TheServerSide.com

Dependency Injection in Java EE 6 - Part 5

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 discussed basic dependency injection, scoping, producers/disposers, component naming, interceptors, decorators, stereotypes, events and conversations. In this article we will discuss CDI's interaction with JSF in detail. In the next and last article of this series, we will cover 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.

CDI and JSF

A majority of CDI features are generic to all tiers of a Java EE application and not specific to JSF. In fact most CDI features are useful even in a Java SE application. However, there are a few CDI features that are specific to the web tier and there are many middle-tier oriented features that can be combined in powerful ways with JSF. In the following sections we will explore how features like CDI component naming, scopes, injection, producers, qualifiers and EJB 3 services can be used effectively with JSF.

Note JSF 2 retains its own web tier component model via the @ManagedBean set of annotations as well as annotations for the view, request, session and application scopes. Indeed, you can use EJB 3 and JSF 2 without CDI if you so wish. However there are few compelling reasons to do so considering the much richer set of CDI features that can be used with JSF 2 and EJB 3, including the ones we will discuss here.

Component naming via the @Named annotation is perhaps the easiest place to start. As discussed in the second article in the series, it is the critical link between the type-based Java component world and the string identifier based JSF/Facelet world. Besides making a CDI component available through EL binding expressions by assigning it a resolvable name, the @Named annotation is also useful for overriding the default name of a bean to make it more readable in EL. As the examples in the second article demonstrate, you could shorten the name of a component from the default "bidItemPostingWizard" to simply "itemWizard":

```
@ConversationScoped @Named("itemWizard")

public class BidItemPostingWizard
```

Besides providing the critical CDI-JSF linkage, the @Named annotation is fairly unremarkable so we won't discuss it further here besides using it in the JSF/CDI example use-cases in the following sections.

Web Frameworks other than JSF

Although this article covers JSF 2 only, CDI can also be used in a more-or-less similar fashion with Web Frameworks like Wicket or even Struts 2. Indeed, Seam 3 already supports Wicket and we are considering Struts 2 integration as part of Resin/CanDI. Would such integration be helpful to you?

CDI Contexts and JSF

CDI automatic context management complements JSF's component-oriented model in very powerful ways by abstracting away boilerplate HTTP request, session and cookie manipulation code. We discussed CDI scopes and context management in detail in the very first article of the series. We also focused on just the innovative conversation scope in the fourth article, including JSF examples. Context management centers on the ability to transparently place managed beans into a declared scope and injecting them from the scope when needed. The striking effect of this capability is generally most noticeable to developers using action-oriented web frameworks like Struts.

Let's take an example common to most applications to illustrate some of this – a login component. Such a component in its most basic form would be able to handle the login event. The login information should also be saved into the session after the login completes so that it can be retrieved where needed throughout the session. You can accomplish this by simply creating a CDI session scoped login component:

```
@Named

@SessionScoped

public class Login implements Serializable {

  @Inject

  private UserService userService;
  private String username;
  private String password;
  private User user;

  public void setUsername(String username) {
    this.username = username;
  }

  public String getUsername() {
    return username;
  }

  public void setPassword(String password) {
    this.password = password;
  }

  public String getPassword() {
    return password;
  }

  public void setUser(User user) {
    this.user = user;
  }

  public User getUser() {
    return user;
  }

  public String login() {
    user = userService.getUser(username);
    ...
    return "account.jsf";
  }
}
```

You would use the component in a JSF page to handle the login event as follows:

```
<h:form>
```

```
  <h1>Login</h1>
  <h:panelGrid columns="2">
    <h:outputLabel for="username">Username:</h:outputLabel>
    <h:inputText id="username" value="#{login.username}"/>
    <h:outputLabel for="password">Password:</h:outputLabel>
    <h:inputSecret id="password" value="#{login.password}"/>
  </h:panelGrid>
  <h:commandButton value="Login" action="#{login.login}"/>
</h:form>
```

The username and password properties are bound to input text fields while the login method is bound to the login event. The login method uses the injected user service to retrieve user details. These user details can then be accessed by any other component since the login component is placed in the session scope. For example the component to add a bid could use the login data as follows:

```
@Named
@RequestScoped
public class BidManager {
  @Inject
  private BidService bidService;

  @Inject
  private Login login;

  @Inject @SelectedItem
  private Item item;

  @Inject
  private Bid bid;

  public String addBid() {
    bid.setBidder(login.getUser());
    bid.setItem(item);
    bidService.addBid(bid);

    return “bid_confirm.xhtml”;
  }
}
```

CDI resolves the injection request to the login component stored in the session scope. Note that the bid manager component itself is in the request scope so in this example we are retrieving data stored in the underlying HTTP session into the HTTP request context. Without automatic context management, you would have to save the user information into the HTTP session yourself and retrieve it manually when needed via boilerplate API calls. Typical manual session manipulation code is usually also accompanied by needless database look-ups. In our example, most naive developers would likely only store a user identifier into the session on login and look-up the user data from the database when the bid is added.

