CDI 1.2 User guide

Antoine Sabot-Durand, Martin Kouba, Matej Novotny, Tomas Remes

Version 1.0, May 2016

Table of Contents

Foreword

This documentation is a fork of the Weld (CDI Reference implementation) documentation. All Weld specific references were removed in order to provide implementation independent introduction to CDI programming model.

This document is maintained by the authors listed below its title yet, it is a collective work including the following people:

- Jozef Hartinger
- Pete Muir
- Dan Allen
- David Allen
- Gavin King

While this document is probably easier to read than the specification, we encourage you to take a look at it, should you need deeper understanding of the CDI platform.

The source of this guide is available on github Feel free to fill ticket or send PR if you want to make this document better.

Beans

The CDI specification defines a set of complementary services that help improve the structure of application code. CDI layers an enhanced lifecycle and interaction model over existing Java component types, including managed beans and Enterprise Java Beans. The CDI services provide:

- an improved lifecycle for stateful objects, bound to well-defined *contexts*,
- a typesafe approach to *dependency injection*,
- object interaction via an *event notification facility*,
- a better approach to binding *interceptors* to objects, 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 to the container.

The CDI services are a core aspect of the Java EE platform and include full support for Java EE modularity and the Java EE component architecture.

CDI is especially useful in the context of web application development, but the problems it solves are general development concerns and it is therefore applicable to a wide variety of application.

An object bound to a lifecycle context is called a bean. CDI includes built-in support for several different kinds of bean, including the following Java EE component types:

- managed beans, and

- EJB session beans.

Both managed beans and EJB session beans may inject other beans. But some other objects, which are not themselves beans in the sense used here, may also have beans injected via CDI. In the Java EE platform, the following kinds of component may have beans injected:

- message-driven beans,
- interceptors,
- servlets, servlet filters and servlet event listeners,
- JAX-WS service endpoints and handlers,
- JAX-RS resources, providers and `javax.ws.rs.core.Application` subclasses, and
- JSP tag handlers and tag library event listeners.

CDI relieves the user of an unfamiliar API of the need 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 get access to it from a client?
- 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 an alternative implementation, so that the implementation can vary at deployment time?
- How should I go about sharing this object between other objects?

CDI is more than a framework. It's a whole, rich programming model. The *theme* of CDI is *loose-coupling 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 interacts with. Even better, the concrete implementation, lifecycle and threading model of a bean may vary according to the deployment scenario, without affecting any client. This loose-coupling makes your code easier to maintain.

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. Instead, CDI uses the typing information that is already available in the Java object model, augmented using a new programming pattern, called *qualifier annotations*, to wire together beans, their dependencies, their interceptors and decorators, and their event consumers. Usage of XML descriptors is minimized to truly deployment-specific information.

But CDI isn't a restrictive programming model. It doesn't tell you how you should to structure your application into layers, how you should handle persistence, or what web framework you have to use. You'll have to decide those kinds of things for yourself.

CDI even provides a comprehensive SPI, allowing other kinds of object defined by future Java EE specifications or by third-party frameworks to be cleanly integrated with CDI, take advantage of the CDI services, and interact with any other kind of bean.

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 and JSR-346 Expert Groups (EG).

Finally, CDI is a Java Community Process (JCP) standard. Java EE 7 requires that all compliant application servers provide support for JSR-346 (even in the web profile).

Chapter 1. Introduction

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 makes it easier to actually use them to build an application!

1.1. What is a bean?

A bean is exactly what you think it is. Only now, it has a true identity in the container environment.

Prior to Java EE 6, there was no clear definition of the term "bean" in the Java EE platform. Of course, we've been calling Java classes used in web and enterprise applications "beans" for years. There were even a couple of different kinds of things called "beans" in EE specifications, including EJB beans and JSF managed beans. Meanwhile, other third-party frameworks such as Spring and Seam introduced their own ideas of what it meant to be a "bean". What we've been missing is a common definition.

Java EE 6 finally laid down that common definition in the Managed Beans specification. Managed Beans are defined as container-managed objects with minimal programming restrictions, otherwise known by the acronym POJO (Plain Old Java Object). 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's a uniform concept of a bean and a lightweight component model that's aligned across the Java EE platform.

With very few exceptions, almost every concrete Java class that has a constructor with no parameters (or a constructor designated with the annotation `@Inject`) is a bean. This includes every JavaBean and every EJB session bean. If you've already got some JavaBeans or session beans lying around, they're already beans—you won't need any additional special metadata.

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 designated context, injecting them into other beans, using them in EL expressions, specializing them with qualifier annotations, even adding interceptors and decorators to them—without modifying your existing code. At most, you'll need to add some annotations.

Now let's see how to create your first bean that actually uses CDI.

1.2. Getting our feet wet

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 into the parameters of the constructor or method.

We may obtain an instance of `TextTranslator` by injecting it into a constructor, method or field of a bean, or a field or method of a Java EE component class such as a servlet. The container chooses the object to be injected based on the type of the injection point, not the name of the field, method or parameter.

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

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` programmatically from an injected instance of `Instance`, parameterized with the bean type:

```
import javax.enterprise.inject.Instance;
import javax.inject.Inject;

....

@Inject Instance<TextTranslator> textTranslatorInstance;
...
public void translate() {
   textTranslatorInstance.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 an injected field directly (even if it's private!), which sometimes helps 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 inform us of an *unsatisfied dependency*. If more than one implementation of `Translator` were available, the container would inform us of the *ambiguous dependency*.

Before we get too deep in the details, let's pause and examine a bean's anatomy. What aspects of the bean are significant, and what gives it its identity? Instead of just giving examples of beans, we're going to define what *makes* something a bean.

Chapter 2. More about beans

A bean is usually an application class that contains business logic. It may be called directly from Java code, or it may be invoked via the 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 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 lifecycle 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 Alternatives.

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.

2.1. The anatomy of a bean

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

> A bean comprises the following attributes:
>
> - A (nonempty) set of bean types
> - A (nonempty) set of qualifiers
> - A scope
> - Optionally, a bean EL name
> - A set of interceptor bindings
> - A bean implementation
>
> Furthermore, a bean may or may not be an alternative.

Let’s see what all this new terminology means.

2.1.1. 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; a type that is 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.

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

The bean types of a session bean include local interfaces and the bean class local view (if any). EJB remote interfaces are not considered bean types of a session bean. You can't inject an EJB using its remote interface unless you define a *resource*, which we'll meet in Java EE component environment resources.

Bean types may be restricted to an explicit set by annotating the bean with the `@Typed` annotation and listing the classes that should be bean types. For instance, the bean types of this bean have been restricted to `Shop<Book>`, together with `java.lang.Object`:

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

Sometimes, a bean type alone does not provide enough information for the container to know which bean to inject. For instance, suppose we have two implementations of the `PaymentProcessor` interface: `CreditCardPaymentProcessor` and `DebitPaymentProcessor`. Injecting a field of type `PaymentProcessor` introduces an ambiguous condition. In these cases, the client must specify some additional quality of the implementation it is interested in. We model this kind of "quality" using a qualifier.

A qualifier is a user-defined annotation that is itself annotated `@Qualifier`. A qualifier annotation is an extension of the type system. It lets us disambiguate a type without having to fall back to string-based names. Here's an example of a qualifier annotation:

```java
@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've encountered one. With CDI, annotation definitions will become a familiar artifact as you'll be creating them from time to time.

Pay attention to the names of the built-in annotations in CDI and EJB. You'll notice that they are often adjectives. We encourage you 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 to disambiguate an injection point. The following injection point has the bean type `PaymentProcessor` and qualifier `@CreditCard`:

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

For each injection point, the container searches for a bean which satisfies the contract, one which has the bean type and all the qualifiers. If it finds exactly one matching bean, it injects an instance of that bean. If it doesn't, it reports an error to the user.

How do we specify that qualifiers of a bean? By annotating the bean class, 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 { ... }
```

If a bean or an injection point does not explicitly specify a qualifier, it has the default qualifier, `@Default`.

That's not quite the end of the story. CDI also defines a simple *resolution rule* 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 Dependency injection and programmatic lookup.

2.1.2. Scope

The *scope* of a bean defines the lifecycle and visibility of its instances. The CDI context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built into 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:

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

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.

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 manually remove a bean from a context. If you don't want the bean to sit in the session indefinitely, consider using another scope with a shorted lifespan, such as the request or conversation scope.

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 object into which they were injected, which means their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in Scopes and contexts.

2.1.3. EL name

If you want to reference a bean in non-Java code that supports Unified EL expressions, for example, in a JSP or JSF page, you must assign the bean an *EL name*.

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

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

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

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

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.

We can let CDI choose a name for us by leaving off the value of the `@Named` annotation:

