TheServerSide.com

Dependency Injection in Java EE 6: Conversations (Part 4)

Dependency Injection in Java EE 6 (Part 4) by Reza Rahman

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 and events. In this article we will discuss CDI Conversations in detail. In future articles, we will cover details of using CDI 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.

The Concept of Conversations

In the first article of the series, we discussed conversations very briefly from the perspective of CDI scopes in general. The conversation scope is an idea that comes from Seam and deserves a detailed look, especially for understanding how to use it with JSF.

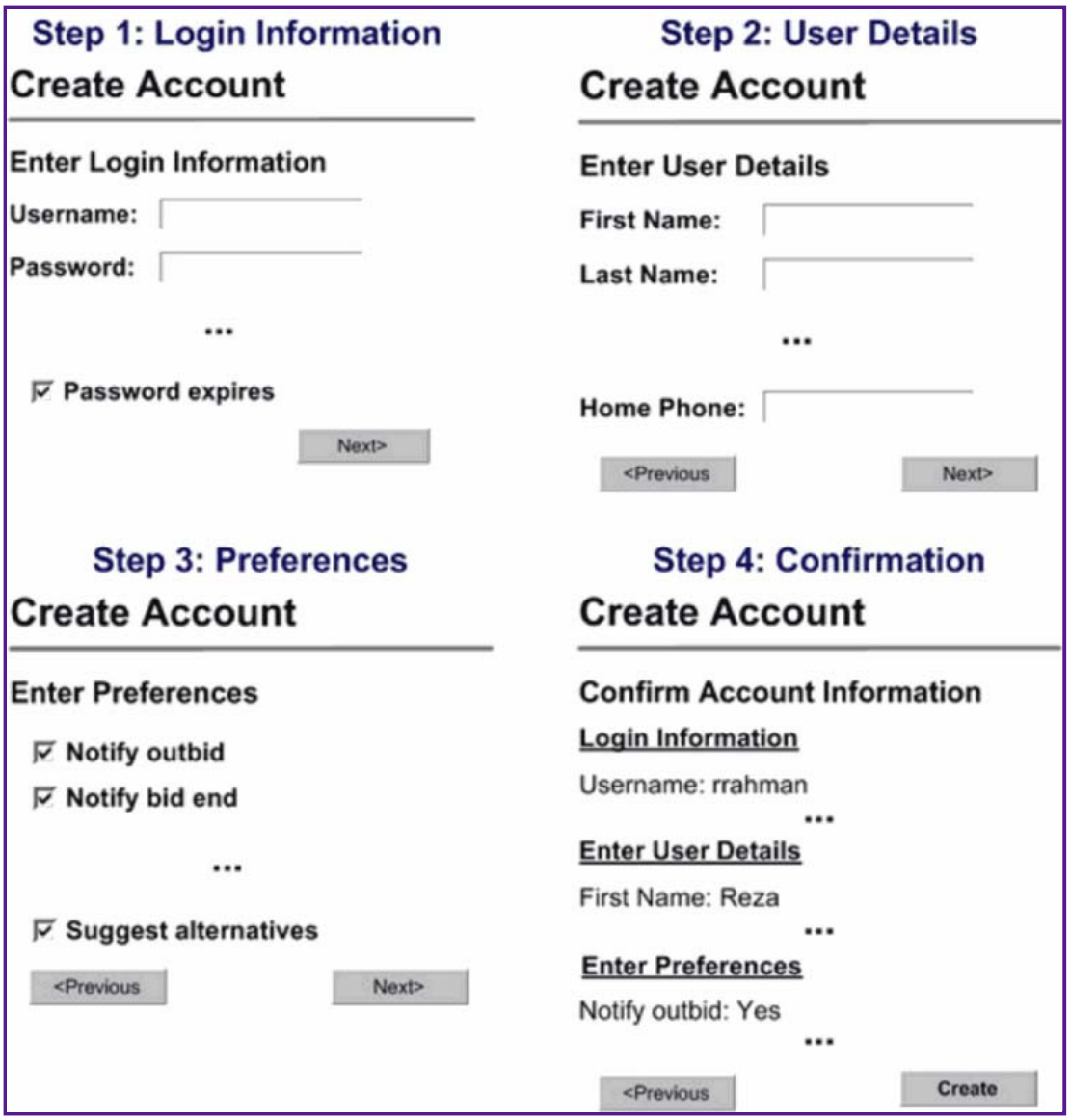
Most server-side Java developers are very familiar with the request and session scopes. That is likely how you started web application state management, probably using the programmatic APIs defined in the Servlet specification. As a result, it is pretty obvious how the CDI request and session scopes are used with JSF through @RequestScoped and @SessionScoped beans (although we will still discuss this in some detail in the next article in the series). The most typical use of a @RequestScoped bean is as a JSF backing bean for a single page handling one HTTP request/response cycle. The vast majority of CDI beans you will use with JSF will likely belong to the request scope. In a similar vein, @SessionScoped beans are used for objects that are used throughout the HTTP session. Examples of this include user login credentials, account details and so on.

