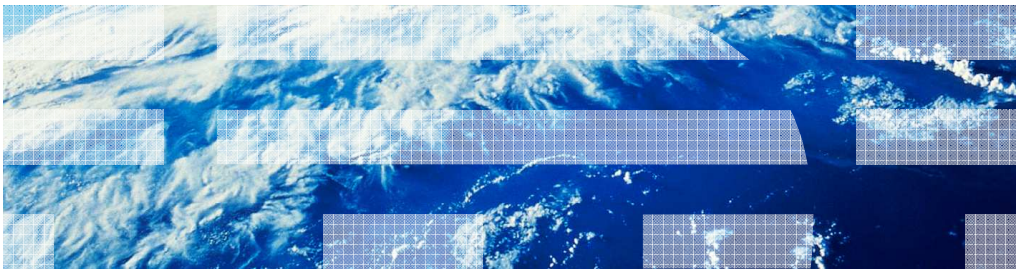
IBM WebSphere Application Server V8

Contexts and dependency injection (CDI) 1.0



WebSphere software

This presentation describes support for the Contexts and Dependency Injection (CDI) for Java EE included in IBM WebSphere Application Server V8.

Table of contents

This session will provide an overview of CDI, usage scenarios of CDI, and will end with a summary of CDI.

Section

Overview

This section contains an overview of CDI.

What is context and dependency injection?

- New spec (JSR 299) in EE6
- EE standard that was inspired by SEAM, Google Guice, Spring
- Uses annotations from JSR 330 (Dependency Injection for Java)
- Promotes loose coupling of components
- Provides extensible life cycle contexts that components are bound to
- Provides integration with the unified expression language

CDI is a new specification in Java EE6, it is JSR 299.

It is a standard that builds upon some other existing technologies such as SEAM, Google Guice, and Spring.

It uses the annotations from JSR 330 which is the standard Java Dependency injection specification.

It was designed to promote loose coupling and object oriented design of components.

Components are extensible, and wrapped inside a context or a life cycle and provide integration through the Unified Expression Language.

Section

Usage scenarios

Next this session will discuss some usage scenarios.

Each scenario will touch on whether the scenario is about defining a CDI bean or it is about how to access those beans, or if it is a service provided to a CDI bean.

The terms bean and component are used interchangeably.

Defining CDI beans

- Following a simple set of rules, any POJO might be treated as a CDI Bean
  - Opt-in by deploying class inside of a Bean Deployment Archive. A EAR/WAR/JAR with a beans.xml in WEB-INF/ or META-INF/ directory.
  - Must have a no-argument constructor or constructor annotated @Inject
  - Must be a concrete class (exception for @Decorator)
  - Must not be a non-static inner class

First, here is how to define a CDI bean or CDI component. Basically any POJO (Plain Old Java Object) can be treated as a CDI bean by following a very simple set of rules.

