.

How Facelets Renders HTML5 Elements shows how pass-through elements are rendered as Facelets tags. The faces implementation uses the element name and the identifying attribute to determine the corresponding Facelets tag that will be used in the server-side processing. The browser, however, interprets the markup that the page author has written.

How Facelets Renders HTML5 Elements

HTML5 Element Name	Identifying Attribute	Facelets Tag
a	faces:action	h:commandLink
a	faces:actionListener	h:commandLink
a	faces:value	h:outputLink
a	faces:outcome	h:link
body		h:body
button		h:commandButton
button	faces:outcome	h:button
form		h:form
head		h:head
img		h:graphicImage
input	type="button"	h:commandButton
input	type="checkbox"	h:selectBooleanCheckbox
input	type="color"	h:inputText
input	type="date"	h:inputText
input	type="datetime"	h:inputText
input	type="datetime-local"	h:inputText
input	type="email"	h:inputText
input	type="month"	h:inputText
input	type="number"	h:inputText
input	type="range"	h:inputText
input	type="search"	h:inputText
input	type="time"	h:inputText
input	type="url"	h:inputText
input	type="week"	h:inputText
input	type="file"	h:inputFile
input	type="hidden"	h:inputHidden
input	type="password"	h:inputSecret
input	type="reset"	h:commandButton
input	type="submit"	h:commandButton
input	type="*"	h:inputText

HTML5 Element Name	Identifying Attribute	Facelets Tag
label		h:outputLabel
link		h:outputStylesheet
script		h:outputScript
select	multiple="*"	h:selectManyListbox
select		h:selectOneListbox
textarea		h:inputTextArea

Using Pass-Through Attributes

Pass-through attributes are the converse of pass-through elements. They allow you to pass attributes that are not Jakarta Faces attributes through to the browser without interpretation. If you specify a pass-through attribute in a Jakarta Faces `UIComponent`, the attribute name and value are passed straight through to the browser without being interpreted by Jakarta Faces components or renderers. There are several ways to specify pass-through attributes.

- Use the Jakarta Faces namespace for pass-through attributes to prefix the attribute names within a Jakarta Faces component. For example, the following code declares the namespace with the short name `p`, then passes the `type`, `min`, `max`, `required`, and `title` attributes through to the HTML5 `input` component:

  ```
  <html ... xmlns:p="jakarta.faces.passthrough"
  ...

  <h:form prependId="false">
  <h:inputText id="nights" p:type="number" value="#{bean.nights}"
               p:min="1" p:max="30" p:required="required"
               p:title="Enter a number between 1 and 30 inclusive.">
          ...
  ```

  This will cause the following markup to be rendered (assuming that `bean.nights` has a default value set to 1):

  ```
  <input id="nights" type="number" value="1" min="1" max="30"
         required="required"
         title="Enter a number between 1 and 30 inclusive.">
  ```

- To pass a single attribute, nest the `f:passThroughAttribute` tag within a component tag. For example:

  ```
  <h:inputText value="#{user.email}">
      <f:passThroughAttribute name="type" value="email" />
  </h:inputText>
  ```

This code would be rendered similarly to the following:

```
<input value="me@me.com" type="email" />
```

- To pass a group of attributes, nest the `f:passThroughAttributes` tag within a component tag, specifying an EL value that must evaluate to a `Map<String, Object>`. For example:

```
<h:inputText value="#{bean.nights}">
    <f:passThroughAttributes value="#{bean.nameValuePairs}" />
</h:inputText>
```

If the bean used the following `Map` declaration and initialized the map in the constructor as follows, the markup would be similar to the output of the code that uses the pass-through attribute namespace:

```
private Map<String, Object> nameValuePairs;
...
public Bean() {
    this.nameValuePairs = new HashMap<>();
    this.nameValuePairs.put("type", "number");
    this.nameValuePairs.put("min", "1");
    this.nameValuePairs.put("max", "30");
    this.nameValuePairs.put("required", "required");
    this.nameValuePairs.put("title",
            "Enter a number between 1 and 4 inclusive.");
}
```

The reservation Example Application

The `reservation` example application provides a set of HTML5 `input` elements of various types to simulate purchasing tickets for a theatrical event. It consists of two Facelets pages, `reservation.xhtml` and `confirmation.xhtml`, and a backing bean, `ReservationBean.java`. The pages use both pass-through attributes and pass-through elements.

The source code for this application is in the `jakartaee-examples/tutorial/web/faces/reservation/` directory.

The Facelets Pages for the reservation Application

The first important feature of the Facelets pages for the `reservation` application is the `DOCTYPE` header. The facelets pages for this application begin simply with the following `DOCTYPE` header, which indicates that the XHTML-generated result is an HTML5 page:

```
<!DOCTYPE html>
```

The namespace declarations in the `html` element of the `reservation.xhtml` page specify both the

`faces` and the `passthrough` namespaces:

```
<html lang="en"
      xmlns:faces="jakarta.faces"
      xmlns:f="jakarta.faces.core"
      xmlns:h="jakarta.faces.html"
      xmlns:p="jakarta.faces.passthrough">
```

Next, an `h:head` tag followed by an `h:outputStylesheet` tag within the `h:body` tag illustrates the use of a relocatable resource (as described in Relocatable Resources):

```
<h:head>
    <title>Reservation Application</title>
</h:head>
<h:body>
    <h:outputStylesheet name="css/stylesheet.css" target="head" />
```

The `reservation.xhtml` page uses a HTML5-specific `input` element type on `h:inputText`, which is `date`.

```
    <h:inputText id="date" type="date"
                 value="#{reservationBean.date}" required="true"
                 title="Enter or choose a date." />
```

The field for the number of tickets uses the `f:passThroughAttributes` tag to pass a `Map` defined in the managed bean. It also recalculates the total in response to a change in the field:

```
    <h:inputText id="tickets" value="#{reservationBean.tickets}">
        <f:passThroughAttributes value="#{reservationBean.ticketAttrs}"/>
        <f:ajax listener="#{reservationBean.calculateTotal}"
                render="total" />
    </h:inputText>
```

The field for the price specifies the `min`, `max` and `step` as a pass-through attribute of the `h:inputText` element, offering a range of four ticket prices. Here, the `p` prefix on the HTML5 attributes passes them through to the browser uninterpreted by the Jakarta Faces input component:

```
    <h:inputText id="price" type="number"
                 value="#{reservationBean.price}" required="true"
                 p:min="80" p:max="120" p:step="20"
                 title="Enter a price: 80, 100, 120, or 140.">
        <f:ajax listener="#{reservationBean.calculateTotal}"
                render="total" />
    </h:inputText>
```

The output of the `calculateTotal` method that is specified as the listener for the Ajax event is

rendered in the output element whose `id` and `name` value is `total`. See [web:faces-ajax::faces-ajax:::_using_ajax_with_jakarta_faces_technology], for more information.

The second Facelets page, `confirmation.xhtml`, uses a pass-through `output` element to display the values entered by the user and provides a Facelets `h:commandButton` tag to allow the user to return to the `reservation.xhtml` page.

The Managed Bean for the reservation Application

The session-scoped managed bean for the reservation application, `ReservationBean.java`, contains properties for all the elements on the Facelets pages. It also contains two methods, `calculateTotal` and `clear`, that act as listeners for Ajax events on the `reservation.xhtml` page.

To Build, Package, and Deploy the reservation Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `reservation` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `reservation` project and select **Build**.

   This option builds the example application and deploys it to your GlassFish Server instance.

To Build, Package, and Deploy the reservation Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/reservation/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `reservation.war`, that is located in the `target` directory. It then deploys the WAR file to your GlassFish Server instance.

To Run the reservation Example

At the time of the publication of this tutorial, the browser that most fully implements HTML5 is Google Chrome, and it is recommended that you use it to run this example. Other browsers are

catching up, however, and may work equally well by the time you read this.

1. Enter the following URL in your web browser:

   ```
   http://localhost:8080/reservation
   ```

2. Enter information in the fields of the `reservation.xhtml` page.

   The Performance Date field has a date field with up and down arrows that allow you to increment and decrement the month, day, and year as well as a larger down arrow that brings up a date editor in calendar form.

   The Number of Tickets and Ticket Price fields also have up and down arrows that allow you to increment and decrement the values within the allowed range and steps. The Estimated Total changes when you change either of these two fields.

   Email addresses and dates are checked for format, but not for validity (you can make a reservation for a past date, for instance).

3. Click Make Reservation to complete the reservation or Clear to restore the fields to their default values.
4. If you click Make Reservation, the `confirmation.xhtml` page appears, displaying the submitted values.

   Click Back to return to the `reservation.xhtml` page.

Using Jakarta Faces Technology in Web Pages



This section was last updated for Jakarta EE Version 9.1

Web pages (Facelets pages, in most cases) represent the presentation layer for web applications. The process of creating web pages for a Jakarta Faces application includes using component tags to add components to the page and wire them to backing beans, validators, listeners, converters, and other server-side objects that are associated with the page.

This chapter explains how to create web pages using various types of component and core tags. In the next chapter, you will learn about adding converters, validators, and listeners to component tags to provide additional functionality to components.

Many of the examples in this chapter are taken from Duke's Bookstore Case Study Example

Setting Up a Page

A typical Jakarta Faces web page includes the following elements:

- A set of namespace declarations that declare the Jakarta Faces tag libraries
- Optionally, the HTML head (`h:head`) and body (`h:body`) tags
- A form tag (`h:form`) that represents the user input components

To add the Jakarta Faces components to your web page, you need to provide the page access to the two standard tag libraries: the Jakarta Faces HTML render kit tag library and the Jakarta Faces core tag library. The Jakarta Faces standard HTML tag library defines tags that represent common HTML user interface components. The Jakarta Faces core tag library defines tags that perform core actions and are independent of a particular render kit.

For a complete list of Jakarta Faces Facelets tags and their attributes, refer to the Jakarta Faces Facelets Tag Library documentation.

To use any of the Jakarta Faces tags, you need to include appropriate directives at the top of each page specifying the tag libraries.

For Facelets applications, the XML namespace directives uniquely identify the tag library URI and the tag prefix.

For example, when you create a Facelets XHTML page, include namespace directives as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html"
      xmlns:f="jakarta.faces.core">
```

The XML namespace URI identifies the tag library location, and the prefix value is used to distinguish the tags belonging to that specific tag library. You can also use other prefixes instead of the standard `h` or `f`. However, when including the tag in the page you must use the prefix that you have chosen for the tag library. For example, in the following web page the `form` tag must be referenced using the `h` prefix because the preceding tag library directive uses the `h` prefix to distinguish the tags defined in the HTML tag library:

```
<h:form ...>
```

The sections Adding Components to a Page Using HTML Tag Library Tags and Using Core Tags describe how to use the component tags from the Jakarta Faces standard HTML tag library and the core tags from the Jakarta Faces core tag library.

Adding Components to a Page Using HTML Tag Library Tags

The tags defined by the Jakarta Faces standard HTML tag library represent HTML form components and other basic HTML elements. These components display data or accept data from the user. This data is collected as part of a form and is submitted to the server, usually when the user clicks a button. This section explains how to use each of the component tags shown in The Component Tags.

The Component Tags

Tag	Functions	Rendered As	Appearance
`h:column`	Represents a column of data in a data component	A column of data in an HTML table	A column in a table

Tag	Functions	Rendered As	Appearance
h:commandButton	Submits a form to the application	An HTML <input type=value> element for which the type value can be “submit”, “reset”, or “image”	A button
h:commandLink	Links to another page or location on a page	An HTML <a href> element	A link
h:dataTable	Represents a data wrapper	An HTML <table> element	A table that can be updated dynamically
h:form	Represents an input form (inner tags of the form receive the data that will be submitted with the form)	An HTML <form> element	No appearance
h:graphicImage	Displays an image	An HTML <img> element	An image
h:inputFile	Allows a user to upload a file	An HTML <input type="file"> element	A field with a Browse... button
h:inputHidden	Allows a page author to include a hidden variable in a page	An HTML <input type="hidden"> element	No appearance
h:inputSecret	Allows a user to input a string without the actual string appearing in the field	An HTML <input type="password"> element	A field that displays a row of characters instead of the actual string entered
h:inputText	Allows a user to input a string	An HTML <input type="text"> element	A field
h:inputTextarea	Allows a user to enter a multiline string	An HTML <textarea> element	A multirow field
h:message	Displays a localized message	An HTML tag if styles are used	A text string
h:messages	Displays localized messages	A set of HTML tags if styles are used	A text string
h:outputFormat	Displays a formatted message	Plain text	Plain text
h:outputLabel	Displays a nested component as a label for a specified input field	An HTML <label> element	Plain text
h:outputLink	Links to another page or location on a page without generating an action event	An HTML <a> element	A link
h:outputText	Displays a line of text	Plain text	Plain text

Tag	Functions	Rendered As	Appearance
`h:panelGrid`	Displays a table	An HTML `<table>` element with `<tr>` and `<td>` elements	A table
`h:panelGroup`	Groups a set of components under one parent	A HTML `<div>` or `<span>` element	A row in a table
`h:selectBooleanCheckbox`	Allows a user to change the value of a Boolean choice	An HTML `<input type="checkbox">` element	A check box
`h:selectManyCheckbox`	Displays a set of check boxes from which the user can select multiple values	A set of HTML `<input>` elements of type `checkbox`	A group of check boxes
`h:selectManyListbox`	Allows a user to select multiple items from a set of items all displayed at once	An HTML `<select>` element	A box
`h:selectManyMenu`	Allows a user to select multiple items from a set of items	An HTML `<select>` element	A menu
`h:selectOneListbox`	Allows a user to select one item from a set of items all displayed at once	An HTML `<select>` element	A box
`h:selectOneMenu`	Allows a user to select one item from a set of items	An HTML `<select>` element	A menu
`h:selectOneRadio`	Allows a user to select one item from a set of items	An HTML `<input type="radio">` element	A group of options For a standalone radio button, use the `group` attribute.

The tags correspond to components in the `jakarta.faces.component` package. The components are discussed in more detail in [web:faces-develop::faces-develop:::_developing_with_jakarta_faces_technology]

The next section explains the important attributes that are common to most component tags. For each of the components discussed in the following sections, Writing Bean Properties explains how to write a bean property bound to that particular component or its value.

For reference information about the tags and their attributes, see the Jakarta Faces Facelets Tag Library documentation.

Common Component Tag Attributes

Most of the component tags support the attributes shown in Common Component Tag Attributes.

Common Component Tag Attributes

Attribute	Description
`binding`	Identifies a bean property and binds the component instance to it.
`id`	Uniquely identifies the component.
`immediate`	If set to `true`, indicates that any events, validation, and conversion associated with the component should happen when request parameter values are applied.
`rendered`	Specifies a condition under which the component should be rendered. If the condition is not satisfied, the component is not rendered.
`style`	Specifies a Cascading Style Sheet (CSS) style for the tag.
`styleClass`	Specifies a CSS class that contains definitions of the styles.
`value`	Specifies the value of the component in the form of a value expression.

All the tag attributes except `id` can accept expressions, as defined by the Expression Language, described in Expression Language.

An attribute such as `rendered` or `value` can be set on the page and then modified in the backing bean for the page.

The id Attribute

The `id` attribute is not usually required for a component tag but is used when another component or a server-side class must refer to the component. If you don't include an `id` attribute, the Jakarta Faces implementation automatically generates a component ID. Unlike most other Jakarta Faces tag attributes, the `id` attribute takes expressions using only the evaluation syntax described in Immediate Evaluation, which uses the `${}` delimiters. For more information on expression syntax, see Value Expressions.

The immediate Attribute

Input components and command components (those that implement the `ActionSource` interface, such as buttons and links) can set the `immediate` attribute to `true` to force events, validations, and conversions to be processed when request parameter values are applied.

You need to carefully consider how the combination of an input component's `immediate` value and a command component's `immediate` value determines what happens when the command component is activated.

Suppose that you have a page with a button and a field for entering the quantity of a book in a shopping cart. If the `immediate` attributes of both the button and the field are set to `true`, the new value entered in the field will be available for any processing associated with the event that is generated when the button is clicked. The event associated with the button as well as the events, validation, and conversion associated with the field are all handled when request parameter values are applied.

If the button's `immediate` attribute is set to `true` but the field's `immediate` attribute is set to `false`, the event associated with the button is processed without updating the field's local value to the model layer. The reason is that any events, conversion, and validation associated with the field occur after request parameter values are applied.

The `bookshowcart.xhtml` page of the Duke's Bookstore case study has examples of components using the `immediate` attribute to control which component's data is updated when certain buttons are clicked. The `quantity` field for each book does not set the `immediate` attribute, so the value is `false` (the default).

```
<h:inputText id="quantity"
             size="4"
             value="#{item.quantity}"
             title="#{bundle.ItemQuantity}">
    <f:validateLongRange minimum="0"/>
    ...
</h:inputText>
```

The `immediate` attribute of the Continue Shopping hyperlink is set to `true`, while the `immediate` attribute of the Update Quantities hyperlink is set to `false`:

```
<h:commandLink id="continue"
               action="bookcatalog"
               immediate="true">
    <h:outputText value="#{bundle.ContinueShopping}"/>
</h:commandLink>
...
<h:commandLink id="update"
               action="#{showcart.update}"
               immediate="false">
    <h:outputText value="#{bundle.UpdateQuantities}"/>
</h:commandLink>
```

If you click the Continue Shopping hyperlink, none of the changes entered into the `quantity` input fields will be processed. If you click the Update Quantities hyperlink, the values in the `quantity` fields will be updated in the shopping cart.

The rendered Attribute

A component tag uses a Boolean EL expression along with the `rendered` attribute to determine whether the component will be rendered. For example, the `commandLink` component in the following section of a page is not rendered if the cart contains no items:

```
<h:commandLink id="check" ... rendered="#{cart.numberOfItems > 0}">
    <h:outputText value="#{bundle.CartCheck}"/>
</h:commandLink>
```

Unlike nearly every other Jakarta Faces tag attribute, the `rendered` attribute is restricted to using rvalue expressions. As explained in Value and Method Expressions, these rvalue expressions can only read data; they cannot write the data back to the data source. Therefore, expressions used with `rendered` attributes can use the arithmetic operators and literals that rvalue expressions can use but lvalue expressions cannot use. For example, the expression in the preceding example uses the `>` operator.



In this example and others, `bundle` refers to a `java.util.ResourceBundle` file that contains locale-specific strings to be displayed. Resource bundles are discussed in [web:webi18n::webi18n:::_internationalizing_and_localizing_web_applications].

The style and styleClass Attributes

The `style` and `styleClass` attributes allow you to specify CSS styles for the rendered output of your tags. Displaying Error Messages with the h:message and h:messages Tags describes an example of using the `style` attribute to specify styles directly in the attribute. A component tag can instead refer to a CSS class.

The following example shows the use of a `dataTable` tag that references the style class `list-background`:

```
<h:dataTable id="items"
             ...
             styleClass="list-background"
             value="#{cart.items}"
             var="book">
```

The style sheet that defines this class is `stylesheet.css`, which will be included in the application. For more information on defining styles, see the Cascading Style Sheets specifications and drafts at https://www.w3.org/Style/CSS/.

The value and binding Attributes

A tag representing an output component uses the `value` and `binding` attributes to bind its component's value or instance, respectively, to a data object. The `value` attribute is used more commonly than the `binding` attribute, and examples appear throughout this chapter. For more information on these attributes, see Creating a Managed Bean, Writing Properties Bound to Component Values, and Writing Properties Bound to Component Instances.

Adding HTML Head and Body Tags

The HTML head (`h:head`) and body (`h:body`) tags add HTML page structure to Jakarta Faces web pages.

- The `h:head` tag represents the head element of an HTML page.
- The `h:body` tag represents the body element of an HTML page.

The following is an example of an XHTML page using the usual head and body markup tags:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Add a title</title>
    </head>
    <body>
        <main>Add content</main>
    </body>
</html>
```

The following is an example of an XHTML page using `h:head` and `h:body` tags:

```
<!DOCTYPE html>
<html xmlns:h="jakarta.faces.html">
    <h:head>
        <title>Add a title</title>
    </h:head>
    <h:body>
        <main>Add content</main>
    </h:body>
</html>
```

Both of the preceding example code segments render the same HTML elements. The head and body tags are useful mainly for resource relocation. For more information on resource relocation, see Resource Relocation Using h:outputScript and h:outputStylesheet Tags.

Adding a Form Component

An `h:form` tag represents an input form, which includes child components that can contain data that is either presented to the user or submitted with the form.

Figure 13, “A Typical Form” shows a typical login form in which a user enters a user name and password, then submits the form by clicking the Login button.

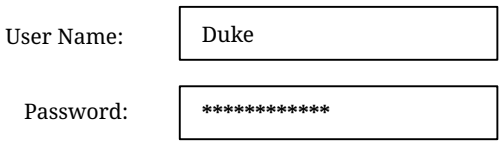


Figure 13. A Typical Form

The `h:form` tag represents the form on the page and encloses all the components that display or collect data from the user, as shown here:

```
<h:form>
... other Jakarta Faces tags and other content...
</h:form>
```

The `h:form` tag can also include HTML markup to lay out the components on the page. Note that the `h:form` tag itself does not perform any layout; its purpose is to collect data and to declare attributes that can be used by other components in the form.

A page can include multiple `h:form` tags, but only the values from the form submitted by the user will be included in the postback request.

Using Text Components

Text components allow users to view and edit text in web applications. The basic types of text components are as follows:

- Label, which displays read-only text
- Field, which allows users to enter text (on one or more lines), often to be submitted as part of a form
- Password field, which is a type of field that displays a set of characters, such as asterisks, instead of the password text that the user enters

Figure 14, "Example Text Components" shows examples of these text components.

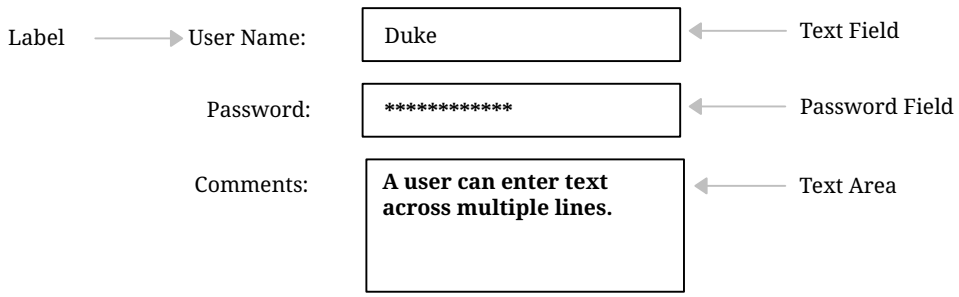


Figure 14. Example Text Components

Text components can be categorized as either input or output. A Jakarta Faces output component, such as a label, is rendered as read-only text. A Jakarta Faces input component, such as a field, is rendered as editable text.

The input and output components can each be rendered in various ways to display more specialized text.

Input Tags lists the tags that represent the input components.

Input Tags

Tag	Function
`h:inputHidden`	Allows a page author to include a hidden variable in a page
`h:inputSecret`	The standard password field: accepts one line of text with no spaces and displays it as a set of asterisks as it is entered
`h:inputText`	The standard field: accepts a one-line text string
`h:inputTextarea`	The standard multiline field: accepts multiple lines of text

The input tags support the tag attributes shown in Input Tag Attributes in addition to those described in Common Component Tag Attributes. Note that this table does not include all the attributes supported by the input tags but just those that are used most often. For the complete list of attributes, refer to the Jakarta Faces Facelets Tag Library documentation.

Input Tag Attributes

Attribute	Description
`converter`	Identifies a converter that will be used to convert the component's local data. See Using the Standard Converters for more information on how to use this attribute.
`converterMessage`	Specifies an error message to display when the converter registered on the component fails.
`dir`	Specifies the direction of the text displayed by this component. Acceptable values are `ltr`, meaning left to right, and `rtl`, meaning right to left.
`label`	Specifies a name that can be used to identify this component in error messages.
`lang`	Specifies the code for the language used in the rendered markup, such as `en` or `pt-BR`.
`required`	Takes a `boolean` value that indicates whether the user must enter a value in this component.
`requiredMessage`	Specifies an error message to display when the user does not enter a value into the component.
`validator`	Identifies a method expression pointing to a managed bean method that performs validation on the component's data. See Referencing a Method That Performs Validation for an example of using the `f:validator` tag.
`validatorMessage`	Specifies an error message to display when the validator registered on the component fails to validate the component's local value.
`valueChangeListener`	Identifies a method expression that points to a managed bean method that handles the event of entering a value in this component. See Referencing a Method That Handles a Value-Change Event for an example of using `valueChangeListener`.

Output Tags lists the tags that represent the output components.

Output Tags

Tag	Function
`h:outputFormat`	Displays a formatted message
`h:outputLabel`	The standard read-only label: displays a component as a label for a specified input field

Tag	Function
`h:outputLink`	Displays an `<a href>` tag that links to another page without generating an action event
`h:outputText`	Displays a one-line text string

The output tags support the `converter` tag attribute in addition to those listed in Common Component Tag Attributes.

The rest of this section explains how to use some of the tags listed in Output Tags. The other tags are written in a similar way.

Rendering a Field with the h:inputText Tag

The `h:inputText` tag is used to display a field. A similar tag, the `h:outputText` tag, displays a read-only, single-line string. This section shows you how to use the `h:inputText` tag. The `h:outputText` tag is written in a similar way.

Here is an example of an `h:inputText` tag:

```
<h:inputText id="name"
             label="Customer Name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             requiredMessage="#{bundle.ReqCustomerName}">
    <f:valueChangeListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

The `label` attribute specifies a user-friendly name that will be used in the substitution parameters of error messages displayed for this component.

The `value` attribute refers to the `name` property of a managed bean named `CashierBean`. This property holds the data for the `name` component. After the user submits the form, the value of the `name` property in `CashierBean` will be set to the text entered in the field corresponding to this tag.

The `required` attribute causes the page to reload, displaying errors, if the user does not enter a value in the `name` field. The Jakarta Faces implementation checks whether the value of the component is null or is an empty string.

If your component must have a non-null value or a `String` value at least one character in length, you should add a `required` attribute to your tag and set its value to `true`. If your tag has a `required` attribute that is set to `true` and the value is null or a zero-length string, no other validators that are registered on the tag are called. If your tag does not have a `required` attribute set to `true`, other validators that are registered on the tag are called, but those validators must handle the possibility of a null or zero-length string. See Validating Null and Empty Strings for more information.

Rendering a Password Field with the h:inputSecret Tag

The `h:inputSecret` tag renders an `<input type="password">` HTML tag. When the user types a string into this field, a row of asterisks is displayed instead of the text entered by the user. Here is an example:

```
<h:inputSecret redisplay="false" value="#{loginBean.password}" />
```

In this example, the `redisplay` attribute is set to `false`. This will prevent the password from being displayed in a query string or in the source file of the resulting HTML page.

Rendering a Label with the h:outputLabel Tag

The `h:outputLabel` tag is used to attach a label to a specified input field for the purpose of making it accessible. The following page uses an `h:outputLabel` tag to render the label of a check box:

```
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}">
    <h:outputText id="fanClubLabel"
                  value="#{bundle.DukeFanClub}" />
</h:outputLabel>
...
```

The `h:selectBooleanCheckbox` tag and the `h:outputLabel` tag have `rendered` attributes that are set to `false` on the page but are set to true in the `CashierBean` under certain circumstances. The `for` attribute of the `h:outputLabel` tag maps to the `id` of the input field to which the label is attached. The `h:outputText` tag nested inside the `h:outputLabel` tag represents the label component. The `value` attribute on the `h:outputText` tag indicates the text that is displayed next to the input field.

Instead of using an `h:outputText` tag for the text displayed as a label, you can simply use the `h:outputLabel` tag's `value` attribute. The following code snippet shows what the previous code snippet would look like if it used the `value` attribute of the `h:outputLabel` tag to specify the text of the label:

```
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}" />
</h:outputLabel>
...
```

Rendering a Link with the h:outputLink Tag

The `h:outputLink` tag is used to render a link that, when clicked, loads another page but does not generate an action event. You should use this tag instead of the `h:commandLink` tag if you always want the URL specified by the `h:outputLink` tag's `value` attribute to open and do not want any processing to be performed when the user clicks the link. Here is an example:

```
<h:outputLink value="javadocs">
    Documentation for this demo
</h:outputLink>
```

The text in the body of the `h:outputLink` tag identifies the text that the user clicks to get to the next page.

Displaying a Formatted Message with the h:outputFormat Tag

The `h:outputFormat` tag allows display of concatenated messages as a `MessageFormat` pattern, as described in the API documentation for `java.text.MessageFormat`. Here is an example of an `h:outputFormat` tag:

```
<h:outputFormat value="Hello, {0}!">
    <f:param value="#{hello.name}"/>
</h:outputFormat>
```

The `value` attribute specifies the `MessageFormat` pattern. The `f:param` tag specifies the substitution parameters for the message. The value of the parameter replaces the `{0}` in the sentence. If the value of `"#{hello.name}"` is "Bill", the message displayed in the page is as follows:

```
Hello, Bill!
```

An `h:outputFormat` tag can include more than one `f:param` tag for those messages that have more than one parameter that must be concatenated into the message. If you have more than one parameter for one message, make sure that you put the `f:param` tags in the proper order so that the data is inserted in the correct place in the message. Here is the preceding example modified with an additional parameter:

```
<h:outputFormat value="Hello, {0}! You are visitor number {1} to the page.">
    <f:param value="#{hello.name}" />
    <f:param value="#{bean.numVisitor}"/>
</h:outputFormat>
```

The value of `{1}` is replaced by the second parameter. The parameter is an EL expression, `bean.numVisitor`, in which the property `numVisitor` of the managed bean `bean` keeps track of visitors to the page. This is an example of a value-expression-enabled tag attribute accepting an EL expression. The message displayed in the page is now as follows:

```
Hello, Bill! You are visitor number 10 to the page.
```

Using Command Component Tags for Performing Actions and Navigation

In Jakarta Faces applications, the button and link component tags are used to perform actions, such as submitting a form, and for navigating to another page. These tags are called command component tags because they perform an action when activated.

The `h:commandButton` tag is rendered as a button. The `h:commandLink` tag is rendered as a link.

In addition to the tag attributes listed in Common Component Tag Attributes, the `h:commandButton` and `h:commandLink` tags can use the following attributes.

- `action`, which is either a logical outcome `String` or a method expression pointing to a bean method that returns a logical outcome `String`. In either case, the logical outcome `String` is used to determine what page to access when the command component tag is activated.
- `actionListener`, which is a method expression pointing to a bean method that processes an action event fired by the command component tag.

See Referencing a Method That Performs Navigation for more information on using the `action` attribute. See Referencing a Method That Handles an Action Event for details on using the `actionListener` attribute.

Rendering a Button with the h:commandButton Tag

If you are using an `h:commandButton` component tag, the data from the current page is processed when a user clicks the button, and the next page is opened. Here is an example of the `h:commandButton` tag:

```
<h:commandButton value="Submit"
                 action="#{cashierBean.submit}"/>
```

Clicking the button will cause the `submit` method of `CashierBean` to be invoked because the `action` attribute references this method. The `submit` method performs some processing and returns a logical outcome.

The `value` attribute of the example `h:commandButton` tag references the button's label. For information on how to use the `action` attribute, see Referencing a Method That Performs Navigation.

Rendering a Link with the h:commandLink Tag

The `h:commandLink` tag represents an HTML link and is rendered as an HTML `<a>` element.

An `h:commandLink` tag must include a nested `h:outputText` tag, which represents the text that the user clicks to generate the event. Here is an example:

```
<h:commandLink id="Duke" action="bookstore">
```

```
    <f:actionListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.LinkBookChangeListener" />
    <h:outputText value="#{bundle.Book201}"/>
</h:commandLink>
```

This tag will render HTML that looks something like the following:

```
<a id="_idt16:Duke" href="#"
     onclick="mojarra.cljs(document.getElementById('j_idt16'),
     {'j_idt16:Duke':'j_idt16:Duke'},'');
     return false;">My Early Years: Growing Up on Star7, by Duke</a>
```



The `h:commandLink` tag will render JavaScript scripting language. If you use this tag, make sure that your browser is enabled for JavaScript technology.

Adding Graphics and Images with the h:graphicImage Tag

In a Jakarta Faces application, use the `h:graphicImage` tag to render an image on a page:

```
<h:graphicImage id="mapImage" url="/resources/images/book_all.jpg"/>
```

In this example, the `url` attribute specifies the path to the image. The URL of the example tag begins with a slash (`/`), which adds the relative context path of the web application to the beginning of the path to the image.

Alternatively, you can use the facility described in Web Resources to point to the image location. Here are two examples:

```
<h:graphicImage id="mapImage"
                name="book_all.jpg"
                library="images"
                alt="#{bundle.ChooseBook}"
                usemap="#bookMap" />

<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
```

You can use similar syntax to refer to an image in a style sheet. The following syntax in a style sheet specifies that the image is to be found at `resources/img/top-background.jpg`:

```
header {
    position: relative;
    height: 150px;
    background: #fff url(#{resource['img:top-background.jpg']}) repeat-x;
    ...
}
```

Laying Out Components with the h:panelGrid and h:panelGroup Tags

In a Jakarta Faces application, you use a panel as a layout container for a set of other components. A panel is rendered as an HTML table. Panel Component Tags lists the tags used to create panels.

Panel Component Tags

Tag	Attributes	Function
`h:panelGrid`	`columns`, `columnClasses`, `footerClass`, `headerClass`, `panelClass`, `rowClasses`, `role`	Displays a table
`h:panelGroup`	`layout`	Groups a set of components under one parent

The `h:panelGrid` tag is used to represent an entire table. The `h:panelGroup` tag is used to represent rows in a table. Other tags are used to represent individual cells in the rows.

The `columns` attribute defines how to group the data in the table and therefore is required if you want your table to have more than one column. The `h:panelGrid` tag also has a set of optional attributes that specify CSS classes: `columnClasses`, `footerClass`, `headerClass`, `panelClass`, and `rowClasses`. The `role` attribute can have the value `"presentation"` to indicate that the purpose of the table is to format the display rather than to show data.

If the `headerClass` attribute value is specified, the `h:panelGrid` tag must have a header as its first child. Similarly, if a `footerClass` attribute value is specified, the `h:panelGrid` tag must have a footer as its last child.

Here is an example:

```
<h:panelGrid columns="2"
             headerClass="list-header"
             styleClass="list-background"
             rowClasses="list-row-even, list-row-odd"
             summary="#{bundle.CustomerInfo}"
             title="#{bundle.Checkout}"
             role="presentation">
    <f:facet name="header">
        <h:outputText value="#{bundle.Checkout}"/>
    </f:facet>

    <h:outputLabel for="name" value="#{bundle.Name}" />
    <h:inputText id="name" size="30"
                 value="#{cashierBean.name}"
                 required="true"
                 requiredMessage="#{bundle.ReqCustomerName}">
         <f:valueChangeListener
             type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
    </h:inputText>
    <h:message styleClass="error-message" for="name"/>
```

```
    <h:outputLabel for="ccno" value="#{bundle.CCNumber}"/>
    <h:inputText id="ccno"
                 size="19"
                 converterMessage="#{bundle.CreditMessage}"
                 required="true"
                 requiredMessage="#{bundle.ReqCreditCard}">
    <f:converter converterId="ccno"/>
    <f:validateRegex
        pattern="\d{16}|\d{4} \d{4} \d{4} \d{4}|\d{4}-\d{4}-\d{4}-\d{4}" />
    </h:inputText>
    <h:message styleClass="error-message"  for="ccno"/>
    ...
</h:panelGrid>
```

The preceding `h:panelGrid` tag is rendered as a table that contains components in which a customer inputs personal information. This `h:panelGrid` tag uses style sheet classes to format the table. The following code shows the `list-header` definition:

```
.list-header {
    background-color: #ffffff;
    color: #000000;
    text-align: center;
}
```

Because the `h:panelGrid` tag specifies a `headerClass`, the `h:panelGrid` tag must contain a header. The example `h:panelGrid` tag uses an `f:facet` tag for the header. Facets can have only one child, so an `h:panelGroup` tag is needed if you want to group more than one component within an `f:facet`. The example `h:panelGrid` tag has only one cell of data, so an `h:panelGroup` tag is not needed. (For more information about facets, see Using Data-Bound Table Components.

The `h:panelGroup` tag has an attribute, `layout`, in addition to those listed in Common Component Tag Attributes. If the `layout` attribute has the value `block`, an HTML `div` element is rendered to enclose the row; otherwise, an HTML `span` element is rendered to enclose the row. If you are specifying styles for the `h:panelGroup` tag, you should set the `layout` attribute to `block` in order for the styles to be applied to the components within the `h:panelGroup` tag. You should do this because styles, such as those that set width and height, are not applied to inline elements, which is how content enclosed by the `span` element is defined.

An `h:panelGroup` tag can also be used to encapsulate a nested tree of components so that the tree of components appears as a single component to the parent component.

Data, represented by the nested tags, is grouped into rows according to the value of the `columns` attribute of the `h:panelGrid` tag. The `columns` attribute in the example is set to `2`, and therefore the table will have two columns. The column in which each component is displayed is determined by the order in which the component is listed on the page modulo 2. So, if a component is the fifth one in the list of components, that component will be in the 5 modulo 2 column, or column 1.

Displaying Components for Selecting One Value

Another commonly used component is one that allows a user to select one value, whether it is the only value available or one of a set of choices. The most common tags for this kind of component are as follows:

- An `h:selectBooleanCheckbox` tag, displayed as a check box, which represents a Boolean state
- An `h:selectOneRadio` tag, displayed as a set of options
- An `h:selectOneMenu` tag, displayed as a scrollable list
- An `h:selectOneListbox` tag, displayed as an unscrollable list

Figure 15, "Example Components for Selecting One Item" shows examples of these components.

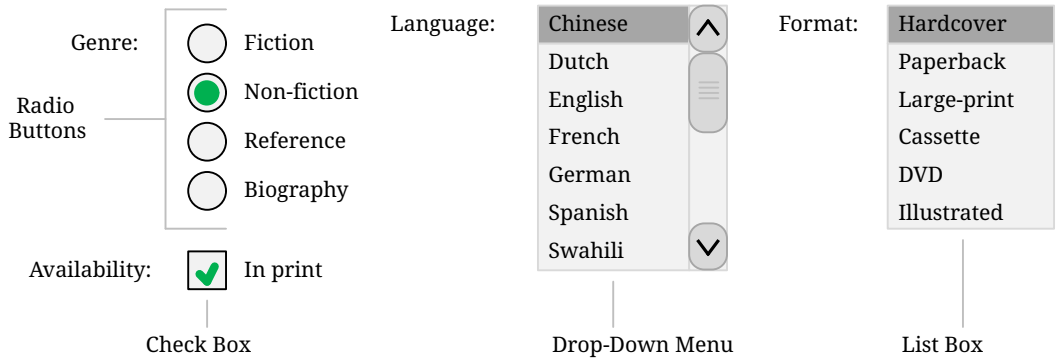


Figure 15. Example Components for Selecting One Item

Displaying a Check Box Using the h:selectBooleanCheckbox Tag

The `h:selectBooleanCheckbox` tag is the only tag that Jakarta Faces technology provides for representing a Boolean state.

Here is an example that shows how to use the `h:selectBooleanCheckbox` tag:

```
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}" />
```

The `h:selectBooleanCheckbox` tag and the `h:outputLabel` tag have `rendered` attributes that are set to `false` on the page but are set to true in the `CashierBean` under certain circumstances. When the `h:selectBooleanCheckbox` tag is rendered, it displays a check box to allow users to indicate whether they want to join the Duke Fan Club. When the `h:outputLabel` tag is rendered, it displays the label for the check box. The label text is represented by the `value` attribute.

Displaying a Menu Using the h:selectOneMenu Tag

A component that allows the user to select one value from a set of values can be rendered as a box

or a set of options. This section describes the `h:selectOneMenu` tag. The `h:selectOneRadio` and `h:selectOneListbox` tags are used in a similar way. The `h:selectOneListbox` tag is similar to the `h:selectOneMenu` tag except that `h:selectOneListbox` defines a `size` attribute that determines how many of the items are displayed at once.

The `h:selectOneMenu` tag represents a component that contains a list of items from which a user can select one item. This menu component is sometimes known as a drop-down list or a combo box. The following code snippet shows how the `h:selectOneMenu` tag is used to allow the user to select a shipping method:

```
<h:selectOneMenu id="shippingOption" required="true"
value="#{cashierBean.shippingOption}">
    <f:selectItem itemValue="2" itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem itemValue="5" itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem itemValue="7" itemLabel="#{bundle.SaverShip}"/>
 </h:selectOneMenu>
```

The `value` attribute of the `h:selectOneMenu` tag maps to the property that holds the currently selected item's value. In this case, the value is set by the backing bean. You are not required to provide a value for the currently selected item. If you don't provide a value, the browser determines which one is selected.

Like the `h:selectOneRadio` tag, the `h:selectOneMenu` tag must contain either an `f:selectItems` tag or a set of `f:selectItem` tags for representing the items in the list. Using the f:selectItem and f:selectItems Tags describes these tags.

Displaying Components for Selecting Multiple Values

In some cases, you need to allow your users to select multiple values rather than just one value from a list of choices. You can do this using one of the following component tags:

- An `h:selectManyCheckbox` tag, displayed as a set of check boxes
- An `h:selectManyMenu` tag, displayed as a menu
- An `h:selectManyListbox` tag, displayed as a box

Figure 16, "Example Components for Selecting Multiple Values" shows examples of these components.

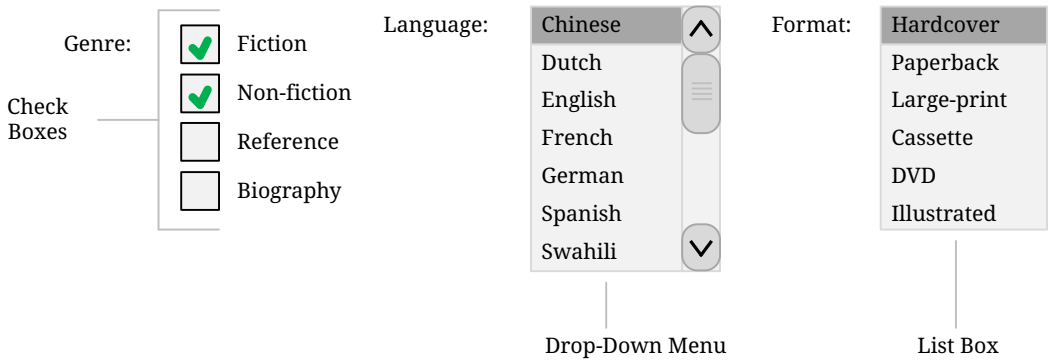


Figure 16. Example Components for Selecting Multiple Values

These tags allow the user to select zero or more values from a set of values. This section explains the `h:selectManyCheckbox` tag. The `h:selectManyListbox` and `h:selectManyMenu` tags are used in a similar way.

Unlike a menu, a list displays a subset of items in a box; a menu displays only one item at a time when the user is not selecting the menu. The `size` attribute of the `h:selectManyListbox` tag determines the number of items displayed at one time. The box includes a scroll bar for scrolling through any remaining items in the list.

The `h:selectManyCheckbox` tag renders a group of check boxes, with each check box representing one value that can be selected:

```
<h:selectManyCheckbox id="newslettercheckbox"
                      layout="pageDirection"
                      value="#{cashierBean.newsletters}">
    <f:selectItems value="#{cashierBean.newsletterItems}"/>
</h:selectManyCheckbox>
```

The `value` attribute of the `h:selectManyCheckbox` tag identifies the `newsletters` property of the `CashierBean` managed bean. This property holds the values of the currently selected items from the set of check boxes. You are not required to provide a value for the currently selected items. If you don't provide a value, the first item in the list is selected by default. In the `CashierBean` managed bean, this value is instantiated to 0, so no items are selected by default.

The `layout` attribute indicates how the set of check boxes is arranged on the page. Because layout is set to `pageDirection`, the check boxes are arranged vertically. The default is `lineDirection`, which aligns the check boxes horizontally.

The `h:selectManyCheckbox` tag must also contain a tag or set of tags representing the set of check boxes. To represent a set of items, you use the `f:selectItems` tag. To represent each item individually, you use the `f:selectItem` tag. The following section explains these tags in more detail.

Using the f:selectItem and f:selectItems Tags

The `f:selectItem` and `f:selectItems` tags represent components that can be nested inside a component that allows you to select one or multiple items. An `f:selectItem` tag contains the value, label, and description of a single item. An `f:selectItems` tag contains the values, labels, and descriptions of the entire list of items.

You can use either a set of `f:selectItem` tags or a single `f:selectItems` tag within your component tag.

The advantages of using the `f:selectItems` tag are as follows.

- Items can be represented by using different data structures, including `Array`, `Map`, and `Collection`. The value of the `f:selectItems` tag can represent even a generic collection of POJOs.
- Different lists can be concatenated into a single component, and the lists can be grouped within the component.

- Values can be generated dynamically at runtime.

The advantages of using `f:selectItem` are as follows.

- Items in the list can be defined from the page.
- Less code is needed in the backing bean for the `f:selectItem` properties.

The rest of this section shows you how to use the `f:selectItems` and `f:selectItem` tags.

Using the f:selectItems Tag

The following example from Displaying Components for Selecting Multiple Values shows how to use the `h:selectManyCheckbox` tag:

```
<h:selectManyCheckbox id="newslettercheckbox"
                      layout="pageDirection"
                      value="#{cashierBean.newsletters}">
    <f:selectItems value="#{cashierBean.newsletterItems}"/>
</h:selectManyCheckbox>
```

The `value` attribute of the `f:selectItems` tag is bound to the managed bean property `cashierBean.newsletterItems`. The individual `SelectItem` objects are created programmatically in the managed bean.

See UISelectItems Properties for information on how to write a managed bean property for one of these tags.

Using the f:selectItem Tag

The `f:selectItem` tag represents a single item in a list of items. Here is the example from Displaying a Menu Using the h:selectOneMenu Tag once again:

```
<h:selectOneMenu id="shippingOption"
                 required="true"
                 value="#{cashierBean.shippingOption}">
    <f:selectItem itemValue="2"
                  itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem itemValue="5"
                  itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem itemValue="7"
                  itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

The `itemValue` attribute represents the value for the `f:selectItem` tag. The `itemLabel` attribute represents the `String` that appears in the list component on the page.

The `itemValue` and `itemLabel` attributes are value-binding enabled, meaning that they can use value-binding expressions to refer to values in external objects. These attributes can also define literal

values, as shown in the example `h:selectOneMenu` tag.

Displaying the Results from Selection Components

If you display components that allow a user to select values, you may also want to display the result of the selection.

For example, you might want to thank a user who selected the checkbox to join the Duke Fan Club, as described in Displaying a Check Box Using the h:selectBooleanCheckbox Tag. Because the checkbox is bound to the `specialOffer` property of `CashierBean`, a `UISelectBoolean` value, you can call the `isSelected` method of the property to determine whether to render a thank-you message:

```
<h:outputText value="#{bundle.DukeFanClubThanks}"
              rendered="#{cashierBean.specialOffer.isSelected()}"/>
```

Similarly, you might want to acknowledge that a user subscribed to newsletters using the `h:selectManyCheckbox` tag, as described in Displaying Components for Selecting Multiple Values. To do so, you can retrieve the value of the `newsletters` property, the `String` array that holds the selected items:

```
<h:outputText value="#{bundle.NewsletterThanks}"
              rendered="#{!empty cashierBean.newsletters}"/>
<ul>
    <ui:repeat value="#{cashierBean.newsletters}" var="nli">
        <li><h:outputText value="#{nli}" /></li>
    </ui:repeat>
</ul>
```

An introductory thank-you message is displayed only if the `newsletters` array is not empty. Then a `ui:repeat` tag, a simple way to show values in a loop, displays the contents of the selected items in an itemized list. (This tag is listed in Facelets Templating Tags.)

Using Data-Bound Table Components

Data-bound table components display relational data in a tabular format. In a Jakarta Faces application, the `h:dataTable` component tag supports binding to a collection of data objects and displays the data as an HTML table. The `h:column` tag represents a column of data within the table, iterating over each record in the data source, which is displayed as a row. Here is an example:

```
<h:dataTable id="items"
             captionClass="list-caption"
             columnClasses="list-column-center, list-column-left,
             list-column-right, list-column-center"
             footerClass="list-footer"
             headerClass="list-header"
             rowClasses="list-row-even, list-row-odd"
             styleClass="list-background"
             summary="#{bundle.ShoppingCart}"
```

```
            value="#{cart.items}"
            border="1"
            var="item">
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemQuantity}" />
        </f:facet>
        <h:inputText id="quantity"
                     size="4"
                     value="#{item.quantity}"
                     title="#{bundle.ItemQuantity}">
            <f:validateLongRange minimum="1"/>
            <f:valueChangeListener
                type="ee.jakarta.tutorial.dukesbookstore.listeners.QuantityChanged"/>
        </h:inputText>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemTitle}"/>
        </f:facet>
        <h:commandLink action="#{showcart.details}">
            <h:outputText value="#{item.item.title}"/>
        </h:commandLink>
    </h:column>
    ...
    <f:facet name="footer">
        <h:panelGroup>
            <h:outputText value="#{bundle.Subtotal}"/>
            <h:outputText value="#{cart.total}" />
                <f:convertNumber currencySymbol="$" type="currency" />
            </h:outputText>
        </h:panelGroup>
    </f:facet>
    <f:facet name="caption">
        <h:outputText value="#{bundle.Caption}"/>
    </f:facet>
</h:dataTable>
```

The example `h:dataTable` tag displays the books in the shopping cart as well as the quantity of each book in the shopping cart, the prices, and a set of buttons the user can click to remove books from the shopping cart.

The `h:column` tags represent columns of data in a data component. While the data component is iterating over the rows of data, it processes the column component associated with each `h:column` tag for each row in the table.

The `h:dataTable` tag shown in the preceding code example iterates through the list of books (`cart.items`) in the shopping cart and displays their titles, authors, and prices. Each time the `h:dataTable` tag iterates through the list of books, it renders one cell in each column.

The h:dataTable and h:column tags use facets to represent parts of the table that are not repeated or updated. These parts include headers, footers, and captions.

In the preceding example, h:column tags include f:facet tags for representing column headers or footers. The h:column tag allows you to control the styles of these headers and footers by supporting the headerClass and footerClass attributes. These attributes accept space-separated lists of CSS classes, which will be applied to the header and footer cells of the corresponding column in the rendered table.

Facets can have only one child, so an h:panelGroup tag is needed if you want to group more than one component within an f:facet. Because the facet tag representing the footer includes more than one tag, the h:panelGroup tag is needed to group those tags. Finally, this h:dataTable tag includes an f:facet tag with its name attribute set to caption, causing a table caption to be rendered above the table.

This table is a classic use case for a data component because the number of books might not be known to the application developer or the page author when that application is developed. The data component can dynamically adjust the number of rows of the table to accommodate the underlying data.

The value attribute of an h:dataTable tag references the data to be included in the table. This data can take the form of any of the following:

- A list of beans
- An array of beans
- A single bean
- A jakarta.faces.model.DataModel object
- A java.sql.ResultSet object
- A jakarta.servlet.jsp.jstl.sql.Result object
- A javax.sql.RowSet object

All data sources for data components have a DataModel wrapper. Unless you explicitly construct a DataModel wrapper, the Jakarta Faces implementation will create one around data of any of the other acceptable types. See Writing Bean Properties for more information on how to write properties for use with a data component.

The var attribute specifies a name that is used by the components within the h:dataTable tag as an alias to the data referenced in the value attribute of h:dataTable.

In the example h:dataTable tag, the value attribute points to a list of books. The var attribute points to a single book in that list. As the h:dataTable tag iterates through the list, each reference to item points to the current book in the list.

The h:dataTable tag also has the ability to display only a subset of the underlying data. This feature is not shown in the preceding example. To display a subset of the data, you use the optional first and rows attributes.

The first attribute specifies the first row to be displayed. The rows attribute specifies the number of

rows, starting with the first row, to be displayed. For example, if you wanted to display records 2 through 10 of the underlying data, you would set `first` to 2 and `rows` to 9. When you display a subset of the data in your pages, you might want to consider including a link or button that causes subsequent rows to display when clicked. By default, both `first` and `rows` are set to zero, and this causes all the rows of the underlying data to display.

Optional Attributes for the h:dataTable Tag shows the optional attributes for the `h:dataTable` tag.

Optional Attributes for the h:dataTable Tag

Attribute	Defines Styles For
`captionClass`	Table caption
`columnClasses`	All the columns
`footerClass`	Footer
`headerClass`	Header
`rowClasses`	Rows
`styleClass`	The entire table

Each of the attributes in Optional Attributes for the h:dataTable Tag can specify more than one style. If `columnClasses` or `rowClasses` specifies more than one style, the styles are applied to the columns or rows in the order that the styles are listed in the attribute. For example, if `columnClasses` specifies styles `list-column-center` and `list-column-right`, and if the table has two columns, the first column will have style `list-column-center`, and the second column will have style `list-column-right`.

If the style attribute specifies more styles than there are columns or rows, the remaining styles will be assigned to columns or rows starting from the first column or row. Similarly, if the style attribute specifies fewer styles than there are columns or rows, the remaining columns or rows will be assigned styles starting from the first style.

Displaying Error Messages with the h:message and h:messages Tags

The `h:message` and `h:messages` tags are used to display error messages when conversion or validation fails. The `h:message` tag displays error messages related to a specific input component, whereas the `h:messages` tag displays the error messages for the entire page.

Here is an example `h:message` tag from the `guessnumber-faces` application:

```
<p>
    <h:inputText id="userNo"
                 title="Type a number from 0 to 10:"
                 value="#{userNumberBean.userNumber}">
        <f:validateLongRange minimum="#{userNumberBean.minimum}"
                             maximum="#{userNumberBean.maximum}"/>
   </h:inputText>
   <h:commandButton id="submit" value="Submit"
                    action="response"/>
</p>
```

```
<h:message showSummary="true" showDetail="false"
           style="color: #d20005;
           font-family: 'New Century Schoolbook', serif;
           font-style: oblique;
           text-decoration: overline"
           id="errors1"
           for="userNo"/>
```

The `for` attribute refers to the ID of the component that generated the error message. The error message is displayed at the same location that the `h:message` tag appears in the page. In this case, the error message will appear below the Submit button.

The `style` attribute allows you to specify the style of the text of the message. In the example in this section, the text will be a shade of red, New Century Schoolbook, serif font family, and oblique style, and a line will appear over the text. The message and messages tags support many other attributes for defining styles. For more information on these attributes, refer to the Jakarta Faces Facelets Tag Library documentation.

Another attribute supported by the `h:messages` tag is the `layout` attribute. Its default value is `list`, which indicates that the messages are displayed in a bullet list using the HTML `ul` and `li` elements. If you set the attribute value to `table`, the messages will be rendered in a table using the HTML `table` element.

The preceding example shows a standard validator that is registered on the input component. The message tag displays the error message that is associated with this validator when the validator cannot validate the input component's value. In general, when you register a converter or validator on a component, you are queueing the error messages associated with the converter or validator on the component. The `h:message` and `h:messages` tags display the appropriate error messages that are queued on the component when the validators or converters registered on that component fail to convert or validate the component's value.

Standard error messages are provided with standard converters and standard validators. An application architect can override these standard messages and supply error messages for custom converters and validators by registering custom error messages with the application.

Creating Bookmarkable URLs with the h:button and h:link Tags

The ability to create bookmarkable URLs refers to the ability to generate links based on a specified navigation outcome and on component parameters.

In HTTP, most browsers by default send GET requests for URL retrieval and POST requests for data processing. The GET requests can have query parameters and can be cached, which is not advised for POST requests, which send data to servers for processing. The other Jakarta Faces tags capable of generating links use either simple GET requests, as in the case of `h:outputLink`, or POST requests, as in the case of `h:commandLink` or `h:commandButton` tags. GET requests with query parameters provide finer granularity to URL strings. These URLs are created with one or more `name=value` parameters appended to the simple URL after a `?` character and separated by either `&;` or `&` strings.

To create a bookmarkable URL, use an `h:link` or `h:button` tag. Both of these tags can generate a link

based on the outcome attribute of the component. For example:

```
<h:link outcome="somepage" value="Message" />
```

The h:link tag will generate a URL link that points to the somepage.xhtml file on the same server. The following sample HTML is generated from the preceding tag, assuming that the application name is simplebookmark:

```
<a href="/simplebookmark/somepage.xhtml">Message</a>
```

This is a simple GET request that cannot pass any data from page to page. To create more complex GET requests and utilize the complete functionality of the h:link tag, use view parameters.

Using View Parameters to Configure Bookmarkable URLs

To pass a parameter from one page to another, use the includeViewParams attribute in your h:link tag and, in addition, use an f:param tag to specify the name and value to be passed. Here the h:link tag specifies the outcome page as personal.xhtml and provides a parameter named Result whose value is a managed bean property:

```
<h:body>
    <h:form>
        <h:graphicImage url="#{resource['images:duke.waving.gif']}"
                        alt="Duke waving his hand"/>
        <h2>Hello, #{hello.name}!</h2>
        <p>I've made your
            <h:link outcome="personal" value="personal greeting page!"
                    includeViewParams="true">
                <f:param name="Result" value="#{hello.name}"/>
            </h:link>
        </p>
        <h:commandButton id="back" value="Back" action="index" />
    </h:form>
</h:body>
```

If the includeViewParams attribute is set on the component, the view parameters are added to the hyperlink. Therefore, the resulting URL will look something like this if the value of hello.name is Timmy:

```
http://localhost:8080/bookmarks/personal.xhtml?Result=Timmy
```

On the outcome page, specify the core tags f:metadata and f:viewparam as the source of parameters for configuring the URLs. View parameters are declared as part of f:metadata for a page, as shown in the following example:

```
<f:metadata>
    <f:viewParam name="Result" value="#{hello.name}"/>
</f:metadata>
```

This allows you to specify the bean property value on the page:

```
<h:outputText value="Howdy, #{hello.name}!" />
```

As a view parameter, the name also appears in the page's URL. If you edit the URL, you change the output on the page.

Because the URL can be the result of various parameter values, the order of the URL creation has been predefined. The order in which the various parameter values are read is as follows:

1. Component
2. Navigation-case parameters
3. View parameters

The bookmarks Example Application

The `bookmarks` example application modifies the `hello1` application described in A Web Module That Uses Jakarta Faces Technology: The hello1 Example to use a bookmarkable URL that uses view parameters.

Like `hello1`, the application includes the `Hello.java` managed bean, an `index.xhtml` page, and a `response.xhtml` page. In addition, it includes a `personal.xhtml page`, to which a bookmarkable URL and view parameters are passed from the `response.xhtml` page, as described in Using View Parameters to Configure Bookmarkable URLs.

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `bookmarks` example. The source code for this example is in the `jakartaee-examples/tutorial/web/faces/bookmarks/` directory.

To Build, Package, and Deploy the bookmarks Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `bookmarks` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `bookmarks` project and select **Build**.

This option builds the example application and deploys it to your GlassFish Server instance.

To Build, Package, and Deploy the bookmarks Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/bookmarks/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `bookmarks.war`, that is located in the `target` directory. It then deploys the WAR file to your GlassFish Server instance.

To Run the bookmarks Example

1. Enter the following URL in your web browser:

   ```
   http://localhost:8080/bookmarks
   ```

2. In the text field, enter a name and click Submit.
3. On the response page, move your mouse over the "personal greeting page" link to view the URL with the view parameter, then click the link.

   The `personal.xhtml` page opens, displaying a greeting to the name you typed.

4. In the URL field, modify the Result parameter value and press Return.

   The name in the greeting changes to what you typed.

Resource Relocation Using h:outputScript and h:outputStylesheet Tags

Resource relocation refers to the ability of a Jakarta Faces application to specify the location where a resource can be rendered. Resource relocation can be defined with the following HTML tags:

- `h:outputScript`
- `h:outputStylesheet`

These tags have `name` and `target` attributes, which can be used to define the render location. For a complete list of attributes for these tags, see the Jakarta Faces Facelets Tag Library documentation.

For the `h:outputScript` tag, the `name` and `target` attributes define where the output of a resource may appear. Here is an example:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html">
    <h:head id="head">
        <title>Resource Relocation</title>
    </h:head>
    <h:body id="body">
        <h:form id="form">
            <h:outputScript name="hello.js"/>
            <h:outputStylesheet name="hello.css"/>
        </h:form>
    </h:body>
</html>
```

Because the `target` attribute is not defined in the tags, the style sheet `hello.css` is rendered in the head element of the page, and the `hello.js` script is rendered in the body of the page.

Here is the HTML generated by the preceding code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Resource Relocation</title>
        <link type="text/css" rel="stylesheet"
              href="/context-root/jakarta.faces.resource/hello.css"/>
    </head>
    <body>
        <form id="form" name="form" method="post"
              action="..." enctype="...">
            <script type="text/javascript"
                    src="/context-root/jakarta.faces.resource/hello.js">
            </script>
        </form>
    </body>
</html>
```

If you set the `target` attribute for the `h:outputScript` tag, the incoming GET request provides the location parameter. Here is an example:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html">
    <h:head id="head">
        <title>Resource Relocation</title>
    </h:head>
    <h:body id="body">
        <h:form id="form">
            <h:outputScript name="hello.js" target="#{param.location}"/>
            <h:outputStylesheet name="hello.css"/>
        </h:form>
    </h:body>
```

```
</html>
```

In this case, if the incoming request does not provide a location parameter, the default locations will still apply: The style sheet is rendered in the head, and the script is rendered inline. However, if the incoming request specifies the location parameter as the head, both the style sheet and the script will be rendered in the `head` element.

The HTML generated by the preceding code is as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Resource Relocation</title>
        <link type="text/css" rel="stylesheet"
              href="/context-root/jakarta.faces.resource/hello.css"/>
        <script type="text/javascript"
                src="/context-root/jakarta.faces.resource/hello.js">
          </script>
    </head>
    <body>
        <form id="form" name="form" method="post"
              action="..." enctype="...">
        </form>
    </body>
</html>
```

Similarly, if the incoming request provides the location parameter as the body, the script will be rendered in the body element.

The preceding section describes simple uses for resource relocation. That feature can add even more functionality for the components and pages. A page author does not have to know the location of a resource or its placement.

By using a `@ResourceDependency` annotation for the components, component authors can define the resources for the component, such as a style sheet and script. This allows the page authors freedom from defining resource locations.

Using Core Tags

The tags included in the Jakarta Faces core tag library are used to perform core actions that are not performed by HTML tags.

Event-Handling Core Tags lists the event-handling core tags.

Event-Handling Core Tags

Tag	Function
`f:actionListener`	Adds an action listener to a parent component
`f:phaseListener`	Adds a `PhaseListener` to a page

Tag	Function
`f:setPropertyActionListener`	Registers a special action listener whose sole purpose is to push a value into a managed bean when a form is submitted
`f:valueChangeListener`	Adds a value-change listener to a parent component

Data-Conversion Core Tags lists the data-conversion core tags.

Data-Conversion Core Tags

Tag	Function
`f:converter`	Adds an arbitrary converter to the parent component
`f:convertDateTime`	Adds a `DateTimeConverter` instance to the parent component
`f:convertNumber`	Adds a `NumberConverter` instance to the parent component

Facet Core Tags lists the facet core tags.

Facet Core Tags

Tag	Function
`f:facet`	Adds a nested component that has a special relationship to its enclosing tag
`f:metadata`	Registers a `facet` on a parent component

Core Tags That Represent Items in a List lists the core tags that represent items in a list.

Core Tags That Represent Items in a List

Tag	Function
`f:selectItem`	Represents one item in a list of items
`f:selectItems`	Represents a set of items

Validator Core Tags lists the validator core tags.

Validator Core Tags

Tag	Function
`f:validateDoubleRange`	Adds a `DoubleRangeValidator` to a component
`f:validateLength`	Adds a `LengthValidator` to a component
`f:validateLongRange`	Adds a `LongRangeValidator` to a component
`f:validator`	Adds a custom validator to a component
`f:validateRegEx`	Adds a `RegExValidator` to a component
`f:validateBean`	Delegates the validation of a local value to a `BeanValidator`
`f:validateRequired`	Enforces the presence of a value in a component

Miscellaneous Core Tags lists the core tags that fall into other categories.

Miscellaneous Core Tags

Tag Category	Tag	Function
Attribute configuration	`f:attribute`	Adds configurable attributes to a parent component
Localization	`f:loadBundle`	Specifies a `ResourceBundle` that is exposed as a `Map`
Parameter substitution	`f:param`	Substitutes parameters into a `MessageFormat` instance and adds query string name-value pairs to a URL
Ajax	`f:ajax`	Associates an Ajax action with a single component or a group of components based on placement
Event	`f:event`	Allows installing a `ComponentSystemEventListener` on a component
WebSocket	`f:websocket`	Allows server-side communications to be pushed to all instances of a socket containing the same channel name.

These tags, which are used in conjunction with component tags, are explained in other sections of this tutorial.

Where the Core Tags Are Explained lists the sections that explain how to use specific core tags.

Where the Core Tags Are Explained

Tags	Where Explained
Event-handling tags	Registering Listeners on Components
Data-conversion tags	Using the Standard Converters
`f:facet`	Using Data-Bound Table Components and Laying Out Components with the h:panelGrid and h:panelGroup Tags
`f:loadBundle`	Setting the Resource Bundle
`f:metadata`	Using View Parameters to Configure Bookmarkable URLs
`f:param`	Displaying a Formatted Message with the h:outputFormat Tag
`f:selectItem` and `f:selectItems`	Using the f:selectItem and f:selectItems Tags
Validator tags	Using the Standard Validators
`f:ajax`	[web:faces-ajax::faces-ajax:::_using_ajax_with_jakarta_faces_technology]
`f:websocket`	[web:faces-ws::faces-ws:::_using_websockets_with_jakarta_faces_technology]

Using Converters, Listeners, and Validators



This section was last updated for Jakarta EE Version 9.1

The previous chapter described components and explained how to add them to a web page. This chapter provides information on adding more functionality to the components through converters,

listeners, and validators.

- Converters are used to convert data that is received from the input components. Converters allow an application to bring the strongly typed features of the Java programming language into the String-based world of HTTP servlet programming.
- Listeners are used to listen to the events happening in the page and perform actions as defined.
- Validators are used to validate the data that is received from the input components. Validators allow an application to express constraints on form input data to ensure that the necessary requirements are met before the input data is processed.

Using the Standard Converters

The Jakarta Faces implementation provides a set of `Converter` implementations that you can use to convert component data. The purpose of conversion is to take the String-based data coming in from the Servlet API and convert it to strongly typed Java objects suitable for the business domain. For more information on the conceptual details of the conversion model, see Conversion Model.

The standard `Converter` implementations are located in the `jakarta.faces.convert` package. Normally, converters are implicitly assigned based on the type of the EL expression pointed to by the value of the component. However, these converters can also be accessed by a converter ID. Converter Classes and Converter IDs shows the converter classes and their associated converter IDs.

Converter Classes and Converter IDs

Class in the jakarta.faces.convert Package	Converter ID
`BigDecimalConverter`	`jakarta.faces.BigDecimal`
`BigIntegerConverter`	`jakarta.faces.BigInteger`
`BooleanConverter`	`jakarta.faces.Boolean`
`ByteConverter`	`jakarta.faces.Byte`
`CharacterConverter`	`jakarta.faces.Character`
`DateTimeConverter`	`jakarta.faces.DateTime`
`DoubleConverter`	`jakarta.faces.Double`
`EnumConverter`	`jakarta.faces.Enum`
`FloatConverter`	`jakarta.faces.Float`
`IntegerConverter`	`jakarta.faces.Integer`
`LongConverter`	`jakarta.faces.Long`
`NumberConverter`	`jakarta.faces.Number`
`ShortConverter`	`jakarta.faces.Short`

A standard error message is associated with each of these converters. If you have registered one of these converters onto a component on your page and the converter is not able to convert the component's value, the converter's error message will display on the page. For example, the following error message appears if `BigIntegerConverter` fails to convert a value:

```
{0} must be a number consisting of one or more digits
```

In this case, the `{0}` substitution parameter will be replaced with the name of the input component on which the converter is registered.

Two of the standard converters (`DateTimeConverter` and `NumberConverter`) have their own tags, which allow you to configure the format of the component data using the tag attributes. For more information about using `DateTimeConverter`, see Using DateTimeConverter. For more information about using `NumberConverter`, see Using NumberConverter. The following section explains how to convert a component's value, including how to register other standard converters with a component.

Converting a Component's Value

To use a particular converter to convert a component's value, you need to register the converter onto the component. You can register any of the standard converters in one of the following ways.

- Nest one of the standard converter tags inside the component's tag. These tags are `f:convertDateTime` and `f:convertNumber`, which are described in Using NumberConverter, respectively.
- Bind the value of the component to a managed bean property of the same type as the converter. This is the most common technique.
- Refer to the converter from the component tag's `converter` attribute, specifying the ID of the converter class.
- Nest an `f:converter` tag inside of the component tag, and use either the `f:converter` tag's `converterId` attribute or its `binding` attribute to refer to the converter.

As an example of the second technique, if you want a component's data to be converted to an `Integer`, you can simply bind the component's value to a managed bean property. Here is an example:

```
Integer age = 0;
public Integer getAge(){ return age;}
public void setAge(Integer age) {this.age = age;}
```

The data from the `h:inputText` tag in the this example will be converted to a `java.lang.Integer` value. The `Integer` type is a supported type of `NumberConverter`. If you don't need to specify any formatting instructions using the `f:convertNumber` tag attributes, and if one of the standard converters will suffice, you can simply reference that converter by using the component tag's `converter` attribute.

You can also nest an `f:converter` tag within the component tag and use either the converter tag's `converterId` attribute or its `binding` attribute to reference the converter.

The `converterId` attribute must reference the converter's ID. Here is an example that uses one of the converter IDs listed in Converter Classes and Converter IDs:

```
<h:inputText value="#{loginBean.age}">
    <f:converter converterId="jakarta.faces.Integer" />
</h:inputText>
```

Instead of using the `converterId` attribute, the `f:converter` tag can use the `binding` attribute. The `binding` attribute must resolve to a bean property that accepts and returns an appropriate `Converter` instance.

You can also create custom converters and register them on components using the `f:converter` tag. For details, see Creating and Using a Custom Converter.

Using DateTimeConverter

You can convert a component's data to a `java.util.Date` by nesting the `convertDateTime` tag inside the component tag. The `convertDateTime` tag has several attributes that allow you to specify the format and type of the data. Attributes for the f:convertDateTime Tag lists the attributes.

Here is a simple example of a `convertDateTime` tag:

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

When binding the `DateTimeConverter` to a component, ensure that the managed bean property to which the component is bound is of type `java.util.Date`. In the preceding example, `cashierBean.shipDate` must be of type `java.util.Date`.

The example tag can display the following output:

```
Saturday, September 21, 2013
```

You can also display the same date and time by using the following tag in which the date format is specified:

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime pattern="EEEEEEEE, MMM dd, yyyy" />
</h:outputText>
```

If you want to display the example date in Spanish, you can use the `locale` attribute:

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime dateStyle="full"
                       locale="es"
                       timeStyle="long" type="both" />
</h:outputText>
```

This tag would display the following output:

```
jueves 24 de octubre de 2013 15:07:04 GMT
```

Refer to the "Customizing Formats" lesson of the Java Tutorial at https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html for more information on how to format the output using the `pattern` attribute of the `convertDateTime` tag.

Attributes for the f:convertDateTime Tag

Attribute	Type	Description
`binding`	`DateTimeConverter`	Used to bind a converter to a managed bean property.
`dateStyle`	`String`	Defines the format, as specified by `java.text.DateFormat`, of a date or the date part of a `date` string. Applied only if `type` is `date` or `both` and if `pattern` is not defined. Valid values: `default`, `short`, `medium`, `long`, and `full`. If no value is specified, `default` is used.
`for`	`String`	Used with composite components. Refers to one of the objects within the composite component inside which this tag is nested.
`locale`	`String` or `Locale`	`Locale` whose predefined styles for dates and times are used during formatting or parsing. If not specified, the `Locale` returned by `FacesContext.getLocale` will be used.
`pattern`	`String`	Custom formatting pattern that determines how the date/time string should be formatted and parsed. If this attribute is specified, `dateStyle` and `timeStyle` attributes are ignored. See Type Attribute and Default Pattern Values for the default values when `pattern` is not specified.
`timeStyle`	`String`	Defines the format, as specified by `java.text.DateFormat`, of a `time` or the time part of a `date` string. Applied only if `type` is time and `pattern` is not defined. Valid values: `default`, `short`, `medium`, `long`, and `full`. If no value is specified, `default` is used.
`timeZone`	`String` or `TimeZone`	Time zone in which to interpret any time information in the `date` string.

Attribute	Type	Description
type	String	Specifies whether the string value will contain a date, a time, or both. Valid values are: date, time, both, LocalDate, LocalTime, LocalDateTime, OffsetTime, OffsetDateTime, or ZonedDateTime. If no value is specified, date is used. See Type Attribute and Default Pattern Values for additional information.

Type Attribute and Default Pattern Values

Type Attribute	Class	Default When Pattern Is Not Specified
both	java.util.Date	DateFormat.getDateTimeInstance(dateStyle,timeStyle)
date	java.util.Date	DateFormat.getDateTimeInstance(dateStyle)
time	java.util.Date	DateFormat.getDateTimeInstance(timeStyle)
localDate	java.time.LocalDate	DateTimeFormatter.ofLocalizedDate(dateStyle)
localTime	java.time.LocalTime	DateTimeFormatter.ofLocalizedTime(dateStyle)
localDateTime	java.time.LocalDateTime	DateTimeFormatter.ofLocalizedDateTime(dateStyle)
offsetTime	java.time.OffsetTime	DateTimeFormatter.ISO_OFFSET_TIME
offsetDateTime	java.time.OffsetDateTime	DateTimeFormatter.ISO_OFFSET_DATE_TIME
zonedDateTime	java.time.ZonedDateTime	DateTimeFormatter.ISO_ZONED_DATE_TIME

Using NumberConverter

You can convert a component's data to a java.lang.Number by nesting the convertNumber tag inside the component tag. The convertNumber tag has several attributes that allow you to specify the format and type of the data. Attributes for the f:convertNumber Tag lists the attributes.

The following example uses a convertNumber tag to display the total prices of the contents of a shopping cart:

```
<h:outputText value="#{cart.total}">
    <f:convertNumber currencySymbol="$" type="currency"/>
</h:outputText>
```

When binding the NumberConverter to a component, ensure that the managed bean property to which the component is bound is of a primitive type or has a type of java.lang.Number. In the preceding example, cart.total is of type double.

Here is an example of a number that this tag can display:

```
$934
```

This result can also be displayed by using the following tag in which the currency pattern is specified:

```
<h:outputText id="cartTotal" value="#{cart.total}">
    <f:convertNumber pattern="$####" />
</h:outputText>
```

See the "Customizing Formats" lesson of the Java Tutorial at https://docs.oracle.com/javase/tutorial/i18n/format/decimalFormat.html for more information on how to format the output by using the `pattern` attribute of the `convertNumber` tag.

Attributes for the f:convertNumber Tag

Attribute	Type	Description
`binding`	`NumberConverter`	Used to bind a converter to a managed bean property.
`currencyCode`	`String`	ISO 4217 currency code, used only when formatting currencies.
`currencySymbol`	`String`	Currency symbol, applied only when formatting currencies.
`for`	`String`	Used with composite components. Refers to one of the objects within the composite component inside which this tag is nested.
`groupingUsed`	`Boolean`	Specifies whether formatted output contains grouping separators.
`integerOnly`	`Boolean`	Specifies whether only the integer part of the value will be parsed.
`locale`	`String` or `Locale`	`Locale` whose number styles are used to format or parse data.
`maxFractionDigits`	`int`	Maximum number of digits formatted in the fractional part of the output.
`maxIntegerDigits`	`int`	Maximum number of digits formatted in the integer part of the output.
`minFractionDigits`	`int`	Minimum number of digits formatted in the fractional part of the output.
`minIntegerDigits`	`int`	Minimum number of digits formatted in the integer part of the output.
`pattern`	`String`	Custom formatting pattern that determines how the number string is formatted and parsed.

Attribute	Type	Description
type	String	Specifies whether the string value is parsed and formatted as a number, currency, or percentage. If not specified, number is used.

Registering Listeners on Components

An application developer can implement listeners as classes or as managed bean methods. If a listener is a managed bean method, the page author references the method from either the component's valueChangeListener attribute or its actionListener attribute. If the listener is a class, the page author can reference the listener from either an f:valueChangeListener tag or an f:actionListener tag and nest the tag inside the component tag to register the listener on the component.

Referencing a Method That Handles an Action Event and Referencing a Method That Handles a Value-Change Event explain how a page author uses the valueChangeListener and actionListener attributes to reference managed bean methods that handle events.

This section explains how to register a NameChanged value-change listener and a BookChange action listener implementation on components. The Duke's Bookstore case study includes both of these listeners.

Registering a Value-Change Listener on a Component

A page author can register a ValueChangeListener implementation on a component that implements EditableValueHolder by nesting an f:valueChangeListener tag within the component's tag on the page. The f:valueChangeListener tag supports the attributes shown in Attributes for the f:valueChangeListener Tag, one of which must be used.

Attributes for the f:valueChangeListener Tag

Attribute	Description
type	References the fully qualified class name of a ValueChangeListener implementation. Can accept a literal or a value expression.
binding	References an object that implements ValueChangeListener. Can accept only a value expression, which must point to a managed bean property that accepts and returns a ValueChangeListener implementation.

The following example shows a value-change listener registered on a component:

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
```

```
            requiredMessage="#{bundle.ReqCustomerName}">
    <f:valueChangeListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

In the example, the core tag `type` attribute specifies the custom `NameChanged` listener as the `ValueChangeListener` implementation registered on the `name` component.

After this component tag is processed and local values have been validated, its corresponding component instance will queue the `ValueChangeEvent` associated with the specified `ValueChangeListener` to the component.

The `binding` attribute is used to bind a `ValueChangeListener` implementation to a managed bean property. This attribute works in a similar way to the `binding` attribute supported by the standard converter tags. See Binding Component Values and Instances to Managed Bean Properties for more information.

Registering an Action Listener on a Component

A page author can register an `ActionListener` implementation on a command component by nesting an `f:actionListener` tag within the component's tag on the page. Similarly to the `f:valueChangeListener` tag, the `f:actionListener` tag supports both the `type` and `binding` attributes. One of these attributes must be used to reference the action listener.

Here is an example of an `h:commandLink` tag that references an `ActionListener` implementation:

```
<h:commandLink id="Duke" action="bookstore">
    <f:actionListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.LinkBookChangeListener" />
    <h:outputText value="#{bundle.Book201}"/>
</h:commandLink>
```

The `type` attribute of the `f:actionListener` tag specifies the fully qualified class name of the `ActionListener` implementation. Similarly to the `f:valueChangeListener` tag, the `f:actionListener` tag also supports the `binding` attribute. See Binding Converters, Listeners, and Validators to Managed Bean Properties for more information about binding listeners to managed bean properties.

In addition to the `actionListener` tag that allows you register a custom listener onto a component, the core tag library includes the `f:setPropertyActionListener` tag. You use this tag to register a special action listener onto the `ActionSource` instance associated with a component. When the component is activated, the listener will store the object referenced by the tag's `value` attribute into the object referenced by the tag's `target` attribute.

The `bookcatalog.xhtml` page of the Duke's Bookstore application uses `f:setPropertyActionListener` with two components: the `h:commandLink` component used to link to the `bookdetails.xhtml` page and the `h:commandButton` component used to add a book to the cart:

```
<h:dataTable id="books"
```

```
    value="#{store.books}"
    var="book"
    headerClass="list-header"
    styleClass="list-background"
    rowClasses="list-row-even, list-row-odd"
    border="1"
    summary="#{bundle.BookCatalog}" >
    ...
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemTitle}"/>
        </f:facet>
        <h:commandLink action="#{catalog.details}"
                       value="#{book.title}">
            <f:setPropertyActionListener target="#{requestScope.book}"
                                         value="#{book}"/>
        </h:commandLink>
    </h:column>
    ...
    <h:column>
        <f:facet name="header">
            <h:outputText value="#{bundle.CartAdd}"/>
        </f:facet>
        <h:commandButton id="add"
                         action="#{catalog.add}"
                         value="#{bundle.CartAdd}">
            <f:setPropertyActionListener target="#{requestScope.book}"
                                         value="#{book}"/>
        </h:commandButton>
    </h:column>
    ...
</h:dataTable>
```

The `h:commandLink` and `h:commandButton` tags are within an `h:dataTable` tag, which iterates over the list of books. The `var` attribute refers to a single book in the list of books.

The object referenced by the `var` attribute of an `h:dataTable` tag is in page scope. However, in this case you need to put this object into request scope so that when the user activates the `commandLink` component to go to `bookdetails.xhtml` or activates the `commandButton` component to go to `bookcatalog.xhtml`, the book data is available to those pages. Therefore, the `f:setPropertyActionListener` tag is used to set the current book object into request scope when the `commandLink` or `commandButton` component is activated.

In the preceding example, the `f:setPropertyActionListener` tag's `value` attribute references the `book` object. The `f:setPropertyActionListener` tag's `target` attribute references the value expression `requestScope.book`, which is where the `book` object referenced by the `value` attribute is stored when the `commandLink` or the `commandButton` component is activated.

Using the Standard Validators

Jakarta Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component's data. The Validator Classes lists all the standard validator classes and the tags that allow you to use the validators from the page.

The Validator Classes

Validator Class	Tag	Function
`BeanValidator`	`validateBean`	Registers a bean validator for the component.
`BeanValidator`	`validateWholeBean`	Allows cross-field validation by enabling class-level bean validation on CDI-based backing beans.
`DoubleRangeValidator`	`validateDoubleRange`	Checks whether the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.
`LengthValidator`	`validateLength`	Checks whether the length of a component's local value is within a certain range. The value must be a `java.lang.String`.
`LongRangeValidator`	`validateLongRange`	Checks whether the local value of a component is within a certain range. The value must be any numeric type or `String` that can be converted to a `long`.
`RegexValidator`	`validateRegex`	Checks whether the local value of a component is a match against a regular expression from the `java.util.regex` package.
`RequiredValidator`	`validateRequired`	Ensures that the local value is not empty on an `EditableValueHolder` component.

All of these validator classes implement the `Validator` interface. Component writers and application developers can also implement this interface to define their own set of constraints for a component's value.

Similar to the standard converters, each of these validators has one or more standard error messages associated with it. If you have registered one of these validators onto a component on your page and the validator is unable to validate the component's value, the validator's error message will display on the page. For example, the error message that displays when the component's value exceeds the maximum value allowed by `LongRangeValidator` is as follows:

```
{1}: Validation Error: Value is greater than allowable maximum of "{0}"
```

In this case, the `{1}` substitution parameter is replaced by the component's label or `id`, and the `{0}` substitution parameter is replaced with the maximum value allowed by the validator.

See Displaying Error Messages with the h:message and h:messages Tags for information on how to display validation error messages on the page when validation fails.

Instead of using the standard validators, you can use Bean Validation to validate data. If you specify bean validation constraints on your managed bean properties, the constraints are automatically

placed on the corresponding fields on your applications web pages. See [beanvalidation:bean-validation::bean-validation:::_introduction_to_jakarta_bean_validation] for more information. You do not need to specify the `validateBean` tag to use Bean Validation, but the tag allows you to use more advanced Bean Validation features. For example, you can use the `validationGroups` attribute of the tag to specify constraint groups.

You can also create and register custom validators, although Bean Validation has made this feature less useful. For details, see Creating and Using a Custom Validator.

Validating a Component's Value

To validate a component's value using a particular validator, you need to register that validator on the component. You can do this in one of the following ways.

- Nest the validator's corresponding tag (shown in The Validator Classes) inside the component's tag. Using Validator Tags explains how to use the `validateLongRange` tag. You can use the other standard tags in the same way.
- Refer to a method that performs the validation from the component tag's `validator` attribute.
- Nest a validator tag inside the component tag, and use either the validator tag's `validatorId` attribute or its `binding` attribute to refer to the validator.

See Referencing a Method That Performs Validation for more information on using the `validator` attribute.

The `validatorId` attribute works similarly to the `converterId` attribute of the `converter` tag, as described in Converting a Component's Value.

Keep in mind that validation can be performed only on components that implement `EditableValueHolder`, because these components accept values that can be validated.

Using Validator Tags

The following example shows how to use the `f:validateLongRange` validator tag on an input component named `quantity`:

```
<h:inputText id="quantity" size="4" value="#{item.quantity}">
    <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```

This tag requires the user to enter a number that is at least 1. The `validateLongRange` tag also has a `maximum` attribute, which sets a maximum value for the input.

The attributes of all the standard validator tags accept EL value expressions. This means that the attributes can reference managed bean properties rather than specify literal values. For example, the `f:validateLongRange` tag in the preceding example can reference managed bean properties called `minimum` and `maximum` to get the minimum and maximum values acceptable to the validator implementation, as shown in this snippet from the `guessnumber-faces` example:

```
<h:inputText id="userNo"
             title="Type a number from 0 to 10:"
             value="#{userNumberBean.userNumber}">
    <f:validateLongRange minimum="#{userNumberBean.minimum}"
                         maximum="#{userNumberBean.maximum}"/>
</h:inputText>
```

The following `f:validateRegex` tag shows how you might ensure that a password is from 4 to 10 characters long and contains at least one digit, at least one lowercase letter, and at least one uppercase letter:

```
<f:validateRegex pattern="((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{4,10})"
                 for="passwordVal"/>
```

Referencing a Managed Bean Method

A component tag has a set of attributes for referencing managed bean methods that can perform certain functions for the component associated with the tag. These attributes are summarized in Component Tag Attributes That Reference Managed Bean Methods.

Component Tag Attributes That Reference Managed Bean Methods

Attribute	Function
`action`	Refers to a managed bean method that performs navigation processing for the component and returns a logical outcome `String`
`actionListener`	Refers to a managed bean method that handles action events
`validator`	Refers to a managed bean method that performs validation on the component's value
`valueChangeListener`	Refers to a managed bean method that handles value-change events

Only components that implement `ActionSource` can use the `action` and `actionListener` attributes. Only components that implement `EditableValueHolder` can use the `validator` or `valueChangeListener` attributes.

The component tag refers to a managed bean method using a method expression as a value of one of the attributes. The method referenced by an attribute must follow a particular signature, which is defined by the tag attribute's definition in the Jakarta Faces Facelets Tag Library documentation. For example, the definition of the `validator` attribute of the `inputText` tag is the following:

```
void validate(jakarta.faces.context.FacesContext,
```

```
            jakarta.faces.component.UIComponent, java.lang.Object)
```

The following sections give examples of how to use the attributes.

Referencing a Method That Performs Navigation

If your page includes a component, such as a button or a link, that causes the application to navigate to another page when the component is activated, the tag corresponding to this component must include an `action` attribute. This attribute does one of the following:

- Specifies a logical outcome `String` that tells the application which page to access next
- References a managed bean method that performs some processing and returns a logical outcome `String`

The following example shows how to reference a navigation method:

```
<h:commandButton value="#{bundle.Submit}"
                 action="#{cashierBean.submit}" />
```

See Writing a Method to Handle Navigation for information on how to write such a method.

Referencing a Method That Handles an Action Event

If a component on your page generates an action event, and if that event is handled by a managed bean method, you refer to the method by using the component's `actionListener` attribute.

The following example shows how such a method could be referenced:

```
<h:commandLink id="Duke" action="bookstore"
               actionListener="#{actionBean.chooseBookFromLink}">
```

The `actionListener` attribute of this component tag references the `chooseBookFromLink` method using a method expression. The `chooseBookFromLink` method handles the event when the user clicks the link rendered by this component. See Writing a Method to Handle an Action Event for information on how to write such a method.

Referencing a Method That Performs Validation

If the input of one of the components on your page is validated by a managed bean method, refer to the method from the component's tag by using the `validator` attribute.

The following simplified example from The guessnumber-cdi CDI Example shows how to reference a method that performs validation on `inputGuess`, an input component:

```
<h:inputText id="inputGuess"
    value="#{userNumberBean.userNumber}"
    required="true" size="3"
```

```
    disabled="#{userNumberBean.number eq userNumberBean.userNumber ...}"
    validator="#{userNumberBean.validateNumberRange}">
</h:inputText>
```

The managed bean method `validateNumberRange` verifies that the input value is within the valid range, which changes each time another guess is made. See Writing a Method to Perform Validation for information on how to write such a method.

Referencing a Method That Handles a Value-Change Event

If you want a component on your page to generate a value-change event and you want that event to be handled by a managed bean method instead of a `ValueChangeListener` implementation, you refer to the method by using the component's `valueChangeListener` attribute:

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             valueChangeListener="#{cashierBean.processValueChange}" />
</h:inputText>
```

The `valueChangeListener` attribute of this component tag references the `processValueChange` method of `CashierBean` by using a method expression. The `processValueChange` method handles the event of a user entering a name in the input field rendered by this component.

Writing a Method to Handle a Value-Change Event describes how to implement a method that handles a `ValueChangeEvent`.

Developing with Jakarta Faces Technology



This section was last updated for Jakarta EE Version 9.1

This chapter provides an overview of managed beans and explains how to write methods and properties of managed beans that are used by a Jakarta Faces application. This chapter also introduces the Bean Validation feature.

Managed Beans in Jakarta Faces Technology

A typical Jakarta Faces application includes one or more managed beans, each of which can be associated with the components used in a particular page. This section introduces the basic concepts of creating, configuring, and using managed beans in an application.



[web:faces-page::faces-page:::_using_jakarta_faces_technology_in_web_pages] and [web:faces-page-core::faces-page-core:::_using_converters_listeners_and_validators] show how to add components to a page and connect them to server-side objects by using component tags and core tags. These chapters also show how to provide additional functionality to the components through converters, listeners, and validators. Developing a Jakarta

> Faces application also involves the task of programming the server-side objects: managed beans, converters, event handlers, and validators.

Creating a Managed Bean

A managed bean is created with a constructor with no arguments, a set of properties, and a set of methods that perform functions for a component. Each of the managed bean properties can be bound to one of the following:

- A component value
- A component instance
- A converter instance
- A listener instance
- A validator instance

The most common functions that managed bean methods perform include the following:

- Validating a component's data
- Handling an event fired by a component
- Performing processing to determine the next page to which the application must navigate

As with all JavaBeans components, a property consists of a private data field and a set of accessor methods, as shown by this code:

```
private Integer userNumber = null;
...
public void setUserNumber(Integer user_number) {
    userNumber = user_number;
}
public Integer getUserNumber() {
    return userNumber;
}
```

When bound to a component's value, a bean property can be any of the basic primitive and numeric types or any Java object type for which the application has access to an appropriate converter. For example, a property can be of type `java.util.Date` if the application has access to a converter that can convert the `Date` type to a `String` and back again. See Writing Bean Properties for information on which types are accepted by which component tags.

When a bean property is bound to a component instance, the property's type must be the same as the component object. For example, if a `jakarta.faces.component.UISelectBoolean` component is bound to the property, the property must accept and return a `UISelectBoolean` object. Likewise, if the property is bound to a converter, validator, or listener instance, the property must be of the appropriate converter, validator, or listener type.

For more information on writing beans and their properties, see Writing Bean Properties.

Using the EL to Reference Managed Beans

To bind component values and objects to managed bean properties or to reference managed bean methods from component tags, page authors use the Expression Language syntax. As explained in Overview of the EL, the following are some of the features that the EL offers:

- Deferred evaluation of expressions
- The ability to use a value expression to both read and write data
- Method expressions

Deferred evaluation of expressions is important because the Jakarta Faces lifecycle is split into several phases in which component event handling, data conversion and validation, and data propagation to external objects are all performed in an orderly fashion. The implementation must be able to delay the evaluation of expressions until the proper phase of the lifecycle has been reached. Therefore, the implementation's tag attributes always use deferred-evaluation syntax, which is distinguished by the `#{}` delimiter.

To store data in external objects, almost all Jakarta Faces tag attributes use lvalue expressions, which are expressions that allow both getting and setting data on external objects.

Finally, some component tag attributes accept method expressions that reference methods that handle component events or validate or convert component data.

To illustrate a Jakarta Faces tag using the EL, the following tag references a method that validates user input:

```
<h:inputText id="inputGuess"
    value="#{userNumberBean.userNumber}"
    required="true" size="3"
    disabled="#{userNumberBean.number eq userNumberBean.userNumber ...}"
    validator="#{userNumberBean.validateNumberRange}">
</h:inputText>
```

This tag binds the `inputGuess` component's value to the `UserNumberBean.userNumber` managed bean property by using an lvalue expression. The tag uses a method expression to refer to the `UserNumberBean.validateNumberRange` method, which performs validation of the component's local value. The local value is whatever the user types into the field corresponding to this tag. This method is invoked when the expression is evaluated.

Nearly all Jakarta Faces tag attributes accept value expressions. In addition to referencing bean properties, value expressions can reference lists, maps, arrays, implicit objects, and resource bundles.

Another use of value expressions is to bind a component instance to a managed bean property. A page author does this by referencing the property from the `binding` attribute:

```
<h:outputLabel for="fanClub"
               rendered="false"
```

```
                binding="#{cashierBean.specialOfferText}"
                value="#{bundle.DukeFanClub}"/>
</h:outputLabel>
```

In addition to using expressions with the standard component tags, you can configure your custom component properties to accept expressions by creating `jakarta.el.ValueExpression` or `jakarta.el.MethodExpression` instances for them.

For information on the EL, see Expression Language.

For information on referencing managed bean methods from component tags, see Referencing a Managed Bean Method.

Writing Bean Properties

As explained in Managed Beans in Jakarta Faces Technology, a managed bean property can be bound to one of the following items:

- A component value
- A component instance
- A converter implementation
- A listener implementation
- A validator implementation

These properties follow the conventions of JavaBeans components (also called beans). For more information on JavaBeans components, see the JavaBeans Tutorial at https://docs.oracle.com/javase/tutorial/javabeans/index.html.

The component's tag binds the component's value to a managed bean property by using its `value` attribute and binds the component's instance to a managed bean property by using its `binding` attribute. Likewise, all the converter, listener, and validator tags use their `binding` attributes to bind their associated implementations to managed bean properties. See Binding Component Values and Instances to Managed Bean Properties and Binding Converters, Listeners, and Validators to Managed Bean Properties for more information.

To bind a component's value to a managed bean property, the type of the property must match the type of the component's value to which it is bound. For example, if a managed bean property is bound to a `UISelectBoolean` component's value, the property should accept and return a `boolean` value or a `Boolean` wrapper `Object` instance.

To bind a component instance to a managed bean property, the property must match the type of component. For example, if a managed bean property is bound to a `UISelectBoolean` instance, the property should accept and return a `UISelectBoolean` value.

Similarly, to bind a converter, listener, or validator implementation to a managed bean property, the property must accept and return the same type of converter, listener, or validator object. For example, if you are using the `convertDateTime` tag to bind a `jakarta.faces.convert.DateTimeConverter` to a property, that property must accept and return a `DateTimeConverter` instance.

The rest of this section explains how to write properties that can be bound to component values, to component instances for the component objects described in Adding Components to a Page Using HTML Tag Library Tags, and to converter, listener, and validator implementations.

Writing Properties Bound to Component Values

To write a managed bean property that is bound to a component's value, you must match the property type to the component's value.

Acceptable Types of Component Values lists the `jakarta.faces.component` classes and the acceptable types of their values.

Acceptable Types of Component Values

Component Class	Acceptable Types of Component Values
`UIInput`, `UIOutput`, `UISelectItem`, `UISelectOne`	Any of the basic primitive and numeric types or any Java programming language object type for which an appropriate `jakarta.faces.convert.Converter` implementation is available
`UIData`	`array` of beans, `List` of beans, single bean, `java.sql.ResultSet`, `jakarta.servlet.jsp.jstl.sql.Result`, `javax.sql.RowSet`
`UISelectBoolean`	`boolean` or `Boolean`
`UISelectItems`	`java.lang.String`, `Collection`, `Array`, `Map`
`UISelectMany`	`array` or `List`, although elements of the `array` or `List` can be any of the standard types

When they bind components to properties by using the `value` attributes of the component tags, page authors need to ensure that the corresponding properties match the types of the components' values.

UIInput and UIOutput Properties

The `UIInput` and `UIOutput` component classes are represented by the component tags that begin with `h:input` and `h:output`, respectively (for example, `h:inputText` and `h:outputText`).

In the following example, an `h:inputText` tag binds the `name` component to the `name` property of a managed bean called `CashierBean`.

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
    ...>
</h:inputText>
```

The following code snippet from the managed bean `CashierBean` shows the bean property type bound by the preceding component tag:

```
protected String name = null;
```

```
public void setName(String name) {
    this.name = name;
}
public String getName() {
    return this.name;
}
```

As described in Using the Standard Converters, to convert the value of an input or output component you can either apply a converter or create the bean property bound to the component with the matching type. Here is the example tag, from Using DateTimeConverter, that displays the date on which items will be shipped.

```
<h:outputText value="#{cashierBean.shipDate}">
    <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

The bean property represented by this tag must have a type of `java.util.Date`. The following code snippet shows the `shipDate` property, from the managed bean `CashierBean`, that is bound by the tag's value in the preceding example:

```
private Date shipDate;

public Date getShipDate() {
    return this.shipDate;
}
public void setShipDate(Date shipDate) {
    this.shipDate = shipDate;
}
```

UIData Properties

The `UIData` component class is represented by the `h:dataTable` component tag.

`UIData` components must be bound to one of the managed bean property types listed in Acceptable Types of Component Values. Data components are discussed in Using Data-Bound Table Components. Here is part of the start tag of `dataTable` from that section:

```
<h:dataTable id="items"
    ...
    value="#{cart.items}"
    ...
    var="item">
```

The value expression points to the `items` property of a shopping cart bean named `cart`. The `cart` bean maintains a map of `ShoppingCartItem` beans.

The `getItems` method from the `cart` bean populates a `List` with `ShoppingCartItem` instances that are saved in the `items` map when the customer adds books to the cart, as shown in the following code segment:

```
public synchronized List<ShoppingCartItem> getItems() {
    List<ShoppingCartItem> results = new ArrayList<ShoppingCartItem>();
    results.addAll(this.items.values());
    return results;
}
```

All the components contained in the `UIData` component are bound to the properties of the `cart` bean that is bound to the entire `UIData` component. For example, here is the `h:outputText` tag that displays the book title in the table:

```
<h:commandLink action="#{showcart.details}">
    <h:outputText value="#{item.item.title}"/>
</h:commandLink>
```

The title is actually a link to the `bookdetails.xhtml` page. The `h:outputText` tag uses the value expression `#{item.item.title}` to bind its `UIOutput` component to the `title` property of the `Book` entity. The first item in the expression is the `ShoppingCartItem` instance that the `h:dataTable` tag is referencing while rendering the current row. The second item in expression refers to the `item` property of `ShoppingCartItem`, which returns an `Object` (in this case, a `Book`). The `title` part of the expression refers to the `title` property of `Book`. The value of the `UIOutput` component corresponding to this tag is bound to the `title` property of the `Book` entity:

```
private String title;
...
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
```

The UIData component (and UIRepeat) supports the `Map` and `Iterable` interfaces, as well as custom types.

For UIData and UIRepeat, the supported types are:

- `null` (becomes empty list)
- `jakarta.faces.model.DataMode`
- `java.util.List`
- `java.lang.Object`[]

- `java.sql.ResultSet`
- `jakarta.servlet.jsp.jstl.sql.Result`
- `java.util.Collection`
- `java.lang.Iterable`
- `java.util.Map`
- `java.lang.Object` (becomes ScalarDataModel)

UISelectBoolean Properties

The `UISelectBoolean` component class is represented by the component tag `h:selectBooleanCheckbox`.

Managed bean properties that hold a `UISelectBoolean` component's data must be of `boolean` or `Boolean` type. The example `selectBooleanCheckbox` tag from the section Displaying Components for Selecting One Value binds a component to a property. The following example shows a tag that binds a component value to a `boolean` property:

```
<h:selectBooleanCheckbox title="#{bundle.receiveEmails}"
                         value="#{custFormBean.receiveEmails}">
</h:selectBooleanCheckbox>
<h:outputText value="#{bundle.receiveEmails}">
```

Here is an example property that can be bound to the component represented by the example tag:

```
private boolean receiveEmails = false;
...
public void setReceiveEmails(boolean receiveEmails) {
    this.receiveEmails = receiveEmails;
}
public boolean getReceiveEmails() {
    return receiveEmails;
}
```

UISelectMany Properties

The `UISelectMany` component class is represented by the component tags that begin with `h:selectMany` (for example, `h:selectManyCheckbox` and `h:selectManyListbox`).

Because a `UISelectMany` component allows a user to select one or more items from a list of items, this component must map to a bean property of type `List` or `array`. This bean property represents the set of currently selected items from the list of available items.

The following example of the `selectManyCheckbox` tag comes from Displaying Components for Selecting Multiple Values:

```
<h:selectManyCheckbox id="newslettercheckbox"
                      layout="pageDirection"
```

```
                    value="#{cashierBean.newsletters}">
    <f:selectItems value="#{cashierBean.newsletterItems}"/>
</h:selectManyCheckbox>
```

Here is the bean property that maps to the `value` of the `selectManyCheckbox` tag from the preceding example:

```
private String[] newsletters;

public void setNewsletters(String[] newsletters) {
    this.newsletters = newsletters;
}
public String[] getNewsletters() {
    return this.newsletters;
}
```

The `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectMany` component. See UISelectItems Properties for information on writing the bean properties for the `UISelectItem` and `UISelectItems` components.

UISelectOne Properties

The `UISelectOne` component class is represented by the component tags that begin with `h:selectOne` (for example, `h:selectOneRadio` and `h:selectOneListbox`).

`UISelectOne` properties accept the same types as `UIInput` and `UIOutput` properties, because a `UISelectOne` component represents the single selected item from a set of items. This item can be any of the primitive types and anything else for which you can apply a converter.

Here is an example of the `h:selectOneMenu` tag from Displaying a Menu Using the h:selectOneMenu Tag:

```
<h:selectOneMenu id="shippingOption"
                 required="true"
                 value="#{cashierBean.shippingOption}">
    <f:selectItem itemValue="2"
                  itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem itemValue="5"
                  itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem itemValue="7"
                  itemLabel="#{bundle.SaverShip}"/>
 </h:selectOneMenu>
```

Here is the bean property corresponding to this tag:

```
private String shippingOption = "2";
```

```
public void setShippingOption(String shippingOption) {
    this.shippingOption = shippingOption;
}
public String getShippingOption() {
    return this.shippingOption;
}
```

Note that `shippingOption` represents the currently selected item from the list of items in the `UISelectOne` component.

The `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectOne` component. This is explained in Displaying a Menu Using the h:selectOneMenu Tag.

For information on how to write the managed bean properties for the `UISelectItem` and `UISelectItems` components, see UISelectItems Properties.

UISelectItem Properties

A `UISelectItem` component represents a single value in a set of values in a `UISelectMany` or a `UISelectOne` component. A `UISelectItem` component must be bound to a managed bean property of type `jakarta.faces.model.SelectItem`. A `SelectItem` object is composed of an `Object` representing the value along with two `Strings` representing the label and the description of the `UISelectItem` object.

The example `selectOneMenu` tag from UISelectOne Properties contains `selectItem` tags that set the values of the list of items in the page. Here is an example of a bean property that can set the values for this list in the bean:

```
SelectItem itemOne = null;

SelectItem getItemOne(){
    return itemOne;
}
void setItemOne(SelectItem item) {
    itemOne = item;
}
```

UISelectItems Properties

`UISelectItems` components are children of `UISelectMany` and `UISelectOne` components. Each `UISelectItems` component is composed of a set of either `UISelectItem` instances or any collection of objects, such as an array, a list, or even POJOs.

The following code snippet from `CashierBean` shows how to write the properties for `selectItems` tags containing `SelectItem` instances.

```
private String[] newsletters;
private static final SelectItem[] newsletterItems = {
    new SelectItem("Duke's Quarterly"),
    new SelectItem("Innovator's Almanac"),
```

```
    new SelectItem("Duke's Diet and Exercise Journal"),
    new SelectItem("Random Ramblings")
};
...
public void setNewsletters(String[] newsletters) {
    this.newsletters = newsletters;
}

public String[] getNewsletters() {
    return this.newsletters;
}

public SelectItem[] getNewsletterItems() {
    return newsletterItems;
}
```

Here, the `newsletters` property represents the `SelectItems` object, whereas the `newsletterItems` property represents a static array of `SelectItem` objects. The `SelectItem` class has several constructors; in this example, the first argument is an `Object` representing the value of the item, whereas the second argument is a `String` representing the label that appears in the `UISelectMany` component on the page.

