PART V

Contexts and Dependency Injection for the Java EE Platform

Part V explores Contexts and Dependency Injection for the Java EE Platform. This part contains the following chapters:

CHAPTER 28

Introduction to Contexts and Dependency Injection for the Java EE Platform

Contexts and Dependency Injection (CDI) for the Java EE platform is one of several Java EE 6 features that help to knit together the web tier and the transactional tier of the Java EE platform. CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with JavaServer 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 is specified by JSR 299, formerly known as Web Beans. Related specifications that CDI uses include the following:

- JSR 330, Dependency Injection for Java
- The Managed Beans specification, which is an offshoot of the Java EE 6 platform specification (JSR 316)

The following topics are addressed here:

Overview of CDI

The most fundamental services provided by CDI are as follows:

- **Contexts**: The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts
- **Dependency injection**: The ability to inject components into an application in a typesafe way, including the ability 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 JavaServer Faces page or a JavaServer 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 Java Servlet specification
- A complete Service Provider Interface (SPI) that allows third-party frameworks to integrate cleanly in the Java EE 6 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 doing the following:
  - 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 Java 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

About Beans

CDI redefines the concept of a *bean* beyond its use in other Java technologies, such as the JavaBeans and Enterprise JavaBeans (EJB) technologies. In CDI, a bean is a source of contextual objects that define application state and/or logic. A Java 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" on page 519)
- A scope (see "Using Scopes" on page 520)
- Optionally, a bean EL name (see "Giving Beans EL Names" on page 522)
- 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 Java EE technology specification, such as the JavaServer 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 EJB component-defining annotation or declared as an EJB 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
- Java EE resources: data sources, Java Message Service topics, queues, connection factories, and the like
- Persistence contexts (JPA `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" on page 522 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, you would 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 java.lang.annotation.Target;

import javax.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, the bean automatically has the qualifier `@Default`. The unannotated `Greeting` class could be declared as follows:

```
package greetings;

import javax.enterprise.inject.Default;

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

Injecting Beans

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

```
import javax.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 javax.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 JavaServer Faces application, the bean needs to be accessible through the EL.

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 Table 28–1, depending on how you are using it.

TABLE 28–1 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).

TABLE 28–1 Scopes *(Continued)*

Scope	Annotation	Duration
Conversation	`@ConversationScoped`	A user's interaction with a JavaServer Faces application, within explicit developer-controlled boundaries that extend the scope across multiple invocations of the JavaServer Faces lifecycle. 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 JSR 299 and the JavaServer Faces API. The last two are defined by JSR 299.

All predefined scopes except `@Dependent` are contextual scopes. CDI places beans of contextual scope in the context whose lifecycle is defined by the Java 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.

Java 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 JavaBeans 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

If, however, you create a Java EE component that is a managed bean, 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 javax.inject.Inject;
import javax.enterprise.context.RequestScoped;

@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.

Overriding the Scope of a Bean at the Point of Injection

Overriding the scope of a bean at the point of injection enables an application to request a new instance of the bean with the default scope `@Dependent`. The `@Dependent` scope specifies that the bean's lifecycle is the lifecycle of the object into which the bean is injected. The CDI container provides no other lifecycle management for the instance.

Note – The effects of overriding the scope of a bean may be unpredictable or undesirable, especially if the overridden scope is `@Request` or `@Session`.

To override the scope of a bean at the point of injection, use the `javax.enterprise.inject.New` annotation instead of the `@Inject` annotation. For more information on the `@Inject` annotation, see "Injecting Beans" on page 520.