```
public @SessionScoped @Named
class ShoppingCart implements Serializable { ... }
```

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

2.1.4. Alternatives

We've already seen how qualifiers let us choose between multiple implementations of an interface at development time. But sometimes we have an interface (or other bean type) whose implementation varies depending upon the deployment environment. For example, we may want to use a mock implementation in a testing environment. An *alternative* may be declared by annotating the bean class with the @Alternative annotation.

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

We normally annotate a bean @Alternative only when there is some other implementation of an interface it implements (or of any of its bean types). We can choose between alternatives at deployment time by *selecting* an alternative in the CDI deployment descriptor META-INF/beans.xml of the jar or Java EE module that uses it. Different modules can specify that they use different alternatives.

We cover alternatives in more detail in Alternatives.

2.1.5. Interceptor binding types

You might be familiar with the use of interceptors in EJB 3. Since Java EE 6, this functionality has been generalized to work with other managed beans. That's right, you no longer have to make your bean an EJB just to intercept its methods. Holler. So what does CDI have to offer above and beyond that? Well, quite a lot actually. Let's cover some background.

The way that interceptors were defined in Java EE 5 was counter-intuitive. You were 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 repeatedly, then there's a good chance that you'll inadvertently define a different order for different beans. Now that's a problem.

CDI provides a new approach to binding interceptors to beans that introduces a level of indirection (and thus control). We must define an *interceptor binding type* to describe the behavior implemented by the interceptor.

An interceptor binding type is a user-defined annotation that is itself annotated @InterceptorBinding. It lets us bind interceptor classes to bean classes with no direct dependency between the two classes.

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

The interceptor that implements transaction management declares this annotation:

```
public @Transactional @Interceptor
class TransactionInterceptor { ... }
```

We can apply the interceptor to a bean by annotating the bean class with the same interceptor binding type:

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

Notice that `ShoppingCart` and `TransactionInterceptor` don't know anything about each other.

Interceptors are deployment-specific. (We don't need a `TransactionInterceptor` in our unit tests!) By default, an interceptor is disabled. We can enable an interceptor using the CDI deployment descriptor `META-INF/beans.xml` of the jar or Java EE module. This is also where we specify the interceptor ordering.

We'll discuss interceptors, and their cousins, decorators, in Interceptors and Decorators.

2.2. What kinds of classes are beans?

We've already seen two types of beans: JavaBeans and EJB session beans. Is that the whole story? Actually, it's just the beginning. Let's explore the various kinds of beans that CDI implementations must support out-of-the-box.

2.2.1. Managed beans

A managed bean is a Java class. The basic lifecycle and semantics of a managed bean are defined by the Managed Beans specification. You can explicitly declare a managed bean by annotating the bean class `@ManagedBean`, but in CDI you don't need to. According to the specification, the CDI container treats any class that satisfies the following conditions as a managed bean:

- 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 does not implement `javax.enterprise.inject.spi.Extension`.
- It has an appropriate constructor—either:
  - the class has a constructor with no parameters, or
  - the class declares a constructor annotated `@Inject`.

According to this definition, JPA entities are technically managed beans. However, entities have their own special lifecycle, state and identity model and are usually instantiated by JPA or using `new`. Therefore we don't recommend directly injecting an entity class. We especially recommend against assigning a scope other than `@Dependent` to an entity class, since JPA is not able to persist injected CDI proxies.

The unrestricted set of bean types for a managed bean contains the bean class, every superclass and all interfaces it implements directly or indirectly.

If a managed bean has a public field, it must have the default scope `@Dependent`.

Managed beans support the `@PostConstruct` and `@PreDestroy` lifecycle callbacks.

Session beans are also, technically, managed beans. However, since they have their own special lifecycle and take advantage of additional enterprise services, the CDI specification considers them to be a different kind of bean.

2.2.2. Session beans

Session beans belong to the EJB specification. They have a special lifecycle, state management and concurrency model that is different to other managed beans and non-managed Java objects. But session beans participate in CDI just like any other bean. You can inject one session bean into another session bean, 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.

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

There's no reason to explicitly declare the scope of a stateless session bean or singleton session bean. The EJB container controls the lifecycle of these beans, according to the semantics of the `@Stateless` or `@Singleton` declaration. On the other hand, a stateful session bean may have any scope.

Stateful session beans may define a *remove method*, annotated `@Remove`, that is used by the application to indicate that an instance should be destroyed. However, for a contextual instance of the bean—an instance under the control of CDI—this method may only be called by the application if the bean has scope `@Dependent`. For beans with other scopes, the application must let the container destroy the bean.

So, when should we use a session bean instead of a plain 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 or web service invocation, or
- timers and asynchronous methods,

When we don’t need any of these things, an ordinary managed bean will serve just fine.

Many beans (including any `@SessionScoped` or `@ApplicationScoped` beans) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.2 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 we’re trying to make is: use a session bean when you need the services it provides, not just because you want to use dependency injection, lifecycle management, or interceptors. Java EE 7 provides a graduated programming model. It’s usually easy to start with an ordinary managed bean, and later turn it into an EJB just by adding one of the following annotations: `@Stateless`, `@Stateful` or `@Singleton`.

On the other hand, don’t be scared to use session beans just because you’ve heard your friends say they’re "heavyweight". It’s nothing more than superstition to think that something is "heavier" just because it’s hosted natively within the Java EE container, instead of by a proprietary bean container or dependency injection framework that runs as an additional layer of obfuscation. And as a general principle, you should be skeptical of folks who use vaguely defined terminology like "heavyweight".

2.2.3. Producer methods

Not everything that needs to be injected can be boiled down to a bean class instantiated by the container using `new`. There are plenty of cases where we need additional control. What if we need to decide at runtime which implementation of a type to instantiate and inject? What if we need to inject an object that is obtained by querying a service or transactional resource, for example by executing a JPA query?

A *producer method* is a method that acts as a source of bean instances. The method declaration itself describes the bean and the container invokes the method to obtain an instance of the bean when no instance exists in the specified context. A producer method lets the application take full control of the bean instantiation process.

A producer method is declared by annotating a method of a bean class with the `@Produces` annotation.

```
import javax.enterprise.inject.Produces;

@ApplicationScoped
public class RandomNumberGenerator {

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

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

}
```

We can't write a bean class that is itself a random number. But we can certainly write a method that returns a random number. By making the method a producer method, we allow the return value of the method—in this case an `Integer`—to be injected. We can even specify a qualifier—in this case `@Random`, a scope—which in this case defaults to `@Dependent`, and an EL name—which in this case defaults to `randomNumber` according to the JavaBeans property name convention. Now we can get a random number anywhere:

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

Even in a Unified EL expression:

```
<p>Your raffle number is #{randomNumber}.</p>
```

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.

Producer methods and fields may have a primitive bean type. For the purpose of resolving dependencies, primitive types are considered to be identical to their corresponding wrapper types in `java.lang`.

If the producer method has method parameters, the container will look for a bean that satisfies the type and qualifiers of each parameter and pass it to the method automatically—another form of dependency injection.

```
@Produces Set<Roles> getRoles(User user) {
   return user.getRoles();
}
```

We'll talk much more about producer methods in Producer methods.

2.2.4. Producer fields

A *producer field* is a simpler alternative to a producer method. A producer field is declared by annotating a field of a bean class with the `@Produces` annotation—the same annotation used for producer methods.

```
import javax.enterprise.inject.Produces;

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

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

A producer field is really just a shortcut that lets us avoid writing a useless getter method. However, in addition to convenience, producer fields serve a specific purpose as an adaptor for Java EE component environment injection, but to learn more about that, you'll have to wait until Java EE component environment resources. Because we can't wait to get to work on some examples.

Chapter 3. JSF web application example

Let's illustrate these ideas with a full example. We're going to implement 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, with constraints defined using annotations from the Bean Validation specification:

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

    @NotNull @Length(min=3, max=25)
    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    @NotNull @Length(min=6, max=20)
    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}">
      <f:validateBean>
         <h:outputLabel for="username">Username:</h:outputLabel>
         <h:inputText id="username" value="#{credentials.username}"/>
         <h:outputLabel for="password">Password:</h:outputLabel>
         <h:inputSecret id="password" value="#{credentials.password}"/>
      </f:validateBean>
   </h:panelGrid>
   <h:commandButton value="Login" action="#{login.login}" rendered="
#{!login.loggedIn}"/>
   <h:commandButton value="Logout" action="#{login.logout}" rendered=
"#{login.loggedIn}"/>
</h:form>
```

Users are represented by a JPA entity:

```
@Entity
public class User {
   private @NotNull @Length(min=3, max=25) @Id String username;
   private @NotNull @Length(min=6, max=20) String password;

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

(Note that we’re also going to need a `persistence.xml` file to configure the JPA persistence unit containing `User`.)

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 implements Serializable {

   @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 adaptor bean to expose our typesafe `EntityManager`:

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

Now `DocumentEditor`, or any other bean, can easily inject the current user:

```
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 gave you a taste of the CDI programming model. In the next chapter, we'll explore dependency injection in greater depth.

Chapter 4. Dependency injection and programmatic lookup

One of the most significant features of CDI—certainly the most recognized—is dependency injection; excuse me, *typesafe* dependency injection.

4.1. Injection points

The `@Inject` annotation lets us define an injection point that is injected during bean instantiation. Injection can occur via three different mechanisms.

Bean constructor parameter injection:

```
public class Checkout {

   private final ShoppingCart cart;

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

}
```

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;
   }

}
```

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:

```java
public class Checkout {

   private @Inject ShoppingCart cart;

}
```

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. Simplifying just a little, things happen in this order:

- 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, if any, is called.

(The only complication is that the container might call initializer methods declared by a superclass before initializing injected fields declared by a subclass.)

One major advantage of constructor injection is that it allows the bean to be immutable.

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

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

This is a case where the `@Inject` annotation *is not* required at the injection point. The same is true for observer methods (which we'll meet in Events) and disposer methods.

4.2. What gets injected

The CDI specification defines a procedure, called *typesafe resolution*, 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.

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 a *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
- allow the beans to be isolated into separate modules.

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 more advanced cases. We'll start by taking a closer look at qualifiers.

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 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 selected at runtime based on the state of the system:

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

If an injected field or a parameter of a bean constructor or initializer method is not explicitly annotated with a qualifier, the default qualifier,`@Default`, is assumed.