You should also note the other interesting things going on in the bid manager component. The injection of the back-end bid service is pretty straightforward. However, the bid and item injection is more interesting. The item object being injected was likely placed into scope before the bid was placed and has the selected item qualifier placed on it. We will see how CDI qualifiers can be used effectively at the presentation tier shortly. Similarly, the injected bid object encapsulates bid data such as the bid amount. This serves to better model the presentation tier in an object oriented fashion and also promotes the reuse of domain objects across tiers (for example, the bid object could be a JPA mapped entity). You can use a similar technique to better model our example login component:

```
@Named
@SessionScoped
public class Login implements Serializable {
  @Inject
  private Credentials credentials;

  @Inject
  private UserService userService;

  private User user;

  public void setUser(User user) {
    this.user = user;
  }

  public User getUser() {
    return user;
  }

  public String login()
  {
    user = userService.getUser(credentials.getUsername());
    ...
    return "account.jsf";
  }
}

@Named
@RequestScoped
public class Credentials {
  private String username;
  private String password;
  public void setUsername(String username) {
    this.username = username;
  }

  public String getUsername() {
    return username;
  }

  public void setPassword(String password) {
    this.password = password;
  }

  public String getPassword() {
    return password;
  }
}
```

We have re-factored out the username and password fields into a request scoped credentials component that is injected into the login component. Because the credentials are in the request scope and not the session scope, they are discarded right after the login event, which is what we really want to have happen anyway. The re-factored JSF page will look like this and is probably a readability improvement as well:

```
<h:form>
  <h1>Login</h1>
  <h:panelGrid columns="2">
    <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}"/>
  </h:panelGrid>
  <h:commandButton value="Login" action="#{login.login}"/>
</h:form>
```

You can freely interweave object scopes as your application requirements evolve. You can inject session scoped objects into request or conversation scoped objects, application scoped objects into session scoped objects, request scoped objects into session scoped objects and the like. CDI proxies make sure that there is not an issue even if a narrower scoped object is injected into a broader scoped object – CDI will always resolve to the currently active instance under the hood. For example, if a request scoped object is injected into a conversational component, CDI will ensure that the conversational component always references the most up-to-date request scoped object via the injected proxy. As an exercise, you should try to think about how you would code a conversation scoped user preferences wizard to add one or more request scoped preferences (hint: you would inject the request scoped preferences object and use it in the method to add a preference to the current set - the correct request scoped reference will be resolved automatically even after initial injection).

Producers in JSF

We discussed CDI producers in detail in the second article in the series. In most cases, components in the web tier will be defined statically. However, just as is the case with back-end components, it is sometimes useful to generate components dynamically in the web tier via producers. Let’s revisit our login component as an example. If you look carefully you’ll notice that the bid manager component really does not need a reference to the login component per se. What it really needs is the user object that the login component holds a reference to. Injecting the login component instead of the user component makes our code less loosely coupled. This problem can easily be solved by introducing a producer in the login component:

```
@Named
@SessionScoped
public class Login implements Serializable {
  @Inject
  private Credentials credentials;

  @Inject
  private UserService userService;

  private User user;

  public String login()
  {
    user = userService.getUser(credentials.username);
    ...
    return "account.jsf";
  }

  @Produces
  public User getCurrentUser()
  {
    return user;
  }
}

@Named
@RequestScoped
public class BidManager {
  @Inject
  private BidService bidService;

  @Inject
  private User user;

  @Inject @SelectedItem
  private Item item;
  @Inject
  private Bid bid;

  public String addBid() {
    bid.setBidder(user);
    bid.setItem(item);
    bidService.addBid(bid);

    return “bid_confirm.xhtml”;
  }
}
```

Although in this example the producer method itself is in a session scoped bean, producers can be encapsulated in pure factory objects just as in the example in the second article. In such a case, you will likely want to attach an appropriate scope to the producer method like this:

```
@Produces @SessionScoped
public User getCurrentUser()
{
  ...
}
```

You can also use CDI producers (in particular producer fields) for better encapsulation and improving readability in JSF. Let’s take a look at the following simplified component to place a bid:

```
@RequestScoped @Named
public class PlaceBid {
  @Inject
  private BidService bidService;

  @Produces @Named
  private Bid bid = new Bid();

  public void placeBid() {
    bidService.addBid(bid);
  }
}
```

In this example, instead of embedding the fields of the bid in the place bid component, we have better encapsulated it in the bid object and exposed the bid object as a named producer field. The corresponding JSF component will look something like this:

```
<h:form>
  <table>
```

```
    <tr>
      <td>Bidder</td>
      <td><h:inputText value="#{bid.bidder}"/></td>
    </tr>
    <tr>
      <td>Item</td>
      <td><h:inputText value="#{bid.item}"/></td>
    </tr>
    <tr>
      <td>Bid Amount</td>
      <td><h:inputText value="#{bid.price}"/></td>
    </tr>
  </table>
  ...
  <h:commandButton type="submit" value="Place Bid"
    action="#{placeBid.placeBid}"/>
  ...
</h:form>
```

Without the named producer, the bid fields could still be referenced in EL but would be much less readable as below:

```
<h:inputText value="#{placeBid.bid.price}"/>
```

This technique is especially helpful for complex forms with many inputs that might be best grouped into separate objects exposed by named producers from a master form handler.

