Open Source

Topics in This Chapter

Chapter 12

When Sun first conceived of JSF, it counted on a vibrant open source community to help drive innovation. Although it took longer than expected, that is exactly what happened with the advent of projects such as Apache Shale, JBoss Seam, and Facelets. Those open source projects—along with other projects based on JSF, such as AjaxFaces—not only provide immediate benefit for JSF developers, but are also shaping JSF's future.

In this chapter, we take a look at three significant innovations:

- Web flow
- Alternate view technologies
- EJB (Enterprise JavaBeans) integration

Web flow is an industrial-strength version of JSF's default navigation mechanism. With web flow you can easily define complicated user interactions.

Starting with JSF 1.2 and JSP 2.1, the differences that caused incompatibilities between the two have now been banished to the dustbin, and you should be able to use them together without difficulty. However, even with that idyllic scenario, there is still a considerable segment of the JSF developer community that would like to replace JSP altogether with a lightweight templating mechanism. Luckily, JSF was built to accomodate just such a scenario.

If you have not taken a look at EJB 3.0, you may be surprised at just how much this most-maligned of specifications has matured to become a viable solution

for many developers. Alas, the EJB and JSF component models are incompatible, which opens the door for frameworks that provide a unified component model.

Next, we take a look at these three innovations through the lens of three open source frameworks that implement them: Shale, Facelets, and Seam.

Web Flow—Shale

From the folks that brought you Struts comes Shale, a set of services layered on top of JSF. Shale has lots of features that make everyday JSF development much easier:

- Web flow
- Remote method calls for Ajax
- Templating and Tapestry-like views
- Client- and server-side validation with Apache Commons Validator
- Testing framework with support for unit and integration testing
- Spring, JNDI, and Tiles integration
- View controllers (concrete implementation of the JSF backing bean concept)

Shale web flow lets you implement a series of interactions between a user and your application. That set of interactions is more commonly called a *user conversation*, *dialog*, or *wizard*. Shale uses the term *dialog*, so that is the term we will use here.

A Shale dialog consists of one or more states. These states have transitions that define how control is transferred from one state to another. Dialogs also have a special state, called the *end state*, that exits the dialog and releases dialog state. Here is an example of a Shale dialog definition:

```
<dialogs>
   <dialog name="Example Dialog"
      start="Starting State">
      ...
      <view name="Starting State"
         viewId="/viewForThisState.jsp">
         <transition outcome="next"
            target="The Next State"/>
         <transition outcome="cancel"
            target="Exit"/>
      </view>
```

```
        <view name="The Next State"
           viewId="/nextView.jsp">
           <transition outcome="next"
              target="Yet Another State"/>
           <transition outcome="cancel"
              target="Exit"/>
        </view>
        ...
        <end name="Exit"/>
     </dialog>
  </dialogs>
```

In the preceding code fragment, we defined—in an XML configuration file—a dialog named `Example Dialog` with three states: `Starting State`, `The Next State`, and `Exit`. The transitions define how Shale navigates from one state to another—for example, in a JSP page, you can do this:

```
<h:commandButton id="next"
   value="#{msgs.nextButtonText}"
   action="next"/>
```

If the preceding dialog's state is `The Next State` and you click the button, Shale uses that outcome—`next`—to send you to the next state, in this case named `Yet Another State`. Because of `The Next State`'s second transition, clicking a button whose action is `cancel` will end the dialog.

That contrived example shows the basics of Shale dialogs, but Figure 12–1 illustrates a more realistic example of a bill pay wizard. The wizard lets you make an online payment and is composed of four steps: Payee Information, Payment Method, Payment Schedule, and Summary. Each of those steps is shown in Figure 12–1, from top to bottom, respectively.

But the bill pay wizard has a twist. If you select "Wire Transfer" for your payment method and click the "Next" button, you do not go directly to the "Payment Schedule" panel, as Figure 12–1 leads you to believe. Instead, a subdialog intervenes that collects the wire transfer information.

That sequence of events is shown in Figure 12–2. Notice that the top and bottom pictures in that figure are from the surrounding dialog, whereas the middle pictures are from the wire transfer subdialog.

So there is our finished application, with two dialogs, one nested in the other. Now we take a look at the key steps in implementing those dialogs.