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. It does not create a direct dependency to any particular implementation. There may be multiple alternative implementations of `@Asynchronous PaymentProcessor`!

4.4. The built-in qualifiers `@Default` and `@Any`

Whenever a bean or injection point does not explicitly declare a qualifier, the container assumes the qualifier `@Default`. From time to time, you'll need to declare an injection point without specifying a qualifier. There's a qualifier for that too. All beans have the qualifier`        @Any`. Therefore, by explicitly specifying `@Any` at an injection point, you suppress the default qualifier, without otherwise restricting the beans that are eligible for injection.

This is especially useful if you want to iterate over all beans with a certain bean type. For example:

```
import javax.enterprise.inject.Instance;

...

@Inject
void initServices(@Any Instance<Service> services) {
   for (Service service: services) {
      service.init();
   }
}
```

4.5. Qualifiers with members

Java annotations can have members. We can use annotation members to further discriminate a qualifier. This prevents a potential explosion of new annotations. For example, instead of creating several qualifiers representing different payment methods, we could aggregate them into a single annotation with a member:

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

Then we select one of the possible member values when applying the qualifier:

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

We can force 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() default "";
}
```

4.6. Multiple qualifiers

An injection point may specify multiple qualifiers:

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

Then only a bean which has *both* qualifier annotations would be eligible for injection.

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

4.7. Alternatives

Alternatives are beans whose implementation is specific to a particular client module or deployment scenario. This alternative defines a mock implementation of both `@Synchronous PaymentProcessor` and `@Asynchronous PaymentProcessor`, all in one:

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

By default, `@Alternative` beans are disabled. We need to *enable* an alternative in the `beans.xml` descriptor of a bean archive to make it available for instantiation and injection. However, this activation only applies to the beans in that archive. From CDI 1.1 onwards the alternative can be enabled for the whole application using `@Priority` annotation.

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
   <alternatives>
         <class>org.mycompany.mock.MockPaymentProcessor</class>
   </alternatives>
</beans>
```

When an ambiguous dependency exists at an injection point, the container attempts to resolve the ambiguity by looking for an enabled alternative among the beans that could be injected. If there is exactly one enabled alternative, that's the bean that will be injected. If there are more beans with priority, the one with the highest priority value is selected.

4.8. Fixing unsatisfied and ambiguous 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 abort deployment, informing us of the unsatisfied or ambiguous dependency.

During the course of your development, you're going to encounter this situation. Let's learn how to resolve it.

To fix an *unsatisfied dependency*, either:

- create a bean which implements the bean type and has all the qualifier types of the injection point,
- make sure that the bean you already have is in the classpath of the module with the injection point, or
- explicitly enable an `@Alternative` bean that implements the bean type and has the appropriate qualifier types, using `beans.xml`.
- enable an `@Alternative` bean that implements the bean type and has the appropriate qualifier types, using `@Priority` annotation.

To fix an *ambiguous dependency*, either:

- introduce a qualifier to distinguish between the two implementations of the bean type,
- exclude one of the beans from discovery (either by means of @Vetoed or `beans.xml`),
- disable one of the beans by annotating it `@Alternative`,
- move one of the implementations to a module that is not in the classpath of the module with the injection point, or

- disable one of two `@Alternative` beans that are trying to occupy the same space, using `beans.xml`,
- change priority value of one of two `@Alternative` beans with the `@Priority` if they have the same highest priority value.

Just remember: "There can be only one."

On the other hand, if you really do have an optional or multivalued injection point, you should change the type of your injection point to `Instance`, as we'll see in Obtaining a contextual instance by programmatic lookup.

Now there's one more issue you need to be aware of when using the dependency injection service.

4.9. Client proxies

Clients of an injected bean do not usually hold a direct reference to a bean instance, unless the bean is a dependent object (scope `@Dependent`).

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.

Unfortunately, due to limitations of the Java language, some Java types cannot be proxied by the container. If an injection point declared with one of these types resolves to a bean with any scope other than `@Dependent`, the container will abort deployment, informing us of the problem.

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 unproxyable dependency problem. If an injection point of type `X` results in an unproxyable dependency, simply:

- add a constructor with no parameters to `X`,
- change the type of the injection point to`Instance<X>`,
- introduce an interface `Y`, implemented by the injected bean, and change the type of the injection

point to `Y`, or

- if all else fails, change the scope of the injected bean to `@Dependent`.

4.10. 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 of `Instance` produces a contextual instance of the bean.

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

Qualifiers can be specified in one of two ways:

- by annotating the `Instance` injection point, or
- by passing qualifiers to the `select()` of `Event`.

Specifying the qualifiers at the injection point is much, much easier:

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

Now, the `PaymentProcessor` returned by `get()` will have the qualifier `@Asynchronous`.

Alternatively, we can specify the qualifier dynamically. First, we add the `@Any` qualifier to the injection point, to suppress the default qualifier. (All beans have the qualifier `@Any` .)

```
import javax.enterprise.inject.Instance;

...

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

Next, we need to obtain an instance of our qualifier type. Since annotations are interfaces, we can't just write `new Asynchronous()`. It's also quite tedious to create a concrete implementation of an annotation type from scratch. Instead, CDI lets us obtain a qualifier instance by subclassing the helper class `AnnotationLiteral`.

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

In some cases, we can use an anonymous class:

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

However, we can't use an anonymous class to implement a qualifier type with members.

Now, finally, we can pass the qualifier to the `select()` method of `Instance`.

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

4.11. The `InjectionPoint` object

There are certain kinds of dependent objects (beans with scope `@Dependent`) that need to know something about the object or injection point into which they are injected in order to be able to do what they do. For example:

- The log category for a `Logger` depends upon the class of the object that owns it.
- Injection of a HTTP parameter or header value depends upon what parameter or header name was specified at the injection point.
- Injection of the result of an EL expression evaluation depends upon the expression that was specified at the injection point.

A bean with scope `@Dependent` may inject an instance of `InjectionPoint` and access metadata relating to the injection point to which it belongs.

Let's look at an example. The following code is verbose, and vulnerable to refactoring problems:

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

This clever little producer method lets you inject a JDK `Logger` without explicitly specifying the log category:

```
import javax.enterprise.inject.spi.InjectionPoint;
import javax.enterprise.inject.Produces;

class LogFactory {

   @Produces Logger createLogger(InjectionPoint injectionPoint) {
      return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName(
));
   }

}
```

We can now write:

```
@Inject Logger log;
```

Not convinced? Then here's a second example. To inject HTTP parameters, we need to define a qualifier type:

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface HttpParam {
   @Nonbinding public String value();
}
```

We would use this qualifier type at injection points as follows:

```
@HttpParam("username") @Inject String username;
@HttpParam("password") @Inject String password;
```

The following producer method does the work:

```
import javax.enterprise.inject.Produces;
import javax.enterprise.inject.spi.InjectionPoint;

class HttpParams

   @Produces @HttpParam("")
   String getParamValue(InjectionPoint ip) {
      ServletRequest request = (ServletRequest) FacesContext.getCurrentInstance()
.getExternalContext().getRequest();
      return request.getParameter(ip.getAnnotated().getAnnotation(HttpParam.class)
.value());
   }

}
```

Note that acquiring of the request in this example is JSF-centric. For a more generic solution you could write your own producer for the request and have it injected as a method parameter.

Note also that the `value()` member of the `HttpParam` annotation is ignored by the container since it is annotated `@Nonbinding.`

The container provides a built-in bean that implements the `InjectionPoint` interface:

```
public interface InjectionPoint {
   public Type getType();
   public Set<Annotation> getQualifiers();
   public Bean<?> getBean();
   public Member getMember();
   public Annotated getAnnotated();
   public boolean isDelegate();
   public boolean isTransient();
}
```

Chapter 5. 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.

JPA entities aren't a great fit for this model. Entities have their whole own lifecycle and identity model which just doesn't map naturally to the model used in CDI. Therefore, we recommend against treating entities as CDI beans. You're certainly going to run into problems if you try to give an entity a scope other than the default scope `@Dependent`. The client proxy will get in the way if you try to pass an injected instance to the JPA `EntityManager`.

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.

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. Furthermore, since CDI 1.1 the conversation context is active during every servlet request.

The request and application scopes are also active:

- during invocations of EJB remote methods,
- during invocations of EJB asynchronous methods,
- during EJB timeouts,
- during message delivery to a message-driven bean,
- during web service invocations, and
- during `@PostConstruct` callback of any bean

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.

Managed beans with scope `@SessionScoped` or `@ConversationScoped` must be serializable, since the container passivates the HTTP session from time to time.

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 web application (browsers tend to share domain cookies, and hence the session cookie, between tabs, so this is not the case for the session scope).

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 servlet request (since CDI 1.1). 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 lifecycle of conversations in a CDI 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:

```
import javax.enterprise.inject.Produces;
import javax.inject.Inject;
import javax.persistence.PersistenceContextType.EXTENDED;

@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) or redirect. 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 `javax.enterprise.context.conversation`.

Therefore, the following link propagates the conversation:

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

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="#{javax.enterprise.context.conversation.id}"/>
</h:link>
```

The conversation context propagates across redirects, making it very easy to implement the common POST-then-redirect pattern, without resort to fragile constructs such as a "flash" object. The container automatically adds the conversation id to the redirect URL as a request parameter.

