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 following directory:

tut-install`/examples/cdi/`

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 following directory:

tut-install`/examples/bp-project/`

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 `producermethods` Example: Using a Producer Method To Choose a Bean Implementation
- The `producerfields` Example: Using Producer Fields to Generate Resources
- The `billpayment` Example: Using Events and Interceptors
- The `decorators` Example: Decorating a Bean

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.

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 TestCoderImplclass, 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
xmlns=
"http://www.w3.org/1999/xhtml"
xmlns:h=
"http://java.sun.com/jsf/html"
>
<h:head>
<title>String Encoder</title>
<link
href="resources/css/default.css"
rel="stylesheet" type="text/css"/>
```

```
</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); }
```

Building, Packaging, Deploying, and Running the `encoder` Example

You can build, package, deploy, and run the `encoder` application by using either NetBeans IDE or the Ant tool.

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 command:

```
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 and Fields.

It is very similar to the `encoder` example described in The `encoder` Example: Using Alternatives.

The example includes the same interface and two implementations of it, a managed bean, a Facelets page, and configuration files.

It also contains a qualifier type.

When you run it, you do not need to edit the `beans.xml` file and redeploy the application to change its behavior.

Components of the `producermethods` Example

The components of `producermethods` are very much like those for `encoder`, with some significant differences.

Neither implementation of the `Coder` bean is annotated `@Alternative`, and 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>
...
```

Building, Packaging, Deploying, and Running the `producermethods` Example

You can build, package, deploy, and run the `producermethods` application by using either NetBeans IDE or the Ant tool.

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 that it can easily be used by other components in the application.

The class uses a producer field to inject an EntityManager that is 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 comments 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 may want to 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{
private static final
long serialVersionUID = 1L;
@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 = null;
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 that an attempt to submit an empty string results in an error.

```
@Named
```

```
@ConversationScoped
public class ListBean implements
Serializable {
private static final
long serialVersionUID = 1L;
@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.

Building, Packaging, Deploying, and Running the `producerfields` Example

You can build, package, deploy, and run the `producerfields` application by using either NetBeans IDE or the Ant tool.

To Build, Package, and Deploy the `producerfields` 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 `producerfields` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

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

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

1. In a terminal window, go to:

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

2. 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.

3. 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 time stamp.

```
public class PaymentEvent
implements Serializable {
private static final
long serialVersionUID = 1L;
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 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.

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 that 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 that is 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{
private static final
long serialVersionUID = 1L;
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 that all their methods are intercepted.

In `PaymentBean`, only the `pay` and `reset` methods are annotated `@Logged`, so only those methods are intercepted.

Building, Packaging, Deploying, and Running the `billpayment` Example

You can build, package, deploy, and run the `billpayment` application by using either NetBeans IDE or the Ant tool.

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 3.1 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:

```
INFO: Entering method: pay in class
billpayment.payment.PaymentBean
```

```
INFO: PaymentHandler created.
INFO: PaymentHandler created.
INFO: PaymentHandler created.
INFO: Entering method: debitPayment in
class billpayment.listener.PaymentHandler
INFO: PaymentHandler - Debit Handler:
Debit = $1234.56 at Tue Dec 14 14:50:28
EST 2010
```

The `decorators` Example: Decorating a Bean

The `decorators` example, which is yet another variation on the `encoder` example, shows how to use a decorator to implement additional business logic for a bean.

Instead of having the user choose between two alternative implementations of an interface at deployment time or runtime, a decorator adds some additional logic to a single implementation of the interface.

The example includes an interface, an implementation of it, a decorator, an interceptor, a managed bean, a Facelets page, and configuration files.

Components of the `decorators` Example

The `decorators` example is very similar to the `encoder` example described in The `encoder` Example: Using Alternatives.

Instead of providing two implementations of the `Coder` interface, however, this example provides only the `CoderImpl` class.

The decorator class, `CoderDecorator`, instead of simply returning the coded string, displays the input and output strings' values and length.

The `CoderDecorator` class, like `CoderImpl`, implements the business method of the `Coder` interface, `codeString`:

```
@Decorator
public abstract class
CoderDecorator implements Coder {
@Inject
@Delegate
@Any
Coder coder;
public String codeString
(String s, int tval){
int len = s.length();
```

```
return "\"" + s + "\" becomes " +
"\"" + coder.codeString(s, tval)
+ "\", " + len +
" characters in length";
}
}
```

The decorator's `codeString` method calls the delegate object's `codeString` method to perform the actual encoding.

The `decorators` example includes the `Logged` interceptor binding and `LoggedInterceptor` class from the `billpayment` example.

For this example, the interceptor is set on the `CoderBean.encodeString` method and the `CoderImpl.codeString` method.

The interceptor code is unchanged; interceptors are usually reusable for different applications.

Except for the interceptor annotations, the `CoderBean` and `CoderImpl` classes are identical to the versions in the `encoder` example.

The `beans.xml` file specifies both the decorator and the interceptor:

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

Building, Packaging, Deploying, and Running the `decorators` Example

You can build, package, deploy, and run the `decorators` application by using either NetBeans IDE or the Ant tool.

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 Bill Payment Options 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 type `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 3.1 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
```