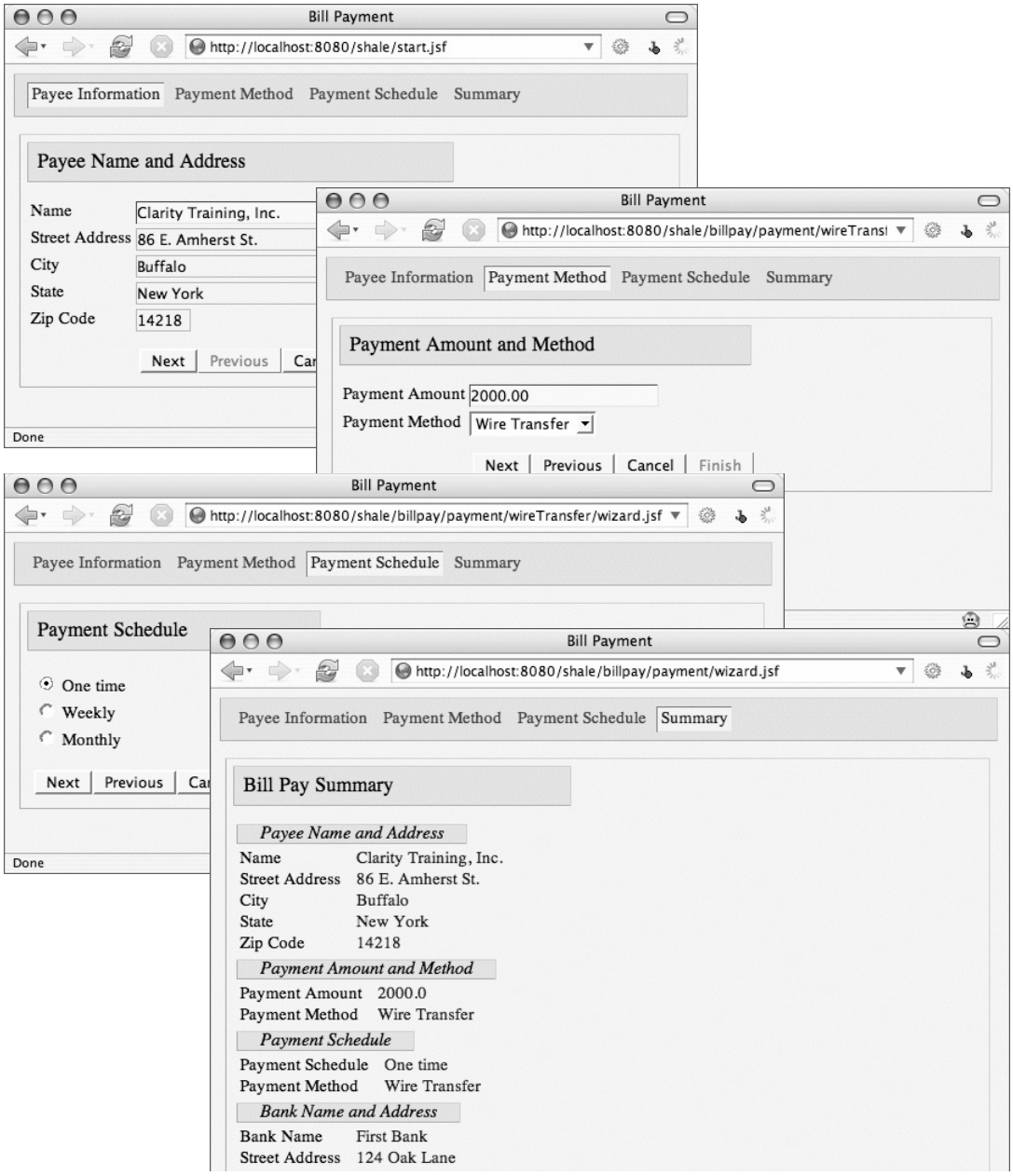

Figure 12–1 The bill pay wizard

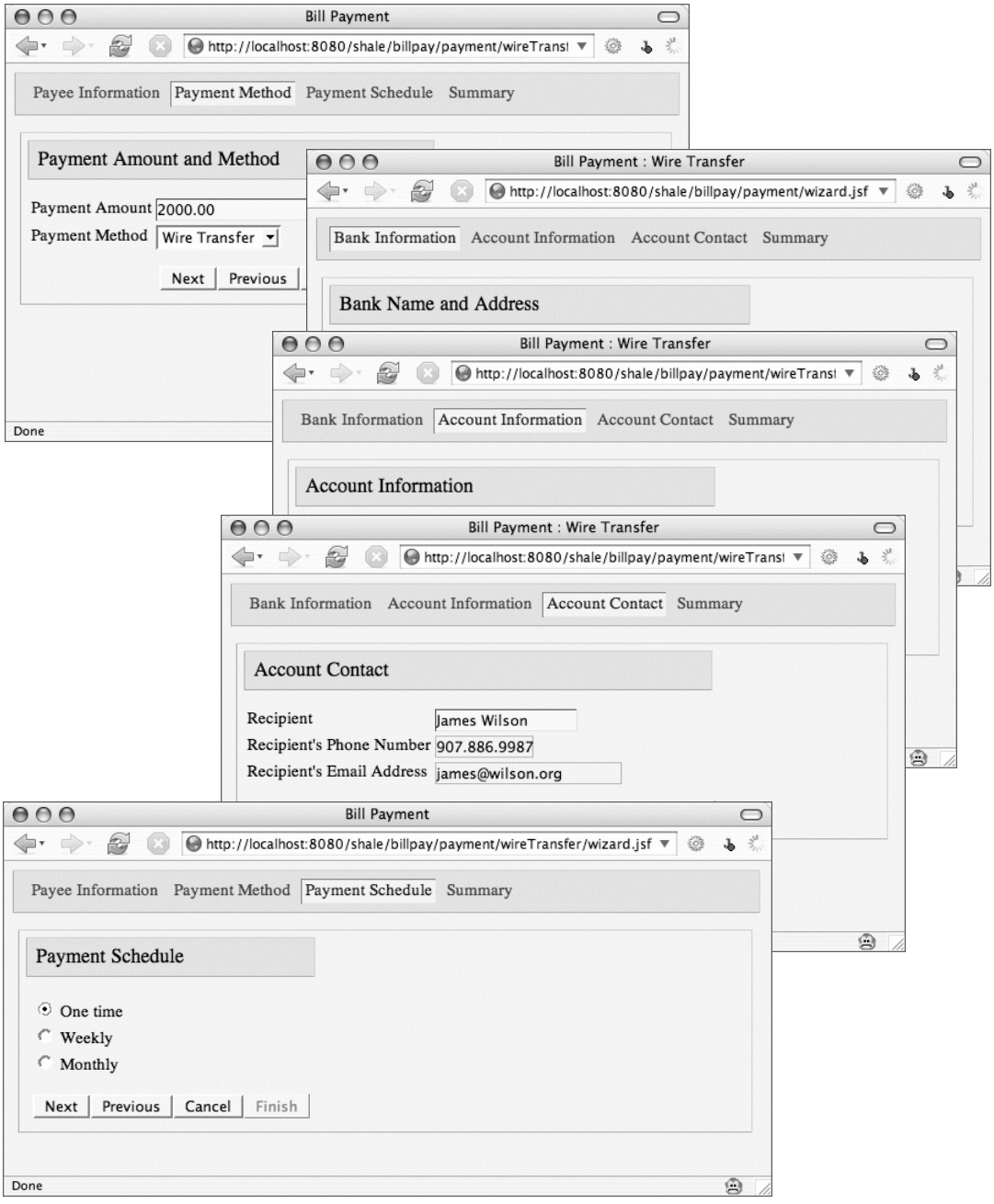

Figure 12–2 Wire transfer subdialog (summary panel not shown)