In certain scenarios it may be desired to suppress propagation of a long-running conversation. The `conversationPropagation` request parameter (introduced in CDI 1.1) may be used for this purpose. If the `conversationPropagation` request parameter has the value `none` , the container will not reassociate the existing conversation but will instead associate the request with a new transient conversation even though the conversation id was propagated.

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 conserve resources. A CDI implementation will normally do this on the basis of some kind of timeout—though this is not required by the 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);
```

Another option how to set conversation timeout is to provide configuration property defining the new time value. See [config-conversation-timeout]. However note that any conversation might be destroyed any time sooner when HTTP session invalidation or timeout occurs.

5.3.4. CDI Conversation filter

The conversation management is not always smooth. For example, if the propagated conversation cannot be restored, the `javax.enterprise.context.NonexistentConversationException` is thrown. Or if there are concurrent requests for a one long-running conversation, \`javax.enterprise.context.BusyConversationException \` is thrown. For such cases, developer has no opportunity to deal with the exception by default, as the conversation associated with a Servlet request is determined at the beginning of the request before calling any service() method of any servlet in the web application, even before calling any of the filters in the web application and before the container calls any ServletRequestListener or AsyncListener in the web application.

To be allowed to handle the exceptions, a filter defined in the CDI 1.1 with the name ` CDI Conversation Filter ` can be used. By mapping the ` CDI Conversation Filter ` in the web.xml just after some other filters, we are able to catch the exceptions in them since the ordering in the web.xml specifies the ordering in which the filters will be called (described in the servlet specification).

In the following example, a filter MyFilter checks for the BusyConversationException thrown during the conversation association. In the web.xml example, the filter is mapped before the CDI Conversation Filter.

```
public class MyFilter implements Filter {
...

@Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
      throws IOException, ServletException {
        try {
            chain.doFilter(request, response);
        } catch (BusyConversationException e) {
            response.setContentType("text/plain");
            response.getWriter().print("BusyConversationException");
        }
    }

...
```

To make it work, we need to map our MyFilter before the CDI Conversation Filter in the web.xml file.

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

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

The mapping of the `CDI Conversation Filter` determines when CDI reads the `cid` request parameter. This process forces request body parsing. If your application relies on setting a custom character encoding for the request or parsing the request body itself by reading an `InputStream` or `Reader`, make sure that this is performed in a filter that executes before the CDI Conversation Filter is executed.

5.4. The singleton pseudo-scope

In addition to the four built-in scopes, CDI also supports two *pseudo-scopes*. The first is the *singleton pseudo-scope*, which we specify using the annotation `@Singleton`.

> Unlike the other scopes, which belong to the package `javax.enterprise.context`, the `@Singleton` annotation is defined in the package `javax.inject`.

You can guess what "singleton" means here. It means a bean that is instantiated once. Unfortunately, there's a little problem with this pseudo-scope. Beans with scope `@Singleton` don't have a proxy object. Clients hold a direct reference to the singleton instance. So we need to consider the case of a client that can be serialized, for example, any bean with scope `@SessionScoped` or `@ConversationScoped`, any dependent object of a bean with scope `@SessionScoped` or `@ConversationScoped`, or any stateful session bean.

Now, if the singleton instance is a simple, immutable, serializable object like a string, a number or a date, we probably don't mind too much if it gets duplicated via serialization. However, that makes it no stop being a true singleton, and we may as well have just declared it with the default scope.

There are several ways to ensure that the singleton bean remains a singleton when its client gets serialized:

- have the singleton bean implement `writeResolve()` and `readReplace()` (as defined by the Java serialization specification),
- make sure the client keeps only a transient reference to the singleton bean, or
- give the client a reference of type `Instance<X>` where `X` is the bean type of the singleton bean.

A fourth, better solution is to instead use `@ApplicationScoped`, allowing the container to proxy the bean, and take care of serialization problems automatically.

5.5. The dependent pseudo-scope

Finally, 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`:

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

An instance of a dependent bean is never shared between different clients or different injection points. It is strictly a *dependent object* of some other object. It is instantiated when the object it belongs to is created, and destroyed when the object it belongs to is destroyed.

If a Unified EL expression refers to a dependent bean by EL name, an instance of the bean is instantiated every time the expression is evaluated. The instance is not reused during any other expression evaluation.

If you need to access a bean directly by EL name in a JSF page, you probably need to give it a scope other than `@Dependent`. Otherwise, any value that gets set to the bean by a JSF input will be lost immediately. That's why CDI features the `@Model` stereotype; it lets you give a bean a name, and set its scope to `@RequestScoped` in one stroke. If you need to access a bean that really *has* to have the scope `@Dependent` from a JSF page, inject it into a different bean, and expose it to EL via a getter method.

Beans with scope `@Dependent` don't need a proxy object. The client holds a direct reference to its instance.

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.

5.6. The `@New` qualifier

The built-in qualifier `@New` allows us to obtain a dependent object of a specified class.

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

The class must be a valid managed bean or session bean, but need not be an enabled bean.

This works 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 Producer methods.

The `@New` qualifier was deprecated in CDI 1.1. CDI applications are encouraged to inject @Dependent scoped beans instead.

Loose coupling with strong typing

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 interface, nor is it required to manage the lifecycle of the implementation. 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.

The second major theme of CDI 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.

Chapter 6. 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, 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, 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:

```java
import javax.enterprise.inject.Produces;

@SessionScoped
public class Preferences implements Serializable {
   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.

A producer method does *not* inherit the scope of the bean that declares the method. There are two different beans here: the producer method, and the bean which declares it. The scope of the producer method determines how often the method will be called, and the lifecycle of the objects returned by the method. The scope of the bean that declares the producer method determines the lifecycle of the object upon which the producer method is invoked.

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.

The @New qualifier was deprecated in CDI 1.1. CDI applications are encouraged to inject @Dependent scoped beans instead.

6.4. Disposer methods

Some producer methods return objects that require explicit destruction. For example, somebody needs to close this JDBC connection:

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

Destruction can be performed by a matching *disposer method*, defined by the same class as the producer method:

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

The disposer method must have at least one parameter, annotated `@Disposes`, with the same type and qualifiers as the producer method. The disposer method is called automatically when the context ends (in this case, at the end of the request), and this parameter receives the object produced by the producer method. If the disposer method has additional method parameters, the container will look for a bean that satisfies the type and qualifiers of each parameter and pass it to the method automatically.

Since CDI 1.1 disposer methods may be used for destroying not only objects produced by producer methods but also objects producer by *producer fields*.

Chapter 7. Interceptors

Interceptor functionality is defined in the Java Interceptors specification.

The Interceptors specification defines three kinds of interception points:

- business method interception,
- lifecycle callback interception, and
- timeout method interception (EJB only).

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) throws Exception { ... }
}
```

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.

A *timeout method interceptor* applies to invocations of EJB timeout methods by the container:

```
public class TimeoutInterceptor {
   @AroundTimeout
   public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

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 type* 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) throws Exception { ... }
}
```

Interceptors can take advantage of dependency injection:

```
@Transactional @Interceptor
public class TransactionInterceptor {

    @Resource UserTransaction transaction;

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

}
```

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. We can do it using `beans.xml` descriptor of a bean archive. However, this activation only applies to the beans in that archive. From CDI 1.1 onwards the interceptor can be enabled for the whole application using `@Priority` annotation.

```xml
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
   <interceptors>
      <class>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans>
```

Whoah! Why the angle bracket stew?

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

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

Having two interceptors without `@Priority`, we could specify that our security interceptor runs before our transaction interceptor.

```xml
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.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 not mentioning them in `beans.xml`! Ah, so simple.

It gets quite tricky when used along with interceptors annotated with `@Priority`. Interceptors enabled using `@Priority` are called before interceptors enabled using `beans.xml`, the lower priority values are called first.

Having an interceptor enabled by `@Priority` and in the same time listed in `beans.xml` leads to a non-portable behaviour! This combination of enablement should therefore be avoided in order to maintain consistent behaviour across different CDI implementations.

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) throws Exception { ... }
}
```

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() { ... }
}
```

```
@Transactional
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 Interceptors specification (and used by the Managed Beans 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.

Chapter 8. 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. Interceptors and decorators, though similar in many ways, are complementary. 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 {
   ...
}
```

The decorator implements the methods of the decorated type that it wants to intercept.

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

   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      ...
   }

   public void deposit(BigDecimal amount);
      ...
   }
}
```

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.

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

8.1. Delegate object

Decorators have a special injection point, called the *delegate injection point*, with the same type as the beans they decorate, and the annotation `@Delegate`. There must be exactly one delegate injection point, which can be a constructor parameter, initializer method parameter or injected field.

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

A decorator is bound to any bean which:

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

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. The decorator will only be bound to beans with the same qualifiers.

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

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.