CDI Qualifiers and JSF

Qualifiers are invaluable particularly while using CDI producers with JSF. We discussed qualifiers in detail in the first article of the series. Unlike business and persistence tier components, you typically will not need to swap out implementations at the web tier using qualifiers. However, while creating dynamic components, it is often necessary to qualify them to reduce the possibility of an accidental dependency collision. Let’s revisit the login component to see why this might be the case with CDI producers:

```
@Named @SessionScoped
public class Login implements Serializable {
  ...
  @Produces
  public User getCurrentUser()
  {
    return user;
  }
}

@Named @RequestScoped
```

```
public class BidManager {
  ...
  @Inject
  private User user;
  ...
}
```

The problem with the code above is that there might be other dynamically produced user objects in the session or request scopes. For example, there might be a currently selected seller or customer service agent that is also a user object for the same session. Imagine this code used from the customer service live chat screen:

```
@Named @SessionScoped
public class CustomerService implements Serializable {
  ...
  @Produces @Agent
  public User getSelectedAgent()
  {
    return selectedAgent;
  }
}
```

Because there will now be two candidates for the injection point in the bid manager it will become ambiguous as to which one to inject at runtime. The best way to resolve the dependency correctly would be to introduce a qualifier:

```
@Named @SessionScoped
public class Login implements Serializable {
  ...
  @Produces @LoggedIn
  public User getCurrentUser()
  {
    return user;
  }
}


@Named @RequestScoped
public class BidManager {
  ...
  @Inject @LoggedIn
  private User user;
  ...
}
```

Besides avoiding possible dependency collisions, adding qualifiers to producers generally improves readability, as is illustrated in our example. Both the producer and the injection point make it clear what type of user is expected at runtime.

Flattened Layers with JSF

Most server-side Java applications are typically built with large teams, heavy maintenance and longevity in mind. This is one reason layering is so popular in Java EE applications. Layers improve testability, enforce separation of concerns, help skills specialization and make it easier to replace the implementation of a specific layer. However, layering can be overkill for smaller teams, smaller applications and prototypes.

Keeping this fact in mind, CDI allows you to flatten the business/transactional and persistence tiers if you so wish. It enables this by allowing you to use EJB 3 and JPA directly with JSF without any intermediate service or DAO tiers. For example, the simplified bid manager component we looked at earlier can be re-factored to be a session bean that uses an injected JPA entity manager directly:

```
@Stateful @RequestScoped @Named

public class BidManager {

  @PersistenceContext

  private EntityManager entityManager;


  @Produces @Named

  private Bid bid = new Bid();


  public void addBid() {

    entityManager.persist(bid);

  }

}
```

Because it is an EJB, the bid manager is automatically thread-safe so it can use the non-thread-safe entity manager reference directly and is also transactional by default. The JSF page using the bean would remain unchanged:

```
<h:form>

  <table>

    <tr>

      <td>Bidder</td>

      <td><h:inputText value="#{bid.bidder}"/></td>

    </tr>

    <tr>

      <td>Item</td>

      <td><h:inputText value="#{bid.item}"/></td>

    </tr>

    <tr>

      <td>Bid Amount</td>

      <td><h:inputText value="#{bid.price}"/></td>

    </tr>

  </table>

  ...

  <h:commandButton type="submit" value="Place Bid"
```

```
    action="#{bidManager.addBid}"/>
  ...
</h:form>
```

As the application evolves, you could, of course, always re-factor the flattened components into web, domain, business and persistence tiers. The capacity to use EJB 3 and JPA directly with JSF simply gives you some added flexibility when you need it.

Using EJB Services Outside the Component Model

You can flatten the application tiers in Resin 4 without needing to use EJB per se by directly adding EJB services such as transactions to CDI managed beans. In our example, you would add the @TransactionAttribute annotation instead of the @Stateful annotation to make the bid manager transactional. Entity manager thread-safety is not an issue in Resin 4 since all JPA provided entity managers are automatically wrapped in a thread-safe proxy.

More to Come

Besides the CDI features we discussed in detail here, there are many other CDI features that can be used in powerful ways with JSF. You can use CDI stereotypes to define components specific to the web tier in addition to the built-in @Model stereotype. For example, you could add a @Wizard stereotype that is named, conversation scoped and attaches interceptors/decorators specific to wizard components. You could use CDI events to trigger actions from the web tier in a loosely-coupled fashion. You could also use the features we described here in many other innovative ways for your particular application. We encourage you to further explore these possibilities on your own.

As we mentioned at the beginning of the article, we are now almost at the end of this series and hope it was useful to you. In the next and last article of this series we will discuss CDI portable extensions, CDI's alignment with Seam, Guice, Spring, etc as well as the current implementations of CDI including Weld/Seam 3, OpenWebBeans and CanDI.

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. Seam.
6. CanDI, the JSR 299 implementation for Caucho Resin.
7. 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's 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.