Dialog Configuration

By default, you define your dialogs in an XML file named `/WEB-INF/dialog-config.xml`. We have two dialogs, so we have chosen to use two configuration files, which we declare in the deployment descriptor:

```
<!-- this is WEB-INF/web.xml -->
<web-app>
   <context-param>
      <param-name>org.apache.shale.dialog.CONFIGURATION</param-name>
      <param-value>/WEB-INF/dialogs/payment.xml,
         /WEB-INF/dialogs/wire-transfer.xml</param-value>
   </context-param>
   ...
</web-app>
```

If you use a single file, named `/WEB-INF/dialog-config.xml`, you do not need to declare it in your deployment descriptor.

Entering a Dialog

The next order of business is entering the dialog. In our example, we use a link, as shown in Figure 12–3.

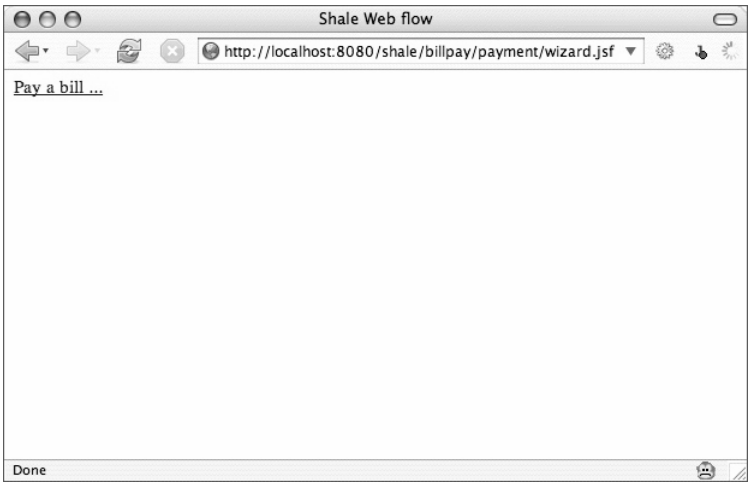

Figure 12–3 Entering the bill payment dialog

The code for that link looks like this:

```
<h:commandLink action="dialog:Payment">
   <h:outputText value="#{msgs.billpayPrompt}"/>
</h:commandLink>
```

As you might suspect, Shale places special meaning on any action that starts with the string `dialog:`. The name that follows the colon represents the name of a dialog, which Shale subsequently enters when a user clicks the link.

Dialog Navigation

By default, Shale dialogs are defined in XML. Here is how the `Payment` dialog, referenced in the preceding code fragment, is defined:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE dialogs PUBLIC
   "-//Apache Software Foundation//DTD Shale Dialog Configuration 1.0//EN"
   "http://struts.apache.org/dtds/shale-dialog-config-1_0.dtd">

<dialogs>
   <dialog name="Payment"
      start="Setup">
      ...
      <action name="Setup"
         method="#{dialogLauncher.setupPaymentDialog}">
         <transition outcome="success"
            target="Payee Information"/>
      </action>

      <!-- Payee Information -->
      <view name="Payee Information"
         viewId="/billpay/payment/wizard.jsp">
         <transition outcome="next"
            target="Payment Method"/>
      </view>
      ...
```

When you enter a dialog, Shale loads the state specified with the `dialog` element's `start` attribute. In our case, that state is `Setup`, which is an *action state*. Action states execute a method and immediately transition to another state, depending upon the string returned from the method (the action's outcome).

When Shale enters our dialog's `Setup` state, it invokes the `setupPaymentDialog` method on a managed bean named `dialogLauncher`. That method stores some objects in dialog scope (see "Dialog Scope" on page 578 for more information about that method) and returns the string `success`.

That `success` outcome causes Shale to load the `Payee Information` state, which is a *view state*. View states load a view, specified with the `viewId` attribute, and wait for the user to click a button or a link, which generates an outcome that Shale uses to navigate to the next state.

So, to summarize, when you click the link to enter the dialog, Shale loads the `Setup` action state, which invokes `dialogLauncher.setupPaymentDialog()`. That method returns `success`, which causes Shale to load `/billpay/payment/wizard.jsp` and wait for the next outcome.

At some point, our dialog ends with the `end` state:

```
        ...
        <end  name="Exit" viewId="/start.jsp"/>
    </dialog>
</dialogs>
```

The `end` state is a special view state that exits the current dialog and subsequently loads the view specified with the `viewId` attribute.

Dialog Scope

Throughout the JSP pages in our wizards, we have input fields wired to an object that is stored as the current dialog's `data` object—for example:

```
<h:inputText id="name"
   size="30"
   value="#{dialog.data.name}"
   ...
   styleClass="input"/>
```

When you enter a dialog, Shale puts an object, named `dialog`, in session scope and when you exit the dialog, Shale removes the `dialog` object from the session, thereby effectively creating a dialog scope.

You can store an object of your choosing in the `dialog` object by setting the dialog's `data` property. In the preceding code, we access that `data` object to wire a text field to the `data` object's `name` property. As is often the case, our `data` object is a simple collection of properties representing the fields in the wizard's panels.

This all begs a question: How does our `data` object get associated with the dialog's `data` property? Like the `Payment` dialog discussed in "Dialog Navigation" on page 577, the `Wire Transfer` dialog also has a `Setup` action state that Shale executes when you enter the dialog. That method stores the `data` object in the dialog's `data` property. Here is how we declare the `Setup` method:

```
<dialogs>
   <dialog name="Wire Transfer Dialog"
      start="Setup">
      ...
      <action name="Setup"
         method="#{dialogLauncher.setupWireTransferDialog}">
```

```
            <transition outcome="success"
               target="Bank Information"/>
         </action>
         ...
         <end name="Exit"/>
      </dialog>
   </dialogs>
```

Shale executes `dialogLauncher.setupWireTransferDialog()` when you enter the `Wire Transfer` dialog. So we have two methods, `setupPaymentDialog` and `setupWireTransferDialog`, that Shale executes immediately after entering the `Payment` and `Wire Transfer` dialogs, respectively. Those methods look like this:

```
public class DialogLauncher extends AbstractFacesBean {
    private BillpayData billpayData = null;

    // Called just afer entering the payment dialog
    public String setupPaymentDialog() {
       billpayData = new BillpayData();
       billpayData.setTransfer(new WireTransferData());

       setValue("#{dialog.data}", billpayData);
       return "success";
    }

    // Called just afer entering the wire transfer dialog
    public String setupWireTransferDialog() {
       setValue("#{dialog.data}", billpayData.getTransfer());
       return "success";
    }
}
```

The `setupPaymentDialog` method creates two objects, that combined, contain all the properties on all the panels for the `Payment` and `Wire Transfer` dialogs. Because the `Wire Transfer` dialog is a subdialog of the `Payment` dialog, we likewise store the wire transfer `data` object in the payment `data` object.

While a dialog is active, the `dialog` object and its associated `data` object, are available via JSF expressions like `#{dialog.data.someProperty}`. When the dialog exits, Shale removes the `dialog` object from "dialog" scope and it is no longer accessible via JSF expressions. It is interesting to note that the `data` object for a dialog could easily be a map, which effectively gives you a bona fide new scope.

NOTE: Notice that the `DialogLauncher` class in the preceding code extends Shale's `AbstractFacesBean`. That class has a number of handy methods, some of which are listed below, that you will find useful for JSF development in general. For example, `DialogLauncher` uses `AbstractFacesBean.setValue()`, which sets a bean property's value to a given string, which can be a value expression.

org.apache.shale.faces.AbstractFacesBean

- `Object getBean(String beanName)`
  Returns a managed bean with the specified `beanName`, if one exists. This method delegates to the JSF variable resolver, which, by default, searches request, session, and application scope—in that order—for managed beans. If no managed bean is found with the given name, the variable resolver looks for a managed bean definition for the bean in question; if the definition exists, JSF creates the bean. If the bean does not exist and has no definition, this method returns `null`.
- `Object getValue(String expr)`
  Given a value expression, this method returns the corresponding object. For example: `LoginPage page = (LoginPage)getValue("#{loginPage}");`
- `void setValue(String expr, Object value)`
  This method is the inverse of `getValue`; it sets a value, given a value expression; for example: `setValue("#{loginPage}", new LoginPage());`

Dialog Context Sensitivity

Look closely at the tabs and buttons for the dialog panels in Figure 12–1 on page 574 and Figure 12–2 on page 575, and you will see that they are context sensitive. That sensitivity is implemented in a `page` object and accessed in JSF expressions. Here is how the sensitivity of the wizard buttons is controlled:

```
<h:commandButton id="next"
   value="#{msgs.nextButtonText}"
   action="next"
   styleClass="wizardButton"
   disabled="#{not dialog.data.page.nextButtonEnabled}"/>
```

If the `page` object stored in the dialog's `data` returns `true` from `isNextButtonEnabled()`, JSF enables the button. Similar properties, such as `previousButtonEnabled`, are contained in the `page` object and accessed in `wizardButtons.jsp`. Here is how the CSS styles for the tabs at the top of each wizard panel are set:

```
<h:panelGrid columns="5">
   <h:outputText value="#{msgs.payeeTabPrompt}"
      styleClass="#{dialog.data.page.payeeStyle}"/>
</h:panelGrid>
```

The page object accesses dialog state programatically. In our application, we implemented a base class that encapsulates the basics:

```
public class BaseDialogPage {
   protected BaseDialogPage() {
       // Base class only...
   }
   protected Status getDialogStatus() {
      Map sessionMap = FacesContext.getCurrentInstance()
         .getExternalContext()
         .getSessionMap();
      return(Status)sessionMap.get(org.apache.shale.dialog.Globals.STATUS);
   }
   protected boolean stateEquals(String stateName) {
      return stateName.equals(getDialogStatus().getStateName());
   }
   protected boolean isStateOneOf(String[] these) {
      String state = getDialogStatus().getStateName();
      for (int i=0; i < these.length; ++i) {
         if(state.equals(these[i]))
         return true;
      }
      return false;
   }
}
```

Shale stores a status object in session scope, which we retrieve in the preceding code. From that status object we can determine the current state. We subsequently put those base class methods to good use in subclasses. Here is the page class for the Payment dialog:

```
public class BillpayPage extends BaseDialogPage {
   // State constants
   private static final String PAYEE_INFORMATION = "Payee Information";
   private static final String PAYMENT_METHOD = "Payment Method";
   private static final String PAYMENT_SCHEDULE = "Payment Schedule";
   private static final String SUMMARY = "Summary";

   public String enterPaymentDialog() {
      return "dialog:Payment";
   }

   // View logic for panels:
```

```
public boolean isPayeeRendered() {
   return stateEquals(PAYEE_INFORMATION);
}
public boolean isPaymentMethodRendered() {
   return stateEquals(PAYMENT_METHOD);
}
public boolean isPaymentScheduleRendered() {
   return stateEquals(PAYMENT_SCHEDULE);
}
public boolean isSummaryRendered() {
   return stateEquals(SUMMARY);
}

// View logic for buttons:

public boolean isNextButtonEnabled() {
   return isStateOneOf(new String[] {
      PAYEE_INFORMATION, PAYMENT_METHOD, PAYMENT_SCHEDULE });
}
public boolean isPreviousButtonEnabled() {
   return isStateOneOf(new String[] {
      PAYMENT_METHOD, PAYMENT_SCHEDULE, SUMMARY });
}
public boolean isCancelButtonEnabled() {
   return true;
}
public boolean isFinishButtonEnabled() {
   return stateEquals(SUMMARY);
}

// View logic for CSS style names

public String getPayeeStyle() {
   return isPayeeRendered() ?
      "selectedHeading" : "unselectedHeading";
}

public String getPaymentMethodStyle() {
   return isPaymentMethodRendered() ?
      "selectedHeading" : "unselectedHeading";
}

public String getPaymentScheduleStyle() {
   return isPaymentScheduleRendered() ?
      "selectedHeading" : "unselectedHeading";
}
```

```
    public String getSummaryStyle() {
       return isSummaryRendered() ?
          "selectedHeading" : "unselectedHeading";
    }
}
```

The preceding class controls the visibility of panels, enabled state of buttons, and CSS tab styles, all by programatically determining the current dialog state.

Subdialogs

In our example, the `Wire Transfer` dialog is a subdialog of the `Payment` dialog. Here is how that is defined:

```
<dialogs>
   <dialog name="Payment"
      start="Setup">
      ...
      <!-- The following action navigates from the Payment
         Method page depending upon the payment method
         that the user selected from a drop-down list.
         The action simply returns that value. -->
      <action name="Navigate Based on Transfer Mechanism"
         method="#{dialog.data.navigateTransfer}">
         <transition outcome="Wire Transfer"
            target="Wire Transfer"/>
      </action>

      <subdialog name="Wire Transfer"
         dialogName="Wire Transfer Dialog">
         <transition outcome="cancel"
            target="Payment Method"/>
         <transition outcome="success"
            target="Payment Schedule"/>
      </subdialog>
      ...
   </dialog>
</dialogs>
```

Inside the `Payment` dialog we declare a `Wire Transfer` subdialog. We navigate to that subdialog through an action state named `Navigate Based on Transfer Mechanism` that returns the string selected from a drop-down list of transfer types. If that string is `Wire Transfer`, Shale navigates to the `Wire Transfer` subdialog.

Notice the transitions for that subdialog: If those outcomes (cancel and success) are not handled in the subdialog, Shale associates them with the states `Payment Method` and `Payment Schedule`, respectively.

The `Wire Transfer` subdialog is a dialog, defined like any other:

```
<dialogs>
   <dialog name="Wire Transfer Dialog"
      start="Setup">

      <action name="Setup"
         method="#{dialogLauncher.setupWireTransferDialog}">
         <transition outcome="success"
            target="Bank Information"/>
      </action>

      <view name="Bank Information"
         viewId="/billpay/payment/wireTransfer/wizard.jsp">
         <transition outcome="next"
            target="Account Information"/>
         <transition outcome="cancel"
            target="Exit"/>
      </view>

      <view name="Account Information"
         viewId="/billpay/payment/wireTransfer/wizard.jsp">
         <transition outcome="previous"
            target="Bank Information"/>
         <transition outcome="next"
            target="Account Contact"/>
         <transition outcome="cancel"
            target="Exit"/>
      </view>

      <view name="Account Contact"
         viewId="/billpay/payment/wireTransfer/wizard.jsp">
         <transition outcome="previous"
            target="Account Information"/>
         <transition outcome="next"
            target="Summary"/>
         <transition outcome="cancel"
            target="Exit"/>
      </view>

      <view name="Summary"
         viewId="/billpay/payment/wireTransfer/wizard.jsp">
         <transition outcome="previous"
            target="Account Contact"/>
         <transition outcome="finish"
            target="Finish"/>
         <transition outcome="cancel"
```

```
            target="Exit"/>
        </view>

        <action name="Finish"
            method="#{dialog.data.finish}">
            <transition outcome="success"
                target="Exit"/>
        </action>