```
@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) );
      }
   }
}
```

8.2. Enabling decorators

By default, all decorators are disabled. We need to *enable* our decorator. We can do it using `beans.xml` descriptor of a bean archive. However, this activation only applies to the beans in that archive. From CDI 1.1 onwards the decorator can be enabled for the whole application using `@Priority` annotation.

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.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 an ordering for decorators in our system, ensuring deterministic behavior, and
- it lets us enable or disable decorator classes at deployment time.

Decorators enabled using `@Priority` are called before decorators enabled using `beans.xml`, the lower priority values are called first.

Having a decorator enabled by `@Priority` and in the same time listed in `beans.xml` leads to a non-portable behaviour! This combination of enablement should therefore be avoided in order to maintain consistent behaviour across different CDI implementations.

Chapter 9. Events

Dependency injection enables loose-coupling by allowing the implementation of the injected bean type to vary, either at deployment time or runtime. Events go one step further, allowing beans to interact with no compile time dependency at all. Event *producers* raise events that are delivered to event *observers* by the container.

This basic schema might sound like the familiar observer/observable pattern, but there are a couple of twists:

- not only are event producers decoupled from observers; observers are completely decoupled from producers,
- observers can specify a combination of "selectors" to narrow the set of event notifications 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.

The CDI event notification facility uses more or less the same typesafe approach that we've already seen with the dependency injection service.

9.1. Event payload

The event object carries state from producer to consumer. The event object is nothing more than an instance of a concrete Java class. (The only restriction is that an event type may not contain type variables). An event may be assigned qualifiers, which allows observers to distinguish it from other events of the same type. The qualifiers function like topic selectors, allowing an observer to narrow the set of events it observes.

An event qualifier is just a normal qualifier, defined using `@Qualifier`. Here's an example:

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

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`. The event parameter may also specify qualifiers.

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

An observer method need not specify any event qualifiers—in this case it is interested in every event whose type is assignable to the observed event type. Such observer will trigger on both events shown below:

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

If the observer does specify qualifiers, it will be notified of an event if the event object is assignable to the observed event type, and if the set of observed event qualifiers is a subset of all the event qualifiers of the event.

The observer method may have additional parameters, which are injection points:

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

9.3. Event producers

Event producers fire events using an instance of the parameterized `Event` interface. An instance of this interface is obtained by injection:

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

A producer raises events by calling the `fire()` method of the `Event` interface, passing the 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 (the `Document`) is assignable, and
- specifies no qualifiers.

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 in one of two ways:

- by annotating the `Event` injection point, or
- by passing qualifiers to the `select()` of `Event`.

Specifying the qualifiers at the injection point is far simpler:

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

Then, every event fired via this instance of `Event` has the event qualifier `@Updated`. The event is 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 specified at the `Event` injection point.

The downside of annotating the injection point is that we can't specify the qualifier dynamically. CDI lets us obtain a qualifier instance by subclassing the helper class `AnnotationLiteral`. That way, we can pass the qualifier to the `select()` method of `Event`.

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

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

9.4. Conditional observer methods

By default, if there is no instance of an observer in the current context, the container will instantiate the observer in order to deliver an event to it. This behavior isn't always desirable. We may want to deliver events only to instances of the observer that already exist in the current contexts.

A conditional observer is specified by adding `receive = IF_EXISTS` to the `@Observes` annotation.

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

A bean with scope `@Dependent` cannot be a conditional observer, since it would never be called!

9.5. Event qualifiers with members

An event qualifier type may have annotation members:

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

Event qualifiers may be combined, for example:

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

An observer method is only notified if all the observed qualifiers are specified when the event is fired. Assume the following observers in this example:

```
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) { ... }}}
```

All of these observer methods will be notified.

However, if there were also an observer method:

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

It would not be notified, as `@Personal` is not a qualifier of the event being fired.

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_PROGRESS` 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:

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@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:

```
import javax.enterprise.event.Event;

@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:

```java
import javax.ejb.Singleton;

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

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

Chapter 10. 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 developer 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 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.

A stereotype is an annotation, annotated `@Stereotype`, that packages several other annotations. 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 apply some other annotations to our stereotype or else it wouldn't be adding much value.

10.1. Default scope for a stereotype

A stereotype may specify a default scope for beans annotated with the stereotype. For example:

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

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 for that bean. 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, the `LoginAction` bean will have the defaulted name `loginAction`.

10.4. Alternative stereotypes

A stereotype can indicate that all beans to which it is applied are `@Alternative`s. An *alternative stereotype* lets us classify beans by deployment scenario.

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

We can apply an alternative stereotype to a whole set of beans, and activate them all with one line of code in `beans.xml`.

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

```
<beans>
   <alternatives>
      <stereotype>org.mycompany.testing.Mock</stereotype>
   </alternatives>
</beans>
```

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

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

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

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

Chapter 11. 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 occurrences 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 unsatisfied 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 alternatives and then show the guarantees that specialization adds.

11.1. Using alternative stereotypes

CDI lets you *override* the implementation of a bean type at deployment time using an alternative. For example, the following bean provides a default implementation of the `PaymentProcessor` interface:

```
public class DefaultPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

But in our staging environment, we don't really want to submit payments to the external system, so we override that implementation of `PaymentProcessor` with a different bean:

```
public @Alternative
class StagingPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

or

```
public @Alternative
class StagingPaymentProcessor
      extends DefaultPaymentProcessor {
   ...
}
```

We've already seen how we can enable this alternative by listing its class in the `beans.xml` descriptor.

But suppose we have many alternatives in the staging environment. It would be much more convenient to be able to enable them all at once. So let's 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:

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

Then we replace the `@Alternative` annotation on our bean with `@Staging`:

```
@Staging
public class StagingPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Finally, we activate the `@Staging` stereotype in the `beans.xml` descriptor:

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.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.

11.2. A minor problem with alternatives

When we enable an alternative, 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.

```
@Inject @LargeTransaction PaymentProcessor paymentProcessor;
```

So we haven't completely replaced the default implementation in this 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.

11.3. 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:

```
@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.

So why does specialization work, and what does it have to do with inheritance?

Since we’re informing the container that our alternative bean is meant to stand in as a replacement for the default implementation, the alternative implementation automatically inherits all qualifiers of the default implementation. Thus, in our example, `MockCreditCardPaymentProcessor` inherits the qualifiers `@Default` and `@CreditCard`.

Furthermore, if the default implementation declares a bean EL name using `@Named`, the name is inherited by the specializing alternative bean.

Chapter 12. Java EE component environment resources

Java EE 5 already introduced some limited support for dependency injection, in the form of component environment injection. A component environment resource is a Java EE component, for example a JDBC datasource, JMS queue or topic, JPA persistence context, remote EJB or web service.

Naturally, there is now a slight mismatch with the new style of dependency injection in CDI. Most notably, component environment injection relies on string-based names to qualify ambiguous types, and there is no real consistency as to the nature of the names (sometimes a JNDI name, sometimes a persistence unit name, sometimes an EJB link, sometimes a non-portable "mapped name"). Producer fields turned out to be an elegant adaptor to reduce all this complexity to a common model and get component environment resources to participate in the CDI system just like any other kind of bean.

Fields have a duality in that they can both be the target of Java EE component environment injection and be declared as a CDI producer field. Therefore, they can define a mapping from a string-based name in the component environment, to a combination of type and qualifiers used in the world of typesafe injection. We call a producer field that represents a reference to an object in the Java EE component environment a *resource*.

12.1. Defining a resource

The CDI specification uses the term *resource* to refer, generically, to any of the following kinds of object which might be available in the Java EE component environment:

- JDBC `Datasource`s, JMS `Queue`s, `Topic`s and `ConnectionFactory`s, JavaMail `Session`s and other transactional resources including JCA connectors,
- JPA `EntityManager`s and `EntityManagerFactory`s,
- remote EJBs, and
- web services.

We declare a resource by annotating a producer field with a component environment injection annotation: `@Resource`, `@EJB`, `@PersistenceContext`, `@PersistenceUnit` or `@WebServiceRef`.

```
@Produces @WebServiceRef(lookup="java:app/service/Catalog")
Catalog catalog;
```

```
@Produces @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
@CustomerDatabase Datasource customerDatabase;
```

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

```
@Produces @PersistenceUnit(unitName="CustomerDatabase")
@CustomerDatabase EntityManagerFactory customerDatabasePersistenceUnit;
```

```
@Produces @EJB(ejbLink="../their.jar#PaymentService")
PaymentService paymentService;
```

The field may be static (but not final).

A resource declaration really contains two pieces of information:

- the JNDI name, EJB link, persistence unit name, or other metadata needed to obtain a reference to the resource from the component environment, and
- the type and qualifiers that we will use to inject the reference into our beans.

It might feel strange to be declaring resources in Java code. Isn't this stuff that might be deployment-specific? Certainly, and that's why it makes sense to declare your resources in a class annotated `@Alternative`.

12.2. Typesafe resource injection

These resources can now be injected in the usual way.

```
@Inject Catalog catalog;
```

```
@Inject @CustomerDatabase Datasource customerDatabase;
```

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

```
@Inject @CustomerDatabase EntityManagerFactory customerDatabaseEntityManagerFactory;
```

```
@Inject PaymentService paymentService;
```

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

It might seem like a pain to have to write these extra producer field declarations, just to gain an additional level of indirection. You could just as well use component environment injection directly, right? But remember that you're going to be using resources like the `EntityManager` in several different beans. Isn't it nicer and more typesafe to write

```
@Inject @CustomerDatabase EntityManager
```

instead of

```
@PersistenceContext(unitName="CustomerDatabase") EntityManager
```

all over the place?

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 de facto 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 DeltaSpike.

Chapter 13. 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.

13.1. Built-in beans

In the Java EE environment, the container provides the following built-in beans, all with the qualifier `@Default`:

- the current JTA `UserTransaction`,
- a `Principal` representing the current caller identity,
- the default Bean Validation `ValidationFactory`,
- a `Validator` for the default `ValidationFactory`,
- `HttpServletRequest`, `HttpSession` and `ServletContext`

The `FacesContext` is not injectable. You can get at it by calling `FacesContext.getCurrentInstance()`. Alternatively you may define the following producer method:

```
import javax.enterprise.inject.Produces;

class FacesContextProducer {
   @Produces @RequestScoped FacesContext getFacesContext() {
      return FacesContext.getCurrentInstance();
   }
}
```

13.2. Injecting Java EE resources into a bean

All managed beans may take advantage of Java EE component environment injection using `@Resource`, `@EJB`, `@PersistenceContext`, `@PersistenceUnit` and `@WebServiceRef`. 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 UserTransaction transaction;

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

```
@SessionScoped
public class Login implements Serializable {
   @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.