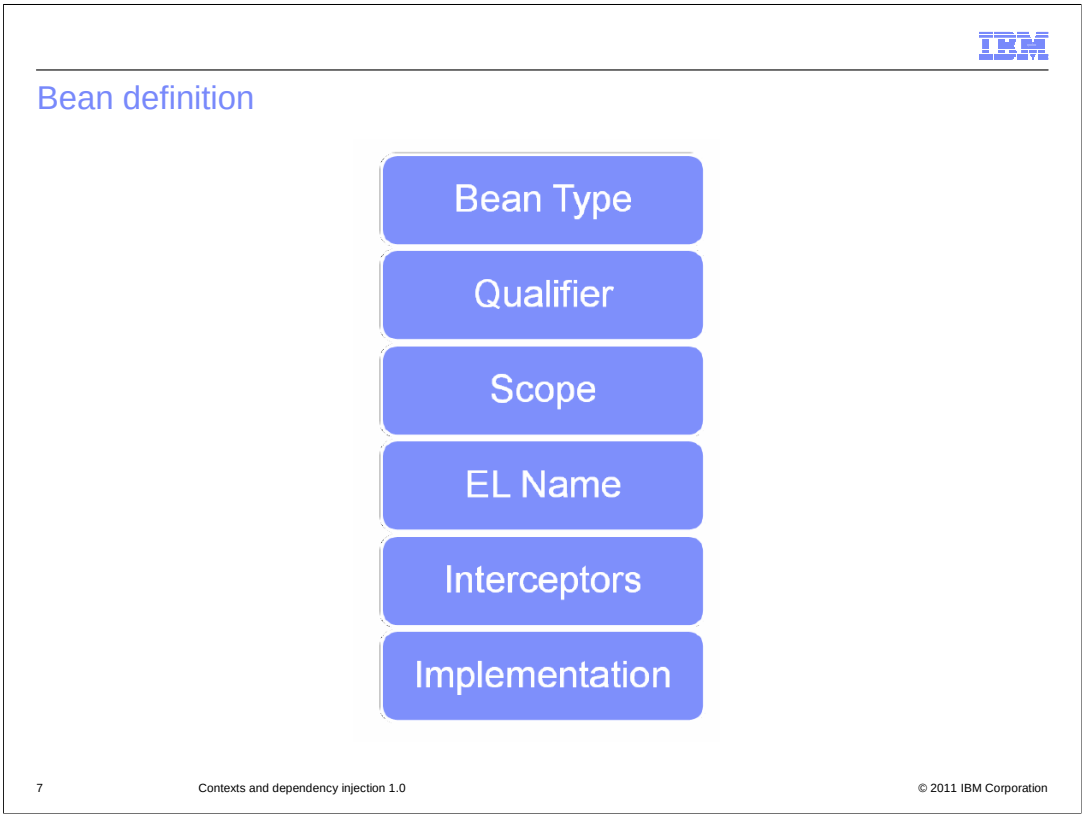
The first rule, and probably the most important one, is that you have to opt into this. You have to put a beans.xml file into a specific location in the archive that the POJO class is defined in. For WARs, that is the WEB-INF directory, and for EJB JARs or even utility JARs it is the META-INF directory. So once you have met that requirement, CDI knows where to go looking for these components.

There are also some specific Java rules.

It has to have a no-argument constructor, or a constructor that is specifically annotated with @Inject that tells the container which constructor to use when it is trying to build an instance of this object.

It has to be a concrete class, with one specific exception of Decorator's which are covered later.

If it is an inner class, it has to be a static inner class.



This slide shows what makes up a Bean Definition.

A Bean definition is made up of types, qualifiers, a scope, EL Name (optionally), Interceptors (optionally), and the concrete implementation class itself.

Bean types

- Set of Java types the Bean provides

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

- Client visible Bean types
  - BookShop, Business, Shop<Book>, Object.
- Can be restricted using the @Typed annotation
  - @Typed(Shop.class) public class BookShop

Bean types were the first box in the previous slide. Bean types are very simple, they are the set of Java types that a bean can provide. So in this code snippet you are looking at a BookShop class, it extends a Business class and implements a Shop<Book> interface. So the client visible types of this BookShop class are BookShop itself, Business, Shop<Book> and of course Object. This is basically everything in the class hierarchy of this class. That can often be quite a lot. So CDI also provides a way to restrict which bean types are defined for a class by using the @Typed annotation.

At the very bottom you will see another example that has @Typed (Shop.class) public class Bookshop. So in this case only Shop is exposed as a bean type for BookShop.

Qualifiers

- Way to distinguish between implementations of the same type

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

@ASynchronous
class AsynchronousPaymentProcessor
implements PaymentProcessor {
...
}
@Synchronous
class SynchronousPaymentProcessor
implements PaymentProcessor {
...
}
```


Qualifiers are additional metadata that you can associate with a bean because you might often have cases where multiple beans provide the same types. You are looking at three code snippets on this slide.

The first snippet is a definition of a qualifier. The qualifier is just another annotation, but with a specific annotation (see in bold) @Qualifier applied to it.

In the second and third snippet you will see two different classes AsynchronousPaymentProcessor and SychronousPaymentProcessor. Note how they both implement the PaymentProcessor type. If the CDI container were trying to pick between these at runtime, it will need some additional information. In this case, in the bottom snippet, the SychronousPaymentProcessor also has an @Sychronous qualifier attached to it, which you see defined at the top. You will also see @Asynchronous applied on the second snippet. These qualifiers enhance the type system and provide the additional information needed.

Life cycle management

- Components are annotated to describe their life cycle
- Examples of normal scopes @Request, @Session, @Application, etc
- Pseudo-scopes such a @Dependent also exist
- Container manages the creation and destruction of components and associated resources
- Client proxies are used by the container so the current instance of the bean is always used.

