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 that CDI provides to enable loose coupling of components with strong typing, as described in Overview of CDI.

The following topics are addressed here:

- Using Alternatives
- Using Producer Methods and Fields
- Using Events
- Using Interceptors
- Using Decorators
- Using Stereotypes

Using Alternatives

When you have more than one version of a bean that you use for different purposes, you can choose between them during the development phase by injecting one qualifier or another, as shown in The `simplegreeting` CDI Example.

Instead of having to change the source code of your application, however, you can make the choice at deployment time by using alternatives.

Alternatives are commonly used for purposes 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 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 and are 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 that you defined the following two beans:

```
@Default @Asynchronous
public class AsynchronousService
implements Service { ... }

@Alternative public class
MockAsynchronousService extends
AsynchronousService { ... }
```

If you then declared `MockAsynchronousService` as an alternative in your `beans.xml` file, the following injection point would resolve to `MockAsynchronousService`:

```
@Inject Service service;
```

The following, however, would resolve to `AsynchronousService` rather than `MockAsynchronousService`, because `MockAsynchronousService` does not have the `@Asynchronous` qualifier:

```
@Inject @Asynchronous
Service service;
```

To make sure that `MockAsynchronousService` is 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 is 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 and Fields

A **producer method** is a method that generates an **object** that can then be injected.

Typically, you **use** producer methods in the following situations:

- **When you want to inject an object that is not itself a bean**

- **When the concrete type of the object to be injected may vary at runtime**

- **When the object requires some custom initialization that the bean constructor does not perform**

For more information on producer methods, see Injecting Objects by Using Producer Methods.

A producer field is a simpler alternative to a producer method; it is a field of a bean that generates an object.

It can be used instead of a simple getter method.

Producer fields are particularly useful for declaring Java EE resources.

A producer method or field is annotated with the `javax.enterprise.inject.Produces` annotation.

A producer method can allow you to select a bean implementation at runtime, instead of at development time or deployment time.

For example, in the example described in The `producermethods` Example: Using a Producer Method To Choose a Bean Implementation, the managed bean defines the following producer method:

```
@Produces
@Chosen
@RequestScoped
```

```
public Coder getCoder(
@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 abort deployment, informing the user of the ambiguous dependency.

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 shows how to use producer fields to generate an entity manager.

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 Events

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, 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 aborts 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 that 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;
```

```
public static final int DEBIT = 1;
public 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

An interceptor is a class that is 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 that 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.

You can now use them with Java EE managed objects of all kinds, including managed beans.

For information on Java EE interceptors, see Chapter 48, 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.

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

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 interceptors are, their logic is specific to a particular application.

For example, instead of using an alternative `TestCoderImpl` class for the `encoder` example, you could create a decorator as follows:

```
@Decorator
public abstract class
CoderDecorator implements Coder{
@Inject
@Delegate
@Any
Coder coder;
```

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

See The `decorators` Example: Decorating a Bean for an example that uses this decorator.

This simple decorator returns more detailed output than the encoded string returned by the `CoderImpl.codeString` method.

A more complex decorator could store information in a database or perform some other business logic.

A decorator can be declared as an abstract class, so that it does not have to implement all the business methods of the interface.

In order for a decorator to be invoked in a CDI application, it must, like an interceptor or an alternative, be specified in the `beans.xml` file.

For example, the `CoderDecorator` class is specified as follows:

```
<decorators>
<class>
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

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 that 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`:

```java
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```