Of course, we advise that component environment injection be used to define CDI resources, and that typesafe injection be used in application code.

13.3. Calling a bean from a servlet

It's easy to use a bean from a servlet in Java EE. Simply inject the bean using field or initializer method injection.

```
public class LoginServlet 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.

13.4. Calling a bean from a message-driven bean

CDI injection applies to all EJBs, even when they aren't CDI beans. In particular, you can use CDI injection in message-driven beans, which are by nature not contextual objects.

You can even use interceptor bindings for message-driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
   @Inject Inventory inventory;
   @PersistenceContext EntityManager em;

   public void onMessage(Message message) {
      ...
   }
}
```

Please note 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.

But how about beans which *send* JMS messages?

13.5. 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:

```
import javax.jms.ConnectionFactory;
import javax.jms.Queue;

public class OrderResources {
   @Resource(name="jms/ConnectionFactory")
   private ConnectionFactory connectionFactory;

   @Resource(name="jms/OrderQueue")
   private Queue orderQueue;

   @Produces @Order
   public Connection createOrderConnection() throws JMSException {
    return connectionFactory.createConnection();
   }

   public void closeOrderConnection(@Disposes @Order Connection connection)
         throws JMSException {
      connection.close();
   }

   @Produces @Order
   public Session createOrderSession(@Order Connection connection)
         throws JMSException {
      return connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
   }

   public void closeOrderSession(@Disposes @Order Session session)
         throws JMSException {
      session.close();
   }

   @Produces @Order
   public MessageProducer createOrderMessageProducer(@Order Session session)
         throws JMSException {
      return session.createProducer(orderQueue);
   }

   public void closeOrderMessageProducer(@Disposes @Order MessageProducer producer)
         throws JMSException {
      producer.close();
   }
}
```

In this example, we can just inject the prepared `MessageProducer`, `Connection` or `QueueSession`:

```
@Inject Order order;
@Inject @Order MessageProducer producer;
@Inject @Order Session 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.

13.6. Packaging and deployment

CDI doesn't define any special deployment archive. You can package CDI beans in JARs, EJB JARs or WARs—any deployment location in the application classpath. However, the archive must be a "bean archive".

Unlike CDI 1.0, the CDI 1.1 specification recognizes two types of bean archives. The type determines the way the container discovers CDI beans in the archive.

CDI 1.1 makes use of a new XSD file for beans.xml descriptor: http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd

13.6.1. Explicit bean archive

An explicit bean archive is an archive which contains a `beans.xml` file:

- with a version number of 1.1 (or later), with the bean-discovery-mode of `all`, or,
- like in CDI 1.0 – with no version number, or, that is an empty file.

It behaves just like a CDI 1.0 bean archive – i.e. CDI discovers each Java class, interface or enum in such an archive.

The `beans.xml` file must be located at:

- `META-INF/beans.xml` (for jar archives), or,
- `WEB-INF/beans.xml` or `WEB-INF/classes/META-INF/beans.xml` (for WAR archives).

You should never place a `beans.xml` file in both of the WEB-INF and the WEB-INF/classes/META-INF directories. Otherwise your application would not be portable.

13.6.2. Implicit bean archive

An implicit bean archive is an archive which contains one or more bean classes with a *bean* *defining annotation*, or one or more session beans. It can also contain a `beans.xml` file with a version number of 1.1 (or later), with the bean-discovery-mode of `annotated`. CDI only discovers Java classes with a bean defining annotation within an implicit bean archive.

Any scope type is a bean defining annotation. If you place a scope type on a bean class, then it has a bean defining annotation. See 2.5. Bean defining annotations to learn more.

13.6.3. What archive is not a bean archive

Although quite obvious, let's sum it up:

- an archive which contains neither a `beans.xml` file nor any bean class with a *bean defining annotation*,
- an archive which contains a `beans.xml` file with the bean-discovery-mode of `none`.

For compatibility with CDI 1.0, each Java EE product (WildFly, GlassFish, etc.) must contain an option to cause an archive to be ignored by the container when no `beans.xml` is present. Consult specific Java EE product documentation to learn more about such option.

13.6.4. Embeddable EJB container

In an embeddable EJB container, beans may be deployed in any location in which EJBs may be deployed.

Chapter 14. 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

14.1. Creating an `Extension`

The first step in creating a portable extension is to write a class that implements `Extension`. This marker interface does not define any methods, but it's needed to satisfy the requirements of Java SE's service provider architecture.

```
 import javax.enterprise.inject.spi.Extension;

class MyExtension implements Extension { ... }
```

Next, we need to register our extension as a service provider by creating a file named `META-INF/services/javax.enterprise.inject.spi.Extension`, which contains the name of our extension class:

```
org.mydomain.extension.MyExtension
```

An extension is not a bean, exactly, since it is instantiated by the container during the initialization process, before any beans or contexts exist. However, it can be injected into other beans once the initialization process is complete.

```
@Inject
MyBean(MyExtension myExtension) {
   myExtension.doSomething();
}
```

And, like beans, extensions can have observer methods. Usually, the observer methods observe *container lifecycle events*.

14.2. Container lifecycle events

During the initialization process, the container fires a series of events, including:

- `BeforeBeanDiscovery`
- `ProcessAnnotatedType` and `ProcessSyntheticAnnotatedType`
- `AfterTypeDiscovery`
- `ProcessInjectionTarget` and `ProcessProducer`
- `ProcessInjectionPoint`
- `ProcessBeanAttributes`
- `ProcessBean`, `ProcessManagedBean`, `ProcessSessionBean`, `ProcessProducerMethod` and `ProcessProducerField`
- `ProcessObserverMethod`
- `AfterBeanDiscovery`
- `AfterDeploymentValidation`

Extensions may observe these events:

```
import javax.enterprise.inject.spi.Extension;

class MyExtension implements Extension {

   void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
      Logger.global.debug("beginning the scanning process");
   }

   <T> void processAnnotatedType(@Observes ProcessAnnotatedType<T> pat) {
      Logger.global.debug("scanning type: " + pat.getAnnotatedType().getJavaClass()
.getName());
   }

   void afterBeanDiscovery(@Observes AfterBeanDiscovery abd) {
      Logger.global.debug("finished the scanning process");
   }

}
```

In fact, the extension can do a lot more than just observe. The extension is permitted to modify the container's metamodel and more. Here's a very simple example:

```
import javax.enterprise.inject.spi.Extension;

class MyExtension implements Extension {

   <T> void processAnnotatedType(@Observes @WithAnnotations({Ignore.class})
ProcessAnnotatedType<T> pat) {
      /* tell the container to ignore the type if it is annotated @Ignore */
      if ( pat.getAnnotatedType().isAnnotationPresent(Ignore.class) ) pat.veto();
   }

}
```

The `@WithAnnotations` annotation causes the container to deliver the ProcessAnnotatedType events only for the types which contain the specified annotation.

The observer method may inject a `BeanManager`

```
<T> void processAnnotatedType(@Observes ProcessAnnotatedType<T> pat, BeanManager
beanManager) { ... }
```

An extension observer method is not allowed to inject any other object.

14.3. The `BeanManager` object

The nerve center for extending CDI is the `BeanManager` object. The `BeanManager` interface lets us obtain beans, interceptors, decorators, observers and contexts programmatically.

```
public interface BeanManager {
   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... qualifiers);
   public Set<Bean<?>> getBeans(String name);
   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... qualifiers);
   public <T> Set<ObserverMethod<? super T>> resolveObserverMethods(T event,
Annotation... qualifiers);
   public List<Decorator<?>> resolveDecorators(Set<Type> types, Annotation...
qualifiers);
   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 isInterceptorBinding(Class<? extends Annotation> annotationType);
    public boolean isStereotype(Class<? extends Annotation> annotationType);
    public Set<Annotation> getInterceptorBindingDefinition(Class<? extends Annotation>
bindingType);
    public Set<Annotation> getStereotypeDefinition(Class<? extends Annotation>
stereotype);
    public boolean areQualifiersEquivalent(Annotation qualifier1, Annotation
qualifier2);
    public boolean areInterceptorBindingsEquivalent(Annotation interceptorBinding1,
Annotation interceptorBinding2);
    public int getQualifierHashCode(Annotation qualifier);
    public int getInterceptorBindingHashCode(Annotation interceptorBinding);
    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);
    public <T> InjectionTargetFactory<T> getInjectionTargetFactory(AnnotatedType<T>
annotatedType);
    public <X> ProducerFactory<X> getProducerFactory(AnnotatedField<? super X> field,
Bean<X> declaringBean);
    public <X> ProducerFactory<X> getProducerFactory(AnnotatedMethod<? super X> method,
Bean<X> declaringBean);
    public <T> BeanAttributes<T> createBeanAttributes(AnnotatedType<T> type);
    public BeanAttributes<?> createBeanAttributes(AnnotatedMember<?> type);
    public <T> Bean<T> createBean(BeanAttributes<T> attributes, Class<T> beanClass,
    public <T, X> Bean<T> createBean(BeanAttributes<T> attributes, Class<X> beanClass,
ProducerFactory<X> producerFactory);
    public InjectionPoint createInjectionPoint(AnnotatedField<?> field);
    public InjectionPoint createInjectionPoint(AnnotatedParameter<?> parameter);
    public <T extends Extension> T getExtension(Class<T> extensionClass);
}
```

Any bean or other Java EE component which supports injection can obtain an instance of `BeanManager` via injection:

```
@Inject BeanManager beanManager;
```

Alternatively, a `BeanManager` reference may be obtained from CDI via a static method call.