        <end name="Exit"/>
    </dialog>
</dialogs>
```

Shale's dialog support is a powerful upgrade to JSF's built-in navigation capabilities. By itself, it is a compelling inducement to add Shale to your tool chest.

Alternate View Technologies—Facelets

In the early days of enterprise Java, developers dealt in HTML directly, by emitting HTML from servlets. Over the years, we have moved to eradicate HTML from our views with the advent of JSP and, especially, JSP custom tags, which do a neat job of encapsulating Java code and removing it from JSP pages.

Everything is a trade-off, but overall the trend toward JSP tags has made JSP pages more readable, maintainable, and extensible.

However, JSP has a dark side: When software developers and graphic designers work independently, the JSP model breaks down badly. Graphic designers are typically not familiar with JSP or the set of custom JSP tags that software developers use on any given project.

Software developers, on the other hand, can have a devil of a time incorporating a look and feel into a web application laden with custom tags that ultimately generate HTML, as is the case for a typical JSF application. If JSP is used on a project in which software developers and graphic designers work separately, only rigid discipline provides any hope of success. Fortunately, there is an alternative.

XHTML Views

Now we cut to the chase. We are going to use Facelets, an open source display technology for JSF, to implement our views in XHTML (Extensible HTML) instead of JSP. In our XHTML files, we will have mock-up HTML that is edited by a graphic designer, *but at runtime, Facelets will swap out that mock-up HTML with JSF components*. This will allow graphic designers to create a look and feel

with mock-up markup that developers can replace with JSF components at runtime. And here is the clincher: *The JSF components absorb the mock-up's look and feel*.

Feel free to take a moment to ponder the ramifications.

This sort of chicanery, first pioneered in the Java web application space by Tapestry, truly borders on magic. Now we see how it works. First, an XHTML page, shown in Figure 12–4.

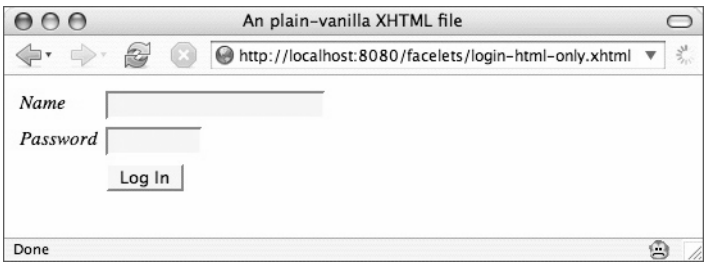

Figure 12–4 A simple XHTML page

Here is the listing for that page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
        <link href="styles.css" rel="stylesheet" type="text/css"/>
        <title>An plain-vanilla XHTML file</title>
   </head>

   <body>
      <form id="login">
         <table cellpadding="2px;">
            <tr>
               <td>
                  <label for="name" class="label">
                     Name
                  </label>
               </td>
               <td>
                  <input type="text"
                     id="name"
                     style="background: #ffa"/>
               </td>
            </tr>
```

```
            <tr>
               <td>
                  <label for="password" class="label">
                     Password
                  </label>
               </td>
               <td>
                  <input type="password"
                     id="password"
                     size="8"
                     class="input"/>
               </td>
            </tr>
            <tr>
               <td></td>
               <td>
                  <input type="submit"
                     value="Log In"/>
               </td>
            </tr>
         </table>
      </form>
   </body>
</html>
```

You cannot get much more vanilla than that. Next we wire those HTML elements to JSF components.

Replacing Markup with JSF Components: The jsfc Attribute

We are going to use JSF components to turn our nonfunctional markup into a thriving web page, as shown in Figure 12–5.

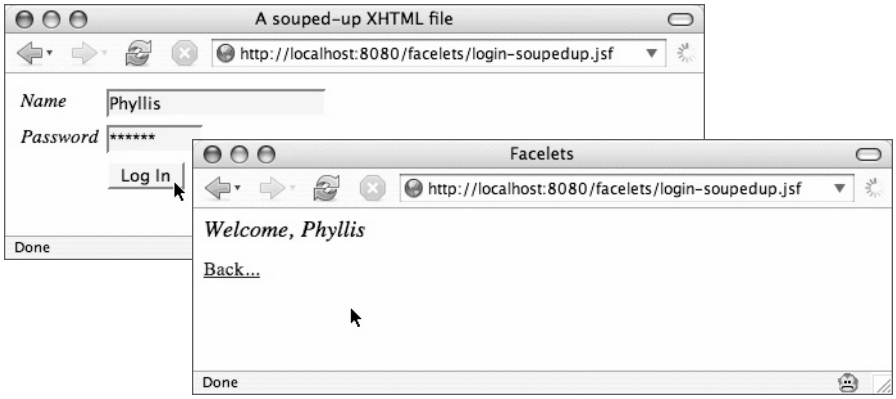

Figure 12–5 Replacing XHTML markup at runtime with JSF components

Here is the transformation of our XHTML file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:f="http://java.sun.com/jsf/core">

   <head>
      <link href="styles.css" rel="stylesheet" type="text/css"/>
      <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
      <title>#{msgs.windowTitle}</title>
   </head>

   <body>
      <form jsfc="h:form" id="login">
         <table cellpadding="2px;">
            <tr>
               <td>
                  <label for="name"
                     class="label"
                     jsfc="h:outputLabel">
                     #{msgs.namePrompt}
                  </label>
               </td>
               <td>
                  <input type="text"
                     id="name"
                     value="#{user.name}"
                     style="background: #ffa;"
                     jsfc="h:inputText"/>
               </td>
            </tr>
            <tr>
               <td>
                  <label for="password"
                     class="label"
                     jsfc="h:outputLabel">
                     #{msgs.passwordPrompt}
                  </label>
               </td>
               <td>
                  <input type="password"
                     jsfc="h:inputSecret"
                     id="password"
                     value="#{user.password}"
                     size="8"
                     class="input"/>
```