```
@RequestScoped
public class RequestScopedBean
{
}
```


Life cycle management falls under both categories. It is both an attribute of the CDI bean, and a service CDI provides. The attribute is described first.

Components use another annotation such as @RequestScoped, @SessionScoped or @ApplicationScoped that defines the life cycle of that component. At the very bottom of the slide you will see a code snippet of a RequestScoped bean with an annotation of @RequestScoped. This tells the CDI container how you expect this bean to be managed. There are other scopes that are not bound to a life cycle in quite the same way. For example there is a pseudo scope @Dependent which means that the bean lives as long as the bean it is injected into. So if it is injected into a request scoped bean, then it will stay around as long as that request scoped bean is active.

The service that CDI provides here is that, it manages the creation and destruction of these beans and all of the associated beans and resources injected into them. It does this through the use of client proxies. For example, if a request scoped bean is injected, you actually get a proxy that ensures that whenever you use a reference to that bean, it always retrieves the correct active request scoped bean from the request context.

Developers should be aware that you can have a longer living bean, for example a session scoped or an application scoped bean that wants to use a request scoped bean. You might not want to manage the life cycle of the request scoped bean. This means that you do not have to worry about if that is the current bean, or if it has it gone away because that request ended, instead the container will ensure you always have the right instance.

Dependency injection

- CDI supports injection into fields and methods
- Injection points are resolved using a combination of Type and Qualifiers

