Weld - JSR-299 Reference Implementation

JSR-299: The new Java standard for dependency injection and contextual lifecycle management

Gavin King
JSR-299 specification lead
Red Hat, Inc.

Pete Muir
Weld (JSR-299 Reference Implementation) lead
Red Hat, Inc.

Dan Allen
Senior Software Engineer
Red Hat, Inc.

David Allen

Italian Translation: Nicola Benaglia, Francesco Milesi
Spanish Translation: Gladys Guerrero

Red Hat, Inc.
Korean Translation: Eun-Ju Ki,
Red Hat, Inc.
Traditional Chinese Translation: Terry Chuang
Red Hat, Inc.
Simplified Chinese Translation: Sean Wu
Kava Community

A note about naming and nomenclature ... vii
I. Loosely-coupled, contextual components with strong typing ... 1
1. Getting started with JSR-299 (CDI) ... 3
1.1. What is a bean? ... 3
1.2. Your first bean (or is it?) ... 4
1.3. The anatomy of a bean ... 6
1.3.1. Bean types, qualifiers and dependency injection ... 8
1.3.2. Scope ... 10
1.3.3. EL name ... 11
1.3.4. Alternatives ... 12
1.3.5. Interceptor binding types ... 12
1.4. What kinds of classes can be beans? ... 13
1.4.1. Managed beans ... 13
1.4.2. Session beans ... 14
1.4.3. Producer methods ... 15
1.4.4. Producer fields ... 17
1.4.5. Java EE resources ... 18
1.4.6. Built-in beans ... 19
2. JSF web application example ... 21
3. Getting started Weld, the JSR-299 reference implementation ... 25
3.1. Prerequisites ... 25
3.2. Deploying to JBoss AS ... 26
3.3. Deploying to GlassFish ... 28
3.4. Deploying to Apache Tomcat ... 29
3.4.1. Deploying with Ant ... 30
3.4.2. Deploying with Maven ... 31
3.5. Deploying to Jetty ... 32
3.6. The numberguess example in depth ... 33
3.6.1. The numberguess example in Apache Tomcat or Jetty ... 39
3.6.2. The numberguess example for Apache Wicket ... 40
3.6.3. The numberguess example for Java SE with Swing ... 47
3.7. The translator example in depth ... 52
4. Dependency injection and programmatic lookup ... 59
4.1. Where you can @Inject ... 59
4.2. What gets injected ... 61
4.3. Qualifier annotations ... 62
4.3.1. Qualifiers with members ... 64
4.3.2. Combining qualifiers ... 64
4.3.3. Qualifiers on producer methods ... 65
4.3.4. The default qualifier ... 65
4.4. Fixing unsatisfied dependencies ... 65
4.5. Client proxies ... 66
4.6. Obtaining a contextual instance by programmatic lookup ... 67
4.7. Bean names and EL lookup ... 68

4.8. Lifecycle callbacks and resource injections 69
5. Scopes and contexts 71
5.1. Scope types 71
5.2. Built-in scopes 72
5.3. The conversation scope 73
5.3.1. Conversation demarcation 73
5.3.2. Conversation propagation 74
5.3.3. Conversation timeout 75
5.4. The dependent pseudo-scope 75
5.4.1. The `@New` annotation 75
6. Producer methods 77
6.1. Scope of a producer method 78
6.2. Injection into producer methods 78
6.3. Use of `@New` with producer methods 79
II. Developing loosely-coupled code 81
7. Interceptors 83
7.1. Interceptor bindings 83
7.2. Implementing interceptors 84
7.3. Enabling interceptors 84
7.4. Interceptor bindings with members 85
7.5. Multiple interceptor binding annotations 86
7.6. Interceptor binding type inheritance 87
7.7. Use of `@Interceptors` 88
8. Decorators 89
8.1. Delegate object 91
8.2. Enabling decorators 91
9. Events 93
9.1. Event payload 93
9.2. Event observers 94
9.3. Event producers (Firing events) 94
9.4. Conditional observer methods 96
9.5. Event qualifiers with members 96
9.6. Multiple event bindings 97
9.7. Transactional observers 98
III. Making the most of strong typing 101
10. Stereotypes 103
10.1. Default scope for a stereotype 103
10.2. Interceptor bindings for stereotypes 104
10.3. Name defaulting with stereotypes 104
10.4. Alternatives as stereotypes 105
10.5. Stereotype stacking 105
10.6. Built-in stereotypes 106
11. Specialization, inheritance and alternatives 107
11.1. Bean alternatives 107

11.2. Using specialization 110
11.3. Inheritance 110
IV. CDI and the Java EE ecosystem 111
12. Java EE integration 113
12.1. Injecting Java EE resources into a bean 113
12.2. Calling a bean from a Servlet 113
12.3. Calling a bean from a Message-Driven Bean 114
12.4. JMS endpoints 115
12.5. Packaging and deployment 116
13. Extending CDI through portable extensions 117
13.1. The `BeanManager` object 117
13.2. The `Bean` class 118
13.3. The `Context` interface 119
14. Next steps 121
V. Weld reference 123
15. Application servers and environments supported by Weld 125
15.1. Using Weld with JBoss AS 125
15.2. GlassFish 125
15.3. Servlet containers (such as Tomcat or Jetty) 126
15.3.1. Tomcat 126
15.3.2. Jetty 127
15.4. Java SE 128
15.4.1. CDI SE Module 128
16. CDI extensions available as part of Weld 131
16.1. Weld Logger 131
17. Alternative view layers 133
17.1. Wicket CDI integration 133
17.1.1. The `WebApplication` class 133
17.1.2. Conversations with Wicket 133
A. Integrating Weld into other environments 135
A.1. The Weld SPI 135
A.1.1. Deployment structure 135
A.1.2. EJB descriptors 137
A.1.3. EE resource injection and resolution services 137
A.1.4. EJB services 138
A.1.5. JPA services 138
A.1.6. Transaction Services 138
A.1.7. Resource Services 139
A.1.8. Injection Services 139
A.1.9. Security Services 139
A.1.10. Bean Validation Services 139
A.1.11. Identifying the BDA being addressed 139
A.1.12. The bean store 140
A.1.13. The application context 140

A.1.14. Initialization and shutdown 140
A.1.15. Resource loading 141
A.2. The contract with the container 141

A note about naming and nomenclature

Shortly before the final draft of JSR-299 was submitted, the specification changed its name from "Web Beans" to "Java Contexts and Dependency Injection for the Java EE platform", abbreviated CDI. For a brief period after the renaming, the reference implementation adopted the name "Web Beans". However, this ended up causing more confusion than it solved and Red Hat decided to change the name of the reference implementation to "Weld". You may still find other documentation, blogs, forum posts, etc. that use the old nomenclature. Please update any references you can. The naming game is over.

You'll also find that some of the functionality that once existed in the specification is now missing, such as defining beans in XML. These features will be available as portable extensions for CDI in the Weld project, and perhaps other implementations.

Note that this reference guide was started while changes were still being made to the specification. We've done our best to update it for accuracy. If you discover a conflict between what is written in this guide and the specification, the specification is the authority—assume it is correct. If you believe you have found an error in the specification, please report it to the JSR-299 EG.

Part I. Loosely-coupled, contextual components with strong typing

The *JSR-299* specification (CDI) defines a set of services for the Java EE environment that make applications much easier to develop. CDI layers an enhanced lifecycle and interaction model over existing Java component types, including Managed Beans (JavaBeans) and Enterprise JavaBeans (EJB session beans). As a complement to the traditional Java EE programming model, the CDI services provide:

- an improved lifecycle for stateful components, bound to well-defined *contexts*,
- a typesafe approach to *dependency injection*,
- component interaction via an *event notification facility*,
- a better approach to binding *interceptors* to components, along with a new kind of interceptor, called a *decorator*, that is more appropriate for use in solving business problems, and
- an *SPI* for developing portable extensions for the Java EE platform.

A component that receives CDI services is referred to, generally, as a bean. The CDI services are a core aspect of the Java EE platform and apply to the following types of components that exist in the platform:

- all Managed Beans (JavaBeans), including JSF managed beans,
- all Enterprise JavaBeans (EJBs), and
- all Servlets.

CDI is especially useful in the context of web applications, but is applicable to a wide variety of applications. It may even be used in the Java SE context, in conjunction with an embeddable EJB Lite container, as defined in the EJB 3.1 specification, or through an extension (see *Section 15.4.1, "CDI SE Module"*).

The concerns handled by CDI save the user who is unfamiliar with an API from having to answer the following questions:

- What is the lifecycle of this object?
- How many simultaneous clients can it have?
- Is it multithreaded?
- How do I retrieve it?

- Do I need to explicitly destroy it?
- Where should I keep the reference to it when I'm not currently using it?
- How can I define alternatives, so that the implementation of this object can vary at deployment time?
- How should I go about sharing this object between other objects?

The main theme of CDI, which is instrumental in removing these complexities, is *loose-couping with strong typing*. Let's study what that phrase means.

A bean specifies only the type and semantics of other beans it depends upon. It need not be aware of the actual lifecycle, concrete implementation, threading model or other clients of any bean it depends upon. Even better, the concrete implementation, lifecycle and threading model of a bean it depends upon may vary according to the deployment scenario, without affecting any client. This loose-coupling is what makes this architecture so simple, yet powerful.

Events, interceptors and decorators enhance the *loose-coupling* inherent in this model:

- *event notifications* decouple event producers from event consumers,
- *interceptors* decouple technical concerns from business logic, and
- *decorators* allow business concerns to be compartmentalized.

What's even more powerful (and comforting) is that CDI provides all these facilities in a *typesafe* way. CDI never relies on string-based identifiers to determine how collaborating objects fit together. (XML is rarely used, reserved only to activate alternatives and define ordering at deployment time). Instead, CDI uses the typing information that is already available in the Java object model, then extends it with a new typing pattern, called *qualifier annotations*, to wire together beans, their dependencies, their interceptors and decorators and their event consumers.

CDI even provides the necessary integration points, through a comprehensive SPI, so that other kinds of components defined by future Java EE specifications or by non-standard frameworks may be cleanly integrated with CDI, take advantage of the CDI services, and interact with any other kind of platform component.

CDI was influenced by a number of existing Java frameworks, including Seam, Guice and Spring. However, CDI has its own, very distinct, character: more typesafe than Seam, more stateful and less XML-centric than Spring, more web and enterprise-application capable than Guice. But it couldn't have been any of these without inspiration from the frameworks mentioned and *lots* of collaboration and hard work by the JSR-299 Expert Group (EG).

Finally, JSR-299 is a *Java Community Process* (JCP) standard that integrates cleanly with the Java EE platform, and with any Java SE environment where embeddable EJB Lite is available. In fact, Java EE 6 requires that all compliant application servers provide support for JSR-299 (even in the web profile).

Getting started with JSR-299 (CDI)

So you're keen to get started writing your first bean? Or perhaps you're skeptical, wondering what kinds of hoops the CDI specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of beans. CDI just helps you make use them in your web or enterprise application.

1.1. What is a bean?

In this section, you'll learn that a bean is exactly what you think it is. Only now, it has a true identity in Java EE.

Prior to Java EE 6, there existed no clear definition of a bean in the platform. Of course, we've been calling Java classes used in web and enterprise applications beans for years. Third-party frameworks, such as Spring, even have facilities for defining and creating beans; and there are a couple of bean variants in the platform itself, including EJB session beans and JSF managed beans. What was missing was a common definition.

Java EE 6 finally lays down the definition of a bean in the Managed Bean specification. Managed Beans are defined as container-managed objects with minimal requirements, otherwise known by their acronym, POJOs (Plain Old Java Objects). They support a small set of basic services, such as resource injection, lifecycle callbacks and interceptors. Companion specifications, such as EJB and CDI, build on this basic model. But, *at last*, there is a universally accepted concept of a bean and a lightweight component model that's aligned across the Java EE platform.

With certain, unique exceptions, every Java class that has a constructor with no parameters (or an alternate constructor designated with the annotation `@Inject`) is a candidate for becoming a bean. Any EJB session bean is also a candidate. They become beans, specifically CDI beans, if they reside in a Java EE module that contains a special marker file (META-INF/beans.xml) and if instances of those classes are managed by the Java EE container. Just know that, for simple JavaBean classes, there's no special metadata you need to add to make them beans. They just are.

The JavaBeans and EJBs you've been writing every day, up until now, have not been able to take advantage of the new services defined by the CDI specification. But you'll be able to use every one of them with CDI -- allowing the container to create and destroy instances of your beans and associate them with a designed context (or scope), injecting them into other beans, using them in EL expressions, specializing them with qualifier annotations, even adding interceptors and decorators to them—from this point forward without modifying your code. At most, you'll have to add some annotations.

Let's see how to create your first bean.

1.2. Your first bean (or is it?)

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
   public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Where `Translator` is the EJB local interface:

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Unfortunately, we don't have a class that translates whole text documents. So let's write a bean for this job:

```
public class TextTranslator {
   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

   @Inject
   TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
      this.sentenceParser = sentenceParser;
      this.sentenceTranslator = sentenceTranslator;
   }

   public String translate(String text) {
      StringBuilder sb = new StringBuilder();
      for (String sentence: sentenceParser.parse(text)) {
```

```
        sb.append(sentenceTranslator.translate(sentence));
      }
      return sb.toString();
   }
}
```

But wait! `TextTranslator` does not have a constructor with no parameters! Is it still a bean? If you remember, a class that does not have a constructor with no parameters can still be a bean if it has a constructor annotated `@Inject`.

As you've guessed, the `@Inject` annotation has something to do with dependency injection! `@Inject` may be applied to a constructor or method of a bean, and tells the container to call that constructor or method when instantiating the bean. The container will inject other beans it finds into the parameters of the constructor or method.

We may obtain an instance of `TextTranslator` by injecting it into a constructor, bean "setter" method, producer method, observer method or field of a JavaBean or EJB session, or a field of a Servlet. The injection is based on the type of the injection point, not the name of the field, method or argument.

Let's create a UI controller bean that uses field injection to obtain an instance of the `TextTranslator`, translating the text entered by a user:

```
@Named @RequestScoped
public class TranslateController {

   @Inject TextTranslator textTranslator;                    (1)

   private String inputText;
   private String translation;

   // JSF action method, perhaps
   public void translate() {
      translation = textTranslator.translate(inputText);
   }

   public String getInputText() {
      return inputText;
   }

   public void setInputText(String text) {
      this.inputText = text;
   }

   public String getTranslation() {
```

```
        return translation;
    }
}
```

(1) Field injection of `TextTranslator` instance

Tip

Notice the controller bean is request-scoped and named. Since this combination is so common in web applications, there's a built-in annotation for it in CDI that we could have used as a shorthand. When the (stereotype) annotation `@Model` is declared on a class, it creates a request-scoped and named bean.

Alternatively, we may obtain an instance of `TextTranslator` programatically from an injected instance of `Instance`, paramaterized with the bean type:

```
@Inject Instance<TextTranslator> textTranslatorSource;
...
public void translate() {
   textTranslatorSource.get().translate(inputText);
}
```

Notice that it isn't necessary to create a getter or setter method to inject one bean into another. CDI can access the field directly (even if it's private!), which should help eliminate some wasteful code. The name of the field is arbitrary. It's the field's type that determines what is injected.

At system initialization time, the container must validate that exactly one bean exists which satisfies each injection point. In our example, if no implementation of `Translator` is available—if the `SentenceTranslator` EJB was not deployed—the container would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` were available, the container would throw an `AmbiguousDependencyException`. The same for the `TextTranslator` injection point.

Now we are starting to venture off into the details, so let's pause and examine a bean's anatomy. What aspects of the bean are significant, and what gives it its identity?

1.3. The anatomy of a bean

A bean is an application class that contains business logic. It may be called directly from Java code, or it may be invoked via Unified EL. A bean may access transactional resources. Dependencies between beans are managed automatically by the container. Most beans are *stateful* and *contextual*. The lifecycle of a bean is always managed by the container.

Let's back up a second. What does it really mean to be *contextual*? Since beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a bean see the bean in different states. The client-visible state depends upon which instance of the bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the bean determines:

- the lifecycle of each instance of the bean and
- which clients share a reference to a particular instance of the bean.

For a given thread in a CDI application, there may be an *active context* associated with the scope of the bean. This context may be unique to the thread (for example, if the bean is request scoped), or it may be shared with certain other threads (for example, if the bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other beans) executing in the same context will see the same instance of the bean. But clients in a different context may see a different instance (depending on the relationship between the contexts).

One great advantage of the contextual model is that it allows stateful beans to be treated like services! The client need not concern itself with managing the lifecycle of the bean it's using, *nor does it even need to know what that lifecyle is.* Beans interact by passing messages, and the bean implementations define the lifecycle of their own state. The beans are loosely coupled because:

- they interact via well-defined public APIs
- their lifecycles are completely decoupled

We can replace one bean with another different bean that implements the same interface and has a different lifecycle (a different scope) without affecting the other bean implementation. In fact, CDI defines a simple facility for overriding bean implementations at deployment time, as we will see in *???* (FIXREF).

Note that not all clients of a bean are beans themselves. Other objects such as Servlets or Message-Driven Beans—which are by nature not injectable, contextual objects—may also obtain references to beans by injection.

Enough hand-waving. More formally, the anatomy of a bean, according to the spec:

> A bean comprises of the following attributes:
>
> - A (nonempty) set of bean types
> - A (nonempty) set of qualifiers

- A scope
- A deployment type
- Optionally, a bean EL name
- A set of interceptor bindings
- A bean implementation

Let's see what some of these terms mean, to the CDI developer.

1.3.1. Bean types, qualifiers and dependency injection

Beans usually acquire references to other beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the bean to be injected. The contract is:

- a bean type, together with
- a set of qualifiers.

A bean type is a user-defined class or interface; types that are client-visible. If the bean is an EJB session bean, the bean type is the `@Local` interface or bean-class local view. A bean may have multiple bean types. For example, the following bean has four bean types:

```
public class BookShop extends Business implements Shop<Book> {
   ...
}
```

The bean types are `BookShop`, `Business` and `Shop<Book>`, as well as the implicit type `java.lang.Object`. (Notice that a parameterized type is a legal bean type). Meanwhile, this session bean has only the local interfaces `BookShop`, `Auditable` and `java.lang.Object` as bean types, since the bean class, `BookShopBean` is not a client-visible type.

```
@Stateful
public class BookShopBean extends Business implements BookShop, Auditable {
     ...
}
```

Note

Most bean types you can probably figure out. One gotcha is primitive types. All primitive types are assumed to be identical to their corresponding wrapper types in `java.lang`.

Bean types may be restricted to an explicit set by annotating the bean with the `@Typed` annotation and listing the bean types in the value of the annotation. For instance, this bean has been restricted to a single bean type, `Shop`:

```
@Typed(Shop.class)
public class BookShop extends Business implements Shop<Book> {
   ...
}
```

The bean types alone often do not provide enough information for the container to know which bean to inject. For instance, say you have two implementations of the `PaymentProcessor` interface: `CreditCardPaymentProcessor` and `DebitPaymentProcessor`. Injecting into a field of type `PaymentProcessor` introduces an ambigious condition. In these cases, the container must rely on some client-visible semantic that is satisified by one implementation of the bean type, but not by others. That semantic is called a qualifier.

Qualifiers are represented by user-defined annotations that are themselves annotated with `@Qualifer`. These annotations extend the type system in Java to let you further qualify the type without having to fall back to string-based names as many dependency injection frameworks use. Here's an example of a qualifier annotation:

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

You may not be used to seeing the definition of an annotation. In fact, this might be the first time you have encountered one. With CDI, they will become very familiar artifact as you will be creating them often. Pay attention to the names of the built-in annotations in CDI and EJB.

Tip

You'll notice that they are all adjectives. You are encouraged to follow this convention when creating your custom annotations, since they serve to describe the behaviors and roles of the class.

Now that we have defined a qualifier annotation, we can use it. The following injection point has the bean type `PaymentProcessor` and qualifier `@CreditCard`:

```
@Inject @CreditCard PaymentProcessor paymentProcessor
```

Note

If no qualifier is specified at an injection point, the default qualifier, `@Default`, is assumed.

For each injection point, the container searches for a bean which satisfies the contract, meaning it matches the bean time and all the qualifiers. It then injects that instance.

We've seen how to indicate that you want to inject qualified bean. But how do you actually qualify the bean? By using the annotation, of course! The following bean has the qualifier `@CreditCard` and implements the bean type `PaymentProcessor`. Therefore, it satisfies our qualified injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
   implements PaymentProcessor { ... }
```

Note

If a bean does not explicitly specify a qualifier, it has exactly one qualifier, the default qualifier, `@Default`.

CDI defines a sophisticated, yet intuitive *resolution algorithm* that helps the container decide what to do if there is more than one bean that satisfies a particular contract. We'll get into the details in ???.

1.3.2. Scope

The *scope* of a bean defines the lifecycle and visibility of instances created from it. The CDI context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the container. Each scope is represented by an annotation type.

For example, any web application may have *session scoped* bean:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session-scoped bean is bound to a user session and is shared by all requests that execute in the context of that session.

Note

Keep in mind that once a bean is bound to a context, it remains in that context until the context is destroyed. There is no way to explicitly remove a bean from a context. If you don't the bean to live in the session indefinitely, consider using another scope such as the request or conversation scope instead.

If a scope is not explicitly specified, then the bean belongs to a special scope called the *dependent pseudo-scope*. Beans with this scope live to serve the bean into which they are injected, which means their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *Chapter 5, Scopes and contexts*.

1.3.3. EL name

If you want to reference a bean outside of Java, such as in JSF view, you must assign the bean a *name*. The name is used as the EL name of the bean, allowing the bean to be used in a Unified EL expression.

The name is specified using the `@Named` annotation, as shown here:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
   ...
</h:dataTable>
```

Note

The `@Named` annotation is not what makes the class a bean. Most classes in a bean archive are already recognized as beans. The `@Named` annotation just makes it possible to reference the bean from the EL, most commonly from a JSF view.