```
            </td>
          </tr>
          <tr>
            <td> </td>
            <td>
              <input type="submit"
                jsfc="h:commandButton"
                value="#{msgs.loginButtonText}"
                action="login"/>
            </td>
          </tr>
        </table>
      </form>
    </body>
  </html>
```

Three things about this example:

- Notice the address bar URLs in Figure 12–4 on page 586 and Figure 12–5 on page 587. The former URLs end in `.xhtml`, whereas the latter end in `.jsf`. This means that Figure 12–4 shows what the graphic designer sees and Figure 12–5 shows what software developers, and ultimately end users, see.
- We use an XML namespace to enable Facelet tags that mimic the JSF core tag library. Subsequently, we use the `f:loadBundle` tag from that namespace to load a resource bundle. Finally, we use value expressions, such as `#{msgs.namePrompt}`, directly in the page—no `h:outputText` is required—to access keys in the resource bundle.
- We have added `jsfc` attributes to our form, labels, text fields, and button. When you run this souped-up XHTML page through Facelets, it swaps the markup for components of the specified type.

You might wonder what the graphic designer sees when she views this souped-up XHTML file. Figure 12–6 provides the answer.

The browser does not know what to do with JSF value expressions. So it just uses them as is, which is convenient for translators, who can glean the context of a translation just by looking at the XHTML.

NOTE: What does the browser, or any other tool that lets you view XHTML, do with those `jsfc` attributes? Nothing! So graphic designers can continue to iterate over your XHTML files to refine the look and feel, while you work on the components that will ultimately be used at runtime.

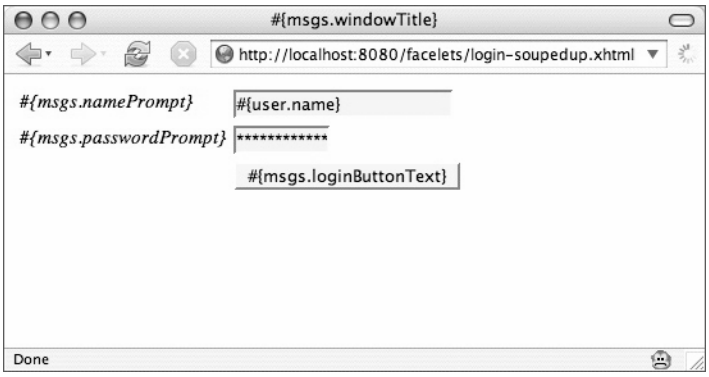

Figure 12–6 What the graphic designer sees

NOTE: Shale has something similar to Facelets, called Clay. Like Facelets, Clay gives you clean separation of graphic design and software development concerns with HTML views. One advantage Clay has over Facelets is that it works with ill-formed HTML, whereas Facelets requires XHTML.

Using JSF Tags

Fundamentally, Facelets, like Tapestry, cleanly separates graphic design from software development. But if you are both a graphic designer and a software developer, there are still viable reasons to prefer the more traditional JSP-based approach. For example, XHTML is more verbose than JSP and therefore can be harder to maintain over the long run. If you yearn for the more traditional approach with JSP tags, Facelets can easily accomodate. It's just that you are not really using JSP at all.

Facelets recognizes tags that look identical to their JSP counterparts, but it is Facelets, not JSP, that processes the tags. As Facelets parses the XHTML, it creates the resulting component tree.

Figure 12–7 shows the login application from page 587 with Facelet JSF tags.

Now our XHTML page looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html">
```

```
<head>
   <link href="styles.css" rel="stylesheet" type="text/css"/>
   <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
   <title>#{msgs.facesTagsWindowTitle}</title>
</head>

<body>
   <h:form id="login">
      <h:panelGrid columns="2" cellpadding="3px">
         <h:outputLabel for="name"
            value="#{msgs.namePrompt}"/>
         <h:inputText type="text"
            id="name"
            value="#{user.name}"
            class="input"/>

         <h:outputLabel for="password"
            class="label"
            value="#{msgs.passwordPrompt}"/>
         <h:inputSecret
            id="password"
            value="#{user.password}"
            size="8"
            class="input"/>

         <h:outputText value=""/>
         <h:commandButton value="#{msgs.loginButtonText}"
            action="login"/>
      </h:panelGrid>
   </h:form>
</body>
</html>
```

Figure 12–7 A Facelets view that uses JSF tags

Notice that with the preceding code, we have purposely chosen to forego the `jsfc` approach, which, as we have seen, lets software developers and graphic designers work separately, in parallel.

In the end of course, it is your call whether to separate graphic design and software development. Either way, Facelets adds some very appealing features to this fundamental replacement of JSF's default display technology.

Page Composition with Templates

Facelets lets you compose web pages from individual XHTML fragments, similar to the popular Tiles framework. Figure 12–8 shows such a web page.

Figure 12–8 Using Facelets composition

Here is the XHTML page shown in Figure 12–8:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets">
   <body>
      <ui:composition template="/layout.xhtml">
         <!-- Header -->
         <ui:define name="header">
```

```
            <img src="images/facelets.jpg"/>
          </ui:define>

          <!-- Menu -->
          <ui:define name="menu">
             <img src="images/whatisit.jpg"/>
             <img src="images/demos.jpg"/>
             <img src="images/downloads.jpg"/>
             <img src="images/bestpractices.jpg"/>
          </ui:define>

          <!-- Content -->
          <ui:define name="content">
          <ui:include src="login-composition.xhtml"/>
        </ui:define>
      </ui:composition>
   </body>
</html>
```

In the preceding code, we define a page *composition* that has a *template*, and we define *content* that is displayed by the template. Since the word "template" is so overloaded, it might help to think of the template as a layout instead. Here is the template for the preceding composition.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:c="http://java.sun.com/jstl/core">

   <link href="styles.css" rel="stylesheet" type="text/css"/>
   <f:loadBundle basename="com.corejsf.messages" var="msgs"/>

   <head>
      <title>Facelets</title>
   </head>

   <body>
      <div class="header">
         <ui:insert name="header"/>
      </div>

      <div class="menu">
         <ui:insert name="menu"/>
      </div>
```

```
        <div class="content">
          <ui:insert name="content"/>
        </div>
      </body>
    </html>
```

Admittedly, there is not much layout there, other than three DIVs. The rest of the layout is encapsulated in CSS for a further separation of concerns. But the result is the same: the template, in this case an XHTML file and its stylesheet, represent the layout for the composition.

