TheServerSide.com

Part 3 of dependency injection in Java EE 6

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 previous articles in the series, we took a high-level look at CDI, discussed basic dependency management, scoping, producers/disposers, component naming and dynamically looking up beans. In this article we will discuss interceptors, decorators, stereotypes and events. In the course of the series, we will cover conversations, CDI interaction with JSF, portable extensions, available implementations as well as CDI alignment with Seam, Spring and Guice. We will 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.

Cross-cutting concerns with CDI interceptors

Besides business logic, it is occasionally necessary to implement system level concerns that are repeated across blocks of code. Examples of this kind of code include logging, auditing, profiling and so on. This type of code is generally termed "cross-cutting concerns" (although subject to much analysis as a complement to object orientation, these types of concerns really don't occur that often in practice). CDI interceptors allow you to isolate cross-cutting concerns in a very concise, type-safe and intuitive way.

The best way to understand how this works is through a simple example. Here is some CDI interceptor code to apply basic auditing at the EJB service layer:

```
@Stateless
public class BidService {
   @Inject
   private BidDao bidDao;

   @Audited
   public void addBid(Bid bid) {
      bidDao.addBid(bid);
   }
   ...
}

@Audited @Interceptor
public class AuditInterceptor {
   @AroundInvoke
   public Object audit(InvocationContext context) throws Exception {
      System.out.print("Invoking: "
         + context.getMethod().getName());
      System.out.println(" with arguments: "
         + context.getParameters());
      return context.proceed();
   }
}
```

@InterceptorBinding

```
@Target({TYPE, METHOD})

@Retention(RUNTIME)

public @interface Audited {}
```

Whenever the addBid method annotated with @Audited is invoked, the audit interceptor is triggered and the audit method is executed. The @Audited annotation acts as the logical link between the interceptor and the bid service. @InterceptorBinding on the annotation definition is used to declare the fact that @Audited is such a logical link. On the interceptor side, the binding annotation (@Audited in this case) is placed with the @Interceptor annotation to complete the binding chain. In other words, the @Audited and @Interceptor annotations placed on AuditInterceptor means that the @Audited annotation placed on a component or method binds it to the interceptor.

Note a single interceptor can have more than one associated interceptor binding. Depending on the interceptor binding definition, a binding can be applied either at the method or class level. When a binding is applied at the class level, the associated interceptor is invoked for all methods of the class. For example, the @Audited annotation can be applied at the class or method level, as denoted by @Target({TYPE, METHOD}). Although in the example we chose to put @Audited at the method level, we could have easily applied it on the bid service class instead.

We encourage you to check out the CDI specification for more details on interceptors including disabling/enabling interceptors and interceptor ordering (alternatively, feel free to check out the Weld reference guide that's a little more reader-friendly).

Custom vs. Built-in Interceptors

EJB declarative transaction annotations like @TransactionAttribute and declarative security annotations like @RolesAllowed, @RunAs can be thought of as interceptor bindings built into the container. In fact, this is not too far from exactly how things are implemented in Resin.

In addition to the EJB service annotations, we could add a number of other built-in interceptors for common application use-cases in Resin including @Logged, @Pooled, @Clustered, @Monitored, etc. Would this be useful to you?

Isolating pseudo business concerns with CDI decorators

Interceptors are ideal for isolating system-level cross-cutting concerns that are not specific to business logic. However, there is a class of cross-cutting logic that is closely related to business logic. In such cases, you will have logic that really should be externalized from the main line of business logic but is still very specific to the interception target type, method or parameter values. CDI decorators are intended for such use cases. Like interceptors, decorators are very concise, type-safe and pretty natural.