Giving Beans EL Names

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

```
import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.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, you need to 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 javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.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, you typically create a form that uses user interface elements to call its methods and display their results. This 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`, 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, 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

An application that uses CDI must have a file named `beans.xml`. The file can be completely empty (it has content only in certain limited situations), but it must be present. For a web application, the `beans.xml` file must be in the `WEB-INF` directory. For EJB modules or JAR files, the `beans.xml` file 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 annotation are defined in JSR 250: Common Annotations for the Java platform (`http://jcp.org/en/jsr/detail?id=250`).

▼ 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 `javax.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.

Note – As mandated in JSR 250, 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 Example" on page 531 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 `javax.annotation.PreDestroy` annotation.**

   CDI calls this method before starting to destroy the bean.

Further Information about CDI

For more information about CDI for the Java EE platform, see

- Contexts and Dependency Injection for the Java EE platform specification:

  `http://jcp.org/en/jsr/detail?id=299`
- An introduction to Contexts and Dependency Injection for the Java EE platform:

  `http://docs.jboss.org/weld/reference/latest/en-US/html/`
- Dependency Injection for Java specification:

  `http://jcp.org/en/jsr/detail?id=330`
- Managed Beans specification, which is part of the Java Platform, Enterprise Edition 6 (Java EE 6) Specification:

  `http://jcp.org/en/jsr/detail?id=316`

CHAPTER 29

Running the Basic Contexts and Dependency Injection Examples

This chapter describes in detail how to build and run simple examples that use CDI. The examples are in the *tut-install*/`examples/cdi/` directory:

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

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

Each example has a `build.xml` file that refers to files in the *tut-install*/`examples/bp-project/` directory.

See Chapter 2, "Using the Tutorial Examples," for basic information on installing, building, and running the examples.

The following topics are addressed here:

- "The `simplegreeting` CDI Example" on page 527
- "The `guessnumber` CDI Example" on page 531

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 JavaServer 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" on page 518
- The `@Informal` qualifier interface definition and the `InformalGreeting` class that implements the interface, both shown in "Using Qualifiers" on page 519
- The `Printer` managed bean class, which injects one of the two interfaces, shown in full in "Adding Setter and Getter Methods" on page 523

The source files are located in the *tut-install*`/examples/cdi/simplegreeting/src/java/greetings/` directory.

The Facelets Template and Page

To use the managed bean in a simple Facelets application, you can use a very simple template file and `index.xhtml` page. The template page, `template.xhtml`, looks like this:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <h:outputStylesheet library="css" name="default.css"/>
        <title>
            <ui:insert name="title">Default Title</ui:insert>
        </title>
    </h:head>

    <body>
        <div id="container">
            <div id="header">
                <h2><ui:insert name="head">Head</ui:insert></h2>
            </div>

            <div id="space">
                <p></p>
            </div>

            <div id="content">
                <ui:insert name="content"/>
            </div>
        </div>
    </body>
</html>
```

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

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
    <ui:composition template="/template.xhtml">

        <ui:define name="title">Simple Greeting</ui:define>
        <ui:define name="head">Simple Greeting</ui:define>
        <ui:define name="content">
            <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>
        </ui:define>

    </ui:composition>
</html>
```

The form asks the user to type 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 typed by the user:

`Hello,` *name*`.`

`Hi,` *name*`!`

The Facelets page and template are located in the *tut-install*`/examples/cdi/simplegreeting/web/` directory.

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

tut-install`/examples/cdi/simplegreeting/web/resources/css/default.css`

Configuration Files

You must create an empty `beans.xml` file to indicate to GlassFish Server that your application is a CDI application. This file can have content in some situations, but not in simple applications like this one.

Your application also needs the basic web application deployment descriptors `web.xml` and `glassfish-web.xml`. These configuration files are located in the *tut-install*`/examples/cdi/simplegreeting/web/WEB-INF/` directory.

Running the simplegreeting Example

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

▼ To Build, Package, and Deploy the simplegreeting Example Using NetBeans IDE

This procedure builds the application into the *tut-install*`/examples/cdi/simplegreeting/build/web/` directory. The contents of this directory are deployed to the GlassFish Server.

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

   tut-install`/examples/cdi/`
3. **Select the `simplegreeting` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **(Optional) 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 edit pane, 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 Deploy.**

▼ To Build, Package, and Deploy the simplegreeting Example Using Ant

1. **In a terminal window, go to:**

   tut-install`/examples/cdi/simplegreeting/`

2 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `simplegreeting.war`, located in the `dist` directory.

3 **Type the following command:**

```
ant deploy
```

Typing this command deploys `simplegreeting.war` to the GlassFish Server.

▼ To Run the simplegreeting Example

1 **In a web browser, type the following URL:**

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

The Simple Greeting page opens.

2 **Type a name in the text field.**

For example, suppose that you type **`Duke`**.

3 **Click the Say Hello button.**

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

```
Hi, Duke!
```

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

```
Hello, Duke.
```

The guessnumber CDI Example

The `guessnumber` 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` example described in Chapter 5, "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 Source Files

The four source files for the `guessnumber` 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 *tut-install*`/examples/cdi/guessnumber/src/java/guessnumber/` directory.

The `@MaxNumber` and `@Random` Qualifier Interfaces

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

```
package guessnumber;