```
public class Order {
  @Inject @Selected Product product; //field injection
  private Quantity quantity;
  @Inject  //Initializer method
  void setQuantity(Quantity q) {
     this.quantity = q;
  }
…
```


Previous slides have covered how to define a CDI Bean, and a little bit about their attributes. You will now discover how to use them. The "DI" letters in "CDI" stand for Dependency Injection, which is also a service that CDI provides. On top of being able to inject CDI beans into other components, you can inject CDI beans into each other.

CDI supports basically two kinds of injection which are field and method injection.

In the example below, you can see a public class Order. On the very first line is a very simple @Inject annotation, which is the JSR 330 annotation for Injection. @Selected, which is a Qualifier, and then Product which is the Java type of the member.

CDI uses two pieces of information to decide how to fulfill a injection point. It uses the Type, which is Product, and the Qualifier which is @Selected. Then it looks for all the beans, hopefully it only finds one, that meet those two requirements.

Then below you can see that there is a method setQuantity also annotated with @Inject. In this case the Quantity Q is the injected parameter to that method. CDI is going to look for a bean with type Quantity to inject.

There is a also a third type that falls under method injection, which is constructor injection which is the same concept but in a constructor rather than a setter method.

Unified Expression Language (EL) Name

- Expose CDI beans through the Unified EL to JSF or JSP
- Specified using the @Named annotation

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

- Easily use the bean in any JSF or JSP page

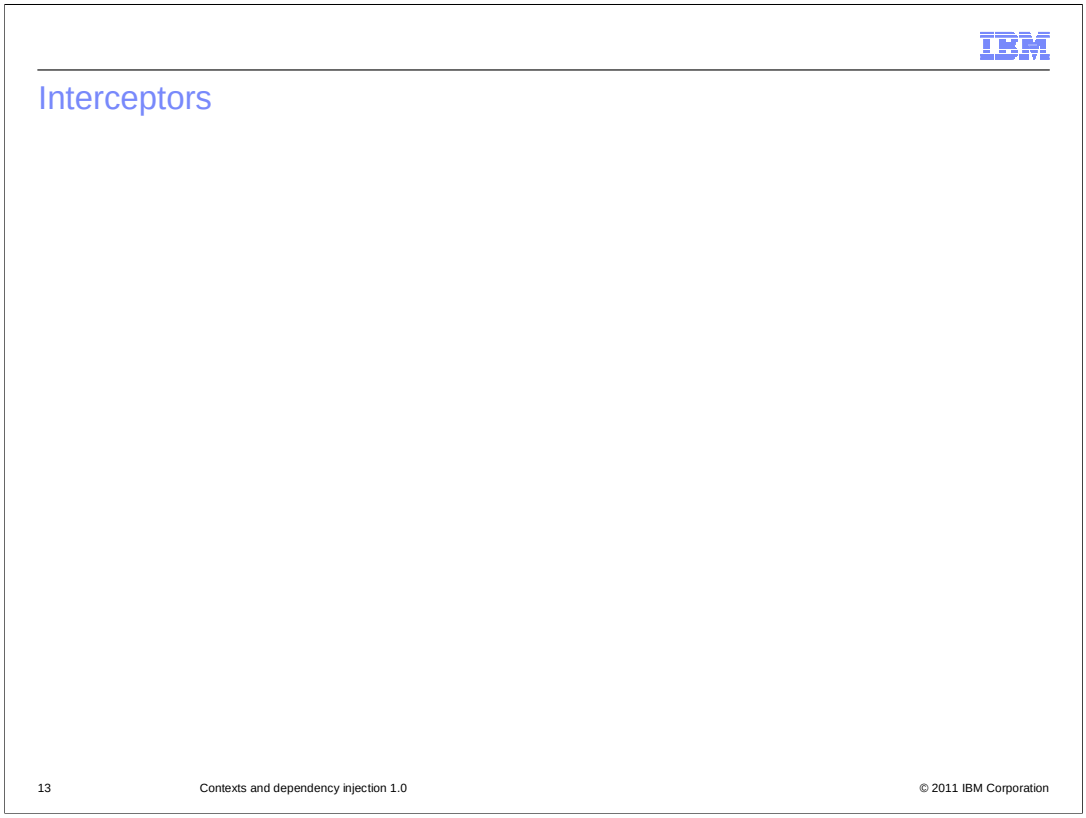
```
<h:dataTable value="#{cart.lineItems}" var="item">
...
</h:dataTable>
```

- Container derives default name in absence of @Named
- Unqualified short class name of the bean class
- @Named is a built-in Qualifier