Writing Properties Bound to Component Instances

A property bound to a component instance returns and accepts a component instance rather than a component value. The following components bind a component instance to a managed bean property:

```
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

The `selectBooleanCheckbox` tag renders a check box and binds the `fanClub` `UISelectBoolean` component to the `specialOffer` property of `CashierBean`. The `outputLabel` tag binds the value of the `value` attribute, which represents the check box's label, to the `specialOfferText` property of `CashierBean`. If the user orders more than $100 worth of books and clicks the Submit button, the `submit` method of `CashierBean` sets both components' `rendered` properties to `true`, causing the check box and label to display when the page is re-rendered.

Because the components corresponding to the example tags are bound to the managed bean properties, these properties must match the components' types. This means that the `specialOfferText` property must be of type `UIOutput`, and the `specialOffer` property must be of type `UISelectBoolean`:

```
UIOutput specialOfferText = null;
UISelectBoolean specialOffer = null;

public UIOutput getSpecialOfferText() {
    return this.specialOfferText;
}
public void setSpecialOfferText(UIOutput specialOfferText) {
    this.specialOfferText = specialOfferText;
}

public UISelectBoolean getSpecialOffer() {
    return this.specialOffer;
}
public void setSpecialOffer(UISelectBoolean specialOffer) {
    this.specialOffer = specialOffer;
}
```

For more general information on component binding, see Managed Beans in Jakarta Faces Technology.

For information on how to reference a managed bean method that performs navigation when a button is clicked, see Referencing a Method That Performs Navigation.

For more information on writing managed bean methods that handle navigation, see Writing a Method to Handle Navigation.

Writing Properties Bound to Converters, Listeners, or Validators

All the standard converter, listener, and validator tags included with Jakarta Faces technology support binding attributes that allow you to bind converter, listener, or validator implementations to managed bean properties.

The following example shows a standard `convertDateTime` tag using a value expression with its `binding` attribute to bind the `jakarta.faces.convert.DateTimeConverter` instance to the `convertDate` property of `LoginBean`:

```
<h:inputText value="#{loginBean.birthDate}">
    <f:convertDateTime binding="#{loginBean.convertDate}" />
</h:inputText>
```

The `convertDate` property must therefore accept and return a `DateTimeConverter` object, as shown here:

```
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
       ...
    return convertDate;
}
```

```
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

Because the converter is bound to a managed bean property, the managed bean property can modify the attributes of the converter or add new functionality to it. In the case of the preceding example, the property sets the date pattern that the converter uses to parse the user's input into a `Date` object.

The managed bean properties that are bound to validator or listener implementations are written in the same way and have the same general purpose.

Writing Managed Bean Methods

Methods of a managed bean can perform several application-specific functions for components on the page. These functions include

- Performing processing associated with navigation
- Handling action events
- Performing validation on the component's value
- Handling value-change events

Why Use Managed Beans

By using a managed bean to perform these functions, you eliminate the need to implement the `jakarta.faces.validator.Validator` interface to handle the validation or one of the listener interfaces to handle events. Also, by using a managed bean instead of a `Validator` implementation to perform validation, you eliminate the need to create a custom tag for the `Validator` implementation.

In general, it is good practice to include these methods in the same managed bean that defines the properties for the components referencing these methods. The reason for doing so is that the methods might need to access the component's data to determine how to handle the event or to perform the validation associated with the component.

The following sections explain how to write various types of managed bean methods.

Writing a Method to Handle Navigation

An action method, a managed bean method that handles navigation processing, must be a public method that takes no parameters and returns an `Object`, which is the logical outcome that the navigation system uses to determine the page to display next. This method is referenced using the component tag's `action` attribute.

The following action method is from the managed bean `CashierBean`, which is invoked when a user clicks the Submit button on the page. If the user has ordered more than $100 worth of books, this method sets the `rendered` properties of the `fanClub` and `specialOffer` components to `true`, causing them to be displayed on the page the next time that page is rendered.

After setting the components' `rendered` properties to `true`, this method returns the logical outcome `null`. This causes the Jakarta Faces implementation to re-render the page without creating a new view of the page, retaining the customer's input. If this method were to return `purchase`, which is the logical outcome to use to advance to a payment page, the page would re-render without retaining the customer's input. In this case, you want to re-render the page without clearing the data.

If the user does not purchase more than $100 worth of books or if the `thankYou` component has already been rendered, the method returns `bookreceipt`. The Jakarta Faces implementation loads the `bookreceipt.xhtml` page after this method returns:

```
public String submit() {
    ...
    if ((cart().getTotal() > 100.00) && !specialOffer.isRendered()) {
        specialOfferText.setRendered(true);
        specialOffer.setRendered(true);
        return null;
    } else if (specialOffer.isRendered() && !thankYou.isRendered()) {
        thankYou.setRendered(true);
        return null;
    } else {
        ...
        cart.clear();
        return ("bookreceipt");
    }
}
```

Typically, an action method will return a `String` outcome, as shown in the preceding example. Alternatively, you can define an `Enum` class that encapsulates all possible outcome strings and then make an action method return an `enum` constant, which represents a particular `String` outcome defined by the `Enum` class.

The following example uses an `Enum` class to encapsulate all logical outcomes:

```
public enum Navigation  {
    main, accountHist, accountList, atm, atmAck, transferFunds,
     transferAck, error
}
```

When it returns an outcome, an action method uses the dot notation to reference the outcome from the `Enum` class:

```
public Object submit(){
    ...
    return Navigation.accountHist;
}
```

The section Referencing a Method That Performs Navigation explains how a component tag references this method. The section Writing Properties Bound to Component Instances explains how to write the bean properties to which the components are bound.

Writing a Method to Handle an Action Event

A managed bean method that handles an action event must be a public method that accepts an action event and returns `void`. This method is referenced using the component tag's `actionListener` attribute. Only components that implement `jakarta.faces.component.ActionSource` can refer to this method.

In the following example, a method from a managed bean named `ActionBean` processes the event of a user clicking one of the links on the page:

```
public void chooseBookFromLink(ActionEvent event) {
    String current = event.getComponent().getId();
    FacesContext context = FacesContext.getCurrentInstance();
    String bookId = books.get(current);
    context.getExternalContext().getSessionMap().put("bookId", bookId);
}
```

This method gets the component that generated the event from the event object; then it gets the component's ID, which is a code for the book. The method matches the code against a `HashMap` object that contains the book codes and corresponding book ID values. Finally, the method sets the book ID by using the selected value from the `HashMap` object.

Referencing a Method That Handles an Action Event explains how a component tag references this method.

Writing a Method to Perform Validation

Instead of implementing the `jakarta.faces.validator.Validator` interface to perform validation for a component, you can include a method in a managed bean to take care of validating input for the component. A managed bean method that performs validation must accept a `jakarta.faces.context.FacesContext`, the component whose data must be validated, and the data to be validated, just as the `validate` method of the `Validator` interface does. A component refers to the managed bean method by using its `validator` attribute. Only values of `UIInput` components or values of components that extend `UIInput` can be validated.

Here is an example of a managed bean method that validates user input, from The guessnumber-cdi CDI Example:

```
public void validateNumberRange(FacesContext context,
                                UIComponent toValidate,
                                Object value) {
    if (remainingGuesses <= 0) {
        ((UIInput) toValidate).setValid(false);
        FacesMessage message = new FacesMessage("No guesses left!");
        context.addMessage(toValidate.getClientId(context), message);
```

```
        return;
    }

    int input = (Integer) value;
    if (input < minimum || input> maximum) {
        ((UIInput) toValidate).setValid(false);

        FacesMessage message = new FacesMessage("Invalid guess");
        context.addMessage(toValidate.getClientId(context), message);
    }
}
```

The `validateNumberRange` method performs two different validations.

- If the user has run out of guesses, the method sets the `valid` property of the `UIInput` component to `false`. Then it queues a message onto the `FacesContext` instance, associating the message with the component ID, and returns.
- If the user has some remaining guesses, the method then retrieves the local value of the component. If the input value is outside the allowable range, the method again sets the `valid` property of the `UIInput` component to `false`, queues a different message on the `FacesContext` instance, and returns.

See Referencing a Method That Performs Validation for information on how a component tag references this method.

Writing a Method to Handle a Value-Change Event

A managed bean that handles a value-change event must use a public method that accepts a value-change event and returns `void`. This method is referenced using the component's `valueChangeListener` attribute. This section explains how to write a managed bean method to replace the `jakarta.faces.event.ValueChangeListener` implementation.

The following example tag comes from Registering a Value-Change Listener on a Component, where the `h:inputText` tag with the `id` of `name` has a `ValueChangeListener` instance registered on it. This `ValueChangeListener` instance handles the event of entering a value in the field corresponding to the component. When the user enters a value, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the `ValueChangeListener` class is invoked:

```
<h:inputText id="name"
             size="30"
             value="#{cashierBean.name}"
             required="true"
             requiredMessage="#{bundle.ReqCustomerName}">
     <f:valueChangeListener
         type="ee.jakarta.tutorial.dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

Instead of implementing `ValueChangeListener`, you can write a managed bean method to handle this

event. To do this, you move the `processValueChange(ValueChangeEvent)` method from the `ValueChangeListener` class, called `NameChanged`, to your managed bean.

Here is the managed bean method that processes the event of entering a value in the `name` field on the page:

```
public void processValueChange(ValueChangeEvent event)
        throws AbortProcessingException {
    if (null != event.getNewValue()) {
        FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put("name", event.getNewValue());
    }
}
```

To make this method handle the `ValueChangeEvent` generated by an input component, reference this method from the component tag's `valueChangeListener` attribute. See Referencing a Method That Handles a Value-Change Event for more information.

Using Ajax with Jakarta Faces Technology



This section was last updated for Jakarta EE Version 9.1

This chapter describes using Ajax functionality in Jakarta Faces web applications. Ajax is an acronym for Asynchronous JavaScript and XML, a group of web technologies that enable creation of dynamic and highly responsive web applications. Using Ajax, web applications can retrieve content from the server without interfering with the display on the client. In the Jakarta EE platform, Jakarta Faces technology provides built-in support for Ajax.

Overview of Ajax

Early web applications were created mostly as static web pages. When a static web page is updated by a client, the entire page has to reload to reflect the update. In effect, every update needs a page reload to reflect the change. Repetitive page reloads can result in excessive network access and can impact application performance. Technologies such as Ajax were created to overcome these deficiencies.

Ajax refers to JavaScript and XML, technologies that are widely used for creating dynamic and asynchronous web content. While Ajax is not limited to JavaScript and XML technologies, more often than not they are used together by web applications. The focus of this tutorial is on using JavaScript based Ajax functionality in Jakarta Faces web applications.

JavaScript is a dynamic scripting language for web applications. It allows users to add enhanced functionality to user interfaces and allows web pages to interact with clients asynchronously. JavaScript runs mainly on the client side (as in a browser) and thereby reduces server access by clients.

When a JavaScript function sends an asynchronous request from the client to the server, the server sends back a response that is used to update the page's Document Object Model (DOM). This

response is often in the format of an XML document. The term Ajax refers to this interaction between the client and server.

The server response need not be in XML only; it can also be in other formats, such as JSON (see Introduction to JSON and https://www.json.org/). This tutorial does not focus on the response formats.

Ajax enables asynchronous and partial updating of web applications. Such functionality allows for highly responsive web pages that are rendered in near real time. Ajax-based web applications can access server and process information and can also retrieve data without interfering with the display and rendering of the current web page on a client (such as a browser).

Some of the advantages of using Ajax are as follows:

- Form data validation in real time, eliminating the need to submit the form for verification
- Enhanced functionality for web pages, such as user name and password prompts
- Partial update of the web content, avoiding complete page reloads

Using Ajax Functionality with Jakarta Faces Technology

Ajax functionality can be added to a Jakarta Faces application in one of the following ways:

- Adding the required JavaScript code to an application
- Using the built-in Ajax resource library

In earlier releases of the Jakarta EE platform, Jakarta Faces applications provided Ajax functionality by adding the necessary JavaScript to the web page. In the Jakarta EE platform, standard Ajax support is provided by a built-in JavaScript resource library.

With the support of this JavaScript resource library, Jakarta Faces standard UI components, such as buttons, labels, or text fields, can be enabled for Ajax functionality. You can also load this resource library and use its methods directly from within the managed bean code. The next sections of the tutorial describe the use of the built-in Ajax resource library.

In addition, because the Jakarta Faces technology component model can be extended, custom components can be created with Ajax functionality.

The tutorial examples include an Ajax version of the `guessnumber` application, `ajaxguessnumber`. See The ajaxguessnumber Example Application for more information.

The Ajax specific `f:ajax` tag and its attributes are explained in the next sections.

Using Ajax with Facelets

As mentioned in the previous section, Jakarta Faces technology supports Ajax by using a built-in JavaScript resource library that is provided as part of the Jakarta Faces core libraries. This built-in Ajax resource can be used in Jakarta Faces web applications in one of the following ways.

- By using the `f:ajax` tag along with another standard component in a Facelets application. This method adds Ajax functionality to any UI component without additional coding and

configuration.

- By using the JavaScript API method `faces.ajax.request()` directly within the Facelets application. This method provides direct access to Ajax methods and allows customized control of component behavior.
- By using the `<h:commandScript>` component to execute arbitrary server-side methods from a view. The component generates a JavaScript function with a given name that when invoked, in turn invokes, a given server-side method via Ajax.

Using the f:ajax Tag

The `f:ajax` tag is a Jakarta Faces core tag that provides Ajax functionality to any regular UI component when used in conjunction with that component. In the following example, Ajax behavior is added to an input component by including the `f:ajax` core tag:

```
<h:inputText value="#{bean.message}">
    <f:ajax />
</h:inputText>
```

In this example, although Ajax is enabled, the other attributes of the `f:ajax` tag are not defined. If an event is not defined, the default action for the component is performed. For the `inputText` component, when no `event` attribute is specified, the default event is `valueChange`. Attributes of the f:ajax Tag lists the attributes of the `f:ajax` tag and their default actions.

Attributes of the f:ajax Tag

Name	Type	Description
`disabled`	`jakarta.el.ValueExpression` that evaluates to a `Boolean`	A `Boolean` value that identifies the tag status. A value of `true` indicates that the Ajax behavior should not be rendered. A value of `false` indicates that the Ajax behavior should be rendered. The default value is `false`.
`event`	`jakarta.el.ValueExpression` that evaluates to a `String`	A `String` that identifies the type of event to which the Ajax action will apply. If specified, it must be one of the events supported by the component. If not specified, the default event (the event that triggers the Ajax request) is determined for the component. The default event is `action` for `jakarta.faces.component.ActionSource` components and `valueChange` for `jakarta.faces.component.EditableValueHolder` components.
`execute`	`jakarta.el.ValueExpression` that evaluates to an `Object`	A `Collection` that identifies a list of components to be executed on the server. If a literal is specified, it must be a space-delimited `String` of component identifiers and/or one of the keywords. If a `ValueExpression` is specified, it must refer to a property that returns a `Collection` of `String` objects. If not specified, the default value is `@this`.

Name	Type	Description
immediate	jakarta.el.ValueExpression that evaluates to a Boolean	A Boolean value that indicates whether inputs are to be processed early in the lifecycle. If true, behavior events generated from this behavior are broadcast during the Apply Request Values phase. Otherwise, the events will be broadcast during the Invoke Application phase.
listener	jakarta.el.MethodExpression	The name of the listener method that is called when a jakarta.faces.event.AjaxBehaviorEvent has been broadcast for the listener.
onevent	jakarta.el.ValueExpression that evaluates to a String	The name of the JavaScript function that handles UI events.
onerror	jakarta.el.ValueExpression that evaluates to a String	The name of the JavaScript function that handles errors.
render	jakarta.el.ValueExpression that evaluates to an Object	A Collection that identifies a list of components to be rendered on the client. If a literal is specified, it must be a space-delimited String of component identifiers and/or one of the keywords. If a ValueExpression is specified, it must refer to a property that returns a Collection of String objects. If not specified, the default value is @none.

The keywords listed in Execute and Render Keywords can be used with the execute and render attributes of the f:ajax tag.

Execute and Render Keywords

Keyword	Description
@all	All component identifiers
@form	The form that encloses the component
@none	No component identifiers
@this	The element that triggered the request

Note that when you use the f:ajax tag in a Facelets page, the JavaScript resource library is loaded implicitly. This resource library can also be loaded explicitly as described in Loading JavaScript as a Resource.

Sending an Ajax Request

To activate Ajax functionality, the web application must create an Ajax request and send it to the server. The server then processes the request.

The application uses the attributes of the f:ajax tag listed in attributes of the f:ajax tag to create the Ajax request. The following sections explain the process of creating and sending an Ajax request using some of these attributes.



Behind the scenes, the `faces.ajax.request()` method of the JavaScript resource library collects the data provided by the `f:ajax` tag and posts the request to the Jakarta Faces lifecycle.

Using the event Attribute

The `event` attribute defines the event that triggers the Ajax action. Some of the possible values for this attribute are `click`, `keyup`, `mouseover`, `focus`, and `blur`.

If not specified, a default event based on the parent component will be applied. The default event is `action` for `jakarta.faces.component.ActionSource` components, such as a `commandButton`, and `valueChange` for `jakarta.faces.component.EditableValueHolder` components, such as `inputText`. In the following example, an Ajax tag is associated with the button component, and the event that triggers the Ajax action is a mouse click:

```
<h:commandButton id="submit" value="Submit">
    <f:ajax event="click" />
</h:commandButton>
<h:outputText id="result" value="#{userNumberBean.response}" />
```



You may have noticed that the listed events are very similar to JavaScript events. In fact, they are based on JavaScript events, but do not have the `on` prefix.

For a command button, the default event is `click`, so you do not actually need to specify `event="click"` to obtain the desired behavior.

Using the execute Attribute

The `execute` attribute defines the component or components to be executed on the server. The component is identified by its `id` attribute. You can specify more than one executable component. If more than one component is to be executed, specify a space-delimited list of components.

When a component is executed, it participates in all phases of the request-processing lifecycle except the Render Response phase.

The `execute` attribute value can also be a keyword, such as `@all`, `@none`, `@this`, or `@form`. The default value is `@this`, which refers to the component within which the `f:ajax` tag is nested.

The following code specifies that the `h:inputText` component with the `id` value of `userNo` should be executed when the button is clicked:

```
<h:inputText id="userNo"
             title="Type a number from 0 to 10:"
             value="#{userNumberBean.userNumber}">
    ...
</h:inputText>
<h:commandButton id="submit" value="Submit">
    <f:ajax event="click" execute="userNo" />
```

```
</h:commandButton>
```

Using the immediate Attribute

The `immediate` attribute indicates whether user inputs are to be processed early in the application lifecycle or later. If the attribute is set to `true`, events generated from this component are broadcast during the Apply Request Values phase. Otherwise, the events will be broadcast during the Invoke Application phase.

If not defined, the default value of this attribute is `false`.

Using the listener Attribute

The `listener` attribute refers to a method expression that is executed on the server side in response to an Ajax action on the client. The listener's `jakarta.faces.event.AjaxBehaviorListener.processAjaxBehavior` method is called once during the Invoke Application phase of the lifecycle. In the following code from the `reservation` example application (see The reservation Example Application), a `listener` attribute is defined by an `f:ajax` tag, which refers to a method from the bean:

```
<f:ajax event="change" render="total"
        listener="#{reservationBean.calculateTotal}"/>
```

Whenever either the price or the number of tickets ordered changes, the `calculateTotal` method of `ReservationBean` recalculates the total cost of the tickets and displays it in the output component named `total`.

Monitoring Events on the Client

To monitor ongoing Ajax requests, use the `onevent` attribute of the `f:ajax` tag. The value of this attribute is the name of a JavaScript function. Jakarta Faces calls the `onevent` function at each stage of the processing of an Ajax request: begin, complete, and success.

When calling the JavaScript function assigned to the `onevent` property, Jakarta Faces passes a data object to it. The data object contains the properties listed in Properties of the onevent Data Object.

Properties of the onevent Data Object

Property	Description
`responseXML`	The response to the Ajax call in XML format
`responseText`	The response to the Ajax call in text format
`responseCode`	The response to the Ajax call in numeric code
`source`	The source of the current Ajax event: the DOM element
`status`	The status of the current Ajax call: `begin`, `complete`, or `success`

Property	Description
`type`	The type of the Ajax call: `event`

By using the `status` property of the data object, you can identify the current status of the Ajax request and monitor its progress. In the following example, `monitormyajaxevent` is a JavaScript function that monitors the Ajax request sent by the event:

```
<f:ajax event="click" render="statusmessage" onevent="monitormyajaxevent"/>
```

Handling Errors

Jakarta Faces handles Ajax errors through use of the `onerror` attribute of the `f:ajax` tag. The value of this attribute is the name of a JavaScript function.

When there is an error in processing a Ajax request, Jakarta Faces calls the defined `onerror` JavaScript function and passes a data object to it. The data object contains all the properties available for the `onevent` attribute and, in addition, the following properties:

- `description`
- `errorName`
- `errorMessage`

The `type` is `error`. The `status` property of the data object contains one of the valid error values listed in Valid Error Values for the Data Object status Property.

Valid Error Values for the Data Object status Property

Values	Description
`emptyResponse`	No Ajax response from server.
`httpError`	One of the valid HTTP errors: `request.status==null` or `request.status==undefined` or `request.status<200` or `request.status>=300`.
`malformedXML`	The Ajax response is not well formed.
`serverError`	The Ajax response contains an `error` element.

In the following example, any errors that occurred in processing the Ajax request are handled by the `handlemyajaxerror` JavaScript function:

```
<f:ajax event="click" render="errormessage" onerror="handlemyajaxerror"/>
```

Receiving an Ajax Response

After the application sends an Ajax request, it is processed on the server side, and a response is sent

back to the client. As described earlier, Ajax allows for partial updating of web pages. To enable such partial updating, Jakarta Faces technology allows for partial processing of the view. The handling of the response is defined by the `render` attribute of the `f:ajax` tag.

Similar to the `execute` attribute, the `render` attribute defines which sections of the page will be updated. The value of a `render` attribute can be one or more component `id` values, one of the keywords `@this`, `@all`, `@none`, or `@form`, or an EL expression. In the following example, the `render` attribute identifies an output component to be displayed when the button component is clicked (the default event for a command button):

```
<h:commandButton id="submit" value="Submit">
    <f:ajax execute="userNo" render="result" />
</h:commandButton>
<h:outputText id="result" value="#{userNumberBean.response}" />
```



Behind the scenes, once again the `faces.ajax.request()` method handles the response. It registers a response-handling callback when the original request is created. When the response is sent back to the client, the callback is invoked. This callback automatically updates the client-side DOM to reflect the rendered response.

Ajax Request Lifecycle

An Ajax request varies from other typical Jakarta Faces requests, and its processing is also handled differently by the Jakarta Faces lifecycle.

As described in Partial Processing and Partial Rendering, when an Ajax request is received, the state associated with that request is captured by the `jakarta.faces.context.PartialViewContext`. This object provides access to information such as which components are targeted for processing/rendering. The `processPartial` method of `PartialViewContext` uses this information to perform partial component tree processing and rendering.

The `execute` attribute of the `f:ajax` tag identifies which segments of the server-side component tree should be processed. Because components can be uniquely identified in the Jakarta Faces component tree, it is easy to identify and process a single component, a few components, or a whole tree. This is made possible by the `visitTree` method of the `UIComponent` class. The identified components then run through the Jakarta Faces request lifecycle phases.

Similar to the `execute` attribute, the `render` attribute identifies which segments of the Jakarta Faces component tree need to be rendered during the render response phase.

During the render response phase, the `render` attribute is examined. The identified components are found and asked to render themselves and their children. The components are then packaged up and sent back to the client as a response.

Grouping of Components

The previous sections describe how to associate a single UI component with Ajax functionality. You

can also associate Ajax with more than one component at a time by grouping them together on a page. The following example shows how a number of components can be grouped by using the `f:ajax` tag:

```
<f:ajax>
    <h:form>
        <h:inputText id="input1" value="#{user.name}"/>
        <h:commandButton id="Submit"/>
    </h:form>
</f:ajax>
```

In the example, neither component is associated with any Ajax `event` or `render` attributes yet. Therefore, no action will take place in case of user input. You can associate the above components with an `event` and a `render` attribute as follows:

```
<f:ajax event="click" render="@all">
    <h:form>
        <h:inputText id="input1" value="#{user.name}"/>
        <h:commandButton id="Submit"/>
    </h:form>
</f:ajax>
```

In the updated example, when the user clicks either component, the updated results will be displayed for all components. You can further fine-tune the Ajax action by adding specific events to each of the components, in which case Ajax functionality becomes cumulative. Consider the following example:

```
<f:ajax event="click" render="@all">
    ...
    <h:commandButton id="Submit">
        <f:ajax event="mouseover"/>
    </h:commandButton>
    ...
</f:ajax>
```

Now the button component will fire an Ajax action in case of a `mouseover` event as well as a mouse-click event.

Loading JavaScript as a Resource

The JavaScript resource file bundled with Jakarta Faces technology is named `faces.js` and is available in the `jakarta.faces` library. This resource library supports Ajax functionality in Jakarta Faces applications.

If you use the `f:ajax` tag on a page, the `faces.js` resource is automatically delivered to the client. It is not necessary to use the `h:outputScript` tag to specify this resource. You may want to use the `h:outputScript` tag to specify other JavaScript libraries.

To use a JavaScript resource directly with a `UIComponent`, you must explicitly load the resource as described in either of the following topics:

- Using JavaScript API in a Facelets Application – Uses the `h:outputScript` tag directly in a Facelets page
- Using the @ResourceDependency Annotation in a Bean Class – Uses the `jakarta.faces.application.ResourceDependency` annotation on a `UIComponent` Java class

Using JavaScript API in a Facelets Application

To use the JavaScript resource API directly in a web application, such as a Facelets page:

1. Identify the default JavaScript resource for the page with the help of the `h:outputScript` tag.

   For example, consider the following section of a Facelets page:

   ```
   <h:form>
       <h:outputScript name="faces.js" library="jakarta.faces" target="head"/>
   </h:form>
   ```

   Specifying the target as `head` causes the script resource to be rendered within the `head` element on the HTML page.

2. Identify the component to which you would like to attach the Ajax functionality.
3. Add the Ajax functionality to the component by using the JavaScript API. For example, consider the following:

   ```
   <h:form>
       <h:outputScript name="faces.js" library="jakarta.faces" target="head">
       <h:inputText id="inputname" value="#{userBean.name}"/>
       <h:outputText id="outputname" value="#{userBean.name}"/>
       <h:commandButton id="submit" value="Submit"
                        onclick="faces.ajax.request(this, event,
                                 {execute:'inputname',render:'outputname'});
                                 return false;" />
   </h:form>
   ```

   The `faces.ajax.request` method takes up to three parameters that specify source, event, and options. The source parameter identifies the DOM element that triggered the Ajax request, typically `this`. The optional event parameter identifies the DOM event that triggered this request. The optional options parameter contains a set of name/value pairs from Possible Values for the Options Parameter.

   Possible Values for the Options Parameter

Name	Value
`execute`	A space-delimited list of client identifiers or one of the keywords listed in Execute And Render Keywords. The identifiers reference the components that will be processed during the Execute phase of the lifecycle.
`render`	A space-delimited list of client identifiers or one of the keywords listed in Execute And Render Keywords. The identifiers reference the components that will be processed during the render phase of the lifecycle.
`onevent`	A `String` that is the name of the JavaScript function to call when an event occurs.
`onerror`	A `String` that is the name of the JavaScript function to call when an error occurs.
`params`	An object that may include additional parameters to include in the request.

If no identifier is specified, the default assumed keyword for the `execute` attribute is `@this`, and for the `render` attribute it is `@none`.

You can also place the JavaScript method in a file and include it as a resource.

Using the @ResourceDependency Annotation in a Bean Class

Use the `jakarta.faces.application.ResourceDependency` annotation to cause the bean class to load the default `faces.js` library.

To load the Ajax resource from the server side:

1. Use the `faces.ajax.request` method within the bean class.



This method is usually used when creating a custom component or a custom renderer for a component.

The following example shows how the resource is loaded in a bean class:

```
@ResourceDependency(name="faces.js" library="jakarta.faces" target="head")
```

The ajaxguessnumber Example Application

To demonstrate the advantages of using Ajax, revisit the `guessnumber` example from [web:faces-facelets::faces-facelets:::_introduction_to_facelets]. If you modify this example to use Ajax, the response need not be displayed on the `response.xhtml` page. Instead, an asynchronous call is made to the bean on the server side, and the response is displayed on the originating page by executing just the input component rather than by form submission.

The source code for this application is in the jakartaee-examples/tutorial/web/faces/ajaxguessnumber/ directory.

The ajaxguessnumber Source Files

The changes to the guessnumber application occur in two source files.

The ajaxgreeting.xhtml Facelets Page

The Facelets page for ajaxguessnumber, ajaxgreeting.xhtml, is almost the same as the greeting.xhtml page for the guessnumber application:

```
<h:head>
    <h:outputStylesheet library="css" name="default.css"/>
    <title>Ajax Guess Number Facelets Application</title>
</h:head>
<h:body>
    <h:form id="AjaxGuess">
        <h:graphicImage value="#{resource['images:wave.med.gif']}"
                        alt="Duke waving his hand"/>
        <h2>
            Hi, my name is Duke. I am thinking of a number from
            #{dukesNumberBean.minimum} to #{dukesNumberBean.maximum}.
            Can you guess it?
        </h2>
        <p>
            <h:inputText id="userNo"
                         title="Enter a number from 0 to 10:"
                         value="#{userNumberBean.userNumber}">
                <f:validateLongRange minimum="#{dukesNumberBean.minimum}"
                                     maximum="#{dukesNumberBean.maximum}"/>
            </h:inputText>

            <h:commandButton id="submit" value="Submit">
                <f:ajax execute="userNo" render="outputGroup" />
            </h:commandButton>
        </p>
        <p>
            <h:panelGroup layout="block" id="outputGroup">
                <h:outputText id="result" style="color:blue"
                              value="#{userNumberBean.response}"
                              rendered="#{!facesContext.validationFailed}"/>
                <h:message id="errors1"
                           showSummary="true"
                           showDetail="false"
                           style="color: #d20005;
                           font-family: 'New Century Schoolbook', serif;
                           font-style: oblique;
                           text-decoration: overline"
                           for="userNo"/>
            </h:panelGroup>
```

```
        </p>
    </h:form>
</h:body>
```

The most important change is in the `h:commandButton` tag. The `action` attribute is removed from the tag, and an `f:ajax` tag is added.

The `f:ajax` tag specifies that when the button is clicked the `h:inputText` component with the `id` value `userNo` is executed. The components within the `outputGroup` panel group are then rendered. If a validation error occurs, the managed bean is not executed, and the validation error message is displayed in the message pane. Otherwise, the result of the guess is rendered in the `result` component.

The UserNumberBean Backing Bean

A small change is also made in the `UserNumberBean` code so that the output component does not display any message for the default (null) value of the property `response`. Here is the modified bean code:

```
public String getResponse() {
    if ((userNumber != null)
            && (userNumber.compareTo(dukesNumberBean.getRandomInt()) == 0)) {
        return "Yay! You got it!";
    }
    if (userNumber == null) {
        return null;
    } else {
        return "Sorry, " + userNumber + " is incorrect.";
    }
}
```

The DukesNumberBean CDI Managed Bean

The `DukesNumberBean` session-scoped CDI managed bean stores the range of guessable numbers and the randomly chosen number from that range. It is injected into `UserNumberBean` with the CDI `@Inject` annotation so that the value of the random number can be compared to the number the user submitted:

```
@Inject
DukesNumberBean dukesNumberBean;
```

You will learn more about CDI in [cdi:cdi-basic::cdi-basic:::_introduction_to_jakarta_contexts_and_dependency_injection].

Running the ajaxguessnumber Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `ajaxguessnumber` application.

To Build, Package, and Deploy the ajaxguessnumber Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `ajaxguessnumber` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `ajaxguessnumber` project and select **Build**.

   This command builds and deploys the project.

To Build, Package, and Deploy the ajaxguessnumber Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/ajaxguessnumber/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `ajaxguessnumber.war`, located in the `target` directory. It then deploys the application.

To Run the ajaxguessnumber Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/ajaxguessnumber
   ```

2. Enter a value in the field and click Submit.

   If the value is in the range of 0 to 10, a message states whether the guess is correct or incorrect. If the value is outside that range or if the value is not a number, an error message appears in red.

Further Information about Ajax in Jakarta Faces Technology

For more information on Ajax in Jakarta Faces Technology, see

- Jakarta Faces project website:

https://jakarta.ee/specifications/faces/4.0/

- Jakarta Faces JavaScript Library APIs

Composite Components: Advanced Topics and an Example



This section was last updated for Jakarta EE Version 9.1

This chapter describes the advanced features of composite components in Jakarta Faces technology.

Attributes of a Composite Component

A composite component is a special type of Jakarta Faces template that acts as a component. If you are new to composite components, see Composite Components before you proceed with this chapter.

You define an attribute of a composite component by using the `composite:attribute` tag. Commonly Used Attributes of the composite:attribute Tag lists the commonly used attributes of this tag.

Commonly Used Attributes of the composite:attribute Tag

Attribute	Description
`name`	Specifies the name of the composite component attribute to be used in the using page. Alternatively, the `name` attribute can specify standard event handlers such as `action`, `actionListener`, and managed bean.
`default`	Specifies the default value of the composite component attribute.
`required`	Specifies whether it is mandatory to provide a value for the attribute.
`method-signature`	Specifies a subclass of `java.lang.Object` as the type of the composite component's attribute. The `method-signature` element declares that the composite component attribute is a method expression. The `type` attribute and the `method-signature` attribute are mutually exclusive. If you specify both, `method-signature` is ignored. The default type of an attribute is `java.lang.Object`. Note: Method expressions are similar to value expressions, but rather than supporting the dynamic retrieval and setting of properties, method expressions support the invocation of a method of an arbitrary object, passing a specified set of parameters and returning the result from the called method (if any).

Attribute	Description
`type`	Specifies a fully qualified class name as the type of the attribute. The `type` attribute and the `method-signature` attribute are mutually exclusive. If you specify both, `method-signature` is ignored. The default type of an attribute is `java.lang.Object`.

The following code snippet defines a composite component attribute and assigns it a default value:

```
<composite:attribute name="username" default="admin"/>
```

The following code snippet uses the `method-signature` element:

```
<composite:attribute name="myaction"
                     method-signature="java.lang.String action()"/>
```

The following code snippet uses the `type` element:

```
<composite:attribute name="dateofjoining" type="java.util.Date"/>
```

Invoking a Managed Bean

To enable a composite component to handle server-side data

1. Invoke a managed bean in one of the following ways:
    - Pass the reference of the managed bean to the composite component.
    - Directly use the properties of the managed bean.

      The example application described in The compositecomponentexample Example Application shows how to use a managed bean with a composite component by passing the reference of the managed bean to the component.

Validating Composite Component Values

Jakarta Faces provides the following tags for validating values of input components. These tags can be used with the `composite:valueHolder` or the `composite:editableValueHolder` tag.

Validator Tags lists commonly used validator tags. See Using the Standard Validators for details and a complete list.

Validator Tags

Tag Name	Description
`f:validateBean`	Delegates the validation of the local value to the Bean Validation API.

Tag Name	Description
f:validateRegex	Uses the pattern attribute to validate the wrapping component. The entire pattern is matched against the String value of the component. If it matches, it is valid.
f:validateRequired	Enforces the presence of a value. Has the same effect as setting the required element of a composite component's attribute to true.

The compositecomponentexample Example Application

The compositecomponentexample application creates a composite component that accepts a name (or any other string). The component interacts with a managed bean that calculates whether the letters in the name, if converted to numeric values, add up to a prime number. The component displays the sum of the letter values and reports whether it is or is not prime.

The compositecomponentexample application has a composite component file, a using page, and a managed bean.

The source code for this application is in the jakartaee-examples/tutorial/web/faces/compositecomponentexample/ directory.

The Composite Component File

The composite component file is an XHTML file, /web/resources/ezcomp/PrimePanel.xhtml. It has a composite:interface section that declares the labels for the name and a command button. It also declares a managed bean, which defines properties for the name.

```
<composite:interface>
    <composite:attribute name="namePrompt"
                         default="Name, word, or phrase: "/>
    <composite:attribute name="calcButtonText" default="Calculate"/>
    <composite:attribute name="calcAction"
                         method-signature="java.lang.String action()"/>
    <composite:attribute name="primeBean"/>
    <composite:editableValueHolder name="nameVal" targets="form:name"/>
</composite:interface>
```

The composite component implementation accepts the input value for the name property of the managed bean. The h:outputStylesheet tag specifies the stylesheet as a relocatable resource. The implementation then specifies the format of the output, using properties of the managed bean, as well as the format of error messages. The sum value is rendered only after it has been calculated, and the report of whether the sum is prime or not is rendered only if the input value is validated.

```
<composite:implementation>
    <h:form id="form">
        <h:outputStylesheet library="css" name="default.css"
```

```
                            target="head"/>
        <h:panelGrid columns="2" role="presentation">
            <h:outputLabel for="name"
                           value="#{cc.attrs.namePrompt}"/>
            <h:inputText id="name"
                         size="45"
                         value="#{cc.attrs.primeBean.name}"
                         required="true"/>
        </h:panelGrid>
        <p>
            <h:commandButton id="calcButton"
                             value="#{cc.attrs.calcButtonText}"
                             action="#{cc.attrs.calcAction}">
                <f:ajax execute="name" render="outputGroup"/>
            </h:commandButton>
        </p>

       <h:panelGroup id="outputGroup" layout="block">
            <p>
                <h:outputText id="result" style="color:blue"
                              rendered="#{cc.attrs.primeBean.totalSum gt 0}"
                              value="Sum is #{cc.attrs.primeBean.totalSum}" />
            </p>
            <p>
                <h:outputText id="response" style="color:blue"
                              value="#{cc.attrs.primeBean.response}"
                              rendered="#{!facesContext.validationFailed}"/>
                <h:message id="errors1"
                           showSummary="true"
                           showDetail="false"
                           style="color: #d20005;
                           font-family: 'New Century Schoolbook', serif;
                           font-style: oblique;
                           text-decoration: overline"
                           for="name"/>
            </p>
        </h:panelGroup>
    </h:form>
</composite:implementation>
```

The Using Page

The using page in this example application, `web/index.xhtml`, is an XHTML file that invokes the `PrimePanel.xhtml` composite component file along with the managed bean. It validates the user's input.

```
<div id="compositecomponent">
    <ez:PrimePanel primeBean="#{primeBean}" calcAction="#{primeBean.calculate}">
    </ez:PrimePanel>
```

```
</div>
```

The Managed Bean

The managed bean, `PrimeBean.java`, defines a method called `calculate`, which performs the calculations on the input string and sets properties accordingly. The bean first creates an array of prime numbers. It calculates the sum of the letters in the string, with `'a'` equal to 1 and `'z'` equal to 26, and determines whether the value can be found in the array of primes. An uppercase letter in the input string has the same value as its lowercase equivalent.

The bean specifies the minimum and maximum size of the `name` string, which is enforced by the Bean Validation `@Size` constraint. The bean uses the `@Model` annotation, a shortcut for `@Named` and `@RequestScoped`, as described in Step 7 of To View the hello1 Web Module Using NetBeans IDE.

```
@Model
public class PrimeBean {
    ...
    @Size(min=1, max=45)
    private String name;
    ...

    public String calculate() {
        ...
    }
}
```

Running the compositecomponentexample Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `compositecomponentexample` example.

To Build, Package, and Deploy the compositecomponentexample Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `compositecomponentexample` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `compositecomponentexample` project and select **Build**.

   This command builds and deploys the application.

To Build, Package, and Deploy the compositecomponentexample Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/compositecomponentexample/
   ```

3. Enter the following command to build and deploy the application:

   ```
   mvn install
   ```

To Run the compositecomponentexample Example

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/compositecomponentexample
   ```

2. On the page that appears, enter a string in the Name, word, or phrase field, then click Calculate.

   The page reports the sum of the letters and whether the sum is prime. A validation error is reported if no value is entered or if the string contains more than 45 characters.

Creating Custom UI Components and Other Custom Objects



This section was last updated for Jakarta EE Version 9.1

This chapter describes creating custom components for applications that have additional functionality not provided by standard Jakarta Faces components.

Introduction to Creating Custom Components

Jakarta Faces technology offers a basic set of standard, reusable UI components that enable quick and easy construction of user interfaces for web applications. These components mostly map one-to-one to the elements in HTML 4. But often an application requires a component that has additional functionality or requires a completely new component. Jakarta Faces technology allows extension of standard components to enhance their functionality or to create custom components. A rich ecosystem of third-party component libraries is built on this extension capability, but it is beyond the scope of this tutorial to examine them. A web search for "Faces Component Libraries" (or "JSF Component Libraries") is a good starting point to learn more about this important aspect of using Jakarta Faces technology.

In addition to extending the functionality of standard components, a component writer might want to give a page author the ability to change the appearance of the component on the page or to alter listener behavior. Alternatively, the component writer might want to render a component to a different kind of client device type, such as a smartphone or a tablet instead of a desktop computer. Enabled by the flexible Jakarta Faces architecture, a component writer can separate the definition

of the component behavior from its appearance by delegating the rendering of the component to a separate renderer. In this way, a component writer can define the behavior of a custom component once but create multiple renderers, each of which defines a different way to render the component to a particular kind of client device.

A `jakarta.faces.component.UIComponent` is a Java class that is responsible for representing a self-contained piece of the user interface during the request-processing lifecycle. It is intended to represent the meaning of the component; the visual representation of the component is the responsibility of the `jakarta.faces.render.Renderer`. There can be multiple instances of the same `UIComponent` class in any given Jakarta Faces view, just as there can be multiple instances of any Java class in any given Java program.

Jakarta Faces technology provides the ability to create custom components by extending the `UIComponent` class, the base class for all standard UI components. A custom component can be used anywhere an ordinary component can be used, such as within a composite component. A `UIComponent` is identified by two names: `component-family` specifies the general purpose of the component (input or output, for instance), and `component-type` indicates the specific purpose of a component, such as a text input field or a command button.

A `Renderer` is a helper to the `UIComponent` that deals with how that specific `UIComponent` class should appear in a specific kind of client device. Renderers are identified by two names: `render-kit-id` and `renderer-type`. A render kit is just a bucket into which a particular group of renderers is placed, and the `render-kit-id` identifies the group. Most Jakarta Faces component libraries provide their own render kits.

A `jakarta.faces.view.facelets.Tag` object is a helper to the `UIComponent` and `Renderer` that allows the page author to include an instance of a `UIComponent` in a Jakarta Faces view. A tag represents a specific combination of `component-type` and `renderer-type`.

See Component, Renderer, and Tag Combinations for information on how components, renderers, and tags interact.

This chapter uses the image map component from the Duke's Bookstore case study example to explain how you can create simple custom components, custom renderers, and associated custom tags, and take care of all the other details associated with using the components and renderers in an application. See Duke's Bookstore Case Study Example for more information about this example.

The chapter also describes how to create other custom objects: custom converters, custom listeners, and custom validators. It also describes how to bind component values and instances to data objects and how to bind custom objects to managed bean properties.

Determining Whether You Need a Custom Component or Renderer

The Jakarta Faces implementation supports a very basic set of components and associated renderers. This section helps you to decide whether you can use standard components and renderers in your application or need a custom component or custom renderer.

When to Use a Custom Component

A component class defines the state and behavior of a UI component. This behavior includes

converting the value of a component to the appropriate markup, queuing events on components, performing validation, and any other behavior related to how the component interacts with the browser and the request-processing lifecycle.

You need to create a custom component in the following situations.

- You need to add new behavior to a standard component, such as generating an additional type of event (for example, notifying another part of the page that something changed in this component as a result of user interaction).
- You need to take a different action in the request processing of the value of a component from what is available in any of the existing standard components.
- You want to take advantage of an HTML capability offered by your target browser, but none of the standard Jakarta Faces components take advantage of the capability in the way you want, if at all. The current release does not contain standard components for complex HTML components, such as frames; however, because of the extensibility of the component architecture, you can use Jakarta Faces technology to create components like these. The Duke's Bookstore case study creates custom components that correspond to the HTML `map` and `area` tags.
- You need to render to a non-HTML client that requires extra components not supported by HTML. Eventually, the standard HTML render kit will provide support for all standard HTML components. However, if you are rendering to a different client, such as a phone, you might need to create custom components to represent the controls uniquely supported by the client. For example, some component architectures for wireless clients include support for tickers and progress bars, which are not available on an HTML client. In this case, you might also need a custom renderer along with the component, or you might need only a custom renderer.

You do not need to create a custom component in the following cases.

- You need to aggregate components to create a new component that has its own unique behavior. In this situation, you can use a composite component to combine existing standard components. For more information on composite components, see Composite Components and [web:faces-advanced-cc::faces-advanced-cc:::_composite_components_advanced_topics_and_an_example].
- You simply need to manipulate data on the component or add application-specific functionality to it. In this situation, you should create a managed bean for this purpose and bind it to the standard component rather than create a custom component. See Managed Beans in Jakarta Faces Technology for more information on managed beans.
- You need to convert a component's data to a type not supported by its renderer. See Using the Standard Converters for more information about converting a component's data.
- You need to perform validation on the component data. Standard validators and custom validators can be added to a component by using the validator tags from the page. See Using the Standard Validators and Creating and Using a Custom Validator for more information about validating a component's data.
- You need to register event listeners on components. You can either register event listeners on components using the `f:valueChangeListener` and `f:actionListener` tags, or you can point at an event-processing method on a managed bean using the component's `actionListener` or

`valueChangeListener` attributes. See Implementing an Event Listener and Writing Managed Bean Methods for more information.

When to Use a Custom Renderer

A renderer, which generates the markup to display a component on a web page, allows you to separate the semantics of a component from its appearance. By keeping this separation, you can support different kinds of client devices with the same kind of authoring experience. You can think of a renderer as a "client adapter." It produces output suitable for consumption and display by the client and accepts input from the client when the user interacts with that component.

If you are creating a custom component, you need to ensure, among other things, that your component class performs these operations that are central to rendering the component:

- Decoding: Converting the incoming request parameters to the local value of the component
- Encoding: Converting the current local value of the component into the corresponding markup that represents it in the response

The Jakarta Faces specification supports two programming models for handling encoding and decoding.

- Direct implementation: The component class itself implements the decoding and encoding.
- Delegated implementation: The component class delegates the implementation of encoding and decoding to a separate renderer.

By delegating the operations to the renderer, you have the option of associating your custom component with different renderers so that you can render the component on different clients. If you don't plan to render a particular component on different clients, it may be simpler to let the component class handle the rendering. However, a separate renderer enables you to preserve the separation of semantics from appearance. The Duke's Bookstore application separates the renderers from the components, although it renders only to HTML 4 web browsers.

If you aren't sure whether you will need the flexibility offered by separate renderers but you want to use the simpler direct-implementation approach, you can actually use both models. Your component class can include some default rendering code, but it can delegate rendering to a renderer if there is one.

Component, Renderer, and Tag Combinations

When you create a custom component, you can create a custom renderer to go with it. To associate the component with the renderer and to reference the component from the page, you will also need a custom tag.

Although you need to write the custom component and renderer, there is no need to write code for a custom tag (called a tag handler). If you specify the component and renderer combination, Facelets creates the tag handler automatically.

In rare situations, you might use a custom renderer with a standard component rather than a custom component. Or you might use a custom tag without a renderer or a component. This section gives examples of these situations and summarizes what is required for a custom component,

renderer, and tag.

You would use a custom renderer without a custom component if you wanted to add some client-side validation on a standard component. You would implement the validation code with a client-side scripting language, such as JavaScript, and then render the JavaScript with the custom renderer. In this situation, you need a custom tag to go with the renderer so that its tag handler can register the renderer on the standard component.

Custom components as well as custom renderers need custom tags associated with them. However, you can have a custom tag without a custom renderer or custom component. For example, suppose that you need to create a custom validator that requires extra attributes on the validator tag. In this case, the custom tag corresponds to a custom validator and not to a custom component or custom renderer. In any case, you still need to associate the custom tag with a server-side object.

Requirements for Custom Components, Custom Renderers and Custom Tags summarizes what you must or can associate with a custom component, custom renderer, or custom tag.

Requirements for Custom Components, Custom Renderers and Custom Tags

Custom Item	Must Have	Can Have
Custom component	Custom tag	Custom renderer or standard renderer
Custom renderer	Custom tag	Custom component or standard component
Custom Jakarta Faces tag	Some server-side object, like a component, a custom renderer, or custom validator	Custom component or standard component associated with a custom renderer

Understanding the Image Map Example

Duke's Bookstore includes a custom image map component on the `index.xhtml` page. This image map displays a selection of six book titles. When the user clicks one of the book titles in the image map, the application goes to a page that displays the title of the selected book as well as information about a featured book. The page allows the user to add either book (or none) to the shopping cart.

Why Use Jakarta Faces Technology to Implement an Image Map?

Jakarta Faces technology is an ideal framework to use for implementing this kind of image map because it can perform the work that must be done on the server without requiring you to create a server-side image map.

In general, client-side image maps are preferred over server-side image maps for several reasons. One reason is that the client-side image map allows the browser to provide immediate feedback when a user positions the mouse over a hotspot. Another reason is that client-side image maps perform better because they don't require round-trips to the server. However, in some situations, your image map might need to access the server to retrieve data or to change the appearance of nonform controls, tasks that a client-side image map cannot do.

Because the image map custom component uses Jakarta Faces technology, it has the best of both styles of image maps: It can handle the parts of the application that need to be performed on the server while allowing the other parts of the application to be performed on the client side.

Understanding the Rendered HTML

Here is an abbreviated version of the form part of the HTML page that the application needs to render:

```
<form id="j_idt13" name="j_idt13" method="post"
        action="/dukesbookstore/index.xhtml" ...>
    ...
    <img id="j_idt13:mapImage"
         src="/dukesbookstore/jakarta.faces.resource/book_all.jpg?ln=images"
         alt="Choose a Book from our Catalog"
         usemap="#bookMap" />
    ...
    <map name="bookMap">
       <area alt="Duke"
          coords="67,23,212,268"
          shape="rect"

onmouseout="document.forms[0]['j_idt13:mapImage'].src='resources/images/book_all.jpg'"

onmouseover="document.forms[0]['j_idt13:mapImage'].src='resources/images/book_201.jpg'
"
          onclick="document.forms[0]['bookMap_current'].value='Duke';
document.forms[0].submit()"
       />
    ...
       <input type="hidden" name="bookMap_current">
    </map>
    ...
</form>
```

The `img` tag associates an image (`book_all.jpg`) with the image map referenced in the `usemap` attribute value.

The `map` tag specifies the image map and contains a set of `area` tags.

Each `area` tag specifies a region of the image map. The `onmouseover`, `onmouseout`, and `onclick` attributes define which JavaScript code is executed when these events occur. When the user moves the mouse over a region, the `onmouseover` function associated with the region displays the map with that region highlighted. When the user moves the mouse out of a region, the `onmouseout` function redisplays the original image. If the user clicks on a region, the `onclick` function sets the value of the `input` tag to the ID of the selected area and submits the page.

The `input` tag represents a hidden control that stores the value of the currently selected area between client-server exchanges so that the server-side component classes can retrieve the value.

The server-side objects retrieve the value of `bookMap_current` and set the locale in the `jakarta.faces.context.FacesContext` instance according to the region that was selected.

Understanding the Facelets Page

Here is an abbreviated form of the Facelets page that the image map component uses to generate the HTML page shown in the preceding section. It uses custom `bookstore:map` and `bookstore:area` tags to represent the custom components:

```
<h:form>
    ...
        <h:graphicImage id="mapImage"
                        name="book_all.jpg"
                        library="images"
                        alt="#{bundle.ChooseBook}"
                        usemap="#bookMap" />
        <bookstore:map id="bookMap"
                       current="map1"
                       immediate="true"
                       action="bookstore">
            <f:actionListener
                type="dukesbookstore.listeners.MapBookChangeListener" />
            <bookstore:area id="map1" value="#{Book201}"
                            onmouseover="resources/images/book_201.jpg"
                            onmouseout="resources/images/book_all.jpg"
                            targetImage="mapImage" />
            <bookstore:area id="map2" value="#{Book202}"
                            onmouseover="resources/images/book_202.jpg"
                            onmouseout="resources/images/book_all.jpg"
                            targetImage="mapImage"/>
            ...
        </bookstore:map>
    ...
</h:form>
```

The `alt` attribute of the `h:graphicImage` tag maps to the localized string `"Choose a Book from our Catalog"`.

The `f:actionListener` tag within the `bookstore:map` tag points to a listener class for an action event. The `processAction` method of the listener places the book ID for the selected map area into the session map. The way this event is handled is explained more in Handling Events for Custom Components.

The `action` attribute of the `bookstore:map` tag specifies a logical outcome `String`, `"bookstore"`, which by implicit navigation rules sends the application to the page `bookstore.xhtml`. For more information on navigation, see Configuring Navigation Rules.

The `immediate` attribute of the `bookstore:map` tag is set to `true`, which indicates that the default `jakarta.faces.event.ActionListener` implementation should execute during the Apply Request

Values phase of the request-processing lifecycle, instead of waiting for the Invoke Application phase. Because the request resulting from clicking the map does not require any validation, data conversion, or server-side object updates, it makes sense to skip directly to the Invoke Application phase.

The `current` attribute of the `bookstore:map` tag is set to the default area, which is `map1` (the book My Early Years: Growing Up on Star7, by Duke).

Notice that the `bookstore:area` tags do not contain any of the JavaScript, coordinate, or shape data that is displayed on the HTML page. The JavaScript is generated by the `dukesbookstore.renderers.AreaRenderer` class. The `onmouseover` and `onmouseout` attribute values indicate the image to be loaded when these events occur. How the JavaScript is generated is explained more in Performing Encoding.

The coordinate, shape, and alternate text data are obtained through the `value` attribute, whose value refers to an attribute in application scope. The value of this attribute is a bean, which stores the `coords`, `shape`, and `alt` data. How these beans are stored in the application scope is explained more in the next section.

Summary of the Image Map Application Classes

Image Map Classes summarizes all the classes needed to implement the image map component.

Image Map Classes

Class	Function
`AreaSelectedEvent`	The `jakarta.faces.event.ActionEvent` indicating that an `AreaComponent` from the `MapComponent` has been selected.
`AreaComponent`	The class that defines `AreaComponent`, which corresponds to the `bookstore:area` custom tag.
`MapComponent`	The class that defines `MapComponent`, which corresponds to the `bookstore:map` custom tag.
`AreaRenderer`	This `jakarta.faces.render.Renderer` performs the delegated rendering for `AreaComponent`.
`ImageArea`	The bean that stores the shape and coordinates of the hotspots.
`MapBookChangeListener`	The action listener for the `MapComponent`.

The Duke's Bookstore source directory, called bookstore-dir, is `jakartaee-examples/tutorial/case-studies/dukes-bookstore/src/main/java/jakarta/tutorial/dukesbookstore/`. The event and listener classes are located in `bookstore-dir/listeners/`. The component classes are located in `bookstore-dir/components/`. The renderer classes are located in `bookstore-dir/renderers/`. `ImageArea` is located in `bookstore-dir/model/`.

Steps for Creating a Custom Component

You can apply the following steps while developing your own custom component.

1. Create a custom component class that does the following:
    a. Overrides the `getFamily` method to return the component family, which is used to look up renderers that can render the component
    b. Includes the rendering code or delegates it to a renderer (explained in Step 2)
    c. Enables component attributes to accept expressions
    d. Queues an event on the component if the component generates events
    e. Saves and restores the component state
2. Delegate rendering to a renderer if your component does not handle the rendering. To do this:
    a. Create a custom renderer class by extending `jakarta.faces.render.Renderer`.
    b. Register the renderer to a render kit.
3. Register the component.
4. Create an event handler if your component generates events.
5. Create a tag library descriptor (TLD) that defines the custom tag.

See Registering a Custom Component and Registering a Custom Renderer with a Render Kit for information on registering the custom component and the renderer. The section Using a Custom Component discusses how to use the custom component in a Jakarta Faces page.

Creating Custom Component Classes

As explained in When to Use a Custom Component, a component class defines the state and behavior of a UI component. The state information includes the component's type, identifier, and local value. The behavior defined by the component class includes the following:

- Decoding (converting the request parameter to the component's local value)
- Encoding (converting the local value into the corresponding markup)
- Saving the state of the component
- Updating the bean value with the local value
- Processing validation on the local value
- Queueing events

The `jakarta.faces.component.UIComponentBase` class defines the default behavior of a component class. All the classes representing the standard components extend from `UIComponentBase`. These classes add their own behavior definitions, as your custom component class will do.

Your custom component class must either extend `UIComponentBase` directly or extend a class representing one of the standard components. These classes are located in the `jakarta.faces.component` package, and their names begin with `UI`.

If your custom component serves the same purpose as a standard component, you should extend

that standard component rather than directly extend `UIComponentBase`. For example, suppose you want to create an editable menu component. It makes sense to have this component extend `UISelectOne` rather than `UIComponentBase` because you can reuse the behavior already defined in `UISelectOne`. The only new functionality you need to define is to make the menu editable.

Whether you decide to have your component extend `UIComponentBase` or a standard component, you might also want your component to implement one or more of these behavioral interfaces defined in the `jakarta.faces.component` package:

- `ActionSource`: Indicates that the component can fire a `jakarta.faces.event.ActionEvent`
- `ActionSource2`: Extends `ActionSource` and allows component properties referencing methods that handle action events to use method expressions as defined by the EL
- `EditableValueHolder`: Extends `ValueHolder` and specifies additional features for editable components, such as validation and emitting value-change events
- `NamingContainer`: Mandates that each component rooted at this component has a unique ID
- `StateHolder`: Denotes that a component has state that must be saved between requests
- `ValueHolder`: Indicates that the component maintains a local value as well as the option of accessing data in the model tier

If your component extends `UIComponentBase`, it automatically implements only `StateHolder`. Because all components directly or indirectly extend `UIComponentBase`, they all implement `StateHolder`. Any component that implements `StateHolder` also implements the `StateHelper` interface, which extends `StateHolder` and defines a `Map`-like contract that makes it easy for components to save and restore a partial view state.

If your component extends one of the other standard components, it might also implement other behavioral interfaces in addition to `StateHolder`. If your component extends `UICommand`, it automatically implements `ActionSource2`. If your component extends `UIOutput` or one of the component classes that extend `UIOutput`, it automatically implements `ValueHolder`. If your component extends `UIInput`, it automatically implements `EditableValueHolder` and `ValueHolder`. See the Jakarta Faces API documentation to find out what the other component classes implement.

You can also make your component explicitly implement a behavioral interface that it doesn't already by virtue of extending a particular standard component. For example, if you have a component that extends `UIInput` and you want it to fire action events, you must make it explicitly implement `ActionSource2` because a `UIInput` component doesn't automatically implement this interface.

The Duke's Bookstore image map example has two component classes: `AreaComponent` and `MapComponent`. The `MapComponent` class extends `UICommand` and therefore implements `ActionSource2`, which means it can fire action events when a user clicks on the map. The `AreaComponent` class extends the standard component `UIOutput`. The `@FacesComponent` annotation registers the components with the Jakarta Faces implementation:

```
@FacesComponent("DemoMap")
public class MapComponent extends UICommand {...}
```

```
@FacesComponent("DemoArea")
public class AreaComponent extends UIOutput {...}
```

The `MapComponent` class represents the component corresponding to the `bookstore:map` tag:

```
<bookstore:map id="bookMap"
               current="map1"
               immediate="true"
               action="bookstore">
    ...
</bookstore:map>
```

The `AreaComponent` class represents the component corresponding to the `bookstore:area` tag:

```
<bookstore:area id="map1" value="#{Book201}"
                onmouseover="resources/images/book_201.jpg"
                onmouseout="resources/images/book_all.jpg"
                targetImage="mapImage"/>
```

`MapComponent` has one or more `AreaComponent` instances as children. Its behavior consists of the following actions:

- Retrieving the value of the currently selected area
- Defining the properties corresponding to the component's values
- Generating an event when the user clicks on the image map
- Queuing the event
- Saving its state
- Rendering the HTML `map` tag and the HTML `input` tag

`MapComponent` delegates the rendering of the HTML `map` and `input` tags to the `MapRenderer` class.

`AreaComponent` is bound to a bean that stores the shape and coordinates of the region of the image map. You will see how all this data is accessed through the value expression in Creating the Renderer Class. The behavior of `AreaComponent` consists of the following:

- Retrieving the shape and coordinate data from the bean
- Setting the value of the hidden tag to the `id` of this component
- Rendering the `area` tag, including the JavaScript for the `onmouseover`, `onmouseout`, and `onclick` functions

Although these tasks are actually performed by `AreaRenderer`, `AreaComponent` must delegate the tasks to `AreaRenderer`. See Delegating Rendering to a Renderer for more information.

The rest of this section describes the tasks that `MapComponent` performs as well as the encoding and decoding that it delegates to `MapRenderer`. Handling Events for Custom Components details how

`MapComponent` handles events.

Specifying the Component Family

If your custom component class delegates rendering, it needs to override the `getFamily` method of `UIComponent` to return the identifier of a component family, which is used to refer to a component or set of components that can be rendered by a renderer or set of renderers. The component family is used along with the renderer type to look up renderers that can render the component:

```
public String getFamily() {
    return ("Map");
}
```

The component family identifier, `Map`, must match that defined by the `component-family` elements included in the component and renderer configurations in the application configuration resource file. Registering a Custom Renderer with a Render Kit explains how to define the component family in the renderer configuration. Registering a Custom Component explains how to define the component family in the component configuration.

Performing Encoding

During the Render Response phase, the Jakarta Faces implementation processes the encoding methods of all components and their associated renderers in the view. The encoding methods convert the current local value of the component into the corresponding markup that represents it in the response.

The `UIComponentBase` class defines a set of methods for rendering markup: `encodeBegin`, `encodeChildren`, and `encodeEnd`. If the component has child components, you might need to use more than one of these methods to render the component; otherwise, all rendering should be done in `encodeEnd`. Alternatively, you can use the `encodeALL` method, which encompasses all the methods.

Because `MapComponent` is a parent component of `AreaComponent`, the `area` tags must be rendered after the beginning `map` tag and before the ending `map` tag. To accomplish this, the `MapRenderer` class renders the beginning `map` tag in `encodeBegin` and the rest of the `map` tag in `encodeEnd`.

The Jakarta Faces implementation automatically invokes the `encodeEnd` method of `AreaComponent`'s renderer after it invokes `MapRenderer`'s `encodeBegin` method and before it invokes `MapRenderer`'s `encodeEnd` method. If a component needs to perform the rendering for its children, it does this in the `encodeChildren` method.

Here are the `encodeBegin` and `encodeEnd` methods of `MapRenderer`:

```
@Override
public void encodeBegin(FacesContext context, UIComponent component)
        throws IOException {
    if ((context == null)|| (component == null)) {
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
```

```
    ResponseWriter writer = context.getResponseWriter();
    writer.startElement("map", map);
    writer.writeAttribute("name", map.getId(), "id");
}

@Override
public void encodeEnd(FacesContext context, UIComponent component)
        throws IOException {
    if ((context == null) || (component == null)){
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    ResponseWriter writer = context.getResponseWriter();
    writer.startElement("input", map);
    writer.writeAttribute("type", "hidden", null);
    writer.writeAttribute("name", getName(context,map), "clientId");
    writer.endElement("input");
    writer.endElement("map");
}
```

Notice that `encodeBegin` renders only the beginning `map` tag. The `encodeEnd` method renders the `input` tag and the ending `map` tag.

The encoding methods accept a `UIComponent` argument and a `jakarta.faces.context.FacesContext` argument. The `FacesContext` instance contains all the information associated with the current request. The `UIComponent` argument is the component that needs to be rendered.

The rest of the method renders the markup to the `jakarta.faces.context.ResponseWriter` instance, which writes out the markup to the current response. This basically involves passing the HTML tag names and attribute names to the `ResponseWriter` instance as strings, retrieving the values of the component attributes, and passing these values to the `ResponseWriter` instance.

The `startElement` method takes a `String` (the name of the tag) and the component to which the tag corresponds (in this case, `map`). (Passing this information to the `ResponseWriter` instance helps design-time tools know which portions of the generated markup are related to which components.)

After calling `startElement`, you can call `writeAttribute` to render the tag's attributes. The `writeAttribute` method takes the name of the attribute, its value, and the name of a property or attribute of the containing component corresponding to the attribute. The last parameter can be null, and it won't be rendered.

The `name` attribute value of the `map` tag is retrieved using the `getId` method of `UIComponent`, which returns the component's unique identifier. The `name` attribute value of the `input` tag is retrieved using the `getName(FacesContext, UIComponent)` method of `MapRenderer`.

If you want your component to perform its own rendering but delegate to a renderer if there is one, include the following lines in the encoding method to check whether there is a renderer associated with this component:

```
if (getRendererType() != null) {
    super.encodeEnd(context);
    return;
}
```

If there is a renderer available, this method invokes the superclass's `encodeEnd` method, which does the work of finding the renderer. The `MapComponent` class delegates all rendering to `MapRenderer`, so it does not need to check for available renderers.

In some custom component classes that extend standard components, you might need to implement other methods in addition to `encodeEnd`. For example, if you need to retrieve the component's value from the request parameters, you must also implement the `decode` method.

Performing Decoding

During the Apply Request Values phase, the Jakarta Faces implementation processes the `decode` methods of all components in the tree. The `decode` method extracts a component's local value from incoming request parameters and uses a `jakarta.faces.convert.Converter` implementation to convert the value to a type that is acceptable to the component class.

A custom component class or its renderer must implement the `decode` method only if it must retrieve the local value or if it needs to queue events. The component queues the event by calling `queueEvent`.

Here is the `decode` method of `MapRenderer`:

```
@Override
public void decode(FacesContext context, UIComponent component) {
    if ((context == null) || (component == null)) {
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    String key = getName(context, map);
    String value = (String) context.getExternalContext().
            getRequestParameterMap().get(key);
    if (value != null)
        map.setCurrent(value);
    }
}
```

The `decode` method first gets the name of the hidden `input` field by calling `getName(FacesContext, UIComponent)`. It then uses that name as the key to the request parameter map to retrieve the current value of the `input` field. This value represents the currently selected area. Finally, it sets the value of the `MapComponent` class's `current` attribute to the value of the `input` field.

Enabling Component Properties to Accept Expressions

Nearly all the attributes of the standard Jakarta Faces tags can accept expressions, whether they are

value expressions or method expressions. It is recommended that you also enable your component attributes to accept expressions because it gives you much more flexibility when you write Facelets pages.

To enable the attributes to accept expressions, the component class must implement getter and setter methods for the component properties. These methods can use the facilities offered by the `StateHelper` interface to store and retrieve not only the values for these properties but also the state of the components across multiple requests.

Because `MapComponent` extends `UICommand`, the `UICommand` class already does the work of getting the `ValueExpression` and `MethodExpression` instances associated with each of the attributes that it supports. Similarly, the `UIOutput` class that `AreaComponent` extends already obtains the `ValueExpression` instances for its supported attributes. For both components, the simple getter and setter methods store and retrieve the key values and state for the attributes, as shown in this code fragment from `AreaComponent`:

```
enum PropertyKeys {
    alt, coords, shape, targetImage;
}
public String getAlt() {
    return (String) getStateHelper().eval(PropertyKeys.alt, null);
}
public void setAlt(String alt) {
    getStateHelper().put(PropertyKeys.alt, alt);
}
...
```

However, if you have a custom component class that extends `UIComponentBase`, you will need to implement the methods that get the `ValueExpression` and `MethodExpression` instances associated with those attributes that are enabled to accept expressions. For example, you could include a method that gets the `ValueExpression` instance for the `immediate` attribute:

```
public boolean isImmediate() {
    if (this.immediateSet) {
        return (this.immediate);
    }
    ValueExpression ve = getValueExpression("immediate");
    if (ve != null) {
        Boolean value = (Boolean) ve.getValue(
            getFacesContext().getELContext());
        return (value.booleanValue());
    } else {
        return (this.immediate);
    }
}
```

The properties corresponding to the component attributes that accept method expressions must accept and return a `MethodExpression` object. For example, if `MapComponent` extended `UIComponentBase`

instead of `UICommand`, it would need to provide an `action` property that returns and accepts a `MethodExpression` object:

```
public MethodExpression getAction() {
    return (this.action);
}
public void setAction(MethodExpression action) {
    this.action = action;
}
```

Saving and Restoring State

As described in Enabling Component Properties to Accept Expressions, use of the `StateHelper` interface facilities allows you to save the component's state at the same time you set and retrieve property values. The `StateHelper` implementation allows partial state saving; it saves only the changes in the state since the initial request, not the entire state, because the full state can be restored during the Restore View phase.

Component classes that implement `StateHolder` may prefer to implement the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods to help the Jakarta Faces implementation save and restore the state of components across multiple requests.

To save a set of values, you can implement the `saveState(FacesContext)` method. This method is called during the Render Response phase, during which the state of the response is saved for processing on subsequent requests. Here is a hypothetical method from `MapComponent`, which has only one attribute, `current`:

```
@Override
public Object saveState(FacesContext context) {
    Object values[] = new Object[2];
    values[0] = super.saveState(context);
    values[1] = current;
    return (values);
}
```

This method initializes an array, which will hold the saved state. It next saves all of the state associated with the component.

A component that implements `StateHolder` may also provide an implementation for `restoreState(FacesContext, Object)`, which restores the state of the component to that saved with the `saveState(FacesContext)` method. The `restoreState(FacesContext, Object)` method is called during the Restore View phase, during which the Jakarta Faces implementation checks whether there is any state that was saved during the last Render Response phase and needs to be restored in preparation for the next postback.

Here is a hypothetical `restoreState(FacesContext, Object)` method from `MapComponent`:

```
public void restoreState(FacesContext context, Object state) {
    Object values[] = (Object[]) state;
    super.restoreState(context, values[0]);
    current = (String) values[1];
}
```

This method takes a `FacesContext` and an `Object` instance, representing the array that is holding the state for the component. This method sets the component's properties to the values saved in the `Object` array.

Whether or not you implement these methods in your component class, you can use the `jakarta.faces.STATE_SAVING_METHOD` context parameter to specify in the deployment descriptor where you want the state to be saved: either `client` or `server`. If state is saved on the client, the state of the entire view is rendered to a hidden field on the page. By default, the state is saved on the server.

The web applications in the Duke's Forest case study save their view state on the client.

Saving state on the client uses more bandwidth as well as more client resources, whereas saving it on the server uses more server resources. You may also want to save state on the client if you expect your users to disable cookies.

Delegating Rendering to a Renderer

Both `MapComponent` and `AreaComponent` delegate all of their rendering to a separate renderer. The section Performing Encoding explains how `MapRenderer` performs the encoding for `MapComponent`. This section explains in detail the process of delegating rendering to a renderer using `AreaRenderer`, which performs the rendering for `AreaComponent`.

To delegate rendering, you perform the tasks described in the following topics:

- Creating the Renderer Class
- Identifying the Renderer Type

Creating the Renderer Class

When delegating rendering to a renderer, you can delegate all encoding and decoding to the renderer, or you can choose to do part of it in the component class. The `AreaComponent` class delegates encoding to the `AreaRenderer` class.

The renderer class begins with a `@FacesRenderer` annotation:

```
@FacesRenderer(componentFamily = "Area", rendererType = "DemoArea")
public class AreaRenderer extends Renderer { }
```

The `@FacesRenderer` annotation registers the renderer class with the Jakarta Faces implementation as a renderer class. The annotation identifies the component family as well as the renderer type.

To perform the rendering for AreaComponent, AreaRenderer must implement an encodeEnd method. The encodeEnd method of AreaRenderer retrieves the shape, coordinates, and alternative text values stored in the ImageArea bean that is bound to AreaComponent. Suppose that the area tag currently being rendered has a value attribute value of "book203". The following line from encodeEnd gets the value of the attribute "book203" from the FacesContext instance:

```
ImageArea ia = (ImageArea)area.getValue();
```

The attribute value is the ImageArea bean instance, which contains the shape, coords, and alt values associated with the book203 AreaComponent instance.

After retrieving the ImageArea object, the method renders the values for shape, coords, and alt by simply calling the associated accessor methods and passing the returned values to the ResponseWriter instance, as shown by these lines of code, which write out the shape and coordinates:

```
writer.startElement("area", area);
writer.writeAttribute("alt", iarea.getAlt(), "alt");
writer.writeAttribute("coords", iarea.getCoords(), "coords");
writer.writeAttribute("shape", iarea.getShape(), "shape");
```

The encodeEnd method also renders the JavaScript for the onmouseout, onmouseover, and onclick attributes. The Facelets page needs to provide only the path to the images that are to be loaded during an onmouseover or onmouseout action:

```
<bookstore:area id="map3" value="#{Book203}"
                onmouseover="resources/images/book_203.jpg"
                onmouseout="resources/images/book_all.jpg"
                targetImage="mapImage"/>
```

The AreaRenderer class takes care of generating the JavaScript for these actions, as shown in the following code from encodeEnd. The JavaScript that AreaRenderer generates for the onclick action sets the value of the hidden field to the value of the current area's component ID and submits the page.

```
sb = new StringBuffer("document.forms[0]['").append(targetImageId).
        append("'].src='");
sb.append(
        getURI(context,
        (String) area.getAttributes().get("onmouseout")));
sb.append("'");
writer.writeAttribute("onmouseout", sb.toString(), "onmouseout");
sb = new StringBuffer("document.forms[0]['").append(targetImageId).
        append("'].src='");
sb.append(
        getURI(context,
```

```
        (String) area.getAttributes().get("onmouseover")));
sb.append("'");
writer.writeAttribute("onmouseover", sb.toString(), "onmouseover");
sb = new StringBuffer("document.forms[0]['");
sb.append(getName(context, area));
sb.append("'].value='");
sb.append(iarea.getAlt());
sb.append("'; document.forms[0].submit()");
writer.writeAttribute("onclick", sb.toString(), "value");
writer.endElement("area");
```

By submitting the page, this code causes the Jakarta Faces lifecycle to return back to the Restore View phase. This phase saves any state information, including the value of the hidden field, so that a new request component tree is constructed. This value is retrieved by the `decode` method of the `MapComponent` class. This decode method is called by the Jakarta Faces implementation during the Apply Request Values phase, which follows the Restore View phase.

In addition to the `encodeEnd` method, `AreaRenderer` contains an empty constructor. This is used to create an instance of `AreaRenderer` so that it can be added to the render kit.

The `@FacesRenderer` annotation registers the renderer class with the Jakarta Faces implementation as a renderer class. The annotation identifies the component family as well as the renderer type.

Identifying the Renderer Type

Register the renderer with a render kit by using the `@FacesRenderer` annotation (or by using the application configuration resource file, as explained in Registering a Custom Renderer with a Render Kit). During the Render Response phase, the Jakarta Faces implementation calls the `getRendererType` method of the component's tag handler to determine which renderer to invoke, if there is one.

You identify the type associated with the renderer in the `rendererType` element of the `@FacesRenderer` annotation for `AreaRenderer` as well as in the `renderer-type` element of the tag library descriptor.

Implementing an Event Listener

The Jakarta Faces technology supports action events and value-change events for components.

Action events occur when the user activates a component that implements `jakarta.faces.component.ActionSource`. These events are represented by the class `jakarta.faces.event.ActionEvent`.

Value-change events occur when the user changes the value of a component that implements `jakarta.faces.component.EditableValueHolder`. These events are represented by the class `jakarta.faces.event.ValueChangeEvent`.

One way to handle events is to implement the appropriate listener classes. Listener classes that handle the action events in an application must implement the interface `jakarta.faces.event.ActionListener`. Similarly, listeners that handle the value-change events must implement the interface `jakarta.faces.event.ValueChangeListener`.

This section explains how to implement the two listener classes.

To handle events generated by custom components, you must implement an event listener and an event handler and manually queue the event on the component. See Handling Events for Custom Components for more information.



You do not need to create an `ActionListener` implementation to handle an event that results solely in navigating to a page and does not perform any other application-specific processing. See Writing a Method to Handle Navigation for information on how to manage page navigation.

Implementing Value-Change Listeners

A `jakarta.faces.event.ValueChangeListener` implementation must include a `processValueChange(ValueChangeEvent)` method. This method processes the specified value-change event and is invoked by the Jakarta Faces implementation when the value-change event occurs. The `ValueChangeEvent` instance stores the old and the new values of the component that fired the event.

In the Duke's Bookstore case study, the `NameChanged` listener implementation is registered on the `name UIInput` component on the `bookcashier.xhtml` page. This listener stores into session scope the name the user entered in the field corresponding to the name component.

The `bookreceipt.xhtml` subsequently retrieves the name from the session scope:

```
<h:outputFormat title="thanks"
                value="#{bundle.ThankYouParam}">
    <f:param value="#{sessionScope.name}"/>
</h:outputFormat>
```

When the `bookreceipt.xhtml` page is loaded, it displays the name inside the message:

```
"Thank you, {0}, for purchasing your books from us."
```

Here is part of the `NameChanged` listener implementation:

```
public class NameChanged extends Object implements ValueChangeListener {

    @Override
    public void processValueChange(ValueChangeEvent event)
            throws AbortProcessingException {

        if (null != event.getNewValue()) {
            FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put("name", event.getNewValue());
        }
    }
}
```

When the user enters the name in the field, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the `NameChanged` listener implementation is invoked. This method first gets the ID of the component that fired the event from the `ValueChangeEvent` object, and it puts the value, along with an attribute name, into the session map of the `FacesContext` instance.

Registering a Value-Change Listener on a Component explains how to register this listener onto a component.

Implementing Action Listeners

A `jakarta.faces.event.ActionListener` implementation must include a `processAction(ActionEvent)` method. The `processAction(ActionEvent)` method processes the specified action event. The Jakarta Faces implementation invokes the `processAction(ActionEvent)` method when the `ActionEvent` occurs.

The Duke's Bookstore case study uses two `ActionListener` implementations, `LinkBookChangeListener` and `MapBookChangeListener`. See Handling Events for Custom Components for details on `MapBookChangeListener`.

Registering an Action Listener on a Component explains how to register this listener onto a component.

Handling Events for Custom Components

As explained in Implementing an Event Listener, events are automatically queued on standard components that fire events. A custom component, on the other hand, must manually queue events from its `decode` method if it fires events.

Performing Decoding explains how to queue an event on `MapComponent` using its `decode` method. This section explains how to write the class that represents the event of clicking on the map and how to write the method that processes this event.

As explained in Understanding the Facelets Page, the `actionListener` attribute of the `bookstore:map` tag points to the `MapBookChangeListener` class. The listener class's `processAction` method processes the event of clicking the image map. Here is the `processAction` method:

```
@Override
public void processAction(ActionEvent actionEvent)
        throws AbortProcessingException {

    AreaSelectedEvent event = (AreaSelectedEvent) actionEvent;
    String current = event.getMapComponent().getCurrent();
    FacesContext context = FacesContext.getCurrentInstance();
    String bookId = books.get(current);
    context.getExternalContext().getSessionMap().put("bookId", bookId);
}
```

When the Jakarta Faces implementation calls this method, it passes in an `ActionEvent` object that represents the event generated by clicking on the image map. Next, it casts it to an

AreaSelectedEvent object (see jakartaee-examples/tutorial/case-studies/dukes-bookstore/src/main/java/jakarta/tutorial/dukesbookstore/listeners/AreaSelectedEvent.java). Then this method gets the MapComponent associated with the event. Next, it gets the value of the MapComponent object's current attribute, which indicates the currently selected area. The method then uses the value of the current attribute to get the book's ID value from a HashMap object, which is constructed elsewhere in the MapBookChangeListener class. Finally, the method places the ID obtained from the HashMap object into the session map for the application.

In addition to the method that processes the event, you need the event class itself. This class is very simple to write; you have it extend ActionEvent and provide a constructor that takes the component on which the event is queued and a method that returns the component. Here is the AreaSelectedEvent class used with the image map:

```
public class AreaSelectedEvent extends ActionEvent {
    public AreaSelectedEvent(MapComponent map) {
        super(map);
    }
    public MapComponent getMapComponent() {
        return ((MapComponent) getComponent());
    }
}
```

As explained in the section Creating Custom Component Classes, in order for MapComponent to fire events in the first place, it must implement ActionSource. Because MapComponent extends UICommand, it also implements ActionSource.

Defining the Custom Component Tag in a Tag Library Descriptor

To use a custom tag, you declare it in a Tag Library Descriptor (TLD). The TLD file defines how the custom tag is used in a Jakarta Faces page. The web container uses the TLD to validate the tag. The set of tags that are part of the HTML render kit are defined in the HTML_BASIC TLD, available in the Jakarta Faces standard HTML tag library.

The TLD file name must end with taglib.xml. In the Duke's Bookstore case study, the custom tags area and map are defined in the file web/WEB-INF/bookstore.taglib.xml.

All tag definitions must be nested inside the facelet-taglib element in the TLD. Each tag is defined by a tag element. Here are the tag definitions for the area and map components:

```
<facelet-taglib xmlns="https://jakarta.ee/xml/ns/jakartaee"
...>
    <namespace>http://dukesbookstore</namespace>
    <tag>
        <tag-name>area</tag-name>
        <component>
            <component-type>DemoArea</component-type>
            <renderer-type>DemoArea</renderer-type>
        </component>
```

```
    </tag>
    <tag>
        <tag-name>map</tag-name>
        <component>
            <component-type>DemoMap</component-type>
            <renderer-type>DemoMap</renderer-type>
        </component>
    </tag>
</facelet-taglib>
```

The `component-type` element specifies the name defined in the `@FacesComponent` annotation, and the `renderer-type` element specifies the `rendererType` defined in the `@FacesRenderer` annotation.

The `facelet-taglib` element must also include a `namespace` element, which defines the namespace to be specified in pages that use the custom component. See Using a Custom Component for information on specifying the namespace in pages.

The TLD file is located in the `WEB-INF` directory. In addition, an entry is included in the web deployment descriptor (`web.xml`) to identify the custom tag library descriptor file, as follows:

```
    <context-param>
        <param-name>jakarta.faces.FACELETS_LIBRARIES</param-name>
        <param-value>/WEB-INF/bookstore.taglib.xml</param-value>
    </context-param>
```

Using a Custom Component

To use a custom component in a page, you add the custom tag associated with the component to the page.

As explained in Defining the Custom Component Tag in a Tag Library Descriptor, you must ensure that the TLD that defines any custom tags is packaged in the application if you intend to use the tags in your pages. TLD files are stored in the `WEB-INF/` directory or subdirectory of the WAR file or in the `META-INF/` directory or subdirectory of a tag library packaged in a JAR file.

You also need to include a namespace declaration in the page so that the page has access to the tags. The custom tags for the Duke's Bookstore case study are defined in `bookstore.taglib.xml`. The `ui:composition` tag on the `index.xhtml` page declares the namespace defined in the tag library:

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="jakarta.faces.facelets"
                xmlns:h="jakarta.faces.html"
                xmlns:f="jakarta.faces.core"
                xmlns:bookstore="http://dukesbookstore"
                template="./bookstoreTemplate.xhtml">
```

Finally, to use a custom component in a page, you add the component's tag to the page.

The Duke's Bookstore case study includes a custom image map component on the `index.xhtml` page. This component allows you to select a book by clicking on a region of the image map:

```
...
<h:graphicImage id="mapImage"
                name="book_all.jpg"
                library="images"
                alt="#{bundle.chooseLocale}"
                usemap="#bookMap" />
<bookstore:map id="bookMap"
               current="map1"
               immediate="true"
               action="bookstore">
    <f:actionListener
        type="ee.jakarta.tutorial.dukesbookstore.listeners.MapBookChangeListener" />
    <bookstore:area id="map1" value="#{Book201}"
                    onmouseover="resources/images/book_201.jpg"
                    onmouseout="resources/images/book_all.jpg"
                    targetImage="mapImage" />
    ...
    <bookstore:area id="map6" value="#{Book207}"
                    onmouseover="resources/images/book_207.jpg"
                    onmouseout="resources/images//book_all.jpg"
                    targetImage="mapImage" />
</bookstore:map>
```

The standard `h:graphicImage` tag associates an image (`book_all.jpg`) with an image map that is referenced in the `usemap` attribute value.

The custom `bookstore:map` tag that represents the custom component, `MapComponent`, specifies the image map and contains a set of `bookstore:area` tags. Each custom `bookstore:area` tag represents a custom `AreaComponent` and specifies a region of the image map.

On the page, the `onmouseover` and `onmouseout` attributes specify the image that is displayed when the user performs the actions described by the attributes. The custom renderer also renders an `onclick` attribute.

In the rendered HTML page, the `onmouseover`, `onmouseout`, and `onclick` attributes define which JavaScript code is executed when these events occur. When the user moves the mouse over a region, the `onmouseover` function associated with the region displays the map with that region highlighted. When the user moves the mouse out of a region, the `onmouseout` function redisplays the original image. When the user clicks a region, the `onclick` function sets the value of a hidden `input` tag to the ID of the selected area and submits the page.

When the custom renderer renders these attributes in HTML, it also renders the JavaScript code. The custom renderer also renders the entire `onclick` attribute rather than letting the page author set it.

The custom renderer that renders the HTML `map` tag also renders a hidden `input` component that

holds the current area. The server-side objects retrieve the value of the hidden `input` field and set the locale in the `FacesContext` instance according to which region was selected.

Creating and Using a Custom Converter

A Jakarta Faces converter class converts strings to objects and objects to strings as required. Several standard converters are provided by Jakarta Faces for this purpose. See Using the Standard Converters for more information on these included converters.

As explained in Conversion Model, if the standard converters included with Jakarta Faces cannot perform the data conversion that you need, you can create a custom converter to perform this specialized conversion. This implementation, at a minimum, must define how to convert data both ways between the two views of the data described in Conversion Model.

All custom converters must implement the `jakarta.faces.convert.Converter` interface. This section explains how to implement this interface to perform a custom data conversion.

The Duke's Bookstore case study uses a custom `Converter` implementation, located in `jakarta-examples/tutorial/case-studies/dukes-bookstore/src/main/java/jakarta/tutorial/dukesbookstore/converters/CreditCardConverter.java`, to convert the data entered in the Credit Card Number field on the `bookcashier.xhtml` page. It strips blanks and hyphens from the text string and formats it so that a blank space separates every four characters.

Another common use case for a custom converter is in a list for a nonstandard object type. In the Duke's Tutoring case study, the `Student` and `Guardian` entities require a custom converter so that they can be converted to and from a `UISelectItems` input component.

Creating a Custom Converter

The `CreditCardConverter` custom converter class is created as follows:

```
@FacesConverter("ccno")
public class CreditCardConverter implements Converter {
    ...
}
```

The `@FacesConverter` annotation registers the custom converter class as a converter with the name of `ccno` with the Jakarta Faces implementation. Alternatively, you can register the converter with entries in the application configuration resource file, as shown in Registering a Custom Converter.

To define how the data is converted from the presentation view to the model view, the `Converter` implementation must implement the `getAsObject(FacesContext, UIComponent, String)` method from the `Converter` interface. Here is the implementation of this method from `CreditCardConverter`:

```
@Override
public Object getAsObject(FacesContext context,
        UIComponent component, String newValue)
        throws ConverterException {
```

```
    if (newValue.isEmpty()) {
        return null;
    }
    // Since this is only a String to String conversion,
    // this conversion does not throw ConverterException.

    String convertedValue = newValue.trim();
    if ( (convertedValue.contains("-")) || (convertedValue.contains(" "))) {
        char[] input = convertedValue.toCharArray();
        StringBuilder builder = new StringBuilder(input.length);
        for (int i = 0; i < input.length; ++i) {
            if ((input[i] == '-') || (input[i] == ' ')) {
            } else {
                builder.append(input[i]);
            }
        }
        convertedValue = builder.toString();
    }
    return convertedValue;
}
```

During the Apply Request Values phase, when the components' `decode` methods are processed, the Jakarta Faces implementation looks up the component's local value in the request and calls the `getAsObject` method. When calling this method, the Jakarta Faces implementation passes in the current `FacesContext` instance, the component whose data needs conversion, and the local value as a `String`. The method then writes the local value to a character array, trims the hyphens and blanks, adds the rest of the characters to a `String`, and returns the `String`.

To define how the data is converted from the model view to the presentation view, the `Converter` implementation must implement the `getAsString(FacesContext, UIComponent, Object)` method from the `Converter` interface. Here is an implementation of this method:

```
@Override
public String getAsString(FacesContext context,
        UIComponent component, Object value)
        throws ConverterException {

    String inputVal = null;
    if ( value == null ) {
        return "";
    }
    // value must be of a type that can be cast to a String.
    try {
        inputVal = (String)value;
    } catch (ClassCastException ce) {
        FacesMessage errMsg = new FacesMessage(CONVERSION_ERROR_MESSAGE_ID);
        FacesContext.getCurrentInstance().addMessage(null, errMsg);
        throw new ConverterException(errMsg.getSummary());
```

```
    }
    // insert spaces after every four characters for better
    // readability if they are not already present.
    char[] input = inputVal.toCharArray();
    StringBuilder builder = new StringBuilder(input.length + 3);
    for (int i = 0; i < input.length; ++i) {
        if ((i % 4) == 0 && (i != 0)) {
            if ((input[i] != ' ') || (input[i] != '-')){
                builder.append(" ");
                // if there are any "-"'s convert them to blanks.
            } else if (input[i] == '-') {
                builder.append(" ");
            }
         }
         builder.append(input[i]);
    }
    String convertedValue = builder.toString();
    return convertedValue;
}
```

During the Render Response phase, in which the components' `encode` methods are called, the Jakarta Faces implementation calls the `getAsString` method in order to generate the appropriate output. When the Jakarta Faces implementation calls this method, it passes in the current `FacesContext`, the `UIComponent` whose value needs to be converted, and the bean value to be converted. Because this converter does a `String`-to-`String` conversion, this method can cast the bean value to a `String`.

If the value cannot be converted to a `String`, the method throws an exception, passing an error message from the resource bundle that is registered with the application. Registering Application Messages explains how to register custom error messages with the application.

If the value can be converted to a `String`, the method reads the `String` to a character array and loops through the array, adding a space after every four characters.

You can also create a custom converter with a `@FacesConverter` annotation that specifies the `forClass` attribute, as shown in the following example from the Duke's Tutoring case study:

```
@FacesConverter(forClass=Guardian.class, value="guardian")
public class GuardianConverter extends EntityConverter implements Converter { ... }
```

The `forClass` attribute registers the converter as the default converter for the `Guardian` class. Therefore, whenever that class is specified by a `value` attribute of an input component, the converter is invoked automatically.

A converter class can be a separate Java POJO class, as in the Duke's Bookstore case study. If it needs to access objects defined in a managed bean class, however, it can be a subclass of a Jakarta Faces managed bean, as in the `address-book` persistence example, in which the converters use an enterprise bean that is injected into the managed bean class.

Using a Custom Converter

To apply the data conversion performed by a custom converter to a particular component's value, you must do one of the following.

- Reference the converter from the component tag's `converter` attribute.
- Nest an `f:converter` tag inside the component's tag and reference the custom converter from one of the `f:converter` tag's attributes.

If you are using the component tag's `converter` attribute, this attribute must reference the `Converter` implementation's identifier or the fully-qualified class name of the converter. Creating and Using a Custom Converter explains how to implement a custom converter.

The identifier for the credit card converter class is `ccno`, the value specified in the `@FacesConverter` annotation:

```
@FacesConverter("ccno")
public class CreditCardConverter implements Converter {
    ...
}
```

Therefore, the `CreditCardConverter` instance can be registered on the `ccno` component as shown in the following example:

```
<h:inputText id="ccno"
             size="19"
             converter="ccno"
             value="#{cashierBean.creditCardNumber}"
             required="true"
             requiredMessage="#{bundle.ReqCreditCard}">
    ...
</h:inputText>
```

By setting the `converter` attribute of a component's tag to the converter's identifier or its class name, you cause that component's local value to be automatically converted according to the rules specified in the `Converter` implementation.

Instead of referencing the converter from the component tag's `converter` attribute, you can reference the converter from an `f:converter` tag nested inside the component's tag. To reference the custom converter using the `f:converter` tag, you do one of the following.

- Set the `f:converter` tag's `converterId` attribute to the `Converter` implementation's identifier defined in the `@FacesConverter` annotation or in the application configuration resource file. This method is shown in `bookcashier.xhtml`:

  ```
  <h:inputText id="ccno"
               size="19"
               value="#{cashierBean.creditCardNumber}"
  ```

```
            required="true"
            requiredMessage="#{bundle.ReqCreditCard}">
    <f:converter converterId="ccno"/>
    <f:validateRegex
       pattern="\d{16}|\d{4} \d{4} \d{4} \d{4}|\d{4}-\d{4}-\d{4}-\d{4}"/>
</h:inputText>
```

- Bind the `Converter` implementation to a managed bean property using the `f:converter` tag's `binding` attribute, as described in Binding Converters, Listeners, and Validators to Managed Bean Properties.

The Jakarta Faces implementation calls the converter's `getAsObject` method to strip spaces and hyphens from the input value. The `getAsString` method is called when the `bookcashier.xhtml` page is redisplayed; this happens if the user orders more than $100 worth of books.

In the Duke's Tutoring case study, each converter is registered as the converter for a particular class. The converter is automatically invoked whenever that class is specified by a `value` attribute of an input component. In the following example, the `itemValue` attribute calls the converter for the `Guardian` class:

```
<h:selectManyListbox id="selectGuardiansMenu"
                     title="#{bundle['action.add.guardian']}"
                     value="#{guardianManager.selectedGuardians}"
                     size="5"
                     converter="guardian">
    <f:selectItems value="#{guardianManager.allGuardians}"
                   var="selectedGuardian"
                   itemLabel="#{selectedGuardian.name}"
                   itemValue="#{selectedGuardian}" />
</h:selectManyListbox>
```

Creating and Using a Custom Validator

If the standard validators or Bean Validation don't perform the validation checking you need, you can create a custom validator to validate user input. As explained in Validation Model, there are two ways to implement validation code.

- Implement a managed bean method that performs the validation.
- Provide an implementation of the `jakarta.faces.validator.Validator` interface to perform the validation.

Writing a Method to Perform Validation explains how to implement a managed bean method to perform validation. The rest of this section explains how to implement the `Validator` interface.

If you choose to implement the `Validator` interface and you want to allow the page author to configure the validator's attributes from the page, you also must specify a custom tag for registering the validator on a component.

If you prefer to configure the attributes in the `Validator` implementation, you can forgo specifying a custom tag and instead let the page author register the validator on a component using the `f:validator` tag, as described in Using a Custom Validator.

You can also create a managed bean property that accepts and returns the `Validator` implementation you create, as described in Writing Properties Bound to Converters, Listeners, or Validators. You can use the `f:validator` tag's binding attribute to bind the `Validator` implementation to the managed bean property.

Usually, you will want to display an error message when data fails validation. You need to store these error messages in a resource bundle.

After creating the resource bundle, you have two ways to make the messages available to the application. You can queue the error messages onto the `FacesContext` programmatically, or you can register the error messages in the application configuration resource file, as explained in Registering Application Messages.

For example, an e-commerce application might use a general-purpose custom validator called `FormatValidator.java` to validate input data against a format pattern that is specified in the custom validator tag. This validator would be used with a Credit Card Number field on a Facelets page. Here is the custom validator tag:

```
<mystore:formatValidator
 formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
```

According to this validator, the data entered in the field must be one of the following:

- A 16-digit number with no spaces
- A 16-digit number with a space between every four digits
- A 16-digit number with hyphens between every four digits

The `f:validateRegex` tag makes a custom validator unnecessary in this situation. However, the rest of this section describes how this validator would be implemented and how to specify a custom tag so that the page author could register the validator on a component.

Implementing the Validator Interface

A `Validator` implementation must contain a constructor, a set of accessor methods for any attributes on the tag, and a `validate` method, which overrides the `validate` method of the `Validator` interface.

The hypothetical `FormatValidator` class also defines accessor methods for setting the `formatPatterns` attribute, which specifies the acceptable format patterns for input into the fields. The setter method calls the `parseFormatPatterns` method, which separates the components of the pattern string into a string array, `formatPatternsList`.

```
public String getFormatPatterns() {
    return (this.formatPatterns);
```

```
}
public void setFormatPatterns(String formatPatterns) {
    this.formatPatterns = formatPatterns;
    parseFormatPatterns();
}
```

In addition to defining accessor methods for the attributes, the class overrides the `validate` method of the `Validator` interface. This method validates the input and also accesses the custom error messages to be displayed when the `String` is invalid.

The `validate` method performs the actual validation of the data. It takes the `FacesContext` instance, the component whose data needs to be validated, and the value that needs to be validated. A validator can validate only data of a component that implements `jakarta.faces.component.EditableValueHolder`.

Here is an implementation of the `validate` method:

```
@FacesValidator
public class FormatValidator implements Validator, StateHolder {
    ...
    public void validate(FacesContext context, UIComponent component,
                         Object toValidate) {

        boolean valid = false;
        String value = null;
        if ((context == null) || (component == null)) {
            throw new NullPointerException();
        }
        if (!(component instanceof UIInput)) {
            return;
        }
        if ( null == formatPatternsList || null == toValidate) {
            return;
        }
        value = toValidate.toString();
        // validate the value against the list of valid patterns.
        Iterator patternIt = formatPatternsList.iterator();
        while (patternIt.hasNext()) {
            valid = isFormatValid(
                ((String)patternIt.next()), value);
            if (valid) {
                break;
            }
        }
        if ( !valid ) {
            FacesMessage errMsg =
                new FacesMessage(FORMAT_INVALID_MESSAGE_ID);
            FacesContext.getCurrentInstance().addMessage(null, errMsg);
            throw new ValidatorException(errMsg);
        }
```

```
    }
}
```

The `@FacesValidator` annotation registers the `FormatValidator` class as a validator with the Jakarta Faces implementation. The `validate` method gets the local value of the component and converts it to a `String`. It then iterates over the `formatPatternsList` list, which is the list of acceptable patterns that was parsed from the `formatPatterns` attribute of the custom validator tag.

While iterating over the list, this method checks the pattern of the component's local value against the patterns in the list. If the pattern of the local value does not match any pattern in the list, this method generates an error message. It then creates a `jakarta.faces.application.FacesMessage` and queues it on the `FacesContext` for display, using a `String` that represents the key in the `Properties` file:

```
public static final String FORMAT_INVALID_MESSAGE_ID =
     "FormatInvalid";
}
```

Finally, the method passes the message to the constructor of `jakarta.faces.validator.ValidatorException`.

When the error message is displayed, the format pattern will be substituted for the `{0}` in the error message, which, in English, is as follows:

```
Input must match one of the following patterns: {0}
```

You may wish to save and restore state for your validator, although state saving is not usually necessary. To do so, you will need to implement the `StateHolder` interface as well as the `Validator` interface. To implement `StateHolder`, you would need to implement its four methods: `saveState(FacesContext)`, `restoreState(FacesContext, Object)`, `isTransient`, and `setTransient(boolean)`. See Saving and Restoring State for more information.

Specifying a Custom Tag

If you implemented a `Validator` interface rather than implementing a managed bean method that performs the validation, you need to do one of the following.

- Allow the page author to specify the `Validator` implementation to use with the `f:validator` tag. In this case, the `Validator` implementation must define its own properties. Using a Custom Validator explains how to use the `f:validator` tag.
- Specify a custom tag that provides attributes for configuring the properties of the validator from the page.

To create a custom tag, you need to add the tag to the tag library descriptor for the application, `bookstore.taglib.xml`:

```
<tag>
    <tag-name>validator</tag-name>
    <validator>
        <validator-id>formatValidator</validator-id>
        <validator-class>
            dukesbookstore.validators.FormatValidator
        </validator-class>
    </validator>
</tag>
```

The `tag-name` element defines the name of the tag as it must be used in a Facelets page. The `validator-id` element identifies the custom validator. The `validator-class` element wires the custom tag to its implementation class.

Using a Custom Validator explains how to use the custom validator tag on the page.

Using a Custom Validator

To register a custom validator on a component, you must do one of the following.

- Nest the validator's custom tag inside the tag of the component whose value you want to be validated.
- Nest the standard `f:validator` tag within the tag of the component and reference the custom `Validator` implementation from the `f:validator` tag.

Here is a hypothetical custom `formatValidator` tag for the Credit Card Number field, nested within the `h:inputText` tag:

```
<h:inputText id="ccno" size="19"
  ...
  required="true">
  <mystore:formatValidator
  formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
</h:inputText>
<h:message styleClass="validationMessage" for="ccno"/>
```

This tag validates the input of the `ccno` field against the patterns defined by the page author in the `formatPatterns` attribute.

You can use the same custom validator for any similar component by simply nesting the custom validator tag within the component tag.

If the application developer who created the custom validator prefers to configure the attributes in the `Validator` implementation rather than allow the page author to configure the attributes from the page, the developer will not create a custom tag for use with the validator.

In this case, the page author must nest the `f:validator` tag inside the tag of the component whose data needs to be validated. Then the page author needs to do one of the following.

- Set the `f:validator` tag's `validatorId` attribute to the ID of the validator that is defined in the application configuration resource file.
- Bind the custom `Validator` implementation to a managed bean property using the `f:validator` tag's `binding` attribute, as described in Binding Converters, Listeners, and Validators to Managed Bean Properties.

The following tag registers a hypothetical validator on a component using an `f:validator` tag and references the ID of the validator:

```
<h:inputText id="name" value="#{CustomerBean.name}"
             size="10" ...>
    <f:validator validatorId="customValidator" />
    ...
</h:inputText>
```

Binding Component Values and Instances to Managed Bean Properties

A component tag can wire its data to a managed bean by one of the following methods:

- Binding its component's value to a bean property
- Binding its component's instance to a bean property

To bind a component's value to a managed bean property, a component tag's `value` attribute uses an EL value expression. To bind a component instance to a bean property, a component tag's `binding` attribute uses a value expression.

When a component instance is bound to a managed bean property, the property holds the component's local value. Conversely, when a component's value is bound to a managed bean property, the property holds the value stored in the managed bean. This value is updated with the local value during the Update Model Values phase of the lifecycle. There are advantages to both of these methods.

Binding a component instance to a bean property has the following advantages.

- The managed bean can programmatically modify component attributes.
- The managed bean can instantiate components rather than let the page author do so.

Binding a component's value to a bean property has the following advantages.

- The page author has more control over the component attributes.
- The managed bean has no dependencies on the Jakarta Faces API (such as the component classes), allowing for greater separation of the presentation layer from the model layer.
- The Jakarta Faces implementation can perform conversions on the data based on the type of the bean property without the developer needing to apply a converter.

In most situations, you will bind a component's value rather than its instance to a bean property. You'll need to use a component binding only when you need to change one of the component's

attributes dynamically. For example, if an application renders a component only under certain conditions, it can set the component's `rendered` property accordingly by accessing the property to which the component is bound.

When referencing the property using the component tag's `value` attribute, you need to use the proper syntax. For example, suppose a managed bean called `MyBean` has this `int` property:

```
protected int currentOption = null;
public int getCurrentOption(){...}
public void setCurrentOption(int option){...}
```

The `value` attribute that references this property must have this value-binding expression:

```
#{myBean.currentOption}
```

In addition to binding a component's value to a bean property, the `value` attribute can specify a literal value or can map the component's data to any primitive (such as `int`), structure (such as an array), or collection (such as a list), independent of a JavaBeans component. Examples of Value-Binding Expressions lists some example value-binding expressions that you can use with the `value` attribute.

Examples of Value-Binding Expressions

Value	Expression
A Boolean	`cart.numberOfItems > 0`
A property initialized from a context initialization parameter	`initParam.quantity`
A bean property	`cashierBean.name`
A value in an array	`books[3]`
A value in a collection	`books["fiction"]`
A property of an object in an array of objects	`books[3].price`

The next two sections explain how to use the `value` attribute to bind a component's value to a bean property or other data objects and how to use the `binding` attribute to bind a component instance to a bean property.

Binding a Component Value to a Property

To bind a component's value to a managed bean property, you specify the name of the bean and the property using the `value` attribute.

This means that the first part of the EL value expression must match the name of the managed bean up to the first period (`.`) and the part of the value expression after the period must match the property of the managed bean.

For example, in the Duke's Bookstore case study, the `h:dataTable` tag in `bookcatalog.xhtml` sets the value of the component to the value of the `books` property of the `BookstoreBean` backing bean, whose name is `store`:

```
<h:dataTable id="books"
             value="#{store.books}"
             var="book"
             headerClass="list-header"
             styleClass="list-background"
             rowClasses="list-row-even, list-row-odd"
             border="1"
             summary="#{bundle.BookCatalog}">
```

The value is obtained by calling the backing bean's `getBooks` method, which in turn calls the `BookRequestBean` session bean's `getBooks` method.

Binding a Component Value to an Implicit Object

One external data source that a `value` attribute can refer to is an implicit object.

The `bookreceipt.xhtml` page of the Duke's Bookstore case study has a reference to an implicit object:

```
<h:outputFormat title="thanks"
                value="#{bundle.ThankYouParam}">
    <f:param value="#{sessionScope.name}"/>
</h:outputFormat>
```

This tag gets the name of the customer from the session scope and inserts it into the parameterized message at the key `ThankYouParam` from the resource bundle. For example, if the name of the customer is Gwen Canigetit, this tag will render:

```
Thank you, Gwen Canigetit, for purchasing your books from us.
```

Retrieving values from other implicit objects is done in a similar way to the example shown in this section. Implicit Objects lists the implicit objects to which a value attribute can refer. All of the implicit objects, except for the scope objects, are read-only and therefore should not be used as values for a `UIInput` component.

Implicit Objects

Implicit Object	What It Is
`applicationScope`	A `Map` of the application scope attribute values, keyed by attribute name
`cookie`	A `Map` of the cookie values for the current request, keyed by cookie name

Implicit Object	What It Is
facesContext	The FacesContext instance for the current request
header	A Map of HTTP header values for the current request, keyed by header name
headerValues	A Map of String arrays containing all the header values for HTTP headers in the current request, keyed by header name
initParam	A Map of the context initialization parameters for this web application
param	A Map of the request parameters for this request, keyed by parameter name
paramValues	A Map of String arrays containing all the parameter values for request parameters in the current request, keyed by parameter name
requestScope	A Map of the request attributes for this request, keyed by attribute name
sessionScope	A Map of the session attributes for this request, keyed by attribute name
view	The root UIComponent in the current component tree stored in the FacesRequest for this request

Binding a Component Instance to a Bean Property

A component instance can be bound to a bean property using a value expression with the binding attribute of the component's tag. You usually bind a component instance rather than its value to a bean property if the bean must dynamically change the component's attributes.

Here are two tags from the bookcashier.xhtml page that bind components to bean properties:

```
<h:selectBooleanCheckbox id="fanClub"
                         rendered="false"
                         binding="#{cashierBean.specialOffer}" />
<h:outputLabel for="fanClub"
               rendered="false"
               binding="#{cashierBean.specialOfferText}"
               value="#{bundle.DukeFanClub}"/>
</h:outputLabel>
```

The h:selectBooleanCheckbox tag renders a check box and binds the fanClub UISelectBoolean component to the specialOffer property of the cashier bean. The h:outputLabel tag binds the component representing the check box's label to the specialOfferText property of the cashier bean. If the application's locale is English, the h:outputLabel tag renders

```
I'd like to join the Duke Fan Club, free with my purchase of over $100
```

The `rendered` attributes of both tags are set to `false` to prevent the check box and its label from being rendered. If the customer makes a large order and clicks the Submit button, the `submit` method of `CashierBean` sets both components' `rendered` properties to `true`, causing the check box and its label to be rendered.

These tags use component bindings rather than value bindings because the managed bean must dynamically set the values of the components' `rendered` properties.

If the tags were to use value bindings instead of component bindings, the managed bean would not have direct access to the components and would therefore require additional code to access the components from the `FacesContext` instance to change the components' `rendered` properties.

Writing Properties Bound to Component Instances explains how to write the bean properties bound to the example components.

Binding Converters, Listeners, and Validators to Managed Bean Properties

As described in Adding Components to a Page Using HTML Tag Library Tags, a page author can bind converter, listener, and validator implementations to managed bean properties using the `binding` attributes of the tags that are used to register the implementations on components.

This technique has similar advantages to binding component instances to managed bean properties, as described in Binding Component Values and Instances to Managed Bean Properties. In particular, binding a converter, listener, or validator implementation to a managed bean property yields the following benefits.

- The managed bean can instantiate the implementation instead of allowing the page author to do so.
- The managed bean can programmatically modify the attributes of the implementation. In the case of a custom implementation, the only other way to modify the attributes outside of the implementation class would be to create a custom tag for it and require the page author to set the attribute values from the page.

Whether you are binding a converter, listener, or validator to a managed bean property, the process is the same for any of the implementations.

- Nest the converter, listener, or validator tag within an appropriate component tag.
- Make sure that the managed bean has a property that accepts and returns the converter, listener, or validator implementation class that you want to bind to the property.
- Reference the managed bean property using a value expression from the `binding` attribute of the converter, listener, or validator tag.

For example, say that you want to bind the standard `DateTime` converter to a managed bean property because you want to set the formatting pattern of the user's input in the managed bean rather than on the Facelets page. First, the page registers the converter onto the component by nesting the `f:convertDateTime` tag within the component tag. Then, the page references the property

with the `binding` attribute of the `f:convertDateTime` tag:

```
<h:inputText value="#{loginBean.birthDate}">
    <f:convertDateTime binding="#{loginBean.convertDate}" />
</h:inputText>
```

The `convertDate` property would look something like this:

```
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ...
    return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

See Writing Properties Bound to Converters, Listeners, or Validators for more information on writing managed bean properties for converter, listener, and validator implementations.