If you want, you can let CDI select a name for you by leaving off the value of the `@Named` annotation:

```
@SessionScoped @Named
```

```
public class ShoppingCart { ... }
```

The name defaults to the unqualified class name, decapitalized; in this case, `shoppingCart`.

1.3.4. Alternatives

There are cases when you want to replace one bean implementation with another at deployment time. For instance, you may want to use a mock implementation in a testing environment. An *alternative* may be declared by annotating the bean class or producer method or field with the `@Alternative` annotation.

```
@Alternative
public class MockPaymentProcessor extends PaymentProcessorImpl { ... }
```

An alternative can also be declared by annotating a bean, producer method or producer fields with a stereotype that has the `@Alternative` annotation. Stereotypes are explained in *Chapter 10, Stereotypes*.

You then enable the alternative using the CDI deployment descriptor, META-INF/beans.xml, in the classpath entry in which you intent to use it. An alternative must be explicitly selected in every bean deployment archive in which the alternative should be available for injection, lookup and EL resolution.

Details on how to enable alternative beans, and how you can use them to specialize (override) beans and producer methods is covered in *Chapter 11, Specialization, inheritance and alternatives*. CDI is very flexible when it comes alternatives that are selected at deployment time.

1.3.5. Interceptor binding types

Since the introduction of the Managed Bean specification in Java EE 6, interceptors are available on JavaBeans in additionn to EJB session beans. That's right, you no longer have to create an EJB just to intercept method calls. Holler. So what does CDI have to offer? Well, a lot actually. Let's cover some background.

The way that interceptors are defined in Java EE 5 is counter-intuitive. You are required to specify the *implementation* of the interceptor directly on the *implementation* of the EJB, either in the `@Interceptors` annotation or in the XML descriptor. You might as well just put the interceptor code *in* the implementation. Second, the order in which the interceptors are applied is taken from the order in which they are declared in the annotation or the XML descriptor. Perhaps this isn't so bad if you're applying the interceptors to a single bean. But, if you are applying them repeatly, then there is a chance that you'll inadvertly define a different order for different beans. Now that is a problem.

CDI provides a new approach to binding interceptors to beans that introduces a level of abstraction (and thus control). You define a special kind of annotation called an *interceptor binding type* whose name describes the role of the interceptor, in this case to add transaction support:

```
@InterceptorBinding
@Inherited
@Target( { TYPE, METHOD })
@Retention(RUNTIME)
public @interface Transactional {}
```

Then you apply the annotation to the class whose methods you wish to have intercepted.

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

Where is the interceptor? You match the interceptor(s) to apply to the bean by also annotating the interceptor class. Never do the two implementations (bean and interceptor) come in direct contact. Activating and ordering of the interceptors is then controlled by the XML descriptor, one of the few times XML is warranted.

We'll discuss bean interceptors, and their cousins, decorators, in *Chapter 7, Interceptors* and *Chapter 8, Decorators*.

1.4. What kinds of classes can be beans?

We've already seen that JavaBeans and EJBs can be (CDI) beans. Is that the whole story? Let's start from what we know, and go from there.

1.4.1. Managed beans

A managed bean is a bean that is implemented by almost any Java class. This class is called the bean class of the managed bean. The basic lifecycle and semantics of a managed bean is defined by the Managed Beans specification. If the class is not picked up as a managed bean by the container, then CDI will allow it to be a bean if:

- It is not a non-static 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—either:

- the class has a constructor with no parameters, or
- the class declares a constructor annotated `@Inject`.

Unlike the managed bean definition, no special declaration is required to define a managed bean. CDI takes advantage of the fact that the Managed Bean specification allows companion specifications to relax the requirement of having to add the `@ManagedBean` annotation.

Some managed beans are much more. Those are the session beans. While still technically managed beans, they have enough additional, enterprise features, that we consider them to be their own kind.

1.4.2. Session beans

EJB 3 session beans belong to the EJB specification. That is, the basic lifecycle and semantics. Beyond that, they get to participate in CDI just like any other bean. There are some restrictions about which scopes can be assigned to a session bean, but other than that, they are interchangable with regular managed beans. That means you can inject one session bean into another, a managed bean into a session bean, a session bean into a managed bean, have a managed bean observe an event raised by a session bean, and so on.

Note

Message-driven and entity beans are by nature non-contextual objects and may not be injected into other objects. Message-driven beans can take advantage of some CDI functionality, such as dependency injection and interceptors. In fact, CDI will perform injection into any message-driven or session bean, even those which are not contextual instances.

The unrestricted set of bean types for a session bean contains all local interfaces of the bean and their superinterfaces. If the session bean has a bean class local view, the unrestricted set of bean types contains the bean class and all superclasses. In addition, `java.lang.Object` is a bean type of every session bean. But, remote interfaces are *not* included in the set of bean types.

Stateful session beans can define a no arguments remove method, annotated `@Remove`, that is used by the application to indicate the instance should be destroyed. However, in a CDI environment, this method can only be executed by the application if the bean is dependent-scoped. Otherwise, it's illegal for the application to invoke this method.

So, when should we use a session instead of a basic managed bean? Whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,

- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

When we don't need any of these things, a basic managed bean will serve just fine.

Many beans (including any session or application scoped bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped beans should be EJBs.

Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

The point is, you use a session bean when you need the services it provides, not just because you want to use dependency injection, lifecycle management, or interceptors. The Java EE programming model makes a whole lot more sense now; you won't need to go off and invent your own bean container to make programming in Java EE palatable.

In fact, it's easy to start with simple managed bean, and later turn it into an EJB just by adding one of the following annotations: `@Stateless`, `@Stateful` or `@Singleton`.

1.4.3. Producer methods

Surely not everything you want to inject can be a bean class. What if you need to inject the implementation of an API that varies at runtime? And how would you inject objects that are created by another mechanism, such as a JPA result? Fortunately, the CDI specification recognized these and other cases and introduced the concept of a producer method.

A *producer method* is a method on a bean that is used as a bean source, meaning the method itself describes the bean and the container invokes the method to obtain an instance of the bean when no instance exists in its specified context. A producer method lets the application take full control of the instantiation process, specifically when:

- the objects to be injected are not required to be instances of beans, or
- the concrete type of the objects to be injected may vary at runtime, or
- the objects require some custom initialization that is not performed by the bean constructor.

For example:

```
@ApplicationScoped
public class RandomNumberGenerator {

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

  @Produces @Named @Random int getRandomNumber() {
    return random.nextInt(100);
  }

}
```

Obviously, you cannot define a bean that is itself a random number. A producer method allows you do define its result as a bean, in this case an `Integer` with qualifier `@Random`, scope `@Dependent` (implied) and name `randomNumber` (derived from bean property convention; otherwise it would be the same as the method name). It can be injected just like any other bean:

```
@Inject @Random int randomNumber;
```

or used in a Unified EL expression:

```
Your raffle number is #{randomNumber}.
```

A producer method must be a non-abstract method of a managed bean class or session bean class. A producer method may be either static or non-static. If the bean is a session bean, the producer method must be either a business method of the EJB or a static method of the bean class.

The bean types of a producer method depend upon the method return type:

- If the return type is an interface, the unrestricted set of bean types contains the return type, all interfaces it extends directly or indirectly and `java.lang.Object`.
- If a return type is primitive or is a Java array type, the unrestricted set of bean types contains exactly two types: the method return type and `java.lang.Object`.
- If the return type is a class, the unrestricted set of bean types contains the return type, every superclass and all interfaces it implements directly or indirectly.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
  return createConnection(user.getId(), user.getPassword());
```

```
}
```

If the producer method has method arguments, as in this example, the container will look for matching beans and pass them into the method automatically—another form of dependency injection.

Producer methods may also define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

The disposal method is called automatically when the context ends (in this case, the end of the request). The disposal method requires at least one parameter, the bean produced by the producer method. Any additional parameters will be satisfied by the container.

We'll talk much more about producer and disposal methods in *Chapter 6, Producer methods*.

1.4.4. Producer fields

A *producer field* is a simpler alternative to a producer method and can also marry dependency injection with bean definition. A producer field may be declared by annotating a field of a managed bean class or session bean class with the `@Produces` annotation previously used on producer methods.

```
public class Shop {
   @Produces PaymentProcessor paymentProcessor = ....;
   @Produces List<Product> products = ....;
}
```

A producer field may also specify scope, name, stereotypes and/or qualifiers. It's a good way to expose state from a bean as a top-level bean.

```
public class Shop {
   @Produces @Wishlist @Named("wishlist") List<Product> products = ....;
}
```

The result can be injected or used in a Unified EL expression.

```
@Inject @Wishlist List<Product> wishlist;
```

```
<h:dataTable var="_product" value="#{wishlist}">...</h:dataTable>
```

The rules for determining the bean types of a producer field parallel the rules for producer method.

Aside from convenience, producer fields serve a specific purpose as an adapter for Java EE resources injections.

1.4.5. Java EE resources

Java EE 5 already introduced some support for dependency injection, in the form of resource injections. A resources is either a component defined in JNDI such as a data source or a container-provided component such as a persistence unit, persistence context, EJB or web service.

Naturally, there remained some mismatch with the new style of dependency injection in CDI. Most notably, resource injections rely on string-based names to qualify ambigious types, and there is no real consistency as to how a resource obtains its name (sometimes JNDI, other times from an XML descriptor or even a default name). Producer fields turned out to be an elegant adapter to rein them in and get them to participate in the CDI system just like any other injectable bean.

Producer fields have a duality in that they can both accept a standard Java EE resource injection and produce a bean that can be injected into another bean in a typesafe way. Here are some examples of these injection points. Notice that a qualifier annotation can be assigned at the injection point to adapt the string-based name required by the injection point into a typesafe qualifier for injection elsewhere.

```
@Produces @WebServiceRef(lookup="java:app/service/PaymentService")
PaymentService paymentService;
```

```
@Produces @PersistenceContext(unitName="CustomerDatabase")
@CustomerDatabase EntityManager customerDatabasePersistenceContext;
```

These resources can then be injected in the usual way.

```
@Inject PaymentService paymentService;
```

```
@Inject @CustomerDatabase EntityManager customerDatabaseEntityManager;
```

The bean type and qualifiers of the resource are determined by the producer field declaration.

While you may have to introduce a couple extra managed beans to serve as typesafe adapters, it's well worth the effort so that you can minimize the number of places in your code you have to repeat the old-style dependency injection and string-based qualifiers.

1.4.6. Built-in beans

Java EE gives you some other goodies in the form of built-in beans. A Java EE (or embeddable EJB) container provide the following built-in beans, all of which have qualifier `@Default`:

- a bean with bean type `javax.transaction.UserTransaction`, allowing injection of a reference to the JTA `UserTransaction`,
- a bean with bean type `javax.security.Principal`, allowing injection of a `Principal` representing the current caller identity.
- a bean with bean type `javax.validation.ValidationFactory`, allowing injection of the default *Bean Validation* `ValidationFactory`, and
- a bean with bean type `javax.validation.Validator`, allowing injection of a Validator for the default Bean Validation `ValidationFactory`.

Aren't you excited to start using this stuff? Wait no longer. In the next two chapters, we'll look at some examples.

JSF web application example

Let's illustrate these ideas with a full example. We're going to implement a user login/logout for an application that uses JSF. First, we'll define a request-scoped bean to hold the username and password entered during login:

```
@Named @RequestScoped
public class Credentials {
    private String username;
    private String password;

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

This bean is bound to the login prompt in the following JSF form:

```
<h:form>
   <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
      <h:outputLabel for="username">Username:</h:outputLabel>
      <h:inputText id="username" value="#{credentials.username}"/>
      <h:outputLabel for="password">Password:</h:outputLabel>
      <h:inputText id="password" value="#{credentials.password}"/>
   </h:panelGrid>
   <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
   <h:commandButton value="Logout" action="#{login.logout}" rendered="#{login.loggedIn}"/>
</h:form>
```

The actual work is done by a session-scoped bean that maintains information about the currently logged-in user and exposes the `User` entity to other beans:

```
@SessionScoped @Named
public class Login {

   @Inject Credentials credentials;
   @Inject @UserDatabase EntityManager userDatabase;
```

```
    private User user;

    public void login() {
        List<User> results = userDatabase.createQuery(
            "select u from User u where u.username = :username and u.password = :password")
            .setParameter("username", credentials.getUsername())
            .setParameter("password", credentials.getPassword())
            .getResultList();

        if (!results.isEmpty()) {
            user = results.get(0);
        }
        else {
            // perhaps add code here to report a failed login
        }
    }

    public void logout() {
        user = null;
    }

    public boolean isLoggedIn() {
        return user != null;
    }

    @Produces @LoggedIn User getCurrentUser() {
        return user;
    }

}
```

`@LoggedIn` and `@UserDatabase` are custom qualifier annotations:

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

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, PARAMETER, FIELD})
```

```
public @interface UserDatabase {}
```

We need an adapter bean to expose our typesafe `EntityManager`:

```
public class UserDatabaseProducer {
   @Produces @UserDatabase @PersistenceContext EntityManager userDatabase;
}
```

Now, any other bean can easily inject the current user, as well as other beans:

```
public class DocumentEditor {
   @Inject Document document;
   @Inject @LoggedIn User currentUser;
   @Inject @DocumentDatabase EntityManager docDatabase;

   public void save() {
      document.setCreatedBy(currentUser);
      docDatabase.persist(document);
   }
}
```

Or we can reference the current user in a JSF view:

```
<h:panelGroup rendered="#{login.loggedIn}">
   signed in as #{currentUser.username}
</h:panelGroup>
```

Hopefully, this example gives a flavor of the CDI services. In the next chapter, we'll explore examples from the CDI reference implementation, Weld, in greater depth.

Getting started Weld, the JSR-299 reference implementation

Weld, the JSR-299 RI (Reference Implementation), is being developed as part of the *Seam project* [http://seamframework.org/Weld]. You can download the latest community release of Weld from the *download page* [http://seamframework.org/Download]. Information about the Weld source code repository and instructions about how to obtain and build the source can be found on the same page.

Weld comes with two starter example applications, in addition to more specialized examples. The first, `weld-numberguess`, is a web (WAR) example containing only non-transactional managed beans. This example can be run on a wide range of servers, including JBoss AS, GlassFish, Apache Tomcat, Jetty, Google App Engine, and any compliant Java EE 6 container. The second example, `weld-translator`, is an enterprise (EAR) example that contains session beans. This example must be run on JBoss AS 5.2 or any compliant Java EE 6 container.

Both examples use JSF 2.0 as the web framework and, as such, can be found in the examples/jsf directory of the Weld distribution.

3.1. Prerequisites

To run the examples with the provided build scripts, you'll need the following:

- the latest release of Weld, which contains the examples
- Ant 1.7.0, to build and deploy the examples
- a supported runtime environment (minimum versions shown)
  - JBoss AS 5.2.0 (currenly only nightly builds of JBoss AS 5.2.0.Beta1 are available), or
  - GlassFish 3.0, or
  - Apache Tomcat 6.0.x (WAR example only), or
  - Jetty 6.1.x (WAR example only)
- (optionally) Maven 2.x, to run the examples in an embedded servlet container

Note

You'll need a full install of Ant 1.7.0. Some linux distributions only supply a partial installation of Ant which cause the build to fail. If you encounter problems, verify that ant-nodeps.jar is on the classpath.

In the next few sections, you'll be using the Ant command (ant) to invoke the Ant build script in each of the two examples to compile, assemble and deploy the example to JBoss AS and, for the WAR example, Apache Tomcat. You can also deploy the generated artifact (WAR or EAR) to any other container that supports Java EE 6, such as GlassFish V3.

If you have Maven installed, you can also use the Maven command (mvn) to compile and assemble the standalone artifact (WAR or EAR) and, for the WAR example, run it in an embedded container.

The sections below cover the steps for deploying with both Ant and Maven in detail. Let's start with JBoss AS.

3.2. Deploying to JBoss AS

To deploy the examples to JBoss AS, you'll need *JBoss AS 5.2.0.Beta1* or above. If a release of the JBoss AS 5.2 line isn't yet available, you can download a *nightly snapshot*. The reason JBoss AS 5.2.0.Beta1 or above is required is because it's the first release that has both CDI and Bean Validation support built-in, making it close enough to Java EE 6 to run the examples. The good news is that there are no additional modifications you have to make to the server. It's ready to go!

After you have downloaded JBoss AS, extract it. (We recommended renaming the folder to include the `as` qualifier so it's clear that it's the application server). You can move the extracted folder anywhere you like. Whereever it lays to rest, that's what we'll call the JBoss AS installation directory, or `JBOSS_HOME`.

```
$> unzip jboss-5.2.*.zip
$> mv jboss-5.2.*/ jboss-as-5.2
```

In order for the build scripts to know where to deploy the example, you have to tell them where to find your JBoss AS installation (i.e., `JBOSS_HOME`). Create a new file named local.build.properties in the examples directory of the Weld distribution and assign the path of your JBoss AS installation to the property key `jboss.home`, as follows:

```
jboss.home=/path/to/jboss-as-5.2
```

You're now ready to deploy your first example!

Switch to the examples/jsf/numberguess directory and execute the Ant deploy target:

```
$> cd examples/jsf/numberguess
$> ant deploy
```

If you haven't already, start JBoss AS. You can either start JBoss AS from a Linux shell:

```
$> cd /path/to/jboss-as-5.2
$> ./bin/run.sh
```

a Windows command window:

```
$> cd c:\path\to\jboss-as-5.2\bin
$> run
```

or you can start the server using an IDE, like Eclipse.

Note

If you are using Eclipse, you should seriously consider installing the *JBoss Tools* add-ons, which include a wide variety of tooling for JSR-299 and Java EE development, as well as an enhanced JBoss AS server view.

Wait a few seconds for the application to deploy (or the application server to start) and see if you can determine the most efficient approach to pinpoint the random number at the local URL *http://localhost:8080/weld-numberguess*.

Tip

The Ant build script includes additional targets for JBoss AS to deploy and undeploy the archive in either exploded or packaged format and to tidy things up.

- `ant restart` - deploy the example in exploded format to JBoss AS
- `ant explode` - update an exploded example, without restarting the deployment
- `ant deploy` - deploy the example in compressed jar format to JBoss AS
- `ant undeploy` - remove the example from JBoss AS
- `ant clean` - clean the example

The second starter example, `weld-translator`, will translate your text into Latin. (Well, not really, but the stub is there for you to implement, at least. Good luck!) To try it out, switch to the translator example directory and execute the deploy target:

```
$> cd examples/jsf/translator
```

```
$> ant deploy
```

Note

The translator uses session beans, which are packaged in an EJB module within an EAR. Java EE 6 will allow session beans to be deployed in WAR modules, but that's a topic for future chapter.

Again, wait a few seconds for the application to deploy (if you are bored, read the log messages), and visit *http://localhost:8080/weld-translator* to begin pseudo-translating.

TODO Insert note about upgrading the Weld deployer that directs reader to section on the JBoss AS environment

3.3. Deploying to GlassFish

Deploying to GlassFish should be easy, right? After all, it's the Java EE 6 reference implementation. Since it's the Java EE 6 reference implementation, that means it also bundles the JSR-299 reference implementation, Weld! So yes, it's very easy.

To deploy the examples to GlassFish, you'll need the final *GlassFish V3* release (the preview release won't do). If the final release isn't yet available, you can download a *promoted build* in the meantime. Select the b69 preview release or above that ends in either -unix.sh or -windows.exe depending on your platform. After the download is complete, execute the installer. On Linux/Unix, you'll need to first make the script executable.

```
$> chmod 755 glassfish-v3-b69-unix.sh
$> ./glassfish-v3-b69-unix.sh
```

On Windows you can just click on the executable. Follow the instructions in the installer. It will create a single domain named `domain1`. You'll use that domain to deploy the example. We recommend that you choose 7070 as the main HTTP port to avoid conflicts with a running instance of JBoss AS (or Apache Tomcat).

If you've deployed either of the starter examples, `weld-numberguess` or `weld-translator`, to JBoss AS, then you already have the deployable artifact you need. If not, switch to either of the two directories and build it.

```
$> cd examples/jsf/numberguess (or examples/jsf/translator)
$> ant package
```

The deployable archive for the `weld-numberguess`, named weld-numberguess.war, ends up in the example's target directory. The archive for the `weld-translator` example, named weld-translator.ear, ends up in the example's ear/target directory. All you need to do now is deploy them to GlassFish.

You deploy applications to GlassFish using the *GlassFish Admin Console*. To get the Admin Console running, you need to start a GlassFish domain, in our case `domain1`. Switch to the bin folder in the directory where you installed GlassFish and execute the following command:

```
$> asadmin start-domain domain1
```

After a few seconds you can visit the Admin Console in the browser at the URL *http://localhost:4848*. In the tree on the left-hand side of the page, click on "Applications", then click on the "Deploy..." button under the heading "Applications" and select the deployable artifact for either of the two examples. The deployer should recognize that you have selected a Java EE artifact and allow you to start it. You can see the examples running at either *http://localhost:7070/weld-numberguess* or *http://localhost:7070/weld-translator*, depending on which example you deployed.

The reason the same artifact can be deployed to both JBoss AS and GlassFish, without any modifications, is because all of the features being used are part of the standard platform. And what a capable platform it has become!

3.4. Deploying to Apache Tomcat

Technically, CDI doesn't support servlet containers. That's because the contexts and dependency injection services are provided by the Java EE container, not the Servlet API. However, CDI implementations can choose to support servlet containers through an extension. Weld does this.

Weld ships with a Servlet listener which bootstraps the CDI environment, registers the BeanManager in JNDI and provides injection into servlets. Basically, it emulates the work done by the Java EE container. The only limitation is that you cannot use enterprise features such as session beans and container-managed transactions.

Let's give the Weld servlet extension a spin on Apache Tomcat. First, you'll need to download Tomcat 6.0.18 or later from *tomcat.apache.org* and extract it.