Another way to access these CDI beans is through the Unified Expression Language (EL). CDI allows you to add an annotation, which is @Named, to the bean that will cause the bean to be exposed through the EL.

The @Named annotation provides an attribute which is the EL name of the bean. For example, in the ShoppingCart case it provides an attribute of “cart“. If no attribute is provided then it will default to the short class name of the bean class. In the case of ShoppingCart it is lower case s shopping capital C Cart.

Now you can access those beans directly from JSF or JSP. There is a snippet here of a JSF dataTable directly referencing cart.



Continuing next with the services CDI provides. One of them is Interceptors, which you might be familiar with from the EJB or the Interceptor specifications. The idea of interceptors is to separate cross cutting concerns from the business logic. Common examples of this are things like logging, security, or some service that does not line up well with your Object types.

The first step to using Interceptors in CDI is to define an InterceptorBinding. The first snippet is defining a new annotation called Transactional. In bold you can see the @InterceptorBinding annotation on this Transaction annotation definition. That tells CDI that @Transactional is an Interceptor Binding.

In the second snippet the @Transactional annotation is applied to the Interceptor definition. This class having both @Transactional and @Interceptor tells CDI that an Interceptor is being defined and associated with the @TransactionInterceptor binding. You can see the Interceptor has a method annotated with @AroundInvoke. This method controls the code flow of the Interceptor.

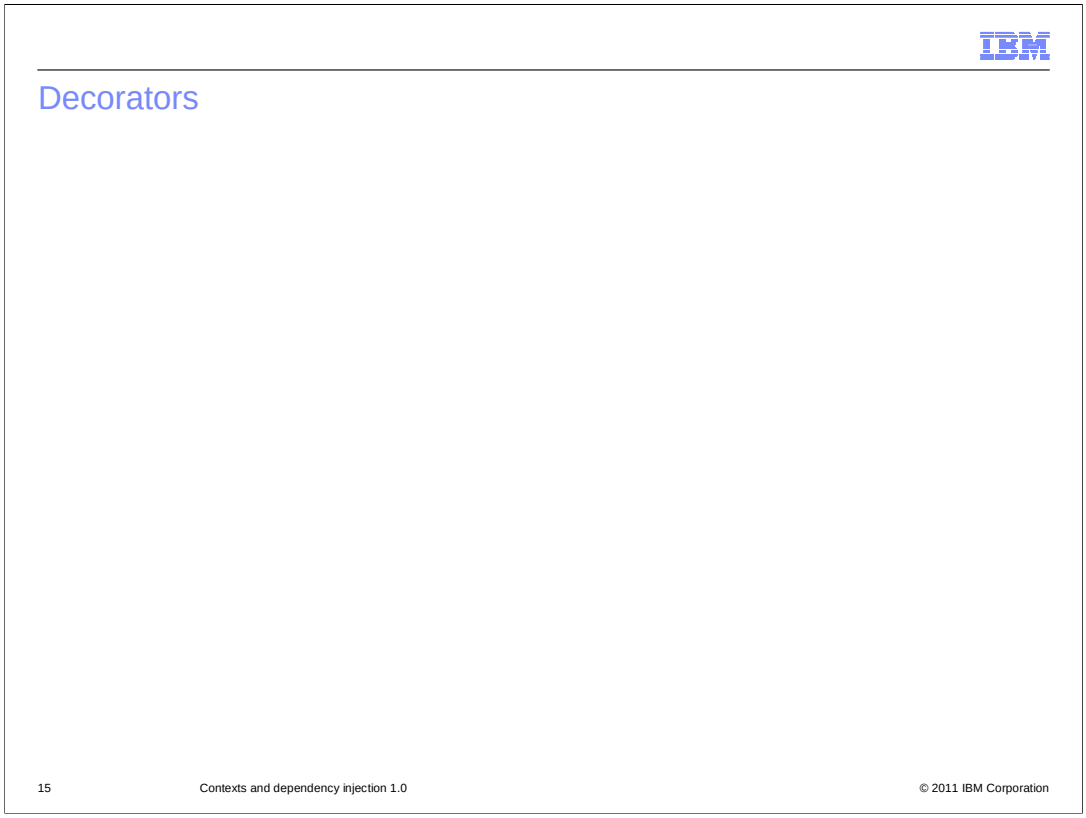
Finally at the bottom you see an example of the ShoppingCart bean from previous slides. The Interceptor binding is applied to both the class and the method. In most cases you will probably only do one or the other, depending on if you want the interceptor to be run for every method in the class, or specific methods.

Interceptors - continued

- Enabled manually in the beans.xml
- Order defined in beans.xml
- @Dependent object of the object it intercepts