```
CDI.current().getBeanManager()
```

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`.

14.4. The `CDI` class

Application components which cannot obtain a `BeanManager` reference via injection nor JNDI lookup can get the reference from the `javax.enterprise.inject.spi.CDI` class via a static method call:

```
BeanManager manager = CDI.current().getBeanManager();
```

The `CDI` class can be used directly to programmatically lookup CDI beans as described in Obtaining a contextual instance by programmatic lookup

```
CDI.select(Foo.class).get()
```

14.5. The `InjectionTarget` interface

The first thing that a framework developer is going to look for in the portable extension SPI is a way to inject CDI beans into objects which are not under the control of CDI. The `InjectionTarget` interface makes this very easy.

> We recommend that frameworks let CDI take over the job of actually instantiating the framework-controlled objects. That way, the framework-controlled objects can take advantage of constructor injection. However, if the framework requires use of a constructor with a special signature, the framework will need to instantiate the object itself, and so only method and field injection will be supported.

```
import javax.enterprise.inject.spi.CDI;

...

//get the BeanManager
BeanManager beanManager = CDI.current().getBeanManager();

//CDI uses an AnnotatedType object to read the annotations of a class
AnnotatedType<SomeFrameworkComponent> type = beanManager.createAnnotatedType
(SomeFrameworkComponent.class);

//The extension uses an InjectionTarget to delegate instantiation, dependency
injection
//and lifecycle callbacks to the CDI container
InjectionTarget<SomeFrameworkComponent> it = beanManager.createInjectionTarget(type);

//each instance needs its own CDI CreationalContext
CreationalContext ctx = beanManager.createCreationalContext(null);

//instantiate the framework component and inject its dependencies
SomeFrameworkComponent instance = it.produce(ctx);  //call the constructor
it.inject(instance, ctx);  //call initializer methods and perform field injection
it.postConstruct(instance);  //call the @PostConstruct method

...

//destroy the framework component instance and clean up dependent objects
it.preDestroy(instance);  //call the @PreDestroy method
it.dispose(instance);  //it is now safe to discard the instance
ctx.release();  //clean up dependent objects
```

14.6. The `Bean` interface

Instances of the interface `Bean` represent beans. There is an instance of `Bean` registered with the `BeanManager` object for every bean in the application. There are even `Bean` objects representing interceptors, decorators and producer methods.

The `BeanAttributes` interface exposes all the interesting things we discussed in The anatomy of a bean.

```
public interface BeanAttributes<T> {
   public Set<Type> getTypes();
   public Set<Annotation> getQualifiers();
   public Class<? extends Annotation> getScope();
   public String getName();
   public Set<Class<? extends Annotation>> getStereotypes();
   public boolean isAlternative();
}
```

The `Bean` interface extends the `BeanAttributes` interface and defines everything the container needs to manage instances of a certain bean.

```
public interface Bean<T> extends Contextual<T>, BeanAttributes<T> {
   public Class<?> getBeanClass();
   public Set<InjectionPoint> getInjectionPoints();
   public boolean isNullable();
}
```

There's an easy way to find out what beans exist in the application:

```
Set<Bean<?>> allBeans = beanManager.getBeans(Obect.class, new AnnotationLiteral<Any>()
{});
```

The `Bean` interface makes it possible for a portable extension 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.

14.7. Registering a `Bean`

The most common kind of CDI portable extension registers a bean (or beans) with the container.

In this example, we make a framework class, `SecurityManager` available for injection. To make things a bit more interesting, we're going to delegate back to the container's `InjectionTarget` to perform instantiation and injection upon the `SecurityManager` instance.

```
import javax.enterprise.inject.spi.Extension;
import javax.enterprise.event.Observes;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import javax.enterprise.inject.spi.InjectionPoint;
...

public class SecurityManagerExtension implements Extension {

    void afterBeanDiscovery(@Observes AfterBeanDiscovery abd, BeanManager bm) {
        //use this to read annotations of the class
        AnnotatedType<SecurityManager> at = bm.createAnnotatedType(SecurityManager
.class);

        //use this to instantiate the class and inject dependencies
        final InjectionTarget<SecurityManager> it = bm.createInjectionTarget(at);

        abd.addBean( new Bean<SecurityManager>() {

            @Override
            public Class<?> getBeanClass() {
                return SecurityManager.class;
            }

            @Override
            public Set<InjectionPoint> getInjectionPoints() {
                return it.getInjectionPoints();
            }

            @Override
            public String getName() {
                return "securityManager";
            }

            @Override
            public Set<Annotation> getQualifiers() {
                Set<Annotation> qualifiers = new HashSet<Annotation>();
                qualifiers.add( new AnnotationLiteral<Default>() {} );
                qualifiers.add( new AnnotationLiteral<Any>() {} );
                return qualifiers;
            }

            @Override
            public Class<? extends Annotation> getScope() {
                return ApplicationScoped.class;
            }

            @Override
            public Set<Class<? extends Annotation>> getStereotypes() {
                return Collections.emptySet();
            }

            @Override
            public Set<Type> getTypes() {
                Set<Type> types = new HashSet<Type>();
                types.add(SecurityManager.class);
                types.add(Object.class);
                return types;
            }

            @Override
            public boolean isAlternative() {
                return false;
            }

            @Override
            public boolean isNullable() {
                return false;
            }

            @Override
            public SecurityManager create(CreationalContext<SecurityManager> ctx) {
                SecurityManager instance = it.produce(ctx);
                it.inject(instance, ctx);
                it.postConstruct(instance);
                return instance;
            }

            @Override
            public void destroy(SecurityManager instance,
                                CreationalContext<SecurityManager> ctx) {
                it.preDestroy(instance);
                it.dispose(instance);
                ctx.release();
            }

        } );
    }

}
```

But a portable extension can also mess with beans that are discovered automatically by the container.

14.8. Wrapping an `AnnotatedType`

One of the most interesting things that an extension class can do is process the annotations of a bean class *before* the container builds its metamodel.

Let's start with an example of an extension that provides support for the use of `@Named` at the package level. The package-level name is used to qualify the EL names of all beans defined in that package. The portable extension uses the `ProcessAnnotatedType` event to wrap the `AnnotatedType` object and override the `value()` of the `@Named` annotation.

```
import java.lang.reflect.Type;
import javax.enterprise.inject.spi.Extension;
import java.lang.annotation.Annotation;
...

public class QualifiedNameExtension implements Extension {
    <X> void processAnnotatedType(@Observes ProcessAnnotatedType<X> pat) {

        /* wrap this to override the annotations of the class */
        final AnnotatedType<X> at = pat.getAnnotatedType();

        /* Only wrap AnnotatedTypes for classes with @Named packages */
        Package pkg = at.getJavaClass().getPackage();
        if ( !pkg.isAnnotationPresent(Named.class) ) {
            return;
        }

        AnnotatedType<X> wrapped = new AnnotatedType<X>() {

            class NamedLiteral extends AnnotationLiteral<Named>
            implements Named {
                @Override
                public String value() {
                    Package pkg = at.getJavaClass().getPackage();

                    String unqualifiedName = "";
                    if (at.isAnnotationPresent(Named.class)) {
                        unqualifiedName = at.getAnnotation(Named.class).value();
                    }

                    if (unqualifiedName.isEmpty()) {
                        unqualifiedName = Introspector.decapitalize(at.getJavaClass()
.getSimpleName());
                    }

                    final String qualifiedName;
                    if ( pkg.isAnnotationPresent(Named.class) ) {
                        qualifiedName = pkg.getAnnotation(Named.class).value()
                            + '.' + unqualifiedName;
                    }
                    else {
                        qualifiedName = unqualifiedName;
                    }

                    return qualifiedName;
                }
            }

            private final NamedLiteral namedLiteral = new NamedLiteral();

            @Override
            public Set<AnnotatedConstructor<X>> getConstructors() {
                return at.getConstructors();
            }

            @Override
            public Set<AnnotatedField<? super X>> getFields() {
                return at.getFields();
            }

            @Override
            public Class<X> getJavaClass() {
                return at.getJavaClass();
            }

            @Override
            public Set<AnnotatedMethod<? super X>> getMethods() {
                return at.getMethods();
            }

            @Override
            public <T extends Annotation> T getAnnotation(final Class<T> annType) {
                if (Named.class.equals(annType)) {
                    return (T) namedLiteral;
                }
                else {
                    return at.getAnnotation(annType);
                }
            }

            @Override
            public Set<Annotation> getAnnotations() {
                Set<Annotation> original = at.getAnnotations();
                Set<Annotation> annotations = new HashSet<Annotation>();

                boolean hasNamed = false;

                for (Annotation annotation : original) {
                    if (annotation.annotationType().equals(Named.class)) {
                        annotations.add(getAnnotation(Named.class));
                        hasNamed = true;
                    }
                    else {
                        annotations.add(annotation);
                    }
                }

                if (!hasNamed) {
                    Package pkg = at.getJavaClass().getPackage();
                    if (pkg.isAnnotationPresent(Named.class)) {
                        annotations.add(getAnnotation(Named.class));
                    }
                }

                return annotations;
            }
            @Override
            public Type getBaseType() {
                return at.getBaseType();
            }

            @Override
            public Set<Type> getTypeClosure() {
                return at.getTypeClosure();
            }

            @Override
            public boolean isAnnotationPresent(Class<? extends Annotation> annType) {
                if (Named.class.equals(annType)) {
                    return true;
                }
                return at.isAnnotationPresent(annType);
            }

        };

        pat.setAnnotatedType(wrapped);
    }

}
```

Here's a second example, which adds the `@Alternative` annotation to any class which implements a certain `Service` interface.