```
$> unzip apache-tomcat-6.0.18.zip
```

You have two choices for how you can deploy the application to Tomcat. You can deploy it by pushing the artifact to the hot deploy directory using Ant or you can deploy to the server across HTTP using a Maven plugin. The Ant approach doesn't require that you have Maven installed, so we'll start there. If you want to use Maven, you can just skip ahead.

3.4.1. Deploying with Ant

In order for Ant to push the artifact to the Tomcat hot deploy directory, it needs to know where the Tomcat installation is located. Again, we need to setup a propery in the local.build.properties file in the examples directory of the Weld distribution. If you haven't yet created this file, do so now. Then assign the path of your Tomcat installation to the property key tomcat.home.

```
tomcat.home=/path/to/apache-tomcat-6
```

Now you're ready to deploy the numberguess example to Tomcat!

Change to the examples/jsf/numberguess directory again and run the Ant deploy target for Tomcat:

```
$> cd examples/jsf/numberguess
$> ant tomcat.deploy
```

Tip

The Ant build script includes additional targets for Tomcat to deploy and undeploy the archive in either exploded or packaged format. They are the same target names used for JBoss AS, prefixed with "tomcat.".

- `ant tomcat.restart` - deploy the example in exploded format to Tomcat
- `ant tomcat.explode` - update an exploded example, without restarting the deployment
- `ant tomcat.deploy` - deploy the example in compressed jar format to Tomcat
- `ant tomcat.undeploy` - remove the example from Tomcat

If you haven't already, start Tomcat. You can either start Tomcat from a Linux shell:

```
$> cd /path/to/apache-tomcat-6
$> ./bin/start.sh
```

a Windows command window:

```
$> cd c:\path\to\apache-tomcat-6\bin
```

```
$> start
```

or you can start the server using an IDE, like Eclipse.

Wait a few seconds for the application to deploy (or the application server to start) and see if you can figure out the most efficient approach to pinpoint the random number at the local URL *http://localhost:8080/weld-numberguess*!

3.4.2. Deploying with Maven

You can also deploy the application to Tomcat using Maven. This section is a bit more advanced, so skip it unless you are itching to use Maven natively. Of course, you'll first need to make sure that you have Maven installed on your path, similar to how you setup Ant.

The Maven plugin communicates with Tomcat over HTTP, so it doesn't care where you have installed Tomcat. However, the plugin configuration assumes you are running Tomcat in its default configuration, with a hostname of localhost and port 8080. The readme.txt file in the example directory has information about how to modify the Maven settings to accomodate a different setup.

To allow Maven to communicate with Tomcat over HTTP, edit the conf/tomcat-users.xml file in your Tomcat installation and add the following line:

```
<user username="admin" password="" roles="manager"/>
```

Restart Tomcat. You can now deploy the application to Tomcat with Maven using this command:

```
$> mvn compile war:exploded tomcat:exploded -Ptomcat
```

Once the application is deployed, you can redeploy it using this command:

```
$> mvn tomcat:redeploy -Ptomcat
```

The `-Ptomcat` argument activates the tomcat profile defined in the Maven POM (pom.xml). Among other things, this profile activates the Tomcat plugin.

Rather than shipping the container off to a standalone Tomcat installation, you can also execute the application in an embedded Tomcat 6 container:

```
$> mvn war:inplace tomcat:run -Ptomcat
```

The advantage of using the embedded server is that changes to assets in src/main/webapp take affect immediately. If a change to a webapp configuration file is made, the application may automatically redeploy (depending on the plugin configuration). If you make a change to a classpath resource, you need to execute a build:

```
$> mvn compile war:inplace -Ptomcat
```

There are several other Maven goals that you can use if you are hacking on the example, which are documented in the example's readme.txt file.

3.5. Deploying to Jetty

Support for Jetty in the examples is a more recent addition. Since Jetty is traditionally used with Maven, there are no Ant targets. You must invoke the Maven build directly to deploy the examples to Jetty out of the box. Also, only the `weld-numberguess` example is configured for Jetty support at the time of this writing.

If you've read through the entire Tomcat section, then you're all ready to go. The Maven build parallels the embedded Tomcat deployment. If not, don't worry. We'll still go over everything in this section that you need to know.

The Maven POM (pom.xml) includes a profile named jetty that activates the Maven Jetty plugin, which you can use to start Jetty in embedded mode and deploy the application in place. You don't need anything else installed except to have the Maven command (mvn) on your path. The rest will be downloaded from the internet when the build is run.

To run the `weld-numberguess` example on Jetty, switch to the example directory and execute the inplace goal of the Maven WAR plugin followed by the run goal of the Maven Jetty plugin with the jetty profile enabled, as follows:

```
$> cd examples/jsf/numberguess
$> mvn war:inplace jetty:run -Pjetty
```

The log output of Jetty will be shown in the console. Once Jetty reports that the application has deployed, you can access it at the following local URL: *http://localhost:9090/weld-numberguess*. The port is defined in the Maven Jetty plugin configuration within the jetty profile.

Any changes to assets in src/main/webapp take affect immediately. If a change to a webapp configuration file is made, the application may automatically redeploy. The redeploy behavior can be fined tuned in the plugin configuration. If you make a change to a classpath resource, you need to execute a build and the inplace goal of the Maven WAR plugin, again with the jetty profile enabled.

```
$> mvn compile war:inplace -Pjetty
```

The war:inplace goal copies the compiled classes and JARs inside src/main/webapp, under WEB-INF/classes and WEB-INF/lib, respectively, mixing source and compiled files. However, the build does work around these temporary files by excluding them from the packaged WAR and cleaning them during the Maven clean phase.

You have two options if you want to run the example on Jetty from the IDE. You can either install the m2eclispe[link] plugin and run the goals as described above. Your other option is to start the Jetty container from a Java application.

First, initialize the Eclipse project:

```
$> mvn clean eclipse:clean eclipse:eclipse -Pjetty-ide
```

Next, assemble all the necessary resources under src/main/webapp:

```
$> mvn war:inplace -Pjetty-ide
```

Now, you are ready to run the server in Eclipse. Import the project into your Eclipse workspace using "Import Existing Project into Workspace. Then, find the start class in src/jetty/java and run its main method as a Java Application. Jetty will launch. You can view the application at the following local URL: *http://localhost:8080*. Pay particular attention to the port in the URL and the lack of a trailing context path.

Now that you have gotten the starter applications deployed on the server of your choice, you probably want to know a little bit about how they actually work. It's time to pull the covers back and dive into the internals of these two examples. Let's start with the simpler of the two examples, `weld-numberguess`.

3.6. The numberguess example in depth

In the numberguess application you get given 10 attempts to guess a number between 1 and 100. After each attempt, you will be told whether you are too high or too low.

The numberguess example is comprised of a number of beans, configuration files and Facelets (JSF) views, packaged as a WAR module. Let's start by examining the configuration files.

All the configuration files for this example are located in `WEB-INF/`, which can be found in the `src/main/webapp` directory of the example. First, we have the JSF 2.0 version of `faces-config.xml`. A standardized version of Facelets is the default view handler in JSF 2.0, so there's really nothing that we have to configure. Thus, the configuration consists of only the root element.

```
<faces-config version="2.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
</faces-config>
```

There's also an empty `beans.xml` file, which tells the container to look for beans in this application and to activate the CDI services.

Finally, there's the familiar `web.xml`:

```
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <display-name>weld-jsf-numberguess-war</display-name>
  <description>Weld JSF numberguess example (WAR)</description>
                                                (1)
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
                                                (2)
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
                                                (3)
  <context-param>
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
  </context-param>
                                                (4)
  <session-config>
```

```
    <session-timeout>10</session-timeout>
  </session-config>

</web-app>
```

1. Enable and initialize the JSF servlet
2. Configure requests for URLs ending in `.jsf` to be handled by JSF
3. Tell JSF that we will be giving our JSF views (Facelets templates) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

Note

This demo uses JSF 2 as the view framework, but you can use Weld with any Servlet-based web framework, such as JSF 1.2 or Wicket.

Let's take a look at the main JSF view, src/main/webapp/home.xhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">


  <ui:composition template="/template.xhtml">                    (1)
    <ui:define name="content">
      <h1>Guess a number...</h1>
      <h:form id="numberGuess">

        <div style="color: red">                                 (2)
          <h:messages id="messages" globalOnly="false"/>
          <h:outputText id="Higher" value="Higher!"
            rendered="#{game.number gt game.guess and game.guess ne 0}"/>
          <h:outputText id="Lower" value="Lower!"
            rendered="#{game.number lt game.guess and game.guess ne 0}"/>
        </div>


        <div>                                                    (3)
          I'm thinking of a number between #{game.smallest} and #{game.biggest}.
          You have #{game.remainingGuesses} guesses remaining.
        </div>
```

```
            <div>
              Your guess:
              <h:inputText id="inputGuess" value="#{game.guess}"          (4)
                size="3" required="true" disabled="#{game.number eq game.guess}"
                validator="#{game.validateNumberRange}"/>                  (5)
              <h:commandButton id="guessButton" value="Guess"            (6)
                action="#{game.check}" disabled="#{game.number eq game.guess}"/>
            </div>
            <div>
                         <h:commandButton  id="restartButton"  value="Reset"  action="#{game.reset}"
 immediate="true"/>
            </div>
          </h:form>
        </ui:define>
      </ui:composition>
</html>
```

1. Facelets is the built-in templating language for JSF. Here we are wrapping our page in a template which defines the layout.
2. There are a number of messages which can be sent to the user, "Higher!", "Lower!" and "Correct!"
3. As the user guesses, the range of numbers they can guess gets smaller - this sentance changes to make sure they know the number range of a valid guess.
4. This input field is bound to a bean property using a value expression.
5. A validator binding is used to make sure the user doesn't accidentally input a number outside of the range in which they can guess - if the validator wasn't here, the user might use up a guess on an out of bounds number.
6. And, of course, there must be a way for the user to send their guess to the server. Here we bind to an action method on the bean.

The example exists of 4 classes, the first two of which are qualifiers. First, there is the `@Random` qualifier, used for injecting a random number:

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

There is also the `@MaxNumber` qualifier, used for injecting the maximum number that can be injected:

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

The application-scoped `Generator` class is responsible for creating the random number, via a producer method. It also exposes the maximum possible number via a producer method:

```
@ApplicationScoped
public class Generator implements Serializable {

  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 `Generator` is application scoped, so we don't get a different random each time.

Note

The package declaration and imports have been excluded from these listings. The complete listing is available in the example source code.

The final bean in the application is the session-scoped `Game` class. This is the primary entry point of the application. It's responsible for setting up or resetting the game, capturing and validating the user's guess and providing feedback to the user with a `FacesMessage`. We've used constructor injection to initialize the game with a random number.

You'll notice that we've also added the `@Named` annotation to this class. This annotation is only required when you want to make the bean accessible to a JSF view via EL (i.e., #{game}).

```
@Named
@SessionScoped
public class Game implements Serializable {

  private int number;
  private int guess;
  private int smallest;
  private int biggest;
  private int remainingGuesses;

  @Inject @MaxNumber private int maxNumber;
  @Inject @Random Instance<Integer> randomNumber;

  public Game() {}

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

  @PostConstruct
  public void reset() {
    this.smallest = 0;
    this.guess = 0;
    this.remainingGuesses = 10;
    this.biggest = maxNumber;
    this.number = randomNumber.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 < smallest || input > biggest) {
      ((UIInput) toValidate).setValid(false);

      FacesMessage message = new FacesMessage("Invalid guess");
      context.addMessage(toValidate.getClientId(context), message);
    }
  }

  public int getNumber() {
    return number;
  }

  public int getGuess() {
    return guess;
  }

  public void setGuess(int guess) {
    this.guess = guess;
  }

  public int getSmallest() {
    return smallest;
  }

  public int getBiggest() {
    return biggest;
  }

  public int getRemainingGuesses() {
    return remainingGuesses;
  }

}
```

3.6.1. The numberguess example in Apache Tomcat or Jetty

A couple of modifications must be made to the numberguess artifact in order to deploy it to Tomcat or Jetty. First, Weld must be deployed as a Web Application library under `WEB-INF/lib` since the

servlet container does not provide the CDI services. For your convenience we provide a single JAR suitable for running Weld in any servlet container (including Jetty), `weld-servlet.jar`.

Tip

You must also include the JARs for JSF, EL, and the common annotations (`jsr250-api.jar`), all of which are provided by the Java EE platorm (a Java EE application server). Are you starting to appreciate why a Java EE platform is worth using?

Second, we need to explicitly specify the servlet listener in `web.xml`, again because the container isn't doing this stuff for you. The servlet listener boots Weld and controls it's interaction with requests.

```
<listener>
   <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener>
```

When Weld boots, it places the `javax.enterprise.inject.spi.BeanManager`, the portable SPI for obtaining bean instances, in the ServletContext under a variable name equal to the fully-qualified interface name. You generally don't need to access this interface, but Weld makes use of it.

3.6.2. The numberguess example for Apache Wicket

Weld includes a number of portable extensions for JSR-299, including an extension for Wicket, which allows you to inject beans into Wicket components and leverage the conversation context. In this section, we'll walk you through the Wicket version of the numberguess example.

Tip

You may want to review the Wicket documentation at *http://wicket.apache.org/* before reading this section, if you aren't already familiar with the framework.

Wicket is another environment that relies on the Weld servlet extension. The use of *Jetty* [http://jetty.mortbay.org] is common in the Wicket community, and is thus chosen here as the runtime container. You've seen already that Jetty is perfectly capable of running CDI applications with Weld add-ons, and this environment is no different.

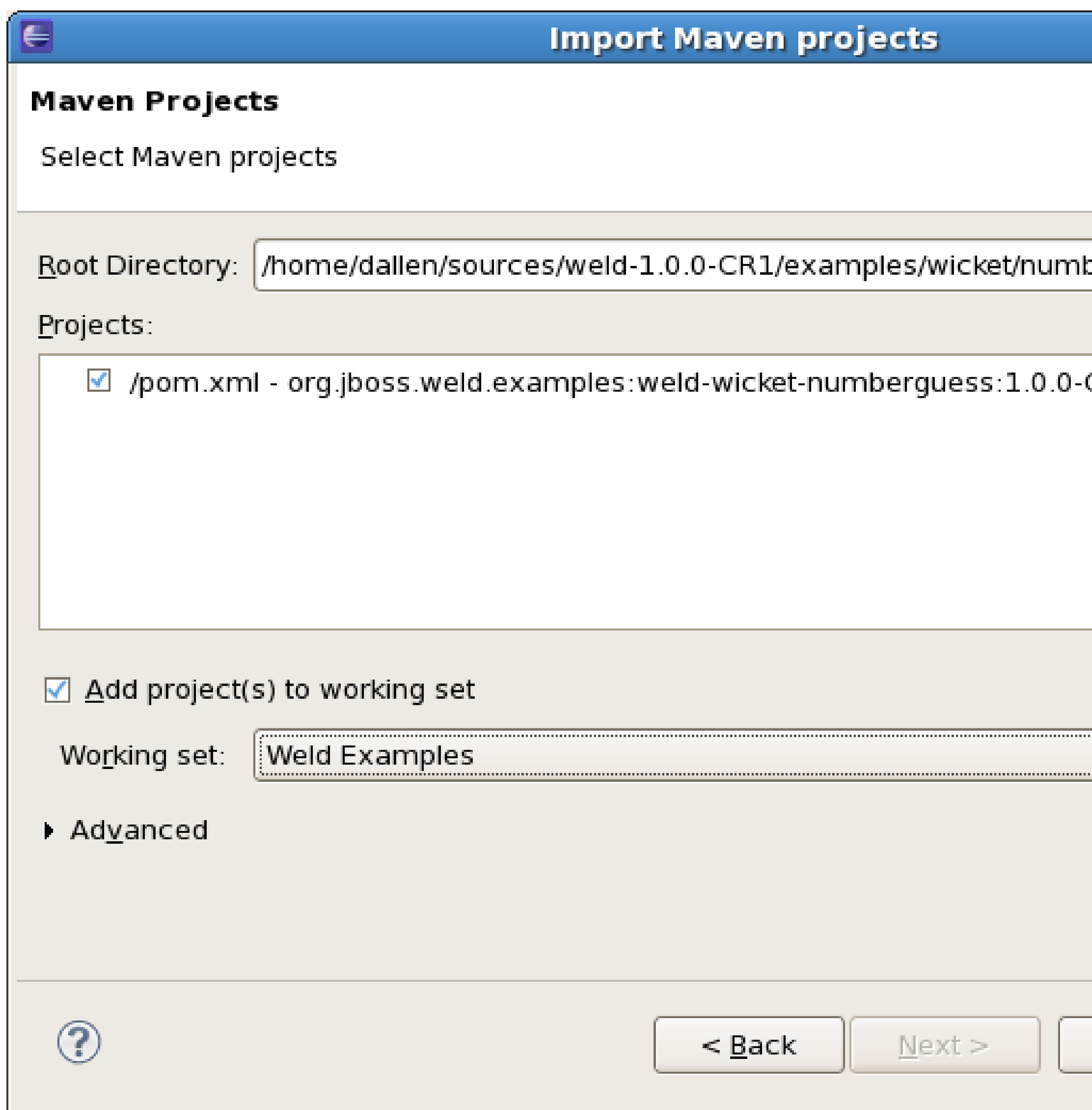
Note

We'll also be using the Eclipse IDE in these examples. Instructions are provided later for running the example from the commandline, but since you'll likely need to do more than just deploy examples, we'll get setup in this full development environment.

3.6.2.1. Creating the Eclipse project

To use the Wicket example in Eclipse, you have one of two choices. You can either use a Maven plugin to generate a regular Eclipse Web project, or you can open the example natively using the *m2eclipse plugin*. Since the Weld source code relies so heavily on Maven, we encourage you to bite the bullet and adopt the m2eclipse plugin. Both approaches are described here for your convenience..

If you have m2eclipse installed, you can open any Maven project directly. From within Eclipse, select *File -> Import... -> Maven Projects*. Then, browse to the location of the Wicket numberguess example. You should see that Eclipse recognizes the existance of a Maven project.



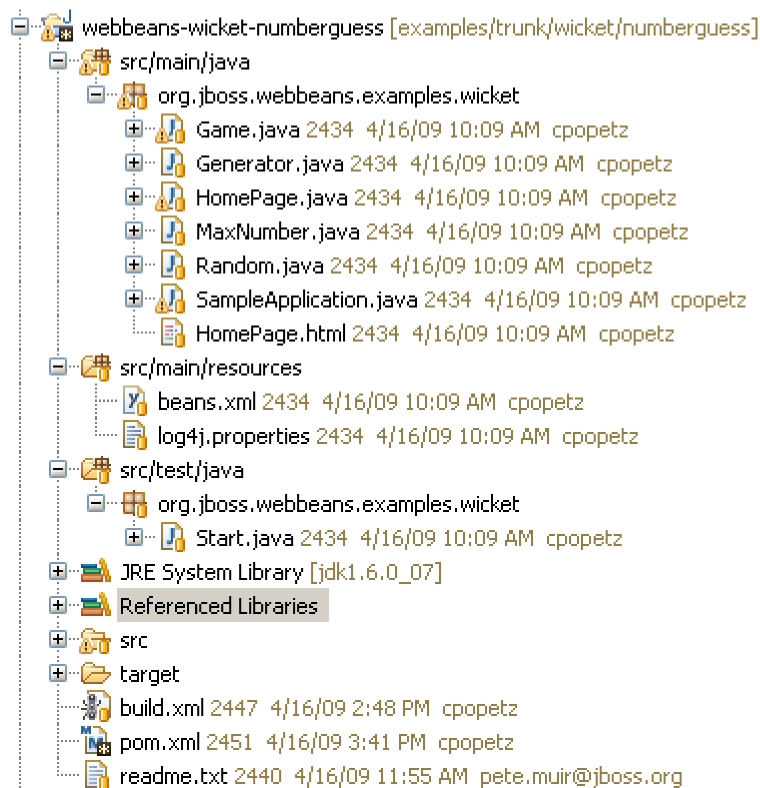
This will create a project in your workspace called `weld-wicket-numberguess`.

You'll notice after importing, the project has a build error. That's because we need to enable a Maven profile. Right-click on the project and select *Properties*, then select the *Maven* tab in the window that appears. In the form field labeled "Active Maven Profiles (comma separated):", type `jetty`. That will enable some extra dependencies that allow the project to compile. You're now ready to develop!

If you are not using the m2eclipse plugin, you have to follow different steps to import the project. First, switch into the Wicket numberguess example, then execute the Maven Eclipse plugin with the jetty profile activated, as follows:

```
$> cd examples/wicket/numberguess
mvn -Pjetty eclipse:eclipse
```

Then, from Eclipse, choose *File -> Import... -> General -> Existing Projects into Workspace*, select the root directory of the numberguess example, and click Finish. This will create a project in your workspace called `weld-wicket-numberguess`.



It's time to get the example running!

3.6.2.2. Running the example from Eclipse

This project follows the `wicket-quickstart` approach of creating an instance of Jetty in the `Start` class. So running the example is as simple as right-clicking on that Start class in `src/test/java` in the *Package Explorer* and choosing *Run as Java Application*. You should see console output related to Jetty starting up; then visit able `http://localhost:8080` to view the app. To debug choose *Debug as Java Application* instead.

3.6.2.3. Running the example from the command line in JBoss AS or Tomcat

This example can also be deployed from the command line in a (similar to the other examples). Assuming you have set up the `local.build.properties` file in the `examples` directory to specify the location of JBoss AS or Tomcat, as previously described, you can run:

```
$> ant deploy
```

to deploy the example to JBoss AS, and:

```
$> ant tomcat.deploy
```

to deploy the example to Tomcat. You can then access application at `http://localhost:8080/weld-numberguess-wicket`.

Alternatively, you can run the application in place on an embedded Jetty container using the following Maven command:

```
$> mvn jetty:run -Pjetty
```

Enough toying with deployment, let's dive into the code.

3.6.2.4. Understanding the code

The code in the wicket numberguess example is very similar to the JSF-based numberguess example. The business layer is identical! Where things differ is in view binding. JSF uses Unified EL expressions to bind XML-based view layer components in JSF views to beans. In contrast, Wicket defines its components in Java. These Java-based view components have a one-to-one mapping with HTML elements in an adjacent (pure) HTML file. All view logic, including binding of components to models and controlling the response of view actions, is handled in Java.

The integration of Weld with Wicket takes advantage of the same qualifier annotations used in your business layer to provide injection into your `WebPage` subclass (or into other custom Wicket component subclasses).

Here's where things differ from the JSF numberguess example:

- Each wicket application must have a `WeldApplication` subclass. In our case, our application class is `NumberGuessApplication`:

```
public class NumberGuessApplication extends WeldApplication {
```

```
  @Override public Class getHomePage() {
    return HomePage.class;
  }
}
```

This class specifies which page Wicket should treat as our home page, in our case, `HomePage.class`

- In `HomePage`, we see typical Wicket code to set up page elements. The bit that is interesting is the injection of the `Game` bean:

```
@Inject Game game;
```

The `Game` bean is can then be used, for example, by the code for submitting a guess:

```
final Component guessButton = new AjaxButton("GuessButton") {
  protected void onSubmit(AjaxRequestTarget target, Form form) {
    if (game.check()) {
      info("Correct!");
      setVisible(false);
      prompt.setVisible(false);
      guessLabel.setVisible(false);
      inputGuess.setVisible(false);
    }
    else if (game.getRemainingGuesses() == 0) {
      info("Sorry, the answer was " + game.getNumber());
      setVisible(false);
      guessLabel.setVisible(false);
      inputGuess.setVisible(false);
    }
    else if (game.getNumber() > game.getGuess()) {
      info("Higher!");
    }
    else if (game.getNumber() < game.getGuess()) {
      info("Lower");
    }
    target.addComponent(form);
  }
};
```

Note

All injections may be serialized; actual storage of the bean is managed by JSR-299. Note that Wicket components, like the `HomePage` and it subcomponents, are *not* JSR-299 beans.

Wicket components allow injection, but they *cannot* use interceptors, decorators and lifecycle callbacks such as `@PostConstruct` or `@Initializer` methods. The components would need to delegate to actual beans to leverage these features.

- The example uses AJAX for processing of button events, and dynamically hides buttons that are no longer relevant, for example when the user has won the game.
- In order to activate Wicket for this webapp, the Wicket filter is added to web.xml, and our application class is specified in web.xml:

```
<filter>
  <filter-name>Wicket Filter</filter-name>
  <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>org.jboss.weld.examples.wicket.NumberGuessApplication</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Wicket Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The servlet listener is still required, as in the Tomcat example, to bootstrap CDI when Jetty starts and to hook CDI into the Jetty servlet request and session lifecycles. However, rather than putting it into the web.xml, it is placed into an override file, src/main/webapp/WEB-INF/jetty-additions-to-web.xml, that is passed to Jetty as extra descriptor to be appended to the web.xml configuration.

```
<web-app version="2.4" ...>
  <listener>
    <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>
  </listener>
```

```
</web-app>
```

3.6.3. The numberguess example for Java SE with Swing

This example can be found in the `examples/se/numberguess` folder of the Weld distribution.

To run this example:

- Open a command line/terminal window in the `examples/se/numberguess` directory
- Ensure that Maven 2 is installed and in your PATH
- Ensure that the `JAVA_HOME` environment variable is pointing to your JDK installation
- execute the following command

```
$> mvn -Drun
```

There is an empty `beans.xml` file in the root package (`src/main/resources/beans.xml`), which marks this application as a CDI application.

The game's main logic is located in `Game.java`. Here is the code for that class, highlighting the changes made from the web application version:

```
@ApplicationScoped                                            (1)(2)
public class Game implements Serializable {

   private int number;
   private int guess;
   private int smallest;
   private int biggest;
   private int remainingGuesses;
   private boolean validNumberRange = true;

   @Inject @MaxNumber private int maxNumber;

   @Inject Generator rndGenerator;

   ...

   public boolean isValidNumberRange() {
      return validNumberRange;
   }
```

```
    public boolean isGameWon() {
        return guess == number;
    }

    public boolean isGameLost() {
        return guess != number && remainingGuesses <= 0;      (3)
    }

    public boolean check() {
        boolean result = false;

        if (checkNewNumberRangeIsValid()) {
            if (guess > number) {
                biggest = guess - 1;
            }

            if (guess < number) {
                smallest = guess + 1;
            }                                  (4)

            if (guess == number) {
                result = true;
            }

            remainingGuesses--;
        }

        return result;
    }

    private boolean checkNewNumberRangeIsValid() {
        return validNumberRange = ((guess >= smallest) && (guess <= biggest));
    }

    @PostConstruct
    public void reset() {
        this.smallest = 0;
        ...
        this.number = rndGenerator.next();
    }
}
```

(1) The bean is application scoped instead of session scoped, since an instance of the application represents a single 'session'.

(2) The bean is not named, since it doesn't need to be accessed via EL

(3) There is no JSF `FacesContext` to add messages to. Instead the `Game` class provides additional information about the state of the current game including:

- If the game has been won or lost
- If the most recent guess was invalid

This allows the Swing UI to query the state of the game, which it does indirectly via a class called `MessageGenerator`, in order to determine the appropriate messages to display to the user during the game.

(4) Validation of user input is performed during the `check()` method, since there is no dedicated validation phase

(5) The `reset()` method makes a call to the injected `rndGenerator` in order to get the random number at the start of each game. It cannot use `manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random>(){})` as the JSF example does because there will not be any active contexts like there is during a JSF request.

The `MessageGenerator` class depends on the current instance of `Game`, and queries its state in order to determine the appropriate messages to provide as the prompt for the user's next guess and the response to the previous guess. The code for `MessageGenerator` is as follows:

```
public class MessageGenerator {
   @Inject Game game;
                                                        (1)

  public String getChallengeMessage() {

      StringBuilder challengeMsg = new StringBuilder( "I'm thinking of a numb (2) er between " );
      challengeMsg.append( game.getSmallest() );
      challengeMsg.append( " and " );
      challengeMsg.append( game.getBiggest() );
      challengeMsg.append( ". Can you guess what it is?" );

      return challengeMsg.toString();
  }

  public String getResultMessage() {
      if ( game.isGameWon() ) {

          return "You guess it! The number was " + game.getNumber();     (3)
      }
```

```
      else if ( game.isGameLost() ) {
         return "You are fail! The number was " + game.getNumber();
      }
      else if ( ! game.isValidNumberRange() ) {
         return "Invalid number range!";
      }
      else if ( game.getRemainingGuesses() == Game.MAX_NUM_GUESSES ) {
         return "What is your first guess?";
      }
      else {
         String direction = null;

         if ( game.getGuess() < game.getNumber() ) {
            direction = "Higher";
         }
         else {
            direction = "Lower";
         }

         return direction + "! You have " + game.getRemainingGuesses() + " guesses left.";
      }
   }

}
```

1. The instance of `Game` for the application is injected here.
2. The `Game`'s state is interrogated to determine the appropriate challenge message.
3. And again to determine whether to congratulate, console or encourage the user to continue.

Finally we come to the `NumberGuessFrame` class which provides the Swing front end to our guessing game.

```
public class NumberGuessFrame extends javax.swing.JFrame {


   private @Inject Game game;                                    (1)

   private @Inject MessageGenerator msgGenerator;                      (2)


   public void start( @Observes @Deployed Manager manager ) {          (3)
      java.awt.EventQueue.invokeLater( new Runnable()
         {
            public void run()
```

```
            {
                initComponents();
                setVisible( true );
            }
        } );
    }

    private void initComponents() {
                                                            (4)
        buttonPanel = new javax.swing.JPanel();
        mainMsgPanel = new javax.swing.JPanel();
        mainLabel = new javax.swing.JLabel();
        messageLabel = new javax.swing.JLabel();
        guessText = new javax.swing.JTextField();
        ...
        mainLabel.setText(msgGenerator.getChallengeMessage());
        mainMsgPanel.add(mainLabel);

        messageLabel.setText(msgGenerator.getResultMessage());
        mainMsgPanel.add(messageLabel);
        ...
    }

    private void guessButtonActionPerformed( java.awt.event.ActionEvent evt ) {
        int guess =  Integer.parseInt(guessText.getText());          (5)

        game.setGuess( guess );
        game.check();
        refreshUI();

    }

    private void replayBtnActionPerformed( java.awt.event.ActionEvent evt ) {
       game.reset();
       refreshUI();                                            (6)
    }

    private void refreshUI() {
        mainLabel.setText( msgGenerator.getChallengeMessage() );
        messageLabel.setText( msgGenerator.getResultMessage() );
        guessText.setText( "" );                                   (7)
        guessesLeftBar.setValue( game.getRemainingGuesses() );
```

```
        guessText.requestFocus();
    }

    // swing components
    private javax.swing.JPanel borderPanel;
    ...
    private javax.swing.JButton replayBtn;

}
```

1. The injected instance of the game (logic and state).
2. The injected message generator for UI messages.
3. This application is started in the usual CDI SE way, by observing the `@Deployed Manager` event.
4. This method initialises all of the Swing components. Note the use of the `msgGenerator`.
5. `guessButtonActionPerformed` is called when the 'Guess' button is clicked, and it does the following:

    - Gets the guess entered by the user and sets it as the current guess in the `Game`
    - Calls `game.check()` to validate and perform one 'turn' of the game
    - Calls `refreshUI`. If there were validation errors with the input, this will have been captured during `game.check()` and as such will be reflected in the messeges returned by `MessageGenerator` and subsequently presented to the user. If there are no validation errors then the user will be told to guess again (higher or lower) or that the game has ended either in a win (correct guess) or a loss (ran out of guesses).
6. `replayBtnActionPerformed` simply calls `game.reset()` to start a new game and refreshes the messages in the UI.
7. `refreshUI` uses the `MessageGenerator` to update the messages to the user based on the current state of the Game.

3.7. The translator example in depth

The translator example will take any sentences you enter, and translate them to Latin. (Well, not really, but the stub is there for you to implement, at least. Good luck!)

The translator example is built as an EAR and contains EJBs. As a result, it's structure is more complex than the numberguess example.

Note

Jave EE 6, which bundles EJB 3.1, allows you to package EJBs in a WAR, which will make this structure much simpler! Still, there are other advantages of using an EAR.

First, let's take a look at the EAR aggregator, which is located in the example's `ear` directory. Maven automatically generates the `application.xml` for us from this plugin configuration:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-ear-plugin</artifactId>
   <configuration>
      <modules>
         <webModule>
            <groupId>org.jboss.weld.examples.jsf.translator</groupId>
            <artifactId>weld-jsf-translator-war</artifactId>
            <contextRoot>/weld-translator</contextRoot>
         </webModule>
      </modules>
   </configuration>
</plugin>
```

This configuration overrides the web context path, resulting in this application URL: *http://localhost:8080/weld-translator*.

Tip

If you weren't using Maven to generate these files, you would need `META-INF/application.xml`:

```
<application version="5"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/application_5.xsd">

   <display-name>weld-jsf-translator-ear</display-name>
   <description>The Weld JSF translator example (EAR)</description>
```

```
  <module>
    <web>
      <web-uri>weld-translator.war</web-uri>
      <context-root>/weld-translator</context-root>
    </web>
  </module>
  <module>
    <ejb>weld-translator.jar</ejb>
  </module>
</application>
```

Next, lets look at the WAR, which is located in the example's `war` directory. Just as in the numberguess example, we have a `faces-config.xml` for JSF 2.0 and a `web.xml` (to activate JSF) under WEB-INF, both sourced from `src/main/webapp/WEB-INF`.

More intersting is the JSF view used to translate text. Just as in the numberguess example we have a template, which surrounds the form (ommitted here for brevity):

```
<h:form id="translator">

  <table>
    <tr align="center" style="font-weight: bold">
      <td>
        Your text
      </td>
      <td>
        Translation
      </td>
    </tr>
    <tr>
      <td>
      <h:inputTextarea id="text" value="#{translator.text}" required="true" rows="5" cols="80"/>
      </td>
      <td>
        <h:outputText value="#{translator.translatedText}"/>
      </td>
    </tr>
  </table>
  <div>
    <h:commandButton id="button" value="Translate" action="#{translator.translate}"/>
  </div>
```

```
</h:form>
```

The user can enter some text in the left-hand textarea, and hit the translate button to see the result to the right.

Finally, let's look at the EJB module, which is located in the example's `ejb` directory. In `src/main/resources/META-INF` there is just an empty `beans.xml`, used to mark the archive as containing beans.

We've saved the most interesting bit to last, the code! The project has two simple beans, `SentenceParser` and `TextTranslator` and two session beans, `TranslatorControllerBean` and `SentenceTranslator`. You should be getting quite familiar with what a bean looks like by now, so we'll just highlight the most interesting bits here.

Both `SentenceParser` and `TextTranslator` are dependent beans, and `TextTranslator` uses constructor injection:

```
public class TextTranslator implements Serializable {

  private SentenceParser sentenceParser;

  @EJB private Translator translator;

  @Inject public TextTranslator(SentenceParser sentenceParser) {
    this.sentenceParser = sentenceParser;
  }

  public String translate(String text) {
    StringBuilder sb = new StringBuilder();
    for (String sentence: sentenceParser.parse(text)) {
      sb.append(translator.translate(sentence)).append(". ");
    }
    return sb.toString().trim();
  }

}
```

`TextTranslator` uses the simple bean (really just a plain Java class!) `SentenceParser` to parse the sentence and then calls on the stateless bean with the local business interface `Translator` to perform the translation. That's where the magic happens. Of course, we couldn't develop a full translator, but it's convincing enough to anyone who doesn't understand Latin!

```
@Stateless
```

```
public class SentenceTranslator implements Translator {

   public String translate(String sentence) {
      return "Lorem ipsum dolor sit amet";
   }

}
```

Finally, there is UI orientated controller. This is a request scoped, named, stateful session bean, which injects the translator. It collects the text from the user and dispatches it to the translator. The bean also has getters and setters for all the fields on the page.

```
@Stateful
@RequestScoped
@Named("translator")
public class TranslatorControllerBean implements TranslatorController {

   @Inject private TextTranslator translator;

   private String inputText;

   private String translatedText;

   public void translate() {
      translatedText = translator.translate(inputText);
   }

   public String getText() {
      return inputText;
   }

   public void setText(String text) {
      this.inputText = text;
   }

   public String getTranslatedText() {
      return translatedText;
   }

   @Remove public void remove() {}

}
```

[CHECK is this still accurate?] Since this is a stateful session bean, we have to have a remove method. The bean manager will call the remove method for you when the bean is destroyed; in this case at the end of the request.

You'll notice in this example the smooth integration between the existing `@EJB` resource injection annotation and the new `@Inject` annotation for injecting any managed or session bean. The tight integration with Java EE is a key value proposition of CDI (not to mention a requirement).

That concludes our short tour of the Weld starter examples. For more information on Weld, or to help out with development, please visit *http://www.seamframework.org/Weld/Development*.

We need help in all areas - bug fixing, writing new features, writing examples and translating this reference guide.

Dependency injection and programmatic lookup

One of the most significant features of CDI, certainly the most recognized, is dependency injection. But not just dependency injection; type-safe dependency injection. In this chapter, you'll learn how CDI is able to leverage the Java type system and annotations to build a dependency injection strategy that is both strongly typed and keeps the implementation hidden from the client.

4.1. Where you can @Inject

Injections are declared using the JSR-330 annotation, `@Inject`, along with an optional set of qualifiers annotations. CDI supports three primary mechanisms for dependency injection during bean construction:

Bean constructor parameter injection:

```
public class Checkout {

  private final ShoppingCart cart;

  @Inject
  public Checkout(ShoppingCart cart) {
    this.cart = cart;
  }

}
```

Note

A bean can only have one injectable constructor.

Initializer method parameter injection:

```
public class Checkout {

  private ShoppingCart cart;

  @Inject
  void setShoppingCart(ShoppingCart cart) {
```

```
    this.cart = cart;
  }

}
```

Note

A bean can have multiple initializer methods. If the bean is a session bean, the initializer method is not required to be a business method of the session bean.

And direct field injection:

```
public class Checkout {

  private @Inject ShoppingCart cart;

}
```

Note

Getter and setter methods are not required for field injection to work (unlike with JSF managed beans).

Dependency injection always occurs when the bean instance is first instantiated by the container.

- First, the container calls the bean constructor (the default constructor or the one annotated @Inject), to obtain an instance of the bean.
- Next, the container initializes the values of all injected fields of the bean.
- Next, the container calls all initializer methods of bean (the call order is not portable, don't rely on it).
- Finally, the `@PostConstruct` method of the bean is called, if defined.

Tip

One major advantage of using constructor injection is that the bean can be made immutable.

Bean constructors, initializer methods and injected fields must be annotated *@Inject*. The parameters of bean constructors and initializers are injection points, which means the container will search out beans matching the bean type and qualifier and pass them in as arguments.

If a parameter of bean constructors and initializer methods is not explicitly annotated with a qualifier, the default qualifier, `@Default`, is applied. The same is goes for an injected field.

CDI also supports parameter injection on methods that are invoked by the container. For instance, parameter injection is supported for producer methods:

```
@Produces Checkout createCheckout(ShoppingCart cart) {
   return new Checkout(cart);
}
```

This is one of the cases where the `@Inject` annotation *is not* required at an injection point. Other cases include observer methods (which we'll meet in *Chapter 9, Events*), disposal methods and destructor methods, which all support parameter injection.

4.2. What gets injected

The CDI specification defines a procedure, called the *typesafe resolution algorithm*, that the container follows when identifying the bean to inject to an injection point. This algorithm looks complex at first, but once you understand it, it's really quite intuitive. Typesafe resolution is performed at system initialization time, which means that the container will inform the developer immediately if a bean's dependencies cannot be satisfied, by throwing a `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

The purpose of this algorithm is to allow multiple beans to implement the same bean type and either:

- allow the client to select which implementation it requires using *qualifier* or
- allow the application deployer to select which implementation is appropriate for a particular deployment, without changes to the client, by enabling or disabling an *alternative*, or

Obviously, if you have exactly one bean of a given type, and an injection point with that same type, then bean A is going to go into slot A. That's the simplest possible scenario. When you first start your application, you'll likely have lots of those.

But then, things start to get complicated. Let's explore how the container determines which bean to inject in those not-so-obvious cases by taking a look at qualifiers and how they help your beans fit into the right picture.

4.3. Qualifier annotations

If we have more than one bean that implements a particular bean type, the injection point can specify exactly which bean should be injected using a qualifier annotation. For example, there might be two implementations of `PaymentProcessor`:

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Where `@Synchronous` and `@Asynchronous` are qualifier annotations:

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

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

A client bean developer uses the qualifier annotation to specify exactly which bean should be injected.

Using field injection:

```
@Inject @Synchronous PaymentProcessor syncPaymentProcessor;
@Inject @Asynchronous PaymentProcessor asyncPaymentProcessor;
```

Using initializer method injection:

```
@Inject
public void setPaymentProcessors(@Synchronous PaymentProcessor syncPaymentProcessor,
                                 @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   this.syncPaymentProcessor = syncPaymentProcessor;
   this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Using using constructor injection:

```
@Inject
public Checkout(@Synchronous PaymentProcessor syncPaymentProcessor,
                @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   this.syncPaymentProcessor = syncPaymentProcessor;
   this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Qualifier annotations can also qualify method arguments of producer, disposer and observer methods. Combining qualified arguments with producer methods is a good way to have an implementation of a bean type at runtime selected at runtime based on the state of the system:

```
@Produces
PaymentProcessor getPaymentProcessor(@Synchronous PaymentProcessor
 syncPaymentProcessor,
                                     @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   return isSynchronous() ? syncPaymentProcessor : asyncPaymentProcessor;
}
```

Now, you may be thinking, *"What's the different between using a qualifier and just specifying the exact implementation class you want?"* It's important to understand that a qualifier is like an extension of the interface. You could potentially implement the asynchronous payment processor differently over the age of the application. However, if you use qualifier annotations to hook injection points with the bean implementing that functionality, the server implementation can vary without affecting the client code. Plus, you get all the benefits of using the bean's interface because you don't have a hard dependency on the implementation.

Qualifiers will likely replace use of the factory pattern as well. The container becomes your factory and the switch between implementations is dictated by the qualifier.

4.3.1. Qualifiers with members

Annotations can have members (i.e., fields) just like regular classes. You can use these fields to furthur discriminate the qualifier. This simply prevents an explosion of annotations that you have to introduce. For instance, if you wanted to create several qualifiers representing different payment methods, you could aggregate them under a single annotation using a member:

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
   PaymentMethod value();
}
```

Then you select one of the possible values when you use the qualifier:

```
private @Inject @PayBy(CHECK) CheckPayment checkPayment;
```

You can tell the container to ignore a member of a qualifier type by annotating the member @NonBinding.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
   PaymentMethod value();
   @NonBinding String comment();
}
```

4.3.2. Combining qualifiers

Both a bean and an injection point may specify multiple qualifiers:

```
@Synchronous @Reliable
public class SynchronousReliablePaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@Inject @Synchronous @Reliable PaymentProcessor syncPaymentProcessor;
```

In this case, only a bean which has *both* qualifier annotations would be eligible for injection.

4.3.3. Qualifiers on producer methods

Even producer methods may specify qualifier annotations:

```
@Produces @Asynchronous PaymentProcessor getPaymentProcessor() { ... }
```

This producer might create an asynchronous implementation powered by third party library such as Quartz (though definitely check out the new scheduler support in EJB 3.1!).

4.3.4. The default qualifier

CDI defines the qualifier annotation `@Default` that is the default qualifier type for any injection point or bean that does not explicitly specify a qualifier.

The most common circumstance when it's necessary to explicitly specify `@Default` is on a bean which as another qualifier in addition to the default one.

4.4. Fixing unsatisfied dependencies

The typesafe resolution algorithm fails when, after considering the qualifier annotations on all beans that implement the bean type of an injection point and filtering out disabled beans (`@Alternative` beans which are not explicitly enabled), the container is unable to identify exactly one bean to inject. The container will either throw an `UnsatisfiedDependencyException` or `AmbiguousDependencyException`, depending on the circumstance.

During the course of your development, your going to encounter these errors. Let's learn how to resolve them. It's usually pretty easy, and a decent IDE can really help out here.

To fix an `UnsatisfiedDependencyException`, simply provide a bean which implements the bean type and has all the qualifier types of the injection point # or explicitly enable an `@Alternative` bean that implements the bean type and has the appropriate qualifier types.

To fix an `AmbiguousDependencyException`, introduce a qualifier to distinguish between the two implementations of the bean type, or disable one of two `@Alternative` beans that are trying to occupy the same space. An `AmbiguousDependencyException` can only occur if two enabled beans share the same qualifier types (and one bean does not specialize the other, which we'll get into specialization in *Chapter 11, Specialization, inheritance and alternatives*).

Tip

Just remember: "There can be only one."

There's one more issue you need to be aware of when using dependency injection in beans: client proxies.

4.5. Client proxies

Clients of an injected bean do not usually hold a direct reference to a bean instance, unless the bean is dependent-scoped—in that case, the instance belongs to the bean.

Imagine that a bean bound to the application scope held a direct reference to a bean bound to the request scope. The application-scoped bean is shared between many different requests. However, each request should see a different instance of the request scoped bean—the current one!

Now imagine that a bean bound to the session scope holds a direct reference to a bean bound to the application scope. From time to time, the session context is serialized to disk in order to use memory more efficiently. However, the application scoped bean instance should not be serialized along with the session scoped bean! It can get that reference any time. No need to hoard it!

Therefore, unless a bean has the default scope `@Dependent`, the container must indirect all injected references to the bean through a proxy object. This *client proxy* is responsible for ensuring that the bean instance that receives a method invocation is the instance that is associated with the current context. The client proxy also allows beans bound to contexts such as the session context to be serialized to disk without recursively serializing other injected beans.

Note

If you recall, Seam also boasted the ability to reference the current instance of a component. However, Seam went about it differently. In Seam, the references are established prior to the method call and cleared afterword using an interceptor—a process known as bijection. This model was not very friendly to multiple threads sharing instances and thefore the client proxy approach was adopted in CDI instead.

Unfortunately, due to limitations of the Java language, some Java types cannot be proxied by the container. If one of these types declares any scope other than `@Dependent`, the container throws an `UnproxyableDependencyException` when processing its use as an injection point.

The following Java types cannot be proxied by the container:

- classes which don't have a non-private constructor with no parameters, and
- classes which are declared `final` or have a `final` method,

- arrays and primitive types.

It's usually very easy to fix an `UnproxyableDependencyException`. Simply add a constructor with no parameters to the injected class, introduce an interface, or, if all else fails, change the scope of the injected bean to `@Dependent`.

4.6. Obtaining a contextual instance by programmatic lookup

In certain situations, injection is not the most convenient way to obtain a contextual reference. For example, it may not be used when:

- the bean type or qualifiers vary dynamically at runtime, or
- depending upon the deployment, there may be no bean which satisfies the type and qualifiers, or
- we would like to iterate over all beans of a certain type.

In these situations, the application may obtain an instance of the interface `Instance` parameterized for the bean type by injection:

```
@Inject Instance<PaymentProcessor> paymentProcessorSource;
```

The `get()` method on `Instance` can be used to produce a contextual instance of the bean programmatically.

```
PaymentProcessor p = paymentProcessorSource.get()
```

Of course, qualifiers can be specified at the injection point, as with a bean injection:

```
@Inject @Asynchronous Instance<PaymentProcessor> paymentProcessorSource;
```

You can also select a bean dynamically from the `Instance` based on the bean's qualifier. First, you need to add the `@Any` annotation at the injection point:

```
@Inject @Any Instance<PaymentProcessor> paymentProcessorSource;
```

Then you specify which bean you want, based on its qualifier, using the `select()` method on `Instance`. You represent a qualifier annotation in code by subclassing the helper class `AnnotationLiteral`, since it's otherwise difficult to instantiate an annotation type in Java.

```
PaymentProcessor p = paymentProcessorSource
  .select(new AnnotationLiteral<Asynchronous>() {});
```

You can choose to create a concreate type for all of your annotation literals, such as:

```
abstract class AsynchronousQualifier
extends AnnotationLiteral<Asynchronous> implements Asynchronous {}
```

Note

The concrete is required if your qualifier has members, since an anonymous subclass cannot define members.

Again, like with annotation definitions, the `AnnotationLiteral` definition may seem a bit foreign at first, but it will soon become second nature as you will see it often in CDI applications.

Here's how you might make use the dynamic selection:

```
Annotation qualifier = synchronously ?
new SynchronousQualifier() : new AsynchronousQualifier();
PaymentProcessor p = anyPaymentProcessor.select(qualifier).get().process(payment);
```

4.7. Bean names and EL lookup

You won't always be able to take advantage of the type safety that CDI provides. One of those places is the JSF view. To accomodate that environment, CDI must learn to speak the binding language that view templates understand. That language is the Unified EL.

In order to participate in the Unified EL, a bean must have a name. The old way of assigning a name to a managed bean was in the faces-config.xml descriptor. This use of XML completely contradicts the approach to metadata that's in CDI. Given that one of the primary goals of JSR-299 is to get JSF talking with session beans (and, in turn, the rest of the Java EE platform), CDI needed to find a compromise. It did in the form of the `@Named`.

The `@Named` annotation, when added to a bean class, producer method or producer field, gives an alternate way for that bean to be referenced—by a string-based name.

```
public @Named("cart") @SessionScoped class ShoppingCart {
 ...
```

```
}
```

```
public @Produces @Named("items") List<Item> getItems() { ... }
```

```
private @Produces @Named("currentUser") User user;
```

Of course, you don't want to use this string-based name in your Java code, but you can use it your JSF views to invoke or read state from a bean. Since a bean could be an session bean, now you can hook your JSF view directly to your business component without any middle man and still keep the layers loosely coupled. Loosely coupled? Yes, because names can be associated with the deployment time and runtime polymorphism that CDI provides.

Although JSF 2 still includes a managed bean facility, it's expected that you are now going to prefer CDI and EJB as your bean container, since they provides all that JSF managed beans have and much, much more. Plus, your business tier isn't tied to JSF in any way.

4.8. Lifecycle callbacks and resource injections

As part of the managed bean contract, all CDI beans support the lifecycle callback methods, `@PostConstruct` and `@PreDestroy` which are called after the bean is initialized (and after injections take place) and before it is destroyed (when the context to which it is bound ends), respectively.

Of course, session beans also support callback methods introduced by the EJB specification, such as `@PrePassivate` and `@PostActivate`.

Managed beans also support Java EE 5-style resource injections, declared using the `@Resource` annotation on a field. CDI beans can also inject session beans with the `@EJB` annotation.

However, as you learned in a previous section, these injections are not fully type-safe, nor do they have the semantics that CDI provides. Therefore, it's the use of `@Inject` for injection is preferred. Relegate the resource injections to a bean that uses producer fields to mold these container-managed objects into injectable beans.

Scopes and contexts

So far, we've seen a few examples of *scope type annotations*. The scope of a bean determines the lifecycle of instances of the bean. The scope also determines which clients refer to which instances of the bean. According to the CDI specification, a scope determines:

- When a new instance of any bean with that scope is created
- When an existing instance of any bean with that scope is destroyed
- Which injected references refer to any instance of a bean with that scope

For example, if we have a session-scoped bean, `CurrentUser`, all beans that are called in the context of the same `HttpSession` will see the same instance of `CurrentUser`. This instance will be automatically created the first time a `CurrentUser` is needed in that session, and automatically destroyed when the session ends.

Note

There's actually no way to remove a bean from a context explicitly. It turns out that's a good thing because there is no confusion as to which instance you are getting.

5.1. Scope types

CDI features an *extensible context model*. It's possible to define new scopes by creating a new scope type annotation:

```
@ScopeType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface ClusterScoped {}
```

Of course, that's the easy part of the job. For this scope type to be useful, we will also need to define a `Context` object that implements the scope! Implementing a `Context` is usually a very technical task, intended for framework development only. You can expect an implemention of the business scope, for instance, in a future version of Seam.

We can apply a scope type annotation to a bean implementation class to specify the scope of the bean:

```
@ClusterScoped
```

```
public class SecondLevelCache { ... }
```

Usually, you'll use one of CDI's built-in scopes.

5.2. Built-in scopes

CDI defines four built-in scopes:

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`

For a web application that uses CDI:

- any servlet request has access to active request, session and application scopes, and, additionally
- any JSF request has access to an active conversation scope.

Note

A CDI extension can support the conversation for other frameworks as well.

The request and application scopes are also active:

- during invocations of EJB remote methods,
- during EJB timeouts,
- during message delivery to a message-driven bean, and
- during web service invocations.

If the application tries to invoke a bean with a scope that does not have an active context, a `ContextNotActiveException` is thrown by the container at runtime.

Three of the four built-in scopes should be extremely familiar to every Java EE developer, so let's not waste time discussing them here. One of the scopes, however, is new.

5.3. The conversation scope

The conversation scope is a bit like the traditional session scope in that it holds state associated with a user of the system, and spans multiple requests to the server. However, unlike the session scope, the conversation scope:

- is demarcated explicitly by the application, and
- holds state associated with a particular web browser tab in a JSF application (browsers tend to share domain cookies, and hence the session cookie, between tabs, which is the root of the issue).

A conversation represents a task—a unit of work from the point of view of the user. The conversation context holds state associated with what the user is currently working on. If the user is doing multiple things at the same time, there are multiple conversations.

The conversation context is active during any JSF request. Most conversations are destroyed at the end of the request. If a conversation should hold state across multiple requests, it must be explicitly promoted to a *long-running conversation*.

5.3.1. Conversation demarcation

CDI provides a built-in bean for controlling the lifecyle of conversations in a JSF application. This bean may be obtained by injection:

```
@Inject Conversation conversation;
```

To promote the conversation associated with the current request to a long-running conversation, call the `begin()` method from application code. To schedule the current long-running conversation context for destruction at the end of the current request, call `end()`.

In the following example, a conversation-scoped bean controls the conversation with which it is associated:

```
@ConversationScoped @Stateful
public class OrderBuilder {
  private Order order;
  private @Inject Conversation conversation;
  private @PersistenceContext(type = EXTENDED) EntityManager em;

  @Produces public Order getOrder() {
    return order;
  }
```

```
  public Order createOrder() {
    order = new Order();
    conversation.begin();
    return order;
  }

  public void addLineItem(Product product, int quantity) {
    order.add(new LineItem(product, quantity));
  }

  public void saveOrder(Order order) {
    em.persist(order);
    conversation.end();
  }

  @Remove
  public void destroy() {}
}
```

This bean is able to control its own lifecycle through use of the `Conversation` API. But some other beans have a lifecycle which depends completely upon another object.

5.3.2. Conversation propagation

The conversation context automatically propagates with any JSF faces request (JSF form submission). It does not automatically propagate with non-faces requests, for example, navigation via a link.

We can force the conversation to propagate with a non-faces request by including the unique identifier of the conversation as a request parameter. The CDI specification reserves the request parameter named `cid` for this use. The unique identifier of the conversation may be obtained from the `Conversation` object, which has the EL bean name `conversation`.

Therefore, the following link propagates the conversation:

```
<a href="/addProduct.jsp?cid=#{conversation.id}">Add Product</a>
```

Though it's probably better to use one of the link components in JSF 2:

```
<h:link outcome="/addProduct.xhtml value="Add Product">
  <f:param name="cid" value="#{conversation.id}"/>
</h:link>
```

The container is also required to propagate conversations across any redirect, even if the conversation is not marked long-running. This makes it very easy to implement the common POST-then-redirect pattern, without resort to fragile constructs such as a "flash" object. In this case, the container automatically adds a request parameter to the redirect URL.

5.3.3. Conversation timeout

The container is permitted to destroy a conversation and all state held in its context at any time in order to preserve resources. A CDI implementation will normally do this on the basis of some kind of timeout # though this is not required by the CDI specification. The timeout is the period of inactivity before the conversation is destroyed (as opposed to the amount of time the conversation is active).

The `Conversation` object provides a method to set the timeout. This is a hint to the container, which is free to ignore the setting.

```
conversation.setTimeout(timeoutInMillis);
```

5.4. The dependent pseudo-scope

In addition to the four built-in scopes, CDI features the so-called *dependent pseudo-scope*. This is the default scope for a bean which does not explicitly declare a scope type.

For example, this bean has the scope type `@Dependent`:

```
public class Calculator { ... }
```

When an injection point of a bean resolves to a dependent bean, a new instance of the dependent bean is created when the bean into which it's being injected is instantiated. Instances of dependent beans are never shared between different beans or different injection points. They are strictly *dependent objects* of some other bean instance.

Dependent bean instances are destroyed when the instance they depend upon is destroyed.

CDI makes it easy to obtain a dependent instance of a bean, even if the bean is already declared as a bean with some other scope type.

5.4.1. The `@New` annotation

The built-in `@New` qualifier annotation allows *implicit* definition of a dependent bean at an injection point. Suppose we declare the following injected field:

```
@Inject @New Calculator calculator;
```

Then a bean with scope `@Dependent`, qualifier type `@New`, API type `Calculator`, implementation class `Calculator` and deployment type `@Standard` is implicitly defined.

This is true even if `Calculator` is *already* declared with a different scope type, for example:

```
@ConversationScoped
public class Calculator { ... }
```

So the following injected attributes each get a different instance of `Calculator`:

```
public class PaymentCalc {
   @Inject Calculator calculator;
   @Inject @New Calculator newCalculator;
}
```

The `calculator` field has a conversation-scoped instance of `Calculator` injected. The `newCalculator` field has a new instance of `Calculator` injected, with a lifecycle that is bound to the owning `PaymentCalc`.

This feature is particularly useful with producer methods, as we'll see in the next chapter.

Producer methods

Producer methods let us overcome certain limitations that arise when a container, instead of the application, is responsible for instantiating objects. They're also the easiest way to integrate objects which are not beans into the CDI environment.

According to the spec:

> A producer method acts as a source of objects to be injected, where:
>
> - the objects to be injected are not required to be instances of beans,
> - the concrete type of the objects to be injected may vary at runtime or
> - the objects require some custom initialization that is not performed by the bean constructor

For example, producer methods let us:

- expose a JPA entity as a bean,
- expose any JDK class as a bean,
- define multiple beans, with different scopes or initialization, for the same implementation class, or
- vary the implementation of a bean type at runtime.

In particular, producer methods let us use runtime polymorphism with CDI. As we've seen, alternative beans are one solution to the problem of deployment-time polymorphism. But once the system is deployed, the CDI implementation is fixed. A producer method has no such limitation:

```
@SessionScoped
public class Preferences {
  private PaymentStrategyType paymentStrategy;
  ...
  @Produces @Preferred
  public PaymentStrategy getPaymentStrategy() {
     switch (paymentStrategy) {
        case CREDIT_CARD: return new CreditCardPaymentStrategy();
        case CHECK: return new CheckPaymentStrategy();
        case PAYPAL: return new PayPalPaymentStrategy();
        default: return null;
     }
  }
```

```
}
```

Consider an injection point:

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

This injection point has the same type and qualifier annotations as the producer method, so it resolves to the producer method using the usual CDI injection rules. The producer method will be called by the container to obtain an instance to service this injection point.

6.1. Scope of a producer method

The scope of the producer method defaults to `@Dependent`, and so it will be called *every time* the container injects this field or any other field that resolves to the same producer method. Thus, there could be multiple instances of the `PaymentStrategy` object for each user session.

To change this behavior, we can add a `@SessionScoped` annotation to the method.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
   ...
}
```

Now, when the producer method is called, the returned `PaymentStrategy` will be bound to the session context. The producer method won't be called again in the same session.

6.2. Injection into producer methods

There's one potential problem with the code above. The implementations of `CreditCardPaymentStrategy` are instantiated using the Java `new` operator. Objects instantiated directly by the application can't take advantage of dependency injection and don't have interceptors.

If this isn't what we want we can use dependency injection into the producer method to obtain bean instances:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          CheckPaymentStrategy cps,
                                          PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
```

```
        case CHEQUE: return cps;
        case PAYPAL: return ppps;
        default: return null;
    }
}
```

Wait, what if `CreditCardPaymentStrategy` is a request-scoped bean? Then the producer method has the effect of "promoting" the current request scoped instance into session scope. This is almost certainly a bug! The request scoped object will be destroyed by the container before the session ends, but the reference to the object will be left "hanging" in the session scope. This error will *not* be detected by the container, so please take extra care when returning bean instances from producer methods!

There's at least three ways we could go about fixing this bug. We could change the scope of the `CreditCardPaymentStrategy` implementation, but this would affect other clients of that bean. A better option would be to change the scope of the producer method to `@Dependent` or `@RequestScoped`.

But a more common solution is to use the special `@New` qualifier annotation.

6.3. Use of `@New` with producer methods

Consider the following producer method:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New CheckPaymentStrategy cps,
                                          @New PayPalPaymentStrategy ppps) {
    switch (paymentStrategy) {
        case CREDIT_CARD: return ccps;
        case CHEQUE: return cps;
        case PAYPAL: return ppps;
        default: return null;
    }
}
```

Then a new *dependent* instance of `CreditCardPaymentStrategy` will be created, passed to the producer method, returned by the producer method and finally bound to the session context. The dependent object won't be destroyed until the `Preferences` object is destroyed, at the end of the session.

Part II. Developing loosely-coupled code

The first major theme of CDI is *loose coupling*. We've already seen three means of achieving loose coupling:

- *alternatives* enable deployment time polymorphism,
- *producer methods* enable runtime polymorphism, and
- *contextual lifecycle management* decouples bean lifecycles.

These techniques serve to enable loose coupling of client and server. The client is no longer tightly bound to an implementation of an API, nor is it required to manage the lifecycle of the server object. This approach lets *stateful objects interact as if they were services*.

Loose coupling makes a system more *dynamic*. The system can respond to change in a well-defined manner. In the past, frameworks that attempted to provide the facilities listed above invariably did it by sacrificing type safety (most notably by using XML descriptors). CDI is the first technology, and certainly the first specification in the Java EE platform, that achieves this level of loose coupling in a typesafe way.

CDI provides three extra important facilities that further the goal of loose coupling:

- *interceptors* decouple technical concerns from business logic,
- *decorators* may be used to decouple some business concerns, and
- *event notifications* decouple event producers from event consumers.

Let's explore interceptors first.

Interceptors

Thanks to the managed bean specification, intereceptors are now part of the core functionality of a bean. That means that you no longer need to make your class an EJB just to use interceptors!

But that doesn't mean there aren't other improvements that can be made. CDI builds on the interceptor architecture of managed beans by introducing a more sophisticated, semantic, annotation-based approach to binding interceptors to beans.

Interceptor functionality is defined in the interceptor specification, which the managed bean, CDI, and EJB specifications all support. The interceptor specification defines two kinds of interception points:

- business method interception, and
- lifecycle callback interception.

A *business method interceptor* applies to invocations of methods of the bean by clients of the bean:

```
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

A *lifecycle callback interceptor* applies to invocations of lifecycle callbacks by the container:

```
public class DependencyInjectionInterceptor {
   @PostConstruct public void injectDependencies(InvocationContext ctx) { ... }
}
```

An interceptor class may intercept both lifecycle callbacks and business methods.

7.1. Interceptor bindings

Suppose we want to declare that some of our beans are transactional. The first thing we need is an *interceptor binding annotation* to specify exactly which beans we're interested in:

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {}
```

Now we can easily specify that our `ShoppingCart` is a transactional object:

```
@Transactional
public class ShoppingCart { ... }
```

Or, if we prefer, we can specify that just one method is transactional:

```
public class ShoppingCart {
  @Transactional public void checkout() { ... }
}
```

7.2. Implementing interceptors

That's great, but somewhere along the line we're going to have to actually implement the interceptor that provides this transaction management aspect. All we need to do is create a standard interceptor, and annotate it `@Interceptor` and `@Transactional`.

```
@Transactional @Interceptor
public class TransactionInterceptor {
  @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

All bean interceptors are beans, and can take advantage of dependency injection and contextual lifecycle management.

```
@ApplicationScoped @Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

Multiple interceptors may use the same interceptor binding type.

7.3. Enabling interceptors

By default, all interceptors are disabled. We need to *enable* our interceptor in the `beans.xml` descriptor of a bean deployment archive. This activation only applies to beans in the same archive.

```
<beans
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <interceptors>
    <class>org.mycompany.myapp.TransactionInterceptor</class>
  </interceptors>
</beans>
```

Whoah! Why the angle bracket stew?

Well, have the XML declaration is actually a *good thing*. It solves two problems:

- it enables us to specify a total ordering for all the interceptors in our system, ensuring deterministic behavior, and
- it lets us enable or disable interceptor classes at deployment time.

For example, we could specify that our security interceptor runs before our transaction interceptor.

```
<beans
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <interceptors>
    <class>org.mycompany.myapp.SecurityInterceptor</class>
    <class>org.mycompany.myapp.TransactionInterceptor</class>
  </interceptors>
</beans>
```

Or we could turn them both off in our test environment by simply taking no action! Ah, so simple.