There is a relatively large class of web application use-cases that fall between these two logical extremes. There are some presentation tier objects that can be used across more than one page/request but are clearly not used across the entire session. Such objects are usually used in multi-step workflows. Unlike session scoped objects that are usually timed-out, conversation scoped objects have well-defined life-cycle start and end-points that can be determined ahead of time. A good example use-case for the conversation scope is an online quiz or questionnaire (both of which I'm sure we've all encountered much more frequently than we would like). While such use-cases are often implemented as part of the HTTP session, there really is no good reason to do so since the life-cycle of a quiz or questionnaire can be pretty well-defined and they are not needed across the entire session. The life-cycle of a quiz/questionnaire would begin with the first question. As part of the application workflow, the user will progress through questions. The user may also go back and forth through responses an arbitrary number of times. Finally, the quiz/questionnaire would end its life-cycle after the user finalizes their responses. Infrequently, the user will simply abandon the quiz/questionnaire, so conversation scoped objects do still need a timeout mechanism in case the event that defines the end of the workflow never happens. An order process that translates a shopping cart into a finalized checkout is another good candidate for the conversation scope since it will typically be a multi-step wizard.

Another way to think about the conversation scope is that it is a truncated custom session with the developer programmatically determining where the scope begins and ends. The concept of conversations will become even clearer as we look at a concrete example with code.

A Conversational Example

To borrow a convenient example from *EJB 3 in Action*, the bidder account creation wizard in the eBay-like *ActionBazaar* makes another great candidate use-case for the conversation scope. As shown in Figure 1, the wizard is composed of fours steps. In the first step, the user will enter login information such a username, password, password confirmation, secret question/answer, etc. When the user clicks the "Next" button, the information in the first step is saved and the user is taken to the second step. In the second step, user details such as the first name, last name, address, email and contact information is collected and saved. The wizard also allows the user to backtrack to the previous step and change the previously entered information.



Similarly, the third step collects user preferences such as notification and display preferences. The final step of the wizard confirms all the collected bidder account information before actually creating the account. The first step of the wizard starts the conversation while the conversation scope should end when the account is created in the last step of the wizard.

The entire account creation wizard can be implemented through a single conversation scoped bean. Before we look at how the bean is implemented, let's take a brief look at the relatively unsophisticated JSF Facelets for the wizard. Figure 2 shows the JSF code that corresponds to each of the pages in Figure 1.

Step 1: enter_login.jsf

```
<h:outputLabel>Username:</h:outputLabel>
<h:inputText id="username"
   value="#{login.username}"/>
<h:outputLabel>Password:</h:outputLabel>
<h:inputSecret id="password"
   value="#{login.password}"/>
...
<h:selectBooleanCheckbox id="expiration"
   value="#{login.expiration}"/>
<h:outputLabel>Password expires</h:outputLabel>
...
<h:commandButton value="Next>"
   action="#{accountCreator.saveLogin}"/>
```

Step 2: enter_user.jsf

```
<h:outputLabel>First Name:</h:outputLabel>
<h:inputText id="firstName"
   value="#{user.firstName}"/>
<h:outputLabel>Last Name:</h:outputLabel>
<h:inputSecret id="lastName"
   value="#{user.lastName}"/>
...
<h:outputLabel>Home Phone:</h:outputLabel>
<h:inputSecret id="homePhone"
   value="#{user.homePhone}"/>
...
<h:commandButton value="<Previous"
   action="enter_login.jsf"/>
...
<h:commandButton value="Next>"
   action="#{accountCreator.saveUser}"/>
```

Step 3: enter_preferences.jsf

```
<h:selectBooleanCheckbox id="notifyOutbid"
   value="#{preferences.notifyOutbid}"/>
<h:outputLabel>Notify outbid</h:outputLabel>
<h:selectBooleanCheckbox id="notifyBidEnd"
   value="#{preferences.notifyBidEnd}"/>
<h:outputLabel>Notify bid end</h:outputLabel>
...
<h:selectBooleanCheckbox id="suggestAlternatives"
   value="#{preferences.suggestAlternatives}"/>
<h:outputLabel>Suggest alternatives</h:outputLabel>
...
<h:commandButton value="<Previous"
   action="enter_user.jsf"/>
...
<h:commandButton value="Next>"
   action="#{accountCreator.savePreferences}"/>
```