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.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.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 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.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.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 javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;

@ApplicationScoped
public class Generator implements Serializable {

    private static final long serialVersionUID = -7213673465118041882L;

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

    private int maxNumber = 100;

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

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

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

}
```

The `UserNumberBean` Managed Bean

The `UserNumberBean` managed bean, the managed bean for the JavaServer 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 javax.annotation.PostConstruct;
import javax.enterprise.context.SessionScoped;
import javax.enterprise.inject.Instance;
import javax.inject.Inject;
import javax.inject.Named;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;

@Named
@SessionScoped
public class UserNumberBean implements Serializable {

    private static final long serialVersionUID = -7698506329160109476L;
    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;

    public UserNumberBean() {
    }

    public int getNumber() {
        return number;
    }

    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public int getMaximum() {
        return (this.maximum);
    }
    public void setMaximum(int maximum) {
        this.maximum = maximum;
    }

    public int getMinimum() {
        return (this.minimum);
    }

    public void setMinimum(int minimum) {
        this.minimum = minimum;
    }

    public int getRemainingGuesses() {
        return remainingGuesses;
    }

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

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

    public void validateNumberRange(FacesContext context,
                                    UIComponent toValidate,
                                    Object value) {
        if (remainingGuesses <= 0) {
            FacesMessage message = new FacesMessage("No guesses left!");
            context.addMessage(toValidate.getClientId(context), message);
            ((UIInput) toValidate).setValid(false);
            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 Facelets Page

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

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
    <ui:composition template="/template.xhtml">

        <ui:define name="title">Guess My Number</ui:define>
        <ui:define name="head">Guess My Number</ui:define>
        <ui:define name="content">
            <h:form id="GuessMain">
                <div style="color: black; font-size: 24px;">
                    <p>I'm thinking of a number from
                    <span style="color: blue">#{userNumberBean.minimum}</span>
                    to
                    <span style="color: blue">#{userNumberBean.maximum}</span>.
                    You have
                    <span style="color: blue">#{userNumberBean.remainingGuesses}</span>
                    guesses.</p>
                </div>
                <h:panelGrid border="0" columns="5" style="font-size: 18px;">
                    <h:outputLabel for="inputGuess">Number:</h:outputLabel>
                    <h:inputText id="inputGuess"
                       value="#{userNumberBean.userNumber}"
                       required="true" size="3"
                       disabled="#{userNumberBean.number eq userNumberBean.userNumber}"
                       validator="#{userNumberBean.validateNumberRange}">
                    </h:inputText>
                    <h:commandButton id="GuessButton" value="Guess"
                       action="#{userNumberBean.check}"
                       disabled="#{userNumberBean.number eq userNumberBean.userNumber}"/>
                    <h:commandButton id="RestartButton" value="Reset"
                       action="#{userNumberBean.reset}"
                       immediate="true" />
                    <h:outputText id="Higher" value="Higher!"
rendered="#{userNumberBean.number gt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"
                       style="color: #d20005"/>
                    <h:outputText id="Lower" value="Lower!"
rendered="#{userNumberBean.number lt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"
                       style="color: #d20005"/>
                </h:panelGrid>
                <div style="color: #d20005; font-size: 14px;">
                    <h:messages id="messages" globalOnly="false"/>
                </div>
            </h:form>
        </ui:define>

    </ui:composition>
</html>
```

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 text field that appears if the guess is higher or lower than the correct number. Every time the user clicks the Guess button, 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 Example

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

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

This procedure builds the application into the *tut-install*`/examples/cdi/guessnumber/build/web/` directory. The contents of this directory are deployed to the GlassFish Server.

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

   tut-install`/examples/cdi/`
3. **Select the `guessnumber` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `guessnumber` project and select Deploy.**

▼ To Build, Package, and Deploy the guessnumber Example Using Ant

1. **In a terminal window, go to:**

   tut-install`/examples/cdi/guessnumber/`
2. **Type the following command:**

   ```
   ant
   ```

   This command calls the `default` target, which builds and packages the application into a WAR file, `guessnumber.war`, located in the `dist` directory.

3 **Type the following command:**

```
ant deploy
```

The `guessnumber.war` file will be deployed to the GlassFish Server.

▼ To Run the guessnumber Example

1 **In a web browser, type the following URL:**

```
http://localhost:8080/guessnumber
```

The Guess My Number page opens.

2 **Type a number in the Number text 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, the input field and Guess button are grayed out.