As in the case with interceptors, the best way to understand how decorators work is through a simple example. We'll use the convenient bid service example again. Let's assume that the bid service is used in multiple locales. For each locale bid monetary amounts are entered and displayed in the currency specific to the locale. However, the bid amounts are internally stored using a standardized currency (such as maybe the Euro or the U.S. Dollar). This means that bid amounts must be converted to/from the locale specific currency, likely at the service tier. Because the currency conversion code is not strictly business logic, it should really be externalized, but it is very specific to bid operations. This is a good use case for decorators, as shown in the code below:

```
@Stateless

public class DefaultBidService implements BidService {

    ...

    public void addBid(Bid bid) {

    ...

}
```

```
@Decorator
public class BidServiceDecorator implements BidService {
    @Inject @Delegate
    private BidService bidService;

    @Inject @CurrentLocale
    private Locale locale;

    @Inject
    private Converter converter;

    public void addBid(Bid bid) {
        bid.setAmount(converter.convert(bid.getAmount(),
            locale.getCurrency(), Converter.STANDARDIZED_CURRENCY));

        bidService.addBid(bid);
    }
    ...
}
```

As you can see from the code example, the currency conversion logic is isolated in the decorator annotated with the @Decorator annotation. The decorator is automatically attached and invoked before the interception target by CDI (just as in the case of interceptors). A decorator cannot be injected directly into a bean but is only used for the purposes of interception. The actual interception target is injected into the decorator using the @Delegate built-in qualifier. As you can also see, decorators can utilize normal bean injection semantics. If the Decorator/Delegate terminology sounds familiar, it is not an accident. CDI Decorators and Delegates essentially implement the well-known Decorator and Delegate OO design patterns. You can use qualifiers with @Delegate to narrow down which class a decorator is applied to like this:

```
@Decorator
public class BidServiceDecorator implements BidService {
    @Inject @Legacy @Delegate
    private BidService bidService;
    ...
}
```

The CDI specification (or the Weld reference guide) has more details on decorators including disabling/enabling decorators and decorator ordering.

Custom component models with CDI stereotypes

CDI stereotypes essentially allow you to define your own custom component model by grouping together meta-data. This is a very powerful way of formalizing the recurring bean roles that often arise as a result of application architectural patterns. For example, in a tiered server-side application, you can imagine component definitions for the service, DAO or presentation-tier model (the ‘M’ in MVC). A stereotype consists of a default component scope and one or more interceptor bindings. A stereotype may also indicate that a bean will have a default name (essentially indirectly decorating it with @Named) or that a bean is an *alternative* (indirectly decorated with @Alternative). A stereotype may also include other stereotypes.

Alternatives

An *alternative* is anything marked with the @Alternative annotation. Unlike regular beans, an alternative must be explicitly enabled in beans.xml. Alternatives are useful as mock objects in unit tests as well as deployment-specific components. Alternatives take precedence over regular beans for injection when they are available.

We won't discuss alternatives beyond this here, but we encourage you to explore them on your own.

A stereotype defined for the DAO layer in our example bidding application could look like the following:

```
@Profiled
@Dependent
@Stereotype
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Dao {
}
```

As you can see, the @Stereotype annotation denotes a stereotype. Our stereotype is declared to have the dependent scope by default. This makes sense since DAOs are likely injected into EJBs in the service tier. The interceptor binding @Profiled is also included in the stereotype. This means that any bean annotated with the @Dao stereotype may be profiled for performance via an interceptor bound to @Profiled. The stereotype would be applied to a DAO like this:

```
@Dao
public class DefaultBidDao implements BidDao {
   @PersistenceContext
   private EntityManager entityManager;
   ...
}
```

To solidify the idea of stereotypes a little more, let's take a look at another example. CDI actually has a built-in stereotype - @Model. Here is how it is defined:

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

The @Model annotation is intended for beans used as JSF model components. This is why they have the request scope by default so that they are bound to the life-cycle of a page and are named so that they can be resolved from EL. This is how @Model might be applied:

```
@Model
```

public class Login {

Note it is possible to override the default scope of a stereotype. For example, you can turn the Login bean into a session scoped component like this:

@SessionScoped @Model

public class Login {

It is also possible to place more than one stereotype on a given class, as well as apply additional interceptors, decorators, etc. As we mentioned earlier, stereotypes can also be cumulative, meaning that a stereotype can include other stereotypes in its definition.