7.4. Interceptor bindings with members

Suppose we want to add some extra information to our `@Transactional` annotation:

```
@InterceptorBinding
```

```
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
  boolean requiresNew() default false;
}
```

CDI will use the value of `requiresNew` to choose between two different interceptors, `TransactionInterceptor` and `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew = true) @Interceptor
public class RequiresNewTransactionInterceptor {
  @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Now we can use `RequiresNewTransactionInterceptor` like this:

```
@Transactional(requiresNew = true)
public class ShoppingCart { ... }
```

But what if we only have one interceptor and we want the container to ignore the value of `requiresNew` when binding interceptors? Perhaps this information is only useful for the interceptor implementation. We can use the `@NonBinding` annotation:

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
  @NonBinding String[] rolesAllowed() default {};
}
```

7.5. Multiple interceptor binding annotations

Usually we use combinations of interceptor bindings types to bind multiple interceptors to a bean. For example, the following declaration would be used to bind `TransactionInterceptor` and `SecurityInterceptor` to the same bean:

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

However, in very complex cases, an interceptor itself may specify some combination of interceptor binding types:

```
@Transactional @Secure @Interceptor
public class TransactionalSecureInterceptor { ... }
```

Then this interceptor could be bound to the `checkout()` method using any one of the following combinations:

```
public class ShoppingCart {
   @Transactional @Secure public void checkout() { ... }
}
```

```
@Secure
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

```
@Transactionl
public class ShoppingCart {
   @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
   public void checkout() { ... }
}
```

7.6. Interceptor binding type inheritance

One limitation of the Java language support for annotations is the lack of annotation inheritance. Really, annotations should have reuse built in, to allow this kind of thing to work:

```
public @interface Action extends Transactional, Secure { ... }
```

Well, fortunately, CDI works around this missing feature of Java. We may annotate one interceptor binding type with other interceptor binding types (termed a *meta-annotation*). The interceptor bindings are transitive # any bean with the first interceptor binding inherits the interceptor bindings declared as meta-annotations.

```
@Transactional @Secure
@InterceptorBinding
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action { ... }
```

Now, any bean annotated `@Action` will be bound to both `TransactionInterceptor` and `SecurityInterceptor`. (And even `TransactionalSecureInterceptor`, if it exists.)

7.7. Use of `@Interceptors`

The `@Interceptors` annotation defined by the interceptor specification (and used by the managed bean and EJB specifications) is still supported in CDI.

```
@Interceptors({TransactionInterceptor.class, SecurityInterceptor.class})
public class ShoppingCart {
  public void checkout() { ... }
}
```

However, this approach suffers the following drawbacks:

- the interceptor implementation is hardcoded in business code,
- interceptors may not be easily disabled at deployment time, and
- the interceptor ordering is non-global # it is determined by the order in which interceptors are listed at the class level.

Therefore, we recommend the use of CDI-style interceptor bindings.

Decorators

Interceptors are a powerful way to capture and separate concerns which are *orthogonal* to the application (and type system). Any interceptor is able to intercept invocations of any Java type. This makes them perfect for solving technical concerns such as transaction management, security and call logging. However, by nature, interceptors are unaware of the actual semantics of the events they intercept. Thus, interceptors aren't an appropriate tool for separating business-related concerns.

The reverse is true of *decorators*. A decorator intercepts invocations only for a certain Java interface, and is therefore aware of all the semantics attached to that interface. Since decorators directly implement operations with business semantics, it makes them the perfect tool for modeling some kinds of business concerns. It also means that a decorator doesn't have the generality of an interceptor. Decorators aren't able to solve technical concerns that cut across many disparate types. The two complement one another. Let's look at some cases where decorators fit the bill.

Suppose we have an interface that represents accounts:

```
public interface Account {
   public BigDecimal getBalance();
   public User getOwner();
   public void withdraw(BigDecimal amount);
   public void deposit(BigDecimal amount);
}
```

Several different beans in our system implement the `Account` interface. However, we have a common legal requirement that; for any kind of account, large transactions must be recorded by the system in a special log. This is a perfect job for a decorator.

A decorator is a bean (possibly even an abstract class) that implements the type it decorates and is annotated `@Decorator`.

```
@Decorator
public abstract class LargeTransactionDecorator
   implements Account {
   ...
}
```

So what goes in the decorator? Well, decorators have to inject the instance of the bean they are decorating, termed the *delegate injection point*. The injection point has the same type as the bean to decorate and the annotation `@Delegate`. There can be at most one injection point, which can be a constructor, initializer method or field injection.

```
@Decorator
public abstract class LargeTransactionDecorator
  implements Account {
  @Inject @Delegate @Any Account account;
  ...
}
```

You'll see in a moment that the beans which are decorated can be further restricted by specifying qualifiers at the injection point, a familiar pattern in CDI.

The decorator then implements any methods of the bean type it wants to decorate. The decorator can in turn invoke the method on the decorated instance. Note that the method on the decorate is being called by the container *instead of* the method on the bean instance. It's up to the decorator to invoke the method on the bean instance. In fact, the decorator can invoke any method on the bean instance.

```
@Decorator
public abstract class LargeTransactionDecorator
  implements Account {
  @Inject @Delegate @Any Account account;

  @PersistenceContext EntityManager em;

  public void withdraw(BigDecimal amount) {
     account.withdraw(amount);
     if ( amount.compareTo(LARGE_AMOUNT)>0 ) {
        em.persist( new LoggedWithdrawl(amount) );
     }
   }

  public void deposit(BigDecimal amount);
    account.deposit(amount);
    if ( amount.compareTo(LARGE_AMOUNT)>0 ) {
       em.persist( new LoggedDeposit(amount) );
    }
  }
}
```

Interceptors for a method are called before decorators that apply to that method.

Unlike other beans, a decorator may be an abstract class. Therefore, if there's nothing special the decorator needs to do for a particular method of the decorated interface, you don't need to implement that method.

8.1. Delegate object

All decorators must have a delegate injection point that injects the delegate object, as shown above. The type and qualifier types of the delegate injection point determine which beans the decorator is bound to. The delegate object type must implement or extend all interfaces implemented by the decorator.

This delegate injection point specifies that the decorator is bound to all beans that implement `Account`:

```
@Inject @Delegate @Any Account account;
```

A delegate injection point may specify any number of qualifier annotations. Then the decorator will only be bound to beans with the same qualifiers.

```
@Inject @Delegate @Foreign Account account;
```

A decorator is bound to any bean which:

- has the type of the delegate object as a bean type, and
- has all qualifiers that are declared at the delegate injection point.

The decorator may invoke the delegate object, which has much the same effect as calling `InvocationContext.proceed()` from an interceptor. The main difference is that the decorator can invoke *any* business method on the delegate object.

8.2. Enabling decorators

By default, all decorators are disabled. We need to *enable* our decorator in the `beans.xml` descriptor of a bean deployment archive. This activation only applies to beans in the same archive.

```
<beans
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
```

```
  <decorators>
      <class>org.mycompany.myapp.LargeTransactionDecorator</class>
  </decorators>
</beans>
```

This declaration serves the same purpose for decorators that the `<Interceptors>` declaration serves for interceptors:

- it enables us to specify a total ordering for all decorators in our system, ensuring deterministic behavior, and
- it lets us enable or disable decorator classes at deployment time.

Events

Dependency injection enables loose-coupling by allowing the implementation of the injected bean type to vary, either a deployment time or runtime. Events provide a whole other level of decoupling in which there is no compile time dependency between the interacting beans at all. Event *producers* raise (or fire) events that are delivered to event *observers*, an exchange orchestrated by the container.

CDI provides an event notification facility like the one just described. It does so using the same type safe approach that you've become accustomed to with CDI's dependency injection service. In fact, it consists of all the same ingredients:

- bean types,
- qualifier annotations and
- type-safe resolution.

It also supports a couple of convenient features that extend beyond the basic observer/observable pattern:

- events are qualified using annotations on the event producer object,
- observers are completely decoupled from producers,
- observers can specify a combination of "selectors" to narrow the set of event notifications (described by qualifiers) they will receive, and
- observers can be notified immediately, or can specify that delivery of the event should be delayed until the end of the current transaction (a very useful feature for enterprise web applications).

Before getting into how events are produced, let's first consider what's used as the event payload and how those events are observed. We'll then hook everything together by looking at how an event is fired.

9.1. Event payload

The event payload carries state from producer to consumer. The event is nothing more than a plain Java object--an instance of any Java type. (The only restriction is that an event type may not contain type variables). No special interfaces or wrappers. In addition, that object can be assigned qualifiers, which helps observers distinguish it from other events of the same type. In a way, the qualifiers are like topic selectors, since they allow the observers to narrow the set of events it observes.

So what's an event qualifier? An event qualifier is just a normal qualifier, the same ones you assigned to beans and injection points. Here's an example.

```
@Qualifier
@Target({FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Updated {}
```

These qualifiers may or may not come into play when selecting an observer. As before, a qualifier type can have members. Those members are considered as well when selecting an observer. However, only if the member is not annotated `@NonBinding`, which causes it to be ignored by the selection process.

9.2. Event observers

An *observer method* is a method of a bean with a parameter annotated `@Observes`.

```
public void onAnyDocumentEvent(@Observes Document document) { ... }
```

The annotated parameter is called the *event parameter*. The type of the event parameter is the observed *event type*, in this case `Document`, a class in the application. Observer methods may also specify "selectors", which are just qualifiers, as just described. When a qualifier is used as an event selector, it's called an *event qualifier type*.

We specify the event qualifiers of the observer method by annotating the event parameter:

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

An observer method need not specify any event qualifiers # in this case it is interested in *all* events of a particular type. If it does specify event bindings, it's only interested in events which also have those qualifiers.

The observer method may have *additional* parameters, which are injected according to the usual bean method parameter injection semantics:

```
public void afterDocumentUpdate(@Observes @Updated Document document, User user) { ... }
```

9.3. Event producers (Firing events)

Producers (i.e., beans) fire events using an instance of the parameterized `Event` interface. Instances of this interface are obtained through injection:

```
@Inject @Any Event<Document> documentEvent;
```

A producer raises events by calling the `fire()` method of the `Event` interface, passing an *event object*:

```
documentEvent.fire(document);
```

This particular event will be delivered to every observer method that:

- has an event parameter to which the event object is assignable (i.e., `Document`), and
- specifies no qualifier bindings.

The container simply calls all the observer methods, passing the event object as the value of the event parameter. If any observer method throws an exception, the container stops calling observer methods, and the exception is rethrown by the `fire()` method.

Qualifiers can be applied to an event type in one of two ways:

- Add qualifier annotations to the field where `Event` is injected or
- pass qualifier annotation literals to the `select()` of `Event`.

The first option is the simplest:

```
@Inject @Updated Event<Document> documentUpdatedEvent;
```

Then, every event fired via this instance of `Event` has the event qualifier `@Updated`. The event will be delivered to every observer method that:

- has an event parameter to which the event object is assignable, and
- does not have any event qualifier *except* for the event qualifiers that match those on the producer (in this case at the `Event` inject point).

The downside of annotating the injection point is that you can not specify the qualifiers dynamically. Fortunately, CDI provides an alternative way of generating derived `Event` instances from an `Event` instance using the `select()` method and passing in qualifier annotation literals. Events can then be fired from that narrower source:

```
documentEvent.select(new AnnotationLiteral<Updated>(){}).fire(document);
```

There is no difference between the firing of this event and the previous one, were the qualifiers were specified at the injection point. Both events carry the `@Updated` qualifier.

As mentioned earlier, the helper class `AnnotationLiteral` makes it possible to instantiate qualifiers inline, since this is otherwise difficult to do in Java.

Event types can have multiple event qualifier types as well, assembled using any combination of annotations at the injection point and annotations passed to the `select()` method.

9.4. Conditional observer methods

Normally, observer beans will be "woken up", so to speak, when an event is fired so that they can respond to the event. This behavior isn't always what you want. You may be interested in delivering an event only to instances of the observers that exist in one of the currently active contexts.

A conditional observer is specified by adding `receive = IF_EXISTS` to the `@Observes` annotation. The default is to always receive, which means creating an instance of the bean.

```
public void refreshOnDocumentUpdate(@Observes(receive = IF_EXISTS) @Updated Document d) { ... }
```

A dependent-scoped bean cannot be a conditional observer, essentially because it's not a contextual bean.

9.5. Event qualifiers with members

An event qualifier type may have annotation members:

```
@Qualifier
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Role {
   RoleType value();
}
```

The member value is used to narrow the messages delivered to the observer:

```
public void adminLoggedIn(@Observes @Role(ADMIN) LoggedIn event) { ... }
```

Event qualifier type members may be specified statically by the event producer, via annotations at the event notifier injection point:

```
@Inject @Role(ADMIN) Event<LoggedIn> loggedInEvent;
```

Alternatively, the value of the event qualifier type member may be determined dynamically by the event producer. We start by writing an abstract subclass of `AnnotationLiteral`:

```
abstract class RoleBinding
  extends AnnotationLiteral<Role>
  implements Role {}
```

The event producer passes an instance of this class to `select()`:

```
documentEvent.select(new RoleBinding() {
  public void value() { return user.getRole(); }
}).fire(document);
```

9.6. Multiple event bindings

Event qualifier types may be combined, for example:

```
@Inject @Blog Event<Document> blogEvent;
...
if (document.isBlog()) blogEvent.select(new AnnotationLiteral<Updated>(){}).fire(document);
```

When this event occurs, all of the following observer methods will be notified:

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

```
public void onAnyBlogEvent(@Observes @Blog Document document) { ... }
```

```
public void onAnyDocumentEvent(@Observes Document document) { ... }}}
```

9.7. Transactional observers

Transactional observers receive their event notifications during the before or after completion phase of the transaction in which the event was raised. For example, the following observer method needs to refresh a query result set that is cached in the application context, but only when transactions that update the `Category` tree succeed:

```
public void refreshCategoryTree(@Observes(during = AFTER_SUCCESS) CategoryUpdateEvent event) { ... }
```

There are five kinds of transactional observers:

- `IN_PROGESS` observers are called immediately (default)

  `AFTER_SUCCESS` observers are called during the after completion phase of the transaction, but only if the transaction completes successfully
- `AFTER_FAILURE` observers are called during the after completion phase of the transaction, but only if the transaction fails to complete successfully
- `AFTER_COMPLETION` observers are called during the after completion phase of the transaction
- `BEFORE_COMPLETION` observers are called during the before completion phase of the transaction

Transactional observers are very important in a stateful object model because state is often held for longer than a single atomic transaction.

Imagine that we have cached a JPA query result set in the application scope:

```
@ApplicationScoped @Singleton
public class Catalog {

   @PersistenceContext EntityManager em;

   List<Product> products;

   @Produces @Catalog
   List<Product> getCatalog() {
      if (products==null) {
         products = em.createQuery("select p from Product p where p.deleted = false")
            .getResultList();
      }
      return products;
   }
```

```
}
```

From time to time, a `Product` is created or deleted. When this occurs, we need to refresh the `Product` catalog. But we should wait until *after* the transaction completes successfully before performing this refresh!

The bean that creates and deletes `Product`s could raise events, for example:

```
@Stateless
public class ProductManager {
   @PersistenceContext EntityManager em;
   @Inject @Any Event<Product> productEvent;

   public void delete(Product product) {
      em.delete(product);
      productEvent.select(new AnnotationLiteral<Deleted>(){}).fire(product);
   }

   public void persist(Product product) {
      em.persist(product);
      productEvent.select(new AnnotationLiteral<Created>(){}).fire(product);
   }
   ...
}
```

And now `Catalog` can observe the events after successful completion of the transaction:

```
@ApplicationScoped @Singleton
public class Catalog {
   ...
   void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product product) {
      products.add(product);
   }

   void addProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product product) {
      products.remove(product);
   }
}
```

Part III. Making the most of strong typing

The second major theme of Weld is *strong typing*. The information about the dependencies, interceptors and decorators of a bean, and the information about event consumers for an event producer, is contained in typesafe Java constructs that may be validated by the compiler.

You don't see string-based identifiers in CDI code, not because the framework is hiding them from you using clever defaulting rules # so-called "configuration by convention" # but because there are simply no strings there to begin with!

The obvious benefit of this approach is that *any* IDE can provide autocompletion, validation and refactoring without the need for special tooling. But there is a second, less-immediately-obvious, benefit. It turns out that when you start thinking of identifying objects, events or interceptors via annotations instead of names, you have an opportunity to lift the semantic level of your code.

CDI encourages you develop annotations that model concepts, for example,

- `@Asynchronous`,
- `@Mock`,
- `@Secure` or
- `@Updated`,

instead of using compound names like

- `asyncPaymentProcessor`,
- `mockPaymentProcessor`,
- `SecurityInterceptor` or
- `DocumentUpdatedEvent`.

The annotations are reusable. They help describe common qualities of disparate parts of the system. They help us categorize and understand our code. They help us deal with common concerns in a common way. They make our code more literate and more understandable.

CDI *stereotypes* take this idea a step further. A stereotype models a common *role* in your application architecture. It encapsulates various properties of the role, including scope, interceptor bindings, qualifiers, etc, into a single reusable package. (Of course, there is also the benefit of tucking some of those annotations away).

We're now ready to meet some more advanced features of CDI. Bear in mind that these features exist to make our code both easier to validate and more understandable. Most of the time you

don't ever really *need* to use these features, but if you use them wisely, you'll come to appreciate their power.

Stereotypes

The CDI specification defines a stereotype as follows:

> In many systems, use of architectural patterns produces a set of recurring bean roles. A stereotype allows a framework de- veloper to identify such a role and declare some common metadata for beans with that role in a central place.
>
> A stereotype encapsulates any combination of:
>
> - a default scope, and
> - a set of interceptor bindings.
>
> A stereotype may also specify that:
>
> - all beans with the stereotype have defaulted bean EL names, or that
> - all beans with the stereotype are alternatives.
>
> A bean may declare zero, one or multiple stereotypes. Stereotype annotations may be applied to a bean class or producer method or field.

In layman's terms, a stereotype is a meta-annotation (an annotation used on another annotation) annotated with `@Stereotype` that bundles other Java annotations to give them a particular semantic. For instance, the following stereotype identifies action classes in some MVC framework:

```
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

We use the stereotype by applying the annotation to a bean.

```
@Action
public class LoginAction { ... }
```

Of course, we need to associate our stereotype annotation with some other annotations or else it isn't doing much for us. Let's see how to add them.

10.1. Default scope for a stereotype

A stereotype may specify a default scope for beans annotated with the stereotype. For example, if the we might specify the following defaults for action classes in a web application:

```
@RequestScoped
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

Of course, a particular action may still override this default if necessary:

```
@Dependent @Action
public class DependentScopedLoginAction { ... }
```

Naturally, overriding a single default isn't much use. But remember, stereotypes can define more than just the default scope.

10.2. Interceptor bindings for stereotypes

A stereotype may specify a set of interceptor bindings to be inherited by all beans with that stereotype.

```
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

This helps us get technical concerns, like transactions and security, even further away from the business code!

10.3. Name defaulting with stereotypes

We can specify that all beans with a certain stereotype have a defaulted EL name when a name is not explicitly defined on that bean. Actions are often referenced in JSF views, so they're a perfect use case for this feature. All we need to do is add an empty `@Named` annotation:

```
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Named
```

```
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

Now, `LoginAction` bean from above will have the name `loginAction`.

10.4. Alternatives as stereotypes

A stereotype can inicate that all beans to which it is applied are `@Alternative` beans. If you can remember back to the early days of the specification, this is the closest match to a deployment type. Whole sets of beans can be enabled or left disabled by activating a single stereotype.

```
@Alternative
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Mock {}
```

Now you apply this stereotype to beans that should be active in mock environments.

```
@Mock
public class MockLoginAction extends LoginAction { ... }
```

10.5. Stereotype stacking

This may blow your mind a bit, but stereotypes may declare other stereotypes, which we'll call *stereotype stacking*. You may want to do this if you have two distinct stereotypes which are meaningful on their own, but in other situation may be meaningful when combined.

Here's an example that combines the `@Action` and `@Auditable` stereotypes:

```
@Auditable
@Action
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface AuditableAction {}
```

10.6. Built-in stereotypes

We've already met two standard stereotypes defined by the CDI specification: `@Interceptor` and `@Decorator`.

CDI defines one further standard stereotype, `@Model`, which is expected to be used frequently in web applications:

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

Instead of using JSF managed beans, just annotate a Web Bean `@Model`, and use it directly in your JSF view!

Specialization, inheritance and alternatives

When you first start developing with CDI, you'll likely be dealing only with a single bean implementation for each bean type. In this case, it's easy to understand how beans get selected for injection. As the complexity of your application grows, multiple occurances of the same bean type start appearing, either because you have multiple implementations or two beans share a common (Java) inheritance. That's when you have to begin studying the specialization, inheritance and alternative rules to work through unsatisified or ambiguous dependencies or to avoid certain beans from being called.

The CDI specification recognizes two distinct scenarios in which one bean extends another:

- The second bean specializes the first bean in certain deployment scenarios. In these deployments, the second bean completely replaces the first, fulfilling the same role in the system.

  The second bean is simply reusing the Java implementation, and otherwise bears no relation to the first bean. The first bean may not even have been designed for use as a contextual object.

The second case is the default assumed by CDI. It's possible to have two beans in the system with the same part bean type (interface or parent class). As you've learned, you select between the two implementations using qualifiers.

The first case is the exception, and also requires more care. In any given deployment, only one bean can fulfill a given role at a time. That means one bean needs to be enabled and the other disabled. There are a two modifiers involved: `@Alternative` and `@Specializes`. We'll start by looking at bean alternatives and then show the guarantees that specialization adds.

11.1. Bean alternatives

CDI let's you *override* the implementation of a bean type at deployment time using a bean alternative. For example, the following bean provides an implementation of the `PaymentProcessor` bean type in the default environment:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor {
  ...
}
```

But in our staging environment, we override that implementation of `PaymentProcessor` with a different bean:

```
@CreditCard @Alternative
public class StagingCreditCardPaymentProcessor
    implements PaymentProcessor {
  ...
}
```

or

```
@CreditCard @Alternative
public class StagingCreditCardPaymentProcessor
    extends CreditCardPaymentProcessor {
  ...
}
```

By default, `@Alternative` beans are disabled. We need to *enable* the alternative—effectively replacing the bean implementation without the `@Alternative` annotation—in the `beans.xml` descriptor of a bean deployment archive by specifying the bean class (or the class that contains the alternative producer method or field). This activation only applies to beans in the same archive.