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

◆ ◆ ◆ CHAPTER 30

Contexts and Dependency Injection for the Java EE Platform: Advanced Topics

This chapter describes more advanced features of Contexts and Dependency Injection for the Java EE Platform. 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" on page 516.

The following topics are addressed here:

Using Alternatives in CDI Applications

When you have more than one version of a bean 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" on page 527.

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 like 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 `javax.enterprise.inject.Alternative` annotation and then use the `alternative` 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" on page 553 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>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.

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 `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 `javax.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" on page 524.

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 Java EE resources such as data sources, JMS resources, and web service references.

A producer method or field is annotated with the `javax.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" on page 558, the managed bean defines the following producer method:

```
@Produces
@Chosen
@RequestScoped
public Coder getCoder(@New TestCoderImpl tci,
        @New CoderImpl ci) {

    switch (coderType) {
        case TEST:
            return tci;
        case SHIFT:
            return ci;
        default:
            return null;
    }
}
```

The `javax.enterprise.inject.New` qualifier instructs the CDI runtime to instantiate both of the coder implementations and provide them as arguments to the producer method. 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 Java Persistence API `EntityManager`. 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" on page 561 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, then inject the object wherever in the application you need it.

Using a Disposer Method

You can use a producer method 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, if you used a producer method instead of a producer field to create the entity manager, you would create and close it as follows:

```
@PersistenceContext
private EntityManager em;

@Produces
@UserDatabase
public EntityManager create() {
    return em;
}

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 method, `create`.

Using Predefined Beans in CDI Applications

CDI provides predefined beans that implement the following interfaces:

`javax.transaction.UserTransaction`
A Java Transaction API (JTA) 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.

`javax.validation.Validator`
A validator for bean instances. The bean that implements this interface enables a `Validator` object for the default bean validation `ValidatorFactory` object to be injected.

`javax.validation.ValidatorFactory`
A factory class for returning initialized `Validator` instances. The bean that implements this interface enables the default bean validation object `ValidatorFactory` to be injected.

To inject a predefined bean, create an injection point by using the `javax.annotation.Resource` annotation to obtain an instance of the bean. For the bean type, specify the class name of the interface the bean implements.

Predefined beans are injected with dependent scope and the predefined default qualifier `@Default`.

For more information about injecting resources, see "Resource Injection" on page 807.

The following code snippet shows how to use the `@Resource` annotation to inject a predefined bean. This code snippet injects a user transaction into the servlet class `TransactionServlet`. The user transaction is an instance of the predefined bean that implements the `javax.transaction.UserTransaction` interface.

```
import javax.annotation.Resource;
import javax.servlet.http.HttpServlet;
import javax.transaction.UserTransaction;
...
public class TransactionServlet extends HttpServlet {
    @Resource UserTransaction transaction;
    ...
}
```

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. 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" on page 568, 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) {
    ...
}
```

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 non-transactional 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 non-transactional, the exception terminates processing of the event, and no other observer methods for the event are called.

Firing Events

To activate an event, call the `javax.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.

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 Java EE platform, they were specific to enterprise beans. On the Java EE 6 platform you can use them with Java EE managed objects of all kinds, including managed beans.

For information on Java EE interceptors, see Chapter 50, “Using Java 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 EJB 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 `javax.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" on page 572.

Every `@AroundInvoke` method takes a `javax.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>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.

Using Decorators in CDI Applications

A *decorator* is a Java class that is annotated javax.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 javax.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" on page 574 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>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.

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 where 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 `javax.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 {}
```

◆ ◆ ◆ CHAPTER 31

Running the Advanced Contexts and Dependency Injection Examples

This chapter describes in detail how to build and run several advanced examples that use CDI. The examples are in the *tut-install*`/examples/cdi/` directory.

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

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

Each example has a `build.xml` file that refers to files in the *tut-install*`/examples/bp-project/` directory.

See Chapter 2, "Using the Tutorial Examples," for basic information on installing, building, and running the examples.

The following topics are addressed here:

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" on page 539. The example includes an interface and two implementations of it, a managed bean, a Facelets page, and configuration files.

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, for Julius Caesar, who 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 javax.enterprise.inject.Alternative;

@Alternative
public class TestCoderImpl implements Coder {

    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>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 type 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="http://java.sun.com/jsf/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="Type a string: " for="inputString"/>
                <h:inputText id="inputString"
                             value="#{coderBean.inputString}"/>
                <h:outputLabel value="Type 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 types 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 Ant to build, package, deploy, and run the `encoder` application.

▼ To Build, Package, and Deploy the `encoder` Example Using NetBeans IDE

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

   tut-install`/examples/cdi/`
3. **Select the `encoder` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `encoder` project and select Deploy.**

▼ To Run the `encoder` Example Using NetBeans IDE

1. **In a web browser, type the following URL:**

   `http://localhost:8080/encoder`

   The String Encoder page opens.

2 **Type a string and the number of letters to shift by, then click Encode.**

The encoded string appears in blue on the Result line. For example, if you type `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 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>encoder.TestCoderImpl</class>
</alternatives>
```