Configuring Jakarta Faces Applications



This section was last updated for Jakarta EE Version 9.1

This chapter describes additional configuration tasks required when you create large and complex applications.

Introduction to Configuring Jakarta Faces Applications

The process of building and deploying simple Jakarta Faces applications is described in earlier chapters of this tutorial, including [web:webapp::webapp:::_getting_started_with_web_applications], [web:faces-facelets::faces-facelets:::_introduction_to_facelets], [web:faces-ajax::faces-ajax:::_using_ajax_with_jakarta_faces_technology] and [web:faces-advanced-cc::faces-advanced-cc:::_composite_components_advanced_topics_and_an_example] When you create large and complex applications, however, various additional configuration tasks are required. These tasks include the following:

- Registering managed beans with the application so that all parts of the application have access to them
- Configuring managed beans and model beans so that they are instantiated with the proper values when a page makes reference to them
- Defining navigation rules for each of the pages in the application so that the application has a smooth page flow, if nondefault navigation is needed
- Packaging the application to include all the pages, resources, and other files so that the

application can be deployed on any compliant container

Using Annotations to Configure Managed Beans



In Jakarta Faces 2.3, managed bean annotations are deprecated; CDI is now the preferred approach.

Jakarta Faces support for bean annotations is introduced in [web:faces-intro::faces-intro:::_jakarta_faces_technology]. Bean annotations can be used for configuring Jakarta Faces applications.

The `@Named` (`jakarta.inject.Named`) annotation in a class, along with a scope annotation, automatically registers that class as a resource with the Jakarta Faces implementation. A bean that uses these annotations is a CDI managed bean.

The following shows the use of the `@Named` and `@SessionScoped` annotations in a class:

```
@Named("cart")
@SessionScoped
public class ShoppingCart ... { ... }
```

The above code snippet shows a bean that is managed by the Jakarta Faces implementation and is available for the length of the session.

You can annotate beans with any of the scopes listed in the next section, Using Managed Bean Scopes.

All classes will be scanned for annotations at startup unless the `faces-config` element in the `faces-config.xml` file has the `metadata-complete` attribute set to `true`.

Annotations are also available for other artifacts, such as components, converters, validators, and renderers, to be used in place of application configuration resource file entries. These are discussed, along with registration of custom listeners, custom validators, and custom converters, in [web:faces-custom::faces-custom:::_creating_custom_ui_components_and_other_custom_objects].

Using Managed Bean Scopes

You can use annotations to define the scope in which the bean will be stored. You can specify one of the following scopes for a bean class.

- Application (`jakarta.enterprise.context.ApplicationScoped`): Application scope persists across all users' interactions with a web application.
- Session (`jakarta.enterprise.context.SessionScoped`): Session scope persists across multiple HTTP requests during a single HTTP session in a web application.
- Flow (`jakarta.faces.flow.FlowScoped`): Flow scope persists during a user's interaction with a specific flow during a single HTTP session in a web application. See Using Faces Flows for more information.
- View (`jakarta.faces.view.ViewScoped`): View scope persists across multiple HTTP postbacks on a

single view during a single HTTP session in a web application.

- Request (`jakarta.enterprise.context.RequestScoped`): Request scope persists during a single HTTP request in a web application.
- Dependent (`jakarta.enterprise.context.Dependent`): Indicates that the bean scope depends on the other bean where it's injected.

You may want to use `@Dependent` when a managed bean references another managed bean. The second bean should not be in a scope (`@Dependent`) if it is supposed to be created only when it is referenced. If you define a bean as `@Dependent`, the bean is instantiated anew each time it is referenced, so it does not get saved in any scope.

If your managed bean is referenced by the `binding` attribute of a component tag, you should define the bean with a request scope. If you placed the bean in session or application scope instead, the bean would need to take precautions to ensure thread safety, because `jakarta.faces.component.UIComponent` instances each depend on running inside of a single thread.

If you are configuring a bean that allows attributes to be associated with the view, you can use the view scope. The attributes persist until the user has navigated to the next view.

Application Configuration Resource File

Jakarta Faces technology provides a portable configuration format (as an XML document) for configuring application resources. One or more XML documents, called application configuration resource files, may use this format to register and configure objects and resources and to define navigation rules for applications. An application configuration resource file is usually named `faces-config.xml`.

You need an application configuration resource file in the following cases:

- To specify configuration elements for your application that are not available through managed bean annotations, such as localized messages and navigation rules
- To override managed bean annotations when the application is deployed

The application configuration resource file must be valid against the XML schema located at `https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd`.

In addition, each file must include the following information, in the following order:

- The XML version number, usually with an `encoding` attribute:

```
<?xml version="1.0" encoding='UTF-8'?>
```

- A `faces-config` tag enclosing all the other declarations:

```
<faces-config version="3.0" xmlns="https://jakarta.ee/xml/ns/jakartaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
              https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd">
```

```
    ...
</faces-config>
```

You can have more than one application configuration resource file for an application. The Jakarta Faces implementation finds the configuration file or files by looking for the following.

- A resource named `/META-INF/faces-config.xml` in any of the JAR files in the web application's `/WEB-INF/lib/` directory and in parent class loaders. If a resource with this name exists, it is loaded as a configuration resource. This method is practical for a packaged library containing some components and renderers. In addition, any file with a name that ends in `faces-config.xml` is also considered a configuration resource and is loaded as such.
- A context initialization parameter, `jakarta.faces.application.CONFIG_FILES`, in your web deployment descriptor file that specifies one or more (comma-delimited) paths to multiple configuration files for your web application. This method is most often used for enterprise-scale applications that delegate to separate groups the responsibility for maintaining the file for each portion of a big application.
- A resource named `faces-config.xml` in the `/WEB-INF/` directory of your application. Simple web applications make their configuration files available in this way.

To access the resources registered with the application, an application developer can use an instance of the `jakarta.faces.application.Application` class, which is automatically created for each application. The `Application` instance acts as a centralized factory for resources that are defined in the XML file.

When an application starts up, the Jakarta Faces implementation creates a single instance of the `Application` class and configures it with the information you provided in the application configuration resource file.

Ordering of Application Configuration Resource Files

Because Jakarta Faces technology allows the use of multiple application configuration resource files stored in different locations, the order in which they are loaded by the implementation becomes important in certain situations (for example, when using application-level objects). This order can be defined through an `ordering` element and its subelements in the application configuration resource file itself. The ordering of application configuration resource files can be absolute or relative.

Absolute ordering is defined by an `absolute-ordering` element in the file. With absolute ordering, the user specifies the order in which application configuration resource files will be loaded. The following example shows an entry for absolute ordering.

File `my-faces-config.xml` contains the following elements:

```
<faces-config>
    <name>myFaces</name>
    <absolute-ordering>
        <name>A</name>
        <name>B</name>
```

```
        <name>C</name>
    </absolute-ordering>
</faces-config>
```

In this example, A, B, and C are different application configuration resource files and are to be loaded in the listed order.

If there is an `absolute-ordering` element in the file, only the files listed by the subelement `name` are processed. To process any other application configuration resource files, an `others` subelement is required. In the absence of the `others` subelement, all other unlisted files will be ignored at load time.

Relative ordering is defined by an `ordering` element and its subelements `before` and `after`. With relative ordering, the order in which application configuration resource files will be loaded is calculated by considering ordering entries from the different files. The following example shows some of these considerations. In the following example, `config-A`, `config-B`, and `config-C` are different application configuration resource files.

File `config-A` contains the following elements:

```
<faces-config>
    <name>config-A</name>
    <ordering>
        <before>
            <name>config-B</name>
        </before>
    </ordering>
</faces-config>
```

File `config-B` (not shown here) does not contain any `ordering` elements.

File `config-C` contains the following elements:

```
<faces-config>
    <name>config-C</name>
    <ordering>
        <after>
            <name>config-B</name>
        </after>
    </ordering>
</faces-config>
```

Based on the `before` subelement entry, file `config-A` will be loaded before the `config-B` file. Based on the `after` subelement entry, file `config-C` will be loaded after the `config-B` file.

In addition, a subelement `others` can also be nested within the `before` and `after` subelements. If the `others` element is present, the specified file may receive highest or lowest preference among both listed and unlisted configuration files.

If an `ordering` element is not present in an application configuration file, then that file will be loaded after all the files that contain `ordering` elements.

Using Faces Flows

The Faces Flows feature of Jakarta Faces technology allows you to create a set of pages with a scope, `FlowScoped`, that is greater than request scope but less than session scope. For example, you might want to create a series of pages for the checkout process in an online store. You could create a set of self-contained pages that could be transferred from one store to another as needed.

Faces Flows are somewhat analogous to subroutines in procedural programming, in the following ways.

- Like a subroutine, a flow has a well defined entry point, list of parameters, and return value. However, unlike a subroutine, a flow can return multiple values.
- Like a subroutine, a flow has a scope, allowing information to be available only during the invocation of the flow. Such information is not available outside the scope of the flow and does not consume any resources once the flow returns.
- Like a subroutine, a flow may call other flows before returning. The invocation of flows is maintained in a call stack: a new flow causes a push onto the stack, and a return causes a pop.

An application can have any number of flows. Each flow includes a set of pages and, usually, one or more managed beans scoped to that flow. Each flow has a starting point, called a start node, and an exit point, called a return node.

The data in a flow is scoped to that flow alone, but you can pass data from one flow to another by specifying parameters and calling the other flow.

Flows can be nested, so that if you call one flow from another and then exit the second flow, you return to the calling flow rather than to the second flow's return node.

You can configure a flow programmatically, by creating a class annotated `@FlowDefinition`, or you can configure a flow by using a configuration file. The configuration file can be limited to one flow, or you can use the `faces-config.xml` file to put all the flows in one place, if you have many flows in an application. The programmatic configuration places the code closer to the rest of the flow code and enables you to modularize the flows.

Figure 17, "Two Faces Flows and Their Interactions" shows two flows and illustrates how they interact.

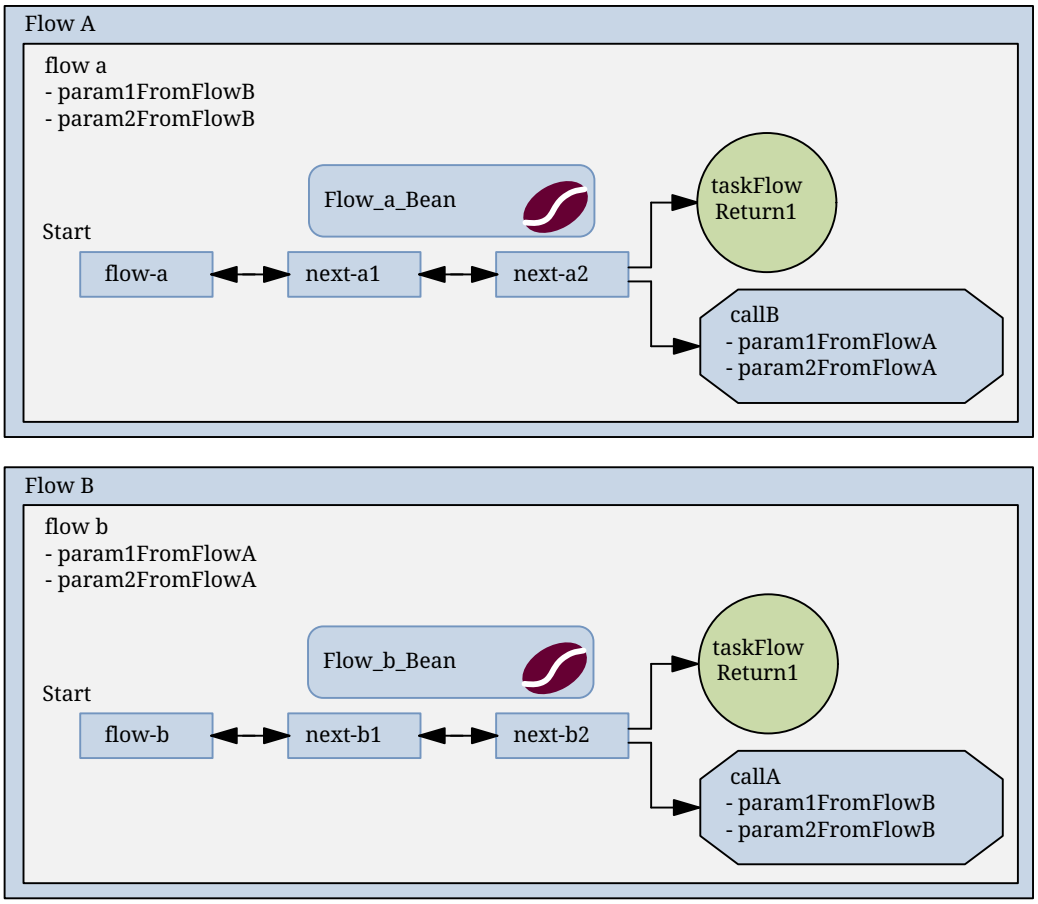


Figure 17. Two Faces Flows and Their Interactions

In this figure, Flow A has a start node named `flow-a` and two additional pages, `next_a1` and `next_a2`. From `next_a2`, a user can either exit the flow using the defined return node, `taskFlowReturn1`, or call Flow B, passing two parameters. Flow A also defines two inbound parameters that it can accept from Flow B. Flow B is identical to Flow A except for the names of the flow and files. Each flow also has an associated managed bean; the beans are `Flow_a_Bean` and `Flow_b_Bean`.

Packaging Flows in an Application

Typically, you package flows in a web application using a directory structure that modularizes the flows. In the `src/main/webapp` directory of a Maven project, for example, you would place the Facelets files that are outside the flow at the top level as usual. Then the `webapp` files for each flow would be in a separate directory, and the Java files would be under `src/main/java`. For example, the files for the application shown in Figure 17, “Two Faces Flows and Their Interactions” might look like this:

```
src/main/webapp/
    index.xhtml
    return.xhtml
    WEB_INF/
        beans.xml
        web.xml
    flow-a/
        flow-a.xhtml
        next_a1.xhtml
        next_a2.xhtml
```

```
    flow-b/
        flow-b-flow.xml
        next_b1.xhtml
        next_b2.xhtml
src/main/java/jakarta/tutorial/flowexample
            FlowA.java
            Flow_a_Bean.java
            Flow_b_Bean.java
```

In this example, `flow-a` is defined programmatically in `FlowA.java`, while `flow-b` is defined by the configuration file `flow-b-flow.xml`.

The Simplest Possible Flow: The simple-flow Example Application

The `simple-flow` example application demonstrates the most basic building blocks of a Faces Flows application and illustrates some of the conventions that make it easy to get started with iterative development using flows. You may want to start with a simple example like this one and build upon it.

This example provides an implicit flow definition by including an empty configuration file. A configuration file that has content, or a class annotated `@FlowDefinition`, provides an explicit flow definition.

The source code for this application is in the `jakartaee-examples/tutorial/web/faces/simple-flow/` directory.

The file layout of the `simple-flow` example looks like this:

```
src/main/webapp
    index.xhtml
    simple-flow-return.xhtml
    WEB_INF/
        web.xml
    simple-flow
        simple-flow-flow.xml
        simple-flow.xhtml
        simple-flow-page2.xhtml
```

The `simple-flow` example has an empty configuration file, which is by convention named flow-name-`flow.xml`. The flow does not require any configuration for the following reasons.

- The flow does not call another flow, nor does it pass parameters to another flow.
- The flow uses default names for the first page of the flow, `flow-name.xhtml`, and the return page, `flow-name-return.xhtml`.

This example has only four Facelets pages.

- `index.xhtml`, the start page, which contains almost nothing but a button that navigates to the first page of the flow:

```
<p><h:commandButton value="Enter Flow" action="simple-flow"/></p>
```

- `simple-flow.xhtml` and `simple-flow-page2.xhtml`, the two pages of the flow itself. In the absence of an explicit flow definition, the page whose name is the same as the name of the flow is assumed to be the start node of the flow. In this case, the flow is named `simple-flow`, so the page named `simple-flow.xhtml` is assumed to be the start node of the flow. The start node is the node navigated to upon entry into the flow. It can be thought of as the home page of the flow.

  The `simple-flow.xhtml` page asks you to enter a flow-scoped value and provides a button that navigates to the next page of the flow:

  ```
  <p>Value: <h:inputText id="input" value="#{flowScope.value}" /></p>

  <p><h:commandButton value="Next" action="simple-flow-page2" /></p>
  ```

  The second page, which can have any name, displays the flow-scoped value and provides a button that navigates to the return page:

  ```
  <p>Value: #{flowScope.value}</p>

  <p><h:commandButton value="Return" action="simple-flow-return" /></p>
  ```

- `simple-flow-return.xhtml`, the return page. The return page, which by convention is named `flow-name-return.xhtml`, must be located outside of the flow. This page displays the flow-scoped value, to show that it has no value outside of the flow, and provides a link that navigates to the `index.xhtml` page:

  ```
  <p>Value (should be empty):
      "<h:outputText id="output" value="#{flowScope.value}" />"</p>

  <p><h:link outcome="index" value="Back to Start" /></p>
  ```

The Facelets pages use only flow-scoped data, so the example does not need a managed bean.

To Build, Package, and Deploy the simple-flow Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `simple-flow` folder.

5. Click **Open Project**.
6. In the **Projects** tab, right-click the `simple-flow` project and select **Build**.

   This command builds and packages the application into a WAR file, `simple-flow.war`, that is located in the `target` directory. It then deploys the application to the server.

To Build, Package, and Deploy the simple-flow Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/simple-flow/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `simple-flow.war`, that is located in the `target` directory. It then deploys the application to the server.

To Run the simple-flow Example

1. Enter the following URL in your web browser:

   ```
   http://localhost:8080/simple-flow
   ```

2. On the `index.xhtml` page, click Enter Flow.
3. On the first page of the flow, enter any string in the Value field, then click Next.
4. On the second page of the flow, you can see the value you entered. Click Return.
5. On the return page, an empty pair of quotation marks encloses the inaccessible value. Click Back to Start to return to the `index.xhtml` page.

The checkout-module Example Application

The `checkout-module` example application is considerably more complex than `simple-flow`. It shows how you might use the Faces Flows feature to implement a checkout module for an online store.

Like the hypothetical example in Figure 17, “Two Faces Flows and Their Interactions”, the example application contains two flows, each of which can call the other. Both flows have explicit flow definitions. One flow, `checkoutFlow`, is specified programmatically. The other flow, `joinFlow`, is specified in a configuration file.

The source code for this application is in the `jakartaee-examples/tutorial/web/faces/checkout-module/` directory.

For the `checkout-module` application, the directory structure is as follows (there is also a `src/main/webapp/resources` directory with a stylesheet and an image):

```
src/main/webapp/
    index.xhtml
    exithome.xhtml
    WEB_INF/
        beans.xml
        web.xml
    checkoutFlow/
        checkoutFlow.xhtml
        checkoutFlow2.xhtml
        checkoutFlow3.xhtml
        checkoutFlow4.xhtml
    joinFlow/
        joinFlow-flow.xml
        joinFlow.xhtml
        joinFlow2.xhtml
src/main/java/jakarta/tutorial/checkoutmodule
    CheckoutBean.java
    CheckoutFlow.java
    CheckoutFlowBean.java
    JoinFlowBean.java
```

For the example, `index.xhtml` is the beginning page for the application as well as the return node for the checkout flow. The `exithome.xhtml` page is the return node for the join flow.

The configuration file `joinFlow-flow.xml` defines the join flow, and the source file `CheckoutFlow.java` defines the checkout flow.

The checkout flow contains four Facelets pages, whereas the join flow contains two.

The managed beans scoped to each flow are `CheckoutFlowBean.java` and `JoinFlowBean.java`, whereas `CheckoutBean.java` is the backing bean for the `index.html` page.

The Facelets Pages for the checkout-module Example

The starting page for the example, `index.xhtml`, summarizes the contents of a hypothetical shopping cart. It allows the user to click either of two buttons to enter one of the two flows:

```
<p><h:commandButton value="Check Out" action="checkoutFlow"/></p>
...
<p><h:commandButton value="Join" action="joinFlow"/></p>
```

This page is also the return node for the checkout flow.

The Facelets page `exithome.xhtml` is the return node for the join flow. This page has a button that allows you to return to the `index.xhtml` page.

The four Facelets pages within the checkout flow, starting with `checkoutFlow.xhtml` and ending with `checkoutFlow4.xhtml`, allow you to proceed to the next page or, in some cases, to return from the flow. The `checkoutFlow.xhtml` page allows you to access parameters passed from the join flow through the flow scope. These appear as empty quotation marks if you have not called the checkout flow from the join flow.

```
<p>If you called this flow from the Join flow, you can see these parameters:
    "<h:outputText value="#{flowScope.param1Value}"/>" and
    "<h:outputText value="#{flowScope.param2Value}"/>"
</p>
```

Only `checkoutFlow2.xhtml` has a button to return to the previous page, but moving between pages is generally permitted within flows. Here are the buttons for `checkoutFlow2.xhtml`:

```
<p><h:commandButton value="Continue" action="checkoutFlow3"/></p>
<p><h:commandButton value="Go Back" action="checkoutFlow"/></p>
<p><h:commandButton value="Exit Flow" action="returnFromCheckoutFlow"/></p>
```

The action `returnFromCheckoutFlow` is defined in the configuration source code file, `CheckoutFlow.java`.

The final page of the checkout flow, `checkoutFlow4.xhtml`, contains a button that calls the join flow:

```
<p><h:commandButton value="Join" action="calljoin"/></p>
<p><h:commandButton value="Exit Flow" action="returnFromCheckoutFlow"/></p>
```

The `calljoin` action is also defined in the configuration source code file, `CheckoutFlow.java`. This action enters the join flow, passing two parameters from the checkout flow.

The two pages in the join flow, `joinFlow.xhtml` and `joinFlow2.xhtml`, are similar to those in the checkout flow. The second page has a button to call the checkout flow as well as one to return from the join flow:

```
<p><h:commandButton value="Check Out" action="callcheckoutFlow"/></p>
<p><h:commandButton value="Exit Flow" action="returnFromJoinFlow"/></p>
```

For this flow, the actions `callcheckoutFlow` and `returnFromJoinFlow` are defined in the configuration file `joinFlow-flow.xml`.

Using a Configuration File to Configure a Flow

If you use an application configuration resource file to configure a flow, it must be named `flowName-flow.xml`. In this example, the join flow uses a configuration file named `joinFlow-flow.xml`. The file is a `faces-config` file that specifies a `flow-definition` element. This element must define the flow name using the `id` attribute. Under the `flow-definition` element, there must be a `flow-return` element that specifies the return point for the flow. Any inbound parameters are specified with

`inbound-parameter` elements. If the flow calls another flow, the `call-flow` element must use the flow-reference element to name the called flow and may use the `outbound-parameter` element to specify any outbound parameters.

The configuration file for the join flow looks like this:

```
<faces-config version="3.0" xmlns="https://jakarta.ee/xml/ns/jakartaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
              https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd">

    <flow-definition id="joinFlow">
        <flow-return id="returnFromJoinFlow">
            <from-outcome>#{joinFlowBean.returnValue}</from-outcome>
        </flow-return>

        <flow-call id="callcheckoutFlow">
            <flow-reference>
                <flow-id>checkoutFlow</flow-id>
            </flow-reference>
            <outbound-parameter>
                <name>param1FromJoinFlow</name>
                <value>#{"param1 joinFlow value"}</value>
            </outbound-parameter>
            <outbound-parameter>
                <name>param2FromJoinFlow</name>
                <value>#{"param2 joinFlow value"}</value>
            </outbound-parameter>
        </flow-call>
        <inbound-parameter>
            <name>param1FromCheckoutFlow</name>
            <value>#{flowScope.param1Value}</value>
        </inbound-parameter>
        <inbound-parameter>
            <name>param2FromCheckoutFlow</name>
            <value>#{flowScope.param2Value}</value>
        </inbound-parameter>
    </flow-definition>
</faces-config>
```

The `id` attribute of the `flow-definition` element defines the name of the flow as `joinFlow`. The value of the `id` attribute of the `flow-return` element identifies the name of the return node, and its value is defined in the `from-outcome` element as the `returnValue` property of the flow-scoped managed bean for the join flow, `JoinFlowBean`.

The names and values of the inbound parameters are retrieved from the flow scope in order (`flowScope.param1Value`, `flowScope.param2Value`), based on the way they were defined in the checkout flow configuration.

The `flow-call` element defines how the join flow calls the checkout flow. The `id` attribute of the

element, `callcheckoutFlow`, defines the action of calling the flow. Within the `flow-call` element, the `flow-reference` element defines the actual name of the flow to call, `checkoutFlow`. The `outbound-parameter` elements define the parameters to be passed when `checkoutFlow` is called. Here they are just arbitrary strings.

Using a Java Class to Configure a Flow

If you use a Java class to configure a flow, it must have the name of the flow. The class for the checkout flow is called `CheckoutFlow.java`.

```
import java.io.Serializable;
import jakarta.enterprise.inject.Produces;
import jakarta.faces.flow.Flow;
import jakarta.faces.flow.builder.FlowBuilder;
import jakarta.faces.flow.builder.FlowBuilderParameter;
import jakarta.faces.flow.builder.FlowDefinition;

class CheckoutFlow implements Serializable {

    private static final long serialVersionUID = 1L;

    @Produces
    @FlowDefinition
    public Flow defineFlow(@FlowBuilderParameter FlowBuilder flowBuilder) {

        String flowId = "checkoutFlow";
        flowBuilder.id("", flowId);
        flowBuilder.viewNode(flowId,
                "/" + flowId + "/" + flowId + ".xhtml").
                markAsStartNode();

        flowBuilder.returnNode("returnFromCheckoutFlow").
                fromOutcome("#{checkoutFlowBean.returnValue}");

        flowBuilder.inboundParameter("param1FromJoinFlow",
                "#{flowScope.param1Value}");
        flowBuilder.inboundParameter("param2FromJoinFlow",
                "#{flowScope.param2Value}");

        flowBuilder.flowCallNode("calljoin").flowReference("", "joinFlow").
                outboundParameter("param1FromCheckoutFlow",
                    "#{checkoutFlowBean.name}").
                outboundParameter("param2FromCheckoutFlow",
                    "#{checkoutFlowBean.city}");
        return flowBuilder.getFlow();
    }
}
```

The class performs actions that are almost identical to those performed by the configuration file `joinFlow-flow.xml`. It contains a single method, `defineFlow`, as a producer method with the

`@FlowDefinition` qualifier that returns a `jakarta.faces.flow.Flow` class. The `defineFlow` method takes one parameter, a `FlowBuilder` with the qualifier `@FlowBuilderParameter`, which is passed in from the Jakarta Faces implementation. The method then calls methods from the `jakarta.faces.flow.Builder.FlowBuilder` class to configure the flow.

First, the method defines the flow `id` as `checkoutFlow`. Then, it explicitly defines the start node for the flow. By default, this is the name of the flow with an `.xhtml` suffix.

The method then defines the return node similarly to the way the configuration file does. The `returnNode` method sets the name of the return node as `returnFromCheckoutFlow`, and the chained `fromOutcome` method specifies its value as the `returnValue` property of the flow-scoped managed bean for the checkout flow, `CheckoutFlowBean`.

The `inboundParameter` method sets the names and values of the inbound parameters from the join flow, which are retrieved from the flow scope in order (`flowScope.param1Value`, `flowScope.param2Value`), based on the way they were defined in the join flow configuration.

The `flowCallNode` method defines how the checkout flow calls the join flow. The argument, `calljoin`, specifies the action of calling the flow. The chained `flowReference` method defines the actual name of the flow to call, `joinFlow`, then calls `outboundParameter` methods to define the parameters to be passed when `joinFlow` is called. Here they are values from the `CheckoutFlowBean` managed bean.

Finally, the `defineFlow` method calls the `getFlow` method and returns the result.

The Flow-Scoped Managed Beans

Each of the two flows has a managed bean that defines properties for the pages within the flow. For example, the `CheckoutFlowBean` defines properties whose values are entered by the user on both the `checkoutFlow.xhtml` page and the `checkoutFlow3.xhtml` page.

Each managed bean has a `getReturnValue` method that sets the value of the return node. For the `CheckoutFlowBean`, the return node is the `index.xhtml` page, specified using implicit navigation:

```
public String getReturnValue() {
    return "index";
}
```

For the `JoinFlowBean`, the return node is the `exithome.xhtml` page.

To Build, Package, and Deploy the checkout-module Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/faces
   ```

4. Select the `checkout-module` folder.

5. Click **Open Project**.
6. In the **Projects** tab, right-click the `checkout-module` project and select **Build**.

   This command builds and packages the application into a WAR file, `checkout-module.war`, that is located in the `target` directory. It then deploys the application to the server.

To Build, Package, and Deploy the checkout-module Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/faces/checkout-module/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command builds and packages the application into a WAR file, `checkout-module.war`, that is located in the `target` directory. It then deploys the application to the server.

To Run the checkout-module Example

1. Enter the following URL in your web browser:

   ```
   http://localhost:8080/checkout-module
   ```

2. The `index.xhtml` page presents hypothetical results of the shopping expedition. Click either Check Out or Join to enter one of the two flows.
3. Follow the flow, providing input as needed and choosing whether to continue, go back, or exit the flow.

   In the checkout flow, only one of the input fields is validated (the credit card field expects 16 digits), so you can enter any values you like. The join flow does not require you to check any boxes in its checkbox menus.

4. On the last page of a flow, select the option to enter the other flow. This allows you to view the inbound parameters from the previous flow.
5. Because flows are nested, if you click Exit Flow from a called flow, you will return to the first page of the calling flow (You may see a warning, which you can ignore). Click Exit Flow on that page to go to the specified return node.

Configuring Managed Beans

When a page references a managed bean for the first time, the Jakarta Faces implementation initializes it either based on a `@Named` annotation and scope annotation in the bean class or according to its configuration in the application configuration resource file. For information on

using annotations to initialize beans, see Using Annotations to Configure Managed Beans.

You can use either annotations or the application configuration resource file to instantiate managed beans that are used in a Jakarta Faces application and to store them in scope. The managed bean creation facility is configured in the application configuration resource file using `managed-bean` XML elements to define each bean. This file is processed at application startup time. For information on using this facility, see Using the managed-bean Element.

Managed beans created in the application configuration resource file are Jakarta Faces managed beans, not CDI managed beans.

With the managed bean creation facility, you can

- Create beans in one centralized file that is available to the entire application, rather than conditionally instantiate beans throughout the application
- Customize a bean's properties without any additional code
- Customize a bean's property values directly from within the configuration file so that it is initialized with these values when it is created
- Using `value` elements, set a property of one managed bean to be the result of evaluating another value expression

This section shows you how to initialize beans using the managed bean creation facility. See Writing Bean Properties and Writing Managed Bean Methods for information on programming managed beans.

Using the managed-bean Element

A managed bean is initiated in the application configuration resource file using a `managed-bean` element, which represents an instance of a bean class that must exist in the application. At runtime, the Jakarta Faces implementation processes the `managed-bean` element. If a page references the bean and no bean instance exists, the Jakarta Faces implementation instantiates the bean as specified by the element configuration.

Here is an example managed bean configuration from the Duke's Bookstore case study:

```
<managed-bean eager="true">
    <managed-bean-name>Book201</managed-bean-name>
    <managed-bean-class>dukesbookstore.model.ImageArea</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
        <property-name>shape</property-name>
        <value>rect</value>
    </managed-property>
    <managed-property>
        <property-name>alt</property-name>
        <value>Duke</value>
    </managed-property>
    <managed-property>
        <property-name>coords</property-name>
```

```
        <value>67,23,212,268</value>
    </managed-property>
</managed-bean>
```

The `managed-bean-name` element defines the key under which the bean will be stored in a scope. For a component's value to map to this bean, the component tag's `value` attribute must match the `managed-bean-name` up to the first period.

The `managed-bean-class` element defines the fully qualified name of the JavaBeans component class used to instantiate the bean.

The `managed-bean` element can contain zero or more `managed-property` elements, each corresponding to a property defined in the bean class. These elements are used to initialize the values of the bean properties. If you don't want a particular property initialized with a value when the bean is instantiated, do not include a `managed-property` definition for it in your application configuration resource file.

If a `managed-bean` element does not contain other `managed-bean` elements, it can contain one `map-entries` element or `list-entries` element. The `map-entries` element configures a set of beans that are instances of `Map`. The `list-entries` element configures a set of beans that are instances of `List`.

In the following example, the `newsletters` managed bean, representing a `UISelectItems` component, is configured as an `ArrayList` that represents a set of `SelectItem` objects. Each `SelectItem` object is in turn configured as a managed bean with properties:

```
<managed-bean>
    <managed-bean-name>newsletters</managed-bean-name>
    <managed-bean-class>java.util.ArrayList</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <list-entries>
        <value-class>jakarta.faces.model.SelectItem</value-class>
        <value>#{newsletter0}</value>
        <value>#{newsletter1}</value>
        <value>#{newsletter2}</value>
        <value>#{newsletter3}</value>
    </list-entries>
</managed-bean>
<managed-bean>
    <managed-bean-name>newsletter0</managed-bean-name>
    <managed-bean-class>jakarta.faces.model.SelectItem</managed-bean-class>
    <managed-bean-scope>none</managed-bean-scope>
    <managed-property>
        <property-name>label</property-name>
        <value>Duke's Quarterly</value>
    </managed-property>
    <managed-property>
        <property-name>value</property-name>
        <value>200</value>
    </managed-property>
```

```
</managed-bean>
...
```

This approach may be useful for quick-and-dirty creation of selection item lists before a development team has had time to create such lists from the database. Note that each of the individual newsletter beans has a `managed-bean-scope` setting of `none` so that they will not themselves be placed into any scope.

See Initializing Array and List Properties for more information on configuring collections as beans.

To map to a property defined by a `managed-property` element, you must ensure that the part of a component tag's `value` expression after the period matches the `managed-property` element's `property-name` element. The next section, Initializing Properties Using the managed-property Element, explains in more detail how to use the `managed-property` element. See Initializing Managed Bean Properties for an example of initializing a managed bean property.

Initializing Properties Using the managed-property Element

A `managed-property` element must contain a `property-name` element, which must match the name of the corresponding property in the bean. A `managed-property` element must also contain one of a set of elements that defines the value of the property. This value must be of the same type as that defined for the property in the corresponding bean. Which element you use to define the value depends on the type of the property defined in the bean. Subelements of managed-property Elements That Define Property Values lists all the elements that are used to initialize a value.

Subelements of managed-property Elements That Define Property Values

Element	Value It Defines
`list-entries`	Defines the values in a list
`map-entries`	Defines the values of a map
`null-value`	Explicitly sets the property to `null`
`value`	Defines a single value, such as a `String`, `int`, or Jakarta Faces EL expression

Using the managed-bean Element includes an example of initializing an `int` property (a primitive type) using the `value` subelement. You also use the `value` subelement to initialize `String` and other reference types. The rest of this section describes how to use the `value` subelement and other subelements to initialize properties of Java `Enum` types, `Map`, `array`, and `Collection`, as well as initialization parameters.

Referencing a Java Enum Type

A managed bean property can also be a Java `Enum` type (see https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html). In this case, the `value` element of the `managed-property` element must be a `String` that matches one of the `String` constants of the `Enum`. In other words, the `String` must be one of the valid values that can be returned if you were to call `valueOf(Class, String)` on `enum`, where `Class` is the `Enum` class and `String` is the contents of the `value` subelement. For example, suppose the managed bean property is the following:

```
public enum Suit { Hearts, Spades, Diamonds, Clubs }
 ...
public Suit getSuit() { ... return Suit.Hearts; }
```

Assuming you want to configure this property in the application configuration resource file, the corresponding `managed-property` element looks like this:

```
<managed-property>
    <property-name>Suit</property-name>
    <value>Hearts</value>
</managed-property>
```

When the system encounters this property, it iterates over each of the members of the `enum` and calls `toString()` on each member until it finds one that is exactly equal to the value from the `value` element.

Referencing a Context Initialization Parameter

Another powerful feature of the managed bean creation facility is the ability to reference implicit objects from a managed bean property.

Suppose you have a page that accepts data from a customer, including the customer's address. Suppose also that most of your customers live in a particular area code. You can make the area code component render this area code by saving it in an implicit object and referencing it when the page is rendered.

You can save the area code as an initial default value in the context `initParam` implicit object by adding a context parameter to your web application and setting its value in the deployment descriptor. For example, to set a context parameter called `defaultAreaCode` to `650`, add a `context-param` element to the deployment descriptor and give the parameter the name `defaultAreaCode` and the value `650`.

Next, write a `managed-bean` declaration that configures a property that references the parameter:

```
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
        <managed-bean-class>CustomerBean</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>areaCode</property-name>
                <value>#{initParam.defaultAreaCode}</value>
            </managed-property>
            ...
</managed-bean>
```

To access the area code at the time the page is rendered, refer to the property from the `area` component tag's `value` attribute:

```
<h:inputText id=area value="#{customer.areaCode}" />
```

Values are retrieved from other implicit objects in a similar way.

Initializing Map Properties

The `map-entries` element is used to initialize the values of a bean property with a type of `Map` if the `map-entries` element is used within a `managed-property` element. A `map-entries` element contains an optional `key-class` element, an optional `value-class` element, and zero or more `map-entry` elements.

Each of the `map-entry` elements must contain a `key` element and either a `null-value` or `value` element. Here is an example that uses the `map-entries` element:

```
<managed-bean>
    ...
    <managed-property>
        <property-name>prices</property-name>
        <map-entries>
            <map-entry>
                <key>My Early Years: Growing Up on *7</key>
                <value>30.75</value>
            </map-entry>
            <map-entry>
                <key>Web Servers for Fun and Profit</key>
                <value>40.75</value>
            </map-entry>
        </map-entries>
    </managed-property>
</managed-bean>
```

The map created from this `map-entries` tag contains two entries. By default, all the keys and values are converted to `String`. If you want to specify a different type for the keys in the map, embed the `key-class` element just inside the `map-entries` element:

```
<map-entries>
    <key-class>java.math.BigDecimal</key-class>
    ...
</map-entries>
```

This declaration will convert all the keys into `java.math.BigDecimal`. Of course, you must make sure that the keys can be converted to the type you specify. The key from the example in this section cannot be converted to a `BigDecimal`, because it is a `String`.

If you want to specify a different type for all the values in the map, include the `value-class` element after the `key-class` element:

```
<map-entries>
```

```
    <key-class>int</key-class>
    <value-class>java.math.BigDecimal</value-class>
    ...
</map-entries>
```

Note that this tag sets only the type of all the `value` subelements.

Each `map-entry` in the preceding example includes a `value` subelement. The `value` subelement defines a single value, which will be converted to the type specified in the bean.

Instead of using a `map-entries` element, it is also possible to assign the entire map using a `value` element that specifies a map-typed expression.

Initializing Array and List Properties

The `list-entries` element is used to initialize the values of an array or `List` property. Each individual value of the array or `List` is initialized using a `value` or `null-value` element. Here is an example:

```
<managed-bean>
    ...
    <managed-property>
        <property-name>books</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>Web Servers for Fun and Profit</value>
            <value>#{myBooks.bookId[3]}</value>
            <null-value/>
        </list-entries>
    </managed-property>
</managed-bean>
```

This example initializes an array or a `List`. The type of the corresponding property in the bean determines which data structure is created. The `list-entries` element defines the list of values in the array or `List`. The `value` element specifies a single value in the array or `List` and can reference a property in another bean. The `null-value` element will cause the `setBooks` method to be called with an argument of `null`. A `null` property cannot be specified for a property whose data type is a Java primitive, such as `int` or `boolean`.

Initializing Managed Bean Properties

Sometimes you might want to create a bean that also references other managed beans so that you can construct a graph or a tree of beans. For example, suppose you want to create a bean representing a customer's information, including the mailing address and street address, each of which is also a bean. The following `managed-bean` declarations create a `CustomerBean` instance that has two `AddressBean` properties: one representing the mailing address and the other representing the street address. This declaration results in a tree of beans with `CustomerBean` as its root and the two `AddressBean` objects as children.

```
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
    <managed-bean-class>
        com.example.mybeans.CustomerBean
    </managed-bean-class>
    <managed-bean-scope> request </managed-bean-scope>
    <managed-property>
        <property-name>mailingAddress</property-name>
        <value>#{addressBean}</value>
    </managed-property>
    <managed-property>
        <property-name>streetAddress</property-name>
        <value>#{addressBean}</value>
    </managed-property>
    <managed-property>
        <property-name>customerType</property-name>
        <value>New</value>
    </managed-property>
</managed-bean>
<managed-bean>
    <managed-bean-name>addressBean</managed-bean-name>
    <managed-bean-class>
        com.example.mybeans.AddressBean
    </managed-bean-class>
    <managed-bean-scope> none </managed-bean-scope>
    <managed-property>
        <property-name>street</property-name>
        <null-value/>
    <managed-property>
    ...
</managed-bean>
```

The first `CustomerBean` declaration (with the `managed-bean-name` of `customer`) creates a `CustomerBean` in request scope. This bean has two properties, `mailingAddress` and `streetAddress`. These properties use the `value` element to reference a bean named `addressBean`.

The second managed bean declaration defines an `AddressBean` but does not create it, because its `managed-bean-scope` element defines a scope of `none`. Recall that a scope of `none` means that the bean is created only when something else references it. Because both the `mailingAddress` and the `streetAddress` properties reference `addressBean` using the `value` element, two instances of `AddressBean` are created when `CustomerBean` is created.

When you create an object that points to other objects, do not try to point to an object with a shorter life span, because it might be impossible to recover that scope's resources when it goes away. A session-scoped object, for example, cannot point to a request-scoped object. And objects with `none` scope have no effective life span managed by the framework, so they can point only to other `none`-scoped objects. Allowable Connections between Scoped Objects outlines all of the allowed connections.

Allowable Connections between Scoped Objects

An Object of This Scope	May Point to an Object of This Scope
`none`	`none`
`application`	`none`, `application`
`session`	`none`, `application`, `session`
`request`	`none`, `application`, `session`, `request`, `view`
`view`	`none`, `application`, `session`, `view`

Be sure not to allow cyclical references between objects. For example, neither of the `AddressBean` objects in the preceding example should point back to the `CustomerBean` object, because `CustomerBean` already points to the `AddressBean` objects.

Initializing Maps and Lists

In addition to configuring `Map` and `List` properties, you can also configure a `Map` and a `List` directly so that you can reference them from a tag rather than referencing a property that wraps a `Map` or a `List`.

Registering Application Messages

Application messages can include any strings displayed to the user as well as custom error messages (which are displayed by the `message` and `messages` tags) for your custom converters or validators. To make messages available at application startup time, do one of the following:

- Queue an individual message onto the `jakarta.faces.context.FacesContext` instance programmatically, as described in Using FacesMessage to Create a Message
- Register all the messages with your application using the application configuration resource file

Here is the section of the `faces-config.xml` file that registers the messages for the Duke's Bookstore case study application:

```
<application>
    <resource-bundle>
        <base-name>
            ee.jakarta.tutorial.dukesbookstore.web.messages.Messages
        </base-name>
        <var>bundle</var>
    </resource-bundle>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>es</supported-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>fr</supported-locale>
    </locale-config>
```

```
</application>
```

This set of elements causes the application to be populated with the messages that are contained in the specified resource bundle.

The `resource-bundle` element represents a set of localized messages. It must contain the fully qualified path to the resource bundle containing the localized messages (in this case, `dukesbookstore.web.messages.Messages`). The `var` element defines the EL name by which page authors refer to the resource bundle.

The `locale-config` element lists the default locale and the other supported locales. The `locale-config` element enables the system to find the correct locale based on the browser's language settings.

The `supported-locale` and `default-locale` tags accept the lowercase, two-character codes defined by ISO 639-1 (see https://www.loc.gov/standards/iso639-2/php/English_list.php). Make sure that your resource bundle actually contains the messages for the locales you specify with these tags.

To access the localized message, the application developer merely references the key of the message from the resource bundle.

You can pull localized text into an `alt` tag for a graphic image, as in the following example:

```
<h:graphicImage id="mapImage"
                name="book_all.jpg"
                library="images"
                alt="#{bundle.ChooseBook}"
                usemap="#bookMap" />
```

The `alt` attribute can accept value expressions. In this case, the `alt` attribute refers to localized text that will be included in the alternative text of the image rendered by this tag.

Using FacesMessage to Create a Message

Instead of registering messages in the application configuration resource file, you can access the `java.util.ResourceBundle` directly from managed bean code. The code snippet below locates an email error message:

```
String message = "";
...
message = ExampleBean.loadErrorMessage(context,
    ExampleBean.EX_RESOURCE_BUNDLE_NAME,
         "EMailError");
context.addMessage(toValidate.getClientId(context),
    new FacesMessage(message));
```

These lines call the bean's `loadErrorMessage` method to get the message from the `ResourceBundle`. Here is the `loadErrorMessage` method:

```
public static String loadErrorMessage(FacesContext context,
     String basename, String key) {
    if ( bundle == null ) {
         try {
            bundle = ResourceBundle.getBundle(basename,
                 context.getViewRoot().getLocale());
        } catch (Exception e) {
            return null;
        }
    }
    return bundle.getString(key);
}
```

Referencing Error Messages

A Jakarta Faces page uses the `message` or `messages` tags to access error messages, as explained in Displaying Error Messages with the h:message and h:messages Tags.

The error messages these tags access include

- The standard error messages that accompany the standard converters and validators that ship with the API (see Section 2.5.2.4 of the Jakarta Faces specification for a complete list of standard error messages).
- Custom error messages contained in resource bundles registered with the application by the application architect using the `resource-bundle` element in the configuration file

When a converter or validator is registered on an input component, the appropriate error message is automatically queued on the component.

A page author can override the error messages queued on a component by using the following attributes of the component's tag:

- `converterMessage`: References the error message to display when the data on the enclosing component cannot be converted by the converter registered on this component.
- `requiredMessage`: References the error message to display when no value has been entered into the enclosing component.
- `validatorMessage`: References the error message to display when the data on the enclosing component cannot be validated by the validator registered on this component.

All three attributes are enabled to take literal values and value expressions. If an attribute uses a value expression, this expression references the error message in a resource bundle. This resource bundle must be made available to the application in one of the following ways:

- By the application architect using the `resource-bundle` element in the configuration file
- By the page author using the `f:loadBundle` tag

Conversely, the `resource-bundle` element must be used to make available to the application those resource bundles containing custom error messages that are queued on the component as a result

of a custom converter or validator being registered on the component.

The following tags show how to specify the `requiredMessage` attribute using a value expression to reference an error message:

```
<h:inputText id="ccno" size="19"
    required="true"
    requiredMessage="#{customMessages.ReqMessage}">
    ...
</h:inputText>
<h:message styleClass="error-message" for="ccno"/>
```

The value expression used by `requiredMessage` in this example references the error message with the `ReqMessage` key in the resource bundle `customMessages`.

This message replaces the corresponding message queued on the component and will display wherever the `message` or `messages` tag is placed on the page.

Using Default Validators

In addition to the validators you declare on the components, you can also specify zero or more default validators in the application configuration resource file. The default validator applies to all `jakarta.faces.component.UIInput` instances in a view or component tree and is appended after the local defined validators. Here is an example of a default validator registered in the application configuration resource file:

```
<faces-config>
    <application>
        <default-validators>
            <validator-id>jakarta.faces.Bean</validator-id>
        </default-validators>
    <application/>
</faces-config>
```

Registering a Custom Validator

If the application developer provides an implementation of the `jakarta.faces.validator.Validator` interface to perform validation, you must register this custom validator either by using the `@FacesValidator` annotation, as described in Implementing the Validator Interface, or by using the `validator` XML element in the application configuration resource file:

```
<validator>
    ...
    <validator-id>FormatValidator</validator-id>
    <validator-class>
        myapplication.validators.FormatValidator
    </validator-class>
    <attribute>
```

```
        ...
        <attribute-name>formatPatterns</attribute-name>
        <attribute-class>java.lang.String</attribute-class>
    </attribute>
</validator>
```

Attributes specified in a `validator` tag override any settings in the `@FacesValidator` annotation.

The `validator-id` and `validator-class` elements are required subelements. The `validator-id` element represents the identifier under which the `Validator` class should be registered. This ID is used by the tag class corresponding to the custom `validator` tag.

The `validator-class` element represents the fully qualified class name of the `Validator` class.

The `attribute` element identifies an attribute associated with the `Validator` implementation. It has required `attribute-name` and `attribute-class` subelements. The `attribute-name` element refers to the name of the attribute as it appears in the `validator` tag. The `attribute-class` element identifies the Java type of the value associated with the attribute.

Creating and Using a Custom Validator explains how to implement the `Validator` interface.

Using a Custom Validator explains how to reference the validator from the page.

Registering a Custom Converter

As is the case with a custom validator, if the application developer creates a custom converter, you must register it with the application either by using the `@FacesConverter` annotation, as described in Creating a Custom Converter, or by using the `converter` XML element in the application configuration resource file. Here is a hypothetical `converter` configuration for `CreditCardConverter` from the Duke's Bookstore case study:

```
<converter>
    <description>
        Converter for credit card numbers that normalizes
        the input to a standard format
    </description>
    <converter-id>CreditCardConverter</converter-id>
    <converter-class>
        dukesbookstore.converters.CreditCardConverter
    </converter-class>
</converter>
```

Attributes specified in a `converter` tag override any settings in the `@FacesConverter` annotation.

The `converter` element represents a `jakarta.faces.convert.Converter` implementation and contains required `converter-id` and `converter-class` elements.

The `converter-id` element identifies an ID that is used by the `converter` attribute of a UI component tag to apply the converter to the component's data. Using a Custom Converter includes an example

of referencing the custom converter from a component tag.

The `converter-class` element identifies the `Converter` implementation.

Creating and Using a Custom Converter explains how to create a custom converter.

Configuring Navigation Rules

Navigation between different pages of a Jakarta Faces application, such as choosing the next page to be displayed after a button or link component is clicked, is defined by a set of rules. Navigation rules can be implicit, or they can be explicitly defined in the application configuration resource file. For more information on implicit navigation rules, see Navigation Model.

Each navigation rule specifies how to navigate from one page to another page or set of pages. The Jakarta Faces implementation chooses the proper navigation rule according to which page is currently displayed.

After the proper navigation rule is selected, the choice of which page to access next from the current page depends on two factors:

- The action method invoked when the component was clicked
- The logical outcome referenced by the component's tag or returned from the action method

The outcome can be anything the developer chooses, but Common Outcome Strings lists some outcomes commonly used in web applications.

Common Outcome Strings

Outcome	What It Means
`success`	Everything worked. Go on to the next page.
`failure`	Something is wrong. Go on to an error page.
`login`	The user needs to log in first. Go on to the login page.
`no results`	The search did not find anything. Go to the search page again.

Usually, the action method performs some processing on the form data of the current page. For example, the method might check whether the user name and password entered in the form match the user name and password on file. If they match, the method returns the outcome `success`. Otherwise, it returns the outcome `failure`. As this example demonstrates, both the method used to process the action and the outcome returned are necessary to determine the correct page to access.

Here is a navigation rule that could be used with the example just described:

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
```

```
        <from-action>#{LoginForm.login}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/storefront.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-action>#{LoginForm.logon}</from-action>
        <from-outcome>failure</from-outcome>
        <to-view-id>/logon.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

This navigation rule defines the possible ways to navigate from `login.xhtml`. Each `navigation-case` element defines one possible navigation path from `login.xhtml`. The first `navigation-case` says that if `LoginForm.login` returns an outcome of `success`, then `storefront.xhtml` will be accessed. The second `navigation-case` says that `login.xhtml` will be re-rendered if `LoginForm.login` returns `failure`.

The configuration of an application's page flow consists of a set of navigation rules. Each rule is defined by the `navigation-rule` element in the `faces-config.xml` file.

Each `navigation-rule` element corresponds to one component tree identifier defined by the optional `from-view-id` element. This means that each rule defines all the possible ways to navigate from one particular page in the application. If there is no `from-view-id` element, the navigation rules defined in the `navigation-rule` element apply to all the pages in the application. The `from-view-id` element also allows wildcard matching patterns. For example, this `from-view-id` element says that the navigation rule applies to all the pages in the `books` directory:

```
<from-view-id>/books/*</from-view-id>
```

A `navigation-rule` element can contain zero or more `navigation-case` elements. The `navigation-case` element defines a set of matching criteria. When these criteria are satisfied, the application will navigate to the page defined by the `to-view-id` element contained in the same `navigation-case` element.

The navigation criteria are defined by optional `from-outcome` and `from-action` elements. The `from-outcome` element defines a logical outcome, such as `success`. The `from-action` element uses a method expression to refer to an action method that returns a `String`, which is the logical outcome. The method performs some logic to determine the outcome and returns the outcome.

The `navigation-case` elements are checked against the outcome and the method expression in the following order.

1. Cases specifying both a `from-outcome` value and a `from-action` value. Both of these elements can be used if the action method returns different outcomes depending on the result of the processing it performs.
2. Cases specifying only a `from-outcome` value. The `from-outcome` element must match either the outcome defined by the `action` attribute of the `jakarta.faces.component.UICommand` component or the outcome returned by the method referred to by the `UICommand` component.

3. Cases specifying only a `from-action` value. This value must match the `action` expression specified by the component tag.

When any of these cases is matched, the component tree defined by the `to-view-id` element will be selected for rendering.

Registering a Custom Renderer with a Render Kit

When the application developer creates a custom renderer, as described in Delegating Rendering to a Renderer, you must register it using the appropriate render kit. Because the image map application implements an HTML image map, the `AreaRenderer` and `MapRenderer` classes in the Duke's Bookstore case study should be registered using the HTML render kit.

You register the renderer either by using the `@FacesRenderer` annotation, as described in Creating the Renderer Class, or by using the `render-kit` element of the application configuration resource file. Here is a hypothetical configuration of `AreaRenderer`:

```
<render-kit>
    <renderer>
        <component-family>Area</component-family>
        <renderer-type>DemoArea</renderer-type>
        <renderer-class>
            dukesbookstore.renderers.AreaRenderer
        </renderer-class>
        <attribute>
            <attribute-name>onmouseout</attribute-name>
            <attribute-class>java.lang.String</attribute-class>
        </attribute>
        <attribute>
            <attribute-name>onmouseover</attribute-name>
            <attribute-class>java.lang.String</attribute-class>
        </attribute>
        <attribute>
            <attribute-name>styleClass</attribute-name>
            <attribute-class>java.lang.String</attribute-class>
        </attribute>
    </renderer>
    ...
```

Attributes specified in a `renderer` tag override any settings in the `@FacesRenderer` annotation.

The `render-kit` element represents a jakarta.faces.render.`RenderKit` implementation. If no `render-kit-id` is specified, the default HTML render kit is assumed. The `renderer` element represents a `jakarta.faces.render.Renderer` implementation. By nesting the `renderer` element inside the `render-kit` element, you are registering the renderer with the `RenderKit` implementation associated with the `render-kit` element.

The `renderer-class` is the fully qualified class name of the `Renderer`.

The `component-family` and `renderer-type` elements are used by a component to find renderers that

can render it. The `component-family` identifier must match that returned by the component class's `getFamily` method. The component family represents a component or set of components that a particular renderer can render. The `renderer-type` must match that returned by the `getRendererType` method of the tag handler class.

By using the component family and renderer type to look up renderers for components, the Jakarta Faces implementation allows a component to be rendered by multiple renderers and allows a renderer to render multiple components.

Each of the `attribute` tags specifies a render-dependent attribute and its type. The `attribute` element doesn't affect the runtime execution of your application. Rather, it provides information to tools about the attributes the `Renderer` supports.

The object responsible for rendering a component (be it the component itself or a renderer to which the component delegates the rendering) can use facets to aid in the rendering process. These facets allow the custom component developer to control some aspects of rendering the component. Consider this custom component tag example:

```
<d:dataScroller>
    <f:facet name="header">
        <h:panelGroup>
            <h:outputText value="Account Id"/>
            <h:outputText value="Customer Name"/>
            <h:outputText value="Total Sales"/>
        </h:panelGroup>
    </f:facet>
    <f:facet name="next">
        <h:panelGroup>
            <h:outputText value="Next"/>
            <h:graphicImage url="/images/arrow-right.gif" />
        </h:panelGroup>
    </f:facet>
    ...
</d:dataScroller>
```

The `dataScroller` component tag includes a component that will render the header and a component that will render the Next button. If the renderer associated with this component renders the facets, you can include the following `facet` elements in the `renderer` element:

```
<facet>
    <description>This facet renders as the header of the table. It should be
         a panelGroup with the same number of columns as the data.
    </description>
    <display-name>header</display-name>
    <facet-name>header</facet-name>
</facet>
<facet>
    <description>This facet renders as the content of the "next" button in
         the scroller. It should be a panelGroup that includes an outputText
```

```
        tag that has the text "Next" and a right arrow icon.
    </description>
    <display-name>Next</display-name>
    <facet-name>next</facet-name>
</facet>
```

If a component that supports facets provides its own rendering and you want to include `facet` elements in the application configuration resource file, you need to put them in the component's configuration rather than the renderer's configuration.

Registering a Custom Component

In addition to registering custom renderers (as explained in the preceding section), you also must register the custom components that are usually associated with the custom renderers. You use either a `@FacesComponent` annotation, as described in Creating Custom Component Classes, or the `component` element of the application configuration resource file.

Here is a hypothetical `component` element from the application configuration resource file that registers `AreaComponent`:

```
<component>
    <component-type>DemoArea</component-type>
    <component-class>
        dukesbookstore.components.AreaComponent
    </component-class>
    <property>
        <property-name>alt</property-name>
        <property-class>java.lang.String</property-class>
    </property>
    <property>
        <property-name>coords</property-name>
        <property-class>java.lang.String</property-class>
    </property>
    <property>
        <property-name>shape</property-name>
        <property-class>java.lang.String</property-class>
    </property>
</component>
```

Attributes specified in a `component` tag override any settings in the `@FacesComponent` annotation.

The `component-type` element indicates the name under which the component should be registered. Other objects referring to this component use this name. For example, the `component-type` element in the configuration for `AreaComponent` defines a value of `DemoArea`, which matches the value returned by the `AreaTag` class's `getComponentType` method.

The `component-class` element indicates the fully qualified class name of the component. The `property` elements specify the component properties and their types.

If the custom component can include facets, you can configure the facets in the component configuration using `facet` elements, which are allowed after the `component-class` elements. See Registering a Custom Renderer with a Render Kit for further details on configuring facets.

Basic Requirements of a Jakarta Faces Application

In addition to configuring your application, you must satisfy other requirements of Jakarta Faces applications, including properly packaging all the necessary files and providing a deployment descriptor. This section describes how to perform these administrative tasks.

Jakarta Faces applications can be packaged in a WAR file, which must conform to specific requirements to execute across different containers. At a minimum, a WAR file for a Jakarta Faces application may contain the following:

- A web application deployment descriptor, called `web.xml`, to configure resources required by a web application (required)
- A specific set of JAR files containing essential classes (optional)
- A set of application classes, Jakarta Faces pages, and other required resources, such as image files

A WAR file may also contain:

- An application configuration resource file, which configures application resources
- A set of tag library descriptor files

For example, a Jakarta Faces web application WAR file using Facelets typically has the following directory structure:

```
$PROJECT_DIR
[Web Pages]
+- /[xhtml or html documents]
+- /resources
+- /WEB-INF
   +- /web.xml
   +- /beans.xml (optional)
   +- /classes (optional)
   +- /lib (optional)
   +- /faces-config.xml (optional)
   +- /*.taglib.xml (optional)
   +- /glassfish-web.xml (optional)
```

The `web.xml` file (or web deployment descriptor), the set of JAR files, and the set of application files must be contained in the `WEB-INF` directory of the WAR file.

Configuring an Application with a Web Deployment Descriptor

Web applications are commonly configured using elements contained in the web application deployment descriptor, `web.xml`. The deployment descriptor for a Jakarta Faces application must

specify certain configurations, including the following:

- The servlet used to process Jakarta Faces requests
- The servlet mapping for the processing servlet
- The path to the configuration resource file, if it exists and is not located in a default location

The deployment descriptor can also include other, optional configurations, such as those that

- Specify where component state is saved
- Encrypt state saved on the client
- Compress state saved on the client
- Restrict access to pages containing Jakarta Faces tags
- Turn on XML validation
- Specify the Project Stage
- Verify custom objects

This section gives more details on these configurations. Where appropriate, it also describes how you can make these configurations using NetBeans IDE.

Identifying the Servlet for Lifecycle Processing

A requirement of a Jakarta Faces application is that all requests to the application that reference previously saved Jakarta Faces components must go through `jakarta.faces.webapp.FacesServlet`. A `FacesServlet` instance manages the request-processing lifecycle for web applications and initializes the resources required by Jakarta Faces technology.

Before a Jakarta Faces application can launch its first web page, the web container must invoke the `FacesServlet` instance in order for the application lifecycle process to start. See The Lifecycle of a Jakarta Faces Application for more information.

The following example shows the default configuration of the `FacesServlet`:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
</servlet>
```

You will provide a mapping configuration entry to make sure that the `FacesServlet` instance is invoked. The mapping to `FacesServlet` can be a prefix mapping, such as `/faces/, or an extension mapping, such as.xhtml`. The mapping is used to identify a page as having Jakarta Faces content. Because of this, the URL to the first page of the application must include the URL pattern mapping.

The following elements specify a prefix mapping:

```
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
```

```
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
...
<welcome-file-list>
    <welcome-file>faces/greeting.xhtml</welcome-file>
</welcome-file-list>
```

The following elements, used in the tutorial examples, specify an extension mapping:

```
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
...
<welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>
```

When you use this mechanism, users access the application as shown in the following example:

```
http://localhost:8080/guessNumber
```

In the case of extension mapping, if a request comes to the server for a page with an `.xhtml` extension, the container will send the request to the `FacesServlet` instance, which will expect a corresponding page of the same name containing the content to exist.

To minimize clutter and allow simple, friendly URLs, you can have extensionless URLs by manually exact mapping the `FacesServlet` to the existing prefix and suffix mapping options in `web.xml`, one or more times.

If you are using NetBeans IDE to create your application, a web deployment descriptor is automatically created for you with default configurations. If you created your application without an IDE, you can create a web deployment descriptor.

To Specify a Path to an Application Configuration Resource File

As explained in Application Configuration Resource File, an application can have multiple application configuration resource files. If these files are not located in the directories that the implementation searches by default or the files are not named `faces-config.xml`, you need to specify paths to these files.

To specify these paths using NetBeans IDE, do the following.

1. Expand the node of your project in the **Projects** tab.
2. Expand the **Web Pages** and **WEB-INF** nodes that are under the project node.
3. Double-click `web.xml`.
4. After the `web.xml` file appears in the editor, click **General** at the top of the editor window.

5. Expand the **Context Parameters** node.
6. Click **Add**.
7. In the Add Context Parameter dialog box:
    a. Enter `jakarta.faces.CONFIG_FILES` in the **Parameter Name** field.
    b. Enter the path to your configuration file in the **Parameter Value** field.
    c. Click **OK**.
    d. Repeat steps 1 through 7 for each configuration file.

To Specify Where State Is Saved

For all the components in a web application, you can specify in your deployment descriptor where you want the state to be saved, on either client or server. You do this by setting a context parameter in your deployment descriptor. By default, state is saved on the server, so you need to specify this context parameter only if you want to save state on the client. See Saving and Restoring State for information on the advantages and disadvantages of each location.

To specify where state is saved using NetBeans IDE, do the following.

1. Expand the node of your project in the **Projects** tab.
2. Expand the **Web Pages** and **WEB-INF** nodes under the project node.
3. Double-click `web.xml`.
4. After the `web.xml` file appears in the editor window, click **General** at the top of the editor window.
5. Expand the **Context Parameters** node.
6. Click **Add**.
7. In the Add Context Parameter dialog box:
    a. Enter `jakarta.faces.STATE_SAVING_METHOD` in the **Parameter Name** field.
    b. Enter `client` or `server` in the **Parameter Value** field.
    c. Click **OK**.

If state is saved on the client, the state of the entire view is rendered to a hidden field on the page. The Jakarta Faces implementation saves the state on the server by default. Duke's Forest saves its state on the client.

Configuring Project Stage

Project Stage is a context parameter identifying the status of a Jakarta Faces application in the software lifecycle. The stage of an application can affect the behavior of the application. For example, error messages can be displayed during the Development stage but suppressed during the Production stage.

The possible Project Stage values are as follows:

- `Development`

- `UnitTest`
- `SystemTest`
- `Production`

Project Stage is configured through a context parameter in the web deployment descriptor file. Here is an example:

```
<context-param>
    <param-name>jakarta.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

If no Project Stage is defined, the default stage is `Production`. You can also add custom stages according to your requirements.

Including the Classes, Pages, and Other Resources

When packaging web applications using the included build scripts, you'll notice that the scripts package resources in the following ways.

- All web pages are placed at the top level of the WAR file.
- The `faces-config.xml` file and the `web.xml` file are packaged in the `WEB-INF` directory.
- All packages are stored in the `WEB-INF/classes/` directory.
- All application JAR files are packaged in the `WEB-INF/lib/` directory.
- All resource files are either under the root of the web application `/resources` directory or in the web application's classpath, the `META-INF/resources/resourceIdentifier` directory. For more information on resources, see Web Resources.

When packaging your own applications, you can use NetBeans IDE or you can use XML files such as those created for Maven. You can modify the XML files to fit your situation. However, you can continue to package your WAR files by using the directory structure described in this section, because this technique complies with the commonly accepted practice for packaging web applications.

Using WebSockets with Jakarta Faces Technology



This section was last updated for Jakarta EE Version 9.1

This chapter describes using WebSockets in Jakarta Faces web applications.

About WebSockets in Jakarta Faces

You use the `f:websocket tag` in a view to allow server-side communications to be pushed to all instances of a socket containing the same channel name. When the communication is received, an `onmessage`, client-side JavaScript event handler can be set that is called whenever a push arrives from the server.

The server side of a WebSocket communication has the ability to push out messages. You can do this using `jakarta.faces.push.PushContext`, which is an injectable context, allowing a server push to a named channel.

Configuring WebSockets

To configure WebSockets for use in faces web applications, first enable the WebSocket endpoint using the context parameter in `web.xml`:

```
<context-param>
    <param-name>jakarta.faces.ENABLE_WEBSOCKET_ENDPOINT</param-name>
    <param-value>true</param-value>
</context-param>
```

If your server is configured to run a WebSocket container on a different TCP port than the HTTP container, you can use the optional `jakarta.faces.WEBSOCKET_ENDPOINT_PORT` integer context parameter to explicitly specify the port:

```
<context-param>
    <param-name>jakarta.faces.WEBSOCKET_ENDPOINT_PORT</param-name>
    <param-value>8000</param-value>
</context-param>
```

WebSocket Usage: Client Side

Declare the `f:websocket tag` in the faces view with a channel name and an `onmessage` JavaScript listener function.

The following example refers to an existing JavaScript listener function:

```
<f:websocket channel="someChannel" onmessage="someWebsocketListener" />

function someWebsocketListener(message, channel, event) { console.log(message); }
```

This example declares an inline JavaScript listener function:

```
<f:websocket channel="someChannel" onmessage="function(m){console.log(m);}" />
```

The `onmessage` JavaScript listener function is invoked with three arguments:

- `message`: The push message as a JSON object
- `channel`: The channel name
- `event`: The `MessageEvent` instance

When successfully connected, the WebSocket is open by default for as long as the document is open,

and it will auto-reconnect, at increasing intervals, when the connection is closed, or aborted, as a result of events such as a network error or server restart. It will not auto-reconnect when the very first connection attempt fails. The WebSocket will be implicitly closed after the document is unloaded.

WebSocket Usage: Server Side

On the Java programming side, inject a `PushContext` using the `@Push` annotation on the given channel in any CDI or container managed artifact, such as `@Named`, or `@WebServlet`, where you want to send a push message. Then invoke `PushContext.send(Object)` with any Java object representing the push message.

For example:

```
@Inject @Push
private PushContext someChannel;

public void sendMessage(Object message) {
    someChannel.send(message);
}
```

By default, the name of the channel is taken from the name of the variable into which the injection takes place.

Optionally, the channel name can be specified using the `channel` attribute. The following example injects the push context for channel name `foo` into a variable named `bar`.

```
@Inject
@Push(channel="foo")
private PushContext bar;
```

The message object will be encoded as JSON and delivered as a message argument of the `onmessage` JavaScript listener function associated with the channel name. It can be a String, but it can also be a collection, map, or a JavaBean.

Using the f:websocket Tag

Attributes of the `f:websocket` Tag describes the attributes of the `f:websocket` tag.

Attributes of the `f:websocket` Tag

Name	Type	Description
`channel`	String	Required. The name of the WebSocket channel. It may not be an EL expression and may only contain alphanumeric characters, hyphens, underscores, and periods. All open WebSockets on the same channel name will receive the same push notification from the server.

Name	Type	Description
`id`	String	Optional. The identifier of the UIWebSocket component to be created.
`scope`	String	Optional. The scope of the WebSocket channel. It may not be an EL expression. Allowed values (case insensitive) are: `application`, `session`, and `view`. When the value is `application`, all channels with the same name throughout the application receive the same push message. When the value is `session`, only the channels with the same name in the current user session receive the same push message. When the value is `view`, only the channel in the current view receives the push message. The default scope is `application`. When the `user` attribute is specified, then the default scope is `session`.
`user`	Serializable	Optional. The user identifier of the WebSocket channel, so that user-targeted push messages can be sent. It must implement Serializable and preferably have a low memory footprint. **Hint**: Use `#{request.remoteUser}` or `#{someLoggedInUser.id}`. All open WebSockets on the same channel and user will receive the same push message from the server.
`onopen`	String	Optional. The JavaScript event handler function that is invoked when the WebSocket is opened. The function is invoked with one argument: the channel name.
`onmessage`	String	Optional. The JavaScript event handler function that is invoked when a push message is received from the server. The function is invoked with three arguments: the push message, the channel name, and the `MessageEvent` instance.

Name	Type	Description
onclose	String	Optional. The JavaScript event handler function that is invoked when the WebSocket is closed. The function is invoked with three arguments: the close reason code, the channel name, and the CloseEvent. Note that this will also be invoked on errors. If an error occurred, you can inspect the close reason code and which code was given (for example, when the code is not 1000).
connected	Boolean	Optional. Specifies whether to auto-reconnect the WebSocket. Defaults to true. It is interpreted as a JavaScript instruction to open or close the WebSocket push connection. This attribute is implicitly re-evaluated on every ajax request by a PreRenderViewEvent listener on the UIViewRoot. You can also explicitly set it to false and then manually control it in JavaScript using faces.push.open(clientId) and faces.push.close(clientId).
rendered	Boolean	Optional. Specifies whether to render the WebSocket scripts. Defaults to true. This attribute is implicitly re-evaluated on every ajax request by a PreRenderViewEvent listener on the UIViewRoot. If the value changes to false while the WebSocket is already opened, then the WebSocket will implicitly be closed.
binding	UIComponent	Optional. The value binding expression to a backing bean property bound to the component instance for the UIComponent created by this tag.

WebSocket Scopes and Users

By default, the WebSocket is application-scoped. For example, any view or session throughout the web application having the same WebSocket channel open will receive the same push message. The push message can be sent by all users and the application. To restrict the push messages to all views in the current user session only, set the optional scope attribute to session . In this case, the push message can only be sent by the user and not by the application.

```
<f:websocket channel="someChannel" scope="session" ... />
```

To restrict the push messages to the current view only, you can set the scope attribute to view. The push message will not show up in other views in the same session, even if it has the same URL. This

push message can be sent only by the user and not by the application.

```
<f:websocket channel="someChannel" scope="view" ... />
```

The scope attribute may not be an EL expression.

Additionally, you can set the optional `user` attribute to the unique identifier of the logged-in user, usually the login name or the user ID.As such, the push message can be targeted to a specific user and can also be sent by other users and the application. The value of the `user` attribute must implement `Serializable` and have a low memory footprint, so an entire user entity is not recommended.

For example, when you are using container managed authentication or a related framework or library:

```
<f:websocket channel="someChannel"user="#{request.remoteUser}" ... />
```

Or, when you have a custom user entity accessible via EL, such as `#{someLoggedInUser}` which has an `id` property representing its identifier:

```
<f:websocket channel="someChannel" user="#{someLoggedInUser.id}" ... />
```

When the `user` attribute is specified, the scope defaults to `session` and cannot be set to `application`.

On the server side, the push message can be targeted to the user specified in the `user` attribute using `PushContext.send(Object, Serializable)`. The push message can be sent by all users and the application.

```
@Inject @Push
private PushContext someChannel;

public void sendMessage(Object message, User recipientUser) {
    Long recipientUserId = recipientUser.getId();
    someChannel.send(message, recipientUserId);
}
```

Multiple users can be targeted by passing a `Collection` holding user identifiers to `PushContext.send(Object, Collection)`.

```
public void sendMessage(Object message, Group recipientGroup) {
    Collection<Long> recipientUserIds = recipientGroup.getUserIds();
    someChannel.send(message, recipientUserIds);
}
```

Conditionally Connecting WebSockets

You can use the optional connected attribute to control whether to auto-reconnect the WebSocket.

```
<f:websocket ... connected="#{bean.pushable}" />
```

The connected attribute defaults to `true` and is interpreted as a JavaScript instruction to open or close the WebSocket push connection. If the value is an EL expression and it becomes `false` during an ajax request, then the push connection will explicitly be closed during `oncomplete` of that ajax request.

You can also explicitly set it to `false` and manually open the push connection on the client side by invoking `faces.push.open(clientId)`, passing the component's client ID.

```
<h:commandButton ... onclick="faces.push.open('foo')">
    <f:ajax ... />
</h:commandButton>
<f:websocket id="foo" channel="bar" scope="view" ... connected="false" />
```

If you intend to have a one-time push and do not expect more messages, you can optionally explicitly close the push connection from the client side by invoking `faces.push.close(clientId)`, passing the component's client ID. For example, in the `onmessage` JavaScript listener function, as seen below:

```
function someWebsocketListener(message) {
// ... faces.push.close('foo');
}
```

WebSocket Events: Server

When a session or view-scoped socket is automatically closed with close reason code `1000` by the server (and thus, not manually closed by the client via `faces.push.close(clientId)`), it means that the session or view has expired.

```
@ApplicationScoped
public class WebsocketObserver {

    public void onOpen(@Observes @Opened WebsocketEvent event) {
        String channel = event.getChannel();
        // Returns <f:websocket channel>. Long userId = event.getUser();
        // Returns <f:websocket user>, if any.
        // ...
    }

    public void onClose(@Observes @Closed WebsocketEvent event) { String channel =
event.getChannel();
        // Returns <f:websocket channel>. Long userId = event.getUser();
```

```
        // Returns <f:websocket user>, if any. CloseCode code = event.getCloseCode();
        // Returns close reason code.
       // ...
    }
}
```

WebSocket Events: Clients

You can use the optional `onopen` JavaScript listener function to listen for the open of a WebSocket on the client side. This function is invoked on the very first connection attempt, regardless of whether it will be successful. It will not be invoked when the WebSocket auto-reconnects a broken connection after the first successful connection.

```
<f:websocket ... onopen="websocketOpenListener" />
function websocketOpenListener(channel) {
// ...
}
```

The `onopen` JavaScript listener function is invoked with one argument: `channel` (the channel name, particularly useful if you have a global listener).

You can use the optional `onclose` JavaScript listener function to listen for a normal or abnormal close of a WebSocket. This function is invoked when the very first connection attempt fails, or the server has returned close reason code `1000` (normal closure) or `1008` (policy violated), or the maximum reconnect attempts have been exceeded. It will not be invoked if the WebSocket makes an auto-reconnect attempt on a broken connection after the first successful connection.

```
<f:websocket ... onclose="websocketCloseListener" />
function websocketCloseListener(code, channel, event) {
    if (code == -1) {
        // Websockets not supported by client.
    } else if (code == 1000) {
        // Normal close (as result of expired session or view).
    } else {
        // Abnormal close reason (as result of an error).
    }
}
```

The `onclose` JavaScript listener function is invoked with three arguments:

- `code`: The close reason code as an integer. If it is `-1`, the WebSocket is not supported by the client. If it is `1000`, then it was normally closed. Otherwise, if it is not `1000`, then there might be an error.
- `channel`: The channel name
- `event`: The `CloseEvent` instance

WebSocket Security Considerations

If the WebSocket is declared in a page which is restricted to logged-in users only with a specific role, then you might want to add the push handshake request URL to the set of restricted URLs.

The push handshake request URL is composed of the URI prefix, `/jakarta.faces.push/`, followed by the channel name. In the example of container managed security, which has already restricted an example page, `/user/foo.xhtml`, to logged-in users with the example role, `USER`, on the example URL pattern, `/user/*`, in `web.xml`, see below:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Restrict access to role USER.</web-resource-name>
        <url-pattern>/user/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>USER</role-name>
    </auth-constraint>
</security-constraint>
```

If the page, `/user/foo.xhtml`, contains `<f:websocket channel="foo">`, then you must add a restriction on the push handshake request URL pattern of `/jakarta.faces.push/foo`, as shown next:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Restrict access to role USER.</web-resource-name>
        <url-pattern>/user/*</url-pattern>
        <url-pattern>/jakarta.faces.push/foo</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>USER</role-name>
    </auth-constraint>
</security-constraint>
```

As extra security, particularly for those public channels which cannot be restricted by security constraints, the `f:websocket` tag will register all previously declared channels in the current HTTP session, and any incoming WebSocket open request will be checked for whether it matches these channels in the current HTTP session. If the channel is unknown (for example, randomly guessed or spoofed by end users or manually reconnected after the session is expired), then the WebSocket will immediately be closed with close reason code, `CloseCodes.VIOLATED_POLICY (1008)`. Also, when the HTTP session gets destroyed, all session and view-scoped channels which are still open will explicitly be closed from the server side with close reason code, `CloseCodes.NORMAL_CLOSURE (1000)`. Only application-scoped sockets remain open and are still reachable from the server even when the session or view associated with the page in the client side is expired.

Using Ajax With WebSockets

If you want to perform complex UI updates depending on the received push message, you can nest

the f:ajax tag inside the f:websocket tag. See the following example:

```
<h:panelGroup id="foo">
    ... (some complex UI here) ...
</h:panelGroup>
<h:form>
    <f:websocket channel="someChannel" scope="view">
        <f:ajax event="someEvent" listener="#{bean.pushed}" render=":foo" />
    </f:websocket>
</h:form>
```

Here, the push message simply represents the ajax event name. You can use any custom event name.

```
someChannel.send("someEvent");
```

An alternative is to combine the f:websocket tag with the h:commandScript tag. The <f:websocket onmessage> references exactly the <h:commandScript name>.

For example:

```
<h:panelGroup id="foo">
    ... (some complex UI here) ...
</h:panelGroup>
<f:websocket channel="someChannel" scope="view" onmessage="pushed" />
<h:form>
    <h:commandScript name="pushed" action="#{bean.pushed}" render=":foo" />
</h:form>
```

If you pass a Map<String,V> or a JavaBean as the push message object, then all entries or properties will transparently be available as request parameters in the command script method #{bean.pushed}.

Internationalizing and Localizing Web Applications



This section was last updated for Jakarta EE Version 9.1

The process of preparing an application to support more than one language and data format is called internationalization. Localization is the process of adapting an internationalized application to support a specific region or locale. Examples of locale-dependent information include messages and user interface labels, character sets and encoding, and date and currency formats. Although all client user interfaces should be internationalized and localized, it is particularly important for web applications because of the global nature of the web.

Java Platform Localization Classes

In the Java platform, `java.util.Locale` (https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html) represents a specific geographical, political, or cultural region. The string representation of a locale consists of the international standard two-character abbreviation for language and country and an optional variant, all separated by underscore (_) characters. Examples of locale strings include `fr` (French), `de_CH` (Swiss German), and `en_US_POSIX` (English on a POSIX-compliant platform).

Locale-sensitive data is stored in a `java.util.ResourceBundle` (https://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html). A resource bundle contains key-value pairs, where the keys uniquely identify a locale-specific object in the bundle. A resource bundle can be backed by a text file (properties resource bundle) or a class (list resource bundle) containing the pairs. You construct a resource bundle instance by appending a locale string representation to a base name.

The Duke's Bookstore application (see Duke's Bookstore Case Study Example) contains resource bundles with the base name `messages.properties` for the locales `de` (German), `es` (Spanish), and `fr` (French). The default locale, `en` (English), which is specified in the `faces-config.xml` file, uses the resource bundle with the base name, `messages.properties`.

For more details on internationalization and localization in the Java platform, see https://docs.oracle.com/javase/tutorial/i18n/index.html.

Providing Localized Messages and Labels

Messages and labels should be tailored according to the conventions of a user's language and region. There are two approaches to providing localized messages and labels in a web application.

- Provide a version of the web page in each of the target locales and have a controller servlet dispatch the request to the appropriate page depending on the requested locale. This approach is useful if large amounts of data on a page or an entire web application need to be internationalized.
- Isolate any locale-sensitive data on a page into resource bundles, and access the data so that the corresponding translated message is fetched automatically and inserted into the page. Thus, instead of creating strings directly in your code, you create a resource bundle that contains translations and read the translations from that bundle using the corresponding key.

The Duke's Bookstore application follows the second approach. Here are a few lines from the default resource bundle `messages.properties`:

```
TitleShoppingCart=Shopping Cart
TitleReceipt=Receipt
TitleBookCatalog=Book Catalog
TitleCashier=Cashier
TitleBookDescription=Book Description
Visitor=You are visitor number
What=What We\'re Reading
```

Establishing the Locale

To get the correct strings for a given user, a web application either retrieves the locale (set by a browser language preference) from the request using the `getLocale` method, or allows the user to explicitly select the locale.

A component can explicitly set the locale by using the `fmt:setLocale` tag.

The `locale-config` element in the configuration file registers the default locale and also registers other supported locales. This element in Duke's Bookstore registers English as the default locale and indicates that German, French, and Spanish are supported locales.

```
<locale-config>
    <default-locale>en</default-locale>
    <supported-locale>es</supported-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>fr</supported-locale>
</locale-config>
```

The `LocaleBean` in the Duke's Bookstore application uses the `getLocale` method to retrieve the locale.

```
public class LocaleBean {

    ...
    private FacesContext ctx = FacesContext.getCurrentInstance();
    private Locale locale = ctx.getViewRoot().getLocale();;

    ...
}
```

Setting the Resource Bundle

The resource bundle is set with the `resource-bundle` element in the configuration file. The setting for Duke's Bookstore looks like this:

```
<resource-bundle>
    <base-name>
        ee.jakarta.tutorial.dukesbookstore.web.messages.Messages
    </base-name>
    <var>bundle</var>
</resource-bundle>
```

After the locale is set, the controller of a web application could retrieve the resource bundle for that locale and save it as a session attribute (see Associating Objects with a Session) for use by other components or simply be used to return a text string appropriate for the selected locale:

```
public String toString(Locale locale) {
```

```
    ResourceBundle res =
        ResourceBundle.getBundle(
            "ee.jakarta.tutorial.dukesbookstore.web.messages.Messages", locale);
    return res.getString(name() + ".string");
}
```

Alternatively, an application could use the `f:loadBundle` tag to set the resource bundle. This tag loads the correct resource bundle according to the locale stored in `FacesContext`.

```
<f:loadBundle basename="ee.jakarta.tutorial.dukesbookstore.web.messages.Messages"
    var="bundle"/>
```

Resource bundles containing messages that are explicitly referenced from a Jakarta Faces tag attribute using a value expression must be registered using the `resource-bundle` element of the configuration file.

For more information on using this element, see Registering Application Messages.

Retrieving Localized Messages

A web component written in the Java programming language retrieves the resource bundle from the session:

```
ResourceBundle messages = (ResourceBundle)session.getAttribute("messages");
```

Then it looks up the string associated with the key `person.lastName` as follows:

```
messages.getString("person.lastName");
```

You can only use a `message` or `messages` tag to display messages that are queued onto a component as a result of a converter or validator being registered on the component. The following example shows a `message` tag that displays the error message queued on the `userNo` input component if the validator registered on the component fails to validate the value the user enters into the component.

```
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}">
    <f:validateLongRange minimum="0" maximum="10" />
</h:inputText>
...
<h:message style="color: red; text-decoration: overline"
           id="errors1" for="userNo"/>
```

For more information on using the `message` or `messages` tags, see Displaying Error Messages with the h:message and h:messages Tags.

Messages that are not queued on a component and are therefore not loaded automatically are referenced using a value expression. You can reference a localized message from almost any Jakarta Faces tag attribute.

The value expression that references a message has the same notation whether you loaded the resource bundle with the `loadBundle` tag or registered it with the `resource-bundle` element in the configuration file.

The value expression notation is `var.message`, in which `var` matches the `var` attribute of the `loadBundle` tag or the `var` element defined in the `resource-bundle` element of the configuration file, and `message` matches the key of the message contained in the resource bundle, referred to by the `var` attribute.

Here is an example from `bookcashier.xhtml` in Duke's Bookstore:

```
<h:outputLabel for="name" value="#{bundle.Name}" />
```

Notice that `bundle` matches the `var` element from the configuration file and that `Name` matches the key in the resource bundle.

Date and Number Formatting

Java programs use the `DateFormat.getDateInstance(int, locale)` method to parse and format dates in a locale-sensitive manner. Java programs use the `NumberFormat.getXXXInstance(locale)` method, where *XXX* can be `Currency`, `Number`, or `Percent`, to parse and format numerical values in a locale-sensitive manner.

An application can use date/time and number converters to format dates and numbers in a locale-sensitive manner. For example, a shipping date could be converted as follows:

```
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full"/>
</h:outputText>
```

For information on Jakarta Faces converters, see Using the Standard Converters.

Character Sets and Encodings

The following sections describe character sets and character encodings.

Character Sets

A character set is a set of textual and graphic symbols, each of which is mapped to a set of nonnegative integers.

The first character set used in computing was US-ASCII. It is limited in that it can represent only American English. US-ASCII contains uppercase and lowercase Latin alphabets, numerals, punctuation, a set of control codes, and a few miscellaneous symbols.

Unicode defines a standardized, universal character set that can be extended to accommodate additions. When the Java program source file encoding doesn't support Unicode, you can represent Unicode characters as escape sequences by using the notation `\u`*XXXX*, where *XXXX* is the character's 16-bit representation in hexadecimal. For example, the Spanish version of a message file could use Unicode for non-ASCII characters, as follows:

```
admin.nav.main=P\u00e1gina principal de administraci\u00f3n
```

Character Encoding

A character encoding maps a character set to units of a specific width and defines byte serialization and ordering rules. Many character sets have more than one encoding. For example, Java programs can represent Japanese character sets using the `EUC-JP` or `Shift-JIS` encodings, among others. Each encoding has rules for representing and serializing a character set.

The ISO 8859 series defines 13 character encodings that can represent texts in dozens of languages. Each ISO 8859 character encoding can have up to 256 characters. ISO-8859-1 (Latin-1) comprises the ASCII character set, characters with diacritics (accents, diaereses, cedillas, circumflexes, and so on), and additional symbols.

UTF-8 (Unicode Transformation Format, 8-bit form) is a variable-width character encoding that encodes 16-bit Unicode characters as one to four bytes. A byte in UTF-8 is equivalent to 7-bit ASCII if its high-order bit is zero; otherwise, the character comprises a variable number of bytes.

UTF-8 is compatible with the majority of existing web content and provides access to the Unicode character set. Current versions of browsers and email clients support UTF-8. In addition, many web standards specify UTF-8 as their character encoding. For example, UTF-8 is one of the two required encodings for XML documents (the other is UTF-16).

Web components usually use `PrintWriter` to produce responses; `PrintWriter` automatically encodes using ISO-8859-1. Servlets can also output binary data using `OutputStream` classes, which perform no encoding. An application that uses a character set that cannot use the default encoding must explicitly set a different encoding.

Jakarta WebSocket



This section was last updated for Jakarta EE Version 9.1

This chapter describes Jakarta WebSocket, which provides support for creating WebSocket applications. WebSocket is an application protocol that provides full-duplex communications between two peers over the TCP protocol.

Introduction to WebSocket

In the traditional request-response model used in HTTP, the client requests resources, and the server provides responses. The exchange is always initiated by the client; the server cannot send any data without the client requesting it first. This model worked well for the World Wide Web

when clients made occasional requests for documents that changed infrequently, but the limitations of this approach are increasingly relevant as content changes quickly and users expect a more interactive experience on the Web. The WebSocket protocol addresses these limitations by providing a full-duplex communication channel between the client and the server. Combined with other client technologies, such as JavaScript and HTML5, WebSocket enables web applications to deliver a richer user experience.

In a WebSocket application, the server publishes a WebSocket endpoint, and the client uses the endpoint's URI to connect to the server. The WebSocket protocol is symmetrical after the connection has been established; the client and the server can send messages to each other at any time while the connection is open, and they can close the connection at any time. Clients usually connect only to one server, and servers accept connections from multiple clients.

The WebSocket protocol has two parts: handshake and data transfer. The client initiates the handshake by sending a request to a WebSocket endpoint using its URI. The handshake is compatible with existing HTTP-based infrastructure: web servers interpret it as an HTTP connection upgrade request. An example handshake from a client looks like this:

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

An example handshake from the server in response to the client looks like this:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

The server applies a known operation to the value of the `Sec-WebSocket-Key` header to generate the value of the `Sec-WebSocket-Accept` header. The client applies the same operation to the value of the `Sec-WebSocket-Key` header, and the connection is established successfully if the result matches the value received from the server. The client and the server can send messages to each other after a successful handshake.

WebSocket supports text messages (encoded as UTF-8) and binary messages. The control frames in WebSocket are close, ping, and pong (a response to a ping frame). Ping and pong frames may also contain application data.

WebSocket endpoints are represented by URIs that have the following form:

```
ws://host:port/path?query
```

```
wss://host:port/path?query
```

The `ws` scheme represents an unencrypted WebSocket connection, and the `wss` scheme represents an encrypted connection. The `port` component is optional; the default port number is 80 for unencrypted connections and 443 for encrypted connections. The `path` component indicates the location of an endpoint within a server. The `query` component is optional.

Modern web browsers implement the WebSocket protocol and provide a JavaScript API to connect to endpoints, send messages, and assign callback methods for WebSocket events (such as opened connections, received messages, and closed connections).

Creating WebSocket Applications in the Jakarta EE Platform

The Jakarta EE platform includes Jakarta WebSocket, which enables you to create, configure, and deploy WebSocket endpoints in web applications. The WebSocket client API specified in Jakarta WebSocket also enables you to access remote WebSocket endpoints from any Java application.

Jakarta WebSocket consists of the following packages.

- The `jakarta.websocket.server` package contains annotations, classes, and interfaces to create and configure server endpoints.
- The `jakarta.websocket` package contains annotations, classes, interfaces, and exceptions that are common to client and server endpoints.

WebSocket endpoints are instances of the `jakarta.websocket.Endpoint` class. Jakarta WebSocket enables you to create two kinds of endpoints: programmatic endpoints and annotated endpoints. To create a programmatic endpoint, you extend the `Endpoint` class and override its lifecycle methods. To create an annotated endpoint, you decorate a Java class and some of its methods with the annotations provided by the packages mentioned previously. After you have created an endpoint, you deploy it to an specific URI in the application so that remote clients can connect to it.



In most cases, it is easier to create and deploy an annotated endpoint than a programmatic endpoint. This chapter provides a simple example of a programmatic endpoint, but it focuses on annotated endpoints.

Creating and Deploying a WebSocket Endpoint

The process for creating and deploying a WebSocket endpoint:

1. Create an endpoint class.
2. Implement the lifecycle methods of the endpoint.
3. Add your business logic to the endpoint.
4. Deploy the endpoint inside a web application.

The process is slightly different for programmatic endpoints and annotated endpoints, and it is covered in detail in the following sections.



As opposed to servlets, WebSocket endpoints are instantiated multiple times. The container creates an instance of an endpoint per connection to its deployment URI. Each instance is associated with one and only one connection. This facilitates keeping user state for each connection and makes development easier, because there is only one thread executing the code of an endpoint instance at any given time.

Programmatic Endpoints

The following example shows how to create an endpoint by extending the `Endpoint` class:

```
public class EchoEndpoint extends Endpoint {
   @Override
   public void onOpen(final Session session, EndpointConfig config) {
      session.addMessageHandler(new MessageHandler.Whole<String>() {
         @Override
         public void onMessage(String msg) {
            try {
               session.getBasicRemote().sendText(msg);
            } catch (IOException e) { ... }
         }
      });
   }
}
```

This endpoint echoes every message received. The `Endpoint` class defines three lifecycle methods: `onOpen`, `onClose`, and `onError`. The `EchoEndpoint` class implements the `onOpen` method, which is the only abstract method in the `Endpoint` class.

The `Session` parameter represents a conversation between this endpoint and the remote endpoint. The `addMessageHandler` method registers message handlers, and the `getBasicRemote` method returns an object that represents the remote endpoint. The `Session` interface is covered in detail in the rest of this chapter.

The message handler is implemented as an anonymous inner class. The `onMessage` method of the message handler is invoked when the endpoint receives a text message.

To deploy this programmatic endpoint, use the following code in your Jakarta EE application:

```
ServerEndpointConfig.Builder.create(EchoEndpoint.class, "/echo").build();
```

When you deploy your application, the endpoint is available at `ws://<host>:<port>/<application>/echo`; for example, `ws://localhost:8080/echoapp/echo`.

Annotated Endpoints

The following example shows how to create the same endpoint from Programmatic Endpoints

using annotations instead:

```
@ServerEndpoint("/echo")
public class EchoEndpoint {
   @OnMessage
   public void onMessage(Session session, String msg) {
      try {
         session.getBasicRemote().sendText(msg);
      } catch (IOException e) { ... }
   }
}
```

The annotated endpoint is simpler than the equivalent programmatic endpoint, and it is deployed automatically with the application to the relative path defined in the `ServerEndpoint` annotation. Instead of having to create an additional class for the message handler, this example uses the `OnMessage` annotation to designate the method invoked to handle messages.

WebSocket Endpoint Lifecycle Annotations lists the annotations available in the `jakarta.websocket` package to designate the methods that handle lifecycle events. The examples in the table show the most common parameters for these methods. See the API reference for details on what combinations of parameters are allowed in each case.

WebSocket Endpoint Lifecycle Annotations

Annotation	Event	Example

`OnOpen`	Connection opened	`@OnOpen`
		`public void open(Session session,`
		`EndpointConfig conf)`
		`{ }`

`OnMessage`	Message received	`@OnMessage`
		`public void message(Session session,`
		`String msg) { }`

`OnError`	Connection error	`@OnError`
		`public void error(Session session,`
		`Throwable error) { }`

Annotation	Event	Example

`OnClose`	Connection closed	`@OnClose`
		`public void close(Session session,`
		`CloseReason reason)`
		`{ }`

Sending and Receiving Messages

WebSocket endpoints can send and receive text and binary messages. In addition, they can also send ping frames and receive pong frames. This section describes how to use the `Session` and `RemoteEndpoint` interfaces to send messages to the connected peer and how to use the `OnMessage` annotation to receive messages from it.

Sending Messages

Follow these steps to send messages in an endpoint.

1. Obtain the `Session` object from the connection.

   The `Session` object is available as a parameter in the annotated lifecycle methods of the endpoint, like those in Websocket Endpoint Lifecycle Annotations. When your message is a response to a message from the peer, you have the `Session` object available inside the method that received the message (the method annotated with `@OnMessage`). If you have to send messages that are not responses, store the `Session` object as an instance variable of the endpoint class in the method annotated with `@OnOpen` so that you can access it from other methods.

2. Use the `Session` object to obtain a `RemoteEndpoint` object.

   The `Session.getBasicRemote` method and the `Session.getAsyncRemote` method return `RemoteEndpoint.Basic` and `RemoteEndpoint.Async` objects respectively. The `RemoteEndpoint.Basic` interface provides blocking methods to send messages; the `RemoteEndpoint.Async` interface provides nonblocking methods.

3. Use the `RemoteEndpoint` object to send messages to the peer.

   The following list shows some of the methods you can use to send messages to the peer.

   - `void RemoteEndpoint.Basic.sendText(String text)`

     Send a text message to the peer. This method blocks until the whole message has been transmitted.

   - `void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)`

     Send a binary message to the peer. This method blocks until the whole message has been transmitted.

   - `void RemoteEndpoint.sendPing(ByteBuffer appData)`

end a ping frame to the peer.

- `void RemoteEndpoint.sendPong(ByteBuffer appData)`

  Send a pong frame to the peer.

The example in Annotated Endpoints demonstrates how to use this procedure to reply to every incoming text message.

Sending Messages to All Peers Connected to an Endpoint

Each instance of an endpoint class is associated with one and only one connection and peer; however, there are cases in which an endpoint instance needs to send messages to all connected peers. Examples include chat applications and online auctions. The `Session` interface provides the `getOpenSessions` method for this purpose. The following example demonstrates how to use this method to forward incoming text messages to all connected peers:

```
@ServerEndpoint("/echoall")
public class EchoAllEndpoint {
   @OnMessage
   public void onMessage(Session session, String msg) {
      try {
         for (Session sess : session.getOpenSessions()) {
            if (sess.isOpen())
               sess.getBasicRemote().sendText(msg);
         }
      } catch (IOException e) { ... }
   }
}
```

Receiving Messages

The `OnMessage` annotation designates methods that handle incoming messages. You can have at most three methods annotated with `@OnMessage` in an endpoint, one for each message type: text, binary, and pong. The following example demonstrates how to designate methods to receive all three types of messages:

```
@ServerEndpoint("/receive")
public class ReceiveEndpoint {
   @OnMessage
   public void textMessage(Session session, String msg) {
      System.out.println("Text message: " + msg);
   }
   @OnMessage
   public void binaryMessage(Session session, ByteBuffer msg) {
      System.out.println("Binary message: " + msg.toString());
   }
   @OnMessage
   public void pongMessage(Session session, PongMessage msg) {
```

```
      System.out.println("Pong message: " +
                         msg.getApplicationData().toString());
   }
}
```

Maintaining Client State

Because the container creates an instance of the endpoint class for every connection, you can define and use instance variables to store client state information. In addition, the `Session.getUserProperties` method provides a modifiable map to store user properties. For example, the following endpoint replies to incoming text messages with the contents of the previous message from each client:

```
@ServerEndpoint("/delayedecho")
public class DelayedEchoEndpoint {
   @OnOpen
   public void open(Session session) {
      session.getUserProperties().put("previousMsg", " ");
   }
   @OnMessage
   public void message(Session session, String msg) {
      String prev = (String) session.getUserProperties()
                                    .get("previousMsg");
      session.getUserProperties().put("previousMsg", msg);
      try {
         session.getBasicRemote().sendText(prev);
      } catch (IOException e) { ... }
   }
}
```

To store information common to all connected clients, you can use class (static) variables; however, you are responsible for ensuring thread-safe access to them.

Using Encoders and Decoders

Jakarta WebSocket provides support for converting between WebSocket messages and custom Java types using encoders and decoders. An encoder takes a Java object and produces a representation that can be transmitted as a WebSocket message; for example, encoders typically produce JSON, XML, or binary representations. A decoder performs the reverse function; it reads a WebSocket message and creates a Java object.

This mechanism simplifies WebSocket applications, because it decouples the business logic from the serialization and deserialization of objects.

Implementing Encoders to Convert Java Objects into WebSocket Messages

The procedure to implement and use encoders in endpoints follows.

1. Implement one of the following interfaces:
   - `Encoder.Text<T>` for text messages
   - `Encoder.Binary<T>` for binary messages

   These interfaces specify the `encode` method. Implement an encoder class for each custom Java type that you want to send as a WebSocket message.
2. Add the names of your encoder implementations to the `encoders` optional parameter of the `ServerEndpoint` annotation.
3. Use the `sendObject(Object data)` method of the `RemoteEndpoint.Basic` or `RemoteEndpoint.Async` interfaces to send your objects as messages. The container looks for an encoder that matches your type and uses it to convert the object to a WebSocket message.

For example, if you have two Java types (`MessageA` and `MessageB`) that you want to send as text messages, implement the `Encoder.Text<MessageA>` and `Encoder.Text<MessageB>` interfaces as follows:

```
public class MessageATextEncoder implements Encoder.Text<MessageA> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(MessageA msgA) throws EncodeException {
      // Access msgA's properties and convert to JSON text...
      return msgAJsonString;
   }
}
```

Implement `Encoder.Text<MessageB>` similarly. Then, add the `encoders` parameter to the `ServerEndpoint` annotation as follows:

```
@ServerEndpoint(
   value = "/myendpoint",
   encoders = { MessageATextEncoder.class, MessageBTextEncoder.class }
)
public class EncEndpoint { ... }
```

Now, you can send `MessageA` and `MessageB` objects as WebSocket messages using the `sendObject` method as follows:

```
MessageA msgA = new MessageA(...);
MessageB msgB = new MessageB(...);
session.getBasicRemote.sendObject(msgA);
session.getBasicRemote.sendObject(msgB);
```

As in this example, you can have more than one encoder for text messages and more than one

encoder for binary messages. Like endpoints, encoder instances are associated with one and only one WebSocket connection and peer, so there is only one thread executing the code of an encoder instance at any given time.

Implementing Decoders to Convert WebSocket Messages into Java Objects

The procedure to implement and use decoders in endpoints follows.

1. Implement one of the following interfaces:
   - `Decoder.Text<T>` for text messages
   - `Decoder.Binary<T>` for binary messages

   These interfaces specify the `willDecode` and `decode` methods.

   Unlike with encoders, you can specify at most one decoder for binary messages and one decoder for text messages.

2. Add the names of your decoder implementations to the `decoders` optional parameter of the `ServerEndpoint` annotation.
3. Use the `OnMessage` annotation in the endpoint to designate a method that takes your custom Java type as a parameter. When the endpoint receives a message that can be decoded by one of the decoders you specified, the container calls the method annotated with `@OnMessage` that takes your custom Java type as a parameter if this method exists.

For example, if you have two Java types (`MessageA` and `MessageB`) that you want to send and receive as text messages, define them so that they extend a common class (`Message`). Because you can only define one decoder for text messages, implement a decoder for the `Message` class as follows:

```
public class MessageTextDecoder implements Decoder.Text<Message> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public Message decode(String string) throws DecodeException {
      // Read message...
      if ( /* message is an A message */ )
         return new MessageA(...);
      else if ( /* message is a B message */ )
         return new MessageB(...);
   }
   @Override
   public boolean willDecode(String string) {
      // Determine if the message can be converted into either a
      // MessageA object or a MessageB object...
      return canDecode;
   }
}
```

Then, add the `decoder` parameter to the `ServerEndpoint` annotation as follows:

```
@ServerEndpoint(
   value = "/myendpoint",
   encoders = { MessageATextEncoder.class, MessageBTextEncoder.class },
   decoders = { MessageTextDecoder.class }
)
public class EncDecEndpoint { ... }
```

Now, define a method in the endpoint class that receives `MessageA` and `MessageB` objects as follows:

```
@OnMessage
public void message(Session session, Message msg) {
   if (msg instanceof MessageA) {
      // We received a MessageA object...
   } else if (msg instanceof MessageB) {
      // We received a MessageB object...
   }
}
```

Like endpoints, decoder instances are associated with one and only one WebSocket connection and peer, so there is only one thread executing the code of a decoder instance at any given time.

Path Parameters

The `ServerEndpoint` annotation enables you to use URI templates to specify parts of an endpoint deployment URI as application parameters. For example, consider this endpoint:

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
   ...
}
```

If the endpoint is deployed inside a web application called `chatapp` at a local Jakarta EE server in port 8080, clients can connect to the endpoint using any of the following URIs:

```
http://localhost:8080/chatapp/chatrooms/currentnews
http://localhost:8080/chatapp/chatrooms/music
http://localhost:8080/chatapp/chatrooms/cars
http://localhost:8080/chatapp/chatrooms/technology
```

Annotated endpoints can receive path parameters as arguments in methods annotated with `@OnOpen`, `@OnMessage`, and `@OnClose`. In this example, the endpoint uses the parameter in the `@OnOpen` method to determine which chat room the client wants to join:

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
   @OnOpen
   public void open(Session session,
                    EndpointConfig c,
                    @PathParam("room-name") String roomName) {
      // Add the client to the chat room of their choice ...
   }
}
```

The path parameters used as arguments in these methods can be strings, primitive types, or the corresponding wrapper types.

Handling Errors

To designate a method that handles errors in an annotated WebSocket endpoint, decorate it with `@OnError`:

```
@ServerEndpoint("/testendpoint")
public class TestEndpoint {
   ...
   @OnError
   public void error(Session session, Throwable t) {
      t.printStackTrace();
      ...
   }
}
```

This method is invoked when there are connection problems, runtime errors from message handlers, or conversion errors when decoding messages.

Specifying an Endpoint Configurator Class

Jakarta WebSocket enables you to configure how the container creates server endpoint instances. You can provide custom endpoint configuration logic to:

- Access the details of the initial HTTP request for a WebSocket connection
- Perform custom checks on the `Origin` HTTP header
- Modify the WebSocket handshake response
- Choose a WebSocket subprotocol from those requested by the client
- Control the instantiation and initialization of endpoint instances

To provide custom endpoint configuration logic, you extend the `ServerEndpointConfig.Configurator` class and override some of its methods. In the endpoint class, you specify the configurator class using the `configurator` parameter of the `ServerEndpoint` annotation.

For example, the following configurator class makes the handshake request object available to endpoint instances:

```
public class CustomConfigurator extends ServerEndpointConfig.Configurator {

    @Override
    public void modifyHandshake(ServerEndpointConfig conf,
                                HandshakeRequest req,
                                HandshakeResponse resp) {

        conf.getUserProperties().put("handshakereq", req);
    }

}
```

The following endpoint class configures endpoint instances with the custom configurator, which enables them to access the handshake request object:

```
@ServerEndpoint(
    value = "/myendpoint",
    configurator = CustomConfigurator.class
)
public class MyEndpoint {

    @OnOpen
    public void open(Session s, EndpointConfig conf) {
        HandshakeRequest req = (HandshakeRequest) conf.getUserProperties()
                                                      .get("handshakereq");
        Map<String,List<String>> headers = req.getHeaders();
        ...
    }
}
```

The endpoint class can use the handshake request object to access the details of the initial HTTP request, such as its headers or the `HttpSession` object.

For more information on endpoint configuration, see the API reference for the `ServerEndpointConfig.Configurator` class.

The dukeetf2 Example Application

The `dukeetf2` example application, located in the `jakartaee-examples/tutorial/web/websocket/dukeetf2/` directory, demonstrates how to use a WebSocket endpoint to provide data updates to web clients. The example resembles a service that provides periodic updates on the price and trading volume of an electronically traded fund (ETF).

Architecture of the dukeetf2 Sample Application

The `dukeetf2` example application consists of a WebSocket endpoint, an enterprise bean, and an HTML page.

- The endpoint accepts connections from clients and sends them updates when new data for price and trading volume becomes available.
- The enterprise bean updates the price and volume information once every second.
- The HTML page uses JavaScript code to connect to the WebSocket endpoint, parse incoming messages, and update the price and volume information without reloading the page.

The Endpoint

The WebSocket endpoint is implemented in the `ETFEndpoint` class, which stores all connected sessions in a queue and provides a method that the enterprise bean calls when there is new information available to send:

```
@ServerEndpoint("/dukeetf")
public class ETFEndpoint {
   private static final Logger logger = Logger.getLogger("ETFEndpoint");
   /* Queue for all open WebSocket sessions */
   static Queue<Session> queue = new ConcurrentLinkedQueue<>();

   /* PriceVolumeBean calls this method to send updates */
   public static void send(double price, int volume) {
      String msg = String.format("%.2f / %d", price, volume);
      try {
         /* Send updates to all open WebSocket sessions */
         for (Session session : queue) {
            session.getBasicRemote().sendText(msg);
            logger.log(Level.INFO, "Sent: {0}", msg);
         }
      } catch (IOException e) {
         logger.log(Level.INFO, e.toString());
      }
    }
    ...
}
```

The lifecycle methods of the endpoint add and remove sessions to and from the queue:

```
@ServerEndpoint("/dukeetf")
public class ETFEndpoint {
   ...
   @OnOpen
   public void openConnection(Session session) {
      /* Register this connection in the queue */
      queue.add(session);
      logger.log(Level.INFO, "Connection opened.");
```

```
    }

    @OnClose
    public void closedConnection(Session session) {
        /* Remove this connection from the queue */
        queue.remove(session);
        logger.log(Level.INFO, "Connection closed.");
    }

    @OnError
    public void error(Session session, Throwable t) {
        /* Remove this connection from the queue */
        queue.remove(session);
        logger.log(Level.INFO, t.toString());
        logger.log(Level.INFO, "Connection error.");
    }
}
```

The Enterprise Bean

The enterprise bean uses the timer service to generate new price and volume information every second:

```
@Startup
@Singleton
public class PriceVolumeBean {
    /* Use the container's timer service */
    @Resource TimerService tservice;
    private Random random;
    private volatile double price = 100.0;
    private volatile int volume = 300000;
    private static final Logger logger = Logger.getLogger("PriceVolumeBean");

    @PostConstruct
    public void init() {
        /* Initialize the EJB and create a timer */
        logger.log(Level.INFO, "Initializing EJB.");
        random = new Random();
        tservice.createIntervalTimer(1000, 1000, new TimerConfig());
    }

    @Timeout
    public void timeout() {
        /* Adjust price and volume and send updates */
        price += 1.0*(random.nextInt(100)-50)/100.0;
        volume += random.nextInt(5000) - 2500;
        ETFEndpoint.send(price, volume);
    }
}
```

The enterprise bean calls the `send` method of the `ETFEndpoint` class in the timeout method. See Using the Timer Service in [entbeans:ejb-basicexamples::ejb-basicexamples:::_running_the_enterprise_bean_examples] for more information on the timer service.

The HTML Page

The HTML page consists of a table and some JavaScript code. The table contains two fields referenced from JavaScript code:

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    ...
    <table>
      ...
      <td id="price">--.--</td>
      ...
      <td id="volume">--</td>
      ...
    </table>
    ...
  </body>
</html>
```

The JavaScript code uses the WebSocket API to connect to the server endpoint and to designate a callback method for incoming messages. The callback method updates the page with the new information.

```
var wsocket;
function connect() {
   wsocket = new WebSocket("ws://localhost:8080/dukeetf2/dukeetf");
   wsocket.onmessage = onMessage;
}
function onMessage(evt) {
   var arraypv = evt.data.split("/");
   document.getElementById("price").innerHTML = arraypv[0];
   document.getElementById("volume").innerHTML = arraypv[1];
}
window.addEventListener("load", connect, false);
```

The WebSocket API is supported by most modern browsers, and it is widely used in HTML5 web client development.