```
<beans
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives>
      <class>org.mycompany.myapp.StagingCreditCardProcessor</class>
  </alternatives>
</beans>
```

Of course, if the goal is to enable all the alternatives for the staging environment, it would make much more sense to make `@Staging` an `@Alternative` stereotype and annotate the staging beans with this stereotype instead. You'll see how this level of indirection pays off. First, we create the stereotype:

```
@Stereotype
@Alternative
@Retention(RUNTIME)
@Target(TYPE)
```

```
public @interface Staging {}
```

Then we replace the `@Alternative` annotation on our bean with `@Staging`:

```
@CreditCard @Staging
public class StagingCreditCardPaymentProcessor
    extends CreditCardPaymentProcessor {
  ...
}
```

Finally, we activate the `@Staging` stereotype in the beans.xml descriptor:

```
<beans
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives>
      <stereotype>org.mycompany.myapp.Staging</stereotype>
  </alternatives>
</beans>
```

Now, no matter how many staging beans we have, they will all be enabled at once. Does that mean the default implementation is disabled? Well, not exactly. If the default implementation has a qualifier, for instance `@LargeTransaction`, and the alternative does not, you could still inject the default implementation.

```
@LargeTransaction @CreditCard PaymentProcessor paymentProcessor;
```

So we haven't completely replaced the default implementation in a particular deployment of the system. The only way one bean can completely override a second bean at all injection points is if it implements all the bean types and declares all the qualifiers of the second bean. However, if the second bean declares a producer method or observer method, then even this is not enough to ensure that the second bean is never called! We need something extra.

CDI provides a special feature, called *specialization*, that helps the developer avoid these traps. Specialization is a way of informing the system of your intent to completely replace and disable an implementation of a bean.

11.2. Using specialization

When the goal is to replace one bean implementation with a second, to help prevent developer error, the first bean may:

- directly extend the bean class of the second bean, or
- directly override the producer method, in the case that the second bean is a producer method, and then

explicitly declare that it *specializes* the second bean:

```
@Alternative @Specializes
public class MockCreditCardPaymentProcessor
    extends CreditCardPaymentProcessor {
  ...
}
```

When an enabled bean specializes another bean, the other bean is never instantiated or called by the container. Even if the other bean defines a producer or observer method, the method will never be called.

11.3. Inheritance

So why does specialization work, and what does it have to do with inheritance?

Since you are informing the container that your alternative bean is meant to stand in as a replacement for the default implementation, the container automatically adds any qualifiers that are on the default implementation to the alternative implementation. Thus, in our example, `MockCreditCardPaymentProcessor` has the qualifiers `@Default` and `@CreditCard`. Therefore, there are no circumstances when the default implementation is going to get used because it is completely shadowed.

Additionally, if the default implementation declares a bean EL name using `@Named`, the name is inherited by the specialized alternative bean.

Part IV. CDI and the Java EE ecosystem

The third theme of CDI is *integration*. We've already seen how CDI helps integrate EJB and JSF, allowing EJBs to be bound directly to JSF pages. That's just the beginning. The CDI services are integrated into the very core of the Java EE platform. Even EJB session beans can take advantage of the dependency injection, event bus, and contextual lifecycle management that CDI provides.

CDI is also designed to work in concert with technologies outside of the platform by providing integration points into the Java EE platform via an SPI. This SPI positions CDI as the foundation for a new ecosystem of *portable* extensions and integration with existing frameworks and technologies. The CDI services will be able to reach a diverse collection of technologies, such as business process management (BPM) engines, existing web frameworks and defacto standard component models. Of course, The Java EE platform will never be able to standardize all the interesting technologies that are used in the world of Java application development, but CDI makes it easier to use the technologies which are not yet part of the platform seamlessly within the Java EE environment.

We're about to see how to take full advantage of the Java EE platform in an application that uses CDI. We'll also briefly meet a set of SPIs that are provided to support portable extensions to CDI. You might not ever need to use these SPIs directly, but don't take them for granted. You will likely be using them indirectly, every time you use a third-party extension, such as Seam.

Java EE integration

CDI is fully integrated into the Java EE environment. Beans have access to Java EE resources and JPA persistence contexts. They may be used in Unified EL expressions in JSF and JSP pages. They may even be injected into other platform components, such as Servlets and Message-Driven Beans, which are not beans themselves.

12.1. Injecting Java EE resources into a bean

All managed beans may take advantage of Java EE dependency injection using `@Resource`, `@EJB` and `@PersistenceContext`. We've already seen a couple of examples of this, though we didn't pay much attention at the time:

```
@Transactional @Interceptor
public class TransactionInterceptor {
   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

```
@SessionScoped
public class Login {
   @Inject Credentials credentials;
   @PersistenceContext EntityManager userDatabase;
    ...
}
```

The Java EE `@PostConstruct` and `@PreDestroy` callbacks are also supported for all managed beans. The `@PostConstruct` method is called after *all* injection has been performed.

There is one restriction to be aware of here: `@PersistenceContext(type=EXTENDED)` is not supported for non-session beans (that's strictly a feature of stateful session beans).

12.2. Calling a bean from a Servlet

It's easy to use a bean from a Servlet in Java EE 6. Simply inject the bean using field or initializer method injection.

```
public class Login extends HttpServlet {
   @Inject Credentials credentials;
```

```
    @Inject Login login;

    @Override
    public void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        credentials.setUsername(request.getParameter("username")):
        credentials.setPassword(request.getParameter("password")):
        login.login();
        if ( login.isLoggedIn() ) {
            response.sendRedirect("/home.jsp");
        }
        else {
            response.sendRedirect("/loginError.jsp");
        }
    }

}
```

Since instances of Servlets are shared across all incoming threads, the bean client proxy takes care of routing method invocations from the Servlet to the correct instances of `Credentials` and `Login` for the current request and HTTP session.

12.3. Calling a bean from a Message-Driven Bean

CDI injection applies to all EJBs, even when they aren't managed beans. In particular, you can use CDI injection in Message-Driven Beans, which are not considered beans because you can't inject them and their instances are not contextual (not even dependent).

You can even use CDI interceptor bindings for Message-Driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
    @Inject Inventory inventory;
    @PersistenceContext EntityManager em;

    public void onMessage(Message message) {
        ...
    }
}
```

Thus, receiving messages is super-easy in an environment with CDI (e.g., Java EE 6). But beware that there is no session or conversation context available when a message is delivered to a Message-Driven Bean. Only `@RequestScoped` and `@ApplicationScoped` beans are available.

It's also easy to send messages using beans, if you require the full event bus of JMS rather than the architecturally simpler CDI event notification facility.

12.4. JMS endpoints

Sending messages using JMS can be quite complex, because of the number of different objects you need to deal with. For queues we have `Queue`, `QueueConnectionFactory`, `QueueConnection`, `QueueSession` and `QueueSender`. For topics we have `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` and `TopicPublisher`. Each of these objects has its own lifecycle and threading model that we need to worry about.

You can use producer fields and methods to prepare all of these resources for injection into a bean:

```
public class OrderResources {
  @Resource(name="jms/ConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Resource(name="jms/OrderQueue")
  private Queue orderQueue;

  @Produces @OrderConnection
  public Connection createOrderConnection() throws JMSException {
   return connectionFactory.createConnection();
  }

  public void closeOrderConnection(@Disposes @OrderConnection Connection connection)
      throws JMSException {
    connection.close();
  }

  @Produces @OrderSession
  public Session createOrderSession(@OrderConnection Connection connection)
      throws JMSException {
    return connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
  }

  public void closeOrderSession(@Disposes @OrderSession Session session)
      throws JMSException {
    session.close();
  }

  @Produces @OrderMessageProducer
  public MessageProducer createOrderMessageProducer(@OrderSession Session session)
      throws JMSException {
```

```
        return session.createProducer(orderQueue);
    }

    public void closeOrderMessageProducer(@Disposes @OrderMessageProducer MessageProducer producer)
        throws JMSException {
        producer.close();
    }
}
```

In this example, we can just inject the prepared `MessageProducer`, `Connection` or `QueueSession`:

```
@Inject Order order;
@Inject @OrderMessageProducer MessageProducer producer;
@Inject @OrderSession QueueSession orderSession;

public void sendMessage() {
    MapMessage msg = orderSession.createMapMessage();
    msg.setLong("orderId", order.getId());
    ...
    producer.send(msg);
}
```

The lifecycle of the injected JMS objects is completely controlled by the container.

12.5. Packaging and deployment

CDI doesn't define any special deployment archive. You can package beans in JARs, EJB-JARs or WARs # any deployment location in the application classpath. However, the archive must be a "bean archive". That means each archive that contains beans *must* include a file named `beans.xml` in the `META-INF` directory of the classpath or `WEB-INF` directory of the web root (for WAR archives). The file may be empty. Beans deployed in archives that do not have a `beans.xml` file (i.e., not in a bean archive) will not be available for use in the application.

For Java SE execution, beans may be deployed in any location in which EJBs may be deployed for execution by the embeddable EJB Lite container. Again, each location must contain a `beans.xml` file. (That doesn't rule out the possibility of having an extention which provides support for normal Java SE execution, like the Weld Java SE module).

Extending CDI through portable extensions

CDI is intended to be a foundation for frameworks, extensions and integration with other technologies. Therefore, CDI exposes a set of SPIs for the use of developers of portable extensions to CDI. For example, the following kinds of extensions were envisaged by the designers of CDI:

- integration with Business Process Management engines,
- integration with third-party frameworks such as Spring, Seam, GWT or Wicket, and
- new technology based upon the CDI programming model.

More formally, according to the spec:

> A portable extension may integrate with the container by:
>
> - Providing its own beans, interceptors and decorators to the container
>
>   Injecting dependencies into its own objects using the dependency injection service
>
>   Providing a context implementation for a custom scope
>
>   Augmenting or overriding the annotation-based metadata with metadata from some other source

The nerve center for extending CDI is the `BeanManager` object.

13.1. The `BeanManager` object

The `BeanManager` interface lets us register and obtain beans, interceptors, decorators, observers and contexts programatically.

```
public interface Manager {
   public Object getReference(Bean<?> bean, Type beanType, CreationalContext<?> ctx);
   public Object getInjectableReference(InjectionPoint ij, CreationalContext<?> ctx);
   public <T> CreationalContext<T> createCreationalContext(Contextual<T> contextual);
   public Set<Bean<?>> getBeans(Type beanType, Annotation... bindings);
   public Set<Bean<?>> getBeans(String name);
   public <X> Bean<? extends X> getMostSpecializedBean(Bean<X> bean);
   public Bean<?> getPassivationCapableBean(String id);
   public <X> Bean<? extends X> resolve(Set<Bean<? extends X>> beans);
```

```
  public void validate(InjectionPoint injectionPoint);
  public void fireEvent(Object event, Annotation... bindings);
  public <T> Set<ObserverMethod<?, T>> resolveObserverMethods(T event, Annotation... bindings);
  public List<Decorator<?>> resolveDecorators(Set<Type> types, Annotation... bindings);
  public List<Interceptor<?>> resolveInterceptors(InterceptionType type, Annotation... interceptorBindings);
  public boolean isScope(Class<? extends Annotation> annotationType);
  public boolean isNormalScope(Class<? extends Annotation> annotationType);
  public boolean isPassivatingScope(Class<? extends Annotation> annotationType);
  public boolean isQualifier(Class<? extends Annotation> annotationType);
  public boolean isInterceptorBindingType(Class<? extends Annotation> annotationType);
  public boolean isStereotype(Class<? extends Annotation> annotationType);
  public Set<Annotation> getInterceptorBindingTypeDefinition(Class<? extends Annotation> bindingType);
  public Set<Annotation> getStereotypeDefinition(Class<? extends Annotation> stereotype);
  public Context getContext(Class<? extends Annotation> scopeType);
  public ELResolver getELResolver();
  public ExpressionFactory wrapExpressionFactory(ExpressionFactory expressionFactory);
  public <T> AnnotatedType<T> createAnnotatedType(Class<T> type);
  public <T> InjectionTarget<T> createInjectionTarget(AnnotatedType<T> type);
}
```

We can obtain an instance of `BeanManager` via injection:

```
@Inject BeanManager beanManager
```

Java EE components may obtain an instance of BeanManager from JNDI by looking up the name `java:comp/BeanManager`. Any operation of BeanManager may be called at any time during the execution of the application.

Let's study some of the interfaces exposed by the `BeanManager`.

13.2. The `Bean` class

Instances of the interface `Bean` represent beans. There is an instance of `Bean` registered with the `BeanManager` object for every bean in the application.

```
public interface class Bean<T> extends Contextual<T> {
  public Set<Type> getTypes();
  public Set<Annotation> getQualifiers();
  public Class<? extends Annotation> getScope();
```

```
    public String getName();
    public Set<Class<? extends Annotation>> getStereotypes();
    public Class<?> getBeanClass();
    public boolean isAlternative();
    public boolean isNullable();
    public Set<InjectionPoint> getInjectionPoints();
}
```

It's possible to implement the `Bean` interface and register instances by calling `AfterBeanDiscovery.addBean()` (`AfterBeanDiscovery` is a built-in event type that an extension can observe) to provide support for new kinds of beans, beyond those defined by the CDI specification. For example, we could use the `Bean` interface to allow objects managed by another framework to be injected into beans.

There are two subinterfaces of `Bean` defined by the CDI specification: `Interceptor` and `Decorator`.

13.3. The `Context` interface

The `Context` interface supports addition of new scopes to CDI, or extension of the built-in scopes to new environments.

```
public interface Context {
    public Class<? extends Annotation> getScope();
    public <T> T get(Contextual<T> contextual, CreationalContext<T> creationalContext);
    public <T> T get(Contextual<T> contextual);
    boolean isActive();
}
```

For example, we might implement `Context` to add a business process scope to CDI, or to add support for the conversation scope to an application that uses Wicket.

Next steps

Because CDI is so new, there's not yet a lot of information available online. That will change over time. Regardless, the CDI specification remains the authority for information on CDI. The spec is kept at around 100 pages intentionally easy to read (don't worry, it's not like your Blue-ray player manual). You may be especially interested in reading it since it covers many details that we've skipped over. The spec is available on the *JSR-299 page* at the JCP website. Updates to the spec are also frequently mailed to the *weld-dev* mailing list.

The CDI reference implementation, Weld, is being developed at the *Seam project*. The RI development team and the CDI spec lead blog at *in.relation.to*. This guide was originally based on a series of blog entries published there while the specification was being developed. It's probably the best source of information about the future of CDI, Weld and Seam.

We encourage you to follow the *weld-dev* mailing list and to get involved in *development*. If you are reading this guide, you likely have something to offer.

Part V. Weld reference

Weld is the reference implementation of JSR-299, and is used by JBoss AS and Glassfish to provide CDI services for Java Enterprise Edition (Java EE) applications. Weld also goes beyond the environments and APIs defined by the JSR-299 specification by providing support for a number of other environments (such as a servlet container such as Tomcat, or Java SE) and additional APIs and modules (such as logging and bean utilities).

Some of the extensions in Weld are portable across JSR-299 implementations (like the logging and bean utilities) and some are specific to Weld (such as the servlet container support). Weld also provides an SPI on which to build extensions, so there are several layers involved.

If you want to get started quickly using Weld (and, in turn, CDI) with JBoss AS, GlassFish or Tomcat and experiment with one of the examples, take a look at *Chapter 3, Getting started Weld, the JSR-299 reference implementation*. Otherwise read on for a exhaustive discussion of using Weld in all the environments and application servers it supports and the Weld extensions.

Application servers and environments supported by Weld

15.1. Using Weld with JBoss AS

No special configuration is required, beyond making your application a bean archive by adding `META-INF/beans.xml` to the classpath or `WEB-INF/beans.xml` to the web root.

If you are using JBoss AS < 5.2, then you'll need to install Weld as an add-on. Fortunately, the distribution has a build that can handle this for you in a single command. First, we need to tell Weld where JBoss AS is located. Create a new file named local.build.properties in the examples directory of the Weld distribution and assign the path of your JBoss AS installation to the property key `jboss.home`, as follows:

```
jboss.home=/path/to/jboss-as-5.x
```

Now we can install the Weld deployer from the jboss-as directory of the Weld distribution:

```
$> cd jboss-as
$> ant update
```

Note

A new deployer, `weld.deployer` is added to JBoss AS. This adds supports for JSR-299 deployments to JBoss AS, and allows Weld to query the EJB 3 container and discover which EJBs are installed in your application. It also performs a necessary upgrade of the Javassist library.

Weld is built into all releases of JBoss AS from 5.2 onwards. Regardless, you can use this script to update the version of Weld deployed to JBoss AS at any time.

15.2. GlassFish

Weld is also built into GlassFish from V3 onwards. Since GlassFish V3 is the Java EE 6 reference implementation, you can be confident that it will support all features of CDI. What better way for it to support these features than to use the JSR-299 reference implementation? Just package up your CDI application and deploy away!

15.3. Servlet containers (such as Tomcat or Jetty)

While JSR-299 does not require support for Servlet environments, Weld can be used in any Servlet container, such as Tomcat 6.0 or Jetty 6.1.

Note

There is a major limitation to using a Servlet container. Weld doesn't support deploying session beans, injection using `@EJB` or `@PersistenceContext`, or using transactional events in Servlet containers. For enterprise features such as these, you should really be looking at a Java EE application server.

Weld should be used as a web application library in a servlet container. You should place `weld-servlet.jar` in `WEB-INF/lib` in the web root. `weld-servlet.jar` is an "uber-jar", meaning it bundles all the bits of Weld and CDI required for running in a Servlet container, provided for your convenience: Alternatively, you could use its component jars:

- `jsr299-api.jar`
- `weld-api.jar`
- `weld-spi.jar`
- `weld-core.jar`
- `weld-logging.jar`
- `weld-servlet-int.jar`
- `javassist.jar`
- `dom4j.jar`
- `google-collections.jar`

You also need to explicitly specify the servlet listener (used to boot Weld, and control its interaction with requests) in `WEB-INF/web.xml` in the web root:

```
<listener>
   <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener>
```

15.3.1. Tomcat

Tomcat has a read-only JNDI, so Weld can't automatically bind the BeanManager extension SPI. To bind the BeanManager into JNDI, you should populate `META-INF/context.xml` in the web root with the following contents:

```
<Context>
  <Resource name="BeanManager"
    auth="Container"
    type="javax.enterprise.inject.spi.BeanManager"
    factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context>
```

and make it available to your deployment by adding this to the bottom of `web.xml`:

```
<resource-env-ref>
  <resource-env-ref-name>BeanManager</resource-env-ref-name>
  <resource-env-ref-type>
    javax.enterprise.inject.spi.BeanManager
  </resource-env-ref-type>
</resource-env-ref>
```

Tomcat only allows you to bind entries to `java:comp/env`, so the BeanManager will be available at `java:comp/env/BeanManager`

Weld also supports Servlet injection in Tomcat. To enable this, place the `weld-tomcat-support.jar` in `$TOMCAT_HOME/lib`, and add the following to `META-INF/context.xml`:

```
<Listener className="org.jboss.weld.environment.tomcat.WeldLifecycleListener"/>
```

15.3.2. Jetty

Like Tomcat, Jetty has a read-only JNDI, so Weld can't automatically bind the Manager. To bind the Manager to JNDI, you should populate `WEB-INF/jetty-env.xml` with the following contents:

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
  "http://jetty.mortbay.org/configure.dtd">
<Configure id="webAppCtx" class="org.mortbay.jetty.webapp.WebAppContext">
  <New id="BeanManager" class="org.mortbay.jetty.plus.naming.Resource">
    <Arg><Ref id="webAppCtx"/></Arg>
    <Arg>BeanManager</Arg>
    <Arg>
      <New class="javax.naming.Reference">
        <Arg>javax.enterprise.inject.spi.BeanManager</Arg>
        <Arg>org.jboss.weld.resources.ManagerObjectFactory</Arg>
        <Arg/>
```

```
      </New>
    </Arg>
  </New>
</Configure>
```

Notice that Jetty doesn't not have built-in support for an `javax.naming.spi.ObjectFactory` like Tomcat, so it's necessary to manually create the `javax.naming.Reference` to wrap around it.

Jetty only allows you to bind entries to `java:comp/env`, so the BeanManager will be available at `java:comp/env/BeanManager`

Weld does not currently support Servlet injection in Jetty.

15.4. Java SE

Apart from improved integration of the Enterprise Java stack, Weld also provides a state of the art typesafe, stateful dependency injection framework. This is useful in a wide range of application types, enterprise or otherwise. To facilitate this, Weld provides a simple means for executing in the Java Standard Edition environment independently of any Enterprise Edition features.

When executing in the SE environment the following features of Weld are available:

- `POJOs (no EJBs)`
- `Typesafe Dependency Injection`
- `Application and Dependent Contexts`
- `Qualifiers`
- `Stereotypes`
- `Typesafe Event Model`

15.4.1. CDI SE Module

To make life easy for developers, Weld provides a special module with a main method which will boot the CDI bean manager, automatically registering all JavaBeans found on the classpath. This eliminates the need for application developers to write any bootstrapping code. The entry point for a CDI SE applications is a bean which observes the special `ContainerInitialized` event provided by the SE module. The command line paramters can be injected using either of the following:

```
@Parameters List<String> params;
@Parameters String[] paramsArray; // useful for compatability with existing classes
```

Here's an example of a simple CDI SE application:

```
@ApplicationScoped
public class HelloWorld {
  @Parameters List<String> parameters;

  public void printHello(@Observes ContainerInitialized event) {
     System.out.println("Hello " + parameters.get(0));
  }
}
```

CDI SE applications are started by running the following main method.

```
java org.jboss.weld.environments.se.StartMain <args>
```

If you need to do any custom initialization of the CDI bean manager, for example registering custom contexts or initializing resources for your beans you can do so in response to the `AfterBeanDiscovery` or `AfterDeploymentValidation` events. The following example registers a custom context:

```
public class PerformSetup {
  public void setup(@Observes AfterBeanDiscovery event) {
     event.addContext( ThreadContext.INSTANCE );
  }
}
```

Note

The command line parameters do not become available for injection until the `ContainerInitialized` event is fired. If you need access to the parameters during initialization you can do so via the `public static String[] getParameters()` method in `StartMain`.