d. **Save the file.**

4 **Right-click the `encoder` project and select Deploy.**

5 **In the web browser, retype the URL to show the String Encoder page for the redeployed project:**

```
http://localhost:8080/encoder/
```

6 **Type a string and the number of letters to shift by, then click Encode.**

This time, the Result line displays your arguments. For example, if you type `Java` and `4`, the result is:

```
Result: input string is Java, shift value is 4
```

▼ To Build, Package, and Deploy the `encoder` Example Using Ant

1 **In a terminal window, go to:**

tut-install`/examples/cdi/encoder/`

2 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `encoder.war`, located in the `dist` directory.

3 **Type the following command:**

```
ant deploy
```

▼ To Run the encoder Example Using Ant

1. **In a web browser, type the following URL:**

   ```
   http://localhost:8080/encoder/
   ```

   The String Encoder page opens.

2. **Type a string and the number of letters to shift by, then click Encode.**

   The encoded string appears in blue on the Result line. For example, if you type `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:**

   tut-install`/examples/cdi/encoder/web/WEB-INF/beans.xml`

   b. **Remove the comment characters that surround the `alternatives` element, so that it looks like this:**

   ```
   <alternatives>
       <class>encoder.TestCoderImpl</class>
   </alternatives>
   ```

   c. **Save and close the file.**

4. **Type the following commands:**

   ```
   ant undeploy
   ant
   ant deploy
   ```

5. **In the web browser, retype the URL to show the String Encoder page for the redeployed project:**

   ```
   http://localhost:8080/encoder
   ```

6. **Type a string and the number of letters to shift by, then click Encode.**

   This time, the Result line displays your arguments. For example, if you type `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" on page 541. It is very similar to the `encoder` example described in "The encoder Example: Using Alternatives" on page 553. 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 the beans.xml file does not contain an alternatives element.

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), takes both implementations as arguments, then returns one or the other, based on the coderType supplied by the user.

```
@Produces
@Chosen
@RequestScoped
public Coder getCoder(@New TestCoderImpl tci,
        @New CoderImpl ci) {

    switch (coderType) {
        case TEST:
            return tci;
        case SHIFT:
            return ci;
        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 radio buttons 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 Ant to build, package, deploy, and run the `producermethods` application.

▼ To Build, Package, and Deploy the `producermethods` Example Using NetBeans IDE

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

   tut-install`/examples/cdi/`
3. **Select the `producermethods` folder.**
4. **Select the Open as Main Project check box.**
5. **Click Open Project.**
6. **In the Projects tab, right-click the `producermethods` project and select Deploy.**

▼ To Build, Package, and Deploy the `producermethods` Example Using Ant

1 In a terminal window, go to:

tut-install`/examples/cdi/producermethods/`

2 Type the following command:

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `producermethods.war`, located in the `dist` directory.

3 Type the following command:

```
ant deploy
```

▼ To Run the `producermethods` Example

1 In a web browser, type the following URL:

`http://localhost:8080/producermethods`

The String Encoder page opens.

2 Select either the Test or Shift Letters radio button, type a string and the number of letters to shift by, 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 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 `javax.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. The timestamp field, `timeCreated`, is annotated with `@Temporal`, which is required for persistent `Date` fields.