Running the dukeetf2 Example Application

This section describes how to run the `dukeetf2` example application using NetBeans IDE and from

the command line.

To Run the dukeetf2 Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/web/websocket
   ```

4. Select the `dukeetf2` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `dukeetf2` project and select **Run**.

   This command builds and packages the application into a WAR file (`dukeetf2.war`) located in the `target/` directory, deploys it to the server, and launches a web browser window with the following URL:

   ```
   http://localhost:8080/dukeetf2/
   ```

   Open the same URL on a different web browser tab or window to see how both pages get price and volume updates simultaneously.

To Run the dukeetf2 Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/websocket/dukeetf2/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

4. Open a web browser window and enter the following URL:

   ```
   http://localhost:8080/dukeetf2/
   ```

   Open the same URL on a different web browser tab or window to see how both pages get price and volume updates simultaneously.

The websocketbot Example Application

The `websocketbot` example application, located in the `jakartaee-examples/tutorial/web/websocket/websocketbot/` directory, demonstrates how to use a WebSocket endpoint to implement a chat. The example resembles a chat room in which many users can join and have a conversation. Users can ask simple questions to a bot agent that is always available in the chat room.

Architecture of the websocketbot Example Application

The `websocketbot` example application consists of the following elements:

- The CDI Bean – A CDI bean (`BotBean`) that contains the logic for the bot agent to reply to messages
- The WebSocket Endpoint – A WebSocket endpoint (`BotEndpoint`) that implements the chat room
- The Application Messages – A set of classes (`Message`, `ChatMessage`, `InfoMessage`, `JoinMessage`, and `UsersMessage`) that represent application messages
- The Encoder Classes – A set of classes (`ChatMessageEncoder`, `InfoMessageEncoder`, `JoinMessageEncoder`, and `UsersMessageEncoder`) that encode application messages into WebSocket text messages as JSON data
- The Message Decoder – A class (`MessageDecoder`) the parses WebSocket text messages as JSON data and decodes them into `JoinMessage` or `ChatMessage` objects
- The HTML Page – An HTML page (`index.html`) that uses JavaScript code to implement the client for the chat room

The CDI Bean

The CDI bean (`BotBean`) is a Java class that contains the `respond` method. This method compares the incoming chat message with a set of predefined questions and returns a chat response.

```
@Named
public class BotBean {
    public String respond(String msg) { ... }
}
```

The WebSocket Endpoint

The WebSocket endpoint (`BotEndpoint`) is an annotated endpoint that performs the following functions:

- Receives messages from clients
- Forwards messages to clients
- Maintains a list of connected clients
- Invokes the bot agent functionality

The endpoint specifies its deployment URI and the message encoders and decoders using the

`ServerEndpoint` annotation. The endpoint obtains an instance of the `BotBean` class and a managed executor service resource through dependency injection:

```
@ServerEndpoint(
   value = "/websocketbot",
   decoders = { MessageDecoder.class },
   encoders = { JoinMessageEncoder.class, ChatMessageEncoder.class,
                InfoMessageEncoder.class, UsersMessageEncoder.class }
)
/* There is a BotEndpoint instance per connection */
public class BotEndpoint {
   private static final Logger logger = Logger.getLogger("BotEndpoint");
   /* Bot functionality bean */
   @Inject private BotBean botbean;
   /* Executor service for asynchronous processing */
   @Resource(name="comp/DefaultManagedExecutorService")
   private ManagedExecutorService mes;

   @OnOpen
   public void openConnection(Session session) {
       logger.log(Level.INFO, "Connection opened.");
   }
   ...
}
```

The `message` method processes incoming messages from clients. The decoder converts incoming text messages into `JoinMessage` or `ChatMessage` objects, which inherit from the `Message` class. The `message` method receives a `Message` object as a parameter:

```
@OnMessage
public void message(Session session, Message msg) {
   logger.log(Level.INFO, "Received: {0}", msg.toString());

   if (msg instanceof JoinMessage) {
      /* Add the new user and notify everybody */
      JoinMessage jmsg = (JoinMessage) msg;
      session.getUserProperties().put("name", jmsg.getName());
      session.getUserProperties().put("active", true);
      logger.log(Level.INFO, "Received: {0}", jmsg.toString());
      sendAll(session, new InfoMessage(jmsg.getName() +
              " has joined the chat"));
      sendAll(session, new ChatMessage("Duke", jmsg.getName(),
              "Hi there!!"));
      sendAll(session, new UsersMessage(this.getUserList(session)));

   } else if (msg instanceof ChatMessage) {
      /* Forward the message to everybody */
      ChatMessage cmsg = (ChatMessage) msg;
      logger.log(Level.INFO, "Received: {0}", cmsg.toString());
```

```
        sendAll(session, cmsg);
        if (cmsg.getTarget().compareTo("Duke") == 0) {
            /* The bot replies to the message */
            mes.submit(new Runnable() {
                @Override
                public void run() {
                    String resp = botbean.respond(cmsg.getMessage());
                    sendAll(session, new ChatMessage("Duke",
                            cmsg.getName(), resp));
                }
            });
        }
    }
}
```

If the message is a join message, the endpoint adds the new user to the list and notifies all connected clients. If the message is a chat message, the endpoint forwards it to all connected clients.

If a chat message is for the bot agent, the endpoint obtains a response using the `BotBean` instance and sends it to all connected clients. The `sendAll` method is similar to the example in Sending Messages to All Peers Connected to an Endpoint.

Asynchronous Processing and Concurrency Considerations

The WebSocket endpoint calls the `BotBean.respond` method to obtain a response from the bot. In this example, this is a blocking operation; the user that sent the associated message would not be able to send or receive other chat messages until the operation completes. To avoid this problem, the endpoint obtains an executor service from the container and executes the blocking operation in a different thread using the `ManagedExecutorService.submit` method from Concurrency Utilities for Jakarta EE.

Jakarta WebSocket specification requires that Jakarta EE implementations instantiate endpoint classes once per connection. This facilitates the development of WebSocket endpoints, because you are guaranteed that only one thread is executing the code in a WebSocket endpoint class at any given time. When you introduce a new thread in an endpoint, as in this example, you must ensure that variables and methods accessed by more than one thread are thread safe. In this example, the code in `BotBean` is thread safe, and the `BotEndpoint.sendAll` method has been declared `synchronized`.

Refer to [supporttechs:concurrency-utilities::concurrency-utilities:::_jakarta_concurrency] for more information on the managed executor service and Concurrency Utilities for Jakarta EE.

The Application Messages

The classes that represent application messages (`Message`, `ChatMessage`, `InfoMessage`, `JoinMessage`, and `UsersMessage`) contain only properties and getter and setter methods. For example, the `ChatMessage` class looks like this:

```
public class ChatMessage extends Message {
```

```
    private String name;
    private String target;
    private String message;
    /* ... Constructor, getters, and setters ... */
}
```

The Encoder Classes

The encoder classes convert application message objects into JSON text using the Java API for JSON Processing. For example, the `ChatMessageEncoder` class is implemented as follows:

```
/* Encode a ChatMessage as JSON.
 * For example, (new ChatMessage("Peter","Duke","How are you?"))
 * is encoded as follows:
 * {"type":"chat","target":"Duke","message":"How are you?"}
 */
public class ChatMessageEncoder implements Encoder.Text<ChatMessage> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(ChatMessage chatMessage) throws EncodeException {
      // Access properties in chatMessage and write JSON text...
   }
}
```

See [web:jsonp::jsonp:::_json_processing] for more information on the Jakarta JSON Processing.

The Message Decoder

The message decoder (`MessageDecoder`) class converts WebSocket text messages into application messages by parsing JSON text. It is implemented as follows:

```
/* Decode a JSON message into a JoinMessage or a ChatMessage.
 * For example, the incoming message
 * {"type":"chat","name":"Peter","target":"Duke","message":"How are you?"}
 * is decoded as (new ChatMessage("Peter", "Duke", "How are you?"))
 */
public class MessageDecoder implements Decoder.Text<Message> {
    /* Stores the name-value pairs from a JSON message as a Map */
    private Map<String,String> messageMap;

    @Override
    public void init(EndpointConfig ec) { }
    @Override
    public void destroy() { }

    /* Create a new Message object if the message can be decoded */
```

```
    @Override
    public Message decode(String string) throws DecodeException {
       Message msg = null;
       if (willDecode(string)) {
          switch (messageMap.get("type")) {
             case "join":
                msg = new JoinMessage(messageMap.get("name"));
                break;
             case "chat":
                msg = new ChatMessage(messageMap.get("name"),
                                      messageMap.get("target"),
                                      messageMap.get("message"));
          }
       } else {
          throw new DecodeException(string, "[Message] Can't decode.");
       }
       return msg;
   }

   /* Decode a JSON message into a Map and check if it contains
    * all the required fields according to its type. */
   @Override
   public boolean willDecode(String string) {
      // Convert JSON data from the message into a name-value map...
      // Check if the message has all the fields for its message type...
   }
}
```

The HTML Page

The HTML page (`index.html`) contains a field for the user name. After the user types a name and clicks Join, three text areas are available: one to type and send messages, one for the chat room, and one with the list of users. The page also contains a WebSocket console that shows the messages sent and received as JSON text.

The JavaScript code on the page uses the WebSocket API to connect to the endpoint, send messages, and designate callback methods. The WebSocket API is supported by most modern browsers and is widely used for web client development with HTML5.

Running the websocketbot Example Application

This section describes how to run the `websocketbot` example application using NetBeans IDE and from the command line.

To Run the websocketbot Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the Open Project dialog box, navigate to:

```
jakartaee-examples/tutorial/web/websocket
```

4. Select the `websocketbot` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `websocketbot` project and select **Run**.

   This command builds and packages the application into a WAR file, `websocketbot.war`, located in the `target/` directory; deploys it to the server; and launches a web browser window with the following URL:

   ```
   http://localhost:8080/websocketbot/
   ```

   See To Test the websocketbot Example Application for more information.

To Run the websocketbot Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/web/websocket/websocketbot/
   ```

3. Enter the following command to deploy the application:

   ```
   mvn install
   ```

4. Open a web browser window and type the following address:

   ```
   http://localhost:8080/websocketbot/
   ```

   See To Test the websocketbot Example Application for more information.

To Test the websocketbot Example Application

1. On the main page, type your name on the first text field and press the Enter key.

   The list of connected users appears on the text area on the right. The text area on the left is the chat room.

2. Type a message on the text area below the login button. For example, type the messages in bold and press enter to obtain responses similar to the following:

   ```
   [--Peter has joined the chat--]
   Duke: @Peter Hi there!!
   ```

```
Peter: @Duke how are you?
Duke: @Peter I'm doing great, thank you!
Peter: @Duke when is your birthday?
Duke: @Peter My birthday is on May 23rd. Thanks for asking!
```

3. Join the chat from another browser window by copying and pasting the URI on the address bar and joining with a different name.

   The new user name appears in the list of users in both browser windows. You can send messages from either window and see how they appear in the other.

4. Click Show WebSocket Console.

   The console shows the messages sent and received as JSON text.

Further Information about WebSocket

For more information on WebSocket in Jakarta EE, see the Jakarta WebSocket specification:

https://jakarta.ee/specifications/websocket/2.1/

Jakarta Persistence

Introduction to Jakarta Persistence



This section was last updated for Jakarta EE Version 9.1

This chapter provides a description of Jakarta Persistence.

Overview of Jakarta Persistence

Jakarta Persistence provides Java developers with an object/relational mapping facility for managing relational data in Java applications. Jakarta Persistence consists of four areas:

- Jakarta Persistence
- The query language
- The Jakarta Persistence Criteria API
- Object/relational mapping metadata

Entities

An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented through either persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Requirements for Entity Classes

An entity class must follow these requirements.

- The class must be annotated with the `jakarta.persistence.Entity` annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared `final`. No methods or persistent instance variables must be declared `final`.
- If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed through either the entity's instance variables or properties. The fields or properties must be of the following Java language types:

- Java primitive types
- `java.lang.String`
- Other serializable types, including:
  - Wrappers of Java primitive types
  - `java.math.BigInteger`
  - `java.math.BigDecimal`
  - `java.util.Date`
  - `java.util.Calendar`
  - `java.sql.Date`
  - `java.sql.Time`
  - `java.sql.TimeStamp`
  - User-defined serializable types
  - `byte[]`
  - `Byte[]`
  - `char[]`
  - `Character[]`
- Enumerated types
- Other entities and/or collections of entities

- Embeddable classes

Entities may use persistent fields, persistent properties, or a combination of both. If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties.

Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity-class instance variables directly. All fields not annotated `jakarta.persistence.Transient` or not marked as Java `transient` will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

Persistent Properties

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names. For every persistent property `property` of type `Type` of the entity, there is a getter method `getProperty` and setter method `setProperty`. If the property is a Boolean, you may use `isProperty` instead of `getProperty`. For example, if a `Customer` entity uses persistent properties and has a private instance variable called `firstName`, the class defines a `getFirstName` and `setFirstName` method for retrieving and setting the state of the `firstName` instance variable.

The method signatures for single-valued persistent properties are as follows:

```
Type getProperty()
void setProperty(Type type)
```

The object/relational mapping annotations for persistent properties must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated `@Transient` or marked `transient`.

Using Collections in Entity Fields and Properties

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

If the entity class uses persistent fields, the type in the preceding method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if it has a persistent property that contains a set of phone numbers, the `Customer` entity would have the

following methods:

```
Set<PhoneNumber> getPhoneNumbers() { ... }
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

If a field or property of an entity consists of a collection of basic types or embeddable classes, use the `jakarta.persistence.ElementCollection` annotation on the field or property.

The two attributes of `@ElementCollection` are `targetClass` and `fetch`. The `targetClass` attribute specifies the class name of the basic or embeddable class and is optional if the field or property is defined using Java programming language generics. The optional `fetch` attribute is used to specify whether the collection should be retrieved lazily or eagerly, using the `jakarta.persistence.FetchType` constants of either `LAZY` or `EAGER`, respectively. By default, the collection will be fetched lazily.

The following entity, `Person`, has a persistent field, `nicknames`, which is a collection of `String` classes that will be fetched eagerly. The `targetClass` element is not required, because it uses generics to define the field:

```
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}
```

Collections of entity elements and relationships may be represented by `java.util.Map` collections. A `Map` consists of a key and a value.

When using `Map` elements or relationships, the following rules apply.

- The `Map` key or value may be a basic Java programming language type, an embeddable class, or an entity.
- When the `Map` value is an embeddable class or basic type, use the `@ElementCollection` annotation.
- When the `Map` value is an entity, use the `@OneToMany` or `@ManyToMany` annotation.
- Use the `Map` type on only one side of a bidirectional relationship.

If the key type of a `Map` is a Java programming language basic type, use the annotation `jakarta.persistence.MapKeyColumn` to set the column mapping for the key. By default, the `name` attribute of `@MapKeyColumn` is of the form *RELATIONSHIP-FIELD/PROPERTY-NAME*`_KEY`. For example, if the referencing relationship field name is `image`, the default `name` attribute is `IMAGE_KEY`.

If the key type of a `Map` is an entity, use the `jakarta.persistence.MapKeyJoinColumn` annotation. If the multiple columns are needed to set the mapping, use the annotation `jakarta.persistence.MapKeyJoinColumns` to include multiple `@MapKeyJoinColumn` annotations. If no `@MapKeyJoinColumn` is present, the mapping column name is by default set to *RELATIONSHIP-*

FIELD/PROPERTY-NAME`_KEY`. For example, if the relationship field name is `employee`, the default `name` attribute is `EMPLOYEE_KEY`.

If Java programming language generic types are not used in the relationship field or property, the key class must be explicitly set using the `jakarta.persistence.MapKeyClass` annotation.

If the `Map` key is the primary key or a persistent field or property of the entity that is the `Map` value, use the `jakarta.persistence.MapKey` annotation. The `@MapKeyClass` and `@MapKey` annotations cannot be used on the same field or property.

If the `Map` value is a Java programming language basic type or an embeddable class, it will be mapped as a collection table in the underlying database. If generic types are not used, the `@ElementCollection` annotation's `targetClass` attribute must be set to the type of the `Map` value.

If the `Map` value is an entity and part of a many-to-many or one-to-many unidirectional relationship, it will be mapped as a join table in the underlying database. A unidirectional one-to-many relationship that uses a `Map` may also be mapped using the `@JoinColumn` annotation.

If the entity is part of a one-to-many/many-to-one bidirectional relationship, it will be mapped in the table of the entity that represents the value of the `Map`. If generic types are not used, the `targetEntity` attribute of the `@OneToMany` and `@ManyToMany` annotations must be set to the type of the `Map` value.

Validating Persistent Fields and Properties

Jakarta Bean Validation provides a mechanism for validating application data. Bean Validation is integrated into the Jakarta EE containers, allowing the same validation logic to be used in any of the tiers of an enterprise application.

Bean Validation constraints may be applied to persistent entity classes, embeddable classes, and mapped superclasses. By default, the Persistence provider will automatically perform validation on entities with persistent fields or properties annotated with Bean Validation constraints immediately after the `PrePersist`, `PreUpdate`, and `PreRemove` lifecycle events.

Bean Validation constraints are annotations applied to the fields or properties of Java programming language classes. Bean Validation provides a set of constraints as well as an API for defining custom constraints. Custom constraints can be specific combinations of the default constraints, or new constraints that don't use the default constraints. Each constraint is associated with at least one validator class that validates the value of the constrained field or property. Custom constraint developers must also provide a validator class for the constraint.

Bean Validation constraints are applied to the persistent fields or properties of persistent classes. When adding Bean Validation constraints, use the same access strategy as the persistent class. That is, if the persistent class uses field access, apply the Bean Validation constraint annotations on the class's fields. If the class uses property access, apply the constraints on the getter methods.

Built In Jakarta Bean Validation Constraints lists Bean Validation's built-in constraints, defined in the `jakarta.validation.constraints` package.

All the built-in constraints listed in Built In Jakarta Bean Validation Constraints have a

corresponding annotation, `ConstraintName.List`, for grouping multiple constraints of the same type on the same field or property. For example, the following persistent field has two `@Pattern` constraints:

```
@Pattern.List({
    @Pattern(regexp="..."),
    @Pattern(regexp="...")
})
```

The following entity class, `Contact`, has Bean Validation constraints applied to its persistent fields:

```
@Entity
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
            + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
            + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"
            + "(?:[a-z0-9-]*[a-z0-9])?",
            message = "{invalid.email}")
    protected String email;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(jakarta.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

The `@NotNull` annotation on the `firstName` and `lastName` fields specifies that those fields are now required. If a new `Contact` instance is created where `firstName` or `lastName` have not been initialized, Bean Validation will throw a validation error. Similarly, if a previously created instance of `Contact` has been modified so that `firstName` or `lastName` are null, a validation error will be thrown.

The `email` field has a `@Pattern` constraint applied to it, with a complicated regular expression that matches most valid email addresses. If the value of `email` doesn't match this regular expression, a validation error will be thrown.

The `homePhone` and `mobilePhone` fields have the same `@Pattern` constraints. The regular expression

matches 10 digit telephone numbers in the United States and Canada of the form `(xxx) xxx-xxxx`.

The `birthday` field is annotated with the `@Past` constraint, which ensures that the value of `birthday` must be in the past.

Primary Keys in Entities

Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number. The unique identifier, or primary key, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the `jakarta.persistence.Id` annotation to denote the primary key property or field.

Composite primary keys are used when a primary key consists of more than one attribute, which corresponds to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the `jakarta.persistence.EmbeddedId` and `jakarta.persistence.IdClass` annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java language types:

- Java primitive types
- Java primitive wrapper types
- `java.lang.String`
- `java.util.Date` (the temporal type should be `DATE`)
- `java.sql.Date`
- `java.math.BigDecimal`
- `java.math.BigInteger`

Floating-point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.

A primary key class must meet these requirements.

- The access control modifier of the class must be `public`.
- The properties of the primary key class must be `public` or `protected` if property-based access is used.
- The class must have a public default constructor.
- The class must implement the `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.
- A composite primary key must be represented and mapped to multiple fields or properties of the entity class or must be represented and mapped as an embeddable class.
- If the class is mapped to multiple fields or properties of the entity class, the names and types of

the primary key fields or properties in the primary key class must match those of the entity class.

The following primary key class is a composite key, and the `customerOrder` and `itemId` fields together uniquely identify an entity:

```
public final class LineItemKey implements Serializable {
    private Integer customerOrder;
    private int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer order, int itemId) {
        this.setCustomerOrder(order);
        this.setItemId(itemId);
    }

    @Override
    public int hashCode() {
        return ((this.getCustomerOrder() == null
                ? 0 : this.getCustomerOrder().hashCode())
                ^ ((int) this.getItemId()));
    }

    @Override
    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder() == null : this.getCustomerOrder()
                .equals(other.getCustomerOrder()))
                && (this.getItemId() == other.getItemId()));
    }

    @Override
    public String toString() {
        return "" + getCustomerOrder() + "-" + getItemId();
    }
    /* Getters and setters */
}
```

Multiplicity in Entity Relationships

Multiplicities are of the following types.

- One-to-one: Each entity instance is related to a single instance of another entity. For example, to model a physical warehouse in which each storage bin contains a single widget, `StorageBin` and `Widget` would have a one-to-one relationship. One-to-one relationships use the `jakarta.persistence.OneToOne` annotation on the corresponding persistent property or field.
- One-to-many: An entity instance can be related to multiple instances of the other entities. A sales order, for example, can have multiple line items. In the order application, `CustomerOrder` would have a one-to-many relationship with `LineItem`. One-to-many relationships use the `jakarta.persistence.OneToMany` annotation on the corresponding persistent property or field.
- Many-to-one: Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship. In the example just mentioned, the relationship to `CustomerOrder` from the perspective of `LineItem` is many-to-one. Many-to-one relationships use the `jakarta.persistence.ManyToOne` annotation on the corresponding persistent property or field.
- Many-to-many: The entity instances can be related to multiple instances of each other. For example, each college course has many students, and every student may take several courses. Therefore, in an enrollment application, `Course` and `Student` would have a many-to-many relationship. Many-to-many relationships use the `jakarta.persistence.ManyToMany` annotation on the corresponding persistent property or field.

Direction in Entity Relationships

The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

Bidirectional Relationships

In a bidirectional relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object. If an entity has a related field, the entity is said to "know" about its related object. For example, if `CustomerOrder` knows what `LineItem` instances it has and if `LineItem` knows what `CustomerOrder` it belongs to, they have a bidirectional relationship.

Bidirectional relationships must follow these rules.

- The inverse side of a bidirectional relationship must refer to its owning side by using the `mappedBy` element of the `@OneToOne`, `@OneToMany`, or `@ManyToMany` annotation. The `mappedBy` element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the `mappedBy` element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships, either side may be the owning side.

Unidirectional Relationships

In a unidirectional relationship, only one entity has a relationship field or property that refers to the other. For example, `LineItem` would have a relationship field that identifies `Product`, but `Product` would not have a relationship field or property for `LineItem`. In other words, `LineItem` knows about `Product`, but `Product` doesn't know which `LineItem` instances refer to it.

Queries and Relationship Direction

Jakarta Persistence query language and Criteria API queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from `LineItem` to `Product` but cannot navigate in the opposite direction. For `CustomerOrder` and `LineItem`, a query could navigate in both directions because these two entities have a bidirectional relationship.

Cascade Operations and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

The `jakarta.persistence.CascadeType` enumerated type defines the cascade operations that are applied in the `cascade` element of the relationship annotations. Cascade Operations for Entities lists the cascade operations for entities.

Cascade Operations for Entities

Cascade Operation	Description
`ALL`	All cascade operations will be applied to the parent entity's related entity. `All` is equivalent to specifying `cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}`
`DETACH`	If the parent entity is detached from the persistence context, the related entity will also be detached.
`MERGE`	If the parent entity is merged into the persistence context, the related entity will also be merged.
`PERSIST`	If the parent entity is persisted into the persistence context, the related entity will also be persisted.
`REFRESH`	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
`REMOVE`	If the parent entity is removed from the current persistence context, the related entity will also be removed.

Cascade delete relationships are specified using the `cascade=REMOVE` element specification for `@OneToOne` and `@OneToMany` relationships. For example:

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
```

```
public Set<CustomerOrder> getOrders() { return orders; }
```

Orphan Removal in Relationships

When a target entity in a one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity. Such target entities are considered "orphans," and the `orphanRemoval` attribute can be used to specify that orphaned entities should be removed. For example, if an order has many line items and one of them is removed from the order, the removed line item is considered an orphan. If `orphanRemoval` is set to `true`, the line item entity will be deleted when the line item is removed from the order.

The `orphanRemoval` attribute in `@OneToMany` and `@oneToOne` takes a Boolean value and is by default false.

The following example will cascade the remove operation to the orphaned `order` entity when the `customer` entity is deleted:

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
public List<CustomerOrder> getOrders() { ... }
```

Embeddable Classes in Entities

Embeddable classes are used to represent the state of an entity but don't have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it. Embeddable classes exist only as the state of another entity. An entity may have single-valued or collection-valued embeddable class attributes.

Embeddable classes have the same rules as entity classes but are annotated with the `jakarta.persistence.Embeddable` annotation instead of `@Entity`.

The following embeddable class, `ZipCode`, has the fields `zip` and `plusFour`:

```
@Embeddable
public class ZipCode {
    String zip;
    String plusFour;
    ...
}
```

This embeddable class is used by the `Address` entity:

```
@Entity
public class Address {
    @Id
    protected long id
    String street1;
    String street2;
```

```
    String city;
    String province;
    @Embedded
    ZipCode zipCode;
    String country;
    ...
}
```

Entities that own embeddable classes as part of their persistent state may annotate the field or property with the `jakarta.persistence.Embedded` annotation but are not required to do so.

Embeddable classes may themselves use other embeddable classes to represent their state. They may also contain collections of basic Java programming language types or other embeddable classes. Embeddable classes may also contain relationships to other entities or collections of entities. If the embeddable class has such a relationship, the relationship is from the target entity or collection of entities to the entity that owns the embeddable class.

Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

The `roster` example application demonstrates entity inheritance, as described in Entity Inheritance in the roster Application.

Abstract Entities

An abstract class may be declared an entity by decorating the class with `@Entity`. Abstract entities are like concrete entities but cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity:

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the `@Entity` annotation and is not mapped as an entity by the Jakarta Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the annotation `jakarta.persistence.MappedSuperclass`:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

Mapped superclasses cannot be queried and cannot be used in `EntityManager` or `Query` operations. You must use entity subclasses of the mapped superclass in `EntityManager` or `Query` operations. Mapped superclasses can't be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the preceding code sample, the underlying tables would be `FULLTIMEEMPLOYEE` and `PARTTIMEEMPLOYEE`, but there is no `EMPLOYEE` table.

Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is nonpersistent, and any state inherited from the non-entity superclass by an entity class is nonpersistent. Non-entity superclasses may not be used in `EntityManager` or `Query` operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

Entity Inheritance Mapping Strategies

You can configure how the Jakarta Persistence provider maps inherited entities to the underlying datastore by decorating the root class of the hierarchy with the annotation `jakarta.persistence.Inheritance`. The following mapping strategies are used to map the entity data

to the underlying database:

- A single table per class hierarchy
- A table per concrete entity class
- A "join" strategy, whereby fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The strategy is configured by setting the `strategy` element of `@Inheritance` to one of the options defined in the `jakarta.persistence.InheritanceType` enumerated type:

```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

The default strategy, `InheritanceType.SINGLE_TABLE`, is used if the `@Inheritance` annotation is not specified on the root class of the entity hierarchy.

The Single Table per Class Hierarchy Strategy

With this strategy, which corresponds to the default `InheritanceType.SINGLE_TABLE`, all classes in the hierarchy are mapped to a single table in the database. This table has a discriminator column containing a value that identifies the subclass to which the instance represented by the row belongs.

The discriminator column, whose elements are shown in @DiscriminatorColumn Elements, can be specified by using the `jakarta.persistence.DiscriminatorColumn` annotation on the root of the entity class hierarchy.

@DiscriminatorColumn Elements

Type	Name	Description
`String`	`name`	The name of the column to be used as the discriminator column. The default is `DTYPE`. This element is optional.
`DiscriminatorType`	`discriminatorType`	The type of the column to be used as a discriminator column. The default is `DiscriminatorType.STRING`. This element is optional.
`String`	`columnDefinition`	The SQL fragment to use when creating the discriminator column. The default is generated by the Persistence provider and is implementation-specific. This element is optional.
`String`	`length`	The column length for `String`-based discriminator types. This element is ignored for non-`String` discriminator types. The default is 31. This element is optional.

The `jakarta.persistence.DiscriminatorType` enumerated type is used to set the type of the

discriminator column in the database by setting the `discriminatorType` element of `@DiscriminatorColumn` to one of the defined types. `DiscriminatorType` is defined as follows:

```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

If `@DiscriminatorColumn` is not specified on the root of the entity hierarchy and a discriminator column is required, the Persistence provider assumes a default column name of `DTYPE` and column type of `DiscriminatorType.STRING`.

The `jakarta.persistence.DiscriminatorValue` annotation may be used to set the value entered into the discriminator column for each entity in a class hierarchy. You may decorate only concrete entity classes with `@DiscriminatorValue`.

If `@DiscriminatorValue` is not specified on an entity in a class hierarchy that uses a discriminator column, the Persistence provider will provide a default, implementation-specific value. If the `discriminatorType` element of `@DiscriminatorColumn` is `DiscriminatorType.STRING`, the default value is the name of the entity.

This strategy provides good support for polymorphic relationships between entities and queries that cover the entire entity class hierarchy. However, this strategy requires the columns that contain the state of subclasses to be nullable.

The Table per Concrete Class Strategy

In this strategy, which corresponds to `InheritanceType.TABLE_PER_CLASS`, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class's table in the database.

This strategy provides poor support for polymorphic relationships and usually requires either SQL `UNION` queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy.

Support for this strategy is optional and may not be supported by all Jakarta Persistence providers. The default Jakarta Persistence provider in GlassFish Server does not support this strategy.

The Joined Subclass Strategy

In this strategy, which corresponds to `InheritanceType.JOINED`, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that contains only those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

This strategy provides good support for polymorphic relationships but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor

performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

Some Jakarta Persistence providers, including the default provider in GlassFish Server, require a discriminator column that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure that the database table is set up correctly for the discriminator column defaults, or use the `@DiscriminatorColumn` annotation to match your database schema. For information on discriminator columns, see The Single Table per Class Hierarchy Strategy.

Managing Entities

Entities are managed by the entity manager, which is represented by `jakarta.persistence.EntityManager` instances. Each `EntityManager` instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store. A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The `EntityManager` interface defines the methods that are used to interact with the persistence context.

The EntityManager Interface

The `EntityManager` API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

Container-Managed Entity Managers

With a container-managed entity manager, an `EntityManager` instance's persistence context is automatically propagated by the container to all application components that use the `EntityManager` instance within a single Jakarta transaction.

Jakarta transactions usually involve calls across application components. To complete a Jakarta transaction, these components usually need access to a single persistence context. This occurs when an `EntityManager` is injected into the application components by means of the `jakarta.persistence.PersistenceContext` annotation. The persistence context is automatically propagated with the current Jakarta transaction, and `EntityManager` references that are mapped to the same persistence unit provide access to the persistence context within that transaction. By automatically propagating the persistence context, application components don't need to pass references to `EntityManager` instances to each other in order to make changes within a single transaction. The Jakarta EE container manages the lifecycle of container-managed entity managers.

To obtain an `EntityManager` instance, inject the entity manager into the application component:

```
@PersistenceContext
EntityManager em;
```

Application-Managed Entity Managers

With an application-managed entity manager, on the other hand, the persistence context is not propagated to application components, and the lifecycle of `EntityManager` instances is managed by

the application.

Application-managed entity managers are used when applications need to access a persistence context that is not propagated with the Jakarta transaction across `EntityManager` instances in a particular persistence unit. In this case, each `EntityManager` creates a new, isolated persistence context. The `EntityManager` and its associated persistence context are created and destroyed explicitly by the application. They are also used when directly injecting `EntityManager` instances can't be done because `EntityManager` instances are not thread-safe. `EntityManagerFactory` instances are thread-safe.

Applications create `EntityManager` instances in this case by using the `createEntityManager` method of `jakarta.persistence.EntityManagerFactory`.

To obtain an `EntityManager` instance, you first must obtain an `EntityManagerFactory` instance by injecting it into the application component by means of the `jakarta.persistence.PersistenceUnit` annotation:

```
@PersistenceUnit
EntityManagerFactory emf;
```

Then obtain an `EntityManager` from the `EntityManagerFactory` instance:

```
EntityManager em = emf.createEntityManager();
```

Application-managed entity managers don't automatically propagate the Jakarta transaction context. Such applications need to manually gain access to the Jakarta transaction manager and add transaction demarcation information when performing entity operations. The `jakarta.transaction.UserTransaction` interface defines methods to begin, commit, and roll back transactions. Inject an instance of `UserTransaction` by creating an instance variable annotated with `@Resource`:

```
@Resource
UserTransaction utx;
```

To begin a transaction, call the `UserTransaction.begin` method. When all the entity operations are complete, call the `UserTransaction.commit` method to commit the transaction. The `UserTransaction.rollback` method is used to roll back the current transaction.

The following example shows how to manage transactions in an application that uses an application-managed entity manager:

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
```

```
...
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
} catch (Exception e) {
    utx.rollback();
}
```

Finding Entities Using the EntityManager

The `EntityManager.find` method is used to look up entities in the data store by the entity's primary key:

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, CustomerOrder newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

Managing an Entity Instance's Lifecycle

You manage entity instances by invoking operations on the entity by means of an `EntityManager` instance. Entity instances are in one of four states: new, managed, detached, or removed.

- New entity instances have no persistent identity and are not yet associated with a persistence context.
- Managed entity instances have a persistent identity and are associated with a persistence context.
- Detached entity instances have a persistent identity and are not currently associated with a persistence context.
- Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

Persisting Entity Instances

New entity instances become managed and persistent either by invoking the `persist` method or by a cascading `persist` operation invoked from related entities that have the `cascade=PERSIST` or `cascade=ALL` elements set in the relationship annotation. This means that the entity's data is stored to the database when the transaction associated with the `persist` operation is completed. If the entity is already managed, the `persist` operation is ignored, although the `persist` operation will cascade to related entities that have the `cascade` element set to `PERSIST` or `ALL` in the relationship

annotation. If `persist` is called on a removed entity instance, the entity becomes managed. If the entity is detached, either `persist` will throw an `IllegalArgumentException`, or the transaction commit will fail. The following method performs a `persist` operation:

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(CustomerOrder order, Product product,
        int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

The `persist` operation is propagated to all entities related to the calling entity that have the `cascade` element set to `ALL` or `PERSIST` in the relationship annotation:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Removing Entity Instances

Managed entity instances are removed by invoking the `remove` method or by a cascading `remove` operation invoked from related entities that have the `cascade=REMOVE` or `cascade=ALL` elements set in the relationship annotation. If the `remove` method is invoked on a new entity, the `remove` operation is ignored, although `remove` will cascade to related entities that have the `cascade` element set to `REMOVE` or `ALL` in the relationship annotation. If `remove` is invoked on a detached entity, either `remove` will throw an `IllegalArgumentException`, or the transaction commit will fail. If invoked on an already removed entity, `remove` will be ignored. The entity's data will be removed from the data store when the transaction is completed or as a result of the `flush` operation.

In the following example, all `LineItem` entities associated with the order are also removed, as `CustomerOrder.getLineItems` has `cascade=ALL` set in the relationship annotation:

```
public void removeOrder(Integer orderId) {
    try {
        CustomerOrder order = em.find(CustomerOrder.class, orderId);
        em.remove(order);
    }...
}
```

Synchronizing Entity Data to the Database

The state of persistent entities is synchronized to the database when the transaction with which the

entity is associated commits. If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted, based on the owning side of the relationship.

To force synchronization of the managed entity to the data store, invoke the `flush` method of the `EntityManager` instance. If the entity is related to another entity and the relationship annotation has the `cascade` element set to `PERSIST` or `ALL`, the related entity's data will be synchronized with the data store when `flush` is called.

If the entity is removed, calling `flush` will remove the entity data from the data store.

Persistence Units

A persistence unit defines a set of all entity classes that are managed by `EntityManager` instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the `persistence.xml` configuration file. The following is an example `persistence.xml` file:

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.CustomerOrder</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

This file defines a persistence unit named `OrderManagement`, which uses a Jakarta Transactions aware data source, `jdbc/MyOrderDB`. The `jar-file` and `class` elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasses. The `jar-file` element specifies JAR files that are visible to the packaged persistence unit that contain managed persistence classes, whereas the `class` element explicitly names managed persistence classes.

The `jta-data-source` (for Jakarta Transactions aware data sources) and `non-jta-data-source` (for non Jakarta Transactions aware data sources) elements specify the global JNDI name of the data source to be used by the container.

The JAR file or directory whose `META-INF` directory contains `persistence.xml` is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit's root. Each persistence unit must be identified with a name that is unique to the persistence unit's scope.

Persistent units can be packaged as part of a WAR or enterprise bean JAR file or can be packaged as a JAR file that can then be included in an WAR or EAR file.

- If you package the persistent unit as a set of classes in an enterprise bean JAR file, `persistence.xml` should be put in the enterprise bean JAR's `META-INF` directory.

- If you package the persistence unit as a set of classes in a WAR file, `persistence.xml` should be located in the WAR file's `WEB-INF/classes/META-INF` directory.
- If you package the persistence unit in a JAR file that will be included in a WAR or EAR file, the JAR file should be located in either
  - The `WEB-INF/lib` directory of a WAR
  - Or the EAR file's library directory



In the Java Persistence API 1.0, JAR files could be located at the root of an EAR file as the root of the persistence unit. This is no longer supported. Portable applications should use the EAR file's library directory as the root of the persistence unit.

Querying Entities

Jakarta Persistence provides the following methods for querying entities.

- The Jakarta Persistence query language (JPQL) is a simple, string-based language similar to SQL used to query entities and their relationships. See [persist:persistence-querylanguage::persistence-querylanguage:::_the_jakarta_persistence_query_language] for more information.
- The Criteria API is used to create typesafe queries using Java programming language APIs to query for entities and their relationships. See [persist:persistence-criteria::persistence-criteria:::_using_the_criteria_api_to_create_queries] for more information.

Both JPQL and the Criteria API have advantages and disadvantages.

Just a few lines long, JPQL queries are typically more concise and more readable than Criteria queries. Developers familiar with SQL will find it easy to learn the syntax of JPQL. JPQL named queries can be defined in the entity class using a Java programming language annotation or in the application's deployment descriptor. JPQL queries are not typesafe, however, and require a cast when retrieving the query result from the entity manager. This means that type-casting errors may not be caught at compile time. JPQL queries don't support open-ended parameters.

Criteria queries allow you to define the query in the business tier of the application. Although this is also possible using JPQL dynamic queries, Criteria queries provide better performance because JPQL dynamic queries must be parsed each time they are called. Criteria queries are typesafe and therefore don't require casting, as JPQL queries do. The Criteria API is just another Java programming language API and doesn't require developers to learn the syntax of another query language. Criteria queries are typically more verbose than JPQL queries and require the developer to create several objects and perform operations on those objects before submitting the query to the entity manager.

Database Schema Creation

The persistence provider can be configured to automatically create the database tables, load data into the tables, and remove the tables during application deployment using standard properties in the application's deployment descriptor. These tasks are typically used during the development

phase of a release, not against a production database.

The following is an example of a `persistence.xml` deployment descriptor that specifies that the provider should drop all database artifacts using a provided script, create the artifacts with a provided script, and load data from a provided script when the application is deployed:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.0" xmlns="https://jakarta.ee/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
 https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="examplePU" transaction-type="JTA">
    <jta-data-source>java:global/ExampleDataSource</jta-data-source>
    <properties>
        <property name="jakarta.persistence.schema-generation.database.action"
                  value="drop-and-create"/>
        <property name="jakarta.persistence.schema-generation.create-source"
                  value="script"/>
        <property name="jakarta.persistence.schema-generation.create-script-source"
                  value="META-INF/sql/create.sql" />
        <property name="jakarta.persistence.sql-load-script-source"
                  value="META-INF/sql/data.sql" />
        <property name="jakarta.persistence.schema-generation.drop-source"
                  value="script" />
        <property name="jakarta.persistence.schema-generation.drop-script-source"
                  value="META-INF/sql/drop.sql" />
    </properties>
  </persistence-unit>
</persistence>
```

Configuring an Application to Create or Drop Database Tables

The `jakarta.persistence.schema-generation.database.action` property is used to specify the action taken by the persistence provider when an application is deployed. If the property is not set, the persistence provider will not create or drop any database artifacts.

Schema Creation Actions

Setting	Description
`none`	No schema creation or deletion will take place.
`create`	The provider will create the database artifacts on application deployment. The artifacts will remain unchanged after application redeployment.
`drop-and-create`	Any artifacts in the database will be deleted, and the provider will create the database artifacts on deployment.
`drop`	Any artifacts in the database will be deleted on application deployment.

In this example, the persistence provider will delete any remaining database artifacts and then create the artifacts when the application is deployed:

```
<property name="jakarta.persistence.schema-generation.database.action"
          value="drop-and-create"/>
```

By default, the object/relational metadata in the persistence unit is used to create the database artifacts. You may also supply scripts used by the provider to create and delete the database artifacts. The `jakarta.persistence.schema-generation.create-source` and `jakarta.persistence.schema-generation.drop-source` properties control how the provider will create or delete the database artifacts.

Settings for Create and Delete Source Properties

Setting	Description
`metadata`	Use the object/relational metadata in the application to create or delete the database artifacts.
`script`	Use a provided script for creating or deleting the database artifacts.
`metadata-then-script`	Use a combination of object/relational metadata, then a user-provided script to create or delete the database artifacts.
`script-then-metadata`	Use a combination of a user-provided script, then the object/relational metadata to create and delete the database artifacts.

In this example, the persistence provider will use a script packaged within the application to create the database artifacts:

```
<property name="jakarta.persistence.schema-generation.create-source"
          value="script"/>
```

If you specify a script in `create-source` or `drop-source`, specify the location of the script using the `jakarta.persistence.schema-generation.create-script-source` or `jakarta.persistence.schema-generation.drop-script-source` property. The location of the script is relative to the root of the persistence unit:

```
<property name="jakarta.persistence.schema-generation.create-script-source"
          value="META-INF/sql/create.sql" />
```

In the above example, the `create-script-source` is set to a SQL file called `create.sql` in the `META-INF/sql` directory relative to root of the persistence unit.

Loading Data Using SQL Scripts

If you want to populate the database tables with data before the application loads, specify the location of a load script in the `jakarta.persistence.sql-load-script-source property`. The location specified in this property is relative to the root of the persistence unit.

In this example, the load script is a file called `data.sql` in the `META-INF/sql` directory relative to the root of the persistence unit:

```
<property name="jakarta.persistence.sql-load-script-source"
          value="META-INF/sql/data.sql" />
```

Further Information about Persistence

For more information about Jakarta Persistence, see

- Jakarta Persistence 3.0 API specification:
  https://jakarta.ee/specifications/persistence/3.0/
- EclipseLink, the Jakarta Persistence implementation in GlassFish Server:
  https://www.eclipse.org/eclipselink/
- EclipseLink wiki documentation:
  https://wiki.eclipse.org/EclipseLink

Running the Persistence Examples



This section was last updated for Jakarta EE Version 9.1

This chapter explains how to use Jakarta Persistence. The material here focuses on the source code and settings of three examples.

Overview of the Persistence Examples

The first example, `order`, is an application that uses a stateful session bean to manage entities related to an ordering system. The second example, `roster`, is an application that manages a community sports system. The third example, `address-book`, is a web application that stores contact data. This chapter assumes that you are familiar with the concepts detailed in [persist:persistence-intro::persistence-intro:::_introduction_to_jakarta_persistence].

The order Application

The `order` application is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts. The application has entities that represent parts, vendors, orders, and line items. These entities are accessed using a stateful session bean that holds the business logic of the application. A simple singleton session bean creates the initial entities on application deployment. A Facelets web application manipulates the data and displays data from the catalog.

The information contained in an order can be divided into elements. What is the order number?

What parts are included in the order? What parts make up that part? Who makes the part? What are the specifications for the part? Are there any schematics for the part? The `order` application is a simplified version of an ordering system that has all these elements.

The `order` application consists of a single WAR module that includes the enterprise bean classes, the entities, the support classes, and the Facelets XHTML and class files.

The database schema in the Derby database for `order` is shown in Figure 18, "Database Schema for the order Application".

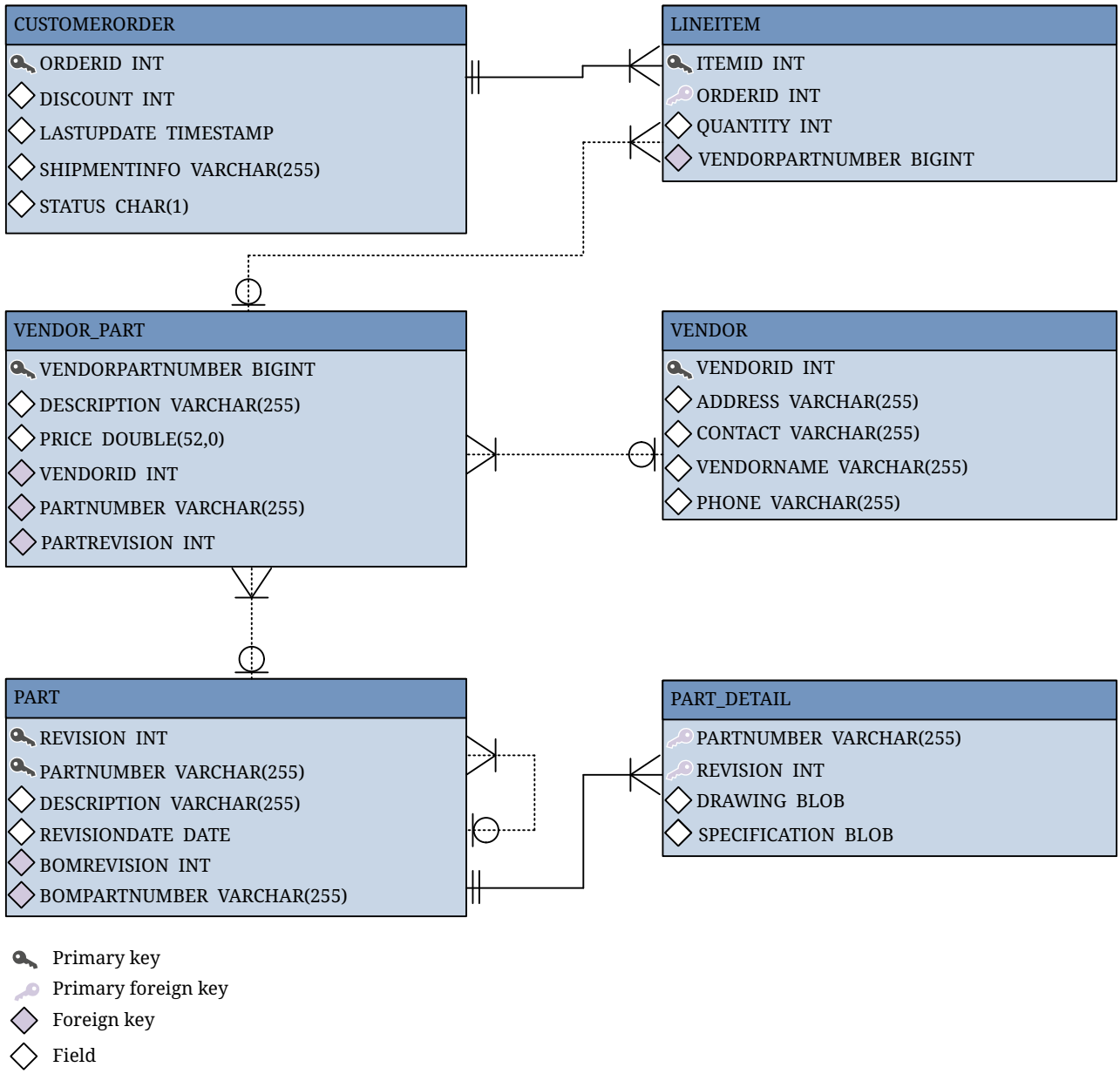


Figure 18. Database Schema for the order Application



In this diagram, for simplicity, the `PERSISTENCE_ORDER_` prefix is omitted from the table names.

Entity Relationships in the order Application

The `order` application demonstrates several types of entity relationships: self-referential, one-to-

one, one-to-many, many-to-one, and unidirectional relationships.

Self-Referential Relationships

A self-referential relationship occurs between relationship fields in the same entity. `Part` has a field, `bomPart`, which has a one-to-many relationship with the field `parts`, which is also in `Part`. That is, a part can be made up of many parts, and each of those parts has exactly one bill-of-material part.

The primary key for `Part` is a compound primary key, a combination of the `partNumber` and `revision` fields. This key is mapped to the `PARTNUMBER` and `REVISION` columns in the `PERSISTENCE_ORDER_PART` table:

```
...
@ManyToOne
@JoinColumns({
    @JoinColumn(name="BOMPARTNUMBER", referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="BOMREVISION", referencedColumnName="REVISION")
})
public Part getBomPart() {
    return bomPart;
}
...
@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts() {
    return parts;
}
...
```

One-to-One Relationships

`Part` has a field, `vendorPart`, that has a one-to-one relationship with `VendorPart`'s `part` field. That is, each part has exactly one vendor part, and vice versa.

Here is the relationship mapping in `Part`:

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Here is the relationship mapping in `VendorPart`:

```
@OneToOne
@JoinColumns({
    @JoinColumn(name="PARTNUMBER", referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION", referencedColumnName="REVISION")
})
public Part getPart() {
```

```
    return part;
}
```

Note that, because `Part` uses a compound primary key, the `@JoinColumns` annotation is used to map the columns in the `PERSISTENCE_ORDER_VENDOR_PART` table to the columns in `PERSISTENCE_ORDER_PART`. The `PERSISTENCE_ORDER_VENDOR_PART` table's `PARTREVISION` column refers to `PERSISTENCE_ORDER_PART`'s `REVISION` column.

One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys

`CustomerOrder` has a field, `lineItems`, that has a one-to-many relationship with `LineItem`'s field `customerOrder`. That is, each order has one or more line item.

`LineItem` uses a compound primary key that is made up of the `orderId` and `itemId` fields. This compound primary key maps to the `ORDERID` and `ITEMID` columns in the `PERSISTENCE_ORDER_LINEITEM` table. `ORDERID` is a foreign key to the `ORDERID` column in the `PERSISTENCE_ORDER_CUSTOMERORDER` table. This means that the `ORDERID` column is mapped twice: once as a primary key field, `orderId`; and again as a relationship field, `order`.

Here is the relationship mapping in `CustomerOrder`:

```
@OneToMany(cascade=ALL, mappedBy="customerOrder")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Here is the relationship mapping in `LineItem`:

```
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

Unidirectional Relationships

`LineItem` has a field, `vendorPart`, that has a unidirectional many-to-one relationship with `VendorPart`. That is, there is no field in the target entity in this relationship:

```
@JoinColumn(name="VENDORPARTNUMBER")
@ManyToOne
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Primary Keys in the order Application

The `order` application uses several types of primary keys: single-valued primary keys, generated primary keys, and compound primary keys.

Generated Primary Keys

`VendorPart` uses a generated primary key value. That is, the application does not assign primary key values for the entities but instead relies on the persistence provider to generate the primary key values. The `@GeneratedValue` annotation is used to specify that an entity will use a generated primary key.

In `VendorPart`, the following code specifies the settings for generating primary key values:

```
@TableGenerator(
    name="vendorPartGen",
    table="PERSISTENCE_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY",
    valueColumnName="GEN_VALUE",
    pkColumnValue="VENDOR_PART_ID",
    allocationSize=10)
@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="vendorPartGen")
public Long getVendorPartNumber() {
    return vendorPartNumber;
}
```

The `@TableGenerator` annotation is used in conjunction with `@GeneratedValue`'s `strategy=TABLE` element. That is, the strategy used to generate the primary keys is to use a table in the database. The `@TableGenerator` annotation is used to configure the settings for the generator table. The name element sets the name of the generator, which is `vendorPartGen` in `VendorPart`.

The `PERSISTENCE_ORDER_SEQUENCE_GENERATOR` table, whose two columns are `GEN_KEY` and `GEN_VALUE`, will store the generated primary key values. This table could be used to generate other entities' primary keys, so the `pkColumnValue` element is set to `VENDOR_PART_ID` to distinguish this entity's generated primary keys from other entities' generated primary keys. The `allocationSize` element specifies the amount to increment when allocating primary key values. In this case, each `VendorPart`'s primary key will increment by 10.

The primary key field `vendorPartNumber` is of type `Long`, as the generated primary key's field must be an integral type.

Compound Primary Keys

A compound primary key is made up of multiple fields and follows the requirements described in Primary Keys in Entities. To use a compound primary key, you must create a wrapper class.

In `order`, two entities use compound primary keys: `Part` and `LineItem`.

- `Part` uses the `PartKey` wrapper class. `Part`'s primary key is a combination of the part number and

the revision number. `PartKey` encapsulates this primary key.

- `LineItem` uses the `LineItemKey` class. `LineItem`'s primary key is a combination of the order number and the item number. `LineItemKey` encapsulates this primary key.

This is the `LineItemKey` compound primary key wrapper class:

```
package ee.jakarta.tutorial.order.entity;

import java.io.Serializable;

public final class LineItemKey implements Serializable {

    private Integer customerOrder;
    private int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer order, int itemId) {
        this.setCustomerOrder(order);
        this.setItemId(itemId);
    }

    @Override
    public int hashCode() {
        return ((this.getCustomerOrder() == null
                ? 0 : this.getCustomerOrder().hashCode())
                ^ ((int) this.getItemId()));
    }

    @Override
    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder == null : this.getOrderId()
                .equals(other.getCustomerOrder()))
                && (this.getItemId == oother.getItemId()));
    }

    @Override
    public String toString() {
        return "" + getCustomerOrder() + "-" + getItemId();
    }

    public Integer getCustomerOrder() {
```

```
        return customerOrder;
    }

    public void setCustomerOrder(Integer order) {
        this.customerOrder = order;
    }

    public int getItemId() {
        return itemId;
    }

    public void setItemId(int itemId) {
        this.itemId = itemId;
    }
}
```

The `@IdClass` annotation is used to specify the primary key class in the entity class. In `LineItem`, `@IdClass` is used as follows:

```
@IdClass(LineItemKey.class)
@Entity
...
public class LineItem implements Serializable {
    ...
}
```

The two fields in `LineItem` are tagged with the `@Id` annotation to mark those fields as part of the compound primary key:

```
@Id
public int getItemId() {
    return itemId;
}
...
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

For `customerOrder`, you also use the `@JoinColumn` annotation to specify the column name in the table and that this column is an overlapping foreign key pointing at the `PERSISTENCE_ORDER_CUSTOMERORDER` table's `ORDERID` column (see One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys). That is, `customerOrder` will be set by the `CustomerOrder` entity.

In `LineItem`'s constructor, the line item number (`LineItem.itemId`) is set using the `CustomerOrder.getNextId` method:

```
public LineItem(CustomerOrder order, int quantity, VendorPart vendorPart) {
    this.customerOrder = order;
    this.itemId = order.getNextId();
    this.quantity = quantity;
    this.vendorPart = vendorPart;
}
```

`CustomerOrder.getNextId` counts the number of current line items, adds 1, and returns that number:

```
@Transient
public int getNextId() {
    return this.lineItems.size() + 1;
}
```

`Part` requires the `@Column` annotation on the two fields that comprise `Part`'s compound primary key, because `Part`'s compound primary key is an overlapping primary key/foreign key:

```
@IdClass(PartKey.class)
@Entity
...
public class Part implements Serializable {
    ...
    @Id
    @Column(nullable=false)
    public String getPartNumber() {
        return partNumber;
    }
    ...
    @Id
    @Column(nullable=false)
    public int getRevision() {
        return revision;
    }
    ...
}
```

Entity Mapped to More Than One Database Table

`Part`'s fields map to more than one database table: `PERSISTENCE_ORDER_PART` and `PERSISTENCE_ORDER_PART_DETAIL`. The `PERSISTENCE_ORDER_PART_DETAIL` table holds the specification and schematics for the part. The `@SecondaryTable` annotation is used to specify the secondary table:

```
...
@Entity
@Table(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(name="PERSISTENCE_ORDER_PART_DETAIL", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="PARTNUMBER",
```

```
        referencedColumnName="PARTNUMBER"),
    @PrimaryKeyJoinColumn(name="REVISION",
        referencedColumnName="REVISION")
})
public class Part implements Serializable {
    ...
}
```

PERSISTENCE_ORDER_PART_DETAIL and PERSISTENCE_ORDER_PART share the same primary key values. The pkJoinColumns element of @SecondaryTable is used to specify that PERSISTENCE_ORDER_PART_DETAIL's primary key columns are foreign keys to PERSISTENCE_ORDER_PART. The @PrimaryKeyJoinColumn annotation sets the primary key column names and specifies which column in the primary table the column refers to. In this case, the primary key column names for both PERSISTENCE_ORDER_PART_DETAIL and PERSISTENCE_ORDER_PART are the same: PARTNUMBER and REVISION, respectively.

Cascade Operations in the order Application

Entities that have relationships to other entities often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, then the line item also should be deleted. This is called a cascade delete relationship.

In order, there are two cascade delete dependencies in the entity relationships. If the CustomerOrder to which a LineItem is related is deleted, the LineItem also should be deleted. If the Vendor to which a VendorPart is related is deleted, the VendorPart also should be deleted.

You specify the cascade operations for entity relationships by setting the cascade element in the inverse (nonowning) side of the relationship. The cascade element is set to ALL in the case of CustomerOrder.lineItems. This means that all persistence operations (deletes, updates, and so on) are cascaded from orders to line items.

Here is the relationship mapping in CustomerOrder:

```
@OneToMany(cascade=ALL, mappedBy="customerOrder")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Here is the relationship mapping in LineItem:

```
@Id
@ManyToOne
@JoinColumn(name="ORDERID")
public CustomerOrder getCustomerOrder() {
    return customerOrder;
}
```

BLOB and CLOB Database Types in the order Application

The PARTDETAIL table in the database has a column, DRAWING, of type BLOB. BLOB stands for binary large objects, which are used for storing binary data, such as an image. The DRAWING column is mapped to the field Part.drawing of type java.io.Serializable. The @Lob annotation is used to denote that the field is a large object:

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}
```

PERSISTENCE_ORDER_PART_DETAIL also has a column, SPECIFICATION, of type CLOB. CLOB stands for character large objects, which are used to store string data too large to be stored in a VARCHAR column. SPECIFICATION is mapped to the field Part.specification of type java.lang.String. The @Lob annotation is also used here to denote that the field is a large object:

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}
```

Both of these fields use the @Column annotation and set the table element to the secondary table.

Temporal Types in the order Application

The CustomerOrder.lastUpdate persistent property, which is of type java.util.Date, is mapped to the PERSISTENCE_ORDER_CUSTOMERORDER.LASTUPDATE database field, which is of the SQL type TIMESTAMP. To ensure the proper mapping between these types, you must use the @Temporal annotation with the proper temporal type specified in @Temporal's element. @Temporal's elements are of type jakarta.persistence.TemporalType. The possible values are

- DATE, which maps to java.sql.Date
- TIME, which maps to java.sql.Time
- TIMESTAMP, which maps to java.sql.Timestamp

Here is the relevant section of CustomerOrder:

```
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

Managing the order Application's Entities

The `RequestBean` stateful session bean contains the business logic and manages the entities of `order`. `RequestBean` uses the `@PersistenceContext` annotation to retrieve an entity manager instance, which is used to manage `order`'s entities in `RequestBean`'s business methods:

```
@PersistenceContext
private EntityManager em;
```

This `EntityManager` instance is a container-managed entity manager, so the container takes care of all the transactions involved in managing `order`'s entities.

Creating Entities

The `RequestBean.createPart` business method creates a new `Part` entity. The `EntityManager.persist` method is used to persist the newly created entity to the database:

```
Part part = new Part(partNumber,
        revision,
        description,
        revisionDate,
        specification,
        drawing);
em.persist(part);
```

The `ConfigBean` singleton session bean is used to initialize the data in `order`. `ConfigBean` is annotated with `@Startup`, which indicates that the enterprise bean container should create `ConfigBean` when `order` is deployed. The `createData` method is annotated with `@PostConstruct` and creates the initial entities used by `order` by calling `RequestBean`'s business methods.

Finding Entities

The `RequestBean.getOrderPrice` business method returns the price of a given order based on the `orderId`. The `EntityManager.find` method is used to retrieve the entity from the database:

```
CustomerOrder order = em.find(CustomerOrder.class, orderId);
```

The first argument of `EntityManager.find` is the entity class, and the second is the primary key.

Setting Entity Relationships

The `RequestBean.createVendorPart` business method creates a `VendorPart` associated with a particular `Vendor`. The `EntityManager.persist` method is used to persist the newly created `VendorPart` entity to the database, and the `VendorPart.setVendor` and `Vendor.setVendorPart` methods are used to associate the `VendorPart` with the `Vendor`:

```
PartKey pkey = new PartKey();
```

```
pkey.setPartNumber(partNumber);
pkey.setRevision(revision);

Part part = em.find(Part.class, pkey);

VendorPart vendorPart = new VendorPart(description, price, part);
em.persist(vendorPart);

Vendor vendor = em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

Using Queries

The `RequestBean.adjustOrderDiscount` business method updates the discount applied to all orders. This method uses the `findAllOrders` named query, defined in `CustomerOrder`:

```
@NamedQuery(
    name="findAllOrders",
    query="SELECT co FROM CustomerOrder co " +
          "ORDER BY co.orderId"
)
```

The `EntityManager.createNamedQuery` method is used to run the query. Because the query returns a `List` of all the orders, the `Query.getResultList` method is used:

```
List orders = em.createNamedQuery(
        "findAllOrders")
        .getResultList();
```

The `RequestBean.getTotalPricePerVendor` business method returns the total price of all the parts for a particular vendor. This method uses a named parameter, `id`, defined in the named query `findTotalVendorPartPricePerVendor` defined in `VendorPart`:

```
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " +
    "FROM VendorPart vp " +
    "WHERE vp.vendor.vendorId = :id"
)
```

When running the query, the `Query.setParameter` method is used to set the named parameter `id` to the value of `vendorId`, the parameter to `RequestBean.getTotalPricePerVendor`:

```
return (Double) em.createNamedQuery(
    "findTotalVendorPartPricePerVendor")
```

```
    .setParameter("id", vendorId)
    .getSingleResult();
```

The `Query.getSingleResult` method is used for this query because the query returns a single value.

Removing Entities

The `RequestBean.removeOrder` business method deletes a given order from the database. This method uses the `EntityManager.remove` method to delete the entity from the database:

```
CustomerOrder order = em.find(CustomerOrder.class, orderId);
em.remove(order);
```

Running the order Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `order` application. First, you will create the database tables in Apache Derby.

To Run the order Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. From the **File** menu, choose **Open Project**.
4. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/persistence
   ```

5. Select the `order` folder.
6. Click **Open Project**.
7. In the **Projects** tab, right-click the `order` project and select **Run**.

   NetBeans IDE opens a web browser to the following URL:

   ```
   http://localhost:8080/order/
   ```

To Run the order Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. In a terminal window, go to:

```
jakartaee-examples/tutorial/persistence/order/
```

4. Enter the following command:

```
mvn install
```

This compiles the source files and packages the application into a WAR file located at `jakartaee-examples/tutorial/persistence/order/target/order.war`. Then the WAR file is deployed to your GlassFish Server instance.

5. To create and update the order data, open a web browser to the following URL:

```
http://localhost:8080/order/
```

The roster Application

The `roster` application maintains the team rosters for players in recreational sports leagues. The application has four components: Jakarta Persistence entities (`Player`, `Team`, and `League`), a stateful session bean (`RequestBean`), an application client (`RosterClient`), and three helper classes (`PlayerDetails`, `TeamDetails`, and `LeagueDetails`).

Functionally, `roster` is similar to the `order` application, with three new features that `order` does not have: many-to-many relationships, entity inheritance, and automatic table creation at deployment time.

The database schema in Apache Derby for the `roster` application is shown in Figure 19, “Database Schema for the roster Application”.

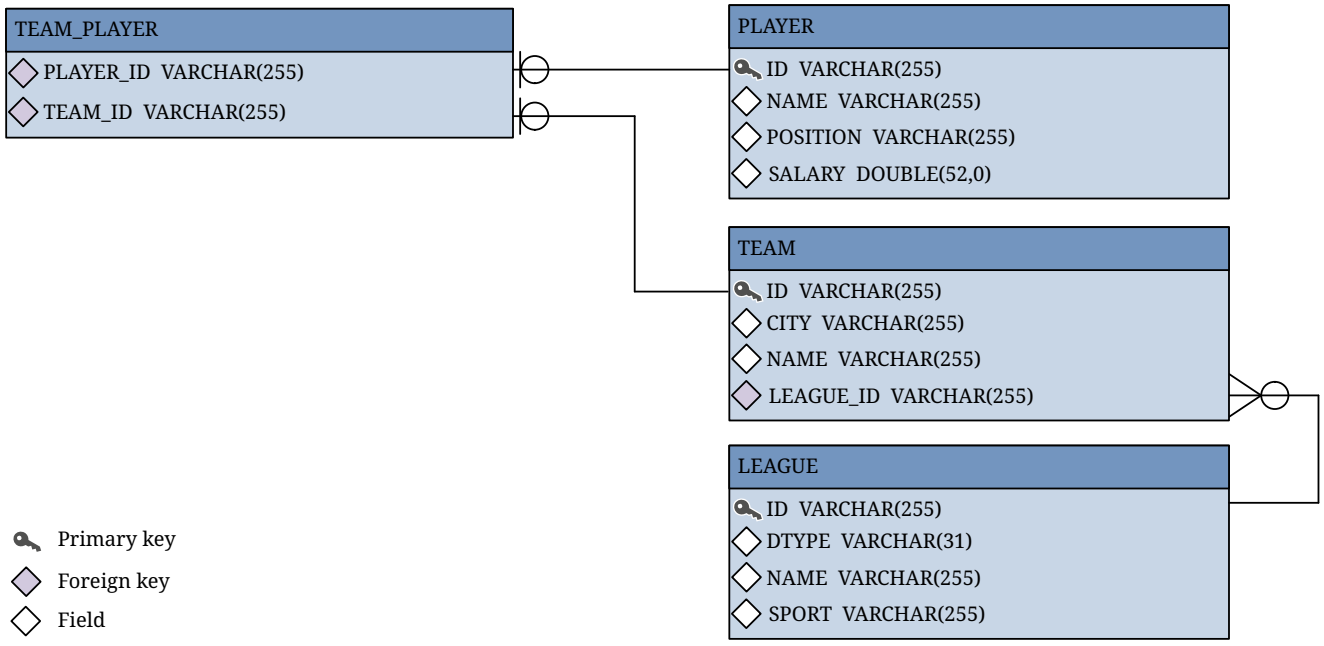


Figure 19. Database Schema for the roster Application



In this diagram, for simplicity, the `PERSISTENCE_ROSTER_` prefix is omitted from the table names.

Relationships in the roster Application

A recreational sports system has the following relationships.

- A player can be on many teams.
- A team can have many players.
- A team is in exactly one league.
- A league has many teams.

In `roster` this system is reflected by the following relationships between the `Player`, `Team`, and `League` entities.

- There is a many-to-many relationship between `Player` and `Team`.
- There is a many-to-one relationship between `Team` and `League`.

The Many-To-Many Relationship in roster

The many-to-many relationship between `Player` and `Team` is specified by using the `@ManyToMany` annotation. In `Team.java`, the `@ManyToMany` annotation decorates the `getPlayers` method:

```
@ManyToMany
@JoinTable(
    name="PERSISTENCE_ROSTER_TEAM_PLAYER",
    joinColumns=
        @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
    inverseJoinColumns=
        @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
)
public Collection<Player> getPlayers() {
    return players;
}
```

The `@JoinTable` annotation is used to specify a database table that will associate player IDs with team IDs. The entity that specifies the `@JoinTable` is the owner of the relationship, so the `Team` entity is the owner of the relationship with the `Player` entity. Because `roster` uses automatic table creation at deployment time, the container will create a join table named `PERSISTENCE_ROSTER_TEAM_PLAYER`.

`Player` is the inverse, or nonowning, side of the relationship with `Team`. As one-to-one and many-to-one relationships, the nonowning side is marked by the `mappedBy` element in the relationship annotation. Because the relationship between `Player` and `Team` is bidirectional, the choice of which entity is the owner of the relationship is arbitrary.

In `Player.java`, the `@ManyToMany` annotation decorates the `getTeams` method:

```
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}
```

Entity Inheritance in the roster Application

The `roster` application shows how to use entity inheritance, as described in Entity Inheritance.

The `League` entity in `roster` is an abstract entity with two concrete subclasses: `SummerLeague` and `WinterLeague`. Because `League` is an abstract class, it cannot be instantiated:

```
@Entity
@Table(name = "PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

Instead, when creating a league, clients use `SummerLeague` or `WinterLeague`. `SummerLeague` and `WinterLeague` inherit the persistent properties defined in `League` and add only a constructor that verifies that the sport parameter matches the type of sport allowed in that seasonal league. For example, here is the `SummerLeague` entity:

```
...
@Entity
public class SummerLeague extends League implements Serializable {

    /** Creates a new instance of SummerLeague */
    public SummerLeague() {
    }

    public SummerLeague(String id, String name, String sport)
            throws IncorrectSportException {
        this.id = id;
        this.name = name;
        if (sport.equalsIgnoreCase("swimming") ||
                sport.equalsIgnoreCase("soccer") ||
                sport.equalsIgnoreCase("basketball") ||
                sport.equalsIgnoreCase("baseball")) {
            this.sport = sport;
        } else {
            throw new IncorrectSportException("Sport is not a summer sport.");
        }
    }
}
```

The `roster` application uses the default mapping strategy of `InheritanceType.SINGLE_TABLE`, so the `@Inheritance` annotation is not required. If you want to use a different mapping strategy, decorate `League` with `@Inheritance` and specify the mapping strategy in the `strategy` element:

```
@Entity
@Inheritance(strategy=JOINED)
@Table(name="PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

The `roster` application uses the default discriminator column name, so the `@DiscriminatorColumn` annotation is not required. Because you are using automatic table generation in `roster`, the Persistence provider will create a discriminator column called `DTYPE` in the `PERSISTENCE_ROSTER_LEAGUE` table, which will store the name of the inherited entity used to create the league. If you want to use a different name for the discriminator column, decorate `League` with `@DiscriminatorColumn` and set the `name` element:

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="PERSISTENCE_ROSTER_LEAGUE")
public abstract class League implements Serializable { ... }
```

Criteria Queries in the roster Application

The `roster` application uses Criteria API queries, as opposed to the JPQL queries used in `order`. Criteria queries are Java programming language, typesafe queries defined in the business tier of `roster`, in the `RequestBean` stateful session bean.

Metamodel Classes in the roster Application

Metamodel classes model an entity's attributes and are used by Criteria queries to navigate to an entity's attributes. Each entity class in `roster` has a corresponding metamodel class, generated at compile time, with the same package name as the entity and appended with an underscore character (_). For example, the `roster.entity.Player` entity has a corresponding metamodel class, `roster.entity.Player_`.

Each persistent field or property in the entity class has a corresponding attribute in the entity's metamodel class. For the `Player` entity, the corresponding metamodel class is as follows:

```
@StaticMetamodel(Player.class)
public class Player_ {
    public static volatile SingularAttribute<Player, String> id;
    public static volatile SingularAttribute<Player, String> name;
    public static volatile SingularAttribute<Player, String> position;
    public static volatile SingularAttribute<Player, Double> salary;
    public static volatile CollectionAttribute<Player, Team> teams;
}
```

Obtaining a CriteriaBuilder Instance in RequestBean

The `CriteriaBuilder` interface defines methods to create criteria query objects and create expressions for modifying those query objects. `RequestBean` creates an instance of `CriteriaBuilder`

by using a `@PostConstruct` method, `init`:

```
@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;

@PostConstruct
private void init() {
    cb = em.getCriteriaBuilder();
}
```

The `EntityManager` instance is injected at runtime, and then that `EntityManager` object is used to create the `CriteriaBuilder` instance by calling `getCriteriaBuilder`. The `CriteriaBuilder` instance is created in a `@PostConstruct` method to ensure that the `EntityManager` instance has been injected by the enterprise bean container.

Creating Criteria Queries in RequestBean's Business Methods

Many of the business methods in `RequestBean` define Criteria queries. One business method, `getPlayersByPosition`, returns a list of players who play a particular position on a team:

```
public List<PlayerDetails> getPlayersByPosition(String position) {
    logger.info("getPlayersByPosition");
    List<Player> players = null;

    try {
        CriteriaQuery<Player> cq = cb.createQuery(Player.class);
        if (cq != null) {
            Root<Player> player = cq.from(Player.class);

            // set the where clause
            cq.where(cb.equal(player.get(Player_.position), position));
            cq.select(player);
            TypedQuery<Player> q = em.createQuery(cq);
            players = q.getResultList();
        }
        return copyPlayersToDetails(players);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

A query object is created by calling the `CriteriaBuilder` object's `createQuery` method, with the type set to `Player` because the query will return a list of players.

The query root, the base entity from which the query will navigate to find the entity's attributes and related entities, is created by calling the `from` method of the query object. This sets the `FROM` clause of the query.

The `WHERE` clause, set by calling the `where` method on the query object, restricts the results of the query according to the conditions of an expression. The `CriteriaBuilder.equal` method compares the two expressions. In `getPlayersByPosition`, the `position` attribute of the `Player_` metamodel class, accessed by calling the `get` method of the query root, is compared to the `position` parameter passed to `getPlayersByPosition`.

The `SELECT` clause of the query is set by calling the `select` method of the query object. The query will return `Player` entities, so the query root object is passed as a parameter to `select`.

The query object is prepared for execution by calling `EntityManager.createQuery`, which returns a `TypedQuery<T>` object with the type of the query, in this case `Player`. This typed query object is used to execute the query, which occurs when the `getResultList` method is called, and a `List<Player>` collection is returned.

Automatic Table Generation in the roster Application

At deployment time, GlassFish Server will automatically drop and create the database tables used by `roster`. This is done by setting the `jakarta.persistence.schema-generation.database.action` property to `drop-and-create` in `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.0"
    xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
        https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="em" transaction-type="JTA">
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <properties>
      <property name="jakarta.persistence.schema-generation.database.action"
                value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Running the roster Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `roster` application.

To Run the roster Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. From the **File** menu, choose **Open Project**.
4. In the **Open Project** dialog box, navigate to:

```
jakartaee-examples/tutorial/persistence
```

5. Select the `roster` folder.
6. Select the **Open Required Projects** check box.
7. Click **Open Project**.
8. In the **Projects** tab, right-click the `roster` project and select **Build**.

   This will compile, package, and deploy the EAR to GlassFish Server.

   You will see the following partial output from the application client in the Output tab:

```
List all players in team T2:
P6 Ian Carlyle goalkeeper 555.0
P7 Rebecca Struthers midfielder 777.0
P8 Anne Anderson forward 65.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder 100.0

List all teams in league L1:
T1 Honey Bees Visalia
T2 Gophers Manteca
T5 Crows Orland

List all defenders:
P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0
P25 Frank Fletcher defender 399.0
```

To Run the roster Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. In a terminal window, go to:

```
jakartaee-examples/tutorial/persistence/roster/roster-ear/
```

4. Enter the following command:

```
mvn install
```

   This compiles the source files and packages the application into an EAR file located at

jakartaee-examples/tutorial/persistence/roster/target/roster.ear. The EAR file is then deployed to GlassFish Server. GlassFish Server will then drop and create the database tables during deployment, as specified in persistence.xml.

After successfully deploying the EAR, the client stubs are retrieved and the application client is run using the appclient application included with GlassFish Server.

You will see the output, which begins as follows:

```
[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0

[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland

[exec] List all defenders:
[exec] P2 Alice Smith defender 505.0
[exec] P5 Barney Bold defender 100.0
[exec] P9 Jan Wesley defender 100.0
[exec] P22 Janice Walker defender 857.0
[exec] P25 Frank Fletcher defender 399.0
```

The address-book Application

The address-book example application is a simple web application that stores contact data. It uses a single entity class, Contact, that uses Jakarta Bean Validation to validate the data stored in the persistent attributes of the entity, as described in Validating Persistent Fields and Properties.

Bean Validation Constraints in address-book

The Contact entity uses the @NotNull, @Pattern, and @Past constraints on the persistent attributes.

The @NotNull constraint marks the attribute as a required field. The attribute must be set to a non-null value before the entity can be persisted or modified. Bean Validation will throw a validation error if the attribute is null when the entity is persisted or modified.

The @Pattern constraint defines a regular expression that the value of the attribute must match before the entity can be persisted or modified. This constraint has two different uses in address-book.

- The regular expression declared in the @Pattern annotation on the email field matches email addresses of the form name@domain name.top level domain, allowing only valid characters for email addresses. For example, username@example.com will pass validation, as will

`firstname.lastname@mail.example.com`. However, `firstname,lastname@example.com`, which contains an illegal comma character in the local name, will fail validation.

- The `mobilePhone` and `homePhone` fields are annotated with a `@Pattern` constraint that defines a regular expression to match phone numbers of the form `(xxx) xxx-xxxx`.

The `@Past` constraint is applied to the birthday field, which must be a `java.util.Date` in the past.

Here are the relevant parts of the `Contact` entity class:

```
@Entity
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
            + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
            + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"
            + "(?:[a-z0-9-]*[a-z0-9])?",
            message = "{invalid.email}")
    protected String email;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(jakarta.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

Specifying Error Messages for Constraints in address-book

Some of the constraints in the `Contact` entity specify an optional message:

```
@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
        message = "{invalid.phonenumber}")
protected String homePhone;
```

The optional message element in the `@Pattern` constraint overrides the default validation message. The message can be specified directly:

```
@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
```

```
        message = "Invalid phone number!")
protected String homePhone;
```

The constraints in `Contact`, however, are strings in the resource bundle `ValidationMessages.properties`, under `jakartaee-examples/tutorial/persistence/address-book/src/main/java`. This allows the validation messages to be located in one single properties file and the messages to be easily localized. Overridden Bean Validation messages must be placed in a resource bundle properties file named `ValidationMessages.properties` in the default package, with localized resource bundles taking the form `ValidationMessages_locale-prefix.properties`. For example, `ValidationMessages_es.properties` is the resource bundle used in Spanish-speaking locales.

Validating Contact Input from a Jakarta Faces Application

The `address-book` application uses a Jakarta Faces web front end to allow users to enter contacts. While Jakarta Faces has a form input validation mechanism using tags in Facelets XHTML files, `address-book` doesn't use these validation tags. Bean Validation constraints in Jakarta Faces managed beans, in this case in the `Contact` entity, automatically trigger validation when the forms are submitted.

The following code snippet from the `Create.xhtml` Facelets file shows some of the input form for creating new `Contact` instances:

```
<h:form>
    <table columns="3" role="presentation">
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_firstName}"
                               for="firstName" /></td>
            <td><h:inputText id="firstName"
                             value="#{contactController.selected.firstName}"
                             title="#{bundle.CreateContactTitle_firstName}"/>
            </td>
            <td><h:message for="firstName" /></td>
        </tr>
        <tr>
            <td><h:outputLabel value="#{bundle.CreateContactLabel_lastName}"
                               for="lastName" /></td>
            <td><h:inputText id="lastName"
                             value="#{contactController.selected.lastName}"
                             title="#{bundle.CreateContactTitle_lastName}" />
            </td>
            <td><h:message for="lastName" /></td>
        </tr>
        ...
    </table>
</h:form>
```

The `<h:inputText>` tags `firstName` and `lastName` are bound to the attributes in the `Contact` entity instance `selected` in the `ContactController` stateless session bean. Each `<h:inputText>` tag has an associated `<h:message>` tag that will display validation error messages. The form doesn't require any

Jakarta Faces validation tags, however.

Running the address-book Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `address-book` application.

To Run the address-book Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. From the **File** menu, choose **Open Project**.
4. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/persistence
   ```

5. Select the `address-book` folder.
6. Click **Open Project**.
7. In the **Projects** tab, right-click the `address-book` project and select **Run**.

   After the application has been deployed, a web browser window appears at the following URL:

   ```
   http://localhost:8080/address-book/
   ```

8. Click Show All Contact Items, then Create New Contact. Enter values in the fields; then click Save.

   If any of the values entered violate the constraints in `Contact`, an error message will appear in red beside the field with the incorrect values.

To Run the address-book Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. If the database server is not already running, start it by following the instructions in Starting and Stopping Apache Derby.
3. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/persistence/address-book/
   ```

4. Enter the following command:

   ```
   mvn install
   ```

This will compile and assemble the `address-book` application into a WAR. The WAR file is then deployed to GlassFish Server.

5. Open a web browser window and enter the following URL:

```
http://localhost:8080/address-book/
```

6. Click Show All Contact Items, then Create New Contact. Enter values in the fields; then click Save.

   If any of the values entered violate the constraints in `Contact`, an error message will appear in red beside the field with the incorrect values.

The Jakarta Persistence Query Language



This section was last updated for Jakarta EE Version 9.1

This chapter describes the Jakarta Persistence query language that defines queries for entities and their persistent state. The query language allows you to write portable queries that work regardless of the underlying data store.

Overview of the Jakarta Persistence Query Language

The query language uses the abstract persistence schemas of entities, including their relationships, for its data model and defines operators and expressions based on this data model. The scope of a query spans the abstract schemas of related entities that are packaged in the same persistence unit. The query language uses an SQL-like syntax to select objects or values based on entity abstract schema types and relationships among them.

This chapter relies on the material presented in earlier chapters. For conceptual information, see [persist:persistence-intro::persistence-intro:::_introduction_to_jakarta_persistence]. For code examples, see [persist:persistence-basicexamples::persistence-basicexamples:::_running_the_persistence_examples].

Query Language Terminology

The following list defines some of the terms referred to in this chapter.

- Abstract schema: The persistent schema abstraction (persistent entities, their state, and their relationships) over which queries operate. The query language translates queries over this persistent schema abstraction into queries that are executed over the database schema to which entities are mapped.
- Abstract schema type: The type to which the persistent property of an entity evaluates in the abstract schema. That is, each persistent field or property in an entity has a corresponding state field of the same type in the abstract schema. The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations.
- Backus-Naur Form (BNF): A notation that describes the syntax of high-level languages. The

syntax diagrams in this chapter are in BNF notation.

- Navigation: The traversal of relationships in a query language expression. The navigation operator is a period.
- Path expression: An expression that navigates to an entity's state or relationship field.
- State field: A persistent field of an entity.
- Relationship field: A persistent field of an entity whose type is the abstract schema type of the related entity.

Creating Queries Using the Jakarta Persistence Query Language

The `EntityManager.createQuery` and `EntityManager.createNamedQuery` methods are used to query the datastore by using Jakarta Persistence query language queries.

The `createQuery` method is used to create dynamic queries, which are queries defined directly within an application's business logic:

```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

The `createNamedQuery` method is used to create static queries, or queries that are defined in metadata by using the `jakarta.persistence.NamedQuery` annotation. The `name` element of `@NamedQuery` specifies the name of the query that will be used with the `createNamedQuery` method. The `query` element of `@NamedQuery` is the query:

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

Here's an example of `createNamedQuery`, which uses the `@NamedQuery`:

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

Named Parameters in Queries

Named parameters are query parameters that are prefixed with a colon (`:`). Named parameters in a query are bound to an argument by the following method:

```
jakarta.persistence.Query.setParameter(String name, Object value)
```

In the following example, the `name` argument to the `findWithName` business method is bound to the `:custName` named parameter in the query by calling `Query.setParameter`:

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .getResultList();
}
```

Named parameters are case-sensitive and may be used by both dynamic and static queries.

Positional Parameters in Queries

You may use positional parameters instead of named parameters in queries. Positional parameters are prefixed with a question mark (`?`) followed by the numeric position of the parameter in the query. The method `Query.setParameter(integer position, Object value)` is used to set the parameter values.

In the following example, the `findWithName` business method is rewritten to use input parameters:

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1, name)
        .getResultList();
}
```

Input parameters are numbered starting from 1. Input parameters are case-sensitive, and may be used by both dynamic and static queries.

Simplified Query Language Syntax

This section briefly describes the syntax of the query language so that you can quickly move on to Example Queries. When you are ready to learn about the syntax in more detail, see Full Query Language Syntax.

Select Statements

A select query has six clauses: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`. The `SELECT` and `FROM` clauses are required, but the `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses are optional. Here is the

high-level BNF syntax of a query language select query:

```
QL_statement ::= select_clause from_clause
  [where_clause][groupby_clause][having_clause][orderby_clause]
```

The BNF syntax defines the following clauses.

- The `SELECT` clause defines the types of the objects or values returned by the query.
- The `FROM` clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the `SELECT` and `WHERE` clauses. An identification variable represents one of the following elements:
  - The abstract schema name of an entity
  - An element of a collection relationship
  - An element of a single-valued relationship
  - A member of a collection that is the multiple side of a one-to-many relationship
- The `WHERE` clause is a conditional expression that restricts the objects or values retrieved by the query. Although the clause is optional, most queries have a `WHERE` clause.
- The `GROUP BY` clause groups query results according to a set of properties.
- The `HAVING` clause is used with the `GROUP BY` clause to further restrict the query results according to a conditional expression.
- The `ORDER BY` clause sorts the objects or values returned by the query into a specified order.

Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities. These statements have the following syntax:

```
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]
```

The update and delete clauses determine the type of the entities to be updated or deleted. The `WHERE` clause may be used to restrict the scope of the update or delete operation.

Example Queries

The following queries are from the `Player` entity of the `roster` application, which is documented in The roster Application.

Simple Queries

If you are unfamiliar with the query language, these simple queries are a good place to start.

A Basic Select Query

```
SELECT p
FROM Player p
```

- Data retrieved: All players.
- Description: The `FROM` clause declares an identification variable named `p`, omitting the optional keyword `AS`. If the `AS` keyword were included, the clause would be written as follows:

  ```
  FROM Player AS p
  ```

  The `Player` element is the abstract schema name of the `Player` entity.
- See also: Identification Variables.

Eliminating Duplicate Values

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = ?1
```

- Data retrieved: The players with the position specified by the query's parameter.
- Description: The `DISTINCT` keyword eliminates duplicate values.

  The `WHERE` clause restricts the players retrieved by checking their `position`, a persistent field of the `Player` entity. The `?1` element denotes the input parameter of the query.
- See also: Input Parameters and The DISTINCT Keyword.

Using Named Parameters

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

- Data retrieved: The players having the specified positions and names.
- Description: The `position` and `name` elements are persistent fields of the `Player` entity. The `WHERE` clause compares the values of these fields with the named parameters of the query, set using the `Query.setNamedParameter` method. The query language denotes a named input parameter using a colon (`:`) followed by an identifier. The first input parameter is `:position`, the second is `:name`.

Queries That Navigate to Related Entities

In the query language, an expression can traverse, or navigate, to related entities. These expressions are the primary difference between the Jakarta Persistence query language and SQL.

Queries navigates to related entities, whereas SQL joins tables.

A Simple Query with Relationships

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
```

- Data retrieved: All players who belong to a team.
- Description: The `FROM` clause declares two identification variables: `p` and `t`. The `p` variable represents the `Player` entity, and the `t` variable represents the related `Team` entity. The declaration for `t` references the previously declared `p` variable. The `IN` keyword signifies that `teams` is a collection of related entities. The `p.teams` expression navigates from a `Player` to its related `Team`. The period in the `p.teams` expression is the navigation operator.

  You may also use the `JOIN` statement to write the same query:

  ```
  SELECT DISTINCT p
  FROM Player p JOIN p.teams t
  ```

  This query could also be rewritten as:

  ```
  SELECT DISTINCT p
  FROM Player p
  WHERE p.team IS NOT EMPTY
  ```

Navigating to Single-Valued Relationship Fields

Use the `JOIN` clause statement to navigate to a single-valued relationship field:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport ='football'
```

In this example, the query will return all teams that are in either soccer or football leagues.

Traversing Relationships with an Input Parameter

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

- Data retrieved: The players whose teams belong to the specified city.
- Description: This query is similar to the previous example but adds an input parameter. The `AS` keyword in the `FROM` clause is optional. In the `WHERE` clause, the period preceding the persistent

variable `city` is a delimiter, not a navigation operator. Strictly speaking, expressions can navigate to relationship fields (related entities) but not to persistent fields. To access a persistent field, an expression uses the period as a delimiter.

Expressions cannot navigate beyond (or further qualify) relationship fields that are collections. In the syntax of an expression, a collection-valued field is a terminal symbol. Because the `teams` field is a collection, the `WHERE` clause cannot specify `p.teams.city` (an illegal expression).

- See also: Path Expressions.

Traversing Multiple Relationships

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

- Data retrieved: The players who belong to the specified league.
- Description: The expressions in this query navigate over two relationships. The `p.teams` expression navigates the `Player-Team` relationship, and the `t.league` expression navigates the `Team-League` relationship.

In the other examples, the input parameters are `String` objects; in this example, the parameter is an object whose type is a `League`. This type matches the `league` relationship field in the comparison expression of the `WHERE` clause.

Navigating According to Related Fields

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

- Data retrieved: The players who participate in the specified sport.
- Description: The `sport` persistent field belongs to the `League` entity. To reach the `sport` field, the query must first navigate from the `Player` entity to `Team` (`p.teams`) and then from `Team` to the `League` entity (`t.league`). Because it is not a collection, the `league` relationship field can be followed by the `sport` persistent field.

Queries with Other Conditional Expressions

Every `WHERE` clause must specify a conditional expression, of which there are several kinds. In the previous examples, the conditional expressions are comparison expressions that test for equality. The following examples demonstrate some of the other kinds of conditional expressions. For descriptions of all conditional expressions, see WHERE Clause.

The LIKE Expression

```
SELECT p
```

```
FROM Player p
WHERE p.name LIKE 'Mich%'
```

- Data retrieved: All players whose names begin with "Mich."
- Description: The `LIKE` expression uses wildcard characters to search for strings that match the wildcard pattern. In this case, the query uses the `LIKE` expression and the `%` wildcard to find all players whose names begin with the string "Mich." For example, "Michael" and "Michelle" both match the wildcard pattern.
- See also: LIKE Expressions.

The IS NULL Expression

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

- Data retrieved: All teams not associated with a league.
- Description: The `IS NULL` expression can be used to check whether a relationship has been set between two entities. In this case, the query checks whether the teams are associated with any leagues and returns the teams that do not have a league.
- See also: NULL Comparison Expressions and NULL Values.

The IS EMPTY Expression

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- Data retrieved: All players who do not belong to a team.
- Description: The `teams` relationship field of the `Player` entity is a collection. If a player does not belong to a team, the `teams` collection is empty, and the conditional expression is `TRUE`.
- See also: Empty Collection Comparison Expressions.

The BETWEEN Expression

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

- Data retrieved: The players whose salaries fall within the range of the specified salaries.
- Description: This `BETWEEN` expression has three arithmetic expressions: a persistent field (`p.salary`) and the two input parameters (`:lowerSalary` and `:higherSalary`). The following expression is equivalent to the `BETWEEN` expression:

```
p.salary >= :lowerSalary AND p.salary <= :higherSalary
```

- See also: BETWEEN Expressions.

Comparison Operators

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

- Data retrieved: All players whose salaries are higher than the salary of the player with the specified name.
- Description: The `FROM` clause declares two identification variables (`p1` and `p2`) of the same type (`Player`). Two identification variables are needed because the `WHERE` clause compares the salary of one player (`p2`) with that of the other players (`p1`).
- See also: Identification Variables.

Bulk Updates and Deletes

The following examples show how to use the `UPDATE` and `DELETE` expressions in queries. `UPDATE` and `DELETE` operate on multiple entities according to the condition or conditions set in the `WHERE` clause. The `WHERE` clause in `UPDATE` and `DELETE` queries follows the same rules as `SELECT` queries.

Update Queries

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

- Description: This query sets the status of a set of players to `inactive` if the player's last game was longer ago than the date specified in `inactiveThresholdDate`.

Delete Queries

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

- Description: This query deletes all inactive players who are not on a team.

Full Query Language Syntax

This section discusses the query language syntax, as defined in the Jakarta Persistence 3.0 specification available at https://jakarta.ee/specifications/persistence/3.0/. Much of the following

material paraphrases or directly quotes the specification.

BNF Symbols

BNF Symbol Summary describes the BNF symbols used in this chapter.

BNF Symbol Summary

Symbol	Description
`::=`	The element to the left of the symbol is defined by the constructs on the right.
`*`	The preceding construct may occur zero or more times.
`{···}`	The constructs within the braces are grouped together.
`[···]`	The constructs within the brackets are optional.
`|`	An exclusive `OR`.
`BOLDFACE`	A keyword; although capitalized in the BNF diagram, keywords are not case-sensitive.
White space	A whitespace character can be a space, a horizontal tab, or a line feed.

BNF Grammar of the Jakarta Persistence Query Language

Here is the entire BNF diagram for the query language:

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
    [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
    FROM identification_variable_declaration
        {, {identification_variable_declaration |
            collection_member_declaration}}*
identification_variable_declaration ::=
        range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS]
        identification_variable
join ::= join_spec join_association_path_expression [AS]
        identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::=
        collection_valued_path_expression |
        single_valued_association_path_expression
join_spec::= [LEFT [OUTER] |INNER] JOIN
```

```
join_association_path_expression ::=
        join_collection_valued_path_expression |
        join_single_valued_association_path_expression
join_collection_valued_path_expression::=
    identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=
        identification_variable.single_valued_association_field
collection_member_declaration ::=
        IN (collection_valued_path_expression) [AS]
        identification_variable
single_valued_path_expression ::=
        state_field_path_expression |
        single_valued_association_path_expression
state_field_path_expression ::=
    {identification_variable |
    single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
    identification_variable.{single_valued_association_field.}*
    single_valued_association_field
collection_valued_path_expression ::=
    identification_variable.{single_valued_association_field.}*
    collection_valued_association_field
state_field ::=
    {embedded_class_state_field.}*simple_state_field
update_clause ::=UPDATE abstract_schema_name [[AS]
    identification_variable] SET update_item {, update_item}*
update_item ::= [identification_variable.]{state_field |
    single_valued_association_field} = new_value
new_value ::=
     simple_arithmetic_expression |
    string_primary |
    datetime_primary |
    boolean_primary |
    enum_primary simple_entity_expression |
    NULL
delete_clause ::= DELETE FROM abstract_schema_name [[AS]
    identification_variable]
select_clause ::= SELECT [DISTINCT] select_expression {,
    select_expression}*
select_expression ::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name(constructor_item {,
    constructor_item}*)
constructor_item ::= single_valued_path_expression |
    aggregate_expression
aggregate_expression ::=
```

```
    {AVG |MAX |MIN |SUM} ([DISTINCT]
        state_field_path_expression) |
    COUNT ([DISTINCT] identification_variable |
        state_field_path_expression |
        single_valued_association_path_expression)
where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ASC |DESC]
subquery ::= simple_select_clause subquery_from_clause
    [where_clause] [groupby_clause] [having_clause]
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
        {, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    association_path_expression [AS] identification_variable |
    collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
    simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable
conditional_expression ::= conditional_term |
    conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND
    conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression |(
    conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    like_expression |
    in_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expressionAND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND
        string_expression |
    datetime_expression [NOT] BETWEEN
        datetime_expression AND datetime_expression
in_expression ::=
    state_field_path_expression [NOT] IN (in_item {, in_item}*
```

```
    | subquery)
in_item ::= literal | input_parameter
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE
        escape_character]
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT]
        NULL
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)
all_or_any_expression ::= {ALL |ANY |SOME} (subquery)
comparison_expression ::=
    string_expression comparison_operator {string_expression |
    all_or_any_expression} |
    boolean_expression {= |<> } {boolean_expression |
    all_or_any_expression} |
    enum_expression {= |<> } {enum_expression |
    all_or_any_expression} |
    datetime_expression comparison_operator
        {datetime_expression | all_or_any_expression} |
    entity_expression {= |<> } {entity_expression |
    all_or_any_expression} |
    arithmetic_expression comparison_operator
        {arithmetic_expression | all_or_any_expression}
comparison_operator ::= = |> |>= |< |<= |<>
arithmetic_expression ::= simple_arithmetic_expression |
    (subquery)
simple_arithmetic_expression ::=
    arithmetic_term | simple_arithmetic_expression {+ |- }
        arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* |/ }
    arithmetic_factor
arithmetic_factor ::= [{+ |- }] arithmetic_primary
arithmetic_primary ::=
    state_field_path_expression |
    numeric_literal |
    (simple_arithmetic_expression) |
    input_parameter |
    functions_returning_numerics |
    aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::=
    state_field_path_expression |
    string_literal |
    input_parameter |
    functions_returning_strings |
    aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
```

```
datetime_primary ::=
    state_field_path_expression |
    input_parameter |
    functions_returning_datetime |
    aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::=
    state_field_path_expression |
    boolean_literal |
    input_parameter
 enum_expression ::= enum_primary | (subquery)
enum_primary ::=
    state_field_path_expression |
    enum_literal |
    input_parameter
entity_expression ::=
    single_valued_association_path_expression |
        simple_entity_expression
simple_entity_expression ::=
    identification_variable |
    input_parameter
functions_returning_numerics::=
    LENGTH(string_primary) |
    LOCATE(string_primary, string_primary[,
        simple_arithmetic_expression]) |
    ABS(simple_arithmetic_expression) |
    SQRT(simple_arithmetic_expression) |
    MOD(simple_arithmetic_expression,
        simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression)
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
functions_returning_strings ::=
    CONCAT(string_primary, string_primary) |
    SUBSTRING(string_primary,
        simple_arithmetic_expression,
        simple_arithmetic_expression)|
    TRIM([[trim_specification] [trim_character] FROM]
        string_primary) |
    LOWER(string_primary) |
    UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH
```

FROM Clause

The `FROM` clause defines the domain of the query by declaring identification variables.

