Dependency Injection in Java EE 6 - Part 2 by Reza Rahman

January 2009

Discuss this Article

This is the second part of a multi-article series. For part 1 please see: Dependency Injection in Java EE 6 - Part 1

This series of articles introduces Contexts and Dependency Injection for Java EE (CDI), a key part of the Java EE 6 platform. Standardized via JSR 299, CDI is the de-facto API for comprehensive next-generation type-safe dependency injection as well as robust context management for Java EE. Led by Gavin King, JSR 299 aims to synthesize the best-of-breed features from solutions like Seam, Guice and Spring while adding many useful innovations of its own.

In the opening article of the series on Dependency Injection, we took a high-level look at CDI, saw how it fits inside Java EE and discussed basic dependency management as well as scoping. In this article, we will cover custom bean factories using producers/disposers, component naming and dynamically looking up beans. In the course of the series, we will cover stereotypes, decorators, interceptors, events, portable extensions and many more. We will also talk about how CDI aligns with Seam, Spring and Guice, as well as augment the discussion with a few implementation details using CanDI, Caucho's independent implementation of JSR 299 included in the open source Resin application server.

Custom Object Factories with CDI Producers/Disposers
In the first part of the series, we showed you how to do basic dependency injection using the @Inject and @Qualifier annotations. In a majority of cases, giving the container static control over object creation via these annotations is fine. However, there are cases where it makes a lot of sense for you to control object creation and destruction yourself in Java code. Good examples are complex object creation/destruction, legacy/third-party API integration, objects that need to be constructed dynamically at runtime and creating injectable strings, collections, numeric values and so on. Along the same lines as the traditional object factory pattern, this is what CDI producers and disposers allow you to do. Another way of looking at this is that the container allows you to step in and take manual control over creating/destroying objects when it makes sense.]

Let's take a look at a somewhat non-trivial example to clarify some of this. In the following code, we'll show you how to create a simple abstraction over the low-level JMS API using CDI producers and disposers. As you might already know, plain JMS code can be pretty verbose because the API is so general purpose. Using CDI, we can create a custom JMS object factory that hides most of the verbose code to handle the JMS connection and session, keeping the actual JMS API client code pretty clean. Here is how the resulting JMS API client code can look:

```
@Stateless
public class OrderService {
  @Inject @OrderSession private Session session;
  @Inject @OrderMessageProducer private MessageProducer producer;

  public void sendOrder(Order order) {
    try {
      ObjectMessage message = session.createObjectMessage();
      message.setObject(order);
      producer.send(message);
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

If you are familiar with JMS resource injection, the code above should look intriguing. Rather than injecting the JMS connection factory and queue, we are instead actually injecting the JMS session and message producer, using qualifiers specific to messaging for orders (@OrderSession, @MessageProducer). The session and message producer is being injected from a shared object factory with a set of CDI producers and disposers handling the JMS session, connection and message producer life-cycle behind the scenes. Here is the code for the custom object factory:

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

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

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

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

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

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

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

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

The underlying order queue and connection factory is being injected into the JMS object factory via the Java EE 5 @Resource annotation. The methods annotated with @Produces are producer methods. The return values of these methods are made available for injection. When needed, you can apply qualifiers, scopes, names and stereotypes to these return values by applying annotations to the producer methods. In our example, we are applying the @OrderConnection, @OrderSession and @OrderMessageProducer qualifiers to the injectable JMS connections, sessions and message producers we are programmatically constructing. The produced objects belong to the default *dependent* scope (recall the scopes we discussed in the first part of the series).

Very interestingly, you should note that producer methods can request injected objects themselves through their method parameters. For example, when the container creates an order session by invoking the createOrderSession method, it sees that the method needs a JMS connection qualified with the @OrderConnection qualifier. It then resolves this dependency by invoking the createOrderConnection producer method. The createOrderMessageProducer producer method is similarly dependent on the createOrderSession producer.

On the other side of the equation any method with a parameter decorated with the @Disposes annotation signifies a disposer. In the example, we have three disposers disposing of JMS connections, sessions and message producers qualified with the @OrderConnection, @OrderSession and @OrderMessageProducer qualifiers. Just as producer methods are called into action as the objects they produce are needed, the disposer associated with an object is called when the object goes out of scope. For the interest of readability, all disposers must belong in the same bean as their matching producer methods.

Having covered some of the mechanical syntax details, let's now take a step back and look at the example from a functional standpoint. When the order service needs the injected order session and message producer, the associated producer methods are invoked by CDI. In the course of creating the session and message producer, CDI also creates the JMS connection, since it is a declared dependency for the producer methods. The producer methods utilize the underlying queue and connection factory injected into our JMS object factory as needed. All these injected objects are then cleaned up properly when they go out of scope.

This example should demonstrate to you how the relatively simple concept of producers and disposers can be combined in innovative and powerful ways when you need it.

Custom Scopes and Resource Factories

Think about the implications of the example above. Because the JMS resources are in the dependent scope, they are tied to the life-cycle of the order service EJB. The service is a stateless session bean that is discarded from the object pool when not in use, along with the injected JMS resources. Now imagine that the service is a singleton or application scoped managed bean instead and is thus very long lived. This means that this code would cause the container to create and inject open connections/sessions that are not released for a while, potentially causing performance problems. Can you think of a way to solve this problem?

Hint: one way to solve this is to use a custom @TransactionScoped or @ThreadScoped for the JMS resources. This is being considered for addition to the Resin container (as such, it is probably a good optimization even for the EJB case). Also under consideration: Adding generic JMS/JDBC abstractions based on CDI similar to the example as part of Resin that you can simply use out-of-the-box. Can you imagine how the JDBC abstraction might look like? Would it be useful to you?

CDI producers and disposers support even more powerful constructs such as utilizing injection point meta-data, using producer fields, or using producers with Java EE resources and so on - you should check out the CDI specification for details (alternatively, feel free to check out the Weld reference guide that's a little more reader-friendly). Utilizing producers and scoped beans with JSF is particularly interesting and we will look at this in a later article focusing on CDI's interaction with JSF.

To give you a taste of some of the more advanced features, let's look at a producer example using injection point meta-data. By using injection point meta-data in producers, you can get runtime information about where the object you are programmatically creating will be getting injected into. You can use this data in very interesting ways in your code. Here is an example that constructs and injects a custom logger into a service:

```
@Stateless
public class BidService {
  @Inject private Logger logger;
  ...
}

public class LoggerFactory {
  @Produces
  public Logger getLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(
      injectionPoint.getMember().getDeclaringClass().getSimpleName());
  }
}
```

As you can see from the example above, all you need to do to get access to injection point meta-data is include the InjectionPoint interface as a producer method parameter. When CDI invokes your producer, it will collect meta-data at the injection point (the logger instance variable in this case) and pass it to your producer. The injection point interface can tell you about the bean being injected, the member being injected, qualifiers applied at the injection point, the Java type at the injection point, any annotations applied and so on. In the example, we are creating a logger specific to the class that holds the logger using the name of the class that is the injection target.

The injection point interface is actually part of the CDI SPI geared toward portable extensions, which we will discuss in a later article.

OpenWebBeans Portable Extensions

One of the goals of the Apache CDI implementation, OpenWebBeans, is to expose various popular Apache projects as CDI portable extensions. It's very likely you will see something similar to the logger example above as a CDI portable extension for Log4J (CDI portable extensions for Apache commons is probably very likely too).

Naming Objects

As we discussed, one of the primary characteristics of CDI is that dependency injection is completely type-safe and not dependent on character-based names that are prone to mistyping and cannot be checked via an IDE, for example. While this is great in Java code, beans would not be resolvable without a character-based name outside Java such as in JSP or Facelet markup. CDI bean name resolution in JSP or Facelets is done via EL. By default, CDI beans are not assigned any names and so are not resolvable via EL binding. To assign a bean a name, it must be annotated with @Named like so:

```
@RequestScoped @Named
```

```
public class Bid {
  ...
}
```

The default name of the bean is assigned to be "bid" - *CamelCase* with the first letter lower-cased. The bean would now be resolvable via EL binding as below:

```
<h:inputText  id="amount" value="#{bid.amount}"/>
```

Similarly, objects created by producer methods are assigned the name of the method or the Java property name by default when the producer is annotated via @Named. The list of products produced in the example below is named "products", for instance:

```
@Produces @ApplicationScoped @Named
public List getProducts() { ... }
```

You can most certainly override default names as needed as shown below:

```
@Produces @ApplicationScoped @Catalog @Named("catalog")
public List getProducts() { ... }