```
<beans>
<interceptors>
<class>org.mycompany.TransactionInterceptor</class>
<class>org.mycompany.LoggingInterceptor</class>
</interceptors>
</beans>
```


Interceptors are a little bit more than just the binding and the definition. They also have to be enabled in the beans.xml. Dropping a jar containing interceptor classes into your application will not cause them to be used. In order to turn them on, you have to enable them in the beans.xml, which also controls the order in which they run. On this slide you can see a very simple example of a beans.xml file that defines two interceptor classes. In this case, if both of these interceptors applied to bean you will see the transactional interceptor run first followed by the logging interceptor, and then finally the bean method.



Decorators are another service CDI provides that is very similar to interceptors. Decorators extend the functionality for existing types. Decorators are tied to the types of the object you are trying to add functionality too.

Decorators implement one or more bean types.

Unlike all other CDI beans they can be abstract, so you can just implement the methods of an interface you are interested in decorating, and the container will take care of the rest.

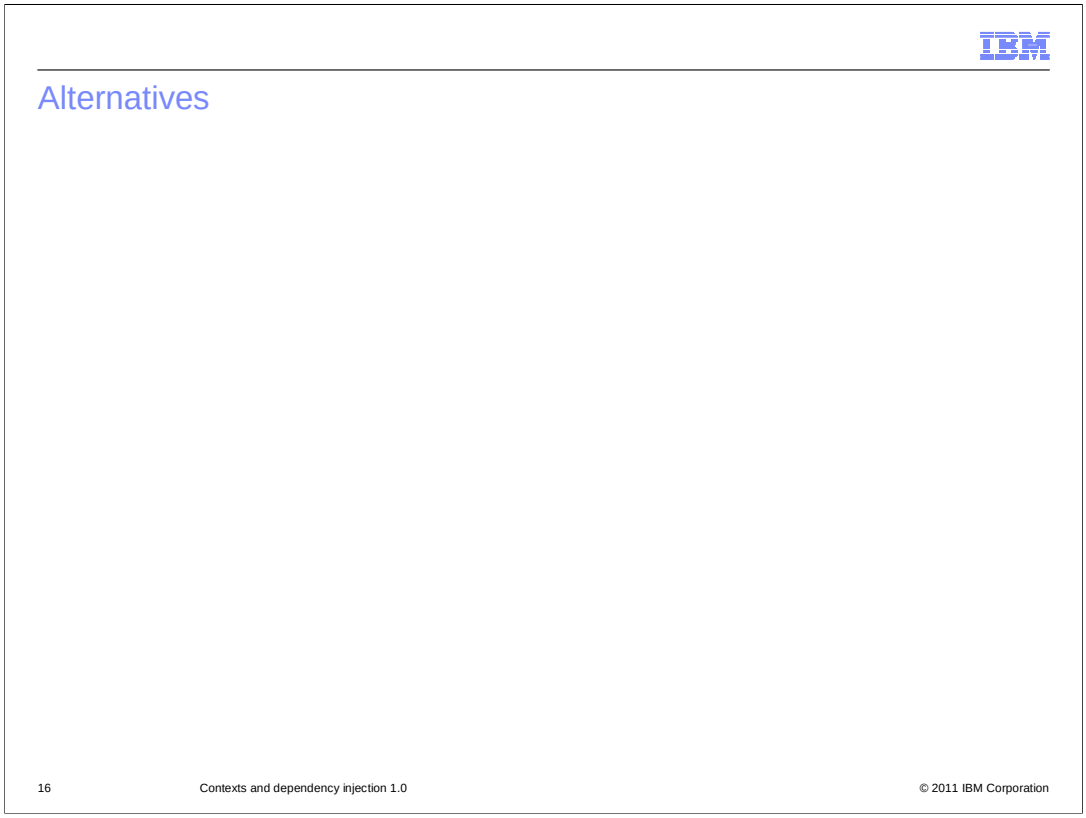
Decorators are called after interceptors and similar to interceptors they have to be enabled in the beans.xml which also controls the ordering.

In these two snippets you are first looking at a definition of an Htmlable interface. This is a very simple interface that has a single method called toHtml.

There is a class, HtmlDate which implements that interface and turns a date into some html output.

Finally at the bottom you see a Decorator. The @Decorator annotation tells the container that StrongDecorator class is a Decorator. On the second line, you will see @Inject used to Inject the Delegate. The Delegate is the Decorators way of interacting with the object it is decorating. In this case of Htmlable, the @Delegate field is the object being decorated.

You can see the decorator has a toHtml method from the interface. In the toHtml method the decorator is adding a strong tag in front of the delegate output, calling the delegate, and then adding a closing strong tag at the end. The decorator is making bold the html output of any bean the implements the Htmlable interface.



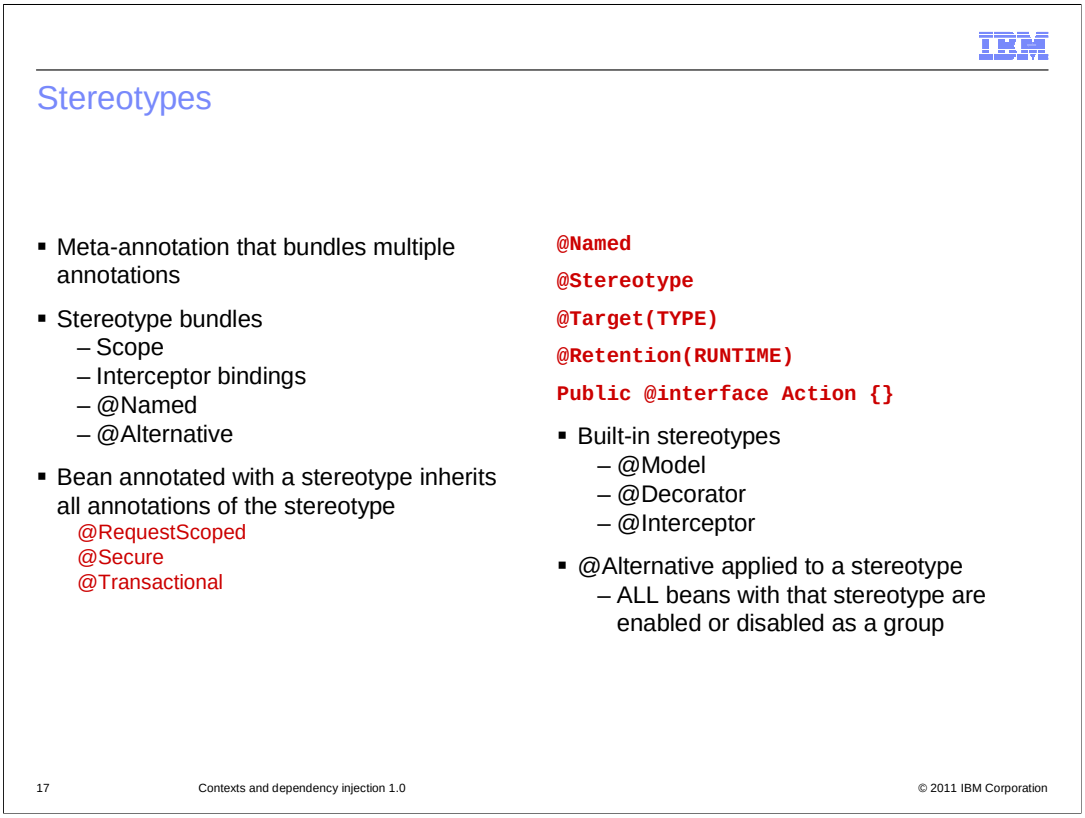
CDI also has a concept of Alternatives. These are beans that will replace other beans based on deployment time configuration.

They are defined using the @Alternative annotation.

You can see here, the MockOrder class which extends Order.