Identifiers

An identifier is a sequence of one or more characters. The first character must be a valid first character (letter, `$`, `_`) in an identifier of the Java programming language, hereafter in this chapter called simply "Java". Each subsequent character in the sequence must be a valid nonfirst character (letter, digit, `$`, `_`) in a Java identifier. (For details, see the Java SE API documentation of the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods of the `Character` class.) The question mark (`?`) is a reserved character in the query language and cannot be used in an identifier.

A query language identifier is case-sensitive, with two exceptions:

- Keywords
- Identification variables

An identifier cannot be the same as a query language keyword. Here is a list of query language keywords:

ABS	ALL	AND	ANY
AS	ASC	AVG	BETWEEN
BIT_LENGTH	BOTH	BY	CASE
CHAR_LENGTH	CHARACTER_LENGTH	CLASS	COALESCE
CONCAT	COUNT	CURRENT_DATE	CURRENT_TIMESTAMP
DELETE	DESC	DISTINCT	ELSE
EMPTY	END	ENTRY	ESCAPE
EXISTS	FALSE	FETCH	FROM
GROUP	HAVING	IN	INDEX
INNER	IS	JOIN	KEY
LEADING	LEFT	LENGTH	LIKE
LOCATE	LOWER	MAX	MEMBER
MIN	MOD	NEW	NOT
NULL	NULLIF	OBJECT	OF
OR	ORDER	OUTER	POSITION
SELECT	SET	SIZE	SOME
SQRT	SUBSTRING	SUM	THEN
TRAILING	TRIM	TRUE	TYPE
UNKNOWN	UPDATE	UPPER	VALUE
WHEN	WHERE		

It is not recommended that you use an SQL keyword as an identifier, because the list of keywords may expand to include other reserved SQL words in the future.

Identification Variables

An identification variable is an identifier declared in the `FROM` clause. Although they can reference identification variables, the `SELECT` and `WHERE` clauses cannot declare them. All identification

variables must be declared in the FROM clause.

Because it is an identifier, an identification variable has the same naming conventions and restrictions as an identifier, with the exception that an identification variable is case-insensitive. For example, an identification variable cannot be the same as a query language keyword. (See Identifiers for more naming rules.) Also, within a given persistence unit, an identification variable name must not match the name of any entity or abstract schema.

The FROM clause can contain multiple declarations, separated by commas. A declaration can reference another identification variable that has been previously declared (to the left). In the following FROM clause, the variable t references the previously declared variable p:

```
FROM Player p, IN (p.teams) AS t
```

Even if it is not used in the WHERE clause, an identification variable's declaration can affect the results of the query. For example, compare the next two queries. The following query returns all players, whether or not they belong to a team:

```
SELECT p
FROM Player p
```

In contrast, because it declares the t identification variable, the next query fetches all players who belong to a team:

```
SELECT p
FROM Player p, IN (p.teams) AS t
```

The following query returns the same results as the preceding query, but the WHERE clause makes it easier to read:

```
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

An identification variable always designates a reference to a single value whose type is that of the expression used in the declaration. There are two kinds of declarations: range variable and collection member.

Range Variable Declarations

To declare an identification variable as an abstract schema type, you specify a range variable declaration. In other words, an identification variable can range over the abstract schema type of an entity. In the following example, an identification variable named p represents the abstract schema named Player:

```
FROM Player p
```

A range variable declaration can include the optional `AS` operator:

```
FROM Player AS p
```

To obtain objects, a query usually uses path expressions to navigate through the relationships. But for those objects that cannot be obtained by navigation, you can use a range variable declaration to designate a starting point, or query root.

If the query compares multiple values of the same abstract schema type, the `FROM` clause must declare multiple identification variables for the abstract schema:

```
FROM Player p1, Player p2
```

For an example of such a query, see Comparison Operators.

Collection Member Declarations

In a one-to-many relationship, the multiple side consists of a collection of entities. An identification variable can represent a member of this collection. To access a collection member, the path expression in the variable's declaration navigates through the relationships in the abstract schema. (For more information on path expressions, see Path Expressions.) Because a path expression can be based on another path expression, the navigation can traverse several relationships. See Traversing Multiple Relationships.

A collection member declaration must include the `IN` operator but can omit the optional `AS` operator.

In the following example, the entity represented by the abstract schema named `Player` has a relationship field called `teams`. The identification variable called `t` represents a single member of the `teams` collection:

```
FROM Player p, IN (p.teams) t
```

Joins

The `JOIN` operator is used to traverse over relationships between entities and is functionally similar to the `IN` operator.

In the following example, the query joins over the relationship between customers and orders:

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

The INNER keyword is optional:

```
SELECT c
FROM Customer c INNER JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

These examples are equivalent to the following query, which uses the IN operator:

```
SELECT c
FROM Customer c, IN (c.orders) o
WHERE c.status = 1 AND o.totalPrice > 10000
```

You can also join a single-valued relationship:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = :sport
```

A LEFT JOIN or LEFT OUTER JOIN retrieves a set of entities where matching values in the join condition may be absent. The OUTER keyword is optional:

```
SELECT c.name, o.totalPrice
FROM CustomerOrder o LEFT JOIN o.customer c
```

A FETCH JOIN is a join operation that returns associated entities as a side effect of running the query. In the following example, the query returns a set of departments and, as a side effect, the associated employees of the departments, even though the employees were not explicitly retrieved by the SELECT clause:

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

Path Expressions

Path expressions are important constructs in the syntax of the query language for several reasons. First, path expressions define navigation paths through the relationships in the abstract schema. These path definitions affect both the scope and the results of a query. Second, path expressions can appear in any of the main clauses of a query (SELECT, DELETE, HAVING, UPDATE, WHERE, FROM, GROUP BY, ORDER BY). Finally, although much of the query language is a subset of SQL, path expressions are extensions not found in SQL.

Examples of Path Expressions

Here, the `WHERE` clause contains a `single_valued_path_expression`; the `p` is an identification variable, and `salary` is a persistent field of `Player`:

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

Here, the `WHERE` clause also contains a `single_valued_path_expression`; `t` is an identification variable, `league` is a single-valued relationship field, and `sport` is a persistent field of `league`:

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

Here, the `WHERE` clause contains a `collection_valued_path_expression`; `p` is an identification variable, and `teams` designates a collection-valued relationship field:

```
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

Expression Types

The type of a path expression is the type of the object represented by the ending element, which can be one of the following:

- Persistent field
- Single-valued relationship field
- Collection-valued relationship field

For example, the type of the expression `p.salary` is `double` because the terminating persistent field (`salary`) is a `double`.

In the expression `p.teams`, the terminating element is a collection-valued relationship field (`teams`). This expression's type is a collection of the abstract schema type named `Team`. Because `Team` is the abstract schema name for the `Team` entity, this type maps to the entity. For more information on the type mapping of abstract schemas, see Return Types.

Navigation

A path expression enables the query to navigate to related entities. The terminating elements of an expression determine whether navigation is allowed. If an expression contains a single-valued relationship field, the navigation can continue to an object that is related to the field. However, an expression cannot navigate beyond a persistent field or a collection-valued relationship field. For example, the expression `p.teams.league.sport` is illegal because `teams` is a collection-valued

relationship field. To reach the `sport` field, the `FROM` clause could define an identification variable named `t` for the `teams` field:

```
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

WHERE Clause

The `WHERE` clause specifies a conditional expression that limits the values returned by the query. The query returns all corresponding values in the data store for which the conditional expression is `TRUE`. Although usually specified, the `WHERE` clause is optional. If the `WHERE` clause is omitted, the query returns all values. The high-level syntax for the `WHERE` clause is as follows:

```
where_clause ::= WHERE conditional_expression
```

Literals

There are four kinds of literals: string, numeric, Boolean, and enum.

- String literals: A string literal is enclosed in single quotes:

  ```
  'Duke'
  ```

  If a string literal contains a single quote, you indicate the quote by using two single quotes:

  ```
  'Duke''s'
  ```

  Like a Java `String`, a string literal in the query language uses the Unicode character encoding.

- Numeric literals: There are two types of numeric literals: exact and approximate.
  - An exact numeric literal is a numeric value without a decimal point, such as 65, –233, and +12. Using the Java integer syntax, exact numeric literals support numbers in the range of a Java `long`.
  - An approximate numeric literal is a numeric value in scientific notation, such as 57., –85.7, and +2.1. Using the syntax of the Java floating-point literal, approximate numeric literals support numbers in the range of a Java `double`.
- Boolean literals: A Boolean literal is either `TRUE` or `FALSE`. These keywords are not case-sensitive.
- Enum literals: The Jakarta Persistence query language supports the use of enum literals using the Java enum literal syntax. The enum class name must be specified as a fully qualified class name:

  ```
  SELECT e
  FROM Employee e
  ```

```
WHERE e.status = com.example.EmployeeStatus.FULL_TIME
```

Input Parameters

An input parameter can be either a named parameter or a positional parameter.

- A named input parameter is designated by a colon (`:`) followed by a string; for example, `:name`.
- A positional input parameter is designated by a question mark (`?`) followed by an integer. For example, the first input parameter is `?1`, the second is `?2`, and so forth.

The following rules apply to input parameters.

- They can be used only in a `WHERE` or `HAVING` clause.
- Positional parameters must be numbered, starting with the integer 1.
- Named parameters and positional parameters may not be mixed in a single query.
- Named parameters are case-sensitive.

Conditional Expressions

A `WHERE` clause consists of a conditional expression, which is evaluated from left to right within a precedence level. You can change the order of evaluation by using parentheses.

Operators and Their Precedence

Query Language Order Precedence lists the query language operators in order of decreasing precedence.

Query Language Order Precedence

Type	Precedence Order
Navigation	`.` (a period)
Arithmetic	`+ -` (unary) `* /` (multiplication and division) `+ -` (addition and subtraction)

Type	Precedence Order
Comparison	= >= < ⇐ <> (not equal) [NOT] BETWEEN [NOT] LIKE [NOT] IN IS [NOT] NULL IS [NOT] EMPTY [NOT] MEMBER OF
Logical	NOT AND OR

BETWEEN Expressions

A BETWEEN expression determines whether an arithmetic expression falls within a range of values.

These two expressions are equivalent:

```
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
```

The following two expressions also are equivalent:

```
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

If an arithmetic expression has a NULL value, the value of the BETWEEN expression is unknown.

IN Expressions

An IN expression determines whether a string belongs to a set of string literals or whether a number belongs to a set of number values.

The path expression must have a string or numeric value. If the path expression has a NULL value, the value of the IN expression is unknown.

In the following example, the expression is TRUE if the country is UK , but FALSE if the country is Peru:

```
o.country IN ('UK', 'US', 'France')
```

You may also use input parameters:

```
o.country IN ('UK', 'US', 'France', :country)
```

LIKE Expressions

A LIKE expression determines whether a wildcard pattern matches a string.

The path expression must have a string or numeric value. If this value is NULL, the value of the LIKE expression is unknown. The pattern value is a string literal that can contain wildcard characters. The underscore (_) wildcard character represents any single character. The percent (%) wildcard character represents zero or more characters. The ESCAPE clause specifies an escape character for the wildcard characters in the pattern value. LIKE Expression Examples shows some sample LIKE expressions.

LIKE Expression Examples

Expression	TRUE	FALSE
`address.phone LIKE '12%3'`	`'123'` `'12993'`	`'1234'`
`asentence.word LIKE 'l_se'`	`'lose'`	`'loose'`
`aword.underscored LIKE '\_%' ESCAPE '\'`	`'_foo'`	`'bar'`
`address.phone NOT LIKE '12%3'`	`'1234'`	`'123'` `'12993'`

NULL Comparison Expressions

A NULL comparison expression tests whether a single-valued path expression or an input parameter has a NULL value. Usually, the NULL comparison expression is used to test whether a single-valued relationship has been set:

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

This query selects all teams where the league relationship is not set. Note that the following query is not equivalent:

```
SELECT t
FROM Team t
```

```
WHERE t.league = NULL
```

The comparison with `NULL` using the equals operator (`=`) always returns an unknown value, even if the relationship is not set. The second query will always return an empty result.

Empty Collection Comparison Expressions

The `IS [NOT] EMPTY` comparison expression tests whether a collection-valued path expression has no elements. In other words, it tests whether a collection-valued relationship has been set.

If the collection-valued path expression is `NULL`, the empty collection comparison expression has a `NULL` value.

Here is an example that finds all orders that do not have any line items:

```
SELECT o
FROM CustomerOrder o
WHERE o.lineItems IS EMPTY
```

Collection Member Expressions

The `[NOT] MEMBER [OF]` collection member expression determines whether a value is a member of a collection. The value and the collection members must have the same type.

If either the collection-valued or single-valued path expression is unknown, the collection member expression is unknown. If the collection-valued path expression designates an empty collection, the collection member expression is `FALSE`.

The `OF` keyword is optional.

The following example tests whether a line item is part of an order:

```
SELECT o
FROM CustomerOrder o
WHERE :lineItem MEMBER OF o.lineItems
```

Subqueries

Subqueries may be used in the `WHERE` or `HAVING` clause of a query. Subqueries must be surrounded by parentheses.

The following example finds all customers who have placed more than ten orders:

```
SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

Subqueries may contain EXISTS, ALL, and ANY expressions.

- EXISTS expressions: The [NOT] EXISTS expression is used with a subquery and is true only if the result of the subquery consists of one or more values; otherwise, it is false.

  The following example finds all employees whose spouses are also employees:

  ```
  SELECT DISTINCT emp
  FROM Employee emp
  WHERE EXISTS (
      SELECT spouseEmp
      FROM Employee spouseEmp
      WHERE spouseEmp = emp.spouse)
  ```

- ALL and ANY expressions: The ALL expression is used with a subquery and is true if all the values returned by the subquery are true or if the subquery is empty.

  The ANY expression is used with a subquery and is true if some of the values returned by the subquery are true. An ANY expression is false if the subquery result is empty or if all the values returned are false. The SOME keyword is synonymous with ANY.

  The ALL and ANY expressions are used with the =, <, <=, >, >=, and <> comparison operators.

  The following example finds all employees whose salaries are higher than the salaries of the managers in the employee's department:

  ```
  SELECT emp
  FROM Employee emp
  WHERE emp.salary > ALL (
      SELECT m.salary
      FROM Manager m
      WHERE m.department = emp.department)
  ```

Functional Expressions

The query language includes several string, arithmetic, and date/time functions that may be used in the SELECT, WHERE, or HAVING clause of a query. The functions are listed in String Expressions, Arithmetic Expressions and Date/Time Expressions.

In String Expressions, the start and length arguments are of type int and designate positions in the String argument. The first position in a string is designated by 1.

String Expressions

Function Syntax	Return Type
CONCAT(String, String)	String
LENGTH(String)	int
LOCATE(String, String [, start])	int

Function Syntax	Return Type
`SUBSTRING(String, start, length)`	`String`
`TRIM([[LEADING|TRAILING|BOTH] char FROM] String)`	`String`
`LOWER(String)`	`String`
`UPPER(String)`	`String`

The `CONCAT` function concatenates two strings into one string.

The `LENGTH` function returns the length of a string in characters as an integer.

The `LOCATE` function returns the position of a given string within a string. This function returns the first position at which the string was found as an integer. The first argument is the string to be located. The second argument is the string to be searched. The optional third argument is an integer that represents the starting string position. By default, `LOCATE` starts at the beginning of the string. The starting position of a string is `1`. If the string cannot be located, `LOCATE` returns `0`.

The `SUBSTRING` function returns a string that is a substring of the first argument based on the starting position and length.

The `TRIM` function trims the specified character from the beginning and/or end of a string. If no character is specified, `TRIM` removes spaces or blanks from the string. If the optional `LEADING` specification is used, `TRIM` removes only the leading characters from the string. If the optional `TRAILING` specification is used, `TRIM` removes only the trailing characters from the string. The default is `BOTH`, which removes the leading and trailing characters from the string.

The `LOWER` and `UPPER` functions convert a string to lowercase or uppercase, respectively.

In Arithmetic Expressions, the `number` argument can be an `int`, a `float`, or a `double`.

Arithmetic Expressions

Function Syntax	Return Type
`ABS(number)`	`int`, `float`, or `double`
`MOD(int, int)`	`int`
`SQRT(double)`	`double`
`SIZE(Collection)`	`int`

The `ABS` function takes a numeric expression and returns a number of the same type as the argument.

The `MOD` function returns the remainder of the first argument divided by the second.

The `SQRT` function returns the square root of a number.

The `SIZE` function returns an integer of the number of elements in the given collection.

In Date/Time Expressions, the date/time functions return the date, time, or timestamp on the database server.

Date/Time Expressions

Function Syntax	Return Type
CURRENT_DATE	java.sql.Date
CURRENT_TIME	java.sql.Time
CURRENT_TIMESTAMP	java.sql.Timestamp

Case Expressions

Case expressions change based on a condition, similar to the `case` keyword of the Java programming language. The `CASE` keyword indicates the start of a case expression, and the expression is terminated by the `END` keyword. The `WHEN` and `THEN` keywords define individual conditions, and the `ELSE` keyword defines the default condition should none of the other conditions be satisfied.

The following query selects the name of a person and a conditional string, depending on the subtype of the `Person` entity. If the subtype is `Student`, the string `kid` is returned. If the subtype is `Guardian` or `Staff`, the string `adult` is returned. If the entity is some other subtype of `Person`, the string `unknown` is returned:

```
SELECT p.name
CASE TYPE(p)
    WHEN Student THEN 'kid'
    WHEN Guardian THEN 'adult'
    WHEN Staff THEN 'adult'
    ELSE 'unknown'
END
FROM Person p
```

The following query sets a discount for various types of customers. Gold-level customers get a 20% discount, silver-level customers get a 15% discount, bronze-level customers get a 10% discount, and everyone else gets a 5% discount:

```
UPDATE Customer c
SET c.discount =
    CASE c.level
        WHEN 'Gold' THEN 20
        WHEN 'SILVER' THEN 15
        WHEN 'Bronze' THEN 10
        ELSE 5
    END
```

NULL Values

If the target of a reference is not in the persistent store, the target is `NULL`. For conditional expressions containing `NULL`, the query language uses the semantics defined by SQL92. Briefly, these semantics are as follows.

- If a comparison or arithmetic operation has an unknown value, it yields a `NULL` value.
- Two `NULL` values are not equal. Comparing two `NULL` values yields an unknown value.
- The `IS NULL` test converts a `NULL` persistent field or a single-valued relationship field to `TRUE`. The `IS NOT NULL` test converts them to `FALSE`.
- Boolean operators and conditional tests use the three-valued logic defined by AND Operator Logic and OR Operator Logic. (In these tables, T stands for `TRUE`, F for `FALSE`, and U for unknown.)

AND Operator Logic

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR Operator Logic

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Equality Semantics

In the query language, only values of the same type can be compared. However, this rule has one exception: Exact and approximate numeric values can be compared. In such a comparison, the required type conversion adheres to the rules of Java numeric promotion.

The query language treats compared values as if they were Java types and not as if they represented types in the underlying data store. For example, a persistent field that could be either an integer or a `NULL` must be designated as an `Integer` object and not as an `int` primitive. This designation is required because a Java object can be `NULL`, but a primitive cannot.

Two strings are equal only if they contain the same sequence of characters. Trailing blanks are significant; for example, the strings `'abc'` and `'abc '` are not equal.

Two entities of the same abstract schema type are equal only if their primary keys have the same value. NOT Operator Logic shows the operator logic of a negation, and Conditional Test shows the truth values of conditional tests.

NOT Operator Logic

NOT Value	Value
T	F
F	T
U	U

Conditional Test

Conditional Test	T	F	U
Expression `IS TRUE`	T	F	F
Expression `IS FALSE`	F	T	F
Expression is unknown	F	F	T

SELECT Clause

The `SELECT` clause defines the types of the objects or values returned by the query.

Return Types

The return type of the `SELECT` clause is defined by the result types of the select expressions contained within it. If multiple expressions are used, the result of the query is an `Object[]`, and the elements in the array correspond to the order of the expressions in the `SELECT` clause and in type to the result types of each expression.

A `SELECT` clause cannot specify a collection-valued expression. For example, the `SELECT` clause `p.teams` is invalid because `teams` is a collection. However, the clause in the following query is valid because `t` is a single element of the `teams` collection:

```
SELECT t
FROM Player p, IN (p.teams) t
```

The following query is an example of a query with multiple expressions in the `SELECT` clause:

```
SELECT c.name, c.country.name
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

This query returns a list of `Object[]` elements; the first array element is a string denoting the customer name, and the second array element is a string denoting the name of the customer's country.

The result of a query may be the result of an aggregate function, listed in Aggregate Functions in Select Statements.

Aggregate Functions in Select Statements

Name	Return Type	Description
`AVG`	`Double`	Returns the mean average of the fields
`COUNT`	`Long`	Returns the total number of results
`MAX`	The type of the field	Returns the highest value in the result set

Name	Return Type	Description
`MIN`	The type of the field	Returns the lowest value in the result set
`SUM`	`Long` (for integral fields) `Double` (for floating-point fields) `BigInteger` (for `BigInteger` fields) `BigDecimal` (for `BigDecimal` fields)	Returns the sum of all the values in the result set

For select method queries with an aggregate function (`AVG`, `COUNT`, `MAX`, `MIN`, or `SUM`) in the `SELECT` clause, the following rules apply.

- The `AVG`, `MAX`, `MIN`, and `SUM` functions return `null` if there are no values to which the function can be applied.
- The `COUNT` function returns 0 if there are no values to which the function can be applied.

The following example returns the average order quantity:

```
SELECT AVG(o.quantity)
FROM CustomerOrder o
```

The following example returns the total cost of the items ordered by Roxane Coss:

```
SELECT SUM(l.price)
FROM CustomerOrder o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

The following example returns the total number of orders:

```
SELECT COUNT(o)
FROM CustomerOrder o
```

The following example returns the total number of items that have prices in Hal Incandenza's order:

```
SELECT COUNT(l.price)
FROM CustomerOrder o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

The DISTINCT Keyword

The `DISTINCT` keyword eliminates duplicate return values. If a query returns a `java.util.Collection`,

which allows duplicates, you must specify the `DISTINCT` keyword to eliminate duplicates.

Constructor Expressions

Constructor expressions allow you to return Java instances that store a query result element instead of an `Object[]`.

The following query creates a `CustomerDetail` instance per `Customer` matching the `WHERE` clause. A `CustomerDetail` stores the customer name and customer's country name.So the query returns a `List` of `CustomerDetail` instances:

```
SELECT NEW com.example.CustomerDetail(c.name, c.country.name)
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

ORDER BY Clause

As its name suggests, the `ORDER BY` clause orders the values or objects returned by the query.

If the `ORDER BY` clause contains multiple elements, the left-to-right sequence of the elements determines the high-to-low precedence.

The `ASC` keyword specifies ascending order, the default, and the `DESC` keyword indicates descending order.

When using the `ORDER BY` clause, the `SELECT` clause must return an orderable set of objects or values. You cannot order the values or objects for values or objects not returned by the `SELECT` clause. For example, the following query is valid because the `ORDER BY` clause uses the objects returned by the `SELECT` clause:

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

The following example is not valid, because the `ORDER BY` clause uses a value not returned by the `SELECT` clause:

```
SELECT p.product_name
FROM CustomerOrder o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

GROUP BY and HAVING Clauses

The `GROUP BY` clause allows you to group values according to a set of properties.

The following query groups the customers by their country and returns the number of customers

per country:

```
SELECT c.country, COUNT(c)
FROM Customer c GROUP BY c.country
```

The `HAVING` clause is used with the `GROUP BY` clause to further restrict the returned result of a query.

The following query groups orders by the status of their customer and returns the customer status plus the average `totalPrice` for all orders where the corresponding customers have the same status. In addition, it considers only customers with status `1`, `2`, or `3`, so orders of other customers are not taken into account:

```
SELECT c.status, AVG(o.totalPrice)
FROM CustomerOrder o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

Using the Criteria API to Create Queries



This section was last updated for Jakarta EE Version 9.1

The Criteria API is used to define queries for entities and their persistent state by creating query-defining objects. Criteria queries are written using Java programming language APIs, are typesafe, and are portable. Such queries work regardless of the underlying data store.

Overview of the Criteria and Metamodel APIs

Similar to JPQL, the Criteria API is based on the abstract schema of persistent entities, their relationships, and embedded objects. The Criteria API operates on this abstract schema to allow developers to find, modify, and delete persistent entities by invoking Jakarta Persistence entity operations. The Metamodel API works in concert with the Criteria API to model persistent entity classes for Criteria queries.

The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries. Developers familiar with JPQL syntax will find equivalent object-level operations in the Criteria API.

The following simple Criteria query returns all instances of the `Pet` entity in the data source:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q.getResultList();
```

The equivalent JPQL query is

```
SELECT p
FROM Pet p
```

This query demonstrates the basic steps to create a Criteria query.

1. Use an `EntityManager` instance to create a `CriteriaBuilder` object.
2. Create a query object by creating an instance of the `CriteriaQuery` interface. This query object's attributes will be modified with the details of the query.
3. Set the query root by calling the `from` method on the `CriteriaQuery` object.
4. Specify what the type of the query result will be by calling the `select` method of the `CriteriaQuery` object.
5. Prepare the query for execution by creating a `TypedQuery<T>` instance, specifying the type of the query result.
6. Execute the query by calling the `getResultList` method on the `TypedQuery<T>` object. Because this query returns a collection of entities, the result is stored in a `List`.

The tasks associated with each step are discussed in detail in this chapter.

To create a `CriteriaBuilder` instance, call the `getCriteriaBuilder` method on the `EntityManager` instance:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Use the `CriteriaBuilder` instance to create a query object:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

The query will return instances of the `Pet` entity. To create a typesafe query, specify the type of the query when you create the `CriteriaQuery` object.

Call the `from` method of the query object to set the `FROM` clause of the query and to specify the root of the query:

```
Root<Pet> pet = cq.from(Pet.class);
```

Call the `select` method of the query object, passing in the query root, to set the `SELECT` clause of the query:

```
cq.select(pet);
```

Now, use the query object to create a `TypedQuery<T>` object that can be executed against the data source. The modifications to the query object are captured to create a ready-to-execute query:

```
TypedQuery<Pet> q = em.createQuery(cq);
```

Execute this typed query object by calling its `getResultList` method, because this query will return multiple entity instances. The following statement stores the results in a `List<Pet>` collection-valued object:

```
List<Pet> allPets = q.getResultList();
```

Using the Metamodel API to Model Entity Classes

Use the Metamodel API to create a metamodel of the managed entities in a particular persistence unit. For each entity class in a particular package, a metamodel class is created with a trailing underscore and with attributes that correspond to the persistent fields or properties of the entity class.

The following entity class, `com.example.Pet`, has four persistent fields: `id`, `name`, `color`, and `owners`:

```
package com.example;
...
@Entity
public class Pet {
    @Id
    protected Long id;
    protected String name;
    protected String color;
    @ManyToOne
    protected Set<Person> owners;
    ...
}
```

The corresponding Metamodel class is as follows:

```
package com.example;
...
@Static Metamodel(Pet.class)
public class Pet_ {

    public static volatile SingularAttribute<Pet, Long> id;
    public static volatile SingularAttribute<Pet, String> name;
    public static volatile SingularAttribute<Pet, String> color;
    public static volatile SetAttribute<Pet, Person> owners;
}
```

Criteria queries use the metamodel class and its attributes to refer to the managed entity classes and their persistent state and relationships.

Using Metamodel Classes

Metamodel classes that correspond to entity classes are of the following type:

```
jakarta.persistence.metamodel.EntityType<T>
```

Annotation processors typically generate metamodel classes either at development time or at runtime. Developers of applications that use Criteria queries may do either of the following:

- Generate static metamodel classes by using the persistence provider's annotation processor
- Obtain the metamodel class by doing one of the following:
  - Call the `getModel` method on the query root object
  - Obtain an instance of the `Metamodel` interface and then pass the entity type to the instance's `entity` method

The following code snippet shows how to obtain the `Pet` entity's metamodel class by calling `Root<T>.getModel`:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
EntityType<Pet> Pet_ = pet.getModel();
```

The following code snippet shows how to obtain the `Pet` entity's metamodel class by first obtaining a metamodel instance by using `EntityManager.getMetamodel` and then calling `entity` on the metamodel instance:

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
```



The most common use case is to generate typesafe static metamodel classes at development time. Obtaining the metamodel classes dynamically, by calling `Root<T>.getModel` or `EntityManager.getMetamodel` and then the `entity` method, doesn't allow for type safety and doesn't allow the application to call persistent field or property names on the metamodel class.

Using the Criteria API and Metamodel API to Create Basic Typesafe Queries

The basic semantics of a Criteria query consists of a `SELECT` clause, a `FROM` clause, and an optional `WHERE` clause, similar to a JPQL query. Criteria queries set these clauses by using Java programming

language objects, so the query can be created in a typesafe manner.

Creating a Criteria Query

The `jakarta.persistence.criteria.CriteriaBuilder` interface is used to construct

- Criteria queries
- Selections
- Expressions
- Predicates
- Ordering

To obtain an instance of the `CriteriaBuilder` interface, call the `getCriteriaBuilder` method on either an `EntityManager` or an `EntityManagerFactory` instance.

The following code shows how to obtain a `CriteriaBuilder` instance by using the `EntityManager.getCriteriaBuilder` method:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Criteria queries are constructed by obtaining an instance of the following interface:

```
jakarta.persistence.criteria.CriteriaQuery
```

`CriteriaQuery` objects define a particular query that will navigate over one or more entities. Obtain `CriteriaQuery` instances by calling one of the `CriteriaBuilder.createQuery` methods. To create typesafe queries, call the `CriteriaBuilder.createQuery` method as follows:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

The `CriteriaQuery` object's type should be set to the expected result type of the query. In the preceding code, the object's type is set to `CriteriaQuery<Pet>` for a query that will find instances of the `Pet` entity.

The following code snippet creates a `CriteriaQuery` object for a query that returns a `String`:

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
```

Query Roots

For a particular `CriteriaQuery` object, the root entity of the query, from which all navigation originates, is called the query root. It is similar to the `FROM` clause in a JPQL query.

Create the query root by calling the `from` method on the `CriteriaQuery` instance. The argument to the

`from` method is either the entity class or an `EntityType<T>` instance for the entity.

The following code sets the query root to the `Pet` entity:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

The following code sets the query root to the `Pet` class by using an `EntityType<T>` instance:

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet_);
```

Criteria queries may have more than one query root. This usually occurs when the query navigates from several entities.

The following code has two `Root` instances:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet1 = cq.from(Pet.class);
Root<Pet> pet2 = cq.from(Pet.class);
```

Querying Relationships Using Joins

For queries that navigate to related entity classes, the query must define a join to the related entity by calling one of the `From.join` methods on the query root object or another join object. The `join` methods are similar to the `JOIN` keyword in JPQL.

The target of the join uses the Metamodel class of type `EntityType<T>` to specify the persistent field or property of the joined entity.

The `join` methods return an object of type `Join<X, Y>`, where `X` is the source entity and `Y` is the target of the join. In the following code snippet, `Pet` is the source entity, `Owner` is the target, and `Pet_` is a statically generated metamodel class:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

You can chain joins together to navigate to related entities of the target entity without having to create a `Join<X, Y>` instance for each join:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

```
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.addresses);
```

Path Navigation in Criteria Queries

`Path` objects, which are used in the `SELECT` and `WHERE` clauses of a Criteria query, can be query root entities, join entities, or other `Path` objects. Use the `Path.get` method to navigate to attributes of the entities of a query.

The argument to the `get` method is the corresponding attribute of the entity's Metamodel class. The attribute can be either a single-valued attribute, specified by `@SingularAttribute` in the Metamodel class, or a collection-valued attribute, specified by one of `@CollectionAttribute`, `@SetAttribute`, `@ListAttribute`, or `@MapAttribute`.

The following query returns the names of all the pets in the data store. The `get` method is called on the query root, `pet`, with the `name` attribute of the `Pet` entity's Metamodel class, `Pet_`, as the argument:

```
CriteriaQuery<String> cq = cb.createQuery(String.class);

Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

Restricting Criteria Query Results

Conditions that are set by calling the `CriteriaQuery.where` method can restrict the results of a query on the `CriteriaQuery` object. Calling the `where` method is analogous to setting the `WHERE` clause in a JPQL query.

The `where` method evaluates instances of the `Expression` interface to restrict the results according to the conditions of the expressions. To create `Expression` instances, use methods defined in the `Expression` and `CriteriaBuilder` interfaces.

The Expression Interface Methods

An `Expression` object is used in a query's `SELECT`, `WHERE`, or `HAVING` clause. Conditional Methods in the Expression Interface shows conditional methods you can use with `Expression` objects.

Conditional Methods in the Expression Interface

Method	Description
`isNull`	Tests whether an expression is null
`isNotNull`	Tests whether an expression is not null
`in`	Tests whether an expression is within a list of values

The following query uses the `Expression.isNull` method to find all pets where the `color` attribute is null:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).isNull());
```

The following query uses the `Expression.in` method to find all brown and black pets:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in("brown", "black"));
```

The `in` method can also check whether an attribute is a member of a collection.

Expression Methods in the CriteriaBuilder Interface

The `CriteriaBuilder` interface defines additional methods for creating expressions. These methods correspond to the arithmetic, string, date, time, and case operators and functions of JPQL. Conditional Methods in the CriteriaBuilder Interface shows conditional methods you can use with `CriteriaBuilder` objects.

Conditional Methods in the CriteriaBuilder Interface

Conditional Method	Description
`equal`	Tests whether two expressions are equal
`notEqual`	Tests whether two expressions are not equal
`gt`	Tests whether the first numeric expression is greater than the second numeric expression
`ge`	Tests whether the first numeric expression is greater than or equal to the second numeric expression
`lt`	Tests whether the first numeric expression is less than the second numeric expression
`le`	Tests whether the first numeric expression is less than or equal to the second numeric expression
`between`	Tests whether the first expression is between the second and third expression in value

Conditional Method	Description
`like`	Tests whether the expression matches a given pattern

The following code uses the `CriteriaBuilder.equal` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

The following code uses the `CriteriaBuilder.gt` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date));
```

The following code uses the `CriteriaBuilder.between` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

The following code uses the `CriteriaBuilder.like` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.like(pet.get(Pet_.name), "*do"));
```

To specify multiple conditional predicates, use the compound predicate methods of the `CriteriaBuilder` interface, as shown in Compound Predicate Methods in the CriteriaBuilder Interface.

Compound Predicate Methods in the CriteriaBuilder Interface

Method	Description
`and`	A logical conjunction of two Boolean expressions
`or`	A logical disjunction of two Boolean expressions

Method	Description
`not`	A logical negation of the given Boolean expression

The following code shows the use of compound predicates in queries:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido")
        .and(cb.equal(pet.get(Pet_.color), "brown")));
```

Managing Criteria Query Results

For queries that return more than one result, it is often helpful to organize those results. The `CriteriaQuery` interface defines the following ordering and grouping methods:

- The `orderBy` method orders query results according to attributes of an entity
- The `groupBy` method groups the results of a query together according to attributes of an entity, and the `having` method restricts those groups according to a condition

Ordering Results

To order the results of a query, call the `CriteriaQuery.orderBy` method, passing in an `Order` object. To create an `Order` object, call either the `CriteriaBuilder.asc` or the `CriteriaBuilder.desc` method. The `asc` method is used to order the results by ascending value of the passed expression parameter. The `desc` method is used to order the results by descending value of the passed expression parameter. The following query shows the use of the `desc` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get(Pet_.birthday)));
```

In this query, the results will be ordered by the pet's birthday from highest to lowest. That is, pets born in December will appear before pets born in May.

The following query shows the use of the `asc` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc(address.get(Address_.postalCode)));
```

In this query, the results will be ordered by the pet owner's postal code from lowest to highest. That is, pets whose owner lives in the 10001 zip code will appear before pets whose owner lives in the

91000 zip code.

If more than one `Order` object is passed to `orderBy`, the precedence is determined by the order in which they appear in the argument list of `orderBy`. The first `Order` object has precedence.

The following code orders results by multiple criteria:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName)), owner.get(Owner_.firstName)));
```

The results of this query will be ordered alphabetically by the pet owner's last name, then first name.

Grouping Results

The `CriteriaQuery.groupBy` method partitions the query results into groups. To set these groups, pass an expression to `groupBy`:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
```

This query returns all `Pet` entities and groups the results by the pet's color.

Use the `CriteriaQuery.having` method in conjunction with `groupBy` to filter over the groups. The `having` method, which takes a conditional expression as a parameter, restricts the query result according to the conditional expression:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).value("brown").value("blonde"));
```

In this example, the query groups the returned `Pet` entities by color, as in the preceding example. However, the only returned groups will be `Pet` entities where the `color` attribute is set to `brown` or `blonde`. That is, no gray-colored pets will be returned in this query.

Executing Queries

To prepare a query for execution, create a `TypedQuery<T>` object with the type of the query result, passing the `CriteriaQuery` object to `EntityManager.createQuery`.

To execute a query, call either `getSingleResult` or `getResultList` on the `TypedQuery<T>` object.

Single-Valued Query Results

Use the `TypedQuery<T>.getSingleResult` method to execute queries that return a single result:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();
```

Collection-Valued Query Results

Use the `TypedQuery<T>.getResultList` method to execute queries that return a collection of objects:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Creating and Using String-Based Criteria Queries



This section was last updated for Jakarta EE Version 9.1

This chapter describes how to create weakly typed string-based Criteria API queries.

Overview of String-Based Criteria API Queries

String-based Criteria API queries ("string-based queries") are Java programming language queries that use strings rather than strongly typed metamodel objects to specify entity attributes when traversing a data hierarchy. String-based queries are constructed similarly to metamodel queries, can be static or dynamic, and can express the same kind of queries and operations as strongly typed metamodel queries.

Strongly typed metamodel queries are the preferred method of constructing Criteria API queries.

The main advantage of string-based queries over metamodel queries is the ability to construct Criteria queries at development time without the need to generate static metamodel classes or otherwise access dynamically generated metamodel classes.

The main disadvantage to string-based queries is their lack of type safety; this problem may lead to runtime errors due to type mismatches that would be caught at development time if you used strongly typed metamodel queries.

For information on constructing criteria queries, see [persist:persistence-criteria::persistence-criteria:::_using_the_criteria_api_to_create_queries].

Creating String-Based Queries

To create a string-based query, specify the attribute names of entity classes directly as strings,

instead of specifying the attributes of the metamodel class. For example, this query finds all `Pet` entities where the value of the `name` attribute is `Fido`:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
```

The name of the attribute is specified as a string. This query is the equivalent of the following metamodel query:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```



Type mismatch errors in string-based queries will not appear until the code is executed at runtime, unlike in the above metamodel query, where type mismatches will be caught at compile time.

Joins are specified in the same way:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join("owners").join("addresses");
```

All the conditional expressions, method expressions, path navigation methods, and result restriction methods used in metamodel queries can also be used in string-based queries. In each case, the attributes are specified using strings. For example, here is a string-based query that uses the `in` expression:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get("color").in("brown", "black"));
```

Here is a string-based query that orders the results in descending order by date:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get("birthday")));
```

Executing String-Based Queries

String-based queries are executed similarly to strongly typed Criteria queries. First create a `jakarta.persistence.TypedQuery` object by passing the criteria query object to the `EntityManager.createQuery` method, then call either `getSingleResult` or `getResultList` on the query object to execute the query:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```

Controlling Concurrent Access to Entity Data with Locking



This section was last updated for Jakarta EE Version 9.1

This chapter details how to handle concurrent access to entity data, and the locking strategies available to Jakarta Persistence application developers.

Overview of Entity Locking and Concurrency

Entity data is concurrently accessed if the data in a data source is accessed at the same time by multiple applications. Ensure that the underlying data's integrity is preserved when it is accessed concurrently.

When data is updated in the database tables in a transaction, the persistence provider assumes the database management system will hold short-term read locks and long-term write locks to maintain data integrity. Most persistence providers will delay database writes until the end of the transaction, except when the application explicitly calls for a flush (that is, the application calls the `EntityManager.flush` method or executes queries with the flush mode set to `AUTO`).

By default, persistence providers use optimistic locking, where, before committing changes to the data, the persistence provider checks that no other transaction has modified or deleted the data since the data was read. This is accomplished by a version column in the database table, with a corresponding version attribute in the entity class. When a row is modified, the version value is incremented. The original transaction checks the version attribute, and if the data has been modified by another transaction, a `jakarta.persistence.OptimisticLockException` will be thrown, and the original transaction will be rolled back. When the application specifies optimistic lock modes, the persistence provider verifies that a particular entity has not changed since it was read from the database even if the entity data was not modified.

Pessimistic locking goes further than optimistic locking. With pessimistic locking, the persistence provider creates a transaction that obtains a long-term lock on the data until the transaction is completed, which prevents other transactions from modifying or deleting the data until the lock has ended. Pessimistic locking is a better strategy than optimistic locking when the underlying data is frequently accessed and modified by many transactions.



Using pessimistic locks on entities that are not subject to frequent modification may result in decreased application performance.

Using Optimistic Locking

Use the `jakarta.persistence.Version` annotation to mark a persistent field or property as a version attribute of an entity. The version attribute enables the entity for optimistic concurrency control. The persistence provider reads and updates the version attribute when an entity instance is modified during a transaction. The application may read the version attribute, but must not modify the value.



Although some persistence providers may support optimistic locking for entities that do not have version attributes, portable applications should always use entities with version attributes when using optimistic locking. If the application attempts to lock an entity that does not have a version attribute, and the persistence provider does not support optimistic locking for non-versioned entities, a `PersistenceException` will be thrown.

The `@Version` annotation has the following requirements.

- Only a single `@Version` attribute may be defined per entity.
- The `@Version` attribute must be in the primary table for an entity mapped to multiple tables.
- The type of the `@Version` attribute must be one of the following: `int`, `Integer`, `long`, `Long`, `short`, `Short`, or `java.sql.Timestamp`.

The following code snippet shows how to define a version attribute in an entity with persistent fields:

```
@Version
protected int version;
```

The following code snippet shows how to define a version attribute in an entity with persistent properties:

```
@Version
protected Short getVersion() { ... }
```

Lock Modes

The application may increase the level of locking for an entity by specifying the use of lock modes. Lock modes may be specified to increase the level of optimistic locking or to request the use of pessimistic locks.

The use of optimistic lock modes causes the persistence provider to check the version attributes for entities that were read (but not modified) during a transaction as well as for entities that were updated.

The use of pessimistic lock modes specifies that the persistence provider is to immediately acquire long-term read or write locks for the database data corresponding to entity state.

You can set the lock mode for an entity operation by specifying one of the lock modes defined in the `jakarta.persistence.LockModeType` enumerated type, listed in Lock Modes for Concurrent Entity Access.

Lock Modes for Concurrent Entity Access

Lock Mode	Description
`OPTIMISTIC`	Obtain an optimistic read lock for all entities with version attributes.
`OPTIMISTIC_FORCE_INCREMENT`	Obtain an optimistic read lock for all entities with version attributes, and increment the version attribute value.
`PESSIMISTIC_READ`	Immediately obtain a long-term read lock on the data to prevent the data from being modified or deleted. Other transactions may read the data while the lock is maintained, but may not modify or delete the data. + The persistence provider is permitted to obtain a database write lock when a read lock was requested, but not vice versa.
`PESSIMISTIC_WRITE`	Immediately obtain a long-term write lock on the data to prevent the data from being read, modified, or deleted.
`PESSIMISTIC_FORCE_INCREMENT`	Immediately obtain a long-term lock on the data to prevent the data from being modified or deleted, and increment the version attribute of versioned entities.
`READ`	A synonym for `OPTIMISTIC`. Use of `LockModeType.OPTIMISTIC` is to be preferred for new applications.
`WRITE`	A synonym for `OPTIMISTIC_FORCE_INCREMENT`. Use of `LockModeType.OPTIMISTIC_FORCE_INCREMENT` is to be preferred for new applications.
`NONE`	No additional locking will occur on the data in the database.

Setting the Lock Mode

To specify the lock mode, use one of the following techniques:

1. Call the `EntityManager.lock` method, passing in one of the lock modes:

```
EntityManager em = ...;
```

```
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);
```

2. Call one of the `EntityManager.find` methods that take the lock mode as a parameter:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK,
    LockModeType.PESSIMISTIC_WRITE);
```

3. Call one of the `EntityManager.refresh` methods that take the lock mode as a parameter:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

4. Call the `Query.setLockMode` or `TypedQuery.setLockMode` method, passing the lock mode as the parameter:

```
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

5. Add a `lockMode` element to the `@NamedQuery` annotation:

```
@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
```

Using Pessimistic Locking

Versioned entities, as well as entities that do not have version attributes, can be locked pessimistically.

To lock entities pessimistically, set the lock mode to `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE`, or `PESSIMISTIC_FORCE_INCREMENT`.

If a pessimistic lock cannot be obtained on the database rows, and the failure to lock the data results in a transaction rollback, a `PessimisticLockException` is thrown. If a pessimistic lock cannot be obtained, but the locking failure doesn't result in a transaction rollback, a `LockTimeoutException` is thrown.

Pessimistically locking a versioned entity with `PESSIMISTIC_FORCE_INCREMENT` results in the version attribute being incremented even if the entity data is unmodified. When pessimistically locking a

versioned entity, the persistence provider will perform the version checks that occur during optimistic locking, and if the version check fails, an `OptimisticLockException` will be thrown. An attempt to lock a non-versioned entity with `PESSIMISTIC_FORCE_INCREMENT` is not portable and may result in a `PersistenceException` if the persistence provider does not support optimistic locks for non-versioned entities. Locking a versioned entity with `PESSIMISTIC_WRITE` results in the version attribute being incremented if the transaction was successfully committed.

Pessimistic Locking Timeouts

Use the `jakarta.persistence.lock.timeout` property to specify the length of time in milliseconds the persistence provider should wait to obtain a lock on the database tables. If the time it takes to obtain a lock exceeds the value of this property, a `LockTimeoutException` will be thrown, but the current transaction will not be marked for rollback. If you set this property to `0`, the persistence provider should throw a `LockTimeoutException` if it cannot immediately obtain a lock.



Portable applications should not rely on the setting of `jakarta.persistence.lock.timeout`, because the locking strategy and underlying database may mean that the timeout value cannot be used. The value of `jakarta.persistence.lock.timeout` is a hint, not a contract.

This property may be set programmatically by passing it to the `EntityManager` methods that allow lock modes to be specified, the `Query.setLockMode` and `TypedQuery.setLockMode` methods, the `@NamedQuery` annotation, and the `Persistence.createEntityManagerFactory` method. It may also be set as a property in the `persistence.xml` deployment descriptor.

If `jakarta.persistence.lock.timeout` is set in multiple places, the value will be determined in the following order:

1. The argument to one of the `EntityManager` or `Query` methods
2. The setting in the `@NamedQuery` annotation
3. The argument to the `Persistence.createEntityManagerFactory` method
4. The value in the `persistence.xml` deployment descriptor

Creating Fetch Plans with Entity Graphs



This section was last updated for Jakarta EE Version 9.1

This chapter explains how to use entity graphs to create fetch plans for Jakarta Persistence operations and queries.

Overview of Using Fetch Plans and Entity Graphs

Entity graphs are templates for a particular Persistence query or operation. They are used when creating fetch plans, or groups of persistent fields that are retrieved at the same time. Application developers use fetch plans to group together related persistent fields to improve runtime performance.

By default, entity fields or properties are fetched lazily. Developers specify fields or properties as

part of a fetch plan, and the persistence provider will fetch them eagerly.

For example, an email application that stores messages as `EmailMessage` entities prioritizes fetching some fields over others. The sender, subject, and date will be viewed the most often, in mailbox views and when the message is displayed. The `EmailMessage` entity has a collection of related `EmailAttachment` entities. For performance reasons the attachments should not be fetched until they are needed, but the file names of the attachment are important. A developer working on this application might make a fetch plan that eagerly fetches the important fields from `EmailMessage` and `EmailAttachment` while fetching the lower priority data lazily.

Entity Graph Basics

You can create entity graphs statically by using annotations or a deployment descriptor, or dynamically by using standard interfaces.

You can use an entity graph with the `EntityManager.find` method or as part of a JPQL or Criteria API query by specifying the entity graph as a hint to the operation or query.

Entity graphs have attributes that correspond to the fields that will be eagerly fetched during a `find` or query operation. The primary key and version fields of the entity class are always fetched and do not need to be explicitly added to an entity graph.

The Default Entity Graph

By default, all fields in an entity are fetched lazily unless the `fetch` attribute of the entity metadata is set to `jakarta.persistence.FetchType.EAGER`. The default entity graph consists of all the fields of an entity whose fields are set to be eagerly fetched.

For example, the following `EmailMessage` entity specifies that some fields will be fetched eagerly:

```
@Entity
public class EmailMessage implements Serializable {
    @Id
    String messageId;
    @Basic(fetch=EAGER)
    String subject;
    String body;
    @Basic(fetch=EAGER)
    String sender;
    @OneToMany(mappedBy="message", fetch=LAZY)
    Set<EmailAttachment> attachments;
    ...
}
```

The default entity graph for this entity would contain the `messageId`, `subject`, and `sender` fields, but not the `body` or `attachments` fields.

Using Entity Graphs in Persistence Operations

Entity graphs are used by creating an instance of the `jakarta.persistence.EntityGraph` interface by

calling either `EntityManager.getEntityGraph` for named entity graphs or `EntityManager.createEntityGraph` for creating dynamic entity graphs.

A named entity graph is an entity graph specified by the `@NamedEntityGraph` annotation applied to entity classes, or the `named-entity-graph` element in the application's deployment descriptors. Named entity graphs defined within the deployment descriptor override any annotation-based entity graphs with the same name.

The created entity graph can be either a fetch graph or a load graph.

Fetch Graphs

To specify a fetch graph, set the `jakarta.persistence.fetchgraph` property when you execute an `EntityManager.find` or query operation. A fetch graph consists of only the fields explicitly specified in the `EntityGraph` instance, and ignores the default entity graph settings.

In the following example, the default entity graph is ignored, and only the `body` field is included in the dynamically created fetch graph:

```
EntityGraph<EmailMessage> eg = em.createEntityGraph(EmailMessage.class);
eg.addAttributeNodes("body");
...
Properties props = new Properties();
props.put("jakarta.persistence.fetchgraph", eg);
EmailMessage message = em.find(EmailMessage.class, id, props);
```

Load Graphs

To specify a load graph, set the `jakarta.persistence.loadgraph` property when you execute an `EntityManager.find` or query operation. A load graph consists of the fields explicitly specified in the `EntityGraph` instance plus any fields in the default entity graph.

In the following example, the dynamically created load graph contains all the fields in the default entity graph plus the `body` field:

```
EntityGraph<EmailMessage> eg = em.createEntityGraph(EmailMessage.class);
eg.addAttributeNodes("body");
...
Properties props = new Properties();
props.put("jakarta.persistence.loadgraph", eg);
EmailMessage message = em.find(EmailMessage.class, id, props);
```

Using Named Entity Graphs

Named entity graphs are created using annotations applied to entity classes or the `named-entity-graph` element and its sub-elements in the application's deployment descriptor. The persistence provider will scan for all named entity graphs, defined in both annotations and in XML, within an application. A named entity graph set using an annotation may be overridden using `named-entity-`

graph.

Applying Named Entity Graph Annotations to Entity Classes

The jakarta.persistence.NamedEntityGraph annotation defines a single named entity graph and is applied at the class level. Multiple @NamedEntityGraph annotations may be defined for a class by adding them within a jakarta.persistence.NamedEntityGraphs class-level annotation.

The @NamedEntityGraph annotation must be applied on the root of the graph of entities. That is, if the EntityManager.find or query operation has as its root entity the EmailMessage class, the named entity graph used in the operation must be defined in the EmailMessage class:

```
@NamedEntityGraph
@Entity
public class EmailMessage {
    @Id
    String messageId;
    String subject;
    String body;
    String sender;
}
```

In this example, the EmailMessage class has a @NamedEntityGraph annotation to define a named entity graph that defaults to the name of the class, EmailMessage. No fields are included in the @NamedEntityGraph annotation as attribute nodes, and the fields are not annotated with metadata to set the fetch type, so the only field that will be eagerly fetched in either a load graph or fetch graph is messageId.

The attributes of a named entity graph are the fields of the entity that should be included in the entity graph. Add the fields to the entity graph by specifying them in the attributeNodes element of @NamedEntityGraph with a jakarta.persistence.NamedAttributeNode annotation:

```
@NamedEntityGraph(name="emailEntityGraph", attributeNodes={
    @NamedAttributeNode("subject"),
    @NamedAttributeNode("sender")
})
@Entity
public class EmailMessage { ... }
```

In this example, the name of the named entity graph is emailEntityGraph and includes the subject and sender fields.

Multiple @NamedEntityGraph definitions may be applied to a class by grouping them within a @NamedEntityGraphs annotation.

In the following example, two entity graphs are defined on the EmailMessage class. One is for a preview pane, which fetches only the sender, subject, and body of the message. The other is for a full view of the message, including any message attachments:

```
@NamedEntityGraphs({
    @NamedEntityGraph(name="previewEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("body")
    }),
    @NamedEntityGraph(name="fullEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("body"),
        @NamedAttributeNode("attachments")
    })
})
@Entity
public class EmailMessage { ... }
```

Obtaining EntityGraph Instances from Named Entity Graphs

Use the `EntityManager.getEntityGraph` method, passing in the named entity graph name, to obtain `EntityGraph` instances for a named entity graph:

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("emailEntityGraph");
```

Using Entity Graphs in Query Operations

To specify entity graphs for both typed and untyped queries, call the `setHint` method on the query object and specify either `jakarta.persistence.loadgraph` or `jakarta.persistence.fetchgraph` as the property name and an `EntityGraph` instance as the value:

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("previewEmailEntityGraph");
List<EmailMessage> messages = em.createNamedQuery("findAllEmailMessages")
        .setParameter("mailbox", "inbox")
        .setHint("jakarta.persistence.loadgraph", eg)
        .getResultList();
```

In this example, the `previewEmailEntityGraph` is used for the `findAllEmailMessages` named query.

Typed queries use the same technique:

```
EntityGraph<EmailMessage> eg = em.getEntityGraph("previewEmailEntityGraph");

CriteriaQuery<EmailMessage> cq = cb.createQuery(EmailMessage.class);
Root<EmailMessage> message = cq.from(EmailMessage.class);
TypedQuery<EmailMessage> q = em.createQuery(cq);
q.setHint("jakarta.persistence.loadgraph", eg);
List<EmailMessage> messages = q.getResultList();
```

Using a Second-Level Cache with Jakarta Persistence Applications



This section was last updated for Jakarta EE Version 9.1

This chapter explains how to modify the second-level cache mode settings to improve the performance of applications that use the Jakarta Persistence.

Overview of the Second-Level Cache

A second-level cache is a local store of entity data managed by the persistence provider to improve application performance. A second-level cache helps improve performance by avoiding expensive database calls, keeping the entity data local to the application. A second-level cache is typically transparent to the application, as it is managed by the persistence provider and underlies the persistence context of an application. That is, the application reads and commits data through the normal entity manager operations without knowing about the cache.



Persistence providers are not required to support a second-level cache. Portable applications should not rely on support by persistence providers for a second-level cache.

The second-level cache for a persistence unit may be configured to one of several second-level cache modes. The following cache mode settings are defined by Jakarta Persistence.

Cache Mode Settings for the Second-Level Cache

Cache Mode Setting	Description
`ALL`	All entity data is stored in the second-level cache for this persistence unit.
`NONE`	No data is cached in the persistence unit. The persistence provider must not cache any data.
`ENABLE_SELECTIVE`	Enable caching for entities that have been explicitly set with the `@Cacheable` annotation.
`DISABLE_SELECTIVE`	Enable caching for all entities except those that have been explicitly set with the `@Cacheable(false)` annotation.
`UNSPECIFIED`	The caching behavior for the persistence unit is undefined. The persistence provider's default caching behavior will be used.

One consequence of using a second-level cache in an application is that the underlying data may have changed in the database tables, while the value in the cache has not, a circumstance called a stale read. To avoid stale reads, use any of these strategies:

- Change the second-level cache to one of the cache mode settings
- Control which entities may be cached (see Controlling whether Entities May Be Cached)
- Change the cache's retrieval or store modes (see Setting the Cache Retrieval and Store Modes)

Which of these strategies works best to avoid stale reads depends upon the application.

Controlling whether Entities May Be Cached

The `jakarta.persistence.Cacheable` annotation is used to specify that an entity class, and any subclasses, may be cached when using the `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE` cache modes. Subclasses may override the `@Cacheable` setting by adding a `@Cacheable` annotation and changing the value.

To specify that an entity may be cached, add a `@Cacheable` annotation at the class level:

```
@Cacheable
@Entity
public class Person { ... }
```

By default, the `@Cacheable` annotation is `true`. The following example is equivalent:

```
@Cacheable(true)
@Entity
public class Person{ ... }
```

To specify that an entity must not be cached, add a `@Cacheable` annotation and set it to `false`:

```
@Cacheable(false)
@Entity
public class OrderStatus { ... }
```

When the `ENABLE_SELECTIVE` cache mode is set, the persistence provider will cache any entities that have the `@Cacheable(true)` annotation and any subclasses of that entity that have not been overridden. The persistence provider will not cache entities that have `@Cacheable(false)` or have no `@Cacheable` annotation. That is, the `ENABLE_SELECTIVE` mode will cache only entities that have been explicitly marked for the cache using the `@Cacheable` annotation.

When the `DISABLE_SELECTIVE` cache mode is set, the persistence provider will cache any entities that do not have the `@Cacheable(false)` annotation. Entities that do not have `@Cacheable` annotations, and entities with the `@Cacheable(true)` annotation, will be cached. That is, the `DISABLE_SELECTIVE` mode will cache all entities that have not been explicitly prevented from being cached.

If the cache mode is set to `UNDEFINED`, or is left unset, the behavior of entities annotated with `@Cacheable` is undefined. If the cache mode is set to `ALL` or `NONE`, the value of the `@Cacheable` annotation is ignored by the persistence provider.

Specifying the Cache Mode Settings to Improve Performance

To adjust the cache mode settings for a persistence unit, specify one of the cache modes as the value of the `shared-cache-mode` element in the `persistence.xml` deployment descriptor (shown in bold):

```
<persistence-unit name="examplePU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```



Because support for a second-level cache is not required by the Jakarta Persistence specification, setting the second-level cache mode in `persistence.xml` will have no effect when you use a persistence provider that does not implement a second-level cache.

Alternatively, you can specify the shared cache mode by setting the `jakarta.persistence.sharedCache.mode` property to one of the shared cache mode settings:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(
        "myExamplePU", new Properties().add(
            "jakarta.persistence.sharedCache.mode", "ENABLE_SELECTIVE"));
```

Setting the Cache Retrieval and Store Modes

If you have enabled the second-level cache for a persistence unit by setting the shared cache mode, you can further modify the behavior of the second-level cache by setting the `jakarta.persistence.cache.retrieveMode` and `jakarta.persistence.cache.storeMode` properties. You can set these properties at the persistence context level by passing the property name and value to the `EntityManager.setProperty` method, or you can set them on a per-`EntityManager` operation (`EntityManager.find` or `EntityManager.refresh`) or on a per-query level.

Cache Retrieval Mode

The cache retrieval mode, set by the `jakarta.persistence.retrieveMode` property, controls how data is read from the cache for calls to the `EntityManager.find` method and from queries.

You can set the `retrieveMode` property to one of the constants defined by the `jakarta.persistence.CacheRetrieveMode` enumerated type, either `USE` (the default) or `BYPASS`.

When the property is set to `USE`, data is retrieved from the second-level cache, if available. If the data is not in the cache, the persistence provider will read it from the database.

When the property is set to `BYPASS`, the second-level cache is bypassed and a call to the database is made to retrieve the data.

Cache Store Mode

The cache store mode, set by the `jakarta.persistence.storeMode` property, controls how data is stored in the cache.

The `storeMode` property can be set to one of the constants defined by the

`jakarta.persistence.CacheStoreMode` enumerated type: either `USE` (the default), `BYPASS`, or `REFRESH`.

When the property is set to `USE`, the cache data is created or updated when data is read from or committed to the database. If data is already in the cache, setting the store mode to `USE` will not force a refresh when data is read from the database.

When the property is set to `BYPASS`, data read from or committed to the database is not inserted or updated in the cache. That is, the cache is unchanged.

When the property is set to `REFRESH`, the cache data is created or updated when data is read from or committed to the database, and a refresh is forced on data in the cache upon database reads.

Setting the Cache Retrieval or Store Mode

To set the cache retrieval or store mode for the persistence context, call the `EntityManager.setProperty` method with the property name and value pair:

```
EntityManager em = ...;
em.setProperty("jakarta.persistence.cache.storeMode", "BYPASS");
```

To set the cache retrieval or store mode when calling the `EntityManager.find` or `EntityManager.refresh` methods, first create a `Map<String, Object>` instance and add a name/value pair as follows:

```
EntityManager em = ...;
Map<String, Object> props = new HashMap<String, Object>();
props.put("jakarta.persistence.cache.retrieveMode", "BYPASS");
String personPK = ...;
Person person = em.find(Person.class, personPK, props);
```



The cache retrieval mode is ignored when calling the `EntityManager.refresh` method, as calls to `refresh` always result in data being read from the database, not the cache.

To set the retrieval or store mode when using queries, call the `Query.setHint` or `TypedQuery.setHint` methods, depending on the type of query:

```
EntityManager em = ...;
CriteriaQuery<Person> cq = ...;
TypedQuery<Person> q = em.createQuery(cq);
q.setHint("jakarta.persistence.cache.storeMode", "REFRESH");
...
```

Setting the store or retrieve mode in a query or when calling the `EntityManager.find` or `EntityManager.refresh` method overrides the setting of the entity manager.

Controlling the Second-Level Cache Programmatically

The `jakarta.persistence.Cache` interface defines methods for interacting with the second-level cache programmatically.

Overview of the jakarta.persistence.Cache Interface

The `Cache` interface defines methods to do the following:

- Check whether a particular entity has cached data
- Remove a particular entity from the cache
- Remove all instances (and instances of subclasses) of an entity class from the cache
- Clear the cache of all entity data



If the second-level cache has been disabled, calls to the `Cache` interface's methods have no effect, except for `contains`, which will always return `false`.

Checking whether an Entity's Data Is Cached

To find out whether a given entity is currently in the second-level cache:

1. Call the `Cache.contains` method . The `contains` method returns `true` if the entity's data is cached, and `false` if the data is not in the cache:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
if (cache.contains(Person.class, personPK)) {
  // the data is cached
} else {
  // the data is NOT cached
}
```

Removing an Entity from the Cache

To remove a particular entity or all entities of a given type from the second-level cache:

1. Call one of the `Cache.evict` methods.
   a. To remove a particular entity from the cache, call the `evict` method and pass in the entity class and the primary key of the entity:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...;
cache.evict(Person.class, personPK);
```

   b. To remove all instances of a particular entity class, including subclasses, call the `evict`

method and specify the entity class:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evict(Person.class);
```

All instances of the `Person` entity class will be removed from the cache. If the `Person` entity has a subclass, `Student`, calls to the above method will remove all instances of `Student` from the cache as well.

Removing All Data from the Cache

To completely clear the second-level cache, call the `Cache.evictAll` method:

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evictAll();
```

Jakarta Enterprise Beans Lite

Enterprise Beans



This section was last updated for Jakarta EE Version 9.1

Enterprise beans are Jakarta EE components that implement Jakarta Enterprise Beans technology. Enterprise beans run in the Enterprise Bean container, a runtime environment within GlassFish Server (see Container Types). Although transparent to the application developer, the Enterprise Bean container provides system-level services, such as transactions and security, to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional Jakarta EE applications.

What Is an Enterprise Bean?

Written in the Java programming language, an enterprise bean is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called `checkInventoryLevel` and `orderProduct`. By invoking these methods, clients can access the inventory services provided by the application.

Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the Enterprise Bean container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The Enterprise Bean container, rather than the bean developer, is responsible for system-level services, such as transaction management and security authorization.

Second, because the beans rather than the clients contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. Provided that they use the standard APIs, these applications can run on any compliant Jakarta EE server.

When to Use Enterprise Beans

You should consider using enterprise beans if your application has any of the following requirements.

- The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.
- Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
- The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

Types of Enterprise Beans

Enterprise Bean Types summarizes the two types of enterprise beans. The following sections discuss each type in more detail.

Enterprise Bean Types

Enterprise Bean Type	Purpose
Session	Performs a task for a client; optionally, may implement a web service
Message-driven	Acts as a listener for a particular messaging type, such as Jakarta Messaging

What Is a Session Bean?

A session bean encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

A session bean is not persistent. (That is, its data is not saved to a database.)

For code samples, see [entbeans:ejb-basicexamples::ejb-basicexamples:::_running_the_enterprise_bean_examples].

Types of Session Beans

Session beans are of three types: stateful, stateless, and singleton.

Stateful Session Beans

The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client/bean session. Because the client interacts ("talks") with its bean, this state is often called the conversational state.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

The state is retained for the duration of the client/bean session. If the client removes the bean, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends, there is no need to retain the state.

Stateless Session Beans

A stateless session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the Enterprise Bean container to assign an instance to any client. That is, the state of a stateless session bean should apply across all clients.

Because they can support multiple clients, stateless session beans can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

A stateless session bean can implement a web service, but a stateful session bean cannot.

Singleton Session Beans

A singleton session bean is instantiated once per application and exists for the lifecycle of the application. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients.

Singleton session beans offer similar functionality to stateless session beans but differ from them in that there is only one singleton session bean per application, as opposed to a pool of stateless session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement web service endpoints.

Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

Applications that use a singleton session bean may specify that the singleton should be instantiated upon application startup, which allows the singleton to perform initialization tasks for the application. The singleton may perform cleanup tasks on application shutdown as well, because the singleton will operate throughout the lifecycle of the application.

When to Use Session Beans

Stateful session beans are appropriate if any of the following conditions are true.

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.

To improve performance, you might choose a stateless session bean if it has any of these traits.

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.
- The bean implements a web service.

Singleton session beans are appropriate in the following circumstances.

- State needs to be shared across the application.
- A single enterprise bean needs to be accessed by multiple threads concurrently.
- The application needs an enterprise bean to perform tasks upon application startup and shutdown.
- The bean implements a web service.

What Is a Message-Driven Bean?

A message-driven bean is an enterprise bean that allows Jakarta EE applications to process messages asynchronously. This type of bean normally acts as a Jakarta Messaging message listener, which is similar to an event listener but receives Jakarta Messaging messages instead of events. The messages can be sent by any Jakarta EE component (an application client, another enterprise bean, or a web component) or by a Jakarta Messaging application or system that does not use Jakarta EE technology. Message-driven beans can process Jakarta Messaging messages or other kinds of messages.

What Makes Message-Driven Beans Different from Session Beans?

The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. Interfaces are described in the section Accessing Enterprise Beans. Unlike a session bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the Enterprise Bean container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages, such as a Jakarta Messaging connection, an open database connection, or an object reference to an enterprise bean object.

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through, for example, Jakarta Messaging by sending messages to the message destination for which the message-driven bean class is the `MessageListener`. You assign a message-driven bean's destination during deployment by using GlassFish Server resources.

Message-driven beans have the following characteristics.

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean's `onMessage` method to process the message. The `onMessage` method normally casts the message to one of the five Jakarta Messaging message types and handles it in accordance with the application's business logic. The `onMessage` method can call helper methods or can invoke a session bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the `onMessage` method are part of a single transaction. If message processing is rolled back, the message will be redelivered. For more information, see Receiving Messages Asynchronously Using a Message-Driven Bean and Transactions.

When to Use Message-Driven Beans

Session beans allow you to send Jakarta Messaging messages and to receive them synchronously but not asynchronously. To avoid tying up server resources, do not to use blocking synchronous receives in a server-side component; in general, Jakarta Messaging messages should not be sent or received synchronously. To receive messages asynchronously, use a message-driven bean.

Accessing Enterprise Beans



The material in this section applies only to session beans and not to message-

> driven beans. Because they have a different programming model, message-driven beans do not have interfaces or no-interface views that define client access.

Clients access enterprise beans either through a no-interface view or through a business interface. A no-interface view of an enterprise bean exposes the public methods of the enterprise bean implementation class to clients. Clients using the no-interface view of an enterprise bean may invoke any public methods in the enterprise bean implementation class or any superclasses of the implementation class. A business interface is a standard Java programming language interface that contains the business methods of the enterprise bean.

A client can access a session bean only through the methods defined in the bean's business interface or through the public methods of an enterprise bean that has a no-interface view. The business interface or no-interface view defines the client's view of an enterprise bean. All other aspects of the enterprise bean (method implementations and deployment settings) are hidden from the client.

Well-designed interfaces and no-interface views simplify the development and maintenance of Jakarta EE applications. Not only do clean interfaces and no-interface views shield the clients from any complexities in the Enterprise Bean tier, but they also allow the enterprise beans to change internally without affecting the clients. For example, if you change the implementation of a session bean business method, you won't have to alter the client code. But if you were to change the method definitions in the interfaces, you might have to modify the client code as well. Therefore, it is important that you design the interfaces and no-interface views carefully to isolate your clients from possible changes in the enterprise beans.

Session beans can have more than one business interface. Session beans should, but are not required to, implement their business interface or interfaces.

Using Enterprise Beans in Clients

The client of an enterprise bean obtains a reference to an instance of an enterprise bean through either dependency injection, using Java programming language annotations, or JNDI lookup, using the Java Naming and Directory Interface syntax to find the enterprise bean instance.

Dependency injection is the simplest way of obtaining an enterprise bean reference. Clients that run within a Jakarta EE server-managed environment, Jakarta Faces web applications, Jakarta RESTful web services, other enterprise beans, or Jakarta EE application clients support dependency injection using the `jakarta.ejb.EJB` annotation.

Applications that run outside a Jakarta EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Jakarta EE components to simplify this explicit lookup.

Portable JNDI Syntax

Three JNDI namespaces are used for portable JNDI lookups: `java:global`, `java:module`, and `java:app`.

- The `java:global` JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups. JNDI addresses are of the following form:

```
java:global[/application name]/module name/enterprise bean name[/interface name]
```

Application name and module name default to the name of the application and module minus the file extension. Application names are required only if the application is packaged within an EAR. The interface name is required only if the enterprise bean implements more than one business interface.

- The `java:module` namespace is used to look up local enterprise beans within the same module. JNDI addresses using the `java:module` namespace are of the following form:

  ```
  java:module/enterprise bean name/[interface name]
  ```

  The interface name is required only if the enterprise bean implements more than one business interface.

- The `java:app` namespace is used to look up local enterprise beans packaged within the same application. That is, the enterprise bean is packaged within an EAR file containing multiple Jakarta EE modules. JNDI addresses using the `java:app` namespace are of the following form:

  ```
  java:app[/module name]/enterprise bean name[/interface name]
  ```

  The module name is optional. The interface name is required only if the enterprise bean implements more than one business interface.

For example, if an enterprise bean, `MyBean`, is packaged within the web application archive `myApp.war`, the module name is `myApp`. The portable JNDI name is `java:module/MyBean`. An equivalent JNDI name using the `java:global` namespace is `java:global/myApp/MyBean`.

Deciding on Remote or Local Access

When you design a Jakarta EE application, one of the first decisions you make is the type of client access allowed by the enterprise beans: remote, local, or web service.

Whether to allow local or remote access depends on the following factors.

- Tight or loose coupling of related beans: Tightly coupled beans depend on one another. For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled. Tightly coupled beans are good candidates for local access. Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access.
- Type of client: If an enterprise bean is accessed by application clients, it should allow remote access. In a production environment, these clients almost always run on machines other than those on which GlassFish Server is running. If an enterprise bean's clients are web components or other enterprise beans, the type of access depends on how you want to distribute your components.

- Component distribution: Jakarta EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the server that the web components run on may not be the one on which the enterprise beans they access are deployed. In this distributed scenario, the enterprise beans should allow remote access.
- Performance: Owing to such factors as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you may improve the application's overall performance. Both of these statements are generalizations; performance can vary in different operational environments. Nevertheless, you should keep in mind how your application design might affect performance.

If you aren't sure which type of access an enterprise bean should have, choose remote access. This decision gives you more flexibility. In the future, you can distribute your components to accommodate the growing demands on your application.

Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. If this is the case, either the business interface of the bean must be explicitly designated as a business interface by being decorated with the `@Remote` or `@Local` annotations, or the bean class must explicitly designate the business interfaces by using the `@Remote` and `@Local` annotations. The same business interface cannot be both a local and a remote business interface.

Local Clients

A local client has these characteristics.

- It must run in the same application as the enterprise bean it accesses.
- It can be a web component or another enterprise bean.
- To the local client, the location of the enterprise bean it accesses is not transparent.

The no-interface view of an enterprise bean is a local view. The public methods of the enterprise bean implementation class are exposed to local clients that access the no-interface view of the enterprise bean. Enterprise beans that use the no-interface view do not implement a business interface.

The local business interface defines the bean's business and lifecycle methods. If the bean's business interface is not decorated with `@Local` or `@Remote`, and if the bean class does not specify the interface using `@Local` or `@Remote`, the business interface is by default a local interface.

To build an enterprise bean that allows only local access, you may, but are not required to, do one of the following.

- Create an enterprise bean implementation class that does not implement a business interface, indicating that the bean exposes a no-interface view to clients. For example:

```
@Session
public class MyBean { ... }
```

- Annotate the business interface of the enterprise bean as a `@Local` interface. For example:

```
@Local
public interface InterfaceName { ... }
```

- Specify the interface by decorating the bean class with `@Local` and specify the interface name. For example:

```
@Local(InterfaceName.class)
public class BeanName implements InterfaceName  { ... }
```

Accessing Local Enterprise Beans Using the No-Interface View

Client access to an enterprise bean that exposes a local, no-interface view is accomplished through either dependency injection or JNDI lookup.

- To obtain a reference to the no-interface view of an enterprise bean through dependency injection, use the `jakarta.ejb.EJB` annotation and specify the enterprise bean's implementation class:

```
@EJB
ExampleBean exampleBean;
```

- To obtain a reference to the no-interface view of an enterprise bean through JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method:

```
ExampleBean exampleBean = (ExampleBean)
        InitialContext.lookup("java:module/ExampleBean");
```

Clients do not use the `new` operator to obtain a new instance of an enterprise bean that uses a no-interface view.

Accessing Local Enterprise Beans That Implement Business Interfaces

Client access to enterprise beans that implement local business interfaces is accomplished through either dependency injection or JNDI lookup.

- To obtain a reference to the local business interface of an enterprise bean through dependency injection, use the `jakarta.ejb.EJB` annotation and specify the enterprise bean's local business interface name:

```
@EJB
Example example;
```

- To obtain a reference to a local business interface of an enterprise bean through JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method:

```
ExampleLocal example = (ExampleLocal)
         InitialContext.lookup("java:module/ExampleLocal");
```

Remote Clients

A remote client of an enterprise bean has the following traits.

- It can run on a different machine and a different JVM from the enterprise bean it accesses. (It is not required to run on a different JVM.)
- It can be a web component, an application client, or another enterprise bean.
- To a remote client, the location of the enterprise bean is transparent.
- The enterprise bean must implement a business interface. That is, remote clients may not access an enterprise bean through a no-interface view.

To create an enterprise bean that allows remote access, you must either

- Decorate the business interface of the enterprise bean with the `@Remote` annotation:

```
@Remote
public interface InterfaceName { ... }
```

- Or decorate the bean class with `@Remote`, specifying the business interface or interfaces:

```
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

The remote interface defines the business and lifecycle methods that are specific to the bean. For example, the remote interface of a bean named `BankAccountBean` might have business methods named `deposit` and `credit`. Figure 20, “Interfaces for an Enterprise Bean with Remote Access” shows how the interface controls the client’s view of an enterprise bean.

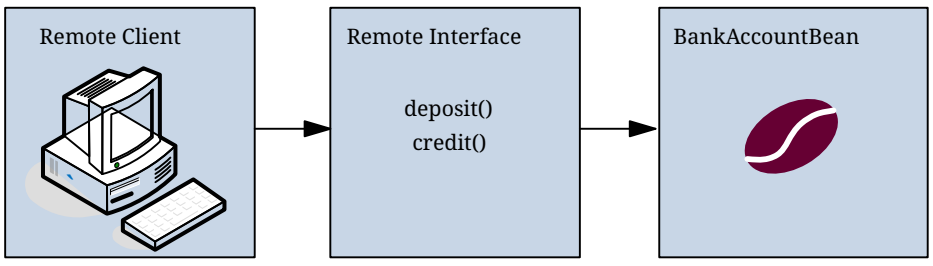


Figure 20. Interfaces for an Enterprise Bean with Remote Access

Client access to an enterprise bean that implements a remote business interface is accomplished through either dependency injection or JNDI lookup.

- To obtain a reference to the remote business interface of an enterprise bean through dependency injection, use the `jakarta.ejb.EJB` annotation and specify the enterprise bean’s

remote business interface name:

```
@EJB
Example example;
```

- To obtain a reference to a remote business interface of an enterprise bean through JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method:

```
ExampleRemote example = (ExampleRemote)
        InitialContext.lookup("java:global/myApp/ExampleRemote");
```

Web Service Clients

A web service client can access a Jakarta EE application in two ways. First, the client can access a web service created with Jakarta XML Web Services. (For more information on Jakarta XML Web Services, see "Building Web Services with Jakarta XML Web Services", available in a previous version of the tutorial.) Second, a web service client can invoke the business methods of a stateless session bean. Message beans cannot be accessed by web service clients.

Provided that it uses the correct protocols (SOAP, HTTP, WSDL), any web service client can access a stateless session bean, whether or not the client is written in the Java programming language. The client doesn't even "know" what technology implements the service: stateless session bean, Jakarta XML Web Services, or some other technology. In addition, enterprise beans and web components can be clients of web services. This flexibility enables you to integrate Jakarta EE applications with web services.

A web service client accesses a stateless session bean through the bean's web service endpoint implementation class. By default, all public methods in the bean class are accessible to web service clients. The `@WebMethod` annotation may be used to customize the behavior of web service methods. If the `@WebMethod` annotation is used to decorate the bean class's methods, only those methods decorated with `@WebMethod` are exposed to web service clients.

For a code sample, see A Web Service Example: helloservice.

Method Parameters and Access

The type of access affects the parameters of the bean methods that are called by clients. The following sections apply not only to method parameters but also to method return values.

Isolation

The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and the bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.

In a local call, both the client and the bean can modify the same parameter object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your

components, replacing the local calls with remote ones.

As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.

Granularity of Accessed Data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

The Contents of an Enterprise Bean

To develop an enterprise bean, you must provide the following files.

- Enterprise bean class: Implements the business methods of the enterprise bean and any lifecycle callback methods.
- Business interfaces: Define the business methods implemented by the enterprise bean class. A business interface is not required if the enterprise bean exposes a local, no-interface view.
- Helper classes: Other classes needed by the enterprise bean class, such as exception and utility classes.

Package the programming artifacts in the preceding list either into an Enterprise Bean JAR file (a stand-alone module that stores the enterprise bean) or within a web application archive (WAR) module. See Packaging Enterprise Beans in enterprise bean JAR Modules and Packaging Enterprise Beans in WAR Modules for more information.

Naming Conventions for Enterprise Beans

Because enterprise beans are composed of multiple parts, it's useful to follow a naming convention for your applications. Naming Conventions for Enterprise Beans summarizes the conventions for the example beans in this tutorial.

Naming Conventions for Enterprise Beans

Item	Syntax	Example
Enterprise bean name	*name*Bean	`AccountBean`
Enterprise bean class	*name*Bean	`AccountBean`
Business interface	*name*	`Account`

The Lifecycles of Enterprise Beans

An enterprise bean goes through various stages during its lifetime, or lifecycle. Each type of enterprise bean (stateful session, stateless session, singleton session, or message-driven) has a different lifecycle.

The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and run the

code examples first.

The Lifecycle of a Stateful Session Bean

Figure 21, “Lifecycle of a Stateful Session Bean” illustrates the stages that a stateful session bean passes through during its lifetime. The client initiates the lifecycle by obtaining a reference to a stateful session bean. The container performs any dependency injection and then invokes the method annotated with `@PostConstruct`, if any. The bean is now ready to have its business methods invoked by the client.

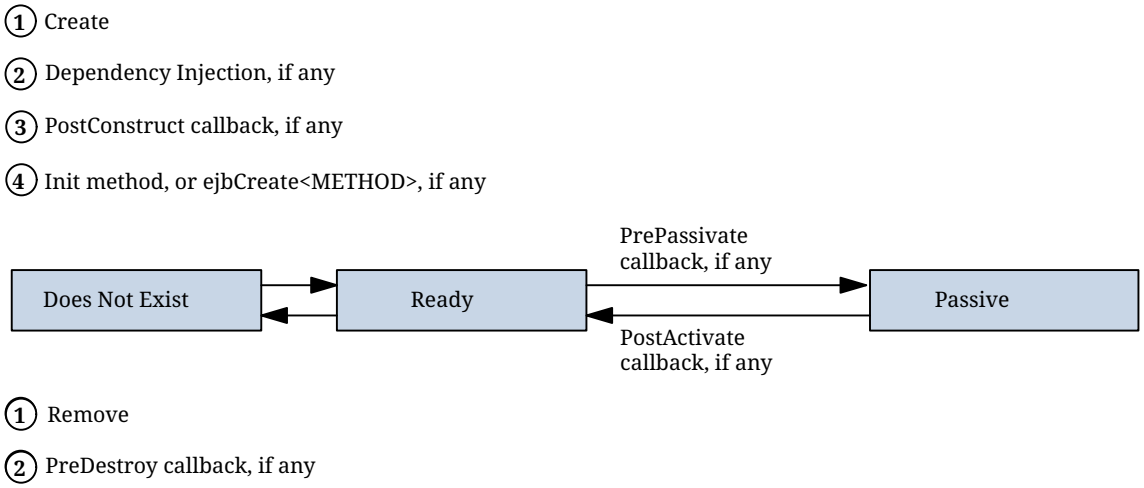


Figure 21. Lifecycle of a Stateful Session Bean

While in the ready stage, the Enterprise Bean container may decide to deactivate, or passivate, the bean by moving it from memory to secondary storage. (Typically, the Enterprise Bean container uses a least-recently-used algorithm to select a bean for passivation.) The Enterprise Bean container invokes the method annotated `@PrePassivate`, if any, immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the Enterprise Bean container activates the bean, calls the method annotated `@PostActivate`, if any, and then moves it to the ready stage.

At the end of the lifecycle, the client invokes a method annotated `@Remove`, and the Enterprise Bean container calls the method annotated `@PreDestroy`, if any. The bean’s instance is then ready for garbage collection.

Your code controls the invocation of only one lifecycle method: the method annotated `@Remove`. All other methods in Figure 21, “Lifecycle of a Stateful Session Bean” are invoked by the Enterprise Bean container. See Resource Adapters and Contracts for more information.

The Lifecycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its lifecycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 22, “Lifecycle of a Stateless or Singleton Session Bean” illustrates the stages of a stateless session bean.

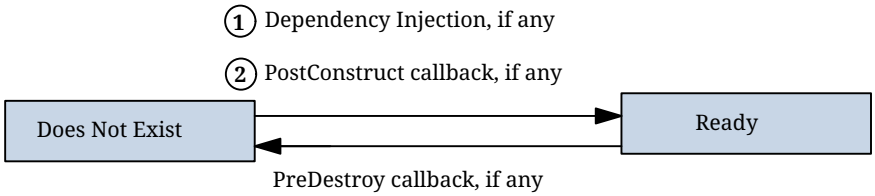


Figure 22. Lifecycle of a Stateless or Singleton Session Bean

The Enterprise Bean container typically creates and maintains a pool of stateless session beans, beginning the stateless session bean's lifecycle. The container performs any dependency injection and then invokes the method annotated `@PostConstruct`, if it exists. The bean is now ready to have its business methods invoked by a client.

At the end of the lifecycle, the Enterprise Bean container calls the method annotated `@PreDestroy`, if it exists. The bean's instance is then ready for garbage collection.

The Lifecycle of a Singleton Session Bean

Like a stateless session bean, a singleton session bean is never passivated and has only two stages, nonexistent and ready for the invocation of business methods, as shown in Figure 22, "Lifecycle of a Stateless or Singleton Session Bean".

The Enterprise Bean container initiates the singleton session bean lifecycle by creating the singleton instance. This occurs upon application deployment if the singleton is annotated with the `@Startup` annotation. The container performs any dependency injection and then invokes the method annotated `@PostConstruct`, if it exists. The singleton session bean is now ready to have its business methods invoked by the client.

At the end of the lifecycle, the Enterprise Bean container calls the method annotated `@PreDestroy`, if it exists. The singleton session bean is now ready for garbage collection.

The Lifecycle of a Message-Driven Bean

Figure 23, "Lifecycle of a Message-Driven Bean" illustrates the stages in the lifecycle of a message-driven bean.

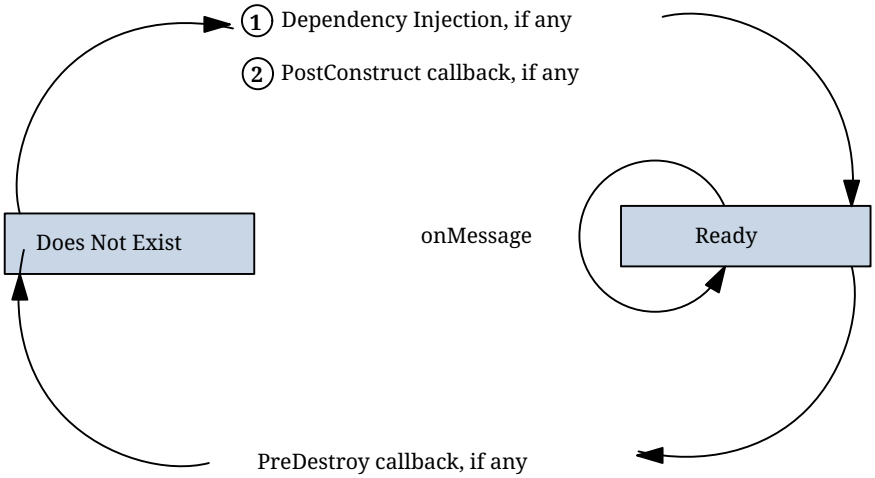


Figure 23. Lifecycle of a Message-Driven Bean

The Enterprise Bean container usually creates a pool of message-driven bean instances. For each instance, the Enterprise Bean container performs these tasks.

1. If the message-driven bean uses dependency injection, the container injects these references before instantiating the instance.
2. The container calls the method annotated `@PostConstruct`, if any.

Like a stateless session bean, a message-driven bean is never passivated and has only two states: nonexistent and ready to receive messages.

At the end of the lifecycle, the container calls the method annotated `@PreDestroy`, if any. The bean's instance is then ready for garbage collection.

Further Information about Enterprise Beans

For more information on Jakarta Enterprise Beans technology, see the Jakarta Enterprise Beans 4.0 specification:
https://jakarta.ee/specifications/enterprise-beans/4.0/

Getting Started with Enterprise Beans



This section was last updated for Jakarta EE Version 9.1

This chapter shows how to develop, deploy, and run a simple Jakarta EE application named `converter` that uses an enterprise bean for its business logic. The purpose of `converter` is to calculate currency conversions among Japanese yen, euros, and US dollars. The `converter` application consists of an enterprise bean, which performs the calculations, and a web client.

Starting With Enterprise Beans

Here's an overview of the steps you'll follow:

1. Create the enterprise bean: `ConverterBean`.
2. Create the web client.
3. Deploy `converter` onto the server.
4. Using a browser, run the web client.

Before proceeding, make sure that you've done the following:

- Read [intro:overview::overview:::_overview]
- Become familiar with enterprise beans (see [entbeans:ejb-intro::ejb-intro:::_enterprise_beans])
- Started the server (see Starting and Stopping GlassFish Server)

Creating the Enterprise Bean

The enterprise bean in our example is a stateless session bean called `ConverterBean`. The source code for `ConverterBean` is in the `jakartaee-examples/tutorial/ejb/converter/src/main/java/` directory.

Creating `ConverterBean` requires these steps:

1. Coding the bean's implementation class (the source code is provided)
2. Compiling the source code

Coding the Enterprise Bean Class

The enterprise bean class for this example is called `ConverterBean`. This class implements two business methods: `dollarToYen` and `yenToEuro`. Because the enterprise bean class doesn't implement a business interface, the enterprise bean exposes a local, no-interface view. The public methods in the enterprise bean class are available to clients that obtain a reference to `ConverterBean`. The source code for the `ConverterBean` class is as follows:

```
package ee.jakarta.tutorial.converter.ejb;

import java.math.BigDecimal;
import jakarta.ejb.*;

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("83.0602");
    private BigDecimal euroRate = new BigDecimal("0.0093016");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

Note the `@Stateless` annotation decorating the enterprise bean class. This annotation lets the container know that `ConverterBean` is a stateless session bean.

Creating the converter Web Client

The web client is contained in the following servlet class under the `jakartaee-examples/tutorial/ejb/converter/src/main/java/` directory:

```
converter/web/ConverterServlet.java
```

A Jakarta servlet is a web component that responds to HTTP requests.

The `ConverterServlet` class uses dependency injection to obtain a reference to `ConverterBean`. The `jakarta.ejb.EJB` annotation is added to the declaration of the private member variable `converter`,

which is of type `ConverterBean`. `ConverterBean` exposes a local, no-interface view, so the enterprise bean implementation class is the variable type:

```
@WebServlet(urlPatterns="/")
public class ConverterServlet extends HttpServlet {
  @EJB
  ConverterBean converter;
  ...
}
```

When the user enters an amount to be converted to yen and euro, the amount is retrieved from the request parameters; then the `ConverterBean.dollarToYen` and the `ConverterBean.yenToEuro` methods are called:

```
...
try {
  String amount = request.getParameter("amount");
  if (amount != null && amount.length() > 0) {
    // convert the amount to a BigDecimal from the request parameter
    BigDecimal d = new BigDecimal(amount);
    // call the ConverterBean.dollarToYen() method to get the amount
    // in Yen
    BigDecimal yenAmount = converter.dollarToYen(d);

    // call the ConverterBean.yenToEuro() method to get the amount
    // in Euros
    BigDecimal euroAmount = converter.yenToEuro(yenAmount);
    ...
  }
  ...
}
```

The results are displayed to the user.

Running the converter Example

Now you are ready to compile the enterprise bean class (`ConverterBean.java`) and the servlet class (`ConverterServlet.java`) and to package the compiled classes into a WAR file. You can use either NetBeans IDE or Maven to build, package, deploy, and run the `converter` example.

To Run the converter Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

```
jakartaee-examples/tutorial/ejb
```

4. Select the `converter` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `converter` project and select **Build**.
7. Open a web browser to the following URL:

```
http://localhost:8080/converter
```

8. On the Servlet ConverterServlet page, enter `100` in the field and click Submit.

   A second page opens, showing the converted values.

To Run the converter Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

```
jakartaee-examples/tutorial/ejb/converter/
```

3. Enter the following command:

```
mvn install
```

   This command compiles the source files for the enterprise bean and the servlet, packages the project into a WAR module (`converter.war`), and deploys the WAR to the server. For more information about Maven, see Building the Examples.

4. Open a web browser to the following URL:

```
http://localhost:8080/converter
```

5. On the Servlet ConverterServlet page, enter `100` in the field and click Submit.

   A second page opens, showing the converted values.

Modifying the Jakarta EE Application

GlassFish Server supports iterative development. Whenever you make a change to a Jakarta EE application, you must redeploy the application.

To Modify a Class File

To modify a class file in an enterprise bean, you change the source code, recompile it, and redeploy the application. For example, to update the exchange rate in the `dollarToYen` business method of the `ConverterBean` class, you would follow these steps.

To modify `ConverterServlet`, the procedure is the same.

1. Edit `ConverterBean.java` and save the file.
2. Recompile the source file.
    a. To recompile `ConverterBean.java` in NetBeans IDE, right-click the `converter` project and select Run.

    This recompiles the `ConverterBean.java` file, replaces the old class file in the build directory, and redeploys the application to GlassFish Server.

    b. Recompile `ConverterBean.java` using Maven.
        i. In a terminal window, go to the `jakartaee-examples/tutorial/ejb/converter/` directory.
        ii. Enter the following command:

        ```
        mvn install
        ```

        This command repackages and deploys the application.

Running the Enterprise Bean Examples



This section was last updated for Jakarta EE Version 9.1

This chapter describes the Jakarta Enterprise Beans examples. Session beans provide a simple but powerful way to encapsulate business logic within an application. They can be accessed from remote Java clients, web service clients, and components running in the same server.

Overview of the Jakarta Enterprise Beans Examples

In [entbeans:ejb-gettingstarted::ejb-gettingstarted:::_getting_started_with_enterprise_beans], you built a stateless session bean named `ConverterBean`. This chapter examines the source code of four more session beans:

- `CartBean`: a stateful session bean that is accessed by a remote client
- `CounterBean`: a singleton session bean
- `HelloServiceBean`: a stateless session bean that implements a web service
- `TimerSessionBean`: a stateless session bean that sets a timer

The cart Example

The `cart` example represents a shopping cart in an online bookstore and uses a stateful session

bean to manage the operations of the shopping cart. The bean's client can add a book to the cart, remove a book, or retrieve the cart's contents. To assemble `cart`, you need the following code:

- Session bean class (`CartBean`)
- Remote business interface (`Cart`)

All session beans require a session bean class. All enterprise beans that permit remote access must have a remote business interface. To meet the needs of a specific application, an enterprise bean may also need some helper classes. The `CartBean` session bean uses two helper classes, `BookException` and `IdVerifier`, which are discussed in the section Helper Classes.

The source code for this example is in the `jakartaee-examples/tutorial/ejb/cart/` directory.

The Business Interface

The `Cart` business interface is a plain Java interface that defines all the business methods implemented in the bean class. If the bean class implements a single interface, that interface is assumed to the business interface. The business interface is a local interface unless it is annotated with the `jakarta.ejb.Remote` annotation; the `jakarta.ejb.Local` annotation is optional in this case.

The bean class may implement more than one interface. In that case, the business interfaces must either be explicitly annotated `@Local` or `@Remote` or be specified by decorating the bean class with `@Local` or `@Remote`. However, the following interfaces are excluded when determining whether the bean class implements more than one interface:

- `java.io.Serializable`
- `java.io.Externalizable`
- Any of the interfaces defined by the `jakarta.ejb` package

The source code for the `Cart` business interface is as follows:

```
package ee.jakarta.tutorial.cart.ejb;

import cart.util.BookException;
import java.util.List;
import jakarta.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id) throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```

Session Bean Class

The session bean class for this example is called `CartBean`. Like any stateful session bean, the `CartBean` class must meet the following requirements.

- The class is annotated `@Stateful`.
- The class implements the business methods defined in the business interface.

Stateful session beans may also do the following.

- Implement the business interface, a plain Java interface. It is good practice to implement the bean's business interface.
- Implement any optional lifecycle callback methods, annotated `@PostConstruct`, `@PreDestroy`, `@PostActivate`, and `@PrePassivate`.
- Implement any optional business methods annotated `@Remove`.

The source code for the `CartBean` class is as follows:

```
package ee.jakarta.tutorial.cart.ejb;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import ee.jakarta.tutorial.cart.util.BookException;
import ee.jakarta.tutorial.cart.util.IdVerifier;
import jakarta.ejb.Remove;
import jakarta.ejb.Stateful;

@Stateful
public class CartBean implements Cart {
    String customerId;
    String customerName;
    List<String> contents;

    @Override
    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        customerId = "0";
        contents = new ArrayList<>();
    }

    @Override
    public void initialize(String person, String id)
                throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
```

```
        } else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();
        if (idChecker.validate(id)) {
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }

        contents = new ArrayList<>();
    }

    @Override
    public void addBook(String title) {
        contents.add(title);
    }

    @Override
    public void removeBook(String title) throws BookException {
        boolean result = contents.remove(title);
        if (result == false) {
            throw new BookException("\"" + title + " not in cart.");
        }
    }

    @Override
    public List<String> getContents() {
        return contents;
    }

    @Remove
    @Override
    public void remove() {
        contents = null;
    }
}
```

Lifecycle Callback Methods

A method in the bean class may be declared as a lifecycle callback method by annotating the method with the following annotations.

- `jakarta.annotation.PostConstruct`: Methods annotated with `@PostConstruct` are invoked by the container on newly constructed bean instances after all dependency injection has completed and before the first business method is invoked on the enterprise bean.
- `jakarta.annotation.PreDestroy`: Methods annotated with `@PreDestroy` are invoked after any method annotated `@Remove` has completed and before the container removes the enterprise bean instance.

- `jakarta.ejb.PostActivate`: Methods annotated with `@PostActivate` are invoked by the container after the container moves the bean from secondary storage to active status.
- `jakarta.ejb.PrePassivate`: Methods annotated with `@PrePassivate` are invoked by the container before it passivates the enterprise bean, meaning that the container temporarily removes the bean from the environment and saves it to secondary storage.

Lifecycle callback methods must return `void` and have no parameters.

Business Methods

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the object reference it gets from dependency injection or JNDI lookup. From the client's perspective, the business methods appear to run locally, although they run remotely in the session bean. The following code snippet shows how the `CartClient` program invokes the business methods:

```
cart.initialize("Duke DeEarl", "123");
...
cart.addBook("Bel Canto");
 ...
List<String> bookList = cart.getContents();
...
cart.removeBook("Gravity's Rainbow");
```

The `CartBean` class implements the business methods in the following code:

```
@Override
public void addBook(String title) {
   contents.add(title);
}

@Override
public void removeBook(String title) throws BookException {
   boolean result = contents.remove(title);
   if (result == false) {
      throw new BookException("\"" + title + "not in cart.");
   }
}

@Override
public List<String> getContents() {
   return contents;
}
```

The signature of a business method must conform to these rules.

- The method name must not begin with `ejb`, to avoid conflicts with callback methods defined by the Jakarta Enterprise Beans architecture. For example, you cannot call a business method

`ejbCreate` or `ejbActivate`.

- The access control modifier must be `public`.
- If the bean allows remote access through a remote business interface, the arguments and return types must be legal types for the Java Remote Method Invocation (RMI) API.
- If the bean is a Jakarta XML Web Services endpoint, the arguments and return types for the methods annotated `@WebMethod` must be legal types for Jakarta XML Web Services.
- If the bean is a Jakarta RESTful Web Services resource, the arguments and return types for the resource methods must be legal types for Jakarta RESTful Web Services.
- The modifier must not be `static` or `final`.

The `throws` clause can include exceptions that you define for your application. The `removeBook` method, for example, throws a `BookException` if the book is not in the cart.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw a `jakarta.ejb.EJBException`. The container will not wrap application exceptions, such as `BookException`. Because `EJBException` is a subclass of `RuntimeException`, you do not need to include it in the `throws` clause of the business method.

The @Remove Method

Business methods annotated with `jakarta.ejb.Remove` in the stateful session bean class can be invoked by enterprise bean clients to remove the bean instance. The container will remove the enterprise bean after a `@Remove` method completes, either normally or abnormally.

In `CartBean`, the `remove` method is a `@Remove` method:

```
@Remove
@Override
public void remove() {
    contents = null;
}
```

Helper Classes

The `CartBean` session bean has two helper classes: `BookException` and `IdVerifier`. The `BookException` is thrown by the `removeBook` method, and the `IdVerifier` validates the `customerId` in one of the `create` methods. Helper classes may reside in an EJB JAR file that contains the enterprise bean class; a WAR file if the enterprise bean is packaged within a WAR; or an EAR file that contains an EJB JAR, a WAR file, or a separate library JAR file. In `cart`, the helper classes are included in a library JAR used by the application client and the EJB JAR.

Running the cart Example

Now you are ready to compile the remote interface (`Cart.java`), the enterprise bean class (`CartBean.java`), the client class (`CartClient.java`), and the helper classes (`BookException.java` and `IdVerifier.java`).

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `cart` application.