Step 4: confirm_account.jsf

```
<h:outputLabel>Username:</h:outputLabel>
<h:outputText id="username"
   value="#{login.username}"/>
...
<h:outputLabel>First Name:</h:outputLabel>
<h:outputText id="firstName"
   value="#{user.firstName}"/>
...
<h:commandButton value="<Previous"
   action="enter_preferences.jsf"/>
...
<h:commandButton value="Create"
   action="#{accountCreator.createAccount}"/>
```

The enter_login.jsf page implements the first page of the wizard (in case of Facelets the actual source code file name is likely enter_login.xhtml). The login input is bound to a bean named login while the “Next” button is bound to accountCreator.saveLogin. As we will see shortly, the accountCreator bean is a conversation scoped bean that models the workflow. The login bean, on the other hand, is a simple data holder backing bean produced by the accountCreator bean. The second page in the workflow, enter_user.jsf similarly uses a produced backing bean named user and the “Next” button is bound to accountCreator.saveUser. The “Previous” button maps directly to the first page in the wizard. The enter_preferences.jsf page is implemented in a very similar fashion. The final page in the wizard, confirm_account.jsf, displays the values collected by the wizard during the conversation and also binds the event handler that triggers the actual creation of the account and the end of the workflow, accountCreator.createAccount.

Let’s now take a close look at the conversation scoped bean that implements the wizard:

```
@Named
@ConversationScoped

public class AccountCreator {

  @Inject
  private AccountService accountService;

  @Inject
  private Conversation conversation;

  @Named
 @Produces
  public Login login = new Login();

  @Named
  @Produces

  public User user = new User();

  @Named
  @Produces
```

```
  public Preferences preferences = new Preferences();

  public String saveLogin() {
   conversation.begin();
   ...
   return “enter_user.jsf”;
  }

  public String saveUser() {
   ...
   return “enter_preferences.jsf”;
  }

  public String savePreferences() {
   ...
   return “confirm_account.jsf”;
  }

  public String createAccount() {
   Account account = new Account();
   account.setLogin(login);
   account.setUser(user);
   account.setPreferences(preferences);
   accountService.createAccount(account);
   conversation.end();
   return “/home.jsf”
  }

}
```

The bean is annotated to be both @Named so that it can be referenced from EL as well as @ConversationScoped. The login, user and preferences fields produce named backing beans for use in the wizard pages. Because these beans are produced by the conversational bean, they are available throughout the conversation as well as being available to the parent bean holding the bean instances. It is very important to note that a handle to the Conversation itself is injected into the bean. As we will discuss in greater detail both in this section as well as the next section, the Conversation interface allows programmatic control over the life-cycle of the conversation scope. This interface is basically what allows you to “custom-fit” the conversation to your application. A back-end service to actually create the account is also injected and is presumably implemented as a transactional stateless session bean. The JSF event listener methods in AccountCreator is actually where most of the interesting stuff is going on. The saveLogin method is called on the first page of the wizard and actually starts the *long-running conversation*. To understand what that means, you’ll have to know about types of conversations in CDI.

CDI has two different types of conversations, *transient* and *long-running*. By default, when you annotate a bean with @ConversationScoped, it is assumed to be transient. A transient conversation ends when the request that originated the conversation ends. This is a sensible fail-safe in case a conversational bean does not really need to be extended beyond the request. Any transient conversation can be turned into a long-running conversation on demand. Unlike a transient conversation, a long-running conversation extends beyond the scope of a request, potentially as long as the whole application session. A transient conversation is turned into a long-running conversation by invoking the Conversation.begin method, as is done in the saveLogin method. If the saveLogin method is not invoked, for example if the user abandoned the wizard at the first step, the bean will never be put into a long-running conversation and will simply be disposed of at the end of the request as part of the transient conversation. Besides starting the long running conversation, a number of other things can possibly be done in the saveLogin method, including perhaps validating that the username does not already exist, the password matches the confirmed password or that the password meets security guidelines. The saveLogin method also moves the wizard to the next page by returning the “enter_user.jsf” URL as the outcome for the event.