Similar to Decorators and Interceptors, Alternatives have to be enabled in order to be used. Once you drop an archive containing this MockOrder class into your module, it will not start automatically replacing all injections of Order with MockOrder. It has to be enabled explicitly inside the beans.xml.

One use case for alternatives might be a test or development environment where you might want to enable a specific set of classes that are designed for that environment. For example, you might want your credit card processing classes to be mocked up inside of a test environment. Alternatives provides a way to enable this change in your beans.xml, while still having all the Order classes inside the archive.



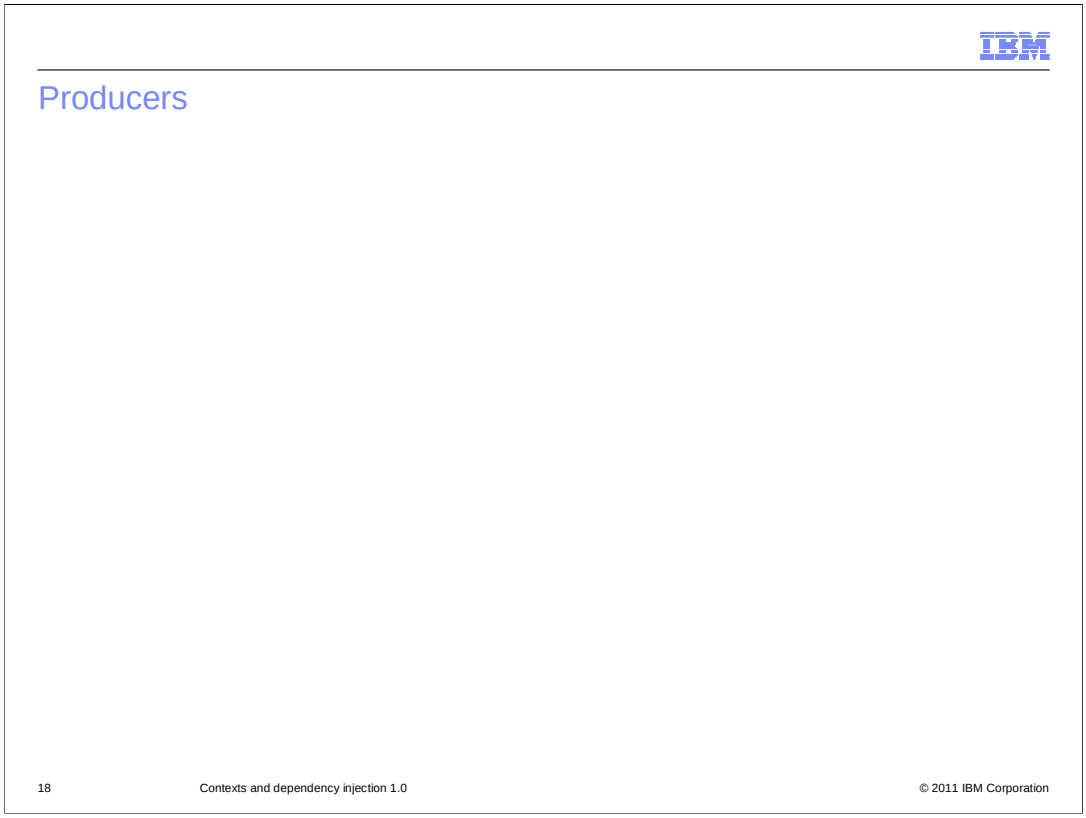
Stereotypes are another concept in CDI. They allow you to bundle lots of annotations together into a single annotation. Effectively it is a meta-annotation.

It can include scope, interceptor bindings, an EL Name, and alternatives all together in a single Stereotype. You can then apply that stereotype annotation directly to your bean.

For example, you see here the Action stereotype. It has the @Stereotype annotation. It contains @RequestScoped, @Secure and @Transactional which might be interceptor bindings defined elsewhere, and it also has @Named. Now any bean that has the @Action annotation is request Scoped, has two interceptor bindings, and is exposed through the EL with the default name.

There are a couple of built-in stereotypes in CDI. @Model is a simple one, that combines @RequestScoped with @Named. So every bean that applies @Model is request scoped and exposed through the EL.

On the previous side Alternatives were covered. Alternatives can also be applied to a Stereotype which makes them a lot more consumable. For example you can define a Mock stereotype, apply that to all of your Mock classes, and then enable and disable them in the beans.xml with a single entry.



CDI also has a concept of producers as a way to provide a bean instance. There can be times when an application needs to have fine grained control over how a bean is constructed or destroyed. For example there might be times that after constructing a bean you want to call some methods on it, or otherwise configure it. Producers give you a way to do that.

CDI has two kinds of producers.

One is producer fields, which is a field inside of a bean that has the @Produces annotation. Other annotations previously covered such as @ApplicationScoped, qualifiers, and @Named can still be used here. The container will then use this field to provide that bean type through injection.

Second is a producer method. Similar to a producer field you see the @Produces annotation, but now it is applied to a method, in this case the method getProducts.