To Run the cart Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/ejb
   ```

4. Select the `cart` folder.
5. Select the **Open Required Projects** check box.
6. Click Open Project.
7. In the **Projects** tab, right-click the `cart` project and select **Build**.

   This builds and packages the application into `cart.ear`, located in `jakartaee-examples/tutorial/ejb/cart/cart-ear/target/`, and deploys this EAR file to your GlassFish Server instance.

   You will see the output of the `cart-app-client` application client in the Output tab:

   ```
   ...
   Retrieving book title from cart: Infinite Jest
   Retrieving book title from cart: Bel Canto
   Retrieving book title from cart: Kafka on the Shore
   Removing "Gravity's Rainbow" from cart.
   Caught a BookException: "Gravity's Rainbow" not in cart.
   ```

To Run the cart Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/ejb/cart/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This command compiles and packages the application into an EAR file, `cart.ear`, located in the `target` directory, and deploys the EAR to your GlassFish Server instance.

   Then, the client stubs are retrieved and run. This is equivalent to running the following command:

```
appclient -client cart-ear/target/cart-earClient.jar
```

The client JAR, `cart-earClient.jar`, contains the application client class, the helper class `BookException`, and the `Cart` business interface.

When you run the client, the application client container injects any component references declared in the application client class, in this case the reference to the `Cart` enterprise bean.

You will see the output of the `cart-app-client` application client in the terminal window:

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
```

A Singleton Session Bean Example: counter

The `counter` example demonstrates how to create a singleton session bean.

Creating a Singleton Session Bean

The `jakarta.ejb.Singleton` annotation is used to specify that the enterprise bean implementation class is a singleton session bean:

```
@Singleton
public class SingletonBean { ... }
```

Initializing Singleton Session Beans

The Enterprise Bean container is responsible for determining when to initialize a singleton session bean instance unless the singleton session bean implementation class is annotated with the `jakarta.ejb.Startup` annotation. In this case, sometimes called eager initialization, the Enterprise Bean container must initialize the singleton session bean upon application startup. The singleton session bean is initialized before the Enterprise Bean container delivers client requests to any enterprise beans in the application. This allows the singleton session bean to perform, for example, application startup tasks.

The following singleton session bean stores the status of an application and is eagerly initialized:

```
@Startup
@Singleton
public class StatusBean {
  private String status;

  @PostConstruct
```

```
  void init {
    status = "Ready";
  }
  ...
}
```

Sometimes multiple singleton session beans are used to initialize data for an application and therefore must be initialized in a specific order. In these cases, use the `jakarta.ejb.DependsOn` annotation to declare the startup dependencies of the singleton session bean. The `@DependsOn` annotation's `value` attribute is one or more strings that specify the name of the target singleton session bean. If more than one dependent singleton bean is specified in `@DependsOn`, the order in which they are listed is not necessarily the order in which the Enterprise Bean container will initialize the target singleton session beans.

The following singleton session bean, `PrimaryBean`, should be started up first:

```
@Singleton
public class PrimaryBean { ... }
```

`SecondaryBean` depends on `PrimaryBean`:

```
@Singleton
@DependsOn("PrimaryBean")
public class SecondaryBean { ... }
```

This guarantees that the Enterprise Bean container will initialize `PrimaryBean` before `SecondaryBean`.

The following singleton session bean, `TertiaryBean`, depends on `PrimaryBean` and `SecondaryBean`:

```
@Singleton
@DependsOn({"PrimaryBean", "SecondaryBean"})
public class TertiaryBean { ... }
```

`SecondaryBean` explicitly requires `PrimaryBean` to be initialized before it is initialized, through its own `@DependsOn` annotation. In this case, the Enterprise Bean container will first initialize `PrimaryBean`, then `SecondaryBean`, and finally `TertiaryBean`.

If, however, `SecondaryBean` did not explicitly depend on `PrimaryBean`, the Enterprise Bean container may initialize either `PrimaryBean` or `SecondaryBean` first. That is, the Enterprise Bean container could initialize the singletons in the following order: `SecondaryBean`, `PrimaryBean`, `TertiaryBean`.

Managing Concurrent Access in a Singleton Session Bean

Singleton session beans are designed for concurrent access, situations in which many clients need to access a single instance of a session bean at the same time. A singleton's client needs only a reference to a singleton in order to invoke any business methods exposed by the singleton and

doesn't need to worry about any other clients that may be simultaneously invoking business methods on the same singleton.

When creating a singleton session bean, concurrent access to the singleton's business methods can be controlled in two ways: container-managed concurrency and bean-managed concurrency.

The `jakarta.ejb.ConcurrencyManagement` annotation is used to specify container-managed or bean-managed concurrency for the singleton. With `@ConcurrencyManagement`, a type attribute must be set to either `jakarta.ejb.ConcurrencyManagementType.CONTAINER` or `jakarta.ejb.ConcurrencyManagementType.BEAN`. If no `@ConcurrencyManagement` annotation is present on the singleton implementation class, the Enterprise Bean container default of container-managed concurrency is used.

Container-Managed Concurrency

If a singleton uses container-managed concurrency, the Enterprise Bean container controls client access to the business methods of the singleton. The `jakarta.ejb.Lock` annotation and a `jakarta.ejb.LockType` type are used to specify the access level of the singleton's business methods or `@Timeout` methods. The `LockType` enumerated types are `READ` and `WRITE`.

Annotate a singleton's business or timeout method with `@Lock(LockType.READ)` if the method can be concurrently accessed, or shared, with many clients. Annotate the business or timeout method with `@Lock(LockType.WRITE)` if the singleton session bean should be locked to other clients while a client is calling that method. Typically, the `@Lock(LockType.WRITE)` annotation is used when clients are modifying the state of the singleton.

Annotating a singleton class with `@Lock` specifies that all the business methods and any timeout methods of the singleton will use the specified lock type unless they explicitly set the lock type with a method-level `@Lock` annotation. If no `@Lock` annotation is present on the singleton class, the default lock type, `@Lock(LockType.WRITE)`, is applied to all business and timeout methods.

The following example shows how to use the `@ConcurrencyManagement`, `@Lock(LockType.READ)`, and `@Lock(LockType.WRITE)` annotations for a singleton that uses container-managed concurrency.

Although by default singletons use container-managed concurrency, the `@ConcurrencyManagement(CONTAINER)` annotation may be added at the class level of the singleton to explicitly set the concurrency management type:

```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class ExampleSingletonBean {
  private String state;

  @Lock(LockType.READ)
  public String getState() {
    return state;
  }

  @Lock(LockType.WRITE)
  public void setState(String newState) {
```

```
    state = newState;
  }
}
```

The `getState` method can be accessed by many clients at the same time because it is annotated with `@Lock(LockType.READ)`. When the `setState` method is called, however, all the methods in `ExampleSingletonBean` will be locked to other clients because `setState` is annotated with `@Lock(LockType.WRITE)`. This prevents two clients from attempting to simultaneously change the `state` variable of `ExampleSingletonBean`.

The `getData` and `getStatus` methods in the following singleton are of type `READ`, and the `setStatus` method is of type `WRITE`:

```
@Singleton
@Lock(LockType.READ)
public class SharedSingletonBean {
  private String data;
  private String status;

  public String getData() {
    return data;
  }

  public String getStatus() {
    return status;
  }

  @Lock(LockType.WRITE)
  public void setStatus(String newStatus) {
    status = newStatus;
  }
}
```

If a method is of locking type `WRITE`, client access to all the singleton's methods is blocked until the current client finishes its method call or an access timeout occurs. When an access timeout occurs, the Enterprise Bean container throws a `jakarta.ejb.ConcurrentAccessTimeoutException`. The `jakarta.ejb.AccessTimeout` annotation is used to specify the number of milliseconds before an access timeout occurs. If added at the class level of a singleton, `@AccessTimeout` specifies the access timeout value for all methods in the singleton unless a method explicitly overrides the default with its own `@AccessTimeout` annotation.

The `@AccessTimeout` annotation can be applied to both `@Lock(LockType.READ)` and `@Lock(LockType.WRITE)` methods. The `@AccessTimeout` annotation has one required element, `value`, and one optional element, `unit`. By default, the `value` is specified in milliseconds. To change the `value` unit, set `unit` to one of the `java.util.concurrent.TimeUnit` constants: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, or `SECONDS`.

The following singleton has a default access timeout value of 120,000 milliseconds, or 2 minutes.

The `doTediousOperation` method overrides the default access timeout and sets the value to 360,000 milliseconds, or 6 minutes:

```
@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {
  private String status;

  @Lock(LockType.WRITE)
  public void setStatus(String new Status) {
    status = newStatus;
  }

  @Lock(LockType.WRITE)
  @AccessTimeout(value=360000)
  public void doTediousOperation {
    ...
  }
}
```

The following singleton has a default access timeout value of 60 seconds, specified using the `TimeUnit.SECONDS` constant:

```
@Singleton
@AccessTimeout(value=60, unit=TimeUnit.SECONDS)
public class StatusSingletonBean { ... }
```

Bean-Managed Concurrency

Singletons that use bean-managed concurrency allow full concurrent access to all the business and timeout methods in the singleton. The developer of the singleton is responsible for ensuring that the state of the singleton is synchronized across all clients. Developers who create singletons with bean-managed concurrency are allowed to use the Java programming language synchronization primitives, such as `synchronization` and `volatile`, to prevent errors during concurrent access.

Add a `@ConcurrencyManagement` annotation with the type set to `ConcurrencyManagementType.BEAN` at the class level of the singleton to specify bean-managed concurrency:

```
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
@Singleton
public class AnotherSingletonBean { ... }
```

Handling Errors in a Singleton Session Bean

If a singleton session bean encounters an error when initialized by the Enterprise Bean container, that singleton instance will be destroyed.

Unlike other enterprise beans, once a singleton session bean instance is initialized, it is not destroyed if the singleton's business or lifecycle methods cause system exceptions. This ensures that the same singleton instance is used throughout the application lifecycle.

The Architecture of the counter Example

The `counter` example consists of a singleton session bean, `CounterBean`, and a Jakarta Faces Facelets web front end.

`CounterBean` is a simple singleton with one method, `getHits`, that returns an integer representing the number of times a web page has been accessed. Here is the code of `CounterBean`:

```
package ee.jakarta.tutorial.counter.ejb;

import jakarta.ejb.Singleton;

/**
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
    public int getHits() {
        return hits++;
    }
}
```

The `@Singleton` annotation marks `CounterBean` as a singleton session bean. `CounterBean` uses a local, no-interface view.

`CounterBean` uses the Enterprise Bean container's default metadata values for singletons to simplify the coding of the singleton implementation class. There is no `@ConcurrencyManagement` annotation on the class, so the default of container-managed concurrency access is applied. There is no `@Lock` annotation on the class or business method, so the default of `@Lock(WRITE)` is applied to the only business method, `getHits`.

The following version of `CounterBean` is functionally equivalent to the preceding version:

```
package ee.jakarta.tutorial.counter.ejb;

import jakarta.ejb.Singleton;
import jakarta.ejb.ConcurrencyManagement;
import static jakarta.ejb.ConcurrencyManagementType.CONTAINER;
import jakarta.ejb.Lock;
import jakarta.ejb.LockType.WRITE;
```

```
/**
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
@ConcurrencyManagement(CONTAINER)
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
    @Lock(WRITE)
    public int getHits() {
        return hits++;
    }
}
```

The web front end of `counter` consists of a Jakarta Faces managed bean, `Count.java`, that is used by the Facelets XHTML files `template.xhtml` and `index.xhtml`. The `Count` Jakarta Faces managed bean obtains a reference to `CounterBean` through dependency injection. `Count` defines a `hitCount` JavaBeans property. When the `getHitCount` getter method is called from the XHTML files, `CounterBean`'s `getHits` method is called to return the current number of page hits.

Here's the `Count` managed bean class:

```
@Named
@ConversationScoped
public class Count implements Serializable {
    @EJB
    private CounterBean counterBean;

    private int hitCount;

    public Count() {
        this.hitCount = 0;
    }

    public int getHitCount() {
        hitCount = counterBean.getHits();
        return hitCount;
    }

    public void setHitCount(int newHits) {
        this.hitCount = newHits;
    }
}
```

The `template.xhtml` and `index.xhtml` files are used to render a Facelets view that displays the number of hits to that view. The `index.xhtml` file uses an expression language statement, `#{count.hitCount}`, to access the `hitCount` property of the `Count` managed bean. Here is the content of

`index.xhtml`:

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="jakarta.faces.facelets"
      xmlns:h="jakarta.faces.html">
    <ui:composition template="/template.xhtml">
        <ui:define name="title">
            This page has been accessed #{count.hitCount} time(s).
        </ui:define>
        <ui:define name="body">
            Hooray!
        </ui:define>
    </ui:composition>
</html>
```

Running the counter Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `counter` example.

To Run the counter Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/ejb
   ```

4. Select the `counter` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `counter` project and select **Run**.

   A web browser will open the URL http://localhost:8080/counter, which displays the number of hits.

7. Reload the page to see the hit count increment.

To Run the counter Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/ejb/counter/
   ```

3. Enter the following command:

```
mvn install
```

This will build and deploy `counter` to your GlassFish Server instance.

4. In a web browser, enter the following URL:

```
http://localhost:8080/counter
```

5. Reload the page to see the hit count increment.

A Web Service Example: helloservice

This example demonstrates a simple web service that generates a response based on information received from the client. `HelloServiceBean` is a stateless session bean that implements a single method: `sayHello`. This method matches the `sayHello` method invoked by the client described in "A Simple XML Web Services Application Client" (available in a previous version of the tutorial).

The Web Service Endpoint Implementation Class

`HelloServiceBean` is the endpoint implementation class, typically the primary programming artifact for enterprise bean web service endpoints. The web service endpoint implementation class has the following requirements.

- The class must be annotated with either the `jakarta.jws.WebService` or the `jakarta.jws.WebServiceProvider` annotation.
- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation but is not required to do so. If no `endpointInterface` is specified in `@WebService`, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public and must not be declared `static` or `final`.
- Business methods that are exposed to web service clients must be annotated with `jakarta.jws.WebMethod`.
- Business methods that are exposed to web service clients must have Jakarta XML Binding-compatible parameters and return types. See the list of Jakarta XML Binding default data type bindings at "Types Supported by XML Web Services" (available in a previous version of the tutorial).
- The implementing class must not be declared `final` and must not be `abstract`.
- The implementing class must have a default public constructor.
- The endpoint class must be annotated `@Stateless`.
- The implementing class must not define the `finalize` method.
- The implementing class may use the `jakarta.annotation.PostConstruct` or `jakarta.annotation.PreDestroy` annotations on its methods for lifecycle event callbacks.

  The `@PostConstruct` method is called by the container before the implementing class begins

responding to web service clients.

The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

Stateless Session Bean Implementation Class

The `HelloServiceBean` class implements the `sayHello` method, which is annotated `@WebMethod`. The source code for the `HelloServiceBean` class is as follows:

```
package ee.jakarta.tutorial.helloservice.ejb;

import jakarta.ejb.Stateless;
import jakarta.jws.WebMethod;
import jakarta.jws.WebService;

@Stateless
@WebService
public class HelloServiceBean {
    private final String message = "Hello, ";

    public void HelloServiceBean() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Running the helloservice Example

You can use either NetBeans IDE or Maven to build, package, and deploy the `helloservice` example. You can then use the Administration Console to test the web service endpoint methods.

To Build, Package, and Deploy the helloservice Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/ejb
   ```

4. Select the `helloservice` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `helloservice` project and select **Build**.

   This builds and packages the application into `helloservice.ear`, located in `jakartaee-`

examples/tutorial/ejb/helloservice/target/, and deploys this EAR file to GlassFish Server.

To Build, Package, and Deploy the helloservice Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

```
jakartaee-examples/tutorial/ejb/helloservice/
```

3. Enter the following command:

```
mvn install
```

This compiles the source files and packages the application into an enterprise bean JAR file located at jakartaee-examples/tutorial/ejb/helloservice/target/helloservice.jar. Then the enterprise bean JAR file is deployed to GlassFish Server.

Upon deployment, GlassFish Server generates additional artifacts required for web service invocation, including the WSDL file.

To Test the Service without a Client

The GlassFish Server Administration Console allows you to test the methods of a web service endpoint. To test the sayHello method of HelloServiceBean, follow these steps.

1. Open the **Administration Console** by opening the following URL in a web browser:

```
http://localhost:4848/
```

2. In the navigation tree, select the **Applications** node.
3. In the **Applications** table, click the helloservice link.
4. In the **Modules and Components** table, click the **View Endpoint** link.
5. On the **Web Service Endpoint Information** page, click the **Tester** xref:

```
/HelloServiceBeanService/HelloServiceBean?Tester
```

6. On the **Web Service Test Links** page, click the non-secure link (the one that specifies port 8080).
7. On the **HelloServiceBeanService Web Service Tester** page, under **Methods**, enter a name as the parameter to the sayHello method.
8. Click **sayHello**.

The sayHello Method invocation page opens. Under Method returned, you'll see the response from the endpoint.

Using the Timer Service

Applications that model business work flows often rely on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 a.m. on May 23, in 30 days, or every 12 hours.

Enterprise bean timers are either programmatic timers or automatic timers. Programmatic timers are set by explicitly calling one of the timer creation methods of the `TimerService` interface. Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the `jakarta.ejb.Schedule` or `jakarta.ejb.Schedules` annotations.

Creating Calendar-Based Timer Expressions

Timers can be set according to a calendar-based schedule, expressed using a syntax similar to the UNIX `cron` utility. Both programmatic and automatic timers can use calendar-based timer expressions. Calendar-Based Timer Attributes shows the calendar-based timer attributes.

Calendar-Based Timer Attributes

Attribute	Description	Default Value	Allowable Values and Examples
`second`	One or more seconds within a minute	`0`	`0` to `59`. For example: `second="30"`.
`minute`	One or more minutes within an hour	`0`	`0` to `59`. For example: `minute="15"`.
`hour`	One or more hours within a day	`0`	`0` to `23`. For example: `hour="13"`.
`dayOfWeek`	One or more days within a week	`*`	`0` to `7` (both 0 and 7 refer to Sunday). For example: `dayOfWeek="3"`. `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`. For example: `dayOfWeek="Mon"`.
`dayOfMonth`	One or more days within a month	`*`	`1` to `31`. For example: `dayOfMonth="15"`. `-7` to `-1` (a negative number means the nth day or days before the end of the month). For example: `dayOfMonth="-3"`. `Last`. For example: `dayOfMonth="Last"`. [`1st`, `2nd`, `3rd`, `4th`, `5th`, `Last`] [`Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`]. For example: `dayOfMonth="2nd Fri"`.

Attribute	Description	Default Value	Allowable Values and Examples
month	One or more months within a year	*	1 to 12. For example: month="7". Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec. For example: month="July".
year	A particular calendar year	*	A four-digit calendar year. For example: year="2011".

Specifying Multiple Values in Calendar Expressions

You can specify multiple values in calendar expressions, as described in the following sections.

Using Wildcards in Calendar Expressions

Setting an attribute to an asterisk symbol (*) represents all allowable values for the attribute.

The following expression represents every minute:

```
minute="*"
```

The following expression represents every day of the week:

```
dayOfWeek="*"
```

Specifying a List of Values

To specify two or more values for an attribute, use a comma (,) to separate the values. A range of values is allowed as part of a list. Wildcards and intervals, however, are not allowed.

Duplicates within a list are ignored.

The following expression sets the day of the week to Tuesday and Thursday:

```
dayOfWeek="Tue, Thu"
```

The following expression represents 4:00 a.m., every hour from 9:00 a.m. to 5:00 p.m. using a range, and 10:00 p.m.:

```
hour="4,9-17,22"
```

Specifying a Range of Values

Use a dash character (-) to specify an inclusive range of values for an attribute. Members of a range

cannot be wildcards, lists, or intervals. A range of the form `x-x`, is equivalent to the single-valued expression `x`. A range of the form `x-y` where `x` is greater than `y` is equivalent to the expression `x-maximumvalue,minimumvalue-y`. That is, the expression begins at `x`, rolls over to the beginning of the allowable values, and continues up to `y`.

The following expression represents 9:00 a.m. to 5:00 p.m.:

```
hour="9-17"
```

The following expression represents Friday through Monday:

```
dayOfWeek="5-1"
```

The following expression represents the twenty-fifth day of the month to the end of the month, and the beginning of the month to the fifth day of the month:

```
dayOfMonth="25-5"
```

It is equivalent to the following expression:

```
dayOfMonth="25-Last,1-5"
```

Specifying Intervals

The forward slash (`/`) constrains an attribute to a starting point and an interval and is used to specify every `N` seconds, minutes, or hours within the minute, hour, or day. For an expression of the form `x/y`, `x` represents the starting point and `y` represents the interval. The wildcard character may be used in the `x` position of an interval and is equivalent to setting `x` to `0`.

Intervals may be set only for `second`, `minute`, and `hour` attributes.

The following expression represents every 10 minutes within the hour:

```
minute="*/10"
```

It is equivalent to:

```
minute="0,10,20,30,40,50"
```

The following expression represents every 2 hours starting at noon:

```
hour="12/2"
```

Programmatic Timers

When a programmatic timer expires (goes off), the container calls the method annotated `@Timeout` in the bean's implementation class. The `@Timeout` method contains the business logic that handles the timed event.

The @Timeout Method

Methods annotated `@Timeout` in the enterprise bean class must return `void` and optionally take a `jakarta.ejb.Timer` object as the only parameter. They may not throw application exceptions:

```
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

Creating Programmatic Timers

To create a timer, the bean invokes one of the `create` methods of the `TimerService` interface. These methods allow single-action, interval, or calendar-based timers to be created.

For single-action or interval timers, the expiration of the timer can be expressed as either a duration or an absolute time. The duration is expressed as a the number of milliseconds before a timeout event is triggered. To specify an absolute time, create a `java.util.Date` object and pass it to the `TimerService.createSingleActionTimer` or the `TimerService.createTimer` method.

The following code sets a programmatic timer that will expire in 1 minute (60,000 milliseconds):

```
long duration = 60000;
Timer timer =
    timerService.createSingleActionTimer(duration, new TimerConfig());
```

The following code sets a programmatic timer that will expire at 12:05 p.m. on May 1, 2015, specified as a `java.util.Date`:

```
SimpleDateFormatter formatter =
    new SimpleDateFormatter("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2015 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

For calendar-based timers, the expiration of the timer is expressed as a `jakarta.ejb.ScheduleExpression` object, passed as a parameter to the `TimerService.createCalendarTimer` method. The `ScheduleExpression` class represents calendar-based timer expressions and has methods that correspond to the attributes described in Creating Calendar-Based Timer Expressions.

The following code creates a programmatic timer using the `ScheduleExpression` helper class:

```
ScheduleExpression schedule = new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.createCalendarTimer(schedule);
```

For details on the method signatures, see the `TimerService` API documentation at https://jakarta.ee/specifications/platform/9/apidocs/jakarta/ejb/TimerService.html.

The bean described in The timersession Example creates a timer as follows:

```
Timer timer = timerService.createTimer(intervalDuration,
        "Created new programmatic timer");
```

In the `timersession` example, the method that calls `createTimer` is invoked in a business method, which is called by a client.

Timers are persistent by default. If the server is shut down or crashes, persistent timers are saved and will become active again when the server is restarted. If a persistent timer expires while the server is down, the container will call the `@Timeout` method when the server is restarted.

Nonpersistent programmatic timers are created by calling `TimerConfig.setPersistent(false)` and passing the `TimerConfig` object to one of the timer-creation methods.

The `Date` and `long` parameters of the `createTimer` methods represent time with the resolution of milliseconds. However, because the timer service is not intended for real-time applications, a callback to the `@Timeout` method might not occur with millisecond precision. The timer service is for business applications, which typically measure time in hours, days, or longer durations.

Automatic Timers

Automatic timers are created by the Enterprise Bean container when an enterprise bean that contains methods annotated with the `@Schedule` or `@Schedules` annotations is deployed. An enterprise bean can have multiple automatic timeout methods, unlike a programmatic timer, which allows only one method annotated with the `@Timeout` annotation in the enterprise bean class.

Automatic timers can be configured through annotations or through the `ejb-jar.xml` deployment descriptor.

Adding a `@Schedule` annotation on an enterprise bean marks that method as a timeout method according to the calendar schedule specified in the attributes of `@Schedule`.

The `@Schedule` annotation has elements that correspond to the calendar expressions detailed in Creating Calendar-Based Timer Expressions and the `persistent`, `info`, and `timezone` elements.

The optional `persistent` element takes a Boolean value and is used to specify whether the automatic timer should survive a server restart or crash. By default, all automatic timers are persistent.

The optional `timezone` element is used to specify that the automatic timer is associated with a particular time zone. If set, this element will evaluate all timer expressions in relation to the

specified time zone, regardless of the time zone in which the Enterprise Bean container is running. By default, all automatic timers set are in relation to the default time zone of the server.

The optional `info` element is used to set an informational description of the timer. A timer's information can be retrieved later by using `Timer.getInfo`.

The following timeout method uses `@Schedule` to set a timer that will expire every Sunday at midnight:

```
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }
```

The `@Schedules` annotation is used to specify multiple calendar-based timer expressions for a given timeout method.

The following timeout method uses the `@Schedules` annotation to set multiple calendar-based timer expressions. The first expression sets a timer to expire on the last day of every month. The second expression sets a timer to expire every Friday at 11:00 p.m.:

```
@Schedules ({
    @Schedule(dayOfMonth="Last"),
    @Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

Canceling and Saving Timers

Timers can be cancelled by the following events.

- When a single-event timer expires, the Enterprise Bean container calls the associated timeout method and then cancels the timer.
- When the bean invokes the `cancel` method of the `Timer` interface, the container cancels the timer.

If a method is invoked on a cancelled timer, the container throws the `jakarta.ejb.NoSuchObjectLocalException`.

To save a `Timer` object for future reference, invoke its `getHandle` method and store the `TimerHandle` object in a database. (A `TimerHandle` object is serializable.) To reinstantiate the `Timer` object, retrieve the handle from the database and invoke `getTimer` on the handle. A `TimerHandle` object cannot be passed as an argument of a method defined in a remote or web service interface. In other words, remote clients and web service clients cannot access a bean's `TimerHandle` object. Local clients, however, do not have this restriction.

Getting Timer Information

In addition to defining the `cancel` and `getHandle` methods, the `Timer` interface defines methods for obtaining information about timers:

```
public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();
```

The `getInfo` method returns the object that was the last parameter of the `createTimer` invocation. For example, in the `createTimer` code snippet of the preceding section, this information parameter is a `String` object with the value `created timer`.

To retrieve all of a bean's active timers, call the `getTimers` method of the `TimerService` interface. The `getTimers` method returns a collection of `Timer` objects.

Transactions and Timers

An enterprise bean usually creates a timer within a transaction. If this transaction is rolled back, the timer creation also is rolled back. Similarly, if a bean cancels a timer within a transaction that gets rolled back, the timer cancellation is rolled back. In this case, the timer's duration is reset as if the cancellation had never occurred.

In beans that use container-managed transactions, the `@Timeout` method usually has the `Required` or `RequiresNew` transaction attribute to preserve transaction integrity. With these attributes, the Enterprise Bean container begins the new transaction before calling the `@Timeout` method. If the transaction is rolled back, the container will call the `@Timeout` method at least one more time.

The timersession Example

The source code for this example is in the `jakartaee-examples/tutorial/ejb/timersession/src/main/java/` directory.

`TimerSessionBean` is a singleton session bean that shows how to set both an automatic timer and a programmatic timer. In the source code listing of `TimerSessionBean` that follows, the `setTimer` and `@Timeout` methods are used to set a programmatic timer. A `TimerService` instance is injected by the container when the bean is created. Because it's a business method, `setTimer` is exposed to the local, no-interface view of `TimerSessionBean` and can be invoked by the client. In this example, the client invokes `setTimer` with an interval duration of 8,000 milliseconds, or 8 seconds. The `setTimer` method creates a new timer by invoking the `createTimer` method of `TimerService`. Now that the timer is set, the Enterprise Bean container will invoke the `programmaticTimeout` method of `TimerSessionBean` when the timer expires, in about 8 seconds:

```
...
    public void setTimer(long intervalDuration) {
        logger.log(Level.INFO,
                "Setting a programmatic timeout for {0} milliseconds from now.",
                intervalDuration);
        Timer timer = timerService.createTimer(intervalDuration,
                "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
```

```
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }
...
```

`TimerSessionBean` also has an automatic timer and timeout method, `automaticTimeout`. The automatic timer is set to expire every 1 minute and is set by using a calendar-based timer expression in the `@Schedule` annotation:

```
...
    @Schedule(minute = "*/1", hour = "*", persistent = false)
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }
...
```

`TimerSessionBean` also has two business methods: `getLastProgrammaticTimeout` and `getLastAutomaticTimeout`. Clients call these methods to get the date and time of the last timeout for the programmatic timer and automatic timer, respectively.

Here's the source code for the `TimerSessionBean` class:

```
package ee.jakarta.tutorial.timersession.ejb;

import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import jakarta.annotation.Resource;
import jakarta.ejb.Schedule;
import jakarta.ejb.Singleton;
import jakarta.ejb.Startup;
import jakarta.ejb.Timeout;
import jakarta.ejb.Timer;
import jakarta.ejb.TimerService;

@Singleton
@Startup
public class TimerSessionBean {
    @Resource
    TimerService timerService;

    private Date lastProgrammaticTimeout;
    private Date lastAutomaticTimeout;

    private static final Logger logger =
            Logger.getLogger("timersession.ejb.TimerSessionBean");
```

```
    public void setTimer(long intervalDuration) {
        logger.log(Level.INFO,
                "Setting a programmatic timeout for {0} milliseconds from now.",
                intervalDuration);
        Timer timer = timerService.createTimer(intervalDuration,
                "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }

    @Schedule(minute = "*/1", hour = "*", persistent = false)
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occured");
    }

    public String getLastProgrammaticTimeout() {
        if (lastProgrammaticTimeout != null) {
            return lastProgrammaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastProgrammaticTimeout(Date lastTimeout) {
        this.lastProgrammaticTimeout = lastTimeout;
    }

    public String getLastAutomaticTimeout() {
        if (lastAutomaticTimeout != null) {
            return lastAutomaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastAutomaticTimeout(Date lastAutomaticTimeout) {
        this.lastAutomaticTimeout = lastAutomaticTimeout;
    }
}
```



GlassFish Server has a default minimum timeout value of 1,000 milliseconds, or 1 second. If you need to set the timeout value lower than 1,000 milliseconds, change the value of the Minimum Delivery Interval setting in the Administration Console.

To modify the minimum timeout value, in the Administration Console expand

> Configurations, then expand server-config, select EJB Container, and click the EJB Timer Service tab. Enter a new timeout value under Minimum Delivery Interval and click Save.
>
> The lowest practical value for `minimum-delivery-interval-in-millis` is around 10 milliseconds, owing to virtual machine constraints.

Running the timersession Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `timersession` example.

To Run the timersession Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/ejb
   ```

4. Select the `timersession` folder.
5. Click **Open Project**.
6. From the **Run** menu, choose **Run Project**.

   This builds and packages the application into a WAR file located at `jakartaee-examples/tutorial/ejb/timersession/target/timersession.war`, deploys this WAR file to your GlassFish Server instance, and then runs the web client.

To Build, Package, and Deploy the timersession Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/ejb/timersession/
   ```

3. Enter the following command:

   ```
   mvn install
   ```

   This builds and packages the application into a WAR file located at `jakartaee-examples/tutorial/ejb/timersession/target/timersession.war` and deploys this WAR file to your GlassFish Server instance.

To Run the Web Client

1. Open a web browser to the following URL:

   ```
   http://localhost:8080/timersession
   ```

2. Click Set Timer to set a programmatic timer.
3. Wait for a while and click the browser's Refresh button.

   You will see the date and time of the last programmatic and automatic timeouts.

   To see the messages that are logged when a timeout occurs, open the `server.log` file located in `domain-dir/logs/`.

Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A system exception indicates a problem with the services that support an application. For example, a connection to an external resource cannot be obtained, or an injected resource cannot be found. If it encounters a system-level problem, your enterprise bean should throw a `jakarta.ejb.EJBException`. Because the `EJBException` is a subclass of `RuntimeException`, you do not have to specify it in the `throws` clause of the method declaration. If a system exception is thrown, the Jakarta Enterprise Beans container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean's client program, but instead requires intervention by a system administrator.

An application exception signals an error in the business logic of an enterprise bean. Application exceptions are typically exceptions that you've coded yourself, such as the `BookException` thrown by the business methods of the `CartBean` example. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the Enterprise Bean container rolls back the transaction. However, if an application exception is thrown within a transaction, the container does not roll back the transaction.

Using Asynchronous Method Invocation in Session Beans



This section was last updated for Jakarta EE Version 9.1

This chapter discusses how to implement asynchronous business methods in session beans and call them from enterprise bean clients.

Asynchronous Method Invocation

Session beans can implement asynchronous methods, business methods where control is returned to the client by the enterprise bean container before the method is invoked on the session bean instance. Clients may then use the Java SE concurrency API to retrieve the result, cancel the

invocation, and check for exceptions. Asynchronous methods are typically used for long-running operations, for processor-intensive tasks, for background tasks, to increase application throughput, or to improve application response time if the method invocation result isn't required immediately.

When a session bean client invokes a typical non-asynchronous business method, control is not returned to the client until the method has completed. Clients calling asynchronous methods, however, immediately have control returned to them by the enterprise bean container. This allows the client to perform other tasks while the method invocation completes. If the method returns a result, the result is an implementation of the `java.util.concurrent.Future<V>` interface, where "V" is the result value type. The `Future<V>` interface defines methods the client may use to check whether the computation is completed, wait for the invocation to complete, retrieve the final result, and cancel the invocation.

Creating an Asynchronous Business Method

Annotate a business method with `jakarta.ejb.Asynchronous` to mark that method as an asynchronous method, or apply `@Asynchronous` at the class level to mark all the business methods of the session bean as asynchronous methods. Session bean methods that expose web services can't be asynchronous.

Asynchronous methods must return either `void` or an implementation of the `Future<V>` interface. Asynchronous methods that return `void` can't declare application exceptions, but if they return `Future<V>`, they may declare application exceptions. For example:

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException { ... }
```

This method will attempt to process the payment of an order, and return the status as a `String`. Even if the payment processor takes a long time, the client can continue working, and display the result when the processing finally completes.

The `jakarta.ejb.AsyncResult<V>` class is a concrete implementation of the `Future<V>` interface provided as a helper class for returning asynchronous results. `AsyncResult` has a constructor with the result as a parameter, making it easy to create `Future<V>` implementations. For example, the `processPayment` method would use `AsyncResult` to return the status as a `String`:

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    String status = ...;
    return new AsyncResult<String>(status);
}
```

The result is returned to the enterprise bean container, not directly to the client, and the enterprise bean container makes the result available to the client. The session bean can check whether the client requested that the invocation be cancelled by calling the `jakarta.ejb.SessionContext.wasCancelled` method. For example:

```
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    if (SessionContext.wasCancelled()) {
        // clean up
    } else {
        // process the payment
    }
    ...
}
```

Calling Asynchronous Methods from Enterprise Bean Clients

Session bean clients call asynchronous methods just like non-asynchronous business methods. If the asynchronous method returns a result, the client receives a `Future<V>` instance as soon as the method is invoked. This instance can be used to retrieve the final result, cancel the invocation, check whether the invocation has completed, check whether any exceptions were thrown during processing, and check whether the invocation was cancelled.

Retrieving the Final Result from an Asynchronous Method Invocation

The client may retrieve the result using one of the `Future<V>.get` methods. If processing hasn't been completed by the session bean handling the invocation, calling one of the `get` methods will result in the client halting execution until the invocation completes. Use the `Future<V>.isDone` method to determine whether processing has completed before calling one of the `get` methods.

The `get()` method returns the result as the type specified in the type value of the `Future<V>` instance. For example, calling `Future<String>.get()` will return a `String` object. If the method invocation was cancelled, calls to `get()` result in a `java.util.concurrent.CancellationException` being thrown. If the invocation resulted in an exception during processing by the session bean, calls to `get()` result in a `java.util.concurrent.ExecutionException` being thrown. The cause of the `ExecutionException` may be retrieved by calling the `ExecutionException.getCause` method.

The `get(long timeout, java.util.concurrent.TimeUnit unit)` method is similar to the `get()` method, but allows the client to set a timeout value. If the timeout value is exceeded, a `java.util.concurrent.TimeoutException` is thrown. See the Javadoc for the `TimeUnit` class for the available units of time to specify the timeout value.

Cancelling an Asynchronous Method Invocation

Call the `cancel(boolean mayInterruptIfRunning)` method on the `Future<V>` instance to attempt to cancel the method invocation. The `cancel` method returns `true` if the cancellation was successful and `false` if the method invocation cannot be cancelled.

When the invocation cannot be cancelled, the `mayInterruptIfRunning` parameter is used to alert the session bean instance on which the method invocation is running that the client attempted to cancel the invocation. If `mayInterruptIfRunning` is set to `true`, calls to `SessionContext.wasCancelled` by the session bean instance will return `true`. If `mayInterruptIfRunning` is to set `false`, calls to `SessionContext.wasCancelled` by the session bean instance will return `false`.

The `Future<V>.isCancelled` method is used to check whether the method invocation was cancelled before the asynchronous method invocation completed by calling `Future<V>.cancel`. The `isCancelled` method returns `true` if the invocation was cancelled.

Checking the Status of an Asynchronous Method Invocation

The `Future<V>.isDone` method returns `true` if the session bean instance completed processing the method invocation. The `isDone` method returns `true` if the asynchronous method invocation completed normally, was cancelled, or resulted in an exception. That is, `isDone` indicates only whether the session bean has completed processing the invocation.

The async Example Application

The `async` example demonstrates how to define an asynchronous business method on a session bean and call it from a web client. This example contains two modules.

- A web application (`async-war`) that contains a stateless session bean and a Jakarta Faces interface. The `MailerBean` stateless session bean defines an asynchronous method, `sendMessage`, which uses the Jakarta Mail API to send an email to an specified email address.
- An auxiliary Java SE program (`async-smtpd`) that simulates an SMTP server. This program listens on TCP port 3025 for SMTP requests and prints the email messages to the standard output (instead of delivering them).

The following section describes the architecture of the `async-war` module.

Architecture of the async-war Module

The `async-war` module consists of a single stateless session bean, `MailerBean`, and a Jakarta Faces web application front end that uses Facelets tags in XHTML files to display a form for users to enter the email address for the recipient of an email. The status of the email is updated when the email is finally sent.

The `MailerBean` session bean injects a Jakarta Mail resource used to send an email message to an address specified by the user. The message is created, modified, and sent using the Jakarta Mail API. The session bean looks like this:

```
@Named
@Stateless
public class MailerBean {
    @Resource(name="mail/myExampleSession")
    private Session session;
    private static final Logger logger =
            Logger.getLogger(MailerBean.class.getName());

    @Asynchronous
    public Future<String> sendMessage(String email) {
        String status;
        try {
            Properties properties = new Properties();
            properties.put("mail.smtp.port", "3025");
```

```
            session = Session.getInstance(properties);
            Message message = new MimeMessage(session);
            message.setFrom();
            message.setRecipients(Message.RecipientType.TO,
                    InternetAddress.parse(email, false));
            message.setSubject("Test message from async example");
            message.setHeader("X-Mailer", "Jakarta Mail");
            DateFormat dateFormatter = DateFormat
                    .getDateTimeInstance(DateFormat.LONG, DateFormat.SHORT);
            Date timeStamp = new Date();
            String messageBody = "This is a test message from the async "
                    + "example of the Jakarta EE Tutorial. It was sent on "
                    + dateFormatter.format(timeStamp)
                    + ".";
            message.setText(messageBody);
            message.setSentDate(timeStamp);
            Transport.send(message);
            status = "Sent";
            logger.log(Level.INFO, "Mail sent to {0}", email);
        } catch (MessagingException ex) {
            logger.severe("Error in sending message.");
            status = "Encountered an error: " + ex.getMessage();
            logger.severe(ex.getMessage());
        }
        return new AsyncResult<>(status);
    }
}
```

The injected Jakarta Mail resource can be configured through the GlassFish Server Administration Console, through a GlassFish Server administrative command, or through a resource configuration file packaged with the application. The resource configuration can be modified at runtime by the GlassFish Server administrator to use a different mail server or transport protocol.

The web client consists of a Facelets template, `template.xhtml`; two Facelets clients, `index.xhtml` and `response.xhtml`. The `index.xhtml` file contains a form for the target email address. When the user submits the form, the method is called that uses an injected instance of the `MailerBean` session bean to call `MailerBean.sendMessage`. The result is sent to the `response.xhtml` Facelets view.

Running the async Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `async` example.

To Run the async Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

```
jakartaee-examples/tutorial/ejb
```

4. Select the `async` folder, select **Open Required Projects**, and click **Open Project**.
5. In the **Projects** tab, right-click the `async-smtpd` project and select **Run**.

   The SMTP server simulator starts accepting connections. The async-smptd output tab shows the following message:

   ```
   [Test SMTP server listening on port 3025]
   ```

6. In the **Projects** tab, right-click the `async-war` project and select **Build**.

   This command configures the Jakarta Mail resource using a GlassFish Server administrative command and builds, packages, and deploys the `async-war` module.

7. Open the following URL in a web browser window:

   ```
   http://localhost:8080/async-war
   ```

8. In the web browser window, enter an email address and click Send email.

   The `MailerBean` stateless bean uses the Jakarta Mail API to deliver an email to the SMTP server simulator. The async-smptd output window in NetBeans IDE shows the resulting email message, including its headers.

9. To stop the SMTP server simulator, click the X button on the right side of the status bar in NetBeans IDE.
10. Delete the Jakarta Mail session resource.
    a. In the Services tab, expand the Servers node, then expand the GlassFish Server server node.
    b. Expand the Resources node, then expand the Jakarta Mail Sessions node.
    c. Right-click mail/myExampleSession and select Unregister.

To Run the async Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/ejb/async/async-smtpd/
   ```

3. Enter the following command to build and package the SMTP server simulator:

   ```
   mvn install
   ```

4. Enter the following command to start the STMP server simulator:

```
mvn exec:java
```

   The following message appears:

```
[Test SMTP server listening on port 3025]
```

   Keep this terminal window open.

5. In a new terminal window, go to:

```
jakartaee-examples/tutorial/ejb/async/async-war
```

6. Enter the following command to configure the Jakarta Mail resource and to build, package, and deploy the `async-war` module:

```
mvn install
```

7. Open the following URL in a web browser window:

```
http://localhost:8080/async-war
```

8. In the web browser window, enter an email address and click Send email.

   The `MailerBean` stateless bean uses the Jakarta Mail API to deliver an email to the SMTP server simulator. The resulting email message appears on the first terminal window, including its headers.

9. To stop the SMTP server simulator, close the terminal window in which you issued the command to start the STMP server simulator.
10. To delete the Jakarta Mail session resource, type the following command:

```
asadmin delete-mail-resource mail/myExampleSession
```

[1] Technically Jakarta Interceptors is an API separate from CDI, but in modern applications they are used exclusively with CDI, hence we use the term "CDI Interceptors" here.

[2] The portlet container is defined in JSR 286 Portlet specification which is not part of Jakarta EE.

[3] Some frameworks use Jakarta REST with non-blocking server such as Eclipse Vert.X or Netty.

Jakarta EE Platform

Jakarta Mail

Jakarta Mail Specification

Jakarta Messaging

Jakarta Messaging Concepts



This section was last updated for Jakarta EE Version 9.1

This chapter provides an introduction to Jakarta Messaging, a Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communication.

Jakarta Messaging Overview

This overview defines the concept of messaging, describes Jakarta Messaging and where it can be used, and explains how Jakarta Messaging works within the Jakarta EE platform.

What Is Messaging?

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is loosely coupled. A component sends a message to a destination, and the recipient can retrieve the message from the destination. What makes the communication loosely coupled is that the destination is all that the sender and receiver have in common. The sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

What Is Jakarta Messaging?

Jakarta Messaging is a Java API that allows applications to create, send, receive, and read messages. Jakarta Messaging defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

Jakarta Messaging minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of Messaging applications across providers.

Jakarta Messaging enables communication that is not only loosely coupled but also

- Asynchronous: A receiving client does not have to receive messages at the same time the sending client sends them. The sending client can send them and go on to other tasks; the receiving client can receive them much later.
- Reliable: A messaging provider that implements Jakarta Messaging can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The current version of the Jakarta Messaging specification is Version 3.0.

When Can You Use Jakarta Messaging?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances.

- The provider wants the components not to depend on information about other components' interfaces, so components can be easily replaced.
- The provider wants the application to run whether or not all components are up and running simultaneously.
- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

For example, components of an enterprise application for an automobile manufacturer can use Jakarta Messaging in situations like the following.

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so the factory can make more cars.
- The factory component can send a message to the parts components so the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
- Both the factory and the parts components can send messages to the accounting component to update budget numbers.
- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources. Figure 24, “Messaging in an Enterprise Application” illustrates how this simple example might work.

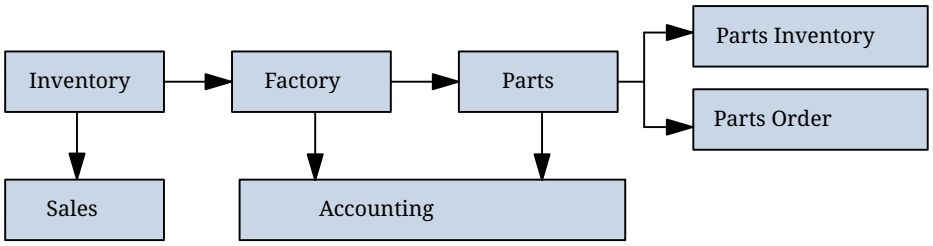


Figure 24. Messaging in an Enterprise Application

Manufacturing is only one example of how an enterprise can use the Jakarta Messaging API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

How Does Jakarta Messaging Work with the Jakarta EE Platform?

When JMS was first introduced, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems. Since that time, many vendors have adopted and implemented JMS, so a Jakarta Messaging product can now provide a complete messaging capability for an enterprise.

Jakarta Messaging is an integral part of the Jakarta EE platform, and application developers can use messaging with Jakarta EE components. Jakarta Messaging 2.0 is part of the Jakarta EE 8 release.

Jakarta Messaging in the Jakarta EE platform has the following features.

- Application clients, Jakarta Enterprise Beans components, and web components can send or synchronously receive a Jakarta Messaging message. Application clients can in addition set a message listener that allows Jakarta Messaging messages to be delivered to it asynchronously by being notified when a message is available.
- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the enterprise bean container. An application server typically pools message-driven beans to implement concurrent processing of messages.
- Message send and receive operations can participate in Jakarta transactions, which allow Jakarta Messaging operations and database accesses to take place within a single transaction.

Jakarta Messaging enhances the other parts of the Jakarta EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Jakarta EE components and legacy systems capable of messaging. A developer can easily add new behavior to a Jakarta EE application that has existing business events by adding a new message-driven bean to operate on specific business events. The Jakarta EE platform, moreover, enhances Jakarta Messaging by providing support for Jakarta Transactions and allowing for the concurrent consumption of messages. For more information, see the Jakarta Enterprise Beans specification, v4.0.

The Jakarta Messaging provider can be integrated with the application server using the Jakarta Connectors. You access the Messaging provider through a resource adapter. This capability allows vendors to create Messaging providers that can be plugged in to multiple application servers, and it allows application servers to support multiple Messaging providers. For more information, see the Jakarta Connectors specification, v2.0.

Basic Jakarta Messaging Concepts

This section introduces the most basic Jakarta Messaging concepts, the ones you must know to get started writing simple application clients that use the Jakarta Messaging.

The next section introduces the Jakarta Messaging programming model. Later sections cover more advanced concepts, including the ones you need in order to write applications that use message-driven beans.

Jakarta Messaging Architecture

A Jakarta Messaging application is composed of the following parts.

- A Jakarta Messaging provider is a messaging system that implements the Messaging interfaces and provides administrative and control features. A Jakarta EE implementation that supports the Jakarta EE Platform also includes a Messaging provider.
- Jakarta Messaging clients are the programs or components, written in the Java programming language, that produce and consume messages. Any Jakarta EE application component can act as a Messaging client.

  Java SE applications can also act as Jakarta Messaging clients; the Message Queue Developer's Guide for Java Clients in the GlassFish Server documentation (https://glassfish.org/documentation) explains how to make this work.
- Messages are the objects that communicate information between Jakarta Messaging clients.
- Administered objects are Jakarta Messaging objects configured for the use of clients. The two kinds of Jakarta Messaging administered objects are destinations and connection factories, described in Jakarta Messaging Administered Objects. An administrator can create objects that are available to all applications that use a particular installation of GlassFish Server; alternatively, a developer can use annotations to create objects that are specific to a particular application.

Figure 25, "Jakarta Messaging Architecture" illustrates the way these parts interact. Administrative tools or annotations allow you to bind destinations and connection factories into a JNDI namespace. A Messaging client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the Jakarta Messaging provider.

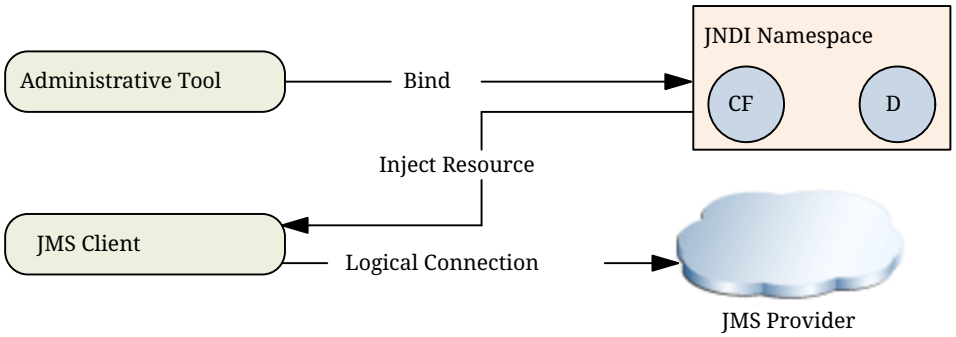


Figure 25. Jakarta Messaging Architecture

Messaging Styles

Before the Jakarta Messaging existed, most messaging products supported either the point-to-point or the publish/subscribe style of messaging. The Jakarta Messaging specification defines compliance for each style. A Messaging provider must implement both styles, and the Jakarta Messaging provides interfaces that are specific to each. The following subsections describe these messaging styles.

Jakarta Messaging, however, makes it unnecessary to use only one of the two styles. It allows you to use the same code to send and receive messages using either the PTP or the pub/sub style. The destinations you use remain specific to one style, and the behavior of the application will depend in part on whether you are using a queue or a topic. However, the code itself can be common to both styles, making your applications flexible and reusable. This tutorial describes and illustrates this coding approach, using the greatly simplified API provided by Jakarta Messaging 2.0.

Point-to-Point Messaging Style

A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.

PTP messaging, illustrated in Figure 26, “Point-to-Point Messaging”, has the following characteristics.

- Each message has only one consumer.
- The receiver can fetch the message whether or not it was running when the client sent the message.

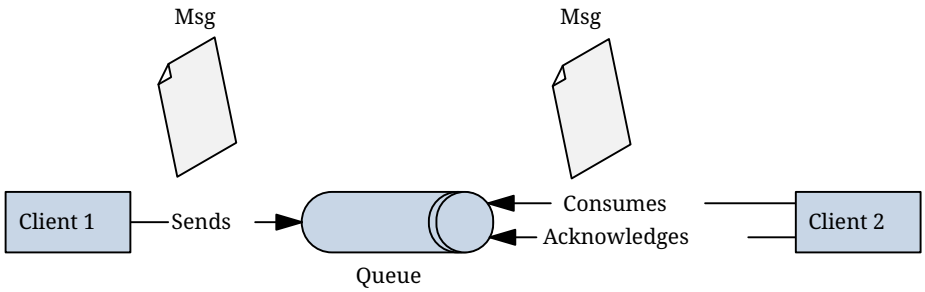


Figure 26. Point-to-Point Messaging

Use PTP messaging when every message you send must be processed successfully by one consumer.

Publish/Subscribe Messaging Style

In a publish/subscribe (pub/sub) product or application, clients address messages to a topic, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic. The system takes care of distributing the messages arriving from a topic’s multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to subscribers.

With pub/sub messaging, it is important to distinguish between the consumer that subscribes to a

topic (the subscriber) and the subscription that is created. The consumer is a Jakarta Messaging object within an application, while the subscription is an entity within the Jakarta Messaging provider. Normally, a topic can have many consumers, but a subscription has only one subscriber. It is possible, however, to create shared subscriptions; see Creating Shared Subscriptions for details. See Consuming Messages from Topics for details on the semantics of pub/sub messaging.

Pub/sub messaging has the following characteristics.

- Each message can have multiple consumers.
- A client that subscribes to a topic can consume only messages sent after the client has created a subscription, and the consumer must continue to be active in order for it to consume messages.

  The Jakarta Messaging relaxes this requirement to some extent by allowing applications to create durable subscriptions, which receive messages sent while the consumers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. For more information about durable subscriptions, see Creating Durable Subscriptions.

Use pub/sub messaging when each message can be processed by any number of consumers (or none). Figure 27, “Publish/Subscribe Messaging” illustrates pub/sub messaging.

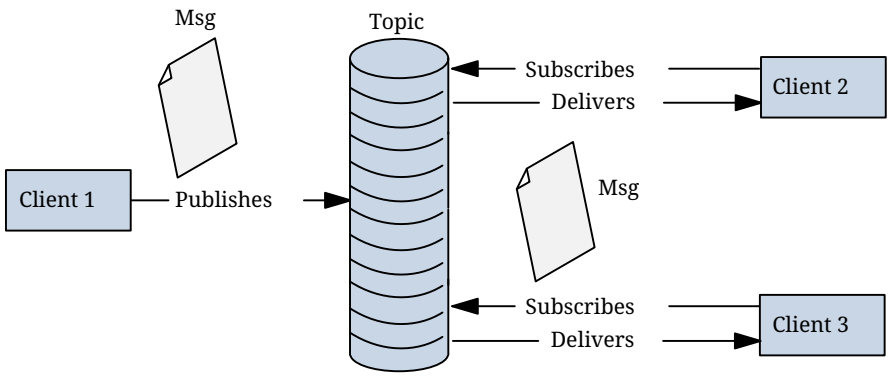


Figure 27. Publish/Subscribe Messaging

Message Consumption

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. However, the Jakarta Messaging specification uses this term in a more precise sense. Messages can be consumed in either of two ways.

- Synchronously: A consumer explicitly fetches the message from the destination by calling the `receive` method. The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- Asynchronously: An application client or a Java SE client can register a message listener with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener’s `onMessage` method, which acts on the contents of the message. In a Jakarta EE application, a message-driven bean serves as a message listener (it too has an `onMessage` method), but a client does not need to register it with a consumer.

Jakarta Messaging Programming Model

The basic building blocks of a Jakarta Messaging application are

- Administered objects: connection factories and destinations
- Connections
- Sessions
- `JMSContext` objects, which combine a connection and a session in one object
- Message producers
- Message consumers
- Messages

Figure 28, “Jakarta Messaging Programming Model” shows how all these objects fit together in a Messaging client application.

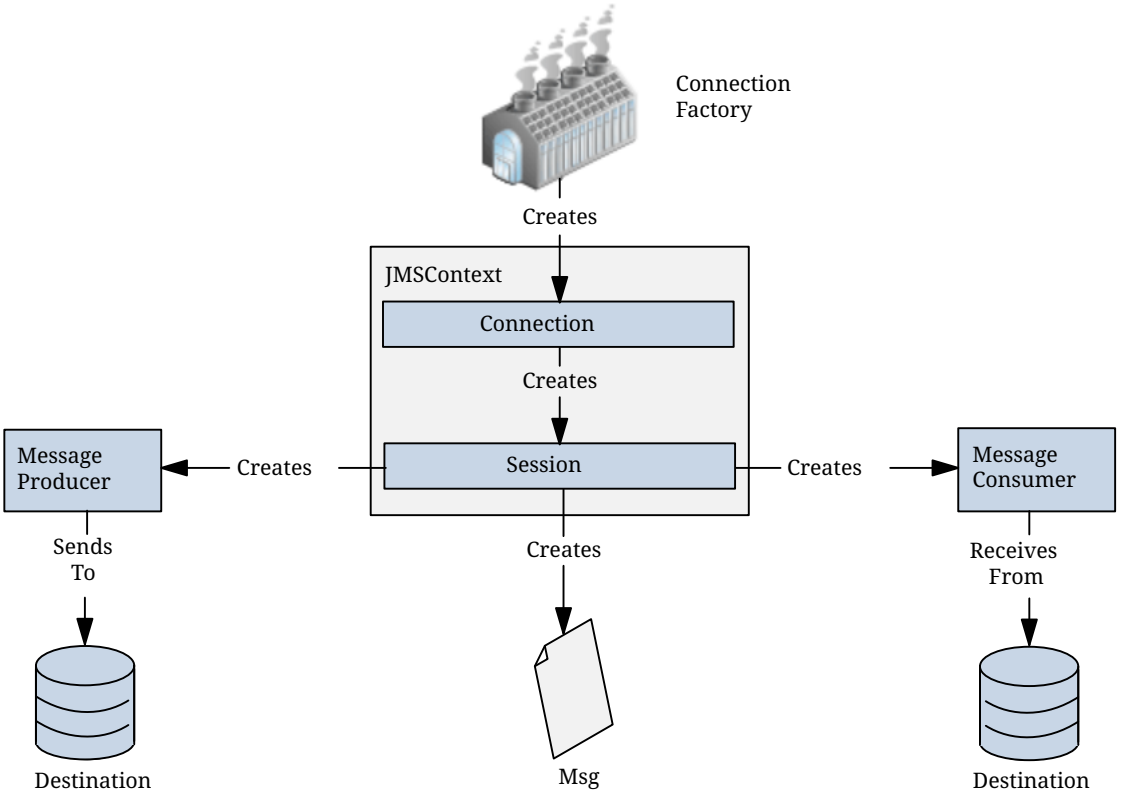


Figure 28. Jakarta Messaging Programming Model

Jakarta Messaging also provides queue browsers, objects that allow an application to browse messages on a queue.

This section describes all these objects briefly and provides sample commands and code snippets that show how to create and use the objects. The last subsection briefly describes Jakarta Messaging API exception handling.

Examples that show how to combine all these objects in applications appear in [messaging:jms-examples::jms-examples:::_jakarta_messaging_examples] beginning with Writing Simple Jakarta Messaging Applications. For more detail, see Jakarta Messaging documentation, part of the Jakarta EE API documentation.

Jakarta Messaging Administered Objects

Two parts of a Jakarta Messaging application, destinations and connection factories, are commonly maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of Jakarta Messaging to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

Messaging clients access administered objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of Jakarta Messaging. Ordinarily, an administrator configures administered objects in a JNDI namespace, and Messaging clients then access them by using resource injection.

With GlassFish Server, you can use the `asadmin create-jms-resource` command or the Administration Console to create Jakarta Messaging administered objects in the form of connector resources. You can also specify the resources in a file named `glassfish-resources.xml` that you can bundle with an application.

NetBeans IDE provides a wizard that allows you to create Jakarta Messaging resources for GlassFish Server. See Creating Jakarta Messaging Administered Objects for details.

The Jakarta EE platform specification allows a developer to create administered objects using annotations or deployment descriptor elements. Objects created in this way are specific to the application for which they are created. See Creating Resources for Jakarta EE Applications for details. Definitions in a deployment descriptor override those specified by annotations.

Jakarta Messaging Connection Factories

A connection factory is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface. To learn how to create connection factories, see Creating Jakarta Messaging Administered Objects.

At the beginning of a Messaging client program, you usually inject a connection factory resource into a `ConnectionFactory` object. A Jakarta EE server must provide a Jakarta Messaging connection factory with the logical JNDI name `java:comp/DefaultJMSConnectionFactory`. The actual JNDI name will be implementation-specific.

For example, the following code fragment looks up the default Jakarta Messaging connection factory and assigns it to a `ConnectionFactory` object:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

Jakarta Messaging Destinations

A destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging style, destinations are called queues. In the pub/sub

messaging style, destinations are called topics. A Jakarta Messaging application can use multiple queues or topics (or both). To learn how to create destination resources, see Creating Jakarta Messaging Administered Objects.

To create a destination using GlassFish Server, you create a Jakarta Messaging destination resource that specifies a JNDI name for the destination.

In the GlassFish Server implementation of Jakarta Messaging, each destination resource refers to a physical destination. You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resource.

In addition to injecting a connection factory resource into a client program, you usually inject a destination resource. Unlike connection factories, destinations are specific to either the PTP or pub/sub messaging style. To create an application that allows you to use the same code for both topics and queues, you assign the destination to a `Destination` object.

The following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace:

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;

@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

In a Jakarta EE application, Jakarta Messaging administered objects are normally placed in the `jms` naming subcontext.

With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the `ConnectionFactory` interface, you can inject a `QueueConnectionFactory` resource and use it with a `Topic`, and you can inject a `TopicConnectionFactory` resource and use it with a `Queue`. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

Connections

A connection encapsulates a virtual connection with a Messaging provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.



In the Jakarta EE platform, the ability to create multiple sessions from a single connection is limited to application clients. In web and enterprise bean components, a connection can create no more than one session.

You normally create a connection by creating a `JMSContext` object. See JMSContext Objects for details.

Sessions

A session is a single-threaded context for producing and consuming messages.

You normally create a session (as well as a connection) by creating a `JMSContext` object. See JMSContext Objects for details. You use sessions to create message producers, message consumers, messages, queue browsers, and temporary destinations.

Sessions serialize the execution of message listeners; for details, see Jakarta Messaging Message Listeners.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work. For details, see Using Jakarta Messaging Local Transactions.

JMSContext Objects

A `JMSContext` object combines a connection and a session in a single object. That is, it provides both an active connection to a Messaging provider and a single-threaded context for sending and receiving messages.

You use the `JMSContext` to create the following objects:

- Message producers
- Message consumers
- Messages
- Queue browsers
- Temporary queues and topics (see Creating Temporary Destinations)

You can create a `JMSContext` in a `try`-with-resources block.

To create a `JMSContext`, call the `createContext` method on the connection factory:

```
JMSContext context = connectionFactory.createContext();
```

When called with no arguments from an application client or a Java SE client, or from the Jakarta EE web or Enterprise Beans container when there is no active Jakarta Transactions transaction in progress, the `createContext` method creates a non-transacted session with an acknowledgment mode of `JMSContext.AUTO_ACKNOWLEDGE`. When called with no arguments from the web or Enterprise Beans container when there is an active JTA transaction in progress, the `createContext` method creates a transacted session. For information about the way Jakarta Messaging transactions work in Jakarta EE applications, see Using Jakarta Messaging in Jakarta EE Applications.

From an application client or a Java SE client, you can also call the `createContext` method with the argument `JMSContext.SESSION_TRANSACTED` to create a transacted session:

```
JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```

The session uses local transactions; see Using Jakarta Messaging Local Transactions for details.

Alternatively, you can specify a non-default acknowledgment mode; see Controlling Message Acknowledgment for more information.

When you use a `JMSContext`, message delivery normally begins as soon as you create a consumer. See Jakarta Messaging Message Consumers for more information.

If you create a `JMSContext` in a `try`-with-resources block, you do not need to close it explicitly. It will be closed when the `try` block comes to an end. Make sure that your application completes all its Jakarta Messaging activity within the `try`-with-resources block. If you do not use a `try`-with-resources block, you must call the `close` method on the `JMSContext` to close the connection when the application has finished its work.

Jakarta Messaging Message Producers

A message producer is an object that is created by a `JMSContext` or a session and used for sending messages to a destination. A message producer created by a `JMSContext` implements the `JMSProducer` interface. You could create it this way:

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSProducer producer = context.createProducer();
    ...
}
```

However, a `JMSProducer` is a lightweight object that does not consume significant resources. For this reason, you do not need to save the `JMSProducer` in a variable; you can create a new one each time you send a message. You send messages to a specific destination by using the `send` method. For example:

```
context.createProducer().send(dest, message);
```

You can create the message in a variable before sending it, as shown here, or you can create it within the `send` call. See Jakarta Messaging Messages for more information.

Jakarta Messaging Message Consumers

A message consumer is an object that is created by a `JMSContext` or a session and used for receiving messages sent to a destination. A message producer created by a `JMSContext` implements the `JMSConsumer` interface. The simplest way to create a message consumer is to use the `JMSContext.createConsumer` method:

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSConsumer consumer = context.createConsumer(dest);
    ...
}
```

A message consumer allows a Messaging client to register interest in a destination with a Messaging provider. The Jakarta Messaging provider manages the delivery of messages from a destination to the registered consumers of the destination.

When you use a `JMSContext` to create a message consumer, message delivery begins as soon as you have created the consumer. You can disable this behavior by calling `setAutoStart(false)` when you create the `JMSContext` and then calling the `start` method explicitly to start message delivery. If you want to stop message delivery temporarily without closing the connection, you can call the `stop` method; to restart message delivery, call `start`.

You use the `receive` method to consume a message synchronously. You can use this method at any time after you create the consumer.

If you specify no arguments or an argument of `0`, the method blocks indefinitely until a message arrives:

```
Message m = consumer.receive();
Message m = consumer.receive(0);
```

For a simple client, this may not matter. But if it is possible that a message might not be available, use a synchronous receive with a timeout: Call the `receive` method with a timeout argument greater than `0`. One second is a recommended timeout value:

```
Message m = consumer.receive(1000); // time out after a second
```

To enable asynchronous message delivery from an application client or a Java SE client, you use a message listener, as described in the next section.

You can use the `JMSContext.createDurableConsumer` method to create a durable topic subscription. This method is valid only if you are using a topic. For details, see Creating Durable Subscriptions. For topics, you can also create shared consumers; see Creating Shared Subscriptions.

Jakarta Messaging Message Listeners

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, you define the actions to be taken when a message arrives.

From an application client or a Java SE client, you register the message listener with a specific message consumer by using the `setMessageListener` method. For example, if you define a class named `Listener` that implements the `MessageListener` interface, you can register the message listener as follows:

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

When message delivery begins, the Messaging provider automatically calls the message listener's

`onMessage` method whenever a message is delivered. The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to another message subtype as needed (see Message Bodies).

In the Jakarta EE web or Enterprise Beans container, you use message-driven beans for asynchronous message delivery. A message-driven bean also implements the `MessageListener` interface and contains an `onMessage` method. For details, see Using Message-Driven Beans to Receive Messages Asynchronously.

Your `onMessage` method should handle all exceptions. Throwing a `RuntimeException` is considered a programming error.

For a simple example of the use of a message listener, see Using a Message Listener for Asynchronous Message Delivery. [messaging:jms-examples::jms-examples:::_jakarta_messaging_examples] contains several more examples of message listeners and message-driven beans.

Jakarta Messaging Message Selectors

If your messaging application needs to filter the messages it receives, you can use a Jakarta Messaging message selector, which allows a message consumer for a destination to specify the messages that interest it. Message selectors assign the work of filtering messages to the Messaging provider rather than to the application. For an example of an application that uses a message selector, see Sending Messages from a Session Bean to an MDB.

A message selector is a `String` that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the example selects any message that has a `NewsType` property that is set to the value `'Sports'` or `'Opinion'`:

```
NewsType = 'Sports' OR NewsType = 'Opinion'
```

The `createConsumer` and `createDurableConsumer` methods, as well as the methods for creating shared consumers, allow you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector. (See Message Headers and Message Properties.) A message selector cannot select messages on the basis of the content of the message body.

Consuming Messages from Topics

The semantics of consuming messages from topics are more complex than the semantics of consuming messages from queues.

An application consumes messages from a topic by creating a subscription on that topic and creating a consumer on that subscription. Subscriptions may be durable or nondurable, and they may be shared or unshared.

A subscription may be thought of as an entity within the Messaging provider itself, whereas a

consumer is a Jakarta Messaging object within the application.

A subscription will receive a copy of every message that is sent to the topic after the subscription is created, unless a message selector is specified. If a message selector is specified, only those messages whose properties match the message selector will be added to the subscription.

Unshared subscriptions are restricted to a single consumer. In this case, all the messages in the subscription are delivered to that consumer. Shared subscriptions allow multiple consumers. In this case, each message in the subscription is delivered to only one consumer. Jakarta Messaging does not define how messages are distributed between multiple consumers on the same subscription.

Subscriptions may be durable or nondurable.

A nondurable subscription exists only as long as there is an active consumer on the subscription. This means that any messages sent to the topic will be added to the subscription only while a consumer exists and is not closed.

A nondurable subscription may be either unshared or shared.

- An unshared nondurable subscription does not have a name and may have only a single consumer object associated with it. It is created automatically when the consumer object is created. It is not persisted and is deleted automatically when the consumer object is closed.

  The `JMSContext.createConsumer` method creates a consumer on an unshared nondurable subscription if a topic is specified as the destination.

- A shared nondurable subscription is identified by name and an optional client identifier, and may have several consumer objects consuming messages from it. It is created automatically when the first consumer object is created. It is not persisted and is deleted automatically when the last consumer object is closed. See Creating Shared Subscriptions for more information.

At the cost of higher overhead, a subscription may be durable. A durable subscription is persisted and continues to accumulate messages until explicitly deleted, even if there are no consumer objects consuming messages from it. See Creating Durable Subscriptions for details.

Creating Durable Subscriptions

To ensure that a pub/sub application receives all sent messages, use durable subscriptions for the consumers on the topic.

Like a nondurable subscription, a durable subscription may be either unshared or shared.

- An unshared durable subscription is identified by name and client identifier (which must be set) and may have only a single consumer object associated with it.
- A shared durable subscription is identified by name and an optional client identifier, and may have several consumer objects consuming messages from it.

A durable subscription that exists but that does not currently have a non-closed consumer object associated with it is described as being inactive.

You can use the `JMSContext.createDurableConsumer` method to create a consumer on an unshared durable subscription. An unshared durable subscription can have only one active consumer at a time.

A consumer identifies the durable subscription from which it consumes messages by specifying a unique identity that is retained by the Messaging provider. Subsequent consumer objects that have the same identity resume the subscription in the state in which it was left by the preceding consumer. If a durable subscription has no active consumer, the Messaging provider retains the subscription's messages until they are received by the subscription or until they expire.

You establish the unique identity of an unshared durable subscription by setting the following:

- A client ID for the connection
- A topic and a subscription name for the subscription

You can set the client ID administratively for a client-specific connection factory using either the command line or the Administration Console. (In an application client or a Java SE client, you can instead call `JMSContext.setClientID`.)

After using this connection factory to create the `JMSContext`, you call the `createDurableConsumer` method with two arguments: the topic and a string that specifies the name of the subscription:

```
String subName = "MySub";
JMSConsumer consumer = context.createDurableConsumer(myTopic, subName);
```

The subscription becomes active after you create the consumer. Later, you might close the consumer:

```
consumer.close();
```

The Messaging provider stores the messages sent to the topic, as it would store messages sent to a queue. If the program or another application calls `createDurableConsumer` using the same connection factory and its client ID, the same topic, and the same subscription name, then the subscription is reactivated and the Messaging provider delivers the messages that were sent while the subscription was inactive.

To delete a durable subscription, first close the consumer, then call the `unsubscribe` method with the subscription name as the argument:

```
consumer.close();
context.unsubscribe(subName);
```

The `unsubscribe` method deletes the state the provider maintains for the subscription.

Figure 30, "Consumers on a Durable Subscription" show the difference between a nondurable and a durable subscription. With an ordinary, nondurable subscription, the consumer and the

subscription begin and end at the same point and are, in effect, identical. When the consumer is closed, the subscription also ends. Here, `create` stands for a call to `JMSContext.createConsumer` with a `Topic` argument, and `close` stands for a call to `JMSConsumer.close`. Any messages sent to the topic between the time of the first `close` and the time of the second `create` are not added to either subscription. In Figure 29, "Nondurable Subscriptions and Consumers", the consumers receive messages M1, M2, M5, and M6, but they do not receive messages M3 and M4.

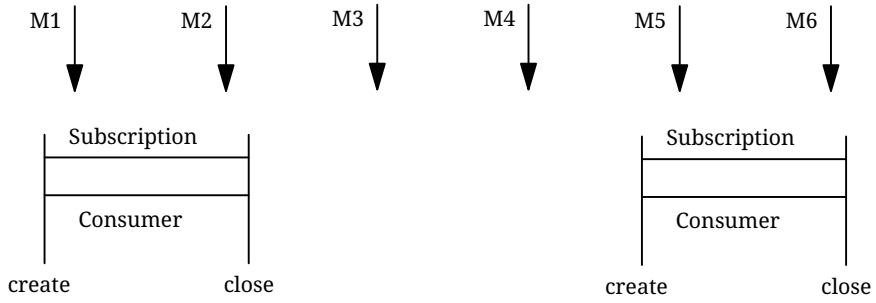


Figure 29. Nondurable Subscriptions and Consumers

With a durable subscription, the consumer can be closed and re-created, but the subscription continues to exist and to hold messages until the application calls the `unsubscribe` method. In Figure 30, "Consumers on a Durable Subscription", `create` stands for a call to `JMSContext.createDurableConsumer`, `close` stands for a call to `JMSConsumer.close`, and `unsubscribe` stands for a call to `JMSContext.unsubscribe`. Messages sent after the first consumer is closed are received when the second consumer is created (on the same durable subscription), so even though messages M2, M4, and M5 arrive while there is no consumer, they are not lost.

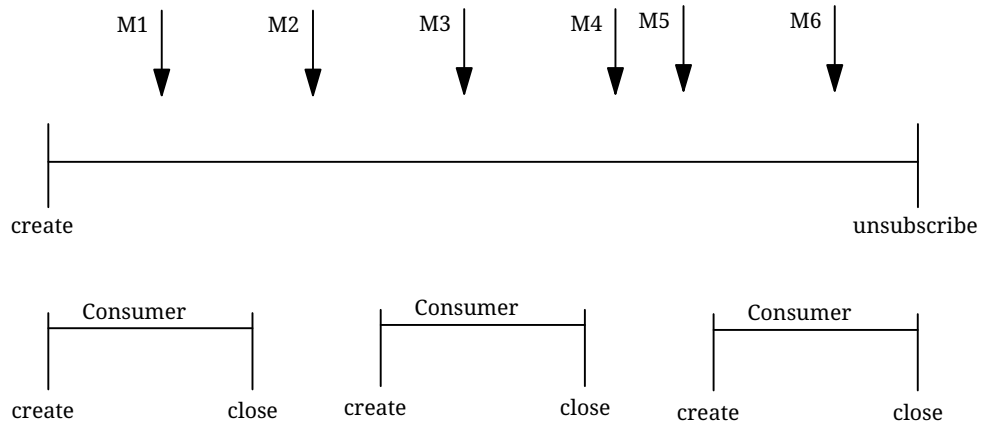


Figure 30. Consumers on a Durable Subscription

A shared durable subscription allows you to use multiple consumers to receive messages from a durable subscription. If you use a shared durable subscription, the connection factory you use does not need to have a client identifier. To create a shared durable subscription, call the `JMSContext.createSharedDurableConsumer` method, specifying the topic and subscription name:

```
JMSConsumer consumer =
        context.createSharedDurableConsumer(topic, "MakeItLast");
```

See Acknowledging Messages, Using Durable Subscriptions, Using Shared Durable Subscriptions, and Sending Messages from a Session Bean to an MDB for examples of Jakarta EE applications that

use durable subscriptions.

Creating Shared Subscriptions

A topic subscription created by the `createConsumer` or `createDurableConsumer` method can have only one consumer (although a topic can have many). Multiple clients consuming from the same topic have, by definition, multiple subscriptions to the topic, and all the clients receive all the messages sent to the topic (unless they filter them with message selectors).

It is, however, possible to create a nondurable shared subscription to a topic by using the `createSharedConsumer` method and specifying not only a destination but a subscription name:

```
consumer = context.createSharedConsumer(topicName, "SubName");
```

With a shared subscription, messages will be distributed among multiple clients that use the same topic and subscription name. Each message sent to the topic will be added to every subscription (subject to any message selectors), but each message added to a subscription will be delivered to only one of the consumers on that subscription, so it will be received by only one of the clients. A shared subscription can be useful if you want to share the message load among several consumers on the subscription rather than having just one consumer on the subscription receive each message. This feature can improve the scalability of Jakarta EE application client applications and Java SE applications. (Message-driven beans share the work of processing messages from a topic among multiple threads.)

See Using Shared Nondurable Subscriptions for a simple example of using shared nondurable consumers.

You can also create shared durable subscriptions by using the `JMSContext.createSharedDurableConsumer` method. For details, see Creating Durable Subscriptions.

Jakarta Messaging Messages

The ultimate purpose of a Jakarta Messaging application is to produce and consume messages that can then be used by other software applications. Jakarta Messaging messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-Jakarta Messaging applications on heterogeneous platforms.

A Jakarta Messaging message can have three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the `Message` interface in the API documentation. For a list of possible message types, see Message Bodies.

Message Headers

A Jakarta Messaging message header contains a number of predefined fields that contain values used by both clients and providers to identify and route messages. How Jakarta Messaging Message Header Field Values Are Set lists and describes the Jakarta Messaging message header fields and

indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field `JMSMessageID`. The value of another header field, `JMSDestination`, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the `Message` interface. Some header fields are intended to be set by a client, but many are set automatically by the `send` method, which overrides any client-set values.

How Jakarta Messaging Message Header Field Values Are Set

Header Field	Description	Set By
`JMSDestination`	Destination to which the message is being sent	JMS provider `send` method
`JMSDeliveryMode`	Delivery mode specified when the message was sent (see Specifying Message Persistence)	Messaging provider `send` method
`JMSDeliveryTime`	The time the message was sent plus the delivery delay specified when the message was sent (see Specifying a Delivery Delay	JMS provider `send` method
`JMSExpiration`	Expiration time of the message (see Allowing Messages to Expire)	JMS provider `send` method
`JMSPriority`	The priority of the message (see Setting Message Priority Levels)	Jakarta Messaging provider `send` method
`JMSMessageID`	Value that uniquely identifies each message sent by a provider	Messaging provider `send` method
`JMSTimestamp`	The time the message was handed off to a provider to be sent	Messaging provider `send` method
`JMSCorrelationID`	Value that links one message to another; commonly the `JMSMessageID` value is used	Client application
`JMSReplyTo`	Destination where replies to the message should be sent	Client application
`JMSType`	Type identifier supplied by client application	Client application
`JMSRedelivered`	Whether the message is being redelivered	Jakarta Messaging provider prior to delivery

Message Properties

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors (see Jakarta Messaging Message Selectors). For an example of setting a property to be used as a message selector, see Sending Messages from a

Session Bean to an MDB.

Jakarta Messaging provides some predefined property names that begin with `JMSX`. A Messaging provider is required to implement only one of these, `JMSXDeliveryCount` (which specifies the number of times a message has been delivered); the rest are optional. The use of these predefined properties or of user-defined properties in applications is optional.

Message Bodies

Jakarta Messaging defines six different types of messages. Each message type corresponds to a different message body. These message types allow you to send and receive data in many different forms. Jakarta Messaging Message Types describes these message types.

Jakarta Messaging Message Types

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an XML file).
`MapMessage`	A set of name-value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object in the Java programming language.
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

Jakarta Messaging provides methods for creating messages of each type and for filling in their contents. For example, to create and send a `TextMessage`, you might use the following statements:

```
TextMessage message = context.createTextMessage();
message.setText(msg_text);     // msg_text is a String
context.createProducer().send(message);
```

At the consuming end, a message arrives as a generic `Message` object. You can then cast the object to the appropriate message type and use more specific methods to access the body and extract the message contents (and its headers and properties if needed). For example, you might use the stream-oriented read methods of `BytesMessage`. You must always cast to the appropriate message type to retrieve the body of a `StreamMessage`.

Instead of casting the message to a message type, you can call the `getBody` method on the `Message`, specifying the type of the message as an argument. For example, you can retrieve a `TextMessage` as a `String`. The following code fragment uses the `getBody` method:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    String message = m.getBody(String.class);
    System.out.println("Reading message: " + message);
} else {
    // Handle error or process another message type
}
```

Jakarta Messaging provides shortcuts for creating and receiving a `TextMessage`, `BytesMessage`, `MapMessage`, or `ObjectMessage`. For example, you do not have to wrap a string in a `TextMessage`; instead, you can send and receive the string directly. For example, you can send a string as follows:

```
String message = "This is a message";
context.createProducer().send(dest, message);
```

You can receive the message by using the `receiveBody` method:

```
String message = receiver.receiveBody(String.class);
```

You can use the `receiveBody` method to receive any type of message except `StreamMessage` and `Message`, as long as the body of the message can be assigned to a particular type.

An empty `Message` can be useful if you want to send a message that is simply a signal to the application. Some of the examples in [messaging:jms-examples::jms-examples:::_jakarta_messaging_examples], send an empty message after sending a series of text messages. For example:

```
context.createProducer().send(dest, context.createMessage());
```

The consumer code can then interpret a non-text message as a signal that all the messages sent have now been received.

The examples in [messaging:jms-examples::jms-examples:::_jakarta_messaging_examples], use messages of type `TextMessage`, `MapMessage`, and `Message`.

Jakarta Messaging Queue Browsers

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. Jakarta Messaging provides a `QueueBrowser` object that allows you to browse the messages in the queue and display the header values for each message. To create a `QueueBrowser` object, use the `JMSContext.createBrowser` method.

For example:

```
QueueBrowser browser = context.createBrowser(queue);
```

See Browsing Messages on a Queue for an example of using a `QueueBrowser` object.

The `createBrowser` method allows you to specify a message selector as a second argument when you create a `QueueBrowser`. For information on message selectors, see Jakarta Messaging Message Selectors.

Jakarta Messaging provides no mechanism for browsing a topic. Messages usually disappear from a topic as soon as they appear: If there are no message consumers to consume them, the Messaging provider removes them. Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, Jakarta Messaging does not define any facility for examining them.

Jakarta Messaging Exception Handling

The root class for all checked exceptions in Jakarta Messaging is `JMSException`. The root cause for all unchecked exceptions in the Jakarta Messaging API is `JMSRuntimeException`.

Catching `JMSException` and `JMSRuntimeException` provides a generic way of handling all exceptions related to Jakarta Messaging.

The `JMSException` and `JMSRuntimeException` classes include the following subclasses, described in the API documentation:

- `IllegalStateException`, `IllegalStateRuntimeException`
- `InvalidClientIDException`, `InvalidClientIDRuntimeException`
- `InvalidDestinationException`, `InvalidDestinationRuntimeException`
- `InvalidSelectorException`, `InvalidSelectorRuntimeException`
- `JMSSecurityException`, `JMSSecurityRuntimeException`
- `MessageEOFException`
- `MessageFormatException`, `MessageFormatRuntimeException`
- `MessageNotReadableException`
- `MessageNotWriteableException`, `MessageNotWriteableRuntimeException`
- `ResourceAllocationException`, `ResourceAllocationRuntimeException`
- `TransactionInProgressException`, `TransactionInProgressRuntimeException`
- `TransactionRolledBackException`, `TransactionRolledBackRuntimeException`

All the examples in the tutorial catch and handle `JMSException` or `JMSRuntimeException` when it is appropriate to do so.

Using Advanced Jakarta Messaging Features

This section explains how to use features of Jakarta Messaging to achieve the level of reliability and performance your application requires. Many people use Jakarta Messaging in their applications because they cannot tolerate dropped or duplicate messages and because they require that every message be received once and only once. Jakarta Messaging provides this functionality.

The most reliable way to produce a message is to send a `PERSISTENT` message, and to do so within a transaction.

Jakarta Messaging messages are `PERSISTENT` by default; `PERSISTENT` messages will not be lost in the event of Messaging provider failure. For details, see Specifying Message Persistence.

Transactions allow multiple messages to be sent or received in an atomic operation.In the Jakarta EE platform they also allow message sends and receives to be combined with database reads and writes in an atomic transaction. A transaction is a unit of work into which you can group a series of operations, such as message sends and receives, so that the operations either all succeed or all fail. For details, see Using Jakarta Messaging Local Transactions.

The most reliable way to consume a message is to do so within a transaction, either from a queue or from a durable subscription to a topic. For details, see Creating Durable Subscriptions, Creating Temporary Destinations, and Using Jakarta Messaging Local Transactions.

Some features primarily allow an application to improve performance. For example, you can set messages to expire after a certain length of time (see Allowing Messages to Expire), so that consumers do not receive unnecessary outdated information. You can send messages asynchronously; see Sending Messages Asynchronously.

You can also specify various levels of control over message acknowledgment; see Controlling Message Acknowledgment.

Other features can provide useful capabilities unrelated to reliability. For example, you can create temporary destinations that last only for the duration of the connection in which they are created. See Creating Temporary Destinations for details.

The following sections describe these features as they apply to application clients or Java SE clients. Some of the features work differently in the Jakarta EE web or enterprise bean container; in these cases, the differences are noted here and are explained in detail in Using Jakarta Messaging in Jakarta EE Applications.

Controlling Message Acknowledgment

Until a Jakarta Messaging message has been acknowledged, it is not considered to be successfully consumed. The successful consumption of a message ordinarily takes place in three stages.

1. The client receives the message.
2. The client processes the message.
3. The message is acknowledged.
   Acknowledgment is initiated either by the Messaging provider or by the client, depending on the session acknowledgment mode.

In locally transacted sessions (see Using Jakarta Messaging Local Transactions), a message is acknowledged when the session is committed. If a transaction is rolled back, all consumed messages are redelivered.

In a Jakarta transaction (in the Jakarta EE web or enterprise bean container) a message is acknowledged when the transaction is committed.

In nontransacted sessions, when and how a message is acknowledged depend on a value that may be specified as an argument of the `createContext` method. The possible argument values are as follows.

- `JMSContext.AUTO_ACKNOWLEDGE`: This setting is the default for application clients and Java SE clients. The `JMSContext` automatically acknowledges a client's receipt of a message either when the client has successfully returned from a call to `receive` or when the `MessageListener` it has called to process the message returns successfully.

  A synchronous receive in a `JMSContext` that is configured to use auto-acknowledgment is the one exception to the rule that message consumption is a three-stage process as described earlier. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.

- `JMSContext.CLIENT_ACKNOWLEDGE`: A client acknowledges a message by calling the message's `acknowledge` method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

  In the Jakarta EE platform, the `JMSContext.CLIENT_ACKNOWLEDGE` setting can be used only in an application client, not in a web component or enterprise bean.

- `JMSContext.DUPS_OK_ACKNOWLEDGE`: This option instructs the `JMSContext` to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the Messaging provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If the Messaging provider redelivers a message, it must set the value of the `JMSRedelivered` message header to `true`.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

If messages have been received from a queue but not acknowledged when a `JMSContext` is closed, the Messaging provider retains them and redelivers them when a consumer next accesses the queue. The provider also retains unacknowledged messages if an application closes a `JMSContext` that has been consuming messages from a durable subscription. (See Creating Durable Subscriptions.) Unacknowledged messages that have been received from a nondurable subscription will be dropped when the `JMSContext` is closed.

If you use a queue or a durable subscription, you can use the `JMSContext.recover` method to stop a nontransacted `JMSContext` and restart it with its first unacknowledged message. In effect, the `JMSContext`'s series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered, if

messages have expired or if higher-priority messages have arrived. For a consumer on a nondurable subscription, the provider may drop unacknowledged messages when the `JMSContext.recover` method is called.

The sample program in Acknowledging Messages demonstrates two ways to ensure that a message will not be acknowledged until processing of the message is complete.

Specifying Options for Sending Messages

You can set a number of options when you send a message. These options enable you to perform the tasks described in the following topics:

- Specifying Message Persistence - Specify that messages are persistent, meaning they must not be lost in the event of a provider failure.
- Setting Message Priority Levels - Set priority levels for messages, which can affect the order in which the messages are delivered.
- Allowing Messages to Expire - Specify an expiration time for messages so they will not be delivered if they are obsolete.
- Specifying a Delivery Delay - Specify a delivery delay for messages so that they will not be delivered until a specified amount of time has expired.
- Using JMSProducer Method Chaining - Method chaining allows you to specify more than one of these options when you create a producer and call the `send` method.

Specifying Message Persistence

Jakarta Messaging supports two delivery modes specifying whether messages are lost if the Messaging provider fails. These delivery modes are fields of the `DeliveryMode` interface.

- The default delivery mode, `PERSISTENT`, instructs the Messaging provider to take extra care to ensure that a message is not lost in transit in case of a Messaging provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.
- The `NON_PERSISTENT` delivery mode does not require the Messaging provider to store the message or otherwise guarantee that it is not lost if the provider fails.

To specify the delivery mode, use the `setDeliveryMode` method of the `JMSProducer` interface to set the delivery mode for all messages sent by that producer.

You can use method chaining to set the delivery mode when you create a producer and send a message. The following call creates a producer with a `NON_PERSISTENT` delivery mode and uses it to send a message:

```
context.createProducer()
       .setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(dest, msg);
```

If you do not specify a delivery mode, the default is `PERSISTENT`. Using the `NON_PERSISTENT` delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

Setting Message Priority Levels

You can use message priority levels to instruct the Messaging provider to deliver urgent messages first. Use the `setPriority` method of the `JMSProducer` interface to set the priority level for all messages sent by that producer.

You can use method chaining to set the priority level when you create a producer and send a message. For example, the following call sets a priority level of 7 for a producer and then sends a message:

```
context.createProducer().setPriority(7).send(dest, msg);
```

The ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4. A Messaging provider tries to deliver higher-priority messages before lower-priority ones, but does not have to deliver messages in exact order of priority.

Allowing Messages to Expire

By default, a message never expires. If a message will become obsolete after a certain period, however, you may want to set an expiration time. Use the `setTimeToLive` method of the `JMSProducer` interface to set a default expiration time for all messages sent by that producer.

For example, a message that contains rapidly changing data such as a stock price will become obsolete after a few minutes, so you might configure messages to expire after that time.

You can use method chaining to set the time to live when you create a producer and send a message. For example, the following call sets a time to live of five minutes for a producer and then sends a message:

```
context.createProducer().setTimeToLive(300000).send(dest, msg);
```

If the specified `timeToLive` value is `0`, the message never expires.

When the message is sent, the specified `timeToLive` is added to the current time to give the expiration time. Any message not delivered before the specified expiration time is destroyed. The destruction of obsolete messages conserves storage and computing resources.

Specifying a Delivery Delay

You can specify a length of time that must elapse after a message is sent before the Messaging provider delivers the message. Use the `setDeliveryDelay` method of the `JMSProducer` interface to set a delivery delay for all messages sent by that producer.

You can use method chaining to set the delivery delay when you create a producer and send a message. For example, the following call sets a delivery delay of 3 seconds for a producer and then sends a message:

```
context.createProducer().setDeliveryDelay(3000).send(dest, msg);
```

Using JMSProducer Method Chaining

The setter methods on the `JMSProducer` interface return `JMSProducer` objects, so you can use method chaining to create a producer, set multiple properties, and send a message. For example, the following chained method calls create a producer, set a user-defined property, set the expiration, delivery mode, and priority for the message, and then send a message to a queue:

```
context.createProducer()
        .setProperty("MyProperty", "MyValue")
        .setTimeToLive(10000)
        .setDeliveryMode(NON_PERSISTENT)
        .setPriority(2)
        .send(queue, body);
```

You can also call the `JMSProducer` methods to set properties on a message and then send the message in a separate `send` method call. You can also set message properties directly on a message.

Creating Temporary Destinations

Normally, you create JMS destinations (queues and topics) administratively rather than programmatically. Your Messaging provider includes a tool to create and remove destinations, and it is common for destinations to be long-lasting.

Jakarta Messaging also enables you to create destinations (`TemporaryQueue` and `TemporaryTopic` objects) that last only for the duration of the connection in which they are created. You create these destinations dynamically using the `JMSContext.createTemporaryQueue` and the `JMSContext.createTemporaryTopic` methods, as in the following example:

```
TemporaryTopic replyTopic = context.createTemporaryTopic();
```

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination. Any message producer can send to the temporary destination. If you close the connection to which a temporary destination belongs, the destination is closed and its contents are lost.

You can use temporary destinations to implement a simple request/reply mechanism. If you create a temporary destination and specify it as the value of the `JMSReplyTo` message header field when you send a message, then the consumer of the message can use the value of the `JMSReplyTo` field as the destination to which it sends a reply. The consumer can also reference the original request by setting the `JMSCorrelationID` header field of the reply message to the value of the `JMSMessageID` header field of the request. For example, an `onMessage` method can create a `JMSContext` so that it can send a reply to the message it receives. It can use code such as the following:

```
replyMsg = context.createTextMessage("Consumer processed message: "
        + msg.getText());
replyMsg.setJMSCorrelationID(msg.getJMSMessageID());
context.createProducer().send((Topic) msg.getJMSReplyTo(), replyMsg);
```

For an example, see Using an Entity to Join Messages from Two MDBs.

Using Jakarta Messaging Local Transactions

A transaction groups a series of operations into an atomic unit of work. If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning. If all the operations succeed, the transaction can be committed.

In an application client or a Java SE client, you can use local transactions to group message sends and receives. You use the `JMSContext.commit` method to commit a transaction. You can send multiple messages in a transaction, and the messages will not be added to the queue or topic until the transaction is committed. If you receive multiple messages in a transaction, they will not be acknowledged until the transaction is committed.

You can use the `JMSContext.rollback` method to roll back a transaction. A transaction rollback means that all produced messages are destroyed and all consumed messages are recovered and redelivered unless they have expired (see Allowing Messages to Expire).

A transacted session is always involved in a transaction. To create a transacted session, call the `createContext` method as follows:

```
JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```

As soon as the `commit` or the `rollback` method is called, one transaction ends and another transaction begins. Closing a transacted session rolls back its transaction in progress, including any pending sends and receives.

In an application running in the Jakarta EE web or enterprise bean container, you cannot use local transactions. Instead, you use Jakarta Transactions, described in Using Jakarta Messaging in Jakarta EE Applications.

You can combine several sends and receives in a single Jakarta Messaging local transaction, so long as they are all performed using the same `JMSContext`.

Do not use a single transaction if you use a request/reply mechanism, in which you send a message and then receive a reply to that message. If you try to use a single transaction, the program will hang, because the send cannot take place until the transaction is committed. The following code fragment illustrates the problem:

```
// Don't do this!
outMsg.setJMSReplyTo(replyQueue);
context.createProducer().send(outQueue, outMsg);
consumer = context.createConsumer(replyQueue);
inMsg = consumer.receive();
context.commit();
```

Because a message sent during a transaction is not actually sent until the transaction is committed,

the transaction cannot contain any receives that depend on that message's having been sent.

The production and the consumption of a message cannot both be part of the same transaction. The reason is that the transactions take place between the clients and the Messaging provider, which intervenes between the production and the consumption of the message. Figure 31, "Using Jakarta Messaging Local Transactions" illustrates this interaction.

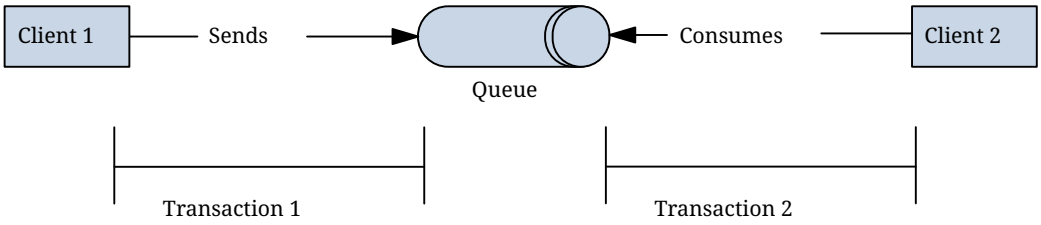


Figure 31. Using Jakarta Messaging Local Transactions

The sending of one or more messages to one or more destinations by Client 1 can form a single transaction, because it forms a single set of interactions with the Messaging provider using a single `JMSContext`. Similarly, the receiving of one or more messages from one or more destinations by Client 2 also forms a single transaction using a single `JMSContext`. But because the two clients have no direct interaction and are using two different `JMSContext` objects, no transactions can take place between them.

Another way of putting this is that a transaction is a contract between a client and a Messaging provider that defines whether a message is sent to a destination or whether a message is received from the destination. It is not a contract between the sending client and the receiving client.

This is the fundamental difference between messaging and synchronized processing. Instead of tightly coupling the sender and the receiver of a message, JMS couples the sender of a message with the destination, and it separately couples the destination with the receiver of the message. Therefore, while the sends and receives each have a tight coupling with the Messaging provider, they do not have any coupling with each other.

When you create a `JMSContext`, you can specify whether it is transacted by using the `JMSContext.SESSION_TRANSACTED` argument to the `createContext` method. For example:

```
try (JMSContext context = connectionFactory.createContext(
        JMSContext.SESSION_TRANSACTED);) {
    ...
}
```

The `commit` and the `rollback` methods for local transactions are associated with the session that underlies the `JMSContext`. You can combine operations on more than one queue or topic, or on a combination of queues and topics, in a single transaction if you use the same session to perform the operations. For example, you can use the same `JMSContext` to receive a message from a queue and send a message to a topic in the same transaction.

The example in Using Local Transactions shows how to use Jakarta Messaging local transactions.

Sending Messages Asynchronously

Normally, when you send a persistent message, the `send` method blocks until the Messaging provider confirms that the message was sent successfully. The asynchronous send mechanism allows your application to send a message and continue work while waiting to learn whether the send completed.

This feature is currently available only in application clients and Java SE clients.

Sending a message asynchronously involves supplying a callback object. You specify a `CompletionListener` with an `onCompletion` method. For example, the following code instantiates a `CompletionListener` named `SendListener`. It then calls the `setAsync` method to specify that sends from this producer should be asynchronous and should use the specified listener:

```
CompletionListener listener = new SendListener();
context.createProducer().setAsync(listener).send(dest, message);
```

The `CompletionListener` class must implement two methods, `onCompletion` and `onException`. The `onCompletion` method is called if the send succeeds, and the `onException` method is called if it fails. A simple implementation of these methods might look like this:

```
@Override
public void onCompletion(Message message) {
    System.out.println("onCompletion method: Send has completed.");
}

@Override
public void onException(Message message, Exception e) {
    System.out.println("onException method: send failed: " + e.toString());
    System.out.println("Unsent message is: \n" + message);
}
```

Using Jakarta Messaging in Jakarta EE Applications

This section describes how using Jakarta Messaging in enterprise bean applications or web applications differs from using it in application clients.

Overview of Using Jakarta Messaging

A general rule in the Jakarta EE platform specification applies to all Jakarta EE components that use Jakarta Messaging within enterprise bean or web containers: Application components in the web and enterprise bean containers must not attempt to create more than one active (not closed) `Session` object per connection. Multiple `JMSContext` objects are permitted, however, since they combine a single connection and a single session.

This rule does not apply to application clients. The application client container supports the creation of multiple sessions for each connection.

Creating Resources for Jakarta EE Applications

You can use annotations to create application-specific connection factories and destinations for Jakarta EE enterprise bean or web components. The resources you create in this way are visible only to the application for which you create them.

You can also use deployment descriptor elements to create these resources. Elements specified in the deployment descriptor override elements specified in annotations. See Packaging Applications for basic information about deployment descriptors. You must use a deployment descriptor to create application-specific resources for application clients.

To create a destination, use a `@JMSDestinationDefinition` annotation like the following on a class:

```
@JMSDestinationDefinition(
    name = "java:app/jms/myappTopic",
    interfaceName = "jakarta.jms.Topic",
    destinationName = "MyPhysicalAppTopic"
  )
```

The `name`, `interfaceName`, and `destinationName` elements are required. You can optionally specify a `description` element. To create multiple destinations, enclose them in a `@JMSDestinationDefinitions` annotation, separated by commas.

To create a connection factory, use a `@JMSConnectionFactoryDefinition` annotation like the following on a class:

```
@JMSConnectionFactoryDefinition(
    name="java:app/jms/MyConnectionFactory"
)
```

The `name` element is required. You can optionally specify a number of other elements, such as `clientId` if you want to use the connection factory for durable subscriptions, or `description`. If you do not specify the `interfaceName` element, the default interface is `jakarta.jms.ConnectionFactory`. To create multiple connection factories, enclose them in a `@JMSConnectionFactoryDefinitions` annotation, separated by commas.

You need to specify the annotation only once for a given application, in any of the components.



If your application contains one or more message-driven beans, you may want to place the annotation on one of the message-driven beans. If you place the annotation on a sending component such as an application client, you need to specify the `mappedName` element to look up the topic, instead of using the `destinationLookup` property of the activation configuration specification.

When you inject the resource into a component, use the value of the `name` element in the definition annotation as the value of the `lookup` element in the `@Resource` annotation:

```
@Resource(lookup = "java:app/jms/myappTopic")
private Topic topic;
```

The following portable JNDI namespaces are available. Which ones you can use depends on how your application is packaged.

- `java:global`: Makes the resource available to all deployed applications
- `java:app`: Makes the resource available to all components in all modules in a single application
- `java:module`: Makes the resource available to all components within a given module (for example, all enterprise beans within a Jakarta Enterprise Beans module)
- `java:comp`: Makes the resource available to a single component only (except in a web application, where it is equivalent to `java:module`)

See the API documentation for details on these annotations. The examples in Sending and Receiving Messages Using a Simple Web Application, Sending Messages from a Session Bean to an MDB, and Using an Entity to Join Messages from Two MDBs all use the `@JMSDestinationDefinition` annotation. The other JMS examples do not use these annotations. The examples that consist only of application clients are not deployed in the application server and must therefore communicate with each other using administratively created resources that exist outside of individual applications.

Using Resource Injection in Enterprise Bean or Web Components

You may use resource injection to inject both administered objects and `JMSContext` objects in Jakarta EE applications.

Injecting a ConnectionFactory, Queue, or Topic

Normally, you use the `@Resource` annotation to inject a `ConnectionFactory`, `Queue`, or `Topic` into your Jakarta EE application. These objects must be created administratively before you deploy your application. You may want to use the default connection factory, whose JNDI name is `java:comp/DefaultJMSConnectionFactory`.

When you use resource injection in an application client component, you normally declare the Messaging resource static:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyQueue")
private static Queue queue;
```

However, when you use this annotation in a session bean, a message-driven bean, or a web component, do not declare the resource static:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private ConnectionFactory connectionFactory;
```

```
@Resource(lookup = "jms/MyTopic")
private Topic topic;
```

If you declare the resource static in these components, runtime errors will result.

Injecting a JMSContext Object

To access a `JMSContext` object in an enterprise bean or web component, instead of injecting the `ConnectionFactory` resource and then creating a `JMSContext`, you can use the `@Inject` and `@JMSConnectionFactory` annotations to inject a `JMSContext`. To use the default connection factory, use code like the following:

```
@Inject
private JMSContext context1;
```

To use your own connection factory, use code like the following:

```
@Inject
@JMSConnectionFactory("jms/MyConnectionFactory")
private JMSContext context2;
```

Using Jakarta EE Components to Produce and to Synchronously Receive Messages

An application that produces messages or synchronously receives them can use a Jakarta EE web or Jakarta Enterprise Beans component, such as a managed bean, a servlet, or a session bean, to perform these operations. The example in Sending Messages from a Session Bean to an MDB uses a stateless session bean to send messages to a topic. The example in Sending and Receiving Messages Using a Simple Web Application uses managed beans to produce and to consume messages.

Because a synchronous receive with no specified timeout ties up server resources, this mechanism usually is not the best application design for a web or Jakarta Enterprise Beans component. Instead, use a synchronous receive that specifies a timeout value, or use a message-driven bean to receive messages asynchronously. For details about synchronous receives, see Jakarta Messaging Message Consumers.

Using Jakarta Messaging in a Jakarta EE component is in many ways similar to using it in an application client. The main differences are the areas of resource management and transactions.

Managing Jakarta Messaging Resources in Web and Jakarta Enterprise Beans Components

The Jakarta Messaging resources are a connection and a session, usually combined in a `JMSContext` object. In general, it is important to release Messaging resources when they are no longer being used. Here are some useful practices to follow.

- If you wish to maintain a Messaging resource only for the life span of a business method, use a `try`-with-resources statement to create the `JMSContext` so that it will be closed automatically at the end of the `try` block.

- To maintain a Messaging resource for the duration of a transaction or request, inject the `JMSContext` as described in Injecting a JMSContext Object. This will also cause the resource to be released when it is no longer needed.
- If you would like to maintain a Messaging resource for the life span of an enterprise bean instance, you can use a `@PostConstruct` callback method to create the resource and a `@PreDestroy` callback method to close the resource. However, there is normally no need to do this, since application servers usually maintain a pool of connections. If you use a stateful session bean and you wish to maintain the Messaging resource in a cached state, you must close the resource in a `@PrePassivate` callback method and set its value to `null`, and you must create it again in a `@PostActivate` callback method.