No specific manipulation of the conversation is done in the next page of the wizard, except for binding more input values to the user produced bean and the saveUser event handler likely only does some form-level validation before forwarding to the enter_preferences.jsf page. The savePreferences event handler is similarly simplistic. The event handler method for the final page in the wizard, createAccount, does a number of interesting things, however. It actually creates the final Account object with all the input collected into the produced fields throughout the conversation and invokes the back-end service to save the newly created account into the database. It then invokes the Conversation.end method. As you can guess, the Conversation.end method ends the long running conversation. This does not mean however that the conversation is immediately destroyed; it simply means that the conversation is “demoted” to becoming transient again. This allows for the conversational bean to be destroyed when the request ends and the user is moved out of the wizard into the “/home.jsf” URL.

An interesting question to think about is what happens if the user abandons the wizard in the middle after the long-running conversation is started. The bean will of course not be destroyed when a request ends. Like sessions, long-running conversations have an implicit timeout. When this timeout value expires, the conversation is destroyed. This timeout value is typically shorter than a session timeout. An astute reader might also wonder what happens if the saveLogin method (and therefore the Conversation.begin) is invoked twice during the same

conversation, or if the Conversation.end method is invoked while the conversation is still transient. In most real applications, the begin and end methods should always be invoked after checking the current state of the conversation using the other methods in the Conversation interface described in detail below.

Programmatic vs. Declarative Conversations

It is an interesting question to ask whether the programmatic model of injecting the conversation and calling the begin and end methods could be converted to a declarative equivalent. For example, instead of injecting the conversation, we could have used something like @Begin and @End on the saveLogin and createAccount methods.

While this is not currently supported in CDI, it is the model that was supported in previous versions of Seam and will likely still be supported in Seam 3. If you believe it is useful, this is something we can support in Resin 4 as well.

More on Conversations

In order to make effective use of the conversation scope, it is helpful to understand a little bit of how it is implemented under-the-hood. CDI keeps track of a long running conversation by propagating an HTTP GET parameter named cid (reserved by the specification) across requests that are part of a workflow. The ID is created when the conversation starts and the cid that is passed around from request-to-request is mapped to the correct conversation context at runtime on subsequent requests in the workflow. When CDI cannot find a cid in the request, it assumes that a new conversation should be started. The ID itself is usually automatically generated, but you can create it manually yourself and retrieve it when needed (see the table below). As matter of fact, the cid can even be propagated back and forth from JSF and non-JSF pages (such as Servlets that are aware of CDI).

Below are all the methods that are supported by the Conversation interface:

Method	Description
void begin()	The method promotes the conversation to being long-running. If the conversation is already long-running, an IllegalStateException is thrown. When the conversation is promoted, CDI automatically generates a unique ID and assigns it to the conversation – this is the ID that is used in the cid parameter.
void begin(String id)	This variant of the begin method allows you to provide an application defined ID for the conversation. You may want to do this, for example, if you wanted to track the conversations for your application by assigning some custom meaning to it.
void end()	This method demotes a long-running conversation to become transient. If the conversation is not long-running, an IllegalStateException is thrown.
String getId()	This method returns the identifier of the current long-running conversation, or a null value if the current conversation is transient. The method can be used to send the conversation ID to a non-JSF page by embedding it into a URL or hidden form value, for example.
long getTimeout()	This method returns the timeout, in milliseconds, of the current conversation.
void setTimeout(long milliseconds)	This method sets the timeout of the current conversation. The method could be used as a performance tuning measure or for customizing application behavior (for example, setting idle time for an on-line quiz).

boolean isTransient()	This method indicates whether the current conversation is transient. It should be used to check conversation state before invoking the begin and end methods.

It is also important to understand that you can start multiple conversations in the same session. For example, you could open two instances of the enter_login.jsf page in separate tabs. Two different conversations with two different conversation IDs would result. Conversations are thread-safe, meaning that even if you started two concurrent requests, two different conversations would still result.

For further details on the conversation scope such as conversation passivation, feel free to look through the CDI specification (or the Weld reference guide referenced below).

More to Come

Although we have discussed CDI's interaction with JSF in somewhat greater detail in this article of the series, JSF's interaction with CDI deserves focused coverage. In the next article of the series, we will focus solely on CDI from a JSF developer's perspective including CDI's interaction with JSF using EL binding, scoping, producers, qualifiers, events 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. 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://openwebbeans.apache.org.

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.

Books on EJB 3 and EJB 3.1 Development

Dependency Injection in Java EE 6: Part I
Dependency Injection in Java EE 6: Part II
Dependency Injection in Java EE 6: Part III