Table of Contents

Beans
Introduction
- What is a bean?
- Getting our feet wet

More about beans
- The anatomy of a bean
  - Bean types, qualifiers and dependency injection
  - Scope
  - EL name
  - Alternatives
  - Interceptor binding types
- What kinds of classes are beans?
  - Managed beans
  - Session beans
  - Producer methods
  - Producer fields

JSF web application example

Dependency injection and programmatic lookup
- Injection points
- What gets injected
- Qualifier annotations
- The built-in qualifiers `@Default` and `@Any`
- Qualifiers with members
- Multiple qualifiers
- Alternatives
- Fixing unsatisfied and ambiguous dependencies
- Client proxies
- Obtaining a contextual instance by programmatic lookup
  - Enhanced version of `jakarta.enterprise.inject.Instance`
- The `InjectionPoint` object

Scopes and contexts
- Scope types
- Built-in scopes
- The conversation scope
  - Conversation demarcation
  - Conversation propagation
  - Conversation timeout
  - CDI Conversation filter
  - Lazy and eager conversation context initialization
- The singleton pseudo-scope
- The dependent pseudo-scope

Weld 4.0.1.Final - CDI Reference Implementation

Beans

The CDI (https://jakarta.ee/specifications/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 Jakarta EE platform and include full support for Jakarta EE modularity and the Jakarta EE component architecture. But the specification does not limit the use of CDI to the Jakarta EE environment. Starting with CDI 2.0, the specification covers the use of CDI in the Java SE environment as well. In Java SE, the services might be provided by a standalone CDI implementation like Weld (see CDI SE Module), or even by a container that also implements the subset of EJB defined for embedded usage by the EJB 3.2 specification. 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 `jakarta.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 Jakarta 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.

Weld 4.0.1.Final - CDI Reference Implementation

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!

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.

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:

JAVA

```
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:

JAVA

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

Where `Translator` is the EJB local interface:

JAVA

```
@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:

JAVA

```
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:

JAVA

```
@Named @RequestScoped
public class TranslateController {
   @Inject TextTranslator textTranslator; // (1)

   private String inputText;
   private String translation;

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

   public String getInputText() {
      return inputText;
   }

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

   public String getTranslation() {
      return translation;
   }
}
```

1. Field injection of `TextTranslator` instance

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

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

JAVA

```
import jakarta.enterprise.inject.Instance;
import jakarta.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.

Weld 4.0.1.Final - CDI Reference Implementation

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.

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.

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:

JAVA

```
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 {
   ...
}
```

NOTE

> 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.

NOTE

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`:

JAVA

```
@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:

JAVA

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

NOTE

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 .

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:

JAVA

```
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.

NOTE

Keep in mind that once a bean is bound to a context, it remains in that context until the context is destroyed. There is no way to 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 .

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:

JAVA

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

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

XML

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

NOTE

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

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

JAVA

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

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

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.

JAVA

```
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. The other way to enable an alternative is to annotate the bean with `@Priority` annotation. This will enable it globally.

We cover alternatives in more detail in Alternatives .

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! Furthermore, 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.

JAVA

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

The interceptor that implements transaction management declares this annotation:

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

JAVA

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 { ... }
```

JAVA

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. Better still, we can use `@Priority` annotation to enable the interceptor and define it's ordering at the same time.

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

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.

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 `jakarta.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` .

NOTE

> 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.

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.

NOTE

> 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".

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.

JAVA

```
import jakarta.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:

JAVA

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

Even in a Unified EL expression:

XML

```
<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.

> **NOTE** 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.

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

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

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.

JAVA
```
import jakarta.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.

Weld 4.0.1.Final - CDI Reference Implementation

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:

JAVA

```
@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:

XML

```
<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:

JAVA

```
@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:

JAVA

```
@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:

JAVA

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

JAVA

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

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

JAVA

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

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

JAVA

```
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:

XML

```
<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.

Weld 4.0.1.Final - CDI Reference Implementation

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.

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:

JAVA

```
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:

JAVA

```
public class Checkout {

   private ShoppingCart cart;

   @Inject
   void setShoppingCart(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

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

And direct field injection:

JAVA

```
public class Checkout {

   private @Inject ShoppingCart cart;

}
```

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

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

> **NOTE** 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.

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.

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`:

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

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

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

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

```java
@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:

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

Using initializer method injection:

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

Using constructor injection:

```java
@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:

JAVA
```
@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`!

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:

JAVA
```
import jakarta.enterprise.inject.Instance;

...

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

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:

JAVA
```
@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:

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

We can force the container to ignore a member of a qualifier type by annotating the member `@Nonbinding`.

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

Multiple qualifiers

An injection point may specify multiple qualifiers:

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

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

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

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:

```java
@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.

```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">
   <alternatives>
         <class>org.mycompany.mock.MockPaymentProcessor</class>
   </alternatives>
</beans>
```

From CDI 1.1 onwards the alternative can be enabled for the whole application using `@Priority` annotation.

```java
@Priority(100) @Alternative @Synchronous @Asynchronous
public class MockPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

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.

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 (http://docs.jboss.org/cdi/api/1.2/javax/enterprise/inject/Vetoed.html) 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.

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`.

> **NOTE** Weld also supports a non-standard workaround for this limitation. See the Configuration chapter for more information.

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;
```
JAVA

The `get()` method of `Instance` produces a contextual instance of the bean.

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

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;
```
JAVA

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 jakarta.enterprise.inject.Instance;

...

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

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 {}
```
JAVA

In some cases, we can use an anonymous class:

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

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);
```
JAVA

NOTE

Since CDI 2.0, most annotations from `jakarta.enterprise` package have their `AnnotationLiteral` implementations. Therefore, in order to programmatically obtain (for instance) `@Any` annotation, you can simply do `Any.Literal.INSTANCE`.