```
import javax.enterprise.inject.spi.Extension;
import java.lang.annotation.Annotation;
...

class ServiceAlternativeExtension implements Extension {

   <T extends Service> void processAnnotatedType(@Observes ProcessAnnotatedType<T>
pat) {

      final AnnotatedType<T> type = pat.getAnnotatedType();

      /* if the class implements Service, make it an @Alternative */
      AnnotatedType<T> wrapped = new AnnotatedType<T>() {

         class AlternativeLiteral extends AnnotationLiteral<Alternative> implements
Alternative {}

         private final AlternativeLiteral alternativeLiteral = new AlternativeLiteral
();

         @Override
         public <X extends Annotation> X getAnnotation(final Class<X> annType) {
            return (X) (annType.equals(Alternative.class) ?  alternativeLiteral :
type.getAnnotation(annType));
         }

         @Override
         public Set<Annotation> getAnnotations() {
            Set<Annotation> annotations = new HashSet<Annotation>(type.getAnnotations
());
            annotations.add(alternativeLiteral);
            return annotations;
         }

         @Override
         public boolean isAnnotationPresent(Class<? extends Annotation>
annotationType) {
            return annotationType.equals(Alternative.class) ?
               true : type.isAnnotationPresent(annotationType);
         }

         /* remaining methods of AnnotatedType */
         ...
      }

      pat.setAnnotatedType(wrapped);
   }
}
```

The AnnotatedType is not the only thing that can be wrapped by an extension.

14.9. Overriding attributes of a bean by wrapping `BeanAttributes`

Wrapping an `AnnotatedType` is a low-level approach to overriding CDI metadata by adding, removing or replacing annotations. Since version 1.1, CDI provides a higher-level facility for overriding attributes of beans discovered by the CDI container.

```
public interface BeanAttributes<T> {

   public Set<Type> getTypes();

   public Set<Annotation> getQualifiers();

   public Class<? extends Annotation> getScope();

   public String getName();

   public Set<Class<? extends Annotation>> getStereotypes();

   public boolean isAlternative();

}
```

The `BeanAttributes` interface exposes attributes of a bean. The container fires a `ProcessBeanAttributes` event for each enabled bean, interceptor and decorator before this object is registered. Similarly to the `ProcessAnnotatedType`, this event allows an extension to modify attributes of a bean or to veto the bean entirely.

```
public interface ProcessBeanAttributes<T> {

    public Annotated getAnnotated();

    public BeanAttributes<T> getBeanAttributes();

    public void setBeanAttributes(BeanAttributes<T> beanAttributes);

    public void addDefinitionError(Throwable t);

    public void veto();

}
```

The `BeanManager` provides two utility methods for creating the `BeanAttributes` object from scratch:

```
public <T> BeanAttributes<T> createBeanAttributes(AnnotatedType<T> type);

public BeanAttributes<?> createBeanAttributes(AnnotatedMember<?> type);
```

14.10. Wrapping an `InjectionTarget`

The `InjectionTarget` interface exposes operations for producing and disposing an instance of a component, injecting its dependencies and invoking its lifecycle callbacks. A portable extension may wrap the `InjectionTarget` for any Java EE component that supports injection, allowing it to intercept any of these operations when they are invoked by the container.

Here's a CDI portable extension that reads values from properties files and configures fields of Java EE components, including servlets, EJBs, managed beans, interceptors and more. In this example, properties for a class such as `org.mydomain.blog.Blogger` go in a resource named `org/mydomain/blog/Blogger.properties`, and the name of a property must match the name of the field to be configured. So `Blogger.properties` could contain:

```
firstName=Gavin
lastName=King
```

The portable extension works by wrapping the containers `InjectionTarget` and setting field values from the `inject()` method.

```
import javax.enterprise.event.Observes;
import javax.enterprise.inject.spi.Extension;
import javax.enterprise.inject.spi.InjectionPoint;

public class ConfigExtension implements Extension {

    <X> void processInjectionTarget(@Observes ProcessInjectionTarget<X> pit) {

          /* wrap this to intercept the component lifecycle */
         final InjectionTarget<X> it = pit.getInjectionTarget();

        final Map<Field, Object> configuredValues = new HashMap<Field, Object>();

        /* use this to read annotations of the class and its members */
        AnnotatedType<X> at = pit.getAnnotatedType();

        /* read the properties file */
        String propsFileName = at.getJavaClass().getSimpleName() + ".properties";
        InputStream stream = at.getJavaClass().getResourceAsStream(propsFileName);
        if (stream!=null) {

            try {
                Properties props = new Properties();
                props.load(stream);
                for (Map.Entry<Object, Object> property : props.entrySet()) {
                    String fieldName = property.getKey().toString();
                    Object value = property.getValue();
                    try {
                        Field field = at.getJavaClass().getDeclaredField(fieldName);
                        field.setAccessible(true);
                        if ( field.getType().isAssignableFrom( value.getClass() ) ) {
                            configuredValues.put(field, value);
                        }
                        else {
                            /* TODO: do type conversion automatically */
                            pit.addDefinitionError( new InjectionException(
                                   "field is not of type String: " + field ) );
                        }
                    }
                    catch (NoSuchFieldException nsfe) {
                        pit.addDefinitionError(nsfe);
                    }
                    finally {
                        stream.close();
                    }
                }
            }
            catch (IOException ioe) {
                pit.addDefinitionError(ioe);
            }
        }

        InjectionTarget<X> wrapped = new InjectionTarget<X>() {

            @Override
            public void inject(X instance, CreationalContext<X> ctx) {
                it.inject(instance, ctx);

                /* set the values onto the new instance of the component */
                for (Map.Entry<Field, Object> configuredValue: configuredValues
.entrySet()) {
                    try {
                        configuredValue.getKey().set(instance, configuredValue
.getValue());
                    }
                    catch (Exception e) {
                        throw new InjectionException(e);
                    }
                }
            }

            @Override
            public void postConstruct(X instance) {
                it.postConstruct(instance);
            }
            @Override
            public void preDestroy(X instance) {
                it.dispose(instance);
            }

            @Override
            public void dispose(X instance) {
                it.dispose(instance);
            }

            @Override
            public Set<InjectionPoint> getInjectionPoints() {
                return it.getInjectionPoints();
            }

            @Override
            public X produce(CreationalContext<X> ctx) {
                return it.produce(ctx);
            }

        };

        pit.setInjectionTarget(wrapped);

    }

}
```

14.11. Overriding `InjectionPoint`

CDI provides a way to override the metadata of an `InjectionPoint`. This works similarly to how metadata of a bean may be overridden using `BeanAttributes`.

For every injection point of each component supporting injection CDI fires an event of type `javax.enterprise.inject.spi.ProcessInjectionPoint`

```
public interface ProcessInjectionPoint<T, X> {
    public InjectionPoint getInjectionPoint();
    public void setInjectionPoint(InjectionPoint injectionPoint);
    public void addDefinitionError(Throwable t);
}
```

An extension may either completely override the injection point metadata or alter it by wrapping the `InjectionPoint` object obtained from `ProcessInjectionPoint.getInjectionPoint()`

There's a lot more to the portable extension SPI than what we've discussed here. Check out the CDI spec or Javadoc for more information. For now, we'll just mention one more extension point.

14.12. Manipulating interceptors, decorators and alternatives enabled for an application

An event of type `javax.enterprise.inject.spi.AfterTypeDiscovery` is fired when the container has fully completed the type discovery process and before it begins the bean discovery process.

```
public interface AfterTypeDiscovery {
    public List<Class<?>> getAlternatives();
    public List<Class<?>> getInterceptors();
    public List<Class<?>> getDecorators();
    public void addAnnotatedType(AnnotatedType<?> type, String id);
}
```

This event exposes a list of enabled alternatives, interceptors and decorators. Extensions may manipulate these collections directly to add, remove or change the order of the enabled records.

In addition, an `AnnotatedType` can be added to the types which will be scanned during bean discovery, with an identifier, which allows multiple annotated types, based on the same underlying type, to be defined.

14.13. The `Context` and `AlterableContext` interfaces

The `Context` and `AlterableContext` interface support 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.

```
import javax.enterprise.context.spi.Context;

public interface AlterableContext extends Context {
    public void destroy(Contextual<?> contextual);
}
```

`AlterableContext` was introduced in CDI 1.1. The `destroy` method allows an application to remove instances of contextual objects from a context.

For more information on implementing a custom context see this blog post.

Chapter 15. Next steps

A lot of additional information on CDI can be found online. Regardless, the CDI specification remains the authority for information on CDI. The spec is less than 100 pages and is quite readable (don’t worry, it’s not like your Blu-ray player manual). Of course, it covers many details we’ve skipped over here. The spec is available on the download page of the CDI Website.

The cdi-spec.org website is probably your main entry point to learn more about CDI. You’ll find information about the spec, link to frameworks and libs supporting CDI and resource to go further on learning CDI.

The source of this guide is available on github Feel free to fill ticket or send PR if you want to make this document better.