Managing Transactions in Session Beans

Instead of using local transactions, you use Jakarta transactions. You can use either container-managed transactions or bean-managed transactions. Normally, you use container-managed transactions for bean methods that perform sends or receives, allowing the enterprise bean container to handle transaction demarcation. Because container-managed transactions are the default, you do not have to specify them.

You can use bean-managed transactions and the `jakarta.transaction.UserTransaction` interface's transaction demarcation methods, but you should do so only if your application has special requirements and you are an expert in using transactions. Usually, container-managed transactions produce the most efficient and correct behavior. This tutorial does not provide any examples of bean-managed transactions.

Using Message-Driven Beans to Receive Messages Asynchronously

The sections What Is a Message-Driven Bean? and How Does Jakarta Messaging Work with the Jakarta EE Platform? describe how the Jakarta EE platform supports a special kind of enterprise bean, the message-driven bean, which allows Jakarta EE applications to process Jakarta Messaging messages asynchronously. Other Jakarta EE web and Jakarta Enterprise Beans components allow you to send messages and to receive them synchronously but not asynchronously.

A message-driven bean is a message listener to which messages can be delivered from either a queue or a topic. The messages can be sent by any Jakarta EE component (from an application client, another enterprise bean, or a web component) or from an application or a system that does not use Jakarta EE technology.

A message-driven bean class has the following requirements.

- It must be annotated with the `@MessageDriven` annotation if it does not use a deployment descriptor.
- The class must be defined as `public`, but not as `abstract` or `final`.
- It must contain a public constructor with no arguments.

It is recommended, but not required, that a message-driven bean class implement the message listener interface for the message type it supports. A bean that supports Jakarta Messaging implements the `jakarta.jms.MessageListener` interface, which means that it must provide an `onMessage` method with the following signature:

```
void onMessage(Message inMessage)
```

The `onMessage` method is called by the bean's container when a message has arrived for the bean to service. This method contains the business logic that handles the processing of the message. It is the message-driven bean's responsibility to parse the message and perform the necessary business logic.

A message-driven bean differs from an application client's message listener in the following ways.

- In an application client, you must create a `JMSContext`, then create a `JMSConsumer`, then call `setMessageListener` to activate the listener. For a message-driven bean, you need only define the class and annotate it, and the enterprise bean container creates it for you.
- The bean class uses the `@MessageDriven` annotation, which typically contains an `activationConfig` element containing `@ActivationConfigProperty` annotations that specify properties used by the bean or the connection factory. These properties can include the connection factory, a destination type, a durable subscription, a message selector, or an acknowledgment mode. Some of the examples in [messaging:jms-examples::jms-examples:::_jakarta_messaging_examples] set these properties. You can also set the properties in the deployment descriptor.
- The application client container has only one instance of a `MessageListener`, which is called on a single thread at a time. A message-driven bean, however, may have multiple instances, configured by the container, which may be called concurrently by multiple threads (although each instance is called by only one thread at a time). Message-driven beans may therefore allow much faster processing of messages than message listeners.
- You do not need to specify a message acknowledgment mode unless you use bean-managed transactions. The message is consumed in the transaction in which the `onMessage` method is invoked.

@ActivationConfigProperty Settings for Message-Driven Beans lists the activation configuration properties defined by the Jakarta Messaging specification.

@ActivationConfigProperty Settings for Message-Driven Beans

Property Name	Description
`acknowledgeMode`	Acknowledgment mode, used only for bean-managed transactions; the default is `Auto-acknowledge` (`Dups-ok-acknowledge` is also permitted)
`destinationLookup`	The lookup name of the queue or topic from which the bean will receive messages
`destinationType`	Either `jakarta.jms.Queue` or `jakarta.jms.Topic`
`subscriptionDurability`	For durable subscriptions, set the value to `Durable`; see Creating Durable Subscriptions for more information
`clientId`	For durable subscriptions, the client ID for the connection (optional)
`subscriptionName`	For durable subscriptions, the name of the subscription

Property Name	Description
`messageSelector`	A string that filters messages; see Jakarta Messaging Message Selectors for information
`connectionFactoryLookup`	The lookup name of the connection factory to be used to connect to the Messaging provider from which the bean will receive messages

For example, here is the message-driven bean used in Receiving Messages Asynchronously Using a Message-Driven Bean:

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "jms/MyQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {

    @Resource
    private MessageDrivenContext mdc;
    static final Logger logger = Logger.getLogger("SimpleMessageBean");

    public SimpleMessageBean() {
    }

    @Override
    public void onMessage(Message inMessage) {

        try {
            if (inMessage instanceof TextMessage) {
                logger.log(Level.INFO,
                        "MESSAGE BEAN: Message received: {0}",
                        inMessage.getBody(String.class));
            } else {
                logger.log(Level.WARNING,
                        "Message of wrong type: {0}",
                        inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            logger.log(Level.SEVERE,
                    "SimpleMessageBean.onMessage: JMSException: {0}",
                    e.toString());
            mdc.setRollbackOnly();
        }
    }
}
```

If Jakarta Messaging is integrated with the application server using a resource adapter, the

Messaging resource adapter handles these tasks for the enterprise bean container.

The bean class commonly injects a `MessageDrivenContext` resource, which provides some additional methods you can use for transaction management (`setRollbackOnly`, for example):

```
@Resource
private MessageDrivenContext mdc;
```

A message-driven bean never has a local or remote interface. Instead, it has only a bean class.

A message-driven bean is similar in some ways to a stateless session bean: Its instances are relatively short-lived and retain no state for a specific client. The instance variables of the message-driven bean instance can contain some state across the handling of client messages: for example, an open database connection, or an object reference to an enterprise bean object.

Like a stateless session bean, a message-driven bean can have many interchangeable instances running at the same time. The container can pool these instances to allow streams of messages to be processed concurrently. The container attempts to deliver messages in chronological order when that would not impair the concurrency of message processing, but no guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class. If message order is essential to your application, you may want to configure your application server to use just one instance of the message-driven bean.

For details on the lifecycle of a message-driven bean, see The Lifecycle of a Message-Driven Bean.

Managing Jakarta Transactions

Jakarta EE application clients and Java SE clients use JMS local transactions (described in Using Jakarta Messaging Local Transactions), which allow the grouping of sends and receives within a specific Messaging session. Jakarta EE applications that run in the web or enterprise bean container commonly use Jakarta Transactions to ensure the integrity of accesses to external resources. The key difference between a Jakarta transaction and a Jakarta Messaging local transaction is that a Jakarta transaction is controlled by the application server's transaction managers. Jakarta transactions may be distributed, which means that they can encompass multiple resources in the same transaction, such as a Messaging provider and a database.

For example, distributed transactions allow multiple applications to perform atomic updates on the same database, and they allow a single application to perform atomic updates on multiple databases.

In a Jakarta EE application that uses Jakarta Messaging, you can use transactions to combine message sends or receives with database updates and other resource manager operations. You can access resources from multiple application components within a single transaction. For example, a servlet can start a transaction, access multiple databases, invoke an enterprise bean that sends a Jakarta Messaging message, invoke another enterprise bean that modifies an EIS system using the Connectors, and finally commit the transaction. Your application cannot, however, both send a Jakarta Messaging message and receive a reply to it within the same transaction.

Jakarta Transactions within the enterprise bean and web containers can be either of two kinds.

- Container-managed transactions: The container controls the integrity of your transactions without your having to call `commit` or `rollback`. Container-managed transactions are easier to use than bean-managed transactions. You can specify appropriate transaction attributes for your enterprise bean methods.

  Use the `Required` transaction attribute (the default) to ensure that a method is always part of a transaction. If a transaction is in progress when the method is called, the method will be part of that transaction; if not, a new transaction will be started before the method is called and will be committed when the method returns. See Transaction Attributes for more information.

- Bean-managed transactions: You can use these in conjunction with the `jakarta.transaction.UserTransaction` interface, which provides its own `commit` and `rollback` methods you can use to delimit transaction boundaries. Bean-managed transactions are recommended only for those who are experienced in programming transactions.

You can use either container-managed transactions or bean-managed transactions with message-driven beans. To ensure that all messages are received and handled within the context of a transaction, use container-managed transactions and use the `Required` transaction attribute (the default) for the `onMessage` method.

When you use container-managed transactions, you can call the following `MessageDrivenContext` methods.

- `setRollbackOnly`: Use this method for error handling. If an exception occurs, `setRollbackOnly` marks the current transaction so that the only possible outcome of the transaction is a rollback.
- `getRollbackOnly`: Use this method to test whether the current transaction has been marked for rollback.

If you use bean-managed transactions, the delivery of a message to the `onMessage` method takes place outside the Jakarta transaction context. The transaction begins when you call the `UserTransaction.begin` method within the `onMessage` method, and it ends when you call `UserTransaction.commit` or `UserTransaction.rollback`. Any call to the `Connection.createSession` method must take place within the transaction.

Using bean-managed transactions allows you to process the message by using more than one transaction or to have some parts of the message processing take place outside a transaction context. However, if you use container-managed transactions, the message is received by the MDB and processed by the `onMessage` method within the same transaction. It is not possible to achieve this behavior with bean-managed transactions.

When you create a `JMSContext` in a Jakarta transaction (in the web or enterprise bean container), the container ignores any arguments you specify, because it manages all transactional properties. When you create a `JMSContext` in the web or enterprise bean container and there is no Jakarta transaction, the value (if any) passed to the `createContext` method should be `JMSContext.AUTO_ACKNOWLEDGE` or `JMSContext.DUPS_OK_ACKNOWLEDGE`.

When you use container-managed transactions, you normally use the `Required` transaction attribute (the default) for your enterprise bean's business methods.

You do not specify the activation configuration property `acknowledgeMode` when you create a

message-driven bean that uses container-managed transactions. The container acknowledges the message automatically when it commits the transaction.

If a message-driven bean uses bean-managed transactions, the message receipt cannot be part of the bean-managed transaction. You can set the activation configuration property `acknowledgeMode` to `Auto-acknowledge` or `Dups-ok-acknowledge` to specify how you want the message received by the message-driven bean to be acknowledged.

If the `onMessage` method throws a `RuntimeException`, the container does not acknowledge processing the message. In that case, the Messaging provider will redeliver the unacknowledged message in the future.

Further Information about Jakarta Messaging

For more information about Jakarta Messaging, see

- Jakarta Messaging website:
  https://projects.eclipse.org/projects/ee4j.jms
- Jakarta Messaging specification, version 3.0, available from:
  https://jakarta.ee/specifications/messaging/3.1/

Jakarta Messaging Examples



This section was last updated for Jakarta EE Version 9.1

This chapter provides examples that show how to use Jakarta Messaging in various kinds of Jakarta EE applications.

Building and Running Jakarta Messaging Examples

The examples are in the `jakartaee-examples/tutorial/jms/` directory.

To build and run each example:

1. Use NetBeans IDE or Maven to compile, package, and in some cases deploy the example.
2. Use NetBeans IDE, Maven, or the `appclient` command to run the application client, or use the browser to run the web application examples.

Before you deploy or run the examples, you need to create resources for them. Some examples have a `glassfish-resources.xml` file that is used to create resources for that example and others. You can use the `asadmin` command to create the resources.

To use the `asadmin` and `appclient` commands, you need to put the GlassFish Server `bin` directories in your command path, as described in GlassFish Server Installation Tips.

Overview of the Jakarta Messaging Examples

The following tables list the examples used in this chapter, describe what they do, and link to the section that describes them fully. The example directory for each example is relative to the

`jakartaee-examples/tutorial/jms/` directory.

Jakarta Messaging Examples That Show the Use of Jakarta EE Application Clients

Example Directory	Description
`simple/producer`	Using an application client to send messages; see Sending Messages
`simple/synchconsumer`	Using an application client to receive messages synchronously; see Receiving Messages Synchronously
`simple/asynchconsumer`	Using an application client to receive messages asynchronously; see Using a Message Listener for Asynchronous Message Delivery
`simple/messagebrowser`	Using an application client to use a `QueueBrowser` to browse a queue; see Browsing Messages on a Queue
`simple/clientackconsumer`	Using an application client to acknowledge messages received synchronously; see Acknowledging Messages
`durablesubscriptionexample`	Using an application client to create a durable subscription on a topic; see Using Durable Subscriptions
`transactedexample`	Using an application client to send and receive messages in local transactions (also uses request-reply messaging); see Using Local Transactions
`shared/sharedconsumer`	Using an application client to create shared nondurable topic subscriptions; see Using Shared Nondurable Subscriptions
`shared/shareddurableconsumer`	Using an application client to create shared durable topic subscriptions; see Using Shared Durable Subscriptions

Jakarta Messaging Examples That Show the Use of Jakarta EE Web and Enterprise Bean Components

Example Directory	Description
`websimplemessage`	Using managed beans to send messages and to receive messages synchronously; see Sending and Receiving Messages Using a Simple Web Application
`simplemessage`	Using an application client to send messages, and using a message-driven bean to receive messages asynchronously; see Receiving Messages Asynchronously Using a Message-Driven Bean
`clientsessionmdb`	Using a session bean to send messages, and using a message-driven bean to receive messages; see Sending Messages from a Session Bean to an MDB

Example Directory	Description
`clientmdbentity`	Using an application client, two message-driven beans, and JPA persistence to create a simple HR application; see Using an Entity to Join Messages from Two MDBs

Writing Simple Jakarta Messaging Applications

This section shows how to create, package, and run simple Messaging clients that are packaged as application clients.

Overview of Writing Simple Jakarta Messaging Application

The clients demonstrate the basic tasks a Jakarta Messaging application must perform:

- Creating a `JMSContext`
- Creating message producers and consumers
- Sending and receiving messages

Each example uses two clients: one that sends messages and one that receives them. You can run the clients in two terminal windows.

When you write a Messaging client to run in an enterprise bean application, you use many of the same methods in much the same sequence as for an application client. However, there are some significant differences. Using Jakarta Messaging in Jakarta EE Applications describes these differences, and this chapter provides examples that illustrate them.

The examples for this section are in the `jakartaee-examples/tutorial/jms/simple/` directory, under the following subdirectories:

```
producer/
synchconsumer/
asynchconsumer/
messagebrowser/
clientackconsumer/
```

Before running the examples, you need to start GlassFish Server and create administered objects.

Starting the Jakarta Messaging Provider

When you use GlassFish Server, your Messaging provider is GlassFish Server. Start the server as described in Starting and Stopping GlassFish Server.

Creating Jakarta Messaging Administered Objects

This example uses the following Jakarta Messaging administered objects:

- A connection factory
- Two destination resources: a topic and a queue

Before you run the applications, you can use the `asadmin add-resources` command to create needed Messaging resources, specifying as the argument a file named `glassfish-resources.xml`. This file can be created in any project using NetBeans IDE, although you can also create it by hand. A file for the needed resources is present in the `jms/simple/producer/src/main/setup/` directory.

The Jakarta Messaging examples use a connection factory with the logical JNDI lookup name `java:comp/DefaultJMSConnectionFactory`, which is preconfigured in GlassFish Server.

You can also use the `asadmin create-jms-resource` command to create resources, the `asadmin list-jms-resources` command to display their names, and the `asadmin delete-jms-resource` command to remove them.

To Create Resources for the Simple Examples

A `glassfish-resources.xml` file in one of the Maven projects can create all the resources needed for the simple examples.

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a command window, go to the `Producer` example.

   ```
   cd jakartaee-examples/tutorial/jms/simple/producer
   ```

3. Create the resources using the `asadmin add-resources` command:

   ```
   asadmin add-resources src/main/setup/glassfish-resources.xml
   ```

4. Verify the creation of the resources:

   ```
   asadmin list-jms-resources
   ```

   The command lists the two destinations and connection factory specified in the `glassfish-resources.xml` file in addition to the platform default connection factory:

   ```
   jms/MyQueue
   jms/MyTopic
   jms/__defaultConnectionFactory
   Command list-jms-resources executed successfully.
   ```

   In GlassFish Server, the Jakarta EE `java:comp/DefaultJMSConnectionFactory` resource is mapped to a connection factory named `jms/__defaultConnectionFactory`.

Building All the Simple Examples

To run the simple examples using GlassFish Server, package each example in an application client JAR file. The application client JAR file requires a manifest file, located in the `src/main/java/META-INF/` directory for each example, along with the `.class` file.

The `pom.xml` file for each example specifies a plugin that creates an application client JAR file. You can build the examples using either NetBeans IDE or Maven.

To Build All the Simple Examples Using NetBeans IDE

1. From the **File** menu, choose **Open Project**.
2. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/jms
   ```

3. Expand the `jms` node and select the `simple` folder.
4. Click **Open Project** to open all the simple examples.
5. In the **Projects** tab, right-click the `simple` project and select **Build** to build all the examples.

   This command places the application client JAR files in the `target` directories for the examples.

To Build All the Simple Examples Using Maven

1. In a terminal window, go to the `simple` directory:

   ```
   cd jakartaee-examples/tutorial/jms/simple/
   ```

2. Enter the following command to build all the projects:

   ```
   mvn install
   ```

   This command places the application client JAR files in the `target` directories for the examples.

Sending Messages

This section describes how to use a client to send messages. The `Producer.java` client will send messages in all of these examples.

General Steps Performed in the Example

General steps this example performs are as follows.

1. Inject resources for the administered objects used by the example.
2. Accept and verify command-line arguments. You can use this example to send any number of messages to either a queue or a topic, so you specify the destination type and the number of messages on the command line when you run the program.
3. Create a `JMSContext`, then send the specified number of text messages in the form of strings, as described in Message Bodies.
4. Send a final message of type `Message` to indicate that the consumer should expect no more messages.

5. Catch any exceptions.

The Producer.java Client

The sending client, `Producer.java`, performs the following steps.

1. Injects resources for a connection factory, queue, and topic:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

2. Retrieves and verifies command-line arguments that specify the destination type and the number of arguments:

```
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if ( ! ( destType.equals("queue") || destType.equals("topic") ) ) {
    System.err.println("Argument must be \"queue\" or " + "\"topic\"");
    System.exit(1);
}
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
} else {
    NUM_MSGS = 1;
}
```

3. Assigns either the queue or the topic to a destination object, based on the specified destination type:

```
Destination dest = null;
try {
    if (destType.equals("queue")) {
        dest = (Destination) queue;
    } else {
        dest = (Destination) topic;
    }
} catch (Exception e) {
    System.err.println("Error setting destination: " + e.toString());
    System.exit(1);
}
```

4. Within a `try-with-resources` block, creates a `JMSContext`:

```
try (JMSContext context = connectionFactory.createContext();) { ... }
```

5. Sets the message count to zero, then creates a `JMSProducer` and sends one or more messages to the destination and increments the count. Messages in the form of strings are of the `TextMessage` message type:

```
    int count = 0;
    for (int i = 0; i < NUM_MSGS; i++) {
        String message = "This is message " + (i + 1)
                + " from producer";
        // Comment out the following line to send many messages
        System.out.println("Sending message: " + message);
        context.createProducer().send(dest, message);
        count += 1;
    }
    System.out.println("Text messages sent: " + count);
```

6. Sends an empty control message to indicate the end of the message stream:

```
    context.createProducer().send(dest, context.createMessage());
```

   Sending an empty message of no specified type is a convenient way for an application to indicate to the consumer that the final message has arrived.

7. Catches and handles any exceptions. The end of the `try-with-resources` block automatically causes the `JMSContext` to be closed:

```
} catch (Exception e) {
    System.err.println("Exception occurred: " + e.toString());
    System.exit(1);
}
System.exit(0);
```

To Run the Producer Client

You can run the client using the `appclient` command. The `Producer` client takes one or two command-line arguments: a destination type and, optionally, a number of messages. If you do not specify a number of messages, the client sends one message.

You will use the client to send three messages to a queue.

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server) and that you have created resources and built the simple Jakarta Messaging examples (see Creating Jakarta Messaging Administered Objects and Building All the Simple Examples).
2. In a terminal window, go to the `producer` directory:

```
cd producer
```

3. Run the `Producer` program, sending three messages to the queue:

```
appclient -client target/producer.jar queue 3
```

The output of the program looks like this (along with some additional output):

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

The messages are now in the queue, waiting to be received.



When you run an application client, the command may take a long time to complete.

Receiving Messages Synchronously

This section describes the receiving client, which uses the `receive` method to consume messages synchronously. This section then explains how to run the clients using GlassFish Server.

The SynchConsumer.java Client

The receiving client, `SynchConsumer.java`, performs the following steps.

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or the topic to a destination object, based on the specified destination type.
3. Within a `try-with-resources` block, creates a `JMSContext`.
4. Creates a `JMSConsumer`, starting message delivery:

```
consumer = context.createConsumer(dest);
```

5. Receives the messages sent to the destination until the end-of-message-stream control message is received:

```
int count = 0;
while (true) {
    Message m = consumer.receive(1000);
    if (m != null) {
        if (m instanceof TextMessage) {
```

```
            System.out.println(
                    "Reading message: " + m.getBody(String.class));
            count += 1;
        } else {
            break;
        }
    }
}
System.out.println("Messages received: " + count);
```

Because the control message is not a `TextMessage`, the receiving client terminates the `while` loop and stops receiving messages after the control message arrives.

6. Catches and handles any exceptions. The end of the `try-with-resources` block automatically causes the `JMSContext` to be closed.

The `SynchConsumer` client uses an indefinite `while` loop to receive messages, calling `receive` with a timeout argument.

To Run the SynchConsumer and Producer Clients

You can run the client using the `appclient` command. The `SynchConsumer` client takes one command-line argument, the destination type.

These steps show how to receive and send messages synchronously using both a queue and a topic. The steps assume you already ran the `Producer` client and have three messages waiting in the queue.

1. In the same terminal window where you ran `Producer`, go to the `synchconsumer` directory:

```
cd ../synchconsumer
```

2. Run the `SynchConsumer` client, specifying the queue:

```
appclient -client target/synchconsumer.jar queue
```

The output of the client looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Messages received: 3
```

3. Now try running the clients in the opposite order. Run the `SynchConsumer` client:

```
appclient -client target/synchconsumer.jar queue
```

The client displays the destination type and then waits for messages.

4. Open a new terminal window and run the `Producer` client:

```
cd jakartaee-examples/tutorial/jms/simple/producer
appclient -client target/producer.jar queue 3
```

When the messages have been sent, the `SynchConsumer` client receives them and exits.

5. Now run the `Producer` client using a topic instead of a queue:

```
appclient -client target/producer.jar topic 3
```

The output of the client looks like this (along with some additional output):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

6. Now, in the other terminal window, run the `SynchConsumer` client using the topic:

```
appclient -client target/synchconsumer.jar topic
```

The result, however, is different. Because you are using a subscription on a topic, messages that were sent before you created the subscription on the topic will not be added to the subscription and delivered to the consumer. (See Publish/Subscribe Messaging Style and Consuming Messages from Topics for details.) Instead of receiving the messages, the client waits for messages to arrive.

7. Leave the `SynchConsumer` client running and run the `Producer` client again:

```
appclient -client target/producer.jar topic 3
```

Now the `SynchConsumer` client receives the messages:

```
Destination type is topic
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

```
Messages received: 3
```

Because these messages were sent after the consumer was started, the client receives them.

Using a Message Listener for Asynchronous Message Delivery

This section describes the receiving clients in an example that uses a message listener for asynchronous message delivery. This section then explains how to compile and run the clients using GlassFish Server.



In the Jakarta EE platform, message listeners can be used only in application clients, as in this example. To allow asynchronous message delivery in a web or enterprise bean application, you use a message-driven bean, shown in later examples in this chapter.

Writing the AsynchConsumer.java and TextListener.java Clients

The sending client is `Producer.java`, the same client used in Receiving Messages Synchronously.

An asynchronous consumer normally runs indefinitely. This one runs until the user types the character `q` or `Q` to stop the client.

1. The client, `AsynchConsumer.java`, performs the following steps.
    a. Injects resources for a connection factory, queue, and topic.
    b. Assigns either the queue or the topic to a destination object, based on the specified destination type.
    c. In a `try-with-resources` block, creates a `JMSContext`.
    d. Creates a `JMSConsumer`.
    e. Creates an instance of the `TextListener` class and registers it as the message listener for the `JMSConsumer`:

    ```
    listener = new TextListener();
    consumer.setMessageListener(listener);
    ```

    f. Listens for the messages sent to the destination, stopping when the user types the character `q` or `Q` (it uses a `java.io.InputStreamReader` to do this).
    g. Catches and handles any exceptions. The end of the `try-with-resources` block automatically causes the `JMSContext` to be closed, thus stopping delivery of messages to the message listener.
2. The message listener, `TextListener.java`, follows these steps:
    a. When a message arrives, the `onMessage` method is called automatically.
    b. If the message is a `TextMessage`, the `onMessage` method displays its content as a string value. If the message is not a text message, it reports this fact:

```
public void onMessage(Message m) {
    try {
        if (m instanceof TextMessage) {
            System.out.println(
                    "Reading message: " + m.getBody(String.class));
        } else {
             System.out.println("Message is not a TextMessage");
        }
    } catch (JMSException | JMSRuntimeException e) {
        System.err.println("JMSException in onMessage(): " + e.toString());
    }
}
```

For this example, you will use the same connection factory and destinations you created in To Create Resources for the Simple Examples.

The steps assume that you have already built and packaged all the examples using NetBeans IDE or Maven.

To Run the AsynchConsumer and Producer Clients

You will need two terminal windows, as you did in Receiving Messages Synchronously.

1. In the terminal window where you ran the `SynchConsumer` client, go to the `asynchconsumer` example directory:

   ```
   cd jakartaee-examples/tutorial/jms/simple/asynchconsumer
   ```

2. Run the `AsynchConsumer` client, specifying the `topic` destination type:

   ```
   appclient -client target/asynchconsumer.jar topic
   ```

   The client displays the following lines (along with some additional output) and then waits for messages:

   ```
   Destination type is topic
   To end program, enter Q or q, then <return>
   ```

3. In the terminal window where you ran the `Producer` client previously, run the client again, sending three messages:

   ```
   appclient -client target/producer.jar topic 3
   ```

   The output of the client looks like this (along with some additional output):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

In the other window, the `AsynchConsumer` client displays the following (along with some additional output):

```
Destination type is topic
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

The last line appears because the client has received the non-text control message sent by the `Producer` client.

4. Enter `Q` or `q` and press Return to stop the `AsynchConsumer` client.
5. Now run the clients using a queue.

   In this case, as with the synchronous example, you can run the `Producer` client first, because there is no timing dependency between the sender and receiver:

   ```
   appclient -client target/producer.jar queue 3
   ```

   The output of the client looks like this:

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   Text messages sent: 3
   ```

6. In the other window, run the `AsynchConsumer` client:

   ```
   appclient -client target/asynchconsumer.jar queue
   ```

   The output of the client looks like this (along with some additional output):

   ```
   Destination type is queue
   To end program, enter Q or q, then <return>
   Reading message: This is message 1 from producer
   ```

```
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

7. Enter `Q` or `q` and press Return to stop the client.

Browsing Messages on a Queue

This section describes an example that creates a `QueueBrowser` object to examine messages on a queue, as described in Jakarta Messaging Queue Browsers. This section then explains how to compile, package, and run the example using GlassFish Server.

The MessageBrowser.java Client

To create a `QueueBrowser` for a queue, you call the `JMSContext.createBrowser` method with the queue as the argument. You obtain the messages in the queue as an `Enumeration` object. You can then iterate through the `Enumeration` object and display the contents of each message.

The `MessageBrowser.java` client performs the following steps.

1. Injects resources for a connection factory and a queue.
2. In a `try-with-resources` block, creates a `JMSContext`.
3. Creates a `QueueBrowser`:

```
QueueBrowser browser = context.createBrowser(queue);
```

4. Retrieves the `Enumeration` that contains the messages:

```
Enumeration msgs = browser.getEnumeration();
```

5. Verifies that the `Enumeration` contains messages, then displays the contents of the messages:

```
if ( !msgs.hasMoreElements() ) {
    System.out.println("No messages in queue");
} else {
    while (msgs.hasMoreElements()) {
        Message tempMsg = (Message)msgs.nextElement();
        System.out.println("Message: " + tempMsg);
    }
}
```

6. Catches and handles any exceptions. The end of the `try-with-resources` block automatically causes the `JMSContext` to be closed.

Dumping the message contents to standard output retrieves the message body and properties in a format that depends on the implementation of the `toString` method. In GlassFish Server, the

message format looks something like this:

```
Text:   This is message 3 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():      ID:8-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1129061034355
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}
```

Instead of displaying the message contents this way, you can call some of the `Message` interface's getter methods to retrieve the parts of the message you want to see.

For this example, you will use the connection factory and queue you created for Receiving Messages Synchronously. It is assumed that you have already built and packaged all the examples.

To Run the QueueBrowser Client

To run the `MessageBrowser` example using the `appclient` command, follow these steps.

You also need the `Producer` example to send the message to the queue, and one of the consumer clients to consume the messages after you inspect them.

To run the clients, you need two terminal windows.

1. In a terminal window, go to the `producer` directory:

   ```
   cd jakartaee-examples/tutorial/jms/simple/producer/
   ```

2. Run the `Producer` client, sending one message to the queue, along with the non-text control message:

   ```
   appclient -client target/producer.jar queue
   ```

   The output of the client looks like this (along with some additional output):

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   Text messages sent: 1
   ```

3. In another terminal window, go to the `messagebrowser` directory:

```
cd jakartaee-examples/tutorial/jms/simple/messagebrowser
```

4. Run the `MessageBrowser` client using the following command:

```
appclient -client target/messagebrowser.jar
```

The output of the client looks something like this (along with some additional output):

```
Message:
Text:   This is message 1 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():      ID:9-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1363100335526
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}

Message:
Class:                  com.sun.messaging.jmq.jmsclient.MessageImpl
getJMSMessageID():      ID:10-10.152.23.26(bf:27:4:e:e7:ec)-55645-1363100335526
getJMSTimestamp():      1363100335526
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             {JMSXDeliveryCount=0}
```

The first message is the `TextMessage`, and the second is the non-text control message.

5. Go to the `synchconsumer` directory.
6. Run the `SynchConsumer` client to consume the messages:

```
appclient -client target/synchconsumer.jar queue
```

The output of the client looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
Messages received: 1
```

Running Multiple Consumers on the Same Destination

To illustrate further the way point-to-point and publish/subscribe messaging works, you can use the `Producer` and `SynchConsumer` examples to send messages that are then consumed by two clients running simultaneously.

1. Open three command windows. In one, go to the `producer` directory. In the other two, go to the `synchconsumer` directory.
2. In each of the `synchconsumer` windows, start running the client, receiving messages from a queue:

   ```
   appclient -client target/synchconsumer.jar queue
   ```

   Wait until you see the "Destination type is queue" message in both windows.
3. In the `producer` window, run the client, sending 20 or so messages to the queue:

   ```
   appclient -client target/producer.jar queue 20
   ```
4. Look at the output in the `synchconsumer` windows. In point-to-point messaging, each message can have only one consumer. Therefore, each of the clients receives some of the messages. One of the clients receives the non-text control message, reports the number of messages received, and exits.
5. In the window of the client that did not receive the non-text control message, enter Control-C to exit the program.
6. Next, run the `synchconsumer` clients using a topic. In each window, run the following command:

   ```
   appclient -client target/synchconsumer.jar topic
   ```

   Wait until you see the "Destination type is topic" message in both windows.
7. In the `producer` window, run the client, sending 20 or so messages to the topic:

   ```
   appclient -client target/producer.jar topic 20
   ```
8. Again, look at the output in the `synchconsumer` windows. In publish/subscribe messaging, a copy of every message is sent to each subscription on the topic. Therefore, each of the clients receives all 20 text messages as well as the non-text control message.

Acknowledging Messages

Jakarta Messaging provides two alternative ways for a consuming client to ensure that a message is not acknowledged until the application has finished processing the message:

- Using a synchronous consumer in a `JMSContext` that has been configured to use the `CLIENT_ACKNOWLEDGE` setting
- Using a message listener for asynchronous message delivery in a `JMSContext` that has been configured to use the default `AUTO_ACKNOWLEDGE` setting



In the Jakarta EE platform, `CLIENT_ACKNOWLEDGE` sessions can be used only in application clients, as in this example.

The `clientackconsumer` example demonstrates the first alternative, in which a synchronous consumer uses client acknowledgment. The `asynchconsumer` example described in Using a Message Listener for Asynchronous Message Delivery demonstrates the second alternative.

For information about message acknowledgment, see Controlling Message Acknowledgment.

The following table describes four possible interactions between types of consumers and types of acknowledgment.

Message Acknowledgment with Synchronous and Asynchronous Consumers

Consumer Type	Acknowledgment Type	Behavior
Synchronous	Client	Client acknowledges message after processing is complete
Asynchronous	Client	Client acknowledges message after processing is complete
Synchronous	Auto	Acknowledgment happens immediately after `receive` call; message cannot be redelivered if any subsequent processing steps fail
Asynchronous	Auto	Message is automatically acknowledged when `onMessage` method returns

The example is under the `jakartaee-examples/tutorial/jms/simple/clientackconsumer/` directory.

The example client, `ClientAckConsumer.java`, creates a `JMSContext` that specifies client acknowledgment:

```
try (JMSContext context =
      connectionFactory.createContext(JMSContext.CLIENT_ACKNOWLEDGE);) {
    ...
}
```

The client uses a `while` loop almost identical to that used by `SynchConsumer.java`, with the exception that after processing each message, it calls the `acknowledge` method on the `JMSContext`:

```
context.acknowledge();
```

The example uses the following objects:

- The `jms/MyQueue` resource that you created for Receiving Messages Synchronously.
- `java:comp/DefaultJMSConnectionFactory`, the platform default connection factory preconfigured with GlassFish Server

To Run the ClientAckConsumer Client

1. In a terminal window, go to the following directory:

   ```
   jakartaee-examples/tutorial/jms/simple/producer/
   ```

2. Run the `Producer` client, sending some messages to the queue:

   ```
   appclient -client target/producer.jar queue 3
   ```

3. In another terminal window, go to the following directory:

   ```
   jakartaee-examples/tutorial/jms/simple/clientackconsumer/
   ```

4. To run the client, use the following command:

   ```
   appclient -client target/clientackconsumer.jar
   ```

   The client output looks like this (along with some additional output):

   ```
   Created client-acknowledge JMSContext
   Reading message: This is message 1 from producer
   Acknowledging TextMessage
   Reading message: This is message 2 from producer
   Acknowledging TextMessage
   Reading message: This is message 3 from producer
   Acknowledging TextMessage
   Acknowledging non-text control message
   ```

   The client acknowledges each message explicitly after processing it, just as a `JMSContext` configured to use `AUTO_ACKNOWLEDGE` does automatically after a `MessageListener` returns successfully from processing a message received asynchronously.

Writing More Advanced Jakarta Messaging Applications

The following examples show how to use some of the more advanced Jakarta Messaging features: durable subscriptions and transactions.

Using Durable Subscriptions

The `durablesubscriptionexample` example shows how unshared durable subscriptions work. It demonstrates that a durable subscription continues to exist and accumulate messages even when there is no active consumer on it.

The example consists of two modules, a `durableconsumer` application that creates a durable subscription and consumes messages, and an `unsubscriber` application that enables you to unsubscribe from the durable subscription after you have finished running the `durableconsumer` application.

For information on durable subscriptions, see Creating Durable Subscriptions.

The main client, `DurableConsumer.java`, is under the `jakartaee-examples/tutorial/jms/durablesubscriptionexample/durableconsumer` directory.

The example uses a connection factory, `jms/DurableConnectionFactory`, that has a client ID.

The `DurableConsumer` client creates a `JMSContext` using the connection factory. It then stops the `JMSContext`, calls `createDurableConsumer` to create a durable subscription and a consumer on the topic by specifying a subscription name, registers a message listener, and starts the `JMSContext` again. The subscription is created only if it does not already exist, so the example can be run repeatedly:

```
try (JMSContext context = durableConnectionFactory.createContext();) {
    context.stop();
    consumer = context.createDurableConsumer(topic, "MakeItLast");
    listener = new TextListener();
    consumer.setMessageListener(listener);
    context.start();
    ...
}
```

To send messages to the topic, you run the `producer` client.

The `unsubscriber` example contains a very simple `Unsubscriber` client, which creates a `JMSContext` on the same connection factory and then calls the `unsubscribe` method, specifying the subscription name:

```
try (JMSContext context = durableConnectionFactory.createContext();) {
    System.out.println("Unsubscribing from durable subscription");
    context.unsubscribe("MakeItLast");
    ...
}
```

To Create Resources for the Durable Subscription Example

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a command window, go to the `durableconsumer` example.

```
cd jakartaee-examples/tutorial/jms/durablesubscriptionexample/durableconsumer
```

3. Create the resources using the `asadmin add-resources` command:

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```

   The command output reports the creation of a connector connection pool and a connector resource.

4. Verify the creation of the resources:

```
asadmin list-jms-resources
```

   In addition to the resources you created for the simple examples, the command lists the new connection factory:

```
jms/MyQueue
jms/MyTopic
jms/__defaultConnectionFactory
jms/DurableConnectionFactory
Command list-jms-resources executed successfully.
```

To Run the Durable Subscription Example

1. In a terminal window, go to the following directory:

```
jakartaee-examples/tutorial/jms/durablesubscriptionexample/
```

2. Build the `durableconsumer` and `unsubscriber` examples:

```
mvn install
```

3. Go to the `durableconsumer` directory:

```
cd durableconsumer
```

4. To run the client, enter the following command:

```
appclient -client target/durableconsumer.jar
```

The client creates the durable consumer and then waits for messages:

```
Creating consumer for topic
Starting consumer
To end program, enter Q or q, then <return>
```

5. In another terminal window, run the `Producer` client, sending some messages to the topic:

```
cd jakartaee-examples/tutorial/jms/simple/producer
appclient -client target/producer.jar topic 3
```

6. After the `DurableConsumer` client receives the messages, enter `q` or `Q` to exit the program. At this point, the client has behaved like any other asynchronous consumer.
7. Now, while the `DurableConsumer` client is not running, use the `Producer` client to send more messages:

```
appclient -client target/producer.jar topic 2
```

If a durable subscription did not exist, these messages would be lost, because no consumer on the topic is currently running. However, the durable subscription is still active, and it retains the messages.

8. Run the `DurableConsumer` client again. It immediately receives the messages that were sent while it was inactive:

```
Creating consumer for topic
Starting consumer
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Message is not a TextMessage
```

9. Enter `q` or `Q` to exit the program.

To Run the unsubscriber Example

After you have finished running the `DurableConsumer` client, run the `unsubscriber` example to unsubscribe from the durable subscription.

1. In a terminal window, go to the following directory:

```
jakartaee-examples/tutorial/jms/durablesubscriptionexample/unsubscriber
```

2. To run the `Unsubscriber` client, enter the following command:

```
appclient -client target/unsubscriber.jar
```

   The client reports that it is unsubscribing from the durable subscription.

Using Local Transactions

The `transactedexample` example demonstrates the use of local transactions in a Messaging client application. It also demonstrates the use of the request/reply messaging pattern described in Creating Temporary Destinations, although it uses permanent rather than temporary destinations. The example consists of three modules, `genericsupplier`, `retailer`, and `vendor`, which can be found under the `jakartaee-examples/tutorial/jms/transactedexample/` directory. The source code can be found in the `src/main/java/jakarta.tutorial` trees for each module. The `genericsupplier` and `retailer` modules each contain a single class, `genericsupplier/GenericSupplier.java` and `retailer/Retailer.java`, respectively. The `vendor` module is more complex, containing four classes: `vendor/Vendor.java`, `vendor/VendorMessageListener.java`, `vendor/Order.java`, and `vendor/SampleUtilities.java`.

The example shows how to use a queue and a topic in a single transaction as well as how to pass a `JMSContext` to a message listener's constructor function. The example represents a highly simplified e-commerce application in which the following actions occur.

1. A retailer (`retailer/src/main/java/jakarta/tutorial/retailer/Retailer.java`) sends a `MapMessage` to a vendor order queue, ordering a quantity of computers, and waits for the vendor's reply:

```
outMessage = context.createMapMessage();
outMessage.setString("Item", "Computer(s)");
outMessage.setInt("Quantity", quantity);
outMessage.setJMSReplyTo(retailerConfirmQueue);
context.createProducer().send(vendorOrderQueue, outMessage);
System.out.println("Retailer: ordered " + quantity + " computer(s)");
orderConfirmReceiver = context.createConsumer(retailerConfirmQueue);
```

2. The vendor (`vendor/src/main/java/jakarta/tutorial/vendor/Vendor.java`) receives the retailer's order message and sends an order message to the supplier order topic in one transaction. This Jakarta Messaging transaction uses a single session, so you can combine a receive from a queue with a send to a topic. Here is the code that uses the same session to create a consumer for a queue:

```
vendorOrderReceiver = session.createConsumer(vendorOrderQueue);
```

   The following code receives the incoming message, sends an outgoing message, and commits

the `JMSContext`. The message processing has been removed to keep the sequence simple:

```
inMessage = vendorOrderReceiver.receive();
// Process the incoming message and format the outgoing
// message
...
context.createProducer().send(supplierOrderTopic, orderMessage);
...
context.commit();
```

For simplicity, there are only two suppliers, one for CPUs and one for hard drives.

3. Each supplier (`genericsupplier/src/main/java/jakarta/tutorial/genericsupplier/GenericSupplier.java`) receives the order from the order topic, checks its inventory, and then sends the items ordered to the queue named in the order message's `JMSReplyTo` field. If it does not have enough of the item in stock, the supplier sends what it has. The synchronous receive from the topic and the send to the queue take place in one Jakarta Messaging transaction:

```
receiver = context.createConsumer(SupplierOrderTopic);
...
inMessage = receiver.receive();
if (inMessage instanceof MapMessage) {
    orderMessage = (MapMessage) inMessage;
} ...
// Process message
outMessage = context.createMapMessage();
// Add content to message
context.createProducer().send(
         (Queue) orderMessage.getJMSReplyTo(),
         outMessage);
// Display message contents
context.commit();
```

4. The vendor receives the suppliers' replies from its confirmation queue and updates the state of the order. Messages are processed by an asynchronous message listener, `VendorMessageListener`; this step shows the use of Jakarta Messaging transactions with a message listener:

```
MapMessage component = (MapMessage) message;
...
int orderNumber = component.getInt("VendorOrderNumber");
Order order = Order.getOrder(orderNumber).processSubOrder(component);
context.commit();
```

5. When all outstanding replies are processed for a given order, the vendor message listener sends a message notifying the retailer whether it can fulfill the order:

```
Queue replyQueue = (Queue) order.order.getJMSReplyTo();
MapMessage retailerConfirmMessage = context.createMapMessage();
// Format the message
context.createProducer().send(replyQueue, retailerConfirmMessage);
context.commit();
```

6. The retailer receives the message from the vendor:

```
inMessage = (MapMessage) orderConfirmReceiver.receive();
```

The retailer then places a second order for twice as many computers as in the first order, so these steps are executed twice.

Figure 32, “Transactions: Messaging Client Example” illustrates these steps.

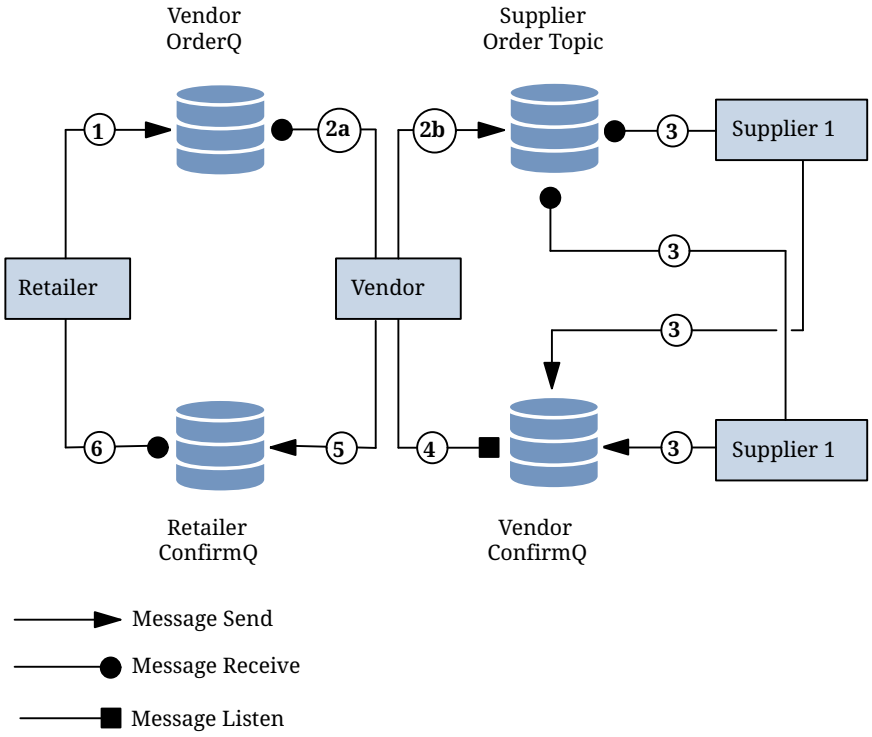


Figure 32. Transactions: Messaging Client Example

All the messages use the `MapMessage` message type. Synchronous receives are used for all message reception except when the vendor processes the replies of the suppliers. These replies are processed asynchronously and demonstrate how to use transactions within a message listener.

At random intervals, the `Vendor` client throws an exception to simulate a database problem and cause a rollback.

All clients except `Retailer` use transacted contexts.

The example uses three queues named `jms/AQueue`, `jms/BQueue`, and `jms/CQueue`, and one topic named `jms/OTopic`.

To Create Resources for the transactedexample Example

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a command window, go to the `genericsupplier` example:

```
cd jakartaee-examples/tutorial/jms/transactedexample/genericsupplier
```

3. Create the resources using the `asadmin add-resources` command:

```
asadmin add-resources src/main/setup/glassfish-resources.xml
```

4. Verify the creation of the resources:

```
asadmin list-jms-resources
```

   In addition to the resources you created for the simple examples and the durable subscription example, the command lists the four new destinations:

```
jms/MyQueue
jms/MyTopic
jms/AQueue
jms/BQueue
jms/CQueue
jms/OTopic
jms/__defaultConnectionFactory
jms/DurableConnectionFactory
Command list-jms-resources executed successfully.
```

To Run the transactedexample Clients

You will need four terminal windows to run the clients. Make sure that you start the clients in the correct order.

1. In a terminal window, go to the following directory:

```
jakartaee-examples/tutorial/jms/transactedexample/
```

2. To build and package all the modules, enter the following command:

```
mvn install
```

3. Go to the `genericsupplier` directory:

```
cd genericsupplier
```

4. Use the following command to start the CPU supplier client:

```
appclient -client target/genericsupplier.jar CPU
```

After some initial output, the client reports the following:

```
Starting CPU supplier
```

5. In a second terminal window, go to the `genericsupplier` directory:

```
cd jakartaee-examples/tutorial/jms/transactedexample/genericsupplier
```

6. Use the following command to start the hard drive supplier client:

```
appclient -client target/genericsupplier.jar HD
```

After some initial output, the client reports the following:

```
Starting Hard Drive supplier
```

7. In a third terminal window, go to the `vendor` directory:

```
cd jakartaee-examples/tutorial/jms/transactedexample/vendor
```

8. Use the following command to start the `Vendor` client:

```
appclient -client target/vendor.jar
```

After some initial output, the client reports the following:

```
Starting vendor
```

9. In another terminal window, go to the `retailer` directory:

```
cd jakartaee-examples/tutorial/jms/transactedexample/retailer
```

10. Use a command like the following to run the `Retailer` client. The argument specifies the number

of computers to order:

```
appclient -client target/retailer.jar 4
```

After some initial output, the `Retailer` client reports something like the following. In this case, the first order is filled, but the second is not:

```
Retailer: Quantity to be ordered is 4
Retailer: Ordered 4 computer(s)
Retailer: Order filled
Retailer: Placing another order
Retailer: Ordered 8 computer(s)
Retailer: Order not filled
```

The `Vendor` client reports something like the following, stating in this case that it is able to send all the computers in the first order, but not in the second:

```
Vendor: Retailer ordered 4 Computer(s)
Vendor: Ordered 4 CPU(s) and hard drive(s)
  Vendor: Committed transaction 1
Vendor: Completed processing for order 1
Vendor: Sent 4 computer(s)
  Vendor: committed transaction 2
Vendor: Retailer ordered 8 Computer(s)
Vendor: Ordered 8 CPU(s) and hard drive(s)
  Vendor: Committed transaction 1
Vendor: Completed processing for order 2
Vendor: Unable to send 8 computer(s)
  Vendor: Committed transaction 2
```

The CPU supplier reports something like the following. In this case, it is able to send all the CPUs for both orders:

```
CPU Supplier: Vendor ordered 4 CPU(s)
CPU Supplier: Sent 4 CPU(s)
  CPU Supplier: Committed transaction
CPU Supplier: Vendor ordered 8 CPU(s)
CPU Supplier: Sent 8 CPU(s)
  CPU Supplier: Committed transaction
```

The hard drive supplier reports something like the following. In this case, it has a shortage of hard drives for the second order:

```
Hard Drive Supplier: Vendor ordered 4 Hard Drive(s)
Hard Drive Supplier: Sent 4 Hard Drive(s)
```

```
  Hard Drive Supplier: Committed transaction
Hard Drive Supplier: Vendor ordered 8 Hard Drive(s)
Hard Drive Supplier: Sent 1 Hard Drive(s)
  Hard Drive Supplier: Committed transaction
```

11. Repeat Step 10 as many times as you wish. Occasionally, the vendor will report an exception that causes a rollback:

```
Vendor: JMSException occurred: jakarta.jms.JMSException: Simulated
database concurrent access exception
  Vendor: Rolled back transaction 1
```

12. After you finish running the clients, you can delete the destination resources by using the following commands:

```
asadmin delete-jms-resource jms/AQueue
asadmin delete-jms-resource jms/BQueue
asadmin delete-jms-resource jms/CQueue
asadmin delete-jms-resource jms/OTopic
```

Writing High Performance and Scalable Jakarta Messaging Applications

This section describes how to use Jakarta Messaging to write applications that can handle high volumes of messages robustly. These examples use both nondurable and durable shared consumers.

Using Shared Nondurable Subscriptions

This section describes the receiving clients in an example that shows how to use a shared consumer to distribute messages sent to a topic among different consumers. This section then explains how to compile and run the clients using GlassFish Server.

You may wish to compare this example to the results of Running Multiple Consumers on the Same Destination using an unshared consumer. In that example, messages are distributed among the consumers on a queue, but each consumer on the topic receives all the messages because each consumer on the topic is using a separate topic subscription.

In this example, however, messages are distributed among multiple consumers on a topic, because all the consumers are sharing the same subscription. Each message added to the topic subscription is received by only one consumer, similarly to the way in which each message added to a queue is received by only one consumer.

A topic may have multiple subscriptions. Each message sent to the topic will be added to each topic subscription. However, if there are multiple consumers on a particular subscription, each message added to that subscription will be delivered to only one of those consumers.

Writing the Clients for the Shared Consumer Example

The sending client is `Producer.java`, the same client used in previous examples.

The receiving client is `SharedConsumer.java`. It is very similar to `AsynchConsumer.java`, except that it always uses a topic. It performs the following steps.

1. Injects resources for a connection factory and topic.
2. In a `try-with-resources` block, creates a `JMSContext`.
3. Creates a consumer on a shared nondurable subscription, specifying a subscription name:

   ```
   consumer = context.createSharedConsumer(topic, "SubName");
   ```

4. Creates an instance of the `TextListener` class and registers it as the message listener for the shared consumer.
5. Listens for the messages published to the destination, stopping when the user types the character `q` or `Q`.
6. Catches and handles any exceptions. The end of the `try-with-resources` block automatically causes the `JMSContext` to be closed.

The `TextListener.java` class is identical to the one for the `asynchconsumer` example.

For this example, you will use the default connection factory and the topic you created in To Create Resources for the Simple Examples.

To Run the SharedConsumer and Producer Clients

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. Open three command windows. In the first, go to the `simple/producer/` directory:

   ```
   cd jakartaee-examples/tutorial/jms/simple/producer/
   ```

3. In the second and third command windows, go to the `shared/sharedconsumer/` directory:

   ```
   cd jakartaee-examples/tutorial/jms/shared/sharedconsumer/
   ```

4. In one of the `sharedconsumer` windows, build the example:

   ```
   mvn install
   ```

5. In each of the two `sharedconsumer` windows, start running the client. You do not need to specify a `topic` argument:

```
appclient -client target/sharedconsumer.jar
```

Wait until you see the following output in both windows:

```
Waiting for messages on topic
To end program, enter Q or q, then <return>
```

6. In the `producer` window, run the client, specifying the topic and a number of messages:

```
appclient -client target/producer.jar topic 20
```

Each consumer client receives some of the messages. Only one of the clients receives the non-text message that signals the end of the message stream.

7. Enter `Q` or `q` and press Return to stop each client and see a report of the number of text messages received.

Using Shared Durable Subscriptions

The `shareddurableconsumer` client shows how to use shared durable subscriptions. It shows how shared durable subscriptions combine the advantages of durable subscriptions (the subscription remains active when the client is not) with those of shared consumers (the message load can be divided among multiple clients).

The example is much more similar to the `sharedconsumer` example than to the `DurableConsumer.java` client. It uses two classes, `SharedDurableConsumer.java` and `TextListener.java`, which can be found under the `jakartaee-examples/tutorial/jms/shared/shareddurableconsumer/` directory.

The client uses `java:comp/DefaultJMSConnectionFactory`, the connection factory that does not have a client identifier, as is recommended for shared durable subscriptions. It uses the `createSharedDurableConsumer` method with a subscription name to establish the subscription:

```
consumer = context.createSharedDurableConsumer(topic, "MakeItLast");
```

You run the example in combination with the `Producer.java` client.

To Run the SharedDurableConsumer and Producer Clients

1. In a terminal window, go to the following directory:

```
jakartaee-examples/tutorial/jms/shared/shareddurableconsumer
```

2. To compile and package the client, enter the following command:

```
mvn install
```

3. Run the client first to establish the durable subscription:

```
appclient -client target/shareddurableconsumer.jar
```

4. The client displays the following and pauses:

```
Waiting for messages on topic
To end program, enter Q or q, then <return>
```

5. In the `shareddurableconsumer` window, enter `q` or `Q` to exit the program. The subscription remains active, although the client is not running.
6. Open another terminal window and go to the `producer` example directory:

```
cd jakartaee-examples/tutorial/jms/simple/producer
```

7. Run the `producer` example, sending a number of messages to the topic:

```
appclient -client target/producer.jar topic 6
```

8. After the producer has sent the messages, open a third terminal window and go to the `shareddurableconsumer` directory.
9. Run the client in both the first and third terminal windows. Whichever client starts first will receive all the messages that were sent when there was no active subscriber:

```
appclient -client target/shareddurableconsumer.jar
```

10. With both `shareddurableconsumer` clients still running, go to the `producer` window and send a larger number of messages to the topic:

```
appclient -client target/producer.jar topic 25
```

    Now the messages will be shared by the two consumer clients. If you continue sending groups of messages to the topic, each client receives some of the messages. If you exit one of the clients and send more messages, the other client will receive all the messages.

Sending and Receiving Messages Using a Simple Web Application

Web applications can use Jakarta Messaging to send and receive messages, as noted in Using Jakarta EE Components to Produce and to Synchronously Receive Messages. This section describes

the components of a very simple web application that uses Jakarta Messaging.

This section assumes that you are familiar with the basics of Jakarta Faces technology, described in the Introduction to Jakarta Faces Technology

The example, `websimplemessage`, is under the `jakartaee-examples/tutorial/jms/` directory. It uses sending and receiving Facelets pages as well as corresponding backing beans. When a user enters a message in the text field of the sending page and clicks a button, the backing bean for the page sends the message to a queue and displays it on the page. When the user goes to the receiving page and clicks another button, the backing bean for that page receives the message synchronously and displays it.

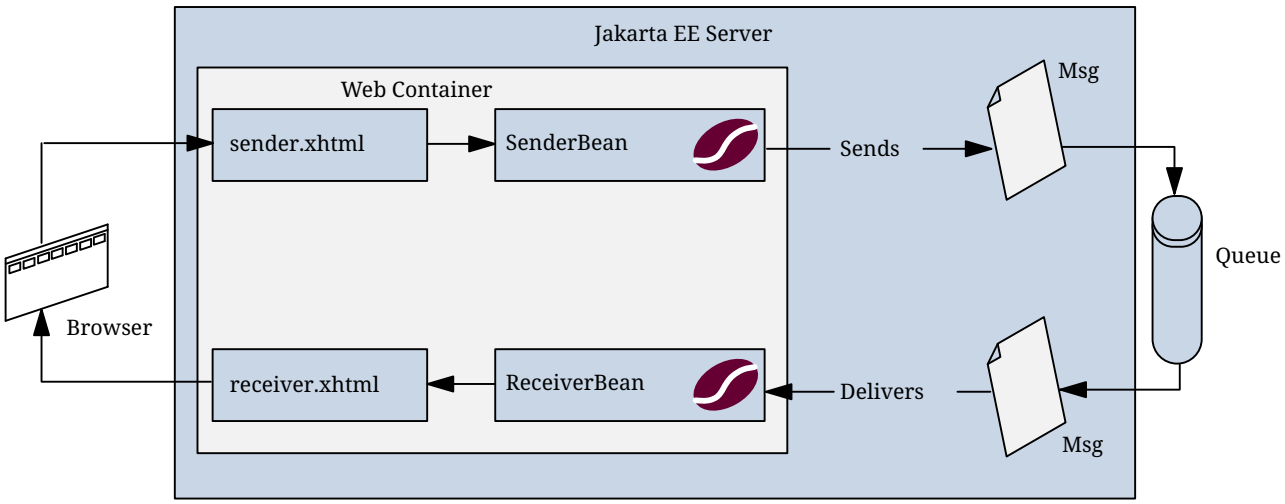


Figure 33. The websimplemessage Application

The websimplemessage Facelets Pages

The Facelets pages for the example are as follows.

- `sender.xhtml`, which provides a labeled `h:InputText` tag where the user enters the message, along with two command buttons. When the user clicks the Send Message button, the `senderBean.sendMessage` method is called to send the message to the queue and display its contents. When the user clicks the Go to Receive Page button, the `receiver.xhtml` page appears.
- `receiver.xhtml`, which also provides two command buttons. When the user clicks the Receive Message button, the `receiverBean.getMessage` method is called to fetch the message from the queue and display its contents. When the user clicks the Send Another Message button, the sender.xhtml page appears again.

The websimplemessage Managed Beans

The two managed beans for the example are as follows.

- `SenderBean.java`, a CDI managed bean with one property, `messageText`, and one business method, `sendMessage`. The class is annotated with `@JMSDestinationDefinition` to create a component-private queue:

```
@JMSDestinationDefinition(
```

```
        name = "java:comp/jms/webappQueue",
        interfaceName = "jakarta.jms.Queue",
        destinationName = "PhysicalWebappQueue")
@Named
@RequestScoped
public class SenderBean { ... }
```

The `sendMessage` method injects a `JMSContext` (using the default connection factory) and the queue, creates a producer, sends the message the user typed on the Facelets page, and creates a `FacesMessage` to display on the Facelets page:

```
@Inject
private JMSContext context;
@Resource(lookup = "java:comp/jms/webappQueue")
private Queue queue;
private String messageText;
...
public void sendMessage() {
    try {
        String text = "Message from producer: " + messageText;
        context.createProducer().send(queue, text);

        FacesMessage facesMessage =
                new FacesMessage("Sent message: " + text);
        FacesContext.getCurrentInstance().addMessage(null, facesMessage);
    } catch (Throwable t) {
        logger.log(Level.SEVERE,
                "SenderBean.sendMessage: Exception: {0}",
                t.toString());
    }
}
```

- `ReceiverBean.java`, a CDI managed bean with one business method, `getMessage`. The method injects a `JMSContext` (using the default connection factory) and the queue that was defined in `SenderBean`, creates a consumer, receives the message, and creates a `FacesMessage` to display on the Facelets page:

```
@Inject
private JMSContext context;
@Resource(lookup = "java:comp/jms/webappQueue")
private Queue queue;
...
public void getMessage() {
    try {
        JMSConsumer receiver = context.createConsumer(queue);
        String text = receiver.receiveBody(String.class);

        if (text != null) {
```

```
            FacesMessage facesMessage =
                    new FacesMessage("Reading message: " + text);
            FacesContext.getCurrentInstance().addMessage(null, facesMessage);
        } else {
            FacesMessage facesMessage =
                    new FacesMessage("No message received after 1 second");
            FacesContext.getCurrentInstance().addMessage(null, facesMessage);
        }
    } catch (Throwable t) {
        logger.log(Level.SEVERE,
                "ReceiverBean.getMessage: Exception: {0}",
                t.toString());
    }
}
```

Running the websimplemessage Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `websimplemessage` application.

Creating Resources for the websimplemessage Example

This example uses an annotation-defined queue and the preconfigured default connection factory `java:comp/DefaultJMSConnectionFactory`.

To Package and Deploy websimplemessage Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/jms
   ```

4. Select the `websimplemessage` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `websimplemessage` project and select **Build**.

   This command builds and deploys the project.

To Package and Deploy websimplemessage Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/jms/websimplemessage/
   ```

3. To compile the source files and package and deploy the application, use the following command:

```
mvn install
```

To Run the websimplemessage Example

1. In a web browser, enter the following URL:

```
http://localhost:8080/websimplemessage
```

2. Enter a message in the text field and click Send Message.

   If, for example, you enter "Hello, Duke", the following appears below the buttons:

```
Sent message: Message from producer: Hello, Duke
```

3. Click Go to Receive Page.
4. Click Receive Message.

   The following appears below the buttons:

```
Reading message: Message from producer: Hello, Duke
```

5. Click Send Another Message to return to the sending page.
6. After you have finished running the application, undeploy it using either the Services tab of NetBeans IDE or the `mvn cargo:undeploy` command.

Receiving Messages Asynchronously Using a Message-Driven Bean

If you are writing an application to run in the Jakarta EE application client container or on the Java SE platform, and you want to receive messages asynchronously, you need to define a class that implements the `MessageListener` interface, create a `JMSConsumer`, and call the method `setMessageListener`.

If you're writing an application to run in the Jakarta EE web or enterprise bean container and want it to receive messages asynchronously, you also need to need to define a class that implements the `MessageListener` interface. However, instead of creating a `JMSConsumer` and calling the method `setMessageListener`, you must configure your message listener class to be a message-driven bean. The application server will then take care of the rest.

Message-driven beans can implement any messaging type. Most commonly, however, they implement the Jakarta Messaging technology.

This section describes a simple message-driven bean example. Before proceeding, you should read

the basic conceptual information in the section What Is a Message-Driven Bean? as well as Using Message-Driven Beans to Receive Messages Asynchronously.

Overview of the simplemessage Example

The `simplemessage` application has the following components:

- `SimpleMessageClient`: An application client that sends several messages to a queue
- `SimpleMessageBean`: A message-driven bean that asynchronously processes the messages that are sent to the queue

Figure 34, "The simplemessage Application" illustrates the structure of this application. The application client sends messages to the queue, which was created administratively using the Administration Console. The Messaging provider (in this case, GlassFish Server) delivers the messages to the instances of the message-driven bean, which then processes the messages.

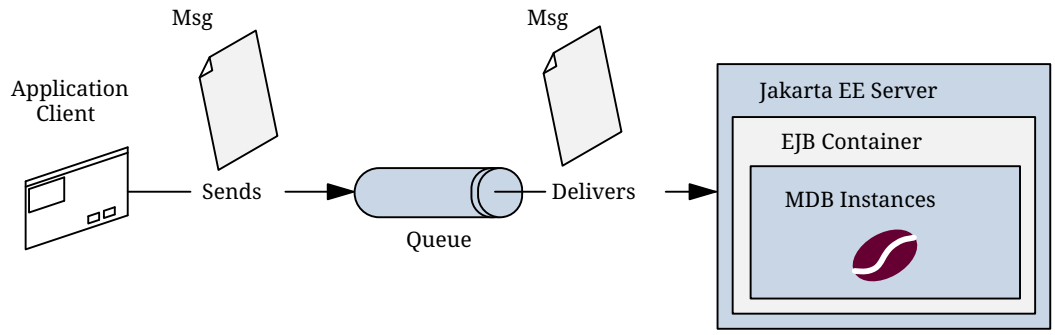


Figure 34. The simplemessage Application

The source code for this application is in the `jakartaee-examples/tutorial/jms/simplemessage/` directory.

The simplemessage Application Client

The `SimpleMessageClient` sends messages to the queue that the `SimpleMessageBean` listens to. The client starts by injecting the connection factory and queue resources:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(lookup = "jms/MyQueue")
private static Queue queue;
```

Next, the client creates the `JMSContext` in a `try-with-resources` block:

```
String text;
final int NUM_MSGS = 3;

try (JMSContext context = connectionFactory.createContext();) { ... }
```

Finally, the client sends several text messages to the queue:

```
for (int i = 0; i < NUM_MSGS; i++) {
    text = "This is message " + (i + 1);
    System.out.println("Sending message: " + text);
    context.createProducer().send(queue, text);
}
```

The simplemessage Message-Driven Bean Class

The code for the `SimpleMessageBean` class illustrates the requirements of a message-driven bean class described in Using Message-Driven Beans to Receive Messages Asynchronously.

The first few lines of the `SimpleMessageBean` class use the `@MessageDriven` annotation's `activationConfig` attribute to specify configuration properties:

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "jms/MyQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Queue")
})
```

See null for a list of the available properties.

See Sending Messages from a Session Bean to an MDB for examples of the `subscriptionDurability`, `clientId`, `subscriptionName`, and `messageSelector` properties.

The onMessage Method

When the queue receives a message, the enterprise bean container invokes the message listener method or methods. For a bean that uses Jakarta Messaging, this is the `onMessage` method of the `MessageListener` interface.

In the `SimpleMessageBean` class, the `onMessage` method casts the incoming message to a `TextMessage` and displays the text:

```
public void onMessage(Message inMessage) {

    try {
        if (inMessage instanceof TextMessage) {
            logger.log(Level.INFO,
                    "MESSAGE BEAN: Message received: {0}",
                    inMessage.getBody(String.class));
        } else {
            logger.log(Level.WARNING,
                    "Message of wrong type: {0}",
                    inMessage.getClass().getName());
```

```
        }
    } catch (JMSException e) {
        logger.log(Level.SEVERE,
                "SimpleMessageBean.onMessage: JMSException: {0}",
                e.toString());
        mdc.setRollbackOnly();
    }
}
```

Running the simplemessage Example

You can use either NetBeans IDE or Maven to build, deploy, and run the `simplemessage` example.

Creating Resources for the simplemessage Example

This example uses the queue named `jms/MyQueue` and the preconfigured default connection factory `java:comp/DefaultJMSConnectionFactory`.

If you have run the simple Jakarta Messaging examples in Writing Simple Jakarta Messaging Applications and have not deleted the resources, you already have the queue. Otherwise, follow the instructions in To Create Resources for the Simple Examples to create it.

For more information on creating Messaging resources, see Creating Jakarta Messaging Administered Objects.

To Run the simplemessage Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/jms/simplemessage
   ```

4. Select the `simplemessage` folder.
5. Make sure that the **Open Required Projects** check box is selected, then click **Open Project**.
6. In the **Projects** tab, right-click the `simplemessage` project and select **Build**. (If NetBeans IDE suggests that you run a priming build, click the box to do so.)

   This command packages the application client and the message-driven bean, then creates a file named `simplemessage.ear` in the `simplemessage-ear/target/` directory. It then deploys the `simplemessage-ear` module, retrieves the client stubs, and runs the application client.

   The output in the output window looks like this (preceded by application client container output):

   ```
   Sending message: This is message 1
   Sending message: This is message 2
   ```

```
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

In the server log file, lines similar to the following appear:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

The received messages may appear in a different order from the order in which they were sent.

7. After you have finished running the application, undeploy it using the **Services** tab.

To Run the simplemessage Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

```
jakartaee-examples/tutorial/jms/simplemessage/
```

3. To compile the source files and package the application, use the following command:

```
mvn install
```

This target packages the application client and the message-driven bean, then creates a file named `simplemessage.ear` in the `simplemessage-ear/target/` directory. It then deploys the `simplemessage-ear` module, retrieves the client stubs, and runs the application client.

The output in the terminal window looks like this (preceded by application client container output):

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

In the server log file, lines similar to the following appear:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

The received messages may appear in a different order from the order in which they were sent.

4. After you have finished running the application, undeploy it using the `mvn cargo:undeploy` command.

Sending Messages from a Session Bean to an MDB

This section explains how to write, compile, package, deploy, and run an application that uses Jakarta Messaging in conjunction with a session bean. The application contains the following components:

- An application client that invokes a session bean
- A session bean that publishes several messages to a topic
- A message-driven bean that receives and processes the messages using a durable topic subscription and a message selector

You will find the source files for this section in the `jakartaee-examples/tutorial/jms/clientsessionmdb/` directory. Path names in this section are relative to this directory.

Writing the Application Components for the clientsessionmdb Example

This application demonstrates how to send messages from an enterprise bean (in this case, a session bean) rather than from an application client, as in the example in Receiving Messages Asynchronously Using a Message-Driven Bean. Figure 35, "An Enterprise Bean Application: Client to Session Bean to Message-Driven Bean" illustrates the structure of this application. Sending messages from an enterprise bean is very similar to sending messages from a managed bean, which was shown in Sending and Receiving Messages Using a Simple Web Application.

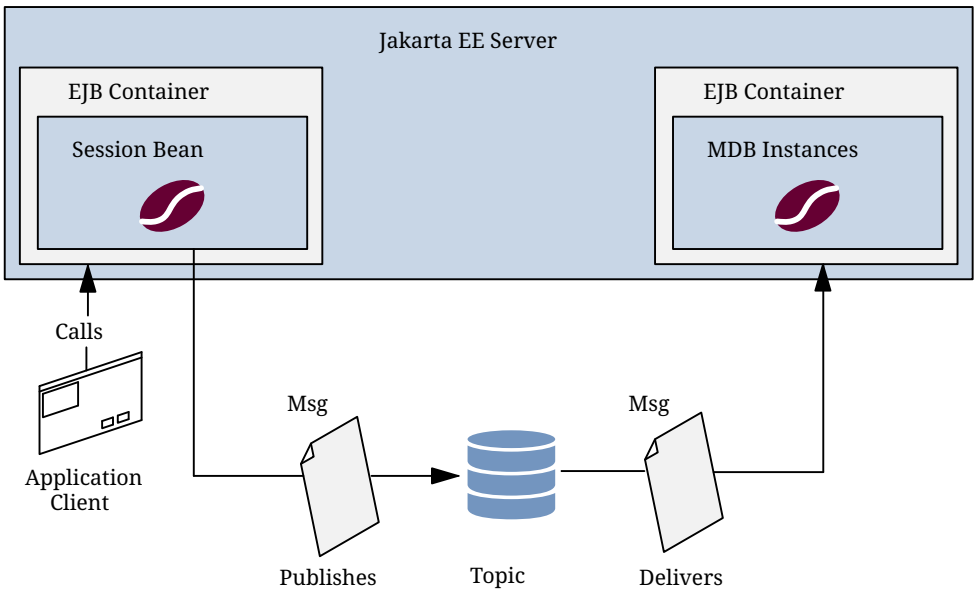


Figure 35. An Enterprise Bean Application: Client to Session Bean to Message-Driven Bean

The Publisher enterprise bean in this example is the enterprise-application equivalent of a wire-service news feed that categorizes news events into six news categories. The message-driven bean could represent a newsroom, where the sports desk, for example, would set up a subscription for

all news events pertaining to sports.

The application client in the example injects the Publisher enterprise bean's remote home interface and then calls the bean's business method. The enterprise bean creates 18 text messages. For each message, it sets a `String` property randomly to one of six values representing the news categories and then publishes the message to a topic. The message-driven bean uses a message selector for the property to limit which of the published messages will be delivered to it.

Coding the Application Client: MyAppClient.java

The application client, `MyAppClient.java`, found under `clientsessionmdb-app-client`, performs no Messaging operations and so is simpler than the client in Receiving Messages Asynchronously Using a Message-Driven Bean. The client uses dependency injection to obtain the Publisher enterprise bean's business interface:

```
@EJB(name="PublisherRemote")
private static PublisherRemote publisher;
```

The client then calls the bean's business method twice.

Coding the Publisher Session Bean

The Publisher bean is a stateless session bean that has one business method. The Publisher bean uses a remote interface rather than a local interface because it is accessed from the application client.

The remote interface, `PublisherRemote.java`, found under `clientsessionmdb-ejb`, declares a single business method, `publishNews`.

The bean class, `PublisherBean.java`, also found under `clientsessionmdb-ejb`, implements the `publishNews` method and its helper method `chooseType`. The bean class injects `SessionContext` and `Topic` resources (the topic is defined in the message-driven bean). It then injects a `JMSContext`, which uses the preconfigured default connection factory unless you specify otherwise. The bean class begins as follows:

```
@Stateless
@Remote({
    PublisherRemote.class
})
public class PublisherBean implements PublisherRemote {

    @Resource
    private SessionContext sc;
    @Resource(lookup = "java:module/jms/newsTopic")
    private Topic topic;
    @Inject
    private JMSContext context;
    ...
```

```
}
```

The business method `publishNews` creates a `JMSProducer` and publishes the messages.

Coding the Message-Driven Bean: MessageBean.java

The message-driven bean class, `MessageBean.java`, found under `clientsessionmdb-ejb`, is almost identical to the one in Receiving Messages Asynchronously Using a Message-Driven Bean. However, the `@MessageDriven` annotation is different, because instead of a queue, the bean is using a topic, a durable subscription, and a message selector. The bean defines a topic for the use of the application; the definition uses the `java:module` scope because both the session bean and the message-driven bean are in the same module. Because the destination is defined in the message-driven bean, the `@MessageDriven` annotation uses the `destinationLookup` activation config property. (See Creating Resources for Jakarta EE Applications for more information.) The annotation also sets the activation config properties `messageSelector`, `subscriptionDurability`, `clientId`, and `subscriptionName`, as follows:

```
@JMSDestinationDefinition(
        name = "java:module/jms/newsTopic",
        interfaceName = "jakarta.jms.Topic",
        destinationName = "PhysicalNewsTopic")
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "java:module/jms/newsTopic"),
    @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Topic"),
    @ActivationConfigProperty(propertyName = "messageSelector",
            propertyValue = "NewsType = 'Sports' OR NewsType = 'Opinion'"),
    @ActivationConfigProperty(propertyName = "subscriptionDurability",
            propertyValue = "Durable"),
    @ActivationConfigProperty(propertyName = "clientId",
            propertyValue = "MyID"),
    @ActivationConfigProperty(propertyName = "subscriptionName",
            propertyValue = "MySub")
})
```

The topic is the one defined in the `PublisherBean`. The message selector in this case represents both the sports and opinion desks, just to demonstrate the syntax of message selectors.

The Jakarta Messaging resource adapter uses these properties to create a connection factory for the message-driven bean that allows the bean to use a durable subscription.

Running the clientsessionmdb Example

You can use either NetBeans IDE or Maven to build, deploy, and run the `simplemessage` example.

This example uses an annotation-defined topic and the preconfigured default connection factory `java:comp/DefaultJMSConnectionFactory`, so you do not have to create resources for it.

To Run clientsessionmdb Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/jms/clientsessionmdb
   ```

4. Select the `clientsessionmdb` folder.
5. Make sure that the **Open Required Projects** check box is selected, then click **Open Project**.
6. In the **Projects** tab, right-click the `clientsessionmdb` project and select **Build**. (If NetBeans IDE suggests that you run a priming build, click the box to do so.)

   This command creates the following:

   a. An application client JAR file that contains the client class file and the session bean's remote interface, along with a manifest file that specifies the main class and places the Jakarta Enterprise Beans JAR file in its classpath

   b. An enterprise bean JAR file that contains both the session bean and the message-driven bean

   c. An application EAR file that contains the two JAR files

   The `clientsessionmdb.ear` file is created in the `clientsessionmdb-ear/target/` directory.

   The command then deploys the EAR file, retrieves the client stubs, and runs the client.

   The client displays these lines:

   ```
   To view the bean output,
    check <install_dir>/domains/domain1/logs/server.log.
   ```

   The output from the enterprise beans appears in the server log file. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose `NewsType` property is `Sports` or `Opinion`.

7. Use the **Services** tab to undeploy the application after you have finished running it.

To Run clientsessionmdb Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. Go to the following directory:

   ```
   jakartaee-examples/tutorial/jms/clientsessionmdb/
   ```

3. To compile the source files and package, deploy, and run the application, enter the following command:

```
mvn install
```

This command creates the following:

- An application client JAR file that contains the client class file and the session bean's remote interface, along with a manifest file that specifies the main class and places the enterprise bean JAR file in its classpath
- An enterprise bean JAR file that contains both the session bean and the message-driven bean
- An application EAR file that contains the two JAR files

The `clientsessionmdb.ear` file is created in the `clientsessionmdb-ear/target/` directory.

The command then deploys the EAR file, retrieves the client stubs, and runs the client.

The client displays these lines:

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

The output from the enterprise beans appears in the server log file. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose `NewsType` property is `Sports` or `Opinion`.

4. Undeploy the application after you have finished running it:

```
mvn cargo:undeploy
```

Using an Entity to Join Messages from Two MDBs

This section explains how to write, compile, package, deploy, and run an application that uses the Jakarta Messaging with an entity. The application uses the following components:

- An application client that both sends and receives messages
- Two message-driven beans
- An entity class

You will find the source files for this section in the `jakartaee-examples/tutorial/jms/clientmdbentity/` directory. Path names in this section are relative to this directory.

Overview of the clientmdbentity Example Application

This application simulates, in a simplified way, the work flow of a company's human resources (HR) department when it processes a new hire. This application also demonstrates how to use the Jakarta EE platform to accomplish a task that many Messaging applications need to perform.

A messaging client must often wait for several messages from various sources. It then uses the information in all these messages to assemble a message that it then sends to another destination. The common term for this design pattern (which is not specific to Jakarta Messaging) is joining messages. Such a task must be transactional, with all the receives and the send as a single transaction. If not all the messages are received successfully, the transaction can be rolled back. For an application client example that illustrates this task, see Using Local Transactions.

A message-driven bean can process only one message at a time in a transaction. To provide the ability to join messages, an application can have the message-driven bean store the interim information in a Jakarta Persistence entity. The entity can then determine whether all the information has been received; when it has, the entity can report this back to one of the message-driven beans, which then creates and sends the message to the other destination. After it has completed its task, the entity can be removed.

The basic steps of the application are as follows.

1. The HR department's application client generates an employee ID for each new hire and then publishes a message (M1) containing the new hire's name, employee ID, and position. It publishes the message to a topic because the message needs to be consumed by two message-driven beans. The client then creates a temporary queue, `ReplyQueue`, with a message listener that waits for a reply to the message. (See Creating Temporary Destinations for more information.)
2. Two message-driven beans process each message: One bean, `OfficeMDB`, assigns the new hire's office number, and the other bean, `EquipmentMDB`, assigns the new hire's equipment. The first bean to process the message creates and persists an entity named `SetupOffice`, then calls a business method of the entity to store the information it has generated. The second bean locates the existing entity and calls another business method to add its information.
3. When both the office and the equipment have been assigned, the entity business method returns a value of `true` to the message-driven bean that called the method. The message-driven bean then sends to the reply queue a message (M2) describing the assignments. Then it removes the entity. The application client's message listener retrieves the information.

Figure 36, "An Enterprise Bean Application: Client to Message-Driven Beans to Entity" illustrates the structure of this application. Of course, an actual HR application would have more components; other beans could set up payroll and benefits records, schedule orientation, and so on.

Figure 36, "An Enterprise Bean Application: Client to Message-Driven Beans to Entity" assumes that `OfficeMDB` is the first message-driven bean to consume the message from the client. `OfficeMDB` then creates and persists the `SetupOffice` entity and stores the office information. `EquipmentMDB` then finds the entity, stores the equipment information, and learns that the entity has completed its work. `EquipmentMDB` then sends the message to the reply queue and removes the entity.

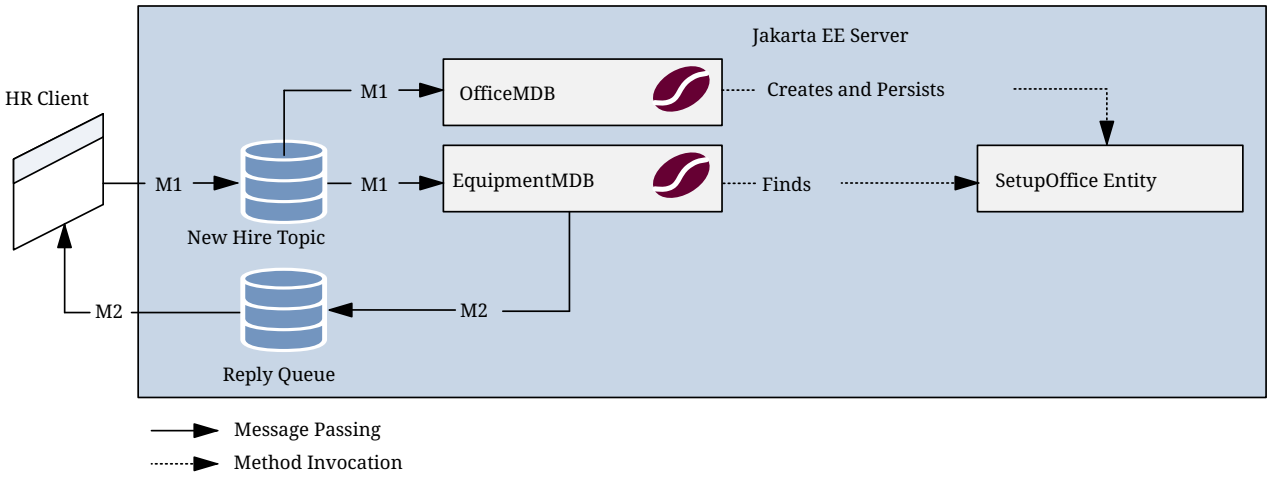


Figure 36. An Enterprise Bean Application: Client to Message-Driven Beans to Entity

Writing the Application Components for the clientmdbentity Example

Writing the components of the application involves coding the application client, the message-driven beans, and the entity class.

Coding the Application Client: HumanResourceClient.java

The application client, `HumanResourceClient.java`, found under `clientmdbentity-app-client`, performs the following steps:

1. Defines a topic for the application, using the `java:app` namespace because the topic is used in both the application client and the Jakarta Enterprise Beans module
2. Injects `ConnectionFactory` and `Topic` resources
3. Creates a `TemporaryQueue` to receive notification of processing that occurs, based on new-hire events it has published
4. Creates a `JMSConsumer` for the `TemporaryQueue`, sets the `JMSConsumer`'s message listener, and starts the connection
5. Creates a `MapMessage`
6. Creates five new employees with randomly generated names, positions, and ID numbers (in sequence) and publishes five messages containing this information

The message listener, `HRListener`, waits for messages that contain the assigned office and equipment for each employee. When a message arrives, the message listener displays the information received and determines whether all five messages have arrived. When they have, the message listener notifies the `main` method, which then exits.

Coding the Message-Driven Beans for the clientmdbentity Example

This example uses two message-driven beans, both under `clientmdbentity-ejb`:

- `EquipmentMDB.java`
- `OfficeMDB.java`

The beans take the following steps.

1. They inject a `MessageDrivenContext` resource, an `EntityManager`, and a `JMSContext`.
2. The `onMessage` method retrieves the information in the message. The `EquipmentMDB`'s `onMessage` method chooses equipment, based on the new hire's position; the `OfficeMDB`'s `onMessage` method randomly generates an office number.
3. After a slight delay to simulate real world processing hitches, the `onMessage` method calls a helper method, `compose`.
4. The `compose` method takes the following steps.
    a. It either creates and persists the `SetupOffice` entity or finds it by primary key.
    b. It uses the entity to store the equipment or the office information in the database, calling either the `doEquipmentList` or the `doOfficeNumber` business method.
    c. If the business method returns `true`, meaning that all of the information has been stored, it retrieves the reply destination information from the message, creates a `JMSProducer`, and sends a reply message that contains the information stored in the entity.
    d. It removes the entity.

Coding the Entity Class for the clientmdbentity Example

The `SetupOffice.java` class, also under `clientmdbentity-ejb`, is an entity class. The entity and the message-driven beans are packaged together in an enterprise bean JAR file. The entity class is declared as follows:

```
@Entity
public class SetupOffice implements Serializable { ... }
```

The class contains a no-argument constructor and a constructor that takes two arguments, the employee ID and name. It also contains getter and setter methods for the employee ID, name, office number, and equipment list. The getter method for the employee ID has the `@Id` annotation to indicate that this field is the primary key:

```
@Id
public String getEmployeeId() {
    return id;
}
```

The class also implements the two business methods, `doEquipmentList` and `doOfficeNumber`, and their helper method, `checkIfSetupComplete`.

The message-driven beans call the business methods and the getter methods.

The `persistence.xml` file for the entity specifies the most basic settings:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="3.0"
             xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
               https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="clientmdbentity-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:comp/DefaultDataSource</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Running the clientmdbentity Example

You can use either NetBeans IDE or Maven to build, deploy, and run the `clientmdbentity` example.

Because the example defines its own application-private topic and uses the preconfigured default connection factory `java:comp/DefaultJMSConnectionFactory` and the preconfigured default JDBC resource `java:comp/DefaultDataSource`, you do not need to create resources for it.

To Run clientmdbentity Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server), as well as the database server (see Starting and Stopping Apache Derby).
2. From the **File** menu, choose **Open Project**.
3. In the **Open Project** dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/jms/clientmdbentity
   ```

4. Select the `clientmdbentity` folder.
5. Click **Open Project**.
6. In the **Projects** tab, right-click the `clientmdbentity` project and select **Build**.

   This command creates the following:

   - An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
   - An enterprise bean JAR file that contains the message-driven beans and the entity class, along with the `persistence.xml` file
   - An application EAR file that contains the two JAR files along with an `application.xml` file

     The `clientmdbentity.ear` file is created in the `clientmdbentity-ear/target/` directory.

     The command then deploys the EAR file, retrieves the client stubs, and runs the application

client.

To Run clientmdbentity Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server), as well as the database server (see Starting and Stopping Apache Derby).
2. Go to the following directory:

   ```
   jakartaee-examples/tutorial/jms/clientmdbentity/
   ```

3. To compile the source files and package, deploy, and run the application, enter the following command:

   ```
   mvn install
   ```

   This command creates the following:

   - An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
   - An enterprise bean JAR file that contains the message-driven beans and the entity class, along with the `persistence.xml` file
   - An application EAR file that contains the two JAR files along with an `application.xml` file

     The command then deploys the application, retrieves the client stubs, and runs the application client.

Viewing the Application Output

The output in the NetBeans IDE output window or in the terminal window looks something like this (preceded by application client container output and Maven output):

```
SENDER: Setting hire ID to 50, name Bill Tudor, position Programmer
SENDER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
SENDER: Setting hire ID to 52, name Mark Wilson, position Manager
SENDER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
SENDER: Setting hire ID to 54, name Joe Lawrence, position Director
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 52
  Name: Mark Wilson
  Equipment: Tablet
  Office number: 294
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
```

```
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Mobile Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
  Name: Carol Jones
  Equipment: Laptop
  Office number: 262
```

The output from the message-driven beans and the entity class appears in the server log.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

To undeploy the application after you have finished running it, use the Services tab or issue the `mvn cargo:undeploy` command.

Using NetBeans IDE to Create Jakarta Messaging Resources

When you write your own Messaging applications, you will need to create resources for them. This section explains how to use NetBeans IDE to create `src/main/setup/glassfish-resources.xml` files similar to those used in the examples in this chapter. It also explains how to use NetBeans IDE to delete the resources.

You can also create, list, and delete Jakarta Messaging resources using the Administration Console or the `asadmin create-jms-resource`, `asadmin list-jms-resources`, and `asadmin delete-jms-resources` commands. For information, consult the GlassFish Server documentation or enter `asadmin help` command-name.

To Create Jakarta Messaging Resources Using NetBeans IDE

Follow these steps to create a Jakarta Messaging resource in GlassFish Server using NetBeans IDE. Repeat these steps for each resource you need.

1. Right-click the project for which you want to create resources and select **New**, then select **Other**.
2. In the **New File** wizard, under **Categories**, select **GlassFish**.
3. Under **File Types**, select **JMS Resource**.
4. On the General Attributes - JMS Resource page, in the JNDI Name field, enter the name of the resource.

   By convention, Messaging resource names begin with `jms/`.

5. Select the option for the resource type.

   Normally, this is either `jakarta.jms.Queue`, `jakarta.jms.Topic`, or `jakarta.jms.ConnectionFactory`.

6. Click **Next**.
7. On the JMS Properties page, for a queue or topic, enter a name for a physical queue in the Value field for the Name property.

   You can enter any value for this required field.

   Connection factories have no required properties. In a few situations, you may need to specify a property.

8. Click **Finish**.

   A file named `glassfish-resources.xml` is created in your Maven project, in a directory named `src/main/setup/`. In the Projects tab, you can find it under the Other Sources node. You will need to run the `asadmin add-resources` command to create the resources in GlassFish Server.

To Delete Jakarta Messaging Resources Using NetBeans IDE

1. In the **Services** tab, expand the **Servers** node, then expand the **GlassFish Server** node.
2. Expand the **Resources** node, then expand the **Connector Resources** node.
3. Expand the **Admin Object Resources** node.
4. Right-click any destination you want to remove and select **Unregister**.
5. Expand the **Connector Connection Pools** node.
6. Right-click the connection pool that corresponds to the connection factory you removed and select **Unregister**.

   When you remove a connector connection pool, the associated connector resource is also deleted. This action removes the connection factory.

Jakarta Batch



This section was last updated for Jakarta EE Version 9.1

This chapter describes Jakarta Batch, which provides support for defining, implementing, and

running batch jobs. Batch jobs are tasks that can be executed without user interaction. The batch framework is composed of a job specification language based on XML, a Java API, and a batch runtime.

Introduction to Batch Processing

Some enterprise applications contain tasks that can be executed without user interaction. These tasks are executed periodically or when resource usage is low, and they often process large amounts of information such as log files, database records, or images. Examples include billing, report generation, data format conversion, and image processing. These tasks are called batch jobs.

Batch processing refers to running batch jobs on a computer system. Jakarta EE includes a batch processing framework that provides the batch execution infrastructure common to all batch applications, enabling developers to concentrate on the business logic of their batch applications. The batch framework consists of a job specification language based on XML, a set of batch annotations and interfaces for application classes that implement the business logic, a batch container that manages the execution of batch jobs, and supporting classes and interfaces to interact with the batch container.

A batch job can be completed without user intervention. For example, consider a telephone billing application that reads phone call records from the enterprise information systems and generates a monthly bill for each account. Since this application does not require any user interaction, it can run as a batch job.

The phone billing application consists of two phases: The first phase associates each call from the registry with a monthly bill, and the second phase calculates the tax and total amount due for each bill. Each of these phases is a step of the batch job.

Batch applications specify a set of steps and their execution order. Different batch frameworks may specify additional elements, like decision elements or groups of steps that run in parallel. The following sections describe steps in more detail and provide information about other common characteristics of batch frameworks.

Steps in Batch Jobs

A step is an independent and sequential phase of a batch job. Batch jobs contain chunk-oriented steps and task-oriented steps.

- Chunk-oriented steps (chunk steps) process data by reading items from a data source, applying some business logic to each item, and storing the results. Chunk steps read and process one item at a time and group the results into a chunk. The results are stored when the chunk reaches a configurable size. Chunk-oriented processing makes storing results more efficient and facilitates transaction demarcation.

  Chunk steps have three parts.

  - The input retrieval part reads one item at a time from a data source, such as entries on a database, files in a directory, or entries in a log file.
  - The business processing part manipulates one item at a time using the business logic

defined by the application. Examples include filtering, formatting, and accessing data from the item for computing a result.

- The output writing part stores a chunk of processed items at a time.

Chunk steps are often long-running because they process large amounts of data. Batch frameworks enable chunk steps to bookmark their progress using checkpoints. A chunk step that is interrupted can be restarted from the last checkpoint. The input retrieval and output writing parts of a chunk step save their current position after the processing of each chunk, and can recover it when the step is restarted.

Figure 37, "Chunk Steps in a Batch Job" shows the three parts of two steps in a batch job.

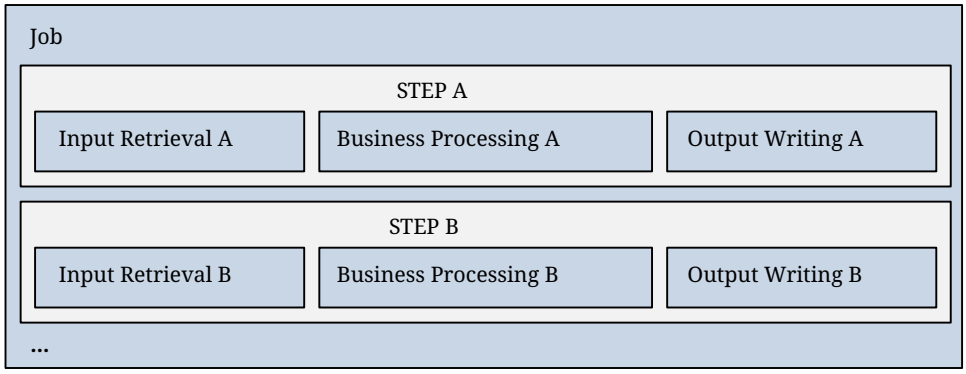


Figure 37. Chunk Steps in a Batch Job

For example, the phone billing application consists of two chunk steps.

- In the first step, the input retrieval part reads call records from the registry; the business processing part associates each call with a bill and creates a bill if one does not exist for an account; and the output writing part stores each bill in a database.
- In the second step, the input retrieval part reads bills from the database; the business processing part calculates the tax and total amount due for each bill; and the output writing part updates the database records and generates printable versions of each bill.

This application could also contain a task step that cleaned up the files from the bills generated for the previous month.

Parallel Processing

Batch jobs often process large amounts of data or perform computationally expensive operations. Batch applications can benefit from parallel processing in two scenarios.

- Steps that do not depend on each other can run on different threads.
- Chunk-oriented steps where the processing of each item does not depend on the results of processing previous items can run on more than one thread.

Batch frameworks provide mechanisms for developers to define groups of independent steps and to split chunk-oriented steps in parts that can run in parallel.

Status and Decision Elements

Batch frameworks keep track of a status for every step in a job. The status indicates if a step is running or if it has completed. If the step has completed, the status indicates one of the following.

- The execution of the step was successful.
- The step was interrupted.
- An error occurred in the execution of the step.

In addition to steps, batch jobs can also contain decision elements. Decision elements use the exit status of the previous step to determine the next step or to terminate the batch job. Decision elements set the status of the batch job when terminating it. Like a step, a batch job can terminate successfully, be interrupted, or fail.

Figure 38, “Steps and Decision Elements in a Job” shows an example of a job that contains chunk steps, task steps and a decision element.

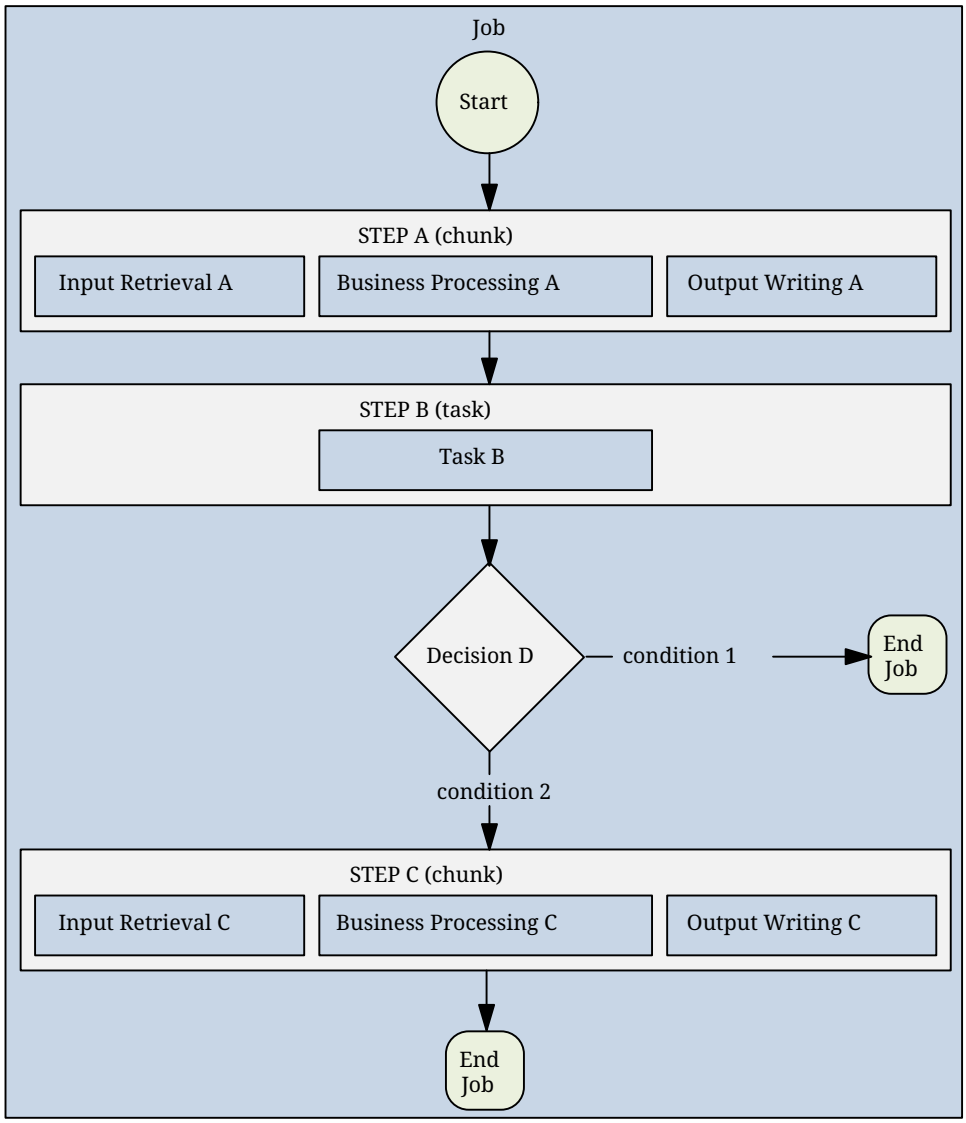


Figure 38. Steps and Decision Elements in a Job

Batch Framework Functionality

Batch applications have the following common requirements.

- Define jobs, steps, decision elements, and the relationships between them.
- Execute some groups of steps or parts of a step in parallel.
- Maintain state information for jobs and steps.
- Launch jobs and resume interrupted jobs.
- Handle errors.

Batch frameworks provide the batch execution infrastructure that addresses the common requirements of all batch applications, enabling developers to concentrate on the business logic of their applications. Batch frameworks consist of a format to specify jobs and steps, an application programming interface (API), and a service available at runtime that manages the execution of batch jobs.

Batch Processing in Jakarta EE

This section lists the components of the batch processing framework in Jakarta EE and provides an overview of the steps you have to follow to create a batch application.

The Batch Processing Framework

Jakarta EE includes a batch processing framework that consists of the following elements:

- A batch runtime that manages the execution of jobs
- A job specification language based on XML
- A Java API to interact with the batch runtime
- A Java API to implement steps, decision elements, and other batch artifacts

Batch applications in Jakarta EE contain XML files and Java classes. The XML files define the structure of a job in terms of batch artifacts and the relationships between them. (A batch artifact is a part of a chunk-oriented step, a task-oriented step, a decision element, or another component of a batch application). The Java classes implement the application logic of the batch artifacts defined in the XML files. The batch runtime parses the XML files and loads the batch artifacts as Java classes to run the jobs in a batch application.

Creating Batch Applications

The process for creating a batch application in Jakarta EE is the following.

1. Design the batch application.
    a. Identify the input sources, the format of the input data, the desired final result, and the required processing phases.
    b. Organize the application as a job with chunk-oriented steps, task-oriented steps, and decision elements. Determine the dependencies between them.

c. Determine the order of execution in terms of transitions between steps.

d. Identify steps that can run in parallel and steps that can run in more than one thread.

2. Create the batch artifacts as Java classes by implementing the interfaces specified by the framework for steps, decision elements, and so on. These Java classes contain the code to read data from input sources, format items, process items, and store results. Batch artifacts can access context objects from the batch runtime using dependency injection.
3. Define jobs, steps, and their execution flow in XML files using the Job Specification Language. The elements in the XML files reference batch artifacts implemented as Java classes. The batch artifacts can access properties declared in the XML files, such as names of files and databases.
4. Use the Java API provided by the batch runtime to launch the batch application.

The following sections describe in detail how to use the components of the batch processing framework in Jakarta EE to create batch applications.

Elements of a Batch Job

A batch job can contain one or more of the following elements:

- Steps
- Flows
- Splits
- Decision elements

Steps are described in Introduction to Batch Processing, and can be chunk-oriented or task-oriented. Chunk-oriented steps can be partitioned steps. In a partitioned chunk step, the processing of one item does not depend on other items, so these steps can run in more than one thread.

A flow is a sequence of steps that execute as a unit. A sequence of related steps can be grouped together into a flow. The steps in a flow cannot transition to steps outside the flow. The flow transitions to the next element when its last step completes.

A split is a set of flows that execute in parallel; each flow runs on a separate thread. The split transitions to the next element when all its flows complete.

Decision elements use the exit status of the previous step to determine the next step or to terminate the batch job.

Properties and Parameters

Jobs and steps can have a number of properties associated with them. You define properties in the job definition file, and batch artifacts access these properties using context objects from the batch runtime. Using properties in this manner enables you to decouple static parameters of the job from the business logic and to reuse batch artifacts in different job definition files.

Specifying properties is described in Using the Job Specification Language, and accessing properties in batch artifacts is described in Creating Batch Artifacts.

Jakarta EE applications can also pass parameters to a job when they submit it to the batch runtime.

This enables you to specify dynamic parameters that are only known at runtime. Parameters are also necessary for partitioned steps, since each partition needs to know, for example, what range of items to process.

Specifying parameters when submitting jobs is described in Submitting Jobs to the Batch Runtime. Specifying parameters for partitioned steps and accessing them in batch artifacts is demonstrated in The phonebilling Example Application.

Job Instances and Job Executions

A job definition can have multiple instances, each with different parameters. A job execution is an attempt to run a job instance. The batch runtime maintains information about job instances and job executions, as described in Checking the Status of a Job.

Batch and Exit Status

The state of jobs, steps, splits, and flows is represented in the batch runtime as a batch status value. Batch status values are listed Batch Status Values. They are represented as strings.

Batch Status Values

Value	Description
`STARTING`	The job has been submitted to the batch runtime.
`STARTED`	The job is running.
`STOPPING`	The job has been requested to stop.
`STOPPED`	The job has stopped.
`FAILED`	The job finished executing because of an error.
`COMPLETED`	The job finished executing successfully.
`ABANDONED`	The job was marked abandoned.

Jakarta EE applications can submit jobs and access the batch status of a job using the `JobOperator` interface, as described in Submitting Jobs to the Batch Runtime. Job definition files can refer to batch status values using the Job Specification Language (JSL), as described in Using the Job Specification Language. Batch artifacts can access batch status values using context objects, as described in Using the Context Objects from the Batch Runtime.

For flows, the batch status is that of its last step. For splits, the batch status is the following:

- `COMPLETED`: If all its flows have a batch status of `COMPLETED`
- `FAILED`: If any flow has a batch status of `FAILED`
- `STOPPED`: If any flow has a batch status of `STOPPED`, and no flows have a batch status of `FAILED`

The batch status for jobs, steps, splits, and flows is set by the batch runtime. Jobs, steps, splits, and

flows also have an exit status, which is a user-defined value based on the batch status. You can set the exit status inside batch artifacts or in the job definition file. You can access the exit status in the same manner as the batch status, described above. The default value for the exit status is the same as the batch status.

Simple Use Case

This section demonstrates how to define a simple job using the Job Specification Language (JSL) and how to implement the corresponding batch artifacts. Refer to the rest of the sections in this chapter for detailed descriptions of the elements in the batch framework.

The following job definition specifies a chunk step and a task step as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="simplejob" xmlns="https://jakarta.ee/xml/ns/jakartaee"
                    version="2.0">
  <properties>
    <property name="input_file" value="input.txt"/>
    <property name="output_file" value="output.txt"/>
  </properties>
  <step id="mychunk" next="mytask">
    <chunk>
      <reader ref="MyReader"></reader>
      <processor ref="MyProcessor"></processor>
      <writer ref="MyWriter"></writer>
    </chunk>
  </step>
  <step id="mytask">
    <batchlet ref="MyBatchlet"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

Chunk Step

In most cases, you have to implement a checkpoint class for chunk-oriented steps. The following class just keeps track of the line number in a text file:

```
public class MyCheckpoint implements Serializable {
    private long lineNum = 0;
    public void increase() { lineNum++; }
    public long getLineNum() { return lineNum; }
}
```

The following item reader implementation continues reading the input file from the provided checkpoint if the job was restarted. The items consist of each line in the text file (in more complex scenarios, the items are custom Java types and the input source can be a database):

```
@Dependent
@Named("MyReader")
public class MyReader implements jakarta.batch.api.chunk.ItemReader {
    private MyCheckpoint checkpoint;
    private BufferedReader breader;
    @Inject
    JobContext jobCtx;

    public MyReader() {}

    @Override
    public void open(Serializable ckpt) throws Exception {
        if (ckpt == null)
            checkpoint = new MyCheckpoint();
        else
            checkpoint = (MyCheckpoint) ckpt;
        String fileName = jobCtx.getProperties()
                                .getProperty("input_file");
        breader = new BufferedReader(new FileReader(fileName));
        for (long i = 0; i < checkpoint.getLineNum(); i++)
            breader.readLine();
    }

    @Override
    public void close() throws Exception {
        breader.close();
    }

    @Override
    public Object readItem() throws Exception {
        String line = breader.readLine();
        return line;
    }
}
```

In the following case, the item processor only converts the line to uppercase. More complex examples can process items in different ways or transform them into custom output Java types:

```
@Dependent
@Named("MyProcessor")
public class MyProcessor implements jakarta.batch.api.chunk.ItemProcessor {
    public MyProcessor() {}

    @Override
    public Object processItem(Object obj) throws Exception {
        String line = (String) obj;
        return line.toUpperCase();
    }
```

```
}
```



The batch processing API does not support generics. In most cases, you need to cast items to their specific type before processing them.

The item writer writes the processed items to the output file. It overwrites the output file if no checkpoint is provided; otherwise, it resumes writing at the end of the file. Items are written in chunks:

```
@Dependent
@Named("MyWriter")
public class MyWriter implements jakarta.batch.api.chunk.ItemWriter {
    private BufferedWriter bwriter;
    @Inject
    private JobContext jobCtx;

    @Override
    public void open(Serializable ckpt) throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("output_file");
        bwriter = new BufferedWriter(new FileWriter(fileName,
                                                    (ckpt != null)));
    }

    @Override
    public void writeItems(List<Object> items) throws Exception {
        for (int i = 0; i < items.size(); i++) {
            String line = (String) items.get(i);
            bwriter.write(line);
            bwriter.newLine();
        }
    }

    @Override
    public Serializable checkpointInfo() throws Exception {
        return new MyCheckpoint();
    }
}
```

Task Step

The task step displays the length of the output file. In more complex scenarios, task steps perform any task that does not fit the chunk processing programming model:

```
@Dependent
@Named("MyBatchlet")
public class MyBatchlet implements jakarta.batch.api.chunk.Batchlet {
    @Inject
```

```
    private JobContext jobCtx;

    @Override
    public String process() throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("output_file");
        System.out.println(""+(new File(fileName)).length());
        return "COMPLETED";
    }
}
```

Using the Job Specification Language

The Job Specification Language (JSL) enables you to define the steps in a job and their execution order using an XML file. The following example shows how to define a simple job that contains one chunk step and one task step:

```
<job id="loganalysis" xmlns="https://jakarta.ee/xml/ns/jakartaee"
                      version="2.0">
  <properties>
    <property name="input_file" value="input1.txt"/>
    <property name="output_file" value="output2.txt"/>
  </properties>

  <step id="logprocessor" next="cleanup">
    <chunk checkpoint-policy="item" item-count="10">
      <reader ref="com.example.pkg.LogItemReader"></reader>
      <processor ref="com.example.pkg.LogItemProcessor"></processor>
      <writer ref="com.example.pkg.LogItemWriter"></writer>
    </chunk>
  </step>

  <step id="cleanup">
    <batchlet ref="com.example.pkg.CleanUp"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

This example defines the `loganalysis` batch job, which consists of the `logprocessor` chunk step and the `cleanup` task step. The `logprocessor` step transitions to the `cleanup` step, which terminates the job when completed.