@ConversationScoped @Named("cart")
public class ShoppingCart {
```

In most cases, you would likely not override bean names, unless overriding names improves readability in the context of JSP/Facelet EL binding.

Names in Dependency Injection
Technically, @Named is defined to be a qualifier by JSR 330. This means that you can use character-based names to resolve dependencies in JSR 299 if you really, really want to. However, this is discouraged in CDI and there are few good reasons to do it instead of using more type-safe Java based qualifiers (can you think of what the reasons might be?).

Looking up CDI Beans
While dependency injection is the way to go most of the time, there are some cases where you cannot rely on it. For example, you may not know the bean type/qualifier until runtime, it may be that there are simply are no beans that match a given bean type/qualifier or you may want to iterate over the beans which match a certain bean type/qualifier. The powerful CDI Instance construct allows for programmatic bean look-up in such cases. Let's see how this works via a few examples.

In the simplest case, you may want to know about the existence of a bean with a certain type (let's say a Discount) because you are not certain if it was deployed given a particular system configuration. Here is how you can do it:

```
@Inject private Instance discount;
...
if (!discount.isUnsatisfied()) {
  Discount currentDiscount = discount.get();
}
```

In the example above, we are injecting the typed Instance for the discount bean. Note, this does not mean that the discount itself is injected, just that you can look-up discounts as needed via the typed Instance. When you need to query for the discount, you can check if it exists via the isUnsatisfied method. You can also check for ambiguous dependencies if you need to. If the discount was deployed, you can then actually get the resolved bean and use it. If you need to, you can apply qualifiers to an Instance. For example, you can look for a holiday discount as below:

```
@Inject @Holiday private Instance discount;
```

Now let's assume that there might be multiple matches for a particular bean type/qualifier - there is likely more than one discount, for example. You can handle this case by placing the @Any annotation on an Instance as below and iterating over all the matches:

```
@Inject @Any private Instance anyDiscount;
...
for (Discount discount: anyDiscount) {
  // Use the discount...
}
```

Note that Instance is an Iterable as a convenience, which is why the abbreviated foreach loop syntax above works. If needed, you can narrow down the matches by qualifier and/or sub-type at runtime. This is how you could filter by qualifier:

```
@Inject @Any private Instance anyDiscount;
...
Annotation qualifier = holiday ? new Holiday() : new Standard();
Discount discount = anyDiscount.select(qualifier).get();
```

The select method allows you to narrow down the potential discount matches by taking one or more qualifiers as arguments. If it's a holiday, you would be interested in holiday discounts instead of standard discounts in the example code. You can also pass a bean sub-type to the select method as below:

```
Instance sellerDiscount =
  anyDiscount.select(SellerDiscount.class, new Standard());
```

There is much more to the Instance construct that you should definitely check out by taking a look at the CDI specification itself (or the Weld reference guide). You can also use the CDI SPI intended for creating portable extensions to look up beans in a more generic fashion. We will cover portable extensions in a future article in the series.

Much More to Come

We want to thank all the readers that have sent in kind words on the article series. We hope that we continue to do justice to your interest. In the coming parts, we will cover more features of the broad CDI API including grouping metadata in stereotypes, lightweight event management via dependency injection, using the new conversation scope to manage application state, CDI's interaction with JSF, interceptors, decorators and much more. For comments on CDI, you are welcome to send an email to jsr-299-comments@jcp.org. You can also send general comments on Java EE 6 to jsr-316-comments@jcp.org. For comments on the article series, Resin or CanDI, our JSR 299 implementation, feel free to email us at reza@caucho.com or ferg@caucho.com. Cheers until next time!

References

1. JSR 299: Contexts and Dependency Injection for Java EE, http://jcp.org/en/jsr/detail?id=299.
2. JSR 299 Specification Final Release, https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_JCP-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=web_beans-1.0-fr-oth-JSpec@CDS-CDS_JCP.
3. Weld, the JBoss reference implementation for JSR 299: http://seamframework.org/Weld.
4. Weld Reference Guide, http://docs.jboss.org/weld/reference/1.0.0/en-US/html/.
5. CanDI, the JSR 299 implementation for Caucho Resin, http://caucho.com/projects/candi/.
6. OpenWebBeans, Apache implementation of JSR 299, http://incubator.apache.org/openwebbeans/.

About the Authors

Reza Rahman is a Resin team member focusing on its EJB 3.1 Lite container. Reza is the author of *EJB 3 in Action* from Manning Publishing and is an independent member of the Java EE 6 and EJB 3.1 expert groups. He is a frequent speaker at seminars, conferences and Java user groups, including JavaOne and TSSJS.

Scott Ferguson is the chief architect of Resin and President of Caucho Technology. Scott is a member of the JSR 299 EG. Besides creating Resin and Hessian, his work includes leading JavaSoft's WebTop server as well as creating Java servers for NFS, DHCP and DNS.