The EJB Component Model as Stereotypes

It is an interesting question to ask whether the EJB component model (@Stateless, @Stateful, etc) can be modeled simply as a set of highly specialized stereotypes for the business/service tier. This is a logical next step from redefining EJBs as managed beans with additional services as was done in Java EE 6 and could open up some very powerful possibilities for the Java EE component model going forward.

This is one possibility we are actively exploring for the Resin EJB 3.1 Lite container.

Lightweight type-safe events with CDI

Events are useful whenever you need to loosely couple one or more invokers from one or more invocation targets. In enterprise applications events can be used to communicate between logically separated tiers, synchronize application state across loosely related components or to serve as application extension points (think about Servlet context listeners, for example). Naturally CDI events are lightweight, type-safe, concise and intuitive. Let's look at this via a brief example to see how events in CDI work.

Let's assume that various components in the bidding system can detect and generate fraud alerts. Similarly, various components in the system need to know about and process the fraud alerts. CDI events are a perfect fit for such a scenario because the producers and consumers are so decoupled in this case. The code to generate a fraud alert event would look like this:

@Inject

private **Event<Fraud>** fraudEvent;

…

Fraud fraud = new Fraud();

…

fraudEvent.fire(fraud);

CDI events are triggered using injected Event objects. The generic type of the Event is the actual event being generated. Like the Fraud object, events are simple Java classes. In our example, we would construct the fraud object and populate it as needed. As you can see, events are triggered by invoking the fire method of Event. When the event is triggered, CDI looks for any matching *observer methods* that are listening for the event and invokes them, passing in the event as an argument. Here is how an observer method for our fraud alert would look like:

public void processFraud(**@Observes** Fraud fraud) { ... }

An observer method is simply a method that has a parameter annotated with the @Observes annotation. The type of the annotated parameter must match the event being triggered. The name Observer mirrors the Observer OO design pattern. You can use qualifiers to filter observed events as needed. For example, if we were only interested in seller fraud, we could place a qualifier on the observer method

like this:

public void processSellerFraud(**@Observes @Seller** Fraud fraud) { ... }

On the producer side, there are a couple of ways to attach qualifiers to trigged events. The most simple (and common) way would be to declaratively place a qualifier on the injected event like this:

@Inject @Seller

private **Event<Fraud>** sellerFraudEvent;

It is also possible to attach qualifiers programmatically using the Event.select method like this:

if (sellerFraud) {

fraudEvent.select(new Seller()).fire(fraudEvent);

}

There is a lot more to events than this like injecting parameters into observer methods, transactional observers and the like that you should investigate on your own.

Events and Messages

There are a lot of obvious parallels between events and traditional messaging with JMS. This is one of the avenues we are exploring further in Resin to see if these models could be merged in a simple and intuitive way – namely if the Event object can be used to send JMS messages and/or if @Observer could listen for JMS messages.

More to come

In the next part of the series we will be focusing on CDI as it relates to JSF developers at the presentation tier (many of you have expressed specific interest in this topic). We will cover using the new conversation scope as well as CDI's interaction with JSF using EL binding, scoping, producers, qualifiers and the like.

In the meanwhile, 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. Adios Amigos!

References

1. JSR 299: Contexts and Dependency Injection for Java EE
2. JSR 299 Specification Final Release
3. Weld, the JBoss reference implementation for JSR 299
4. Weld Reference Guide
5. CanDI, the JSR 299 implementation for Caucho Resin
6. OpenWebBeans, Apache implementation of JSR 299

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'stable WebTop server as well as creating Java servers for NFS, DHCP and DNS.

He lead performance for Sun Web Server 1.0, the fastest web server on Solaris.