Enhanced version of `jakarta.enterprise.inject.Instance`

Weld also provides `org.jboss.weld.inject.WeldInstance` - an enhanced version of `jakarta.enterprise.inject.Instance`. There are three additional methods. The first one - `getHandler()` - allows to obtain a contextual reference handler which not only holds the contextual reference but also allows to inspect the metadata of the relevant bean and to destroy the underlying contextual instance. Moreover, the handler implements `AutoCloseable`:

JAVA

```
import org.jboss.weld.inject.WeldInstance;

class Foo {

  @Inject
  WeldInstance<Bar> instance;

  void doWork() {
    try (Handler<Bar> barHandler = instance.getHandler()) {
        barHandler.get().doBusiness();
        // Note that Bar will be automatically destroyed at the end of the try-with-resources statement
    }
    Handler<Bar> barHandler = instance.getHandler()
    barHandler.get().doBusiness();
    // Calls Instance.destroy()
    barHandler.destroy();
  }

}
```

The next method - `handlers()` - returns an `Iterable` which allows to iterate over handlers for all the beans that have the required type and required qualifiers and are eligible for injection. This might be useful if you need more control inside the loop:

JAVA

```
@ApplicationScoped
class OrderService {

  @Inject
  @Any
  WeldInstance<OrderProcessor> instance;

  void create(Order order) {
    for (Handler<OrderProcessor> handler : instance.handlers()) {
            handler.get().process(order);
            if (Dependent.class.equals(handler.getBean().getScope()) {
                // Destroy only dependent processors
                handler.destroy();
            }
        }
    }
  }
```

Third method is a twist on the `select()` method, but it accepts `java.lang.reflect.Type` as parameter and optionally qualifier(s). This allows for generic selection of instances which can be handy while dealing with third party beans through extensions. However, in order to stay type-safe, this method has a limitation - it can only be invoked on `WeldInstance<Object>`. Invocation on any other type than `Object` will result in an `IllegalStateException`. Please note that the return value if such select will always be `WeldInstance<Object>` unless you specify it further using `<SomeType>` before invoking this `select()`. Let's look at actual code:

JAVA

```
class MyCustomExtension implements Extension {

  @Inject
  @Any
  WeldInstance<Object> instance;

  private Set<Type> allTypes = new HashSet<>();

  public void observe(@Observes ProcessBean<?> bean) {
    // gather all bean types, even those that we do not own
    allTypes.add(bean.getAnnotated().getBaseType());
  }

  public void doWorkWithBeans(@Observes AfterDeploymentValidation adv) {
    for (Type t : allTypes) {
      // now we can select based on Type once we are sure all beans are initialized
      instance.select(t).isResolvable() ? logValidBeanFound(t) : logInvalidBeanFound(t);
    }
  }
```

`WeldInstance` is automatically available in Weld SE and Weld Servlet where the Weld API is always on the class path. It is also available in Weld-powered EE containers. In this case, users would have to compile their application against the Weld API and exclude the Weld API artifact from the deployment (e.g. use `provided` scope in Maven).

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:

JAVA

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

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

JAVA

```
import jakarta.enterprise.inject.spi.InjectionPoint;
import jakarta.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 jakarta.enterprise.inject.Produces;
import jakarta.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();
}
```

Weld 4.0.1.Final - CDI Reference Implementation

Scopes and contexts

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

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

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

NOTE

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`.

Scope types

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

JAVA

```
@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:

JAVA

```
@ClusterScoped
public class SecondLevelCache { ... }
```

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

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.

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*.

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:

JAVA

```
@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:

JAVA

```
import jakarta.enterprise.inject.Produces;
import jakarta.inject.Inject;
import jakarta.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.

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 `jakarta.enterprise.context.conversation`.

Therefore, the following link propagates the conversation:

XML

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

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

XML

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

TIP

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.

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.

JAVA

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

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

CDI Conversation filter

The conversation management is not always smooth. For example, if the propagated conversation cannot be restored, the `jakarta.enterprise.context.NonexistentConversationException` is thrown. Or if there are concurrent requests for a one long-running conversation, `jakarta.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.

JAVA

```
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.

XML

```
<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>
```

TIP

> The mapping of the `CDI Conversation Filter` determines when Weld 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. See this FAQ page for details (http://weld.cdi-spec.org/documentation/#3). Alternatively, the lazy conversation context initialization (see below) may be used.

Lazy and eager conversation context initialization

Conversation context may be initialized lazily or eagerly.

When initialized lazily, the conversation context (no matter if transient or long-running) is only initialized when a `@ConversationScoped` bean is accessed for the first time. At that point, the `cid` parameter is read and the conversation is restored. The conversation context may not be initialized at all throughout the request processing if no conversation state is accessed. Note that if a problem occurs during this delayed initialization, the conversation state access (bean method invocation) may result in `BusyConversationException` or `NonexistentConversationException` being thrown.

When initialized eagerly, the conversation context is initialized at a predefined time. Either at the beginning of the request processing before any listener, filter or servlet is invoked or, if the `CDI Conversation Filter` is mapped, during execution of this filter.

Conversation context initialization mode may be configured using the `org.jboss.weld.context.conversation.lazy` init parameter.

```xml
<context-param>
   <param-name>org.jboss.weld.context.conversation.lazy</param-name>
   <param-value>true</param-value>
</context-param>
```

If the init parameter is not set, the following default behavior applies:

- If the `CDI Conversation Filter` is mapped, the conversation context is initialized eagerly within this filter
- Otherwise, the conversation context is initialized lazily

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`.

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

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.

NOTE

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.

Last updated 2021-03-19 14:04:52 +0100