The `job` element defines two properties, `input_file` and `output_file`. Specifying properties in this manner enables you to run a batch job with different configuration parameters without having to recompile its Java batch artifacts. The batch artifacts can access these properties using the context objects from the batch runtime.

The `logprocessor` step is a chunk step that specifies batch artifacts for the reader (`LogItemReader`), the processor (`LogItemProcessor`), and the writer (`LogItemWriter`). This step creates a checkpoint for

every ten items processed.

The `cleanup` step is a task step that specifies the `CleanUp` class as its batch artifact. The job terminates when this step completes.

The following sections describe the elements of the Job Specification Language (JSL) in more detail and show the most common attributes and child elements.

The job Element

The `job` element is always the top-level element in a job definition file. Its main attributes are `id` and `restartable`. The `job` element can contain one `properties` element and zero or more of each of the following elements: `listener`, `step`, `flow`, and `split`. For example:

```
<job id="jobname" restartable="true">
  <listeners>
    <listener ref="com.example.pkg.ListenerBatchArtifact"/>
  </listeners>
  <properties>
    <property name="propertyName1" value="propertyValue1"/>
    <property name="propertyName2" value="propertyValue2"/>
  </properties>
  <step ...> ... </step>
  <step ...> ... </step>
  <decision ...> ... </decision>
  <flow ...> ... </flow>
  <split ...> ... </split>
</job>
```

The `listener` element specifies a batch artifact whose methods are invoked before and after the execution of the job. The batch artifact is an implementation of the `jakarta.batch.api.listener.JobListener` interface. See The Listener Batch Artifacts for an example of a job listener implementation.

The first `step`, `flow`, or `split` element inside the `job` element executes first.

The step Element

The `step` element can be a child of the `job` and `flow` elements. Its main attributes are `id` and `next`. The `step` element can contain the following elements.

- One `chunk` element for chunk-oriented steps or one `batchlet` element for task-oriented steps.
- One `properties` element (optional).

  This element specifies a set of properties that batch artifacts can access using batch context objects.

- One `listener` element (optional); one `listeners` element if more than one listener is specified.

  This element specifies listener artifacts that intercept various phases of step execution.

For chunk steps, the batch artifacts for these listeners can be implementations of the following interfaces: `StepListener`, `ItemReadListener`, `ItemProcessListener`, `ItemWriteListener`, `ChunkListener`, `RetryReadListener`, `RetryProcessListener`, `RetryWriteListener`, `SkipReadListener`, `SkipProcessListener`, and `SkipWriteListener`.

For task steps, the batch artifact for these listeners must be an implementation of the `StepListener` interface.

See The Listener Batch Artifacts for an example of an item processor listener implementation.

- One `partition` element (optional).

  This element is used in partitioned steps which execute in more than one thread.

- One `end` element if this is the last step in a job.

  This element sets the batch status to `COMPLETED`.

- One `stop` element (optional) to stop a job at this step.

  This element sets the batch status to `STOPPED`.

- One `fail` element (optional) to terminate a job at this step.

  This element sets the batch status to `FAILED`.

- One or more `next` elements if the `next` attribute is not specified.

  This element is associated with an exit status and refers to another step, a flow, a split, or a decision element.

The following is an example of a chunk step:

```
<step id="stepA" next="stepB">
  <properties> ... </properties>
  <listeners>
    <listener ref="MyItemReadListenerImpl"/>
    ...
  </listeners>
  <chunk ...> ... </chunk>
  <partition> ... </partition>
  <end on="COMPLETED" exit-status="MY_COMPLETED_EXIT_STATUS"/>
  <stop on="MY_TEMP_ISSUE_EXIST_STATUS" restart="step0"/>
  <fail on="MY_ERROR_EXIT_STATUS" exit-status="MY_ERROR_EXIT_STATUS"/>
</step>
```

The following is an example of a task step:

```
<step id="stepB" next="stepC">
  <batchlet ...> ... </batchlet>
```

```
  <properties> ... </properties>
  <listener ref="MyStepListenerImpl"/>
</step>
```

The chunk Element

The `chunk` element is a child of the `step` element for chunk-oriented steps. The attributes of this element are listed in Attributes of the chunk Element.

Attributes of the chunk Element

Attribute Name	Description	Default Value
`checkpoint-policy`	Specifies how to commit the results of processing each chunk: • `"item"`: the chunk is committed after processing `item-count` items • `"custom"`: the chunk is committed according to a checkpoint algorithm specified with the `checkpoint-algorithm` element The checkpoint is updated when the results of a chunk are committed. Every chunk is processed in a global Jakarta EE transaction. If the processing of one item in the chunk fails, the transaction is rolled back and no processed items from this chunk are stored.	`"item"`
`item-count`	Specifies the number of items to process before committing the chunk and taking a checkpoint.	10
`time-limit`	Specifies the number of seconds before committing the chunk and taking a checkpoint when `checkpoint-policy="item"`. If `item-count` items have not been processed by `time-limit` seconds, the chunk is committed and a checkpoint is taken.	0 (no limit)
`buffer-items`	Specifies if processed items are buffered until it is time to take a checkpoint. If true, a single call to the item writer is made with a list of the buffered items before committing the chunk and taking a checkpoint.	true
`skip-limit`	Specifies the number of skippable exceptions to skip in this step during chunk processing. Skippable exception classes are specified with the `skippable-exception-classes` element.	No limit
`retry-limit`	Specifies the number of attempts to execute this step if retryable exceptions occur. Retryable exception classes are specified with the `retryable-exception-classes` element.	No limit

The `chunk` element can contain the following elements.

- One `reader` element.

  This element specifies a batch artifact that implements the `ItemReader` interface.

- One `processor` element.

  This element specifies a batch artifact that implements the `ItemProcessor` interface.

- One `writer` element.

  This element specifies a batch artifact that implements the `ItemWriter` interface.

- One `checkpoint-algorithm` element (optional).

  This element specifies a batch artifact that implements the `CheckpointAlgorithm` interface and provides a custom checkpoint policy.

- One `skippable-exception-classes` element (optional).

  This element specifies a set of exceptions thrown from the reader, writer, and processor batch artifacts that chunk processing should skip. The `skip-limit` attribute from the `chunk` element specifies the maximum number of skipped exceptions.

- One `retryable-exception-classes` element (optional).

  This element specifies a set of exceptions thrown from the reader, writer, and processor batch artifacts that chunk processing will retry. The `retry-limit` attribute from the `chunk` element specifies the maximum number of attempts.

- One `no-rollback-exception-classes` element (optional).

  This element specifies a set of exceptions thrown from the reader, writer, and processor batch artifacts that should not cause the batch runtime to roll back the current chunk, but to retry the current operation without a rollback instead.

  For exception types not specified in this element, the current chunk is rolled back by default when an exception occurs.

The following is an example of a chunk-oriented step:

```
<step id="stepC" next="stepD">
  <chunk checkpoint-policy="item" item-count="5" time-limit="180"
         buffer-items="true" skip-limit="10" retry-limit="3">
    <reader ref="pkg.MyItemReaderImpl"></reader>
    <processor ref="pkg.MyItemProcessorImpl"></processor>
    <writer ref="pkg.MyItemWriterImpl"></writer>
    <skippable-exception-classes>
      <include class="pkg.MyItemException"/>
      <exclude class="pkg.MyItemSeriousSubException"/>
    </skippable-exception-classes>
    <retryable-exception-classes>
      <include class="pkg.MyResourceTempUnavailable"/>
    </retryable-exception-classes>
  </chunk>
</step>
```

This example defines a chunk step and specifies its reader, processor, and writer artifacts. The step updates a checkpoint and commits each chunk after processing five items. It skips all `MyItemException` exceptions and all its subtypes, except for `MyItemSeriousSubException`, up to a maximum of ten skipped exceptions. The step retries a chunk when a `MyResourceTempUnavailable` exception occurs, up to a maximum of three attempts.

The batchlet Element

The `batchlet` element is a child of the `step` element for task-oriented steps. This element only has the `ref` attribute, which specifies a batch artifact that implements the `Batchlet` interface. The `batch` element can contain a `properties` element.

The following is an example of a task-oriented step:

```
<step id="stepD" next="stepE">
  <batchlet ref="pkg.MyBatchletImpl">
    <properties>
      <property name="pname" value="pvalue"/>
    </properties>
  </batchlet>
</step>
```

This example defines a batch step and specifies its batch artifact.

The partition Element

The `partition` element is a child of the `step` element. It indicates that a step is partitioned. Most partitioned steps are chunk steps where the processing of each item does not depend on the results of processing previous items. You specify the number of partitions in a step and provide each partition with specific information on which items to process, such as the following.

- A range of items. For example, partition 1 processes items 1 through 500, and partition 2 processes items 501 through 1000.
- An input source. For example, partition 1 processes the items in `input1.txt` and partition 2 processes the items in `input2.txt`.

When the number of partitions, the number of items, and the input sources for a partitioned step are known at development or deployment time, you can use partition properties in the job definition file to specify partition-specific information and access these properties from the step batch artifacts. The runtime creates as many instances of the step batch artifacts (reader, processor, and writer) as partitions, and each artifact instance receives the properties specific to its partition.

In most cases, the number of partitions, the number of items, or the input sources for a partitioned step can only be determined at runtime. Instead of specifying partition-specific properties statically in the job definition file, you provide a batch artifact that can access your data sources at runtime and determine how many partitions are needed and what range of items each partition should process. This batch artifact is an implementation of the `PartitionMapper` interface. The batch runtime invokes this artifact and then uses the information it provides to instantiate the step batch artifacts (reader, writer, and processor) for each partition and to pass them partition-specific data

as parameters.

The rest of this section describes the `partition` element in detail and shows two examples of job definition files: one that uses partition properties to specify a range of items for each partition, and one that relies on a `PartitionMapper` implementation to determine partition-specific information.

See The Phone Billing Chunk Step in The phonebilling Example Application for a complete example of a partitioned chunk step.

The `partition` element can contain the following elements.

- One `plan` element, if the `mapper` element is not specified.

  This element defines the number of partitions, the number of threads, and the properties for each partition in the job definition file. The `plan` element is useful when this information is known at development or deployment time.

- One `mapper` element, if the `plan` element is not specified.

  This element specifies a batch artifact that provides the number of partitions, the number of threads, and the properties for each partition. The batch artifact is an implementation of the `PartitionMapper` interface. You use this option when the information required for each partition is only known at runtime.

- One `reducer` element (optional).

  This element specifies a batch artifact that receives control when a partitioned step begins, ends, or rolls back. The batch artifact enables you to merge results from different partitions and perform other related operations. The batch artifact is an implementation of the `PartitionReducer` interface.

- One `collector` element (optional).

  This element specifies a batch artifact that sends intermediary results from each partition to a partition analyzer. The batch artifact sends the intermediary results after each checkpoint for chunk steps and at the end of the step for task steps. The batch artifact is an implementation of the `PartitionCollector` interface.

- One `analyzer` element (optional).

  This element specifies a batch artifact that analyzes the intermediary results from the partition collector instances. The batch artifact is an implementation of the `PartitionAnalyzer` interface.

The following is an example of a partitioned step using the `plan` element:

```
<step id="stepE" next="stepF">
  <chunk>
    <reader ...></reader>
    <processor ...></processor>
    <writer ...></writer>
  </chunk>
```

```
  <partition>
    <plan partitions="2" threads="2">
      <properties partition="0">
        <property name="firstItem" value="0"/>
        <property name="lastItem" value="500"/>
      </properties>
      <properties partition="1">
        <property name="firstItem" value="501"/>
        <property name="lastItem" value="999"/>
      </properties>
    </plan>
  </partition>
  <reducer ref="MyPartitionReducerImpl"/>
  <collector ref="MyPartitionCollectorImpl"/>
  <analyzer ref="MyPartitionAnalyzerImpl"/>
</step>
```

In this example, the `plan` element specifies the properties for each partition in the job definition file.

The following example uses a `mapper` element instead of a `plan` element. The `PartitionMapper` implementation dynamically provides the same information as the `plan` element provides in the job definition file:

```
<step id="stepE" next="stepF">
  <chunk>
    <reader ...></reader>
    <processor ...></processor>
    <writer ...></writer>
  </chunk>
  <partition>
    <mapper ref="MyPartitionMapperImpl"/>
    <reducer ref="MyPartitionReducerImpl"/>
    <collector ref="MyPartitionCollectorImpl"/>
    <analyzer ref="MyPartitionAnalyzerImpl"/>
  </partition>
</step>
```

Refer to The phonebilling Example Application for an example implementation of the `PartitionMapper` interface.

The flow Element

The `flow` element can be a child of the `job`, `flow`, and `split` elements. Its attributes are `id` and `next`. Flows can transition to flows, steps, splits, and decision elements. The `flow` element can contain the following elements:

- One or more `step` elements
- One or more `flow` elements (optional)

- One or more `split` elements (optional)
- One or more `decision` elements (optional)

The last `step` in a flow is the one with no `next` attribute or `next` element. Steps and other elements in a flow cannot transition to elements outside the flow.

The following is an example of the `flow` element:

```
<flow id="flowA" next="stepE">
  <step id="flowAstepA" next="flowAstepB">...</step>
  <step id="flowAstepB" next="flowAflowC">...</step>
  <flow id="flowAflowC" next="flowAsplitD">...</flow>
  <split id="flowAsplitD" next="flowAstepE">...</split>
  <step id="flowAstepE">...</step>
</flow>
```

This example flow contains three steps, one flow, and one split. The last step does not have the `next` attribute. The flow transitions to `stepE` when its last step completes.

The split Element

The `split` element can be a child of the `job` and `flow` elements. Its attributes are `id` and `next`. Splits can transition to splits, steps, flows, and decision elements. The `split` element can only contain one or more `flow` elements that can only transition to other `flow` elements in the split.

The following is an example of a split with three flows that execute concurrently:

```
<split id="splitA" next="stepB">
  <flow id="splitAflowA">...</flow>
  <flow id="splitAflowB">...</flow>
  <flow id="splitAflowC">...</flow>
</split>
```

The decision Element

The `decision` element can be a child of the `job` and `flow` elements. Its attributes are `id` and `next`. Steps, flows, and splits can transition to a `decision` element. This element specifies a batch artifact that decides the next step, flow, or split to execute based on information from the execution of the previous step, flow, or split. The batch artifact implements the `Decider` interface. The `decision` element can contain the following elements.

- One or more `end` elements (optional).

  This element sets the batch status to `COMPLETED`.

- One or more `stop` elements (optional).

  This element sets the batch status to `STOPPED`.

- One or more `fail` elements (optional).

  This element sets the batch status to `FAILED`.

- One or more `next` elements (optional).
- One `properties` element (optional).

The following is an example of the `decider` element:

```
<decision id="decisionA" ref="MyDeciderImpl">
  <fail on="FAILED" exit-status="FAILED_AT_DECIDER"/>
  <end on="COMPLETED" exit-status="COMPLETED_AT_DECIDER"/>
  <stop on="MY_TEMP_ISSUE_EXIST_STATUS" restart="step2"/>
</decision>
```

Creating Batch Artifacts

After you define a job in terms of its batch artifacts using the Job Specification Language (JSL), you create these artifacts as Java classes that implement the interfaces in the `jakarta.batch.api` package and its subpackages.

This section lists the main batch artifact interfaces, demonstrates how to access context objects from the batch runtime, and provides some examples.

Batch Artifact Interfaces

The following tables list the interfaces that you implement to create batch artifacts. The interface implementations are referenced from the elements described in Using the Job Specification Language.

Main Batch Artifact Interfaces lists the interfaces to implement batch artifacts for chunk steps, task steps, and decision elements.

Partition Batch Artifact Interfaces lists the interfaces to implement batch artifacts for partitioned steps.

Listener Batch Artifact Interfaces lists the interfaces to implement batch artifacts for job and step listeners.

Main Batch Artifact Interfaces

Package	Interface	Description
`jakarta.batch.api`	`Batchlet`	Implements the business logic of a task-oriented step. It is referenced from the `batchlet` element.
`jakarta.batch.api`	`Decider`	Decides the next step, flow, or split to execute based on information from the execution of the previous step, flow, or split. It is referenced from the `decision` element.

Package	Interface	Description
jakarta.batch.api.chunk	CheckPointAlgorithm	Implements a custom checkpoint policy for chunk steps. It is referenced from the checkpoint-algorithm element inside the chunk element.
jakarta.batch.api.chunk	ItemReader	Reads items from an input source in a chunk step. It is referenced from the reader element inside the chunk element.
jakarta.batch.api.chunk	ItemProcessor	Processes input items to obtain output items in chunk steps. It is referenced from the processor element inside the chunk element.
jakarta.batch.api.chunk	ItemWriter	Writes output items in chunk steps. It is referenced from the writer element inside the chunk element.

Partition Batch Artifact Interfaces

Package	Interface	Description
jakarta.batch.api.partition	PartitionPlan	Provides details on how to execute a partitioned step, such as the number of partitions, the number of threads, and the parameters for each partition. This artifact is not referenced directly from the job definition file.
jakarta.batch.api.partition	PartitionMapper	Provides a PartitionPlan object. It is referenced from the mapper element inside the partition element.
jakarta.batch.api.partition	PartitionReducer	Receives control when a partitioned step begins, ends, or rolls back. It is referenced from the reducer element inside the partition element.
jakarta.batch.api.partition	PartitionCollector	Sends intermediary results from each partition to a partition analyzer. It is referenced from the collector element inside the partition element.
jakarta.batch.api.partition	PartitionAnalyzer	Processes data and final results from each partition. It is referenced from the analyzer element inside the partition element.

Listener Batch Artifact Interfaces

Package	Interface	Description
jakarta.batch.api.listener	JobListener	Intercepts job execution before and after running a job. It is referenced from the listener element inside the job element.
jakarta.batch.api.listener	StepListener	Intercepts step execution before and after running a step. It is referenced from the listener element inside the step element
jakarta.batch.api.chunk.listener	ChunkListener	Intercepts chunk processing in chunk steps before and after processing each chunk, and on errors. It is referenced from the listener element inside the step element.
jakarta.batch.api.chunk.listener	ItemReadListener	Intercepts item reading in chunk steps before and after reading each item, and on errors. It is referenced from the listener element inside the step element.

Package	Interface	Description
jakarta.batch.api.chunk.listener	ItemProcessListener	Intercepts item processing in chunk steps before and after processing each item, and on errors. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	ItemWriteListener	Intercepts item writing in chunk steps before and after writing each item, and on errors. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	RetryReadListener	Intercepts retry item reading in chunk steps when an exception occurs. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	RetryProcessListener	Intercepts retry item processing in chunk steps when an exception occurs. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	RetryWriteListener	Intercepts retry item writing in chunk steps when an exception occurs. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	SkipReadListener	Intercepts skippable exception handling for item readers in chunk steps. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	SkipProcessListener	Intercepts skippable exception handling for item processors in chunk steps. It is referenced from the `listener` element inside the `step` element.
jakarta.batch.api.chunk.listener	SkipWriteListener	Intercepts skippable exception handling for item writers in chunk steps. It is referenced from the `listener` element inside the `step` element.

Dependency Injection in Batch Artifacts

To ensure that Jakarta Contexts and Dependency Injection (CDI) works in your batch artifacts, follow these steps.

1. Define your batch artifact implementations as CDI named beans using the `Named` annotation.

   For example, define an item reader implementation in a chunk step as follows:

   ```
   @Named("MyItemReaderImpl")
   public class MyItemReaderImpl implements ItemReader {
       /* ... Override the ItemReader interface methods ... */
   }
   ```

2. Provide a public, empty, no-argument constructor for your batch artifacts.

   For example, provide the following constructor for the artifact above:

```
public MyItemReaderImpl() {}
```

3. Specify the CDI name for the batch artifacts in the job definition file, instead of using the fully qualified name of the class.

   For example, define the step for the artifact above as follows:

   ```
   <step id="stepA" next="stepB">
     <chunk>
       <reader ref="MyItemReaderImpl"></reader>
       ...
     </chunk>
   </step>
   ```

   This example uses the CDI name (`MyItemReaderImpl`) instead of the fully qualified name of the class (`com.example.pkg.MyItemReaderImpl`) to specify a batch artifact.

4. Ensure that your module is a CDI bean archive by annotating your batch artifacts with the `jakarta.enterprise.context.Dependent` annotation or by including an empty `beans.xml` deployment description with your application. For example, the following batch artifact is annotated with `@Dependent`:

   ```
   @Dependent
   @Named("MyItemReaderImpl")
   public class MyItemReaderImpl implements ItemReader { ... }
   ```

   For more information on bean archives, see Packaging CDI Applications in [cdi:cdi-adv::cdi-adv:::_jakarta_contexts_and_dependency_injection_advanced_topics].



Jakarta Contexts and Dependency Injection (CDI) is required in order to access context objects from the batch runtime in batch artifacts.

You may encounter the following errors if you do not follow this procedure.

- The batch runtime cannot locate some batch artifacts.
- The batch artifacts throw null pointer exceptions when accessing injected objects.

Using the Context Objects from the Batch Runtime

The batch runtime provides context objects that implement the `JobContext` and `StepContext` interfaces in the `jakarta.batch.runtime.context` package. These objects are associated with the current job and step, respectively, and enable you to do the following:

- Get information from the current job or step, such as its name, instance ID, execution ID, batch status, and exit status
- Set the user-defined exit status

- Store user data
- Get property values from the job or step definition

You can inject context objects from the batch runtime inside batch artifact implementations like item readers, item processors, item writers, batchlets, listeners, and so on. The following example demonstrates how to access property values from the job definition file in an item reader implementation:

```
@Dependent
@Named("MyItemReaderImpl")
public class MyItemReaderImpl implements ItemReader {
    @Inject
    JobContext jobCtx;

    public MyItemReaderImpl() {}

    @Override
    public void open(Serializable checkpoint) throws Exception {
        String fileName = jobCtx.getProperties()
                                .getProperty("log_file_name");
        ...
    }
    ...
}
```

See Dependency Injection in Batch Artifacts for instructions on how to define your batch artifacts to use dependency injection.



Do not access batch context objects inside artifact constructors.

Because the job does not run until you submit it to the batch runtime, the batch context objects are not available when CDI instantiates your artifacts upon loading your application. The instantiation of these beans fails and the batch runtime cannot find your batch artifacts when your application submits the job.

Submitting Jobs to the Batch Runtime

The `JobOperator` interface in the `jakarta.batch.operations` package enables you to submit jobs to the batch runtime and obtain information about existing jobs. This interface provides the following functionality.

- Obtain the names of all known jobs.
- Start, stop, restart, and abandon jobs.
- Obtain job instances and job executions.

The `BatchRuntime` class in the `jakarta.batch.runtime` package provides the `getJobOperator` factory method to obtain `JobOperator` objects.

Starting a Job

The following example code demonstrates how to obtain a `JobOperator` object and submit a batch job:

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
Properties props = new Properties();
props.setProperty("parameter1", "value1");
...
long execID = jobOperator.start("simplejob", props);
```

The first argument of the `JobOperator.start` method is the name of the job as specified in its job definition file. The second parameter is a `Properties` object that represents the parameters for this job execution. You can use job parameters to pass to a job information that is only known at runtime.

Checking the Status of a Job

The `JobExecution` interface in the `jakarta.batch.runtime` package provides methods to obtain information about submitted jobs. This interface provides the following functionality.

- Obtain the batch and exit status of a job execution.
- Obtain the time the execution was started, updated, or ended.
- Obtain the job name.
- Obtain the execution ID.

The following example code demonstrates how to obtain the batch status of a job using its execution ID:

```
JobExecution jobExec = jobOperator.getJobExecution(execID);
String status = jobExec.getBatchStatus().toString();
```

Invoking the Batch Runtime in Your Application

The component from which you invoke the batch runtime depends on the architecture of your particular application. For example, you can invoke the batch runtime from an enterprise bean, a servlet, a managed bean, and so on.

See The webserverlog Example Application and The phonebilling Example Application for details on how to invoke the batch runtime from a managed bean driven by a Jakarta Faces user interface.

Packaging Batch Applications

Job definition files and batch artifacts do not require separate packaging and can be included in any Jakarta EE application.

Package the batch artifact classes with the rest of the classes of your application, and include the

job definition files in one of the following directories:

- `META-INF/batch-jobs/` for `jar` packages
- `WEB-INF/classes/META-INF/batch-jobs/` for `war` packages

The name of each job definition file must match its job ID. For example, if you define a job as follows, and you are packaging your application as a WAR file, include the job definition file in `WEB-INF/classes/META-INF/batch-jobs/simplejob.xml`:

```
<job id="simplejob" xmlns="https://jakarta.ee/xml/ns/jakartaee"
                    version="2.0">
  ...
</job>
```

The webserverlog Example Application

The `webserverlog` example application, located in the `jakartaee-examples/tutorial/batch/webserverlog/` directory, demonstrates how to use the batch framework in Jakarta EE to analyze the log file from a web server. This example application reads a log file and finds what percentage of page views from tablet devices are product sales.

Architecture of the webserverlog Example Application

The `webserverlog` example application consists of the following elements:

- A job definition file (`webserverlog.xml`) that uses the Job Specification Language (JSL) to define a batch job with a chunk step and a task step. The chunk step acts as a filter, and the task step calculates statistics on the remaining entries.
- A log file (`log1.txt`) that serves as input data to the batch job.
- Two Java classes (`LogLine` and `LogFilteredLine`) that represent input items and output items for the chunk step.
- Three batch artifacts (`LogLineReader`, `LogLineProcessor`, and `LogFilteredLineWriter`) that implement the chunk step of the application. This step reads items from the web server log file, filters them by the web browser used by the client, and writes the results to a text file.
- Two batch artifacts (`InfoJobListener` and `InfoItemProcessListener`) that implement two simple listeners.
- A batch artifact (`MobileBatchlet.java`) that calculates statistics on the filtered items.
- Two Facelets pages (`index.xhtml` and `jobstarted.xhtml`) that provide the front end of the batch application. The first page shows the log file that will be processed by the batch job, and the second page enables the user to check on the status of the job and shows the results.
- A managed bean (`FacesBean`) that is accessed from the Facelets pages. The bean submits the job to the batch runtime, checks on the status of the job, and reads the results from a text file.

The Job Definition File

The `webserverlog.xml` job definition file is located in the `WEB-INF/classes/META-INF/batch-jobs/` directory. The file specifies seven job-level properties and two steps:

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="webserverlog" xmlns="https://jakarta.ee/xml/ns/jakartaee"
     version="2.0">
    <properties>
        <property name="log_file_name" value="log1.txt"/>
        <property name="filtered_file_name" value="filtered1.txt"/>
        <property name="num_browsers" value="2"/>
        <property name="browser_1" value="Tablet Browser D"/>
        <property name="browser_2" value="Tablet Browser E"/>
        <property name="buy_page" value="/auth/buy.html"/>
        <property name="out_file_name" value="result1.txt"/>
    </properties>
    <listeners>
        <listener ref="InfoJobListener"/>
    </listeners>
    <step id="mobilefilter" next="mobileanalyzer"> ... </step>
    <step id="mobileanalyzer"> ... </step>
</job>
```

The first step is defined as follows:

```
<step id="mobilefilter" next="mobileanalyzer">
    <listeners>
        <listener ref="InfoItemProcessListeners"/>
    </listeners>
    <chunk checkpoint-policy="item" item-count="10">
        <reader ref="LogLineReader"></reader>
        <processor ref="LogLineProcessor"></processor>
        <writer ref="LogFilteredLineWriter"></writer>
    </chunk>
</step>
```

This step is a normal chunk step that specifies the batch artifacts that implement each phase of the step. The batch artifact names are not fully qualified class names, so the batch artifacts are CDI beans annotated with `@Named`.

The second step is defined as follows:

```
<step id="mobileanalyzer">
    <batchlet ref="MobileBatchlet"></batchlet>
    <end on="COMPLETED"/>
</step>
```

This step is a task step that specifies the batch artifact that implements it. This is the last step of the job.

The LogLine and LogFilteredLine Items

The `LogLine` class represents entries in the web server log file and it is defined as follows:

```
public class LogLine {
    private final String datetime;
    private final String ipaddr;
    private final String browser;
    private final String url;

    /* ... Constructor, getters, and setters ... */
}
```

The `LogFileteredLine` class is similar to this class but only has two fields: the IP address of the client and the URL.

The Chunk Step Batch Artifacts

The first step is composed of the `LogLineReader`, `LogLineProcessor`, and `LogFilteredLineWriter` batch artifacts.

The `LogLineReader` artifact reads records from the web server log file:

```
@Dependent
@Named("LogLineReader")
public class LogLineReader implements ItemReader {
    private ItemNumberCheckpoint checkpoint;
    private String fileName;
    private BufferedReader breader;
    @Inject
    private JobContext jobCtx;

    public LogLineReader() { }

    /* ... Override the open, close, readItem, and
     *     checkpointInfo methods ... */
}
```

The `open` method reads the `log_file_name` property and opens the log file with a buffered reader. In this example, the log file has been included with the application under `webserverlog/WEB-INF/classes/log1.txt`:

```
fileName = jobCtx.getProperties().getProperty("log_file_name");
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
InputStream iStream = classLoader.getResourceAsStream(fileName);
```

```
breader = new BufferedReader(new InputStreamReader(iStream));
```

If a checkpoint object is provided, the `open` method advances the reader up to the last checkpoint. Otherwise, this method creates a new checkpoint object. The checkpoint object keeps track of the line number from the last committed chunk.

The `readItem` method returns a new `LogLine` object or null at the end of the log file:

```
@Override
public Object readItem() throws Exception {
    String entry = breader.readLine();
    if (entry != null) {
        checkpoint.nextLine();
        return new LogLine(entry);
    } else {
        return null;
    }
}
```

The `LogLineProcessor` artifact obtains a list of browsers from the job properties and filters the log entries according to the list:

```
@Override
public Object processItem(Object item) {
    /* Obtain a list of browsers we are interested in */
    if (nbrowsers == 0) {
        Properties props = jobCtx.getProperties();
        nbrowsers = Integer.parseInt(props.getProperty("num_browsers"));
        browsers = new String[nbrowsers];
        for (int i = 1; i < nbrowsers + 1; i++)
            browsers[i - 1] = props.getProperty("browser_" + i);
    }

    LogLine logline = (LogLine) item;
    /* Filter for only the mobile/tablet browsers as specified */
    for (int i = 0; i < nbrowsers; i++) {
        if (logline.getBrowser().equals(browsers[i])) {
            return new LogFilteredLine(logline);
        }
    }
    return null;
}
```

The `LogFilteredLineWriter` artifact reads the name of the output file from the job properties. The `open` method opens the file for writing. If a checkpoint object is provided, the artifact continues writing at the end of the file; otherwise, it overwrites the file if it exists. The `writeItems` method writes filtered items to the output file:

```
@Override
public void writeItems(List<Object> items) throws Exception {
    /* Write the filtered lines to the output file */
    for (int i = 0; i < items.size(); i++) {
        LogFilteredLine filtLine = (LogFilteredLine) items.get(i);
        bwriter.write(filtLine.toString());
        bwriter.newLine();
    }
}
```

The Listener Batch Artifacts

The `InfoJobListener` batch artifact implements a simple listener that writes log messages when the job starts and when it ends:

```
@Dependent
@Named("InfoJobListener")
public class InfoJobListener implements JobListener {
    ...
    @Override
    public void beforeJob() throws Exception {
        logger.log(Level.INFO, "The job is starting");
    }

    @Override
    public void afterJob() throws Exception { ... }
}
```

The `InfoItemProcessListener` batch artifact implements the `ItemProcessListener` interface for chunk steps:

```
@Dependent
@Named("InfoItemProcessListener")
public class InfoItemProcessListener implements ItemProcessListener {
    ...
    @Override
    public void beforeProcess(Object o) throws Exception {
        LogLine logline = (LogLine) o;
        llogger.log(Level.INFO, "Processing entry {0}", logline);
    }
    ...
}
```

The Task Step Batch Artifact

The task step is implemented by the `MobileBatchlet` artifact, which computes what percentage of the filtered log entries are purchases:

```
@Override
public String process() throws Exception {
    /* Get properties from the job definition file */
    ...
    /* Count from the output of the previous chunk step */
    breader = new BufferedReader(new FileReader(fileName));
    String line = breader.readLine();
    while (line != null) {
        String[] lineSplit = line.split(", ");
        if (buyPage.compareTo(lineSplit[1]) == 0)
            pageVisits++;
        totalVisits++;
        line = breader.readLine();
    }
    breader.close();
    /* Write the result */
    ...
}
```

The Jakarta Faces Pages

The `index.xhtml` page contains a text area that shows the web server log. The page provides a button for the user to submit the batch job and navigate to the next page:

```
<body>
    ...
    <textarea cols="90" rows="25"
              readonly="true">#{facesBean.getInputLog()}</textarea>
    <p> </p>
    <h:form>
        <h:commandButton value="Start Batch Job"
                         action="#{facesBean.startBatchJob()}" />
    </h:form>
</body>
```

This page calls the methods of the managed bean to show the log file and submit the batch job.

The `jobstarted.xhtml` page provides a button to check the current status of the batch job and displays the results when the job finishes:

```
<p>Current Status of the Job: <b>#{facesBean.jobStatus}</b></p>
<p>#{facesBean.showResults()}</p>
<h:form>
    <h:commandButton value="Check Status"
                     action="jobstarted"
                     rendered="#{facesBean.completed==false}" />
</h:form>
```

The Managed Bean

The `FacesBean` managed bean submits the job to the batch runtime, checks on the status of the job, and reads the results from a text file.

The `startBatchJob` method submits the job to the batch runtime:

```
/* Submit the batch job to the batch runtime.
 * Faces Navigation method (return the name of the next page) */
public String startBatchJob() {
    jobOperator = BatchRuntime.getJobOperator();
    execID = jobOperator.start("webserverlog", null);
    return "jobstarted";
}
```

The `getJobStatus` method checks the status of the job:

```
/* Get the status of the job from the batch runtime */
public String getJobStatus() {
    return jobOperator.getJobExecution(execID).getBatchStatus().toString();
}
```

The `showResults` method reads the results from a text file.

Running the webserverlog Example Application

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `webserverlog` example application.

To Run the webserverlog Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/batch
   ```

4. Select the `webserverlog` folder.
5. Click Open Project.
6. In the Projects tab, right-click the `webserverlog` project and select Run.

   This command builds and packages the application into a WAR file, `webserverlog.war`, located in the `target/` directory; deploys it to the server; and launches a web browser window at the following URL:

```
http://localhost:8080/webserverlog/
```

To Run the webserverlog Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

```
jakartaee-examples/tutorial/batch/webserverlog/
```

3. Enter the following command to deploy the application:

```
mvn install
```

4. Open a web browser window at the following URL:

```
http://localhost:8080/webserverlog/
```

The phonebilling Example Application

The `phonebilling` example application, located in the `jakartaee-examples/tutorial/batch/phonebilling/` directory, demonstrates how to use the batch framework in Jakarta EE to implement a phone billing system. This example application processes a log file of phone calls and creates a bill for each customer.

Architecture of the phonebilling Example Application

The `phonebilling` example application consists of the following elements.

- A job definition file (`phonebilling.xml`) that uses the Job Specification Language (JSL) to define a batch job with two chunk steps. The first step reads call records from a log file and associates them with a bill. The second step computes the amount due and writes each bill to a text file.
- A Java class (`CallRecordLogCreator`) that creates the log file for the batch job. This is an auxiliary component that does not demonstrate any key functionality in this example.
- Two Jakarta Persistence entities (`CallRecord` and `PhoneBill`) that represent call records and customer bills. The application uses a Jakarta Persistence entity manager to store instances of these entities in a database.
- Three batch artifacts (`CallRecordReader`, `CallRecordProcessor`, and `CallRecordWriter`) that implement the first step of the application. This step reads call records from the log file, associates them with a bill, and stores them in a database.
- Four batch artifacts (`BillReader`, `BillProcessor`, `BillWriter`, and `BillPartitionMapper`) that implement the second step of the application. This step is a partitioned step that gets each bill from the database, calculates the amount due, and writes it to a text file.

- Two Facelets pages (`index.xhtml` and `jobstarted.xhtml`) that provide the front end of the batch application. The first page shows the log file that will be processed by the batch job, and the second page enables the user to check on the status of the job and shows the resulting bill for each customer.
- A managed bean (`FacesBean`) that is accessed from the Facelets pages. The bean submits the job to the batch runtime, checks on the status of the job, and reads the text files for each bill.

The Job Definition File

The `phonebilling.xml` job definition file is located in the `WEB-INF/classes/META-INF/batch-jobs/` directory. The file specifies three job-level properties and two steps:

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="phonebilling" xmlns="https://jakarta.ee/xml/ns/jakartaee"
     version="2.0">
    <properties>
        <property name="log_file_name" value="log1.txt"/>
        <property name="airtime_price" value="0.08"/>
        <property name="tax_rate" value="0.07"/>
    </properties>
    <step id="callrecords" next="bills"> ... </step>
    <step id="bills"> ... </step>
</job>
```

The first step is defined as follows:

```
<step id="callrecords" next="bills">
    <chunk checkpoint-policy="item" item-count="10">
        <reader ref="CallRecordReader"></reader>
        <processor ref="CallRecordProcessor"></processor>
        <writer ref="CallRecordWriter"></writer>
    </chunk>
</step>
```

This step is a normal chunk step that specifies the batch artifacts that implement each phase of the step. The batch artifact names are not fully qualified class names, so the batch artifacts are CDI beans annotated with `@Named`.

The second step is defined as follows:

```
<step id="bills">
    <chunk checkpoint-policy="item" item-count="2">
        <reader ref="BillReader">
            <properties>
                <property name="firstItem" value="#{partitionPlan['firstItem']}"/>
                <property name="numItems" value="#{partitionPlan['numItems']}"/>
            </properties>
```

```
        </reader>
        <processor ref="BillProcessor"></processor>
        <writer ref="BillWriter"></writer>
    </chunk>
    <partition>
        <mapper ref="BillPartitionMapper"/>
    </partition>
    <end on="COMPLETED"/>
</step>
```

This step is a partitioned chunk step. The partition plan is specified through the `BillPartitionMapper` artifact instead of using the `plan` element.

The CallRecord and PhoneBill Entities

The `CallRecord` entity is defined as follows:

```
@Entity
public class CallRecord implements Serializable {
    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.DATE)
    private Date datetime;
    private String fromNumber;
    private String toNumber;
    private int minutes;
    private int seconds;
    private BigDecimal price;

    public CallRecord() { }

    public CallRecord(String datetime, String from,
            String to, int min, int sec)            throws ParseException { ... }

    public CallRecord(String jsonData) throws ParseException { ... }

    /* ... Getters and setters ... */
}
```

The `id` field is generated automatically by the Jakarta Persistence implementation to store and retrieve `CallRecord` objects to and from a database.

The second constructor creates a `CallRecord` object from an entry of JSON data in the log file using Jakarta JSON Processing. Log entries look as follows:

```
{"datetime":"03/01/2013 04:03","from":"555-0101",
"to":"555-0114","length":"03:39"}
```

The `PhoneBill` entity is defined as follows:

```
@Entity
public class PhoneBill implements Serializable {
    @Id
    private String phoneNumber;
    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    @OrderBy("datetime ASC")
    private List<CallRecord> calls;
    private BigDecimal amountBase;
    private BigDecimal taxRate;
    private BigDecimal tax;
    private BigDecimal amountTotal;

    public PhoneBill() { }

    public PhoneBill(String number) {
        this.phoneNumber = number;
        calls = new ArrayList<>();
    }

    public void addCall(CallRecord call) {
        calls.add(call);
    }

    public void calculate(BigDecimal taxRate) { ... }

    /* ... Getters and setters ... */
}
```

The `OneToMany` annotation defines the relationship between a bill and its call records. The `FetchType.EAGER` attribute specifies that the collection should be retrieved eagerly. The `CascadeType.PERSIST` attribute indicates that the elements in the call list should be automatically persisted when the phone bill is persisted. The `OrderBy` annotation defines an order for retrieving the elements of the call list from the database.

The batch artifacts use instances of these two entities as items to read, process, and write.

For more information on Jakarta Persistence, see [persist:persistence-intro::persistence-intro:::_introduction_to_jakarta_persistence]. For more information on Jakarta JSON Processing, see [web:jsonp::jsonp:::_json_processing].

The Call Records Chunk Step

The first step is composed of the `CallRecordReader`, `CallRecordProcessor`, and `CallRecordWriter` batch artifacts.

The `CallRecordReader` artifact reads call records from the log file:

```
@Dependent
@Named("CallRecordReader")
public class CallRecordReader implements ItemReader {
    private ItemNumberCheckpoint checkpoint;
    private String fileName;
    private BufferedReader breader;
    @Inject
    JobContext jobCtx;

    /* ... Override the open, close, readItem,
     *     and checkpointInfo methods ... */
}
```

The `open` method reads the `log_filename` property and opens the log file with a buffered reader:

```
fileName = jobCtx.getProperties().getProperty("log_file_name");
breader = new BufferedReader(new FileReader(fileName));
```

If a checkpoint object is provided, the `open` method advances the reader up to the last checkpoint. Otherwise, this method creates a new checkpoint object. The checkpoint object keeps track of the line number from the last committed chunk.

The `readItem` method returns a new `CallRecord` object or null at the end of the log file:

```
@Override
public Object readItem() throws Exception {
    /* Read a line from the log file and
     * create a CallRecord from JSON */
    String callEntryJson = breader.readLine();
    if (callEntryJson != null) {
        checkpoint.nextItem();
        return new CallRecord(callEntryJson);
    } else
        return null;
}
```

The `CallRecordProcessor` artifact obtains the airtime price from the job properties, calculates the price of each call, and returns the call object. This artifact overrides only the `processItem` method.

The `CallRecordWriter` artifact associates each call record with a bill and stores the bill in the database. This artifact overrides the `open`, `close`, `writeItems`, and `checkpointInfo` methods. The `writeItems` method looks like this:

```
@Override
public void writeItems(List<Object> callList) throws Exception {

    for (Object callObject : callList) {
```

```
        CallRecord call = (CallRecord) callObject;
        PhoneBill bill = em.find(PhoneBill.class, call.getFromNumber());
        if (bill == null) {
            /* No bill for this customer yet, create one */
            bill = new PhoneBill(call.getFromNumber());
            bill.addCall(call);
            em.persist(bill);
        } else {
            /* Add call to existing bill */
            bill.addCall(call);
        }
    }
}
```

The Phone Billing Chunk Step

The second step is composed of the `BillReader`, `BillProcessor`, `BillWriter`, and `BillPartitionMapper` batch artifacts. This step gets the phone bills from the database, computes the tax and total amount due, and writes each bill to a text file. Since the processing of each bill is independent of the others, this step can be partitioned and run in more than one thread.

The `BillPartitionMapper` artifact specifies the number of partitions and the parameters for each partition. In this example, the parameters represent the range of items each partition should process. The artifact obtains the number of bills in the database to calculate these ranges. It provides a partition plan object that overrides the `getPartitions` and `getPartitionProperties` methods of the `PartitionPlan` interface. The `getPartitions` method looks like this:

```
@Override
public Properties[] getPartitionProperties() {
    /* Assign an (approximately) equal number of elements
     * to each partition. */
    long totalItems = getBillCount();
    long partItems = (long) totalItems / getPartitions();
    long remItems = totalItems % getPartitions();

    /* Populate a Properties array. Each Properties element
     * in the array corresponds to a partition. */
    Properties[] props = new Properties[getPartitions()];

    for (int i = 0; i < getPartitions(); i++) {
        props[i] = new Properties();
        props[i].setProperty("firstItem",
                String.valueOf(i * partItems));
        /* Last partition gets the remainder elements */
        if (i == getPartitions() - 1) {
            props[i].setProperty("numItems",
                    String.valueOf(partItems + remItems));
        } else {
            props[i].setProperty("numItems",
                    String.valueOf(partItems));
```

```
        }
    }
    return props;
}
```

The `BillReader` artifact obtains the partition parameters as follows:

```
@Dependent
@Named("BillReader")
public class BillReader implements ItemReader {

    @Inject @BatchProperty(name = "firstItem")
    private String firstItemValue;
    @Inject @BatchProperty(name = "numItems")
    private String numItemsValue;
    private ItemNumberCheckpoint checkpoint;
    @PersistenceContext
    private EntityManager em;
    private Iterator iterator;

    @Override
    public void open(Serializable ckpt) throws Exception {
        /* Get the range of items to work on in this partition */
        long firstItem0 = Long.parseLong(firstItemValue);
        long numItems0 = Long.parseLong(numItemsValue);

        if (ckpt == null) {
            /* Create a checkpoint object for this partition */
            checkpoint = new ItemNumberCheckpoint();
            checkpoint.setItemNumber(firstItem0);
            checkpoint.setNumItems(numItems0);
        } else {
            checkpoint = (ItemNumberCheckpoint) ckpt;
        }

        /* Adjust range for this partition from the checkpoint */
        long firstItem = checkpoint.getItemNumber();
        long numItems = numItems0 - (firstItem - firstItem0);
        ...
    }
    ...
}
```

This artifact also obtains an iterator to read items from the Jakarta Persistence entity manager:

```
/* Obtain an iterator for the bills in this partition */
String query = "SELECT b FROM PhoneBill b ORDER BY b.phoneNumber";
Query q = em.createQuery(query).setFirstResult((int) firstItem)
        .setMaxResults((int) numItems);
```

```
iterator = q.getResultList().iterator();
```

The `BillProcessor` artifact iterates over the list of call records in a bill and calculates the tax and total amount due for each bill.

The `BillWriter` artifact writes each bill to a plain text file.

The Jakarta Faces Pages

The `index.xhtml` page contains a text area that shows the log file of call records. The page provides a button for the user to submit the batch job and navigate to the next page:

```
<body>
    <h1>The Phone Billing Example Application</h1>
    <h2>Log file</h2>
    <p>The batch job analyzes the following log file:</p>
    <textarea cols="90" rows="25"
              readonly="true">#{facesBean.createAndShowLog()}</textarea>
    <p> </p>
    <h:form>
        <h:commandButton value="Start Batch Job"
                         action="#{facesBean.startBatchJob()}" />
    </h:form>
</body>
```

This page calls the methods of the managed bean to show the log file and submit the batch job.

The `jobstarted.xhtml` page provides a button to check the current status of the batch job and displays the bills when the job finishes:

```
<p>Current Status of the Job: <b>#{facesBean.jobStatus}</b></p>
<h:dataTable var="_row" value="#{facesBean.rowList}"
             border="1" rendered="#{facesBean.completed}">
    <!-- ... show results from facesBean.rowList ... -->
</h:dataTable>
<!-- Render the check status button if the job has not finished -->
<h:form>
    <h:commandButton value="Check Status"
                     rendered="#{facesBean.completed==false}"
                     action="jobstarted" />
</h:form>
```

The Managed Bean

The `FacesBean` managed bean submits the job to the batch runtime, checks on the status of the job, and reads the text files for each bill.

The `startBatchJob` method of the bean submits the job to the batch runtime:

```
/* Submit the batch job to the batch runtime.
 * Faces Navigation method (return the name of the next page) */
public String startBatchJob() {
    jobOperator = BatchRuntime.getJobOperator();
    execID = jobOperator.start("phonebilling", null);
    return "jobstarted";
}
```

The `getJobStatus` method of the bean checks the status of the job:

```
/* Get the status of the job from the batch runtime */
public String getJobStatus() {
    return jobOperator.getJobExecution(execID).getBatchStatus().toString();
}
```

The `getRowList` method of the bean creates a list of bills to be displayed on the `jobstarted.xhtml` faces page using a table.

Running the phonebilling Example Application

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `phonebilling` example application.

To Run the phonebilling Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/batch
   ```

4. Select the `phonebilling` folder.
5. Click Open Project.
6. In the Projects tab, right-click the `phonebilling` project and select Run.

   This command builds and packages the application into a WAR file, `phonebilling.war`, located in the `target/` directory; deploys it to the server; and launches a web browser window at the following URL:

   ```
   http://localhost:8080/phonebilling/
   ```

To Run the phonebilling Example Application Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).

2. In a terminal window, go to:

```
jakartaee-examples/tutorial/batch/phonebilling/
```

3. Enter the following command to deploy the application:

```
mvn install
```

4. Open a web browser window at the following URL:

```
http://localhost:8080/phonebilling/
```

Further Information about Batch Processing

For more information on batch processing in Jakarta EE, see Jakarta Batch:
https://jakarta.ee/specifications/batch/2.1/

Advanced

Web Profile

Jakarta Authentication

Jakarta Authentication Specification

Jakarta Authorization

Jakarta Authorization Specification

Jakarta Transactions



This section was last updated for Jakarta EE Version 9.1

This chapter describes types of transactions and how they are managed in different applications.

Overview of Transactions

A typical enterprise application accesses and stores information in one or more databases. Because this information is critical for business operations, it must be accurate, current, and reliable. Data integrity would be lost if multiple programs were allowed to update the same information simultaneously or if a system that failed while processing a business transaction were to leave the affected data only partially updated. By preventing both of these scenarios, software transactions ensure data integrity. Transactions control the concurrent access of data by multiple programs. In the event of a system failure, transactions make sure that after recovery, the data will be in a consistent state.

Transactions in Jakarta EE Applications

In a Jakarta EE application, a transaction is a series of actions that must all complete successfully, or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.

Jakarta Transactions allows applications to access transactions in a manner that is independent of specific implementations. Jakarta Transactions specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the transactional application, the Jakarta EE server, and the manager that controls access to the shared resources affected by the transactions.

Jakarta Transactions defines the `UserTransaction` interface that applications use to start, commit, or roll back transactions. Application components get a `UserTransaction` object through a JNDI lookup by using the name `java:comp/UserTransaction` or by requesting injection of a `UserTransaction` object. An application server uses a number of Jakarta Transactions defined interfaces to communicate with a transaction manager; a transaction manager uses Jakarta Transactions defined interfaces to interact with a resource manager.

The Jakarta Transactions 2.0 specification is available at https://jakarta.ee/specifications/transactions/2.0/.

What Is a Transaction?

To emulate a business transaction, a program may need to perform several steps. A financial program, for example, might transfer funds from a checking account to a savings account by using the steps listed in the following pseudocode:

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

Either all or none of the three steps must complete. Otherwise, data integrity is lost. Because the steps within a transaction are a unified whole, a transaction is often defined as an indivisible unit of work.

A transaction can end in two ways: with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction. In the pseudocode, for example, if a disk drive were to crash during the `credit` step, the transaction would roll back and undo the data modifications made by the `debit` statement. Although the transaction fails, data integrity would be intact because the accounts still balance.

In the preceding pseudocode, the `begin` and `commit` statements mark the boundaries of the transaction. When designing an enterprise bean, you determine how the boundaries are set by specifying either container-managed or bean-managed transactions.

Container-Managed Transactions

In an enterprise bean with container-managed transaction demarcation, the enterprise bean container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction. By default, if no transaction demarcation is specified, enterprise beans use container-managed transaction demarcation.

Typically, the container begins a transaction immediately before an enterprise bean method starts and commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When developing a bean, you can set the transaction attributes to specify which of the bean's methods are associated with transactions.

Enterprise beans that use container-managed transaction demarcation must not use any

transaction-management methods that interfere with the container's transaction demarcation boundaries. Examples of such methods are the `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection` or the `commit` and `rollback` methods of `jakarta.jms.Session`. If you require control over the transaction demarcation, you must use application-managed transaction demarcation.

Enterprise beans that use container-managed transaction demarcation also must not use the `jakarta.transaction.UserTransaction` interface.

Transaction Attributes

A transaction attribute controls the scope of a transaction. Figure 39, "Transaction Scope" illustrates why controlling the scope is important. In the diagram, `method-A` begins a transaction and then invokes `method-B` of `Bean-2`. When `method-B` executes, does it run within the scope of the transaction started by `method-A`, or does it execute with a new transaction? The answer depends on the transaction attribute of `method-B`.

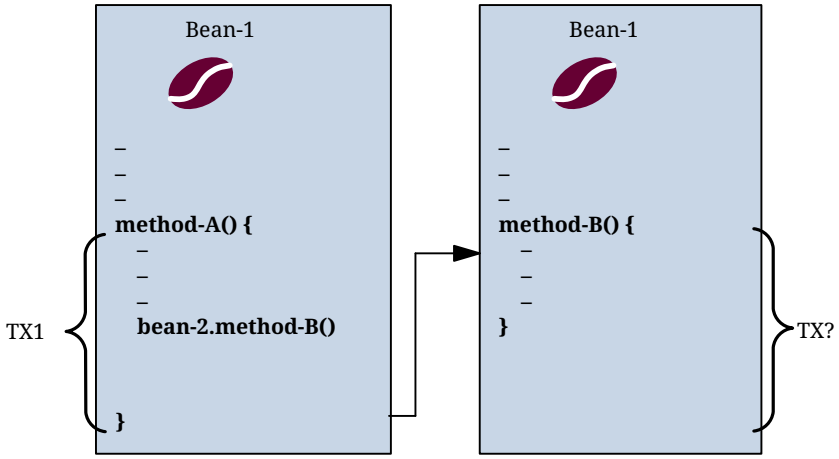


Figure 39. Transaction Scope

A transaction attribute can have one of the following values:

- `Required`
- `RequiresNew`
- `Mandatory`
- `NotSupported`
- `Supports`
- `Never`

Required Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The `Required` attribute is the implicit transaction attribute for all enterprise bean methods running with container-managed transaction demarcation. You typically do not set the `Required` attribute

unless you need to override another transaction attribute. Because transaction attributes are declarative, you can easily change them later.

RequiresNew Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:

1. Suspends the client's transaction
2. Starts a new transaction
3. Delegates the call to the method
4. Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the `RequiresNew` attribute when you want to ensure that the method always runs within a new transaction.

Mandatory Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws a `TransactionRequiredException`.

Use the `Mandatory` attribute if the enterprise bean's method must use the transaction of the client.

NotSupported Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the `NotSupported` attribute for methods that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

Supports Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the `Supports` attribute with caution.

Never Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container throws a `RemoteException`. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Summary of Transaction Attributes

Transaction Attributes and Scope summarizes the effects of the transaction attributes. Both the `T1` and the `T2` transactions are controlled by the container. A `T1` transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A `T2` transaction is started by the container just before the method executes.

In the last column of Transaction Attributes and Scope, the word "None" means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the database management system.

Transaction Attributes and Scope

Transaction Attribute	Client's Transaction	Business Method's Transaction
`Required`	None	T2
`Required`	T1	T1
`RequiresNew`	None	T2
`RequiresNew`	T1	T2
`Mandatory`	None	Error
`Mandatory`	T1	T1
`NotSupported`	None	None
`NotSupported`	T1	None
`Supports`	None	None
`Supports`	T1	T1
`Never`	None	None
`Never`	T1	Error

Setting Transaction Attributes

Transaction attributes are specified by decorating the enterprise bean class or method with a `jakarta.ejb.TransactionAttribute` annotation and setting it to one of the `jakarta.ejb.TransactionAttributeType` constants.

If you decorate the enterprise bean class with `@TransactionAttribute`, the specified `TransactionAttributeType` is applied to all the business methods in the class. Decorating a business method with `@TransactionAttribute` applies the `TransactionAttributeType` only to that method. If a `@TransactionAttribute` annotation decorates both the class and the method, the method

`TransactionAttributeType` overrides the class `TransactionAttributeType`.

The `TransactionAttributeType` constants shown in TransactionAttributeType Constants encapsulate the transaction attributes described earlier in this section.

TransactionAttributeType Constants

Transaction Attribute	TransactionAttributeType Constant
Required	`TransactionAttributeType.REQUIRED`
RequiresNew	`TransactionAttributeType.REQUIRES_NEW`
Mandatory	`TransactionAttributeType.MANDATORY`
NotSupported	`TransactionAttributeType.NOT_SUPPORTED`
Supports	`TransactionAttributeType.SUPPORTS`
Never	`TransactionAttributeType.NEVER`

The following code snippet demonstrates how to use the `@TransactionAttribute` annotation:

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

    public void fourthMethod() {...}
}
```

In this example, the `TransactionBean` class's transaction attribute has been set to `NotSupported`, `firstMethod` has been set to `RequiresNew`, and `secondMethod` has been set to `Required`. Because a `@TransactionAttribute` set on a method overrides the class `@TransactionAttribute`, calls to `firstMethod` will create a new transaction, and calls to `secondMethod` will either run in the current transaction or start a new transaction. Calls to `thirdMethod` or `fourthMethod` do not take place within

a transaction.

Rolling Back a Container-Managed Transaction

There are two ways to roll back a container-managed transaction. First, if a system exception is thrown, the container will automatically roll back the transaction. Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to `setRollbackOnly`.

Synchronizing a Session Bean's Instance Variables

The `SessionSynchronization` interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications. For example, you could synchronize the instance variables of an enterprise bean with their corresponding values in the database. The container invokes the `SessionSynchronization` methods (`afterBegin`, `beforeCompletion`, and `afterCompletion`) at each of the main stages of a transaction.

The `afterBegin` method informs the instance that a new transaction has begun. The container invokes `afterBegin` immediately before it invokes the business method.

The container invokes the `beforeCompletion` method after the business method has finished but just before the transaction commits. The `beforeCompletion` method is the last opportunity for the session bean to roll back the transaction (by calling `setRollbackOnly`).

The `afterCompletion` method indicates that the transaction has completed. This method has a single `boolean` parameter whose value is `true` if the transaction was committed and `false` if it was rolled back.

Methods Not Allowed in Container-Managed Transactions

You should not invoke any method that might interfere with the transaction boundaries set by the container. The following methods are prohibited:

- The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- The `getUserTransaction` method of `jakarta.ejb.EJBContext`
- Any method of `jakarta.transaction.UserTransaction`

You can, however, use these methods to set boundaries in application-managed transactions.

Bean-Managed Transactions

In bean-managed transaction demarcation, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using bean-managed transactions.

The following pseudocode illustrates the kind of fine-grained control you can obtain with application-managed transactions. By checking various conditions, the pseudocode decides

whether to start or stop certain transactions within the business method:

```
begin transaction
...
    update table-a
...
if (condition-x)
   commit transaction
else if (condition-y)
   update table-b
   commit transaction
else
   rollback transaction
   begin transaction
   update table-c
   commit transaction
```

When coding an application-managed transaction for session or message-driven beans, you must decide whether to use Java Database Connectivity or Jakarta transactions. The sections that follow discuss both types of transactions.

Jakarta Transactions

Jakarta Transactions allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. GlassFish Server implements the transaction manager with the Java Transaction Service (JTS). However, your code doesn't call the JTS methods directly but instead invokes the Jakarta Transactions methods, which then call the lower-level JTS routines.

A Jakarta transaction is controlled by the Jakarta EE transaction manager. You may want to use a Jakarta transaction because it can span updates to multiple databases from different vendors. A particular DBMS's transaction manager may not work with heterogeneous databases. However, the Jakarta EE transaction manager does have one limitation: It does not support nested transactions. In other words, it cannot start a transaction for an instance until the preceding transaction has ended.

To demarcate a Jakarta transaction, you invoke the `begin`, `commit`, and `rollback` methods of the `jakarta.transaction.UserTransaction` interface.

Returning without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a Jakarta transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a JDBC transaction, the JDBC connection retains the association

between the bean instance and the transaction across multiple calls. If the connection is closed, the association is not retained.

Methods Not Allowed in Bean-Managed Transactions

Do not invoke the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface in bean-managed transactions. These methods should be used only in container-managed transactions. For bean-managed transactions, invoke the `getStatus` and `rollback` methods of the `UserTransaction` interface.

Transaction Timeouts

For container-managed transactions, you can use the Administration Console to configure the transaction timeout interval. See Starting the Administration Console.

For enterprise beans with bean-managed Jakarta transactions, you invoke the `setTransactionTimeout` method of the `UserTransaction` interface.

To Set a Transaction Timeout

1. In the Administration Console, expand the Configurations node, then expand the server-config node and select Transaction Service.
2. On the Transaction Service page, set the value of the Transaction Timeout field to the value of your choice (for example, 5).

   With this setting, if the transaction has not completed within 5 seconds, the enterprise bean container rolls it back.

   The default value is 0, meaning that the transaction will not time out.
3. Click Save.

Updating Multiple Databases

The Jakarta EE transaction manager controls all enterprise bean transactions except for bean-managed JDBC transactions. The Jakarta EE transaction manager allows an enterprise bean to update multiple databases within a transaction. Figure 40, “Updating Multiple Databases” and Figure 41, “Updating Multiple Databases Across Jakarta EE Servers” show two scenarios for updating multiple databases in a single transaction.

In Figure 40, “Updating Multiple Databases”, the client invokes a business method in `Bean-A`. The business method begins a transaction, updates Database X, updates Database Y, and invokes a business method in `Bean-B`. The second business method updates Database Z and returns control to the business method in `Bean-A`, which commits the transaction. All three database updates occur in the same transaction.

In Figure 41, “Updating Multiple Databases Across Jakarta EE Servers”, the client calls a business method in `Bean-A`, which begins a transaction and updates Database X. Then `Bean-A` invokes a method in `Bean-B`, which resides in a remote Jakarta EE server. The method in `Bean-B` updates Database Y. The transaction managers of the Jakarta EE servers ensure that both databases are

updated in the same transaction.

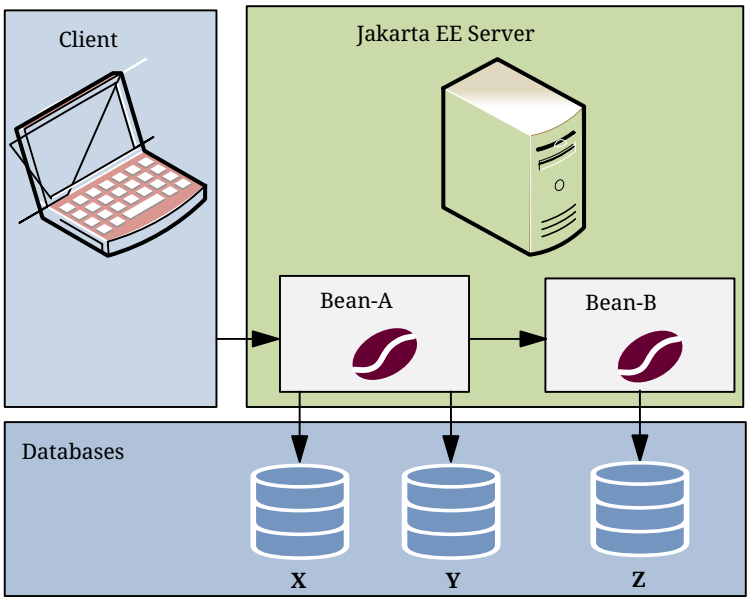


Figure 40. Updating Multiple Databases

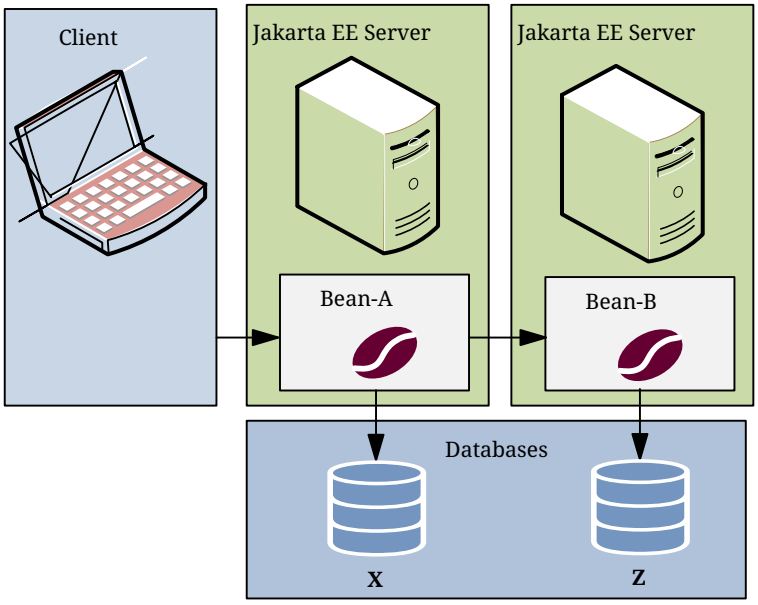


Figure 41. Updating Multiple Databases Across Jakarta EE Servers

Transactions in Web Components

You can demarcate a transaction in a web component by using either the `java.sql.Connection` or the `jakarta.transaction.UserTransaction` interface. These are the same interfaces that a session bean with bean-managed transactions can use. Transactions demarcated with the `UserTransaction` interface are discussed in Jakarta Transactions.

Further Information about Transactions

For more information about transactions, see the Jakarta Transactions 2.0 specification at https://jakarta.ee/specifications/transactions/2.0/.

Jakarta Concurrency



This section was last updated for Jakarta EE Version 9.1

This chapter describes Jakarta Concurrency spec.

Concurrency Basics

Concurrency is the concept of executing two or more tasks at the same time (in parallel). Tasks may include methods (functions), parts of a program, or even other programs. With current computer architectures, support for multiple cores and multiple processors in a single CPU is very common.

The Java Platform has always offered support for concurrent programming, which was the basis for implementing many of the services offered by Jakarta EE containers. Since Java SE 5, additional high-level API support for concurrency was provided by the `java.util.concurrent` package.

Threads and Processes

The two main concurrency concepts are processes and threads.

Processes are primarily associated with applications running on the operating system (OS). A process has specific runtime resources to interact with the underlying OS and allocate other resources, such as its own memory, just as the JVM process does. A JVM is in fact a process.

The Java programming language and platform are primarily concerned with threads.

Threads share some features with processes, since both consume resources from the OS or the execution environment. But threads are easier to create and consume many fewer resources than a process.

Because threads are so lightweight, any modern CPU that has a couple of cores and a few gigabytes of RAM can handle thousands of threads in a single JVM process. The precise number of threads will depend on the combined output of the CPU, OS, and RAM available, as well as on correct configuration (tuning) of the JVM.

Although concurrent programming solves many problems and can improve performance for most applications, there are a number of situations where multiple execution lines (threads or processes) can cause major problems. These situations include the following:

- Deadlocks
- Thread starvation
- Concurrent accessing of shared resources
- Situations when the program generates incorrect data

Main Components of the Concurrency Utilities

Concurrent resources are managed objects that provide concurrency capabilities to Jakarta EE applications. In GlassFish Server, you configure concurrent resources and then make them available for use by application components such as servlets and enterprise beans. Concurrent

resources are accessed through JNDI lookup or resource injection.

The primary components of the concurrency utilities are as follows.

- `ManagedExecutorService`: A managed executor service is used by applications to execute submitted tasks asynchronously. Tasks are executed on threads that are started and managed by the container. The context of the container is propagated to the thread executing the task.

  For example, by using an `ManagedExecutorService.submit()` call, a task, such as the GenerateReportTask, could be submitted to execute at a later time and then, by using the `Future` object callback, retrieve the result when it becomes available.

- `ManagedScheduledExecutorService`: A managed scheduled executor service is used by applications to execute submitted tasks asynchronously at specific times. Tasks are executed on threads that are started and managed by the container. The context of the container is propagated to the thread executing the task. The API provides the scheduling functionality that allows users to set a specific date/time for the Task execution programmatically in the application.
- `ContextService`: A context service is used to create dynamic proxy objects that capture the context of a container and enable applications to run within that context at a later time or be submitted to a Managed Executor Service. The context of the container is propagated to the thread executing the task.
- `ManagedThreadFactory`: A managed thread factory is used by applications to create managed threads. The threads are started and managed by the container. The context of the container is propagated to the thread executing the task. This object can also be used to provide custom factories for specific use cases (with custom Threads) and, for example, set specific/proprietary properties to these objects.

Concurrency and Transactions

The most basic operations for transactions are commit and rollback, but, in a distributed environment with concurrent processing, it can be difficult to guarantee that commit or rollback operations will be successfully processed, and the transaction can be spread among different threads, CPU cores, physical machines, and networks.

Ensuring that a rollback operation will successfully execute in such a scenario is crucial. Concurrency Utilities relies on Jakarta Transactions to implement and support transactions on its components through `jakarta.transaction.UserTransaction`, allowing application developers to explicitly manage transaction boundaries. More information is available in the Jakarta Transactions specification.

Optionally, context objects can begin, commit, or roll back transactions, but these objects cannot enlist in parent component transactions.

The following code snippet illustrates a `Runnable` task that obtains a `UserTransaction` and then starts and commits a transaction while interacting with other transactional components, such as an enterprise bean and a database:

```
public class MyTransactionalTask implements Runnable {
```

```
    UserTransaction ut = ... // obtained through JNDI or injection

    public void run() {

        // Start a transaction
        ut.begin();

        // Invoke a Service or an EJB
        myEJB.businessMethod();

        // Update a database entity using an XA JDBC driver
        myEJB.updateCustomer(customer);

        // Commit the transaction
        ut.commit();

    }
}
```

Concurrency and Security

Jakarta Concurrency defers most security decisions to the application server implementation. If, however, the container supports a security context, that context can be propagated to the thread of execution. The `ContextService` can support several runtime behaviors, and the `security` attribute, if enabled, will propagate the container security principal.

The jobs Concurrency Example

This section describes a very basic example that shows how to use some of the basic concurrency features in an enterprise application. Specifically, this example uses one of the main components of Jakarta Concurrency, a Managed Executor Service.

The example demonstrates a scenario where a RESTful web service, exposed as a public API, is used to submit generic jobs for execution. These jobs are processed in the background. Each job prints a "Starting" and a "Finished" message at the beginning and end of the execution. Also, to simulate background processing, each job takes 10 seconds to execute.

The RESTful service exposes two methods:

- `/token`: Exposed as a GET method that registers and returns valid API tokens
- `/process`: Exposed as a POST method that receives a `jobID` query parameter, which is the identifier for the job to be executed, and a custom HTTP header named `X-REST-API-Key,` which will be used internally to validate requests with tokens

The token is used to differentiate the Quality of Service (QoS) offered by the API. Users that provide a token in a service request can process multiple concurrent jobs. However, users that do not provide a token can process only one job at a time. Since every job takes 10 seconds to execute, users that provide no token will be able to execute only one call to the service every 10 seconds. For users that provide a token, processing will be much faster.

This differentiation is made possible by the use of two different Managed Executor Services, one for each type of request.

Running the jobs Example

After configuring GlassFish Server by adding two Managed Executor Services, you can use either NetBeans IDE or Maven to build, package, deploy, and run the `jobs` example.

To Configure GlassFish Server for the Basic Concurrency Example

To configure GlassFish Server, follow these steps.

1. Open the Administration Console at http://localhost:4848.
2. Expand the Resources node.
3. Expand the Concurrent Resources node.
4. Click Managed Executor Services.
5. On the Managed Executor Services page, click New to open the New Managed Executor Services page.
6. In the JNDI Name field, enter `MES_High` to create the high-priority Managed Executor Service. Use the following settings (keep the default values for other settings):
    - Thread Priority: 10
    - Core Size: 2
    - Maximum Pool Size: 5
    - Task Queue Capacity: 2
7. Click OK.
8. On the On the Managed Executor Services page, click New again.
9. In the JNDI Name field, enter `MES_Low` to create the low-priority Managed Executor Service. Use the following settings (keep the default values for other settings):
    - Thread Priority: 1
    - Core Size: 1
    - Maximum Pool Size: 1
    - Task Queue Capacity: 0
10. Click OK.

To Build, Package, and Deploy the jobs Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

```
jakartaee-examples/tutorial/concurrency
```

4. Select the `jobs` folder.
5. Click Open Project.
6. In the Projects tab, right-click the `jobs` project and select Build.

   This command builds and deploys the application.

To Build, Package, and Deploy the jobs Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/concurrency/jobs
   ```

3. Enter the following command to build and deploy the application:

   ```
   mvn install
   ```

To Run the jobs Example and Submit Jobs with Low Priority

To run the example as a user who submits jobs with low priority, follow these steps:

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/jobs
   ```

2. In the Jobs Client page, enter the value 1 in the Enter a JobID field, enter nothing in the Enter a Token field, then click Submit Job.

   The following message should be displayed at the bottom of the page:

   ```
   Job 1 successfully submitted
   ```

   The server log includes the following messages:

   ```
   INFO:   Invalid or missing token!
   INFO:   Task started LOW-1
   INFO:   Job 1 successfully submitted
   INFO:   Task finished LOW-1
   ```

   You submitted a job with low priority. This means that you cannot submit another job for 10 seconds. If you try to do so, the RESTful API will return a service unavailable (HTTP 503) response and the following message will be displayed at the bottom of the page:

```
Job 2 was NOT submitted
```

The server log will include the following messages:

```
INFO:   Invalid or missing token!
INFO:   Job 1 successfully submitted
INFO:   Task started LOW-1
INFO:   Invalid or missing token!
INFO:   Job 2 was NOT submitted
INFO:   Task finished LOW-1
```

To Run the jobs Example and Submit Jobs with High Priority

To run the example as a user who submits jobs with high priority, follow these steps:

1. In a web browser, enter the following URL:

   ```
   http://localhost:8080/jobs
   ```

2. In the Jobs Client page, enter a value of one to ten digits in the Enter a JobID field.
3. Click the here link on the line "Get a token here" to get a token. The page that displays the token will open in a new tab.
4. Copy the token and return to the Jobs Client page.
5. Paste the token in the Enter a Token field, then click Submit Job.

   A message like the following should be displayed at the bottom of the page:

   ```
   Job 11 successfully submitted
   ```

   The server log includes the following messages:

   ```
   INFO:   Token accepted. Execution with high priority.
   INFO:   Task started HIGH-11
   INFO:   Job 11 successfully submitted
   INFO:   Task finished HIGH-11
   ```

   You submitted a job with high priority. This means that you can submit multiple jobs, each with a token, and not face the 10 second per job restriction that the low priority submitters face. If you submit 3 jobs with tokens in rapid succession, messages like the following will be displayed at the bottom of the page:

   ```
   Job 1 was submitted
   Job 2 was submitted
   ```

```
Job 3 was submitted
```

The server log will include the following messages:

```
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-1
INFO:   Job 1 successfully submitted
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-2
INFO:   Job 2 successfully submitted
INFO:   Task finished HIGH-1
INFO:   Token accepted. Execution with high priority.
INFO:   Task started HIGH-3
INFO:   Job 3 successfully submitted
INFO:   Task finished HIGH-2
INFO:   Task finished HIGH-3
```

The taskcreator Concurrency Example

The `taskcreator` example demonstrates how to use Jakarta Concurrency to run tasks immediately, periodically, or after a fixed delay. This example provides a Jakarta Faces interface that enables users to submit tasks to be executed and displays information messages for each task. The example uses the Managed Executor Service to run tasks immediately and the Managed Scheduled Executor Service to run tasks periodically or after a fixed delay. (See Main Components of the Concurrency Utilities for information about these services.)

The `taskcreator` example consists of the following components.

- A Jakarta Faces page (`index.xhtml`) that contains three elements: a form to submit tasks, a task execution log, and a form to cancel periodic tasks. This page submits Ajax requests to create and cancel tasks. This page also receives WebSocket messages, using JavaScript code to update the task execution log.
- A CDI managed bean (`TaskCreatorBean`) that processes the requests from the Jakarta Faces page. This bean invokes the methods in `TaskEJB` to submit new tasks and to cancel periodic tasks.
- An enterprise bean (`TaskEJB`) that obtains executor service instances using resource injection and submits tasks for execution. This bean is also a Jakarta RESTful web service endpoint. The tasks send information messages to this endpoint.
- A WebSocket endpoint (`InfoEndpoint`) that the enterprise bean uses to send information messages to the clients.
- A task class (`Task`) that implements the `Runnable` interface. The `run` method in this class sends information messages to the web service endpoint in `TaskEJB` and sleeps for 1.5 seconds.

Figure 42, “Architecture of the taskcreator Example” shows the architecture of the `taskcreator` example.

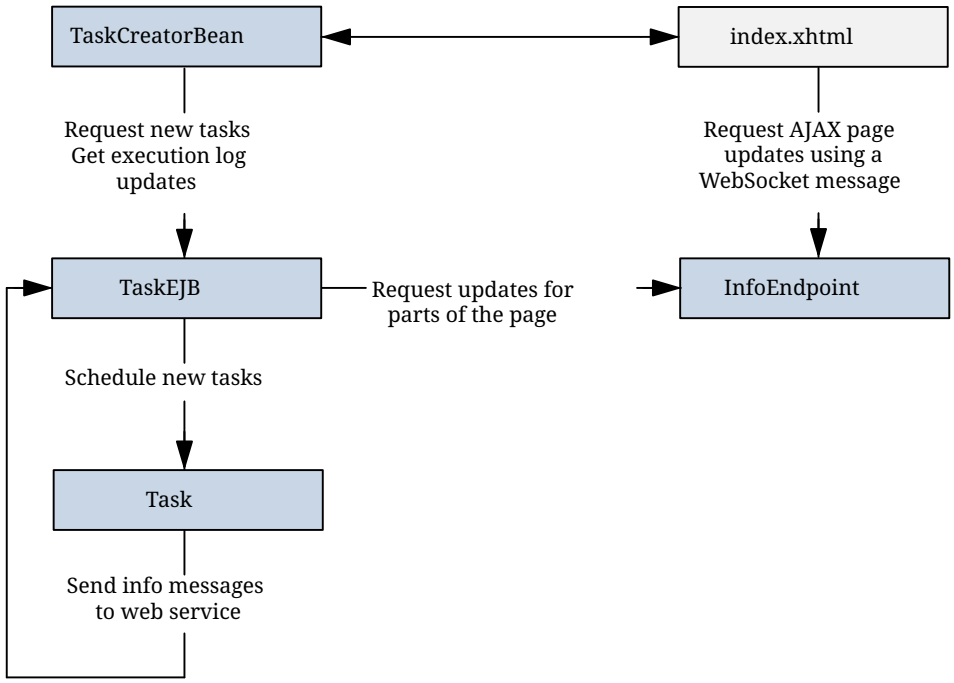


Figure 42. Architecture of the taskcreator Example

The `TaskEJB` class obtains the default executor service objects from the application server as follows:

```
@Resource(name="java:comp/DefaultManagedExecutorService")
ManagedExecutorService mExecService;

@Resource(name="java:comp/DefaultManagedScheduledExecutorService")
ManagedScheduledExecutorService sExecService;
```

The `submitTask` method in `TaskEJB` uses these objects to submit tasks for execution as follows:

```
public void submitTask(Task task, String type) {
    /* Use the managed executor objects from the app server */
    switch (type) {
        case "IMMEDIATE":
            mExecService.submit(task);
            break;
        case "DELAYED":
            sExecService.schedule(task, 3, TimeUnit.SECONDS);
            break;
        case "PERIODIC":
            ScheduledFuture<?> fut;
            fut = sExecService.scheduleAtFixedRate(task, 0, 8,
                    TimeUnit.SECONDS);
            periodicTasks.put(task.getName(), fut);
            break;
    }
}
```

For periodic tasks, `TaskEJB` keeps a reference to the `ScheduledFuture` object, so that the user can cancel the task at any time.

Running the taskcreator Example

This section describes how to build, package, deploy, and run the `taskcreator` example using NetBeans IDE or Maven.

To Build, Package, and Deploy the taskcreator Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

   ```
   jakartaee-examples/tutorial/concurrency
   ```

4. Select the `taskcreator` folder.
5. Click Open Project.
6. In the Projects tab, right-click the `taskcreator` project and select Build.

   This command builds and deploys the application.

To Build, Package, and Deploy the taskcreator Example Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   jakartaee-examples/tutorial/concurrency/taskcreator
   ```

3. Enter the following command to build and deploy the application:

   ```
   mvn install
   ```

To Run the taskcreator Example

1. Open the following URL in a web browser:

   ```
   http://localhost:8080/taskcreator/
   ```

   The page contains a form to submit tasks, a task execution log, and a form to cancel periodic tasks.

2. Select the Immediate task type, enter a task name, and click the Submit button. Messages like the following appear in the task execution log:

```
12:40:47 - IMMEDIATE Task TaskA finished
12:40:45 - IMMEDIATE Task TaskA started
```

3. Select the Delayed (3 sec) task type, enter a task name, and click the Submit button. Messages like the following appear in the task execution log:

```
12:43:26 - DELAYED Task TaskB finished
12:43:25 - DELAYED Task TaskB started
12:43:22 - DELAYED Task TaskB submitted
```

4. Select the Periodic (8 sec) task type, enter a task name, and click the Submit button. Messages like the following appear in the task execution log:

```
12:45:25 - PERIODIC Task TaskC finished run #2
12:45:23 - PERIODIC Task TaskC started run #2
12:45:17 - PERIODIC Task TaskC finished run #1
12:45:15 - PERIODIC Task TaskC started run #1
```

   You can add more than one periodic task. To cancel a periodic task, select it from the form and click Cancel Task.

Further Information about Jakarta Concurrency

For more information about concurrency, see

- Jakarta Concurrency 3.0 specification: https://jakarta.ee/specifications/concurrency/3.0/
- Concurrency Lesson in The Java Tutorials: https://docs.oracle.com/javase/tutorial/essential/concurrency/

Jakarta EE Platform

Jakarta Connectors

Jakarta Connectors mediate communications between Jakarta EE servers and EIS systems

Jakarta Connectors Specification

Optional Components

Jakarta Expression Language



This section was last updated for Jakarta EE Version 9.1

This chapter introduces the Expression Language (also referred to as the EL), which provides an

important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans). The EL is used by several Jakarta EE technologies, such as Jakarta Faces technology, Jakarta Server Pages technology, and Dependency Injection for Jakarta EE (CDI). The EL can also be used in stand-alone environments. This chapter only covers the use of the EL in Jakarta EE containers.

Overview of the EL

The EL allows page authors to use simple expressions to dynamically access data from JavaBeans components. For example, the `test` attribute of the following conditional tag is supplied with an EL expression that compares 0 with the number of items in the session-scoped bean named `cart`.

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
  ...
</c:if>
```

See Using the EL to Reference Managed Beans for more information on how to use the EL in Jakarta Faces applications.

To summarize, the EL provides a way to use simple expressions to perform the following tasks:

- Dynamically read application data stored in JavaBeans components, various data structures, and implicit objects
- Dynamically write data, such as user input into forms, to JavaBeans components
- Invoke arbitrary static and public methods
- Dynamically perform arithmetic, boolean, and string operations
- Dynamically construct collection objects and perform operations on collections

In a Jakarta Faces page, an EL expression can be used either in static text or in the attribute of a custom tag or standard action.

Finally, the EL provides a pluggable API for resolving expressions so that custom resolvers that can handle expressions not already supported by the EL can be implemented.

Immediate and Deferred Evaluation Syntax

The EL supports both immediate and deferred evaluation of expressions. Immediate evaluation means that the expression is evaluated and the result returned as soon as the page is first rendered. Deferred evaluation means that the technology using the expression language can use its own machinery to evaluate the expression sometime later during the page's lifecycle, whenever it is appropriate to do so.

Those expressions that are evaluated immediately use the `${}` syntax. Expressions whose evaluation is deferred use the `#{}` syntax.

Because of its multiphase lifecycle, Jakarta Faces technology uses mostly deferred evaluation expressions. During the lifecycle, component events are handled, data is validated, and other tasks

are performed in a particular order. Therefore, a Jakarta Faces implementation must defer evaluation of expressions until the appropriate point in the lifecycle.

Other technologies using the EL might have different reasons for using deferred expressions.

Immediate Evaluation

All expressions using the `${}` syntax are evaluated immediately. These expressions can appear as part of a template (static) text or as the value of a tag attribute that can accept runtime expressions.

The following example shows a tag whose `value` attribute references an immediate evaluation expression that updates the quantity of books retrieved from the backing bean named `catalog`:

```
<h:outputText value="${catalog.bookQuantity}" />
```

The Jakarta Faces implementation evaluates the expression `${catalog.bookQuantity}`, converts it, and passes the returned value to the tag handler. The value is updated on the page.

Deferred Evaluation

Deferred evaluation expressions take the form `#{expr}` and can be evaluated at other phases of a page lifecycle as defined by whatever technology is using the expression. In the case of Jakarta Faces technology, its controller can evaluate the expression at different phases of the lifecycle, depending on how the expression is being used in the page.

The following example shows a Jakarta Faces `h:inputText` tag, which represents a field component into which a user enters a value. The `h:inputText` tag's `value` attribute references a deferred evaluation expression that points to the `name` property of the `customer` bean:

```
<h:inputText id="name" value="#{customer.name}" />
```

For an initial request of the page containing this tag, the Jakarta Faces implementation evaluates the `#{customer.name}` expression during the render-response phase of the lifecycle. During this phase, the expression merely accesses the value of `name` from the `customer` bean, as is done in immediate evaluation.

For a postback request, the Jakarta Faces implementation evaluates the expression at different phases of the lifecycle, during which the value is retrieved from the request, validated, and propagated to the `customer` bean.

As shown in this example, deferred evaluation expressions can be

- Value expressions that can be used to both read and write data
- Method expressions

Value expressions (both immediate and deferred) and method expressions are explained in the next section.

Value and Method Expressions

The EL defines two kinds of expressions: value expressions and method expressions. Value expressions can be evaluated to yield a value, and method expressions are used to reference a method.

Value Expressions

Value expressions can be further categorized into *rvalue* and *lvalue* expressions. An *lvalue* expression can specify a target, such as an object, a bean property, or elements of a collection, that can be assigned a value. An *rvalue* expression cannot specify such a target.

All expressions that are evaluated immediately use the `${}` delimiters, and although the expression can be an *lvalue* expression, no assignments will ever happen. Expressions whose evaluation can be deferred use the `#{}` delimiters and can act as both *rvalue* and *lvalue* expressions; if the expression is an *lvalue* expression, it can be assigned a new value. Consider the following two value expressions:

```
${customer.name}

#{customer.name}
```

The former uses immediate evaluation syntax, whereas the latter uses deferred evaluation syntax. The first expression accesses the `name` property, gets its value, and passes the value to the tag handler. With the second expression, the tag handler can defer the expression evaluation to a later time in the page lifecycle if the technology using this tag allows.

In the case of Jakarta Faces technology, the latter tag's expression is evaluated immediately during an initial request for the page. During a postback request, this expression can be used to set the value of the `name` property with user input.

Referencing Objects

A top-level identifier (such as `customer` in the expression `customer.name`) can refer to the following objects:

- Lambda parameters
- EL variables
- Managed beans
- Implicit objects
- Classes of static fields and methods

To refer to these objects, you write an expression using a variable that is the name of the object. The following expression references a managed bean called `customer`:

```
${customer}
```

You can use a custom EL resolver to alter the way variables are resolved. For instance, you can provide an EL resolver that intercepts objects with the name `customer`, so that `${customer}` returns a value in the EL resolver instead. (Jakarta Faces technology uses an EL resolver to handle managed beans.)

An `enum` constant is a special case of a static field, and you can reference such a constant directly. For example, consider this `enum` class:

```
public enum Suit {hearts, spades, diamonds, clubs}
```

In the following expression, in which `mySuit` is an instance of `Suit`, you can compare `suit.hearts` to the instance:

```
${mySuit == suit.hearts}
```

Referencing Object Properties or Collection Elements

To refer to properties of a bean, static fields or methods of a class, or items of a collection, you use the `.` or `[]` notation. The same syntax can be used for attributes of an implicit object, because attributes are placed in a map.

To reference the `name` property of the `customer` bean, use either the expression `${customer.name}` or the expression `${customer["name"]}`. Here, the part inside the brackets is a `String` literal that is the name of the property to reference. The `[]` syntax is more general than the `.` syntax, because the part inside the brackets can be any `String` expression, not just literals.

You can use double or single quotes for the `String` literal. You can also combine the `[]` and `.` notations, as shown here:

```
${customer.address["street"]}
```

You can reference a static field or method using the syntax classname.field, as in the following example:

```
Boolean.FALSE
```

The classname is the name of the class without the package name. By default, all the `java.lang` packages are imported. You can import other packages, classes, and static fields as needed.

If you are accessing an item in an array or list, you must use the `[]` notation and specify an index in the array or list. The index is an expression that can be converted to `int`. The following example references the first of the customer orders, assuming that `customer.orders` is a `List`:

```
${customer.orders[1]}
```

If you are accessing an item in a `Map`, you must specify the key for the `Map`. If the key is a `String` literal, the dot (`.`) notation can be used. Assuming that `customer.orders` is a `Map` with a `String` key, the following examples reference the item with the key `"socks"`:

```
${customer.orders["socks"]}

${customer.orders.socks}
```

Referencing Literals

The EL defines the following literals:

- Boolean: `true` and `false`
- Integer: As in Java
- Floating-point: As in Java
- String: With single and double quotes; `"` is escaped as `\"`, `'` is escaped as `\'`, and `\` is escaped as `\\`
- Null: `null`

Here are some examples:

- `${"literal"}`
- `${true}`
- `${57}`

Parameterized Method Calls

The EL offers support for parameterized method calls.

Both the `.` and `[]` operators can be used for invoking method calls with parameters, as shown in the following expression syntax:

- `expr-a[expr-b](parameters)`
- `expr-a.identifier-b(parameters)`

In the first expression syntax, *expr-a* is evaluated to represent a bean object. The expression *expr-b* is evaluated and cast to a string that represents a method in the bean represented by *expr-a*. In the second expression syntax, *expr-a* is evaluated to represent a bean object, and *identifier-b* is a string that represents a method in the bean object. The *parameters* in parentheses are the arguments for the method invocation. Parameters can be zero or more values of expressions, separated by commas.

Parameters are supported for both value expressions and method expressions. In the following example, which is a modified tag from the `guessnumber` application, a random number is provided as an argument rather than from user input to the method call:

```
<h:inputText value="#{userNumberBean.userNumber('5')}">
```

The preceding example uses a value expression.

Consider the following example of a Jakarta Faces component tag that uses a method expression:

```
<h:commandButton action="#{trader.buy}" value="buy"/>
```

The EL expression `trader.buy` calls the `trader` bean's `buy` method. You can modify the tag to pass on a parameter. Here is the revised tag in which a parameter is passed:

```
<h:commandButton action="#{trader.buy('SOMESTOCK')}" value="buy"/>
```

In the preceding example, you are passing the string `'SOMESTOCK'` (a stock symbol) as a parameter to the `buy` method.

Where Value Expressions Can Be Used

Value expressions using the `${}` delimiters can be used

- In static text
- In any standard or custom tag attribute that can accept an expression

The value of an expression in static text is computed and inserted into the current output. Here is an example of an expression embedded in static text:

```
<some:tag>
    some text ${expr} some text
</some:tag>
```

A tag attribute can be set in the following ways.

- With a single expression construct:

  ```
  <some:tag value="${expr}"/>

  <another:tag value="#{expr}"/>
  ```

  These expressions are evaluated, and the result is converted to the attribute's expected type.

- With one or more expressions separated or surrounded by text:

  ```
  <some:tag value="some${expr}${expr}text${expr}"/>
  ```

```
<another:tag value="some#{expr}#{expr}text#{expr}"/>
```

These kinds of expression, called composite expressions, are evaluated from left to right. Each expression embedded in the composite expression is converted to a `String` and then concatenated with any intervening text. The resulting `String` is then converted to the attribute's expected type.

- With text only:

```
<some:tag value="sometext"/>
```

The attribute's `String` value is converted to the attribute's expected type.

You can use the string concatenation operator `+=` to create a single expression from what would otherwise be a composite expression. For example, you could change the composite expression

```
<some:tag value="sometext ${expr} moretext"/>
```

to

```
<some:tag value="${sometext += expr += moretext}"/>
```

All expressions used to set attribute values are evaluated in the context of an expected type. If the result of the expression evaluation does not match the expected type exactly, a type conversion will be performed. For example, the expression `${1.2E4}` provided as the value of an attribute of type `float` will result in the following conversion:

```
Float.valueOf("1.2E4").floatValue()
```

Method Expressions

Another feature of the EL is its support of deferred method expressions. A method expression is used to refer to a public method of a bean and has the same syntax as an *lvalue* expression.

In Jakarta Faces technology, a component tag represents a component on a page. The component tag uses method expressions to specify methods that can be invoked to perform some processing for the component. These methods are necessary for handling events that the components generate and for validating component data, as shown in this example:

```
<h:form>
    <h:inputText id="name"
                 value="#{customer.name}"
                 validator="#{customer.validateName}"/>
    <h:commandButton id="submit"
                     action="#{customer.submit}" />
```

```
</h:form>
```

The `h:inputText` tag displays as a field. The `validator` attribute of this `h:inputText` tag references a method, called `validateName`, in the bean, called `customer`.

Because a method can be invoked during different phases of the lifecycle, method expressions must always use the deferred evaluation syntax.

Like lvalue expressions, method expressions can use the `.` and the `[]` operators. For example, `#{object.method}` is equivalent to `#{object["method"]}`. The literal inside the `[]` is converted to `String` and is used to find the name of the method that matches it.

Method expressions can be used only in tag attributes and only in the following ways:

- With a single expression construct, where bean refers to a JavaBeans component and method refers to a method of the JavaBeans component:

  ```
  <some:tag value="#{bean.method}"/>
  ```

  The expression is evaluated to a method expression, which is passed to the tag handler. The method represented by the method expression can then be invoked later.

- With text only:

  ```
  <some:tag value="sometext"/>
  ```

  Method expressions support literals primarily to support `action` attributes in Jakarta Faces technology. When the method referenced by this method expression is invoked, the method returns the `String` literal, which is then converted to the expected return type, as defined in the tag's tag library descriptor.

Lambda Expressions

A lambda expression is a value expression with parameters. The syntax is similar to that of the lambda expression in the Java programming language, except that in the EL, the body of the lambda expression is an EL expression.

For basic information on lambda expressions, see https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.



Lambda expressions are part of Java SE 8

A lambda expression uses the arrow token (`->`) operator. The identifiers to the left of the operator are called lambda parameters. The body, to the right of the operator, must be an EL expression. The lambda parameters are enclosed in parentheses; the parentheses can be omitted if there is only one parameter. Here are some examples:

```
x -> x+1
(x, y) -> x + y
() -> 64
```

A lambda expression behaves like a function. It can be invoked immediately. For example, the following invocation evaluates to 7:

```
((x, y) -> x + y)(3, 4)
```

You can use a lambda expression in conjunction with the assignment and semicolon operators. For example, the following code assigns the previous lambda expression to a variable and then invokes it. The result is again 7:

```
v = (x, y) -> x + y; v(3, 4)
```

A lambda expression can also be passed as an argument to a method and be invoked in the method. It can also be nested in another lambda expression.

Operations on Collection Objects

The EL supports operations on collection objects: sets, lists, and maps. It allows the dynamic creation of collection objects, which can then be operated on using streams and pipelines.



Like lambda expressions, operations on collection objects are part of Java SE 8.

For example, you can construct a set as follows:

```
{1,2,3}
```

You can construct a list as follows; a list can contain various types of items:

```
[1,2,3]
[1, "two", [three,four]]
```

You can construct a map by using a colon to define the entries, as follows:

```
{"one":1, "two":2, "three":3}
```

You operate on collection objects using method calls to the stream of elements derived from the collection. Some operations return another stream, which allows additional operations. Therefore, you can chain these operations together in a pipeline.

A stream pipeline consists of the following:

- A source (the `Stream` object)
- Any number of intermediate operations that return a stream (for example, `filter` and `map`)
- A terminal operation that does not return a stream (for example, `toList()`)

The `stream` method obtains a `Stream` from a `java.util.Collection` or a Java array. The stream operations do not modify the original collection object.

For example, you might generate a list of titles of history books as follows:

```
books.stream().filter(b->b.category == "history")
              .map(b->b.title)
              .toList()
```

The following simpler example returns a sorted version of the original list:

```
[1,3,5,2].stream().sorted().toList()
```

Streams and stream operations are documented in the Java SE 8 API documentation, available at https://docs.oracle.com/javase/8/docs/api/. The following subset of operations is supported by the EL:

`allMatch`	`anyMatch`	`average`	`count`
`distinct`	`filter`	`findFirst`	`flatMap`
`forEach`	`iterator`	`limit`	`map`
`max`	`min`	`noneMatch`	`peek`
`reduce`	`sorted`	`substream`	`sum`
`toArray`	`toList`		

See the Expression Language specification at https://jakarta.ee/specifications/expression-language/4.0/ for details on these operations.

Operators

In addition to the `.` and `[]` operators discussed in Value and Method Expressions, the EL provides the following operators, which can be used in rvalue expressions only.

- Arithmetic: `+`, `-` (binary), `*`, `/` and `div`, `%` and `mod`, `-` (unary).
- String concatenation: `+=`.
- Logical: `and`, `&&`, `or`, `||`, `not`, `!`.
- Relational: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`. Comparisons can be made against other values or against Boolean, string, integer, or floating-point literals.
- Empty: The `empty` operator is a prefix operation that can be used to determine whether a value is `null` or empty.
- Conditional: `A ? B : C`. Evaluate `B` or `C`, depending on the result of the evaluation of `A`.

- Lambda expression: `->`, the arrow token.
- Assignment: `=`.
- Semicolon: `;`.

The precedence of operators, highest to lowest, left to right, is as follows:

- `[] .`
- `()` (used to change the precedence of operators)
- `-` (unary) `not ! empty`
- `* / div % mod`
- `+ -` (binary)
- `+=`
- `<> <= >= lt gt le ge`
- `== != eq ne`
- `&& and`
- `|| or`
- `? :`
- `->`
- `=`
- `;`

Reserved Words

The following words are reserved for the EL and should not be used as identifiers:

`and`	`or`	`not`	`eq`
`ne`	`lt`	`gt`	`le`
`ge`	`true`	`false`	`null`
`instanceof`	`empty`	`div`	`mod`

Examples of EL Expressions

Example Expressions contains example EL expressions and the result of evaluating them.

Example Expressions

EL Expression	Result
`${1 > (4/2)}`	`false`
`${4.0 >= 3}`	`true`
`${100.0 == 100}`	`true`
`${(10*10) ne 100}`	`false`
`${'a' > 'b'}`	`false`

EL Expression	Result
`${'hip' lt 'hit'}`	`true`
`${4 > 3}`	`true`
`${1.2E4 + 1.4}`	`12001.4`
`${3 div 4}`	`0.75`
`${10 mod 4}`	`2`
`${((x, y) -> x + y)(3, 5.5)}`	`8.5`
`[1,2,3,4].stream().sum()`	`10`
`[1,3,5,2].stream().sorted().toList()`	`[1, 2, 3, 5]`
`${!empty param.Add}`	`False` if the request parameter named `Add` is `null` or an empty string
`${pageContext.request.contextPath}`	The context path
`${sessionScope.cart.numberOfItems}`	The value of the `numberOfItems` property of the session-scoped attribute named `cart`
`${param['mycom.productId']}`	The value of the request parameter named `mycom.productId`
`${header["host"]}`	The host
`${departments[deptName]}`	The value of the entry named `deptName` in the `departments` map
`${requestScope['jakarta.servlet.forward.servlet_path']}`	The value of the request-scoped attribute named `jakarta.servlet.forward.servlet_path`
`#{customer.lName}`	Gets the value of the property `lName` from the `customer` bean during an initial request; sets the value of `lName` during a postback
`#{customer.calcTotal}`	The return value of the method `calcTotal` of the `customer` bean

Further Information about the Expression Language

For more information about the Expression Language, see

- The Expression Language 5.0 specification: https://jakarta.ee/specifications/expression-language/5.0/
- The EL specification website: https://github.com/eclipse-ee4j/el-ri/tree/master/spec

Jakarta XML Binding

Archived

Web Profile

Jakarta Pages

Jakarta Pages was previously known as "JSP".

It has been archived in favor of Facelets.

- JavaServer Pages Technology
- JavaServer Pages Documents
- Custom Tags in JSP Pages

See also the Jakarta Pages Specification.

Jakarta Tags

Jakarta Tags was previously known as "JSTL".

A subset of Jakarta Tags has been archived in favor of Jakarta XML Binding, Jakarta Faces and Jakarta Persistence. The Core Tag Library and JSTL Functions are still available in Facelets.

- XML Tag Library
- Internationalization Tag Library
- SQL Tag Library

See also the Jakarta Tags Specification.

Jakarta EE Platform

XML Web Services

You can find information about Jakarta XML Web Services in a previous version of the tutorial:

- https://jakarta.ee/learn/docs/jakartaee-tutorial/9.1/websvcs/webservices-intro/webservices-intro.html
- https://jakarta.ee/learn/docs/jakartaee-tutorial/9.1/websvcs/jaxws/jaxws.html

Jakarta Enterprise Beans Full

You can find information about Jakarta Enterprise Beans Full in a previous version of the tutorial:

- https://jakarta.ee/learn/docs/jakartaee-tutorial/9.1/entbeans/ejb-embedded/ejb-embedded.html