The idea here is that the bean that contains this method might need to do some additional configuration to this list before returning it.

Finally there might be cases where additional cleanup work might be required. For that CDI, has a concept of disposer methods. If an instance is created from a producer method, then the disposer will then later be called when that instance ends its life cycle.

Enterprise Beans (EJB) and CDI

- Enhances the EJB component model with contextual life cycle management
- CDI allows EJB integration with JSF and JSP through EL with @Named
- CDI allows EJBs to be injected in CDI beans and vice versa
- Use @Inject EJB instead of @EJB
- CDI interceptors and decorators can be used with EJBs

CDI enhances the EJB component model with contextual life cycle management.

The container performs dependency injection on EJB session beans.

CDI allows EJB integration with JSF and JSP through EL with @Named. This allows a JSF component or a JSP to directly call methods on an @Named EJB.

CDI allows EJB to be injected in CDI beans and vice versa.

Use @Inject EJB to inject contextual instances instead of @EJB JEE5 style injection.

CDI interceptors and decorators can be used with EJBs. EJB interceptors run before CDI interceptors and decorators.

This section provides a summary of this presentation.

Summary

- CDI provides a set of services to components that either injected with @Inject or exposed through the EL
- Contextual state and life cycle management
- Typesafe dependency injection
- Interceptors and decorators
  - Extend behavior either with or across types
- Designed to help decouple application logic and ease integration across EE components

Contexts and Dependency Injection for the Java EE Platform (CDI) is one of key Java EE6 features that help to knit together the web tier and the transactional tier of the Java EE platform. It is a set of services that, used together, makes it easy for developers to use enterprise beans along with JSF technology in web applications. These set of services include contextual state and life cycle management, typesafe dependency injection, interceptors and decorators. These services help decouple application logic and ease integration across EE components.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WASV8_CDI.ppt

This module is also available in PDF format at: ../WASV8_CDI.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information

IBM, the IBM logo, ibm.com, and WebSphere are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of other IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. Other company, product, or service names may be trademarks or service marks of others.

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.