CDI extensions available as part of Weld

Important

These modules are usable on any JSR-299 implementation, not just Weld!

16.1. Weld Logger

Adding logging to your application is now even easier with simple injection of a logger object into any CDI bean. Simply annotate a `org.jboss.weld.log.Log` type member with the *@Logger* qualifier annotation and an appropriate logger object will be injected into any instance of the bean.

```
import org.jboss.weld.annotation.Logger;
import org.jboss.weld.log.Log;

public class Checkout {
  private @Inject @Logger Log log;

  public void invoiceItems() {
    ShoppingCart cart;
    ...
    log.debug("Items invoiced for {0}", cart);
  }
}
```

The example shows how objects can be interpolated into a message. This interpolation is done using *java.text.MessageFormat*, so see the JavaDoc for that class for more details. In this case, the `ShoppingCart` should have implemented the *toString()* method to produce a human readable value that is meaningful in messages. Normally, this call would have involved evaluating cart.toString() with String concatenation to produce a single String argument. Thus it was necessary to surround the call with an if-statement using the condition *log.isDebugEnabled()* to avoid the expensive String concatenation if the message was not actually going to be used. However, when using `@Logger`-injected logging, the conditional test can be left out since the object arguments are not evaluated unless the message is going to be logged.

Note

You can add the Weld Logger to your project by including weld-logger.jar, sl4j-api.jar and sl4j-jdk14.jar to your project. Alternatively, express a dependency on the `org.jboss.weld:weld-logger` Maven artifact.

If you are using Weld as your JSR-299 implementation, there's no need to include sl4j as it's already included (and used internally).

Alternative view layers

17.1. Wicket CDI integration

Weld provides integration between the Apache Wicket web framework and CDI. This functionality is provided by the `weld-wicket` extension module, which naturally must be on the classpath of the Wicket application.

This section describes some of the utilities provided by the Wicket extension module to support the CDI integration.

17.1.1. The `WebApplication` class

Each wicket application must have a `WebApplication` subclass; Weld provides, for your utility, a subclass of this which sets up the Wicket CDI integration. You should subclass `org.jboss.weld.wicket.WeldApplication`.

Note

If you would prefer not to subclass `WeldApplication`, you can manually add a (small!) number of overrides and listeners to your own `WebApplication` subclass. The JavaDocs of `WeldApplication`detail this.

For example:

```
public class SampleApplication extends WeldApplication {
   @Override
   public Class getHomePage() {
      return HomePage.class;
   }
}
```

17.1.2. Conversations with Wicket

Wicket can also take advantage of the conversation scope from CDI, provided by the Wicket extension module. This module takes care of:

- Setting up the conversation context at the beginning of a Wicket request, and tearing it down afterwards
- Storing the id of any long-running conversation in Wicket's metadata when the page response is complete

- Activating the correct long-running conversation based upon which page is being accessed
- Propagating the conversation context for any long-running conversation to new pages

17.1.2.1. Starting and stopping conversations in Wicket

As in JSF applications, a conversation *always* exists for any request to Wicket, but its lifetime is only that of the current request unless it is marked as *long-running*. The boundaries of a long-running conversation are controlled in the same way as in JSF applications, by injecting the `Conversation` instance and invoking either the `begin()` or `end()` methods:

```
private @Inject Conversation conversation;
...
// begin a conversation
conversation.begin();
...
// end a conversation
conversation.end();
```

17.1.2.2. Long running conversation propagation in Wicket

When a conversation is marked as long-running, the id of that conversation will be stored in Wicket's metadata for the current page. If a new page is created and set as the response target through `setResponsePage()`, this new page will also participate in this conversation. This occurs for both directly instantiated pages (`setResponsePage(new OtherPage())`), as well as for bookmarkable pages created with `setResponsePage(OtherPage.class)` where `OtherPage.class` is mounted as bookmarkable from your `WebApplication` subclass (or through annotations). In the latter case, because the new page instance is not created until after a redirect, the conversation id will be propagated through a request parameter, and then stored in page metadata after the redirect.

Appendix A. Integrating Weld into other environments

If you want to use Weld in another environment, you will need to provide certain information to Weld via the integration SPI. In this Appendix we will briefly discuss the steps needed.

Enterprise Services

If you just want to use managed beans, and not take advantage of enterprise services (EE resource injection, CDI injection into EE component classes, transactional events, support for CDI services in EJBs) and non-flat deployments, then the generic servlet support provided by the "Weld: Servlets" extension will be sufficient, and will work in any container supporting the Servlet API.

All SPIs and APIs described have extensive JavaDoc, which spell out the detailed contract between the container and Weld.

A.1. The Weld SPI

The Weld SPI is located in the `weld-spi` module, and packaged as `weld-spi.jar`. Some SPIs are optional, and should only be implemented if you need to override the default behavior; others are required.

All interfaces in the SPI support the decorator pattern and provide a `Forwarding` class located in the `helpers` sub package. Additional, commonly used, utility classes, and standard implementations are also located in the `helpers` sub package.

Weld supports multiple environments. An environment is defined by an implementation of the `Environment` interface. A number of standard environments are built in, and described by the `Environments` enumeration. Different environments require different services to be present (for example a Servlet container doesn't require transaction, EJB or JPA services). By default an EE environment is assumed, but you can adjust the environment by calling `bootstrap.setEnvironment()`.

Weld uses a generic-typed service registry to allow services to be registered. All services implement the `Service` interface. The service registry allows services to be added and retrieved.

A.1.1. Deployment structure

An application is often comprised of a number of modules. For example, a Java EE deployment may contain a number of EJB modules (containing business logic) and WAR modules (containing the user interface). A container may enforce certain *accessibility* rules which limit the visibility of

classes between modules. CDI allows these same rules to apply to bean and observer method resolution. As the accessibility rules vary between containers, Weld requires the container to *describe* the deployment structure, via the `Deployment` SPI.

The CDI specification discusses *Bean Deployment Archives* (BDAs)—archives which are marked as containing beans which should be deployed to the CDI container, and made available for injection and resolution. Weld reuses this description of *Bean Deployment Archives* in its deployment structure SPI. Each deployment exposes the BDAs which it contains; each BDA may also reference other which it can access. Together, the transitive closure of this graph forms the beans which are deployed in the application.

To describe the deployment structure to Weld, the container should provide an implementation of `Deployment`. `Deployment.getBeanDeploymentArchives()` allows Weld to discover the modules which make up the application. The CDI specification also allows beans to be specified programmatically as part of the bean deployment. These beans may, or may not, be in an existing BDA. For this reason, Weld will call `Deployment.loadBeanDeploymentArchive(Class clazz)` for each programmatically described bean.

As programmatically described beans may result in additional BDAs being added to the graph, Weld will discover the BDA structure every time an unknown BDA is returned by `Deployment.loadBeanDeploymentArchive`.

Virtual BDAs

In a strict container, each BDA might have to explicitly specify which other BDAs it can access. However many containers will allow an easy mechanism to make BDAs bi-directionally accessible (such as a library directory). In this case, it is allowable (and reasonable) to describe all such archives as a single, 'virtual' `BeanDeploymentArchive`.

A container, might, for example, use a flat accessibility structure for the application. In this case, a single `BeanDeploymentArchive` would be attached to the `Deployment`.

`BeanDeploymentArchive` provides three methods which allow it's contents to be discovered by Weld—`BeanDeploymentArchive.getBeanClasses()` must return all the classes in the BDA, `BeanDeploymentArchive.getBeansXml()` must return all the deployment descriptors in the archive, and `BeanDeploymentArchive.getEjbs()` must provide an EJB descriptor for every EJB in the BDA, or an empty list if it is not an EJB archive.

BDA X may also reference another BDA Y whose beans can be resolved by, and injected into, any bean in BDA X. These are the accessible BDAs, and every BDA that is directly accessible by BDA X should be returned. A BDA will also have BDAs which are accessible transtivively, and the transitive closure of the sub-graph of BDA X describes all the beans resolvable by BDA X.

Matching the classloader structure for the deployment

In practice, you can regard the deployment structure represented by `Deployment`, and the virtual BDA graph as a mirror of the classloader structure for a deployment. If a class can from BDA X can be loaded by another in BDA Y, it is accessible, and therefore BDA Y's accessible BDAs should include BDA X.

To specify the directly accessible BDAs, the container should provide an implementation of `BeanDeploymentArchive.getBeanDeploymentArchives()`.

Tip

Weld allows the container to describe a circular graph, and will convert a graph to a tree as part of the deployment process.

Certain services are provided for the whole deployment, whilst some are provided per-BDA. BDA services are provided using `BeanDeploymentArchive.getServices()` and only apply to the BDA on which they are provided.

A.1.2. EJB descriptors

Weld delegates EJB 3 bean discovery to the container so that it doesn't duplicate the work done by the EJB container, and respects any vendor-extensions to the EJB definition.

The `EjbDescriptor` should return the relevant metadata as defined in the EJB specification. Each business interface of a session bean should be described using a `BusinessInterfaceDescriptor`.

A.1.3. EE resource injection and resolution services

All the EE resource services are per-BDA services, and may be provided using one of two methods. Which method to use is at the discretion of the integrator.

The integrator may choose to provide all EE resource injection services themselves, using another library or framework. In this case the integrator should use the `EE` environment, and implement the *Section A.1.8, “Injection Services”* SPI.

Alternatively, the integrator may choose to use CDI to provide EE resource injection. In this case, the `EE_INJECT` environment should be used, and the integrator should implement the *Section A.1.4, “EJB services”[138]*, *Section A.1.7, “Resource Services”*and *Section A.1.5, “JPA services”*.

Important

CDI only provides annotation-based EE resource injection; if you wish to provide deployment descriptor (e.g. `ejb-jar.xml`) injection, you must use *Section A.1.8, "Injection Services"*.

If the container performs EE resource injection, the injected resources must be serializable. If EE resource injection is provided by Weld, the resolved resource must be serializable.

Tip

If you use a non-EE environment then you may implement any of the EE service SPIs, and Weld will provide the associated functionality. There is no need to implement those services you don't need!

A.1.4. EJB services

EJB services are split between two interfaces which are both per-BDA.

`EJBServices` is used to resolve local EJBs used to back session beans, and must always be provided in an EE environment. `EJBServices.resolveEjb(EjbDescriptor ejbDescriptor)` returns a wrapper—`SessionObjectReference`—around the EJB reference. This wrapper allows Weld to request a reference that implements the given business interface, and, in the case of SFSBs, both request the removal of the EJB from the container and query whether the EJB has been previously removed.

`EJBResolutionServices.resolveEjb(InjectionPoint ij)` allows the resolution of `@EJB` (for injection into managed beans). This service is not required if the implementation of *Section A.1.8, "Injection Services"* takes care of `@EJB` injection.

A.1.5. JPA services

Just as EJB resolution is delegated to the container, resolution of `@PersistenceContext` for injection into managed beans (with the `InjectionPoint` provided), is delegated to the container.

To allow JPA integration, the `JpaServices` interface should be implemented. This service is not required if the implementation of *Section A.1.8, "Injection Services"* takes care of `@PersistenceContext` injection.

A.1.6. Transaction Services

Weld delegates JTA activities to the container. The SPI provides a couple hooks to easily achieve this with the `TransactionServices` interface.

Any `javax.transaction.Synchronization` implementation may be passed to the `registerSynchronization()` method and the SPI implementation should immediately register the synchronization with the JTA transaction manager used for the EJBs.

To make it easier to determine whether or not a transaction is currently active for the requesting thread, the `isTransactionActive()` method can be used. The SPI implementation should query the same JTA transaction manager used for the EJBs.

A.1.7. Resource Services

The resolution of `@Resource` (for injection into managed beans) is delegated to the container. You must provide an implementation of `ResourceServices` which provides these operations. This service is not required if the implementation of *Section A.1.8, "Injection Services"* takes care of `@Resource` injection.

A.1.8. Injection Services

An integrator may wish to use `InjectionServices` to provide additional field or method injection over-and-above that provided by Weld. An integration into a Java EE environment may use `InjectionServices` to provide EE resource injection for managed beans.

`InjectionServices` provides a very simple contract, the `InjectionServices.aroundInject(InjectionContext ic);` intercepter will be called for every instance that CDI injects, whether it is a contextual instance, or a non-contexutal instance injected by `InjectionTarget.inject()`.

The `InjectionContext` can be used to discover additional information about the injection being performed, including the `target` being injected. `ic.proceed()` should be called to perform CDI-style injection, and call initializer methods.

A.1.9. Security Services

In order to obtain the `Principal` representing the current caller identity, the container should provide an implementation of `SecurityServices`.

A.1.10. Bean Validation Services

In order to obtain the default `ValidatorFactory` for the application deployment, the container should provide an implementation of `ValidationServices`.

A.1.11. Identifying the BDA being addressed

When a client makes a request to an application which uses Weld, the request may be addressed at any of the BDAs in the application deployment. To allow Weld to correctly service the request, it needs to know which BDA the request is addressed at. Where possible, Weld will provide some context, but use of these by the integrator is optional.

Tip

Most Servlet contains use a classloader-per-WAR, this may provide a good way to identify the BDA in use for web requests.

When Weld needs to identify the BDA, it will use one of these services, depending on what is servicing the request:

`ServletServices.getBeanDeploymentArchive(ServletContext ctx)`
: Identify the WAR in use. The `ServletContext` is provided for additional context.

A.1.12. The bean store

Weld uses a map like structure to store bean instances - `org.jboss.weld.context.api.BeanStore`. You may find `org.jboss.weld.context.api.helpers.ConcurrentHashMapBeanStore` useful.

A.1.13. The application context

Weld expects the Application Server or other container to provide the storage for each application's context. The `org.jboss.weld.context.api.BeanStore` should be implemented to provide an application scoped storage.

A.1.14. Initialization and shutdown

The `org.jboss.weld.bootstrap.api.Bootstrap` interface defines the initialization for Weld, bean deployment and bean validation. To boot Weld, you must create an instance of `org.jboss.weld.bootstrap.WeldBeansBootstrap` (which implements `Boostrap`), tell it about the services in use, and then request the container start.

The bootstrap is split into phases, container initialization, bean deployment, bean validation and shutdown. Initialization will create a manager, and add the built-in contexts, and examine the deployment structure. Bean deployment will deploy any beans (defined using annotations, programtically, or built in). Bean validation will validate all beans.

To initialize the container, you call `Bootstrap.startInitialization()`. Before calling `startInitialization()`, you must register any services required by the environment. You can do this by calling, for example, `bootstrap.getServices().add(JpaServices.class, new MyJpaServices())`. You must also provide the application context bean store.

Having called `startInitialization()`, the `Manager` for each BDA can be obtained by calling `Bootstrap.getManager(BeanDeploymentArchive bda)`.

To deploy the discovered beans, call `Bootstrap.deployBeans()`.

To validate the deployed beans, call `Bootstrap.validateBeans()`.

To place the container into a state where it can service requests, call `Bootstrap.endInitialization()`

To shutdown the container you call `Bootstrap.shutdown()`. This allows the container to perform any cleanup operations needed.

A.1.15. Resource loading

Weld needs to load classes and resources from the classpath at various times. By default, they are loaded from the Thread Context ClassLoader if available, if not the same classloader that was used to load Weld, however this may not be correct for some environments. If this is case, you can implement `org.jboss.weld.spi.ResourceLoader`.

A.2. The contract with the container

There are a number of requirements that Weld places on the container for correct functioning that fall outside implementation of APIs.

Classloader isolation

If you are integrating Weld into an environment that supports deployment of multiple applications, you must enable, automatically, or through user configuation, classloader isolation for each CDI application.

Servlet

If you are integrating Weld into a Servlet environment you must register `org.jboss.weld.servlet.WeldListener` as a Servlet listener, either automatically, or through user configuration, for each CDI application which uses Servlet.

JSF

If you are integrating Weld into a JSF environment you must register `org.jboss.weld.jsf.WeldPhaseListener` as a phase listener.

If you are integrating Weld into a JSF environment you must register `org.jboss.weld.el.WeldELContextListener` as as an EL Context listener.

If you are integrating Weld into a JSF environment you must obtain the bean manager for the module and then call `BeanManager.wrapExpressionFactory()`, passing `Application.getExpressionFactory()` as the argument. The wrapped expression factory must be used in all EL expression evaluations performed by JSF in this web application.

If you are integrating Weld into a JSF environment you must obtain the bean manager for the module and then call `BeanManager.getELResolver()`, The returned EL resolver should be registered with JSF for this web application.

Tip

There are a number of ways you can obtain the bean manager for the module. You could call `Bootstrap.getManager()`, passing in the BDA for this module.

Alternatively, you could use the injection into Java EE component classes, or look up the bean manager in JNDI.

If you are integrating Weld into a JSF environment you must register `org.jboss.weld.servlet.ConversationPropagationFilter` as a Servlet listener, either automatically, or through user configuration, for each CDI application which uses JSF. This filter can be registered for all Servlet deployment safely.

Note

Weld only supports JSF 1.2 and above.

JSP

If you are integrating Weld into a JSP environment you must register `org.jboss.weld.el.WeldELContextListener` as as an EL Context listener.

If you are integrating Weld into a JSP environment you must obtain the bean manager for the module and then call `BeanManager.wrapExpressionFactory()`, passing `Application.getExpressionFactory()` as the argument. The wrapped expression factory must be used in all EL expression evaluations performed by JSP.

If you are integrating Weld into a JSP environment you must obtain the bean manager for the module and then call `BeanManager.getELResolver()`, The returned EL resolver should be registered with JSP for this web application.

Tip

There are a number of ways you can obtain the bean manager for the module. You could call `Bootstrap.getManager()`, passing in the BDA for this module. Alternatively, you could use the injection into Java EE component classes, or look up the bean manager in JNDI.

Session Bean Interceptor

If you are integrating Weld into an EJB environment you must register `org.jboss.weld.ejb.SessionBeanInterceptor` as a EJB interceptor for all EJBs in the application, either automatically, or through user configuration, for each CDI application which uses enterprise beans.

Important

You must register the `SessionBeanInterceptor` as the inner most interceptor in the stack for all EJBs.

The `weld-core.jar`
: Weld can reside on an isolated classloader, or on a shared classloader. If you choose to use an isolated classloader, the default `SingletonProvider`, `IsolatedStaticSingletonProvider`, can be used. If you choose to use a shared classloader, then you will need to choose another strategy.

  You can provide your own implementation of `Singleton` and `SingletonProvider` and register it for use using `SingletonProvider.initialize(SingletonProvider provider)`.

  Weld also provides an implementation of Thread Context Classloader per application strategy, via the `TCCLSingletonProvider`.

Binding the manager in JNDI
: You should bind the bean manager for the bean deployment archive into JNDI at `java:comp/Manager`. The type should be `javax.enterprise.inject.spi.BeanManager`. To obtain the correct bean manager for the bean deployment archive, you may call `bootstrap.getBeanManager(beanDeploymentArchive)`

Performing CDI injection on Java EE component classes
: The CDI specification requires the container to provide injection into non-contextual resources for all Java EE component classes. Weld delegates this responsibility to the container. This can be achieved using the CDI defined `InjectionTarget` SPI. Furthermore, you must perform this operation on the correct bean manager for the bean deployment archive containing the EE component class.

  The CDI specification also requires that a `ProcessInjectionTarget` event is fired for every Java EE component class. Furthermore, if an observer calls `ProcessInjectionTarget.setInjectionTarget()` the container must use *the specified* injection target to perform injection.

  To help the integrator, Weld provides `WeldManager.fireProcessInjectionTarget()` which returns the `InjectionTarget` to use.

```
// Fire ProcessInjectionTarget, returning the InjectionTarget
// to use
InjectionTarget it = weldBeanManager.fireProcessInjectionTarget(clazz);

// Per instance required, create the creational context
CreationalContext<?> cc = beanManager.createCreationalContext(null);

// Produce the instance, performing any constructor injection required
Object instance = it.produce();

// Perform injection and call initializers
it.inject(instance, cc);
```

```
// Call the post-construct callback
it.postConstruct(instance);

// Call the pre-destroy callback
it.preDestroy(instance);

// Clean up the instance
it.dispose();
cc.release();
```

The container may intersperse other operations between these calls. Further, the integrator may choose to implement any of these calls in another manner, assuming the contract is fulfilled.

When performing injections on EJBs you must use the Weld-defined SPI, `WeldManager`. Furthermore, you must perform this operation on the correct bean manager for the bean deployment archive containing the EJB.

```
// Obtain the EjbDescriptor for the EJB
// You may choose to use this utility method to get the descriptor
EjbDescriptor<?> ejbDescriptor = beanManager.getEjbDescriptor(ejbName);

// Get an the Bean object
Bean<?> bean = beanManager.getBean(ejbDescriptor);

// Create the injection target
InjectionTarget it = deploymentBeanManager.createInjectionTarget(ejbDescriptor);

// Per instance required, create the creational context
CreationalContext<?> cc = deploymentBeanManager.createCreationalContext(bean);

// Perform injection and call initializers
it.inject(instance, cc);

// You may choose to have CDI call the post construct and pre destroy
// lifecycle callbacks

// Call the post-construct callback
it.postConstruct(instance);

// Call the pre-destroy callback
it.preDestroy(instance);
```

```
// Clean up the instance
it.dispose();
cc.release();
```