```
@Entity
public class ToDo implements Serializable {

    ...
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    protected String taskText;
    @Temporal(TIMESTAMP)
    protected Date timeCreated;

    public ToDo() {
    }

    public ToDo(Long id, String taskText, Date 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) {
        ToDo toDo;
        Date currentTime = Calendar.getInstance().getTime();

        try {
            toDo = new ToDo();
            toDo.setTaskText(inputString);
            toDo.setTimeCreated(currentTime);
            em.persist(toDo);
            return toDo;
        } catch (Exception e) {
            throw new EJBException(e.getMessage());
        }
    }

    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>Type a task to be completed.</p>
        <h:form id="todolist">
            <p><h:outputLabel value="Type 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 `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to `true`:

```
<context-param>
  <param-name>
      javax.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 Ant to build, package, deploy, and run the `producerfields` application.

▼ To Build, Package, and Deploy the `producerfields` Example Using NetBeans IDE

1 **If the database server is not already running, start it by following the instructions in "Starting and Stopping the Java DB Server" on page 75.**

2 **From the File menu, choose Open Project.**

3 **In the Open Project dialog, navigate to:**

tut-install`/examples/cdi/`

4 **Select the `producerfields` folder.**

5 **Select the Open as Main Project check box.**

6 **Click Open Project.**

7 **In the Projects tab, right-click the `producerfields` project and select Deploy.**

▼ To Build, Package, and Deploy the `producerfields` Example Using Ant

1 **If the database server is not already running, start it by following the instructions in "Starting and Stopping the Java DB Server" on page 75.**

2 **In a terminal window, go to:**

tut-install`/examples/cdi/producerfields/`

3 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `producerfields.war`, located in the `dist` directory.

4 **Type the following command:**

```
ant deploy
```

▼ To Run the `producerfields` Example

1 **In a web browser, type the following URL:**

`http://localhost:8080/producerfields`

The Create To Do List page opens.

2 **Type a string in the text field and click Submit.**

You can type additional strings and click Submit to create a task list with multiple items.

3 **Click the Show Items button.**

The To Do List page opens, showing the timestamp and text for each item you created.

4 **Click the Back button 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 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" on page 569.

Like `PaymentEvent`, 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>Type 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 text field takes a payment amount, passed to paymentBean.value. Two radio buttons 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, while 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 radio button 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(Calendar.getInstance().getTime());
        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.xhtml";
    }
```

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>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 Ant to build, package, deploy, and run the `billpayment` application.

▼ To Build, Package, and Deploy the `billpayment` Example Using NetBeans IDE

1 **From the File menu, choose Open Project.**

2 **In the Open Project dialog, navigate to:**

tut-install`/examples/cdi/`

3 **Select the `billpayment` folder.**

4 **Select the Open as Main Project check box.**

5 **Click Open Project.**

6 **In the Projects tab, right-click the `billpayment` project and select Deploy.**

▼ To Build, Package, and Deploy the `billpayment` Example Using Ant

1 **In a terminal window, go to:**

tut-install`/examples/cdi/billpayment/`

2 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `billpayment.war`, located in the `dist` directory.

3 **Type the following command:**

```
ant deploy
```

▼ To Run the `billpayment` Example

1 **In a web browser, type the following URL:**

`http://localhost:8080/billpayment`

The Bill Payment Options page opens.

2 **Type a value in the Amount field.**

The amount can contain up to 10 digits and include up to 2 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 **(Optional) 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 3+ output window. 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.

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. 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" on page 553. 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;

    @Override
    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>decorators.CoderDecorator</class>
    </decorators>
    <interceptors>
        <class>decorators.LoggedInterceptor</class>
    </interceptors>
```

Running the `decorators` Example

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

▼ To Build, Package, and Deploy the `decorators` Example Using NetBeans IDE

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

   tut-install`/examples/cdi/`
3. **Select the `decorators` folder.**
4. **Select the Open as Main Project check box.**

5 **Click Open Project.**

6 **In the Projects tab, right-click the `decorators` project and select Deploy.**

▼ To Build, Package, and Deploy the `decorators` Example Using Ant

1 **In a terminal window, go to:**

tut-install/examples/cdi/decorators/

2 **Type the following command:**

```
ant
```

This command calls the `default` target, which builds and packages the application into a WAR file, `decorators.war`, located in the `dist` directory.

3 **Type the following command:**

```
ant deploy
```

▼ To Run the `decorators` Example

1 **In a web browser, type the following URL:**

```
http://localhost:8080/decorators
```

The Decorated String Encoder page opens.

2 **Type a string and the number of letters to shift by, then click Encode.**

The output from the decorator method appears in blue on the Result line. For example, if you typed `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 3+ output window. Otherwise, view *domain-dir*/`logs/server.log`.

The output from the interceptors appears:

```
INFO: Entering method: encodeString in class decorators.CoderBean
INFO: Entering method: codeString in class decorators.CoderImpl
```