Notice the three `ui:define` tags in the code beginning on page 592. The first `ui:define` tag, representing the header of the page, contains a lone image. The next tag, representing the menu, contains four images. The most interesting tag is the third `ui:define` tag, which includes content from another XHTML file with the `ui:include` tag. That XHTML file looks like this:

```
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:c="http://java.sun.com/jstl/core">
   <!-- the contents of this tag are the same as the body of the
      listing on page 586 -->
</ui:composition>
```

The body of the preceding `ui:composition` is the same as the body of the XHTML file that begins on page 586.

NOTE: Why bother to use Facelets composition, when we can just neatly encapsulate the layout and its content in one file? Because if we have multiple views that share layout, we want to define that layout in only one place and reuse it for multiple views, much the same as you include content with JSP's `jsp:include`, or Facelets' `ui:include`. Realize that in the preceding example, any composition that defines `header`, `menu`, and `content` regions can reuse the same layout. That is pretty powerful.

Facelets Custom Tags

Facelets has many more features than we have covered here, so at this point we refer you to the Faclets documentation, but before we do, we will show you one more feature: Facelets custom tags.

You can easily create your own XHTML tags and use them in your Facelets views. For example, Figure 12–9 shows the application discussed in "Page Composition with Templates" on page 592, equipped with a custom tag that

shows the HTTP headers for the current request, but only when there is a request parameter named `debug` whose value is `true`.

A severely abbreviated listing of Figure 12–9 follows:

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:debug="http://corejsf/facelets/debug">
   ...
      <debug:headers/>
   ...
</html>
```

Figure 12–9 A debug tag that is triggered by a request attribute

We must include our namespace declaration and then we are free to use the tag. The tag is defined in an XML file:

```
<facelet-taglib>
   <namespace>http://corejsf/facelets/debug</namespace>
   <tag>
      <tag-name>headers</tag-name>
      <source>tags/corejsf/debug/headers.xhtml</source>
   </tag>
</facelet-taglib>
```

We specify the name of the tag, `headers`, and the file it represents: `tags/corejsf/debug/headers.xhtml`. Here is that file:

```
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:c="http://java.sun.com/jstl/core">
   <c:choose>
      <c:when test="#{not empty param.debug and param.debug == 'true'}">
         <h:outputText value="#{msgs.debugHeaders}"/>
         <p><h:outputText value="#{header}" style="color: red;"/></p>
      </c:when>
   </c:choose>
</ui:composition>
```

That's it. A Facelets custom tag in three easy steps.

EJB Integration—Seam

One of the things that makes developing web applications in Java harder than it should be is a mismatch between user interface and persistence frameworks. The two sides of the enterprise Java coin exist and mature independently, without much collaboration or synergy.

Traditionally, implementing Java-based web applications meant learning two frameworks—one for the user interface (UI) and another for the backend. For example: JSF and Hibernate; Tapestry and EJB3; Webwork and IBATIS, etc. Then you must learn to use the two frameworks together. Synergy between the two could make development much easier, if only UI and persistence frameworks could somehow be united.

Enter Seam, from JBoss. Seam is a new approach to web development that unites JSF and EJB3 (or Hibernate) into a single potent framework with compelling productivity gains over traditional Java-based web frameworks. Seam works with either Hibernate or EJB3, and you can run it either in the JBoss server or Tomcat 5.5. Next, we see how it works.

An Address Book

To illustrate Seam fundamentals, let's explore the implementation of a Seam address book application, which maintains a list of contacts in a database. The address book is a typical create-read-update-delete application. Figure 12–10 shows how to add contacts to the database. Figure 12–11 on page 599 and Figure 12–12 on page 600 show how to delete and edit contacts, respectively.

The address book has three JSP pages: the address book page, which lists all the contacts in the address book, a page to add a contact, and a page to edit a contact. From looking at those JSP pages, you cannot discern that this is a Seam application.

Figure 12–10 Adding two contacts to an empty address book

For example, here is an excerpt from addressBook.jsp, as shown in Figure 12–10:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:view>
   <html>
      <head>
         <title>Users</title>
         <link href="styles.css" type="text/css" rel="stylesheet"/>
      </head>
      <body>
         <div class="heading">
            <h:outputText value="Address Book"
            styleClass="headingText"/>
         </div>

         <h:form>
            <h:dataTable value="#{contacts}"
                  var="currentContact"
                  styleClass="contacts"
                  rowClasses="contactsEvenRow, contactsOddRow">
               <h:column>
                  <h:commandButton value="delete"
                     action="#{addressBook.delete}"/>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="Name"/>
                  </f:facet>
                  <h:commandLink value="#{currentContact.name}"
                     action="#{addressBook.beginEdit}"/>
               </h:column>

               <%-- The rest of the columns in the table have been ommitted to
                    save space.
               --%>

            </h:dataTable>
            <h:commandLink value="Create more contacts..." action="addContact"/>
         </h:form>
      </body>
   </html>
</f:view>
```

As you might expect, the contacts table is implemented with an `h:dataTable` tag. The `value` attribute for that tag points to a managed bean named `contacts`, and in the body of the `h:dataTable` tag we access another managed bean named `addressBook`. We use this bean to wire buttons and links to JSF action methods.

The application's other two JSP pages, `addContact.jsp` and `editContact.jsp`, are equally innocuous, plain-vanilla JSF views without the slightest hint of any framework other than JSF.

The interesting part of this application lies in its managed beans. You would never know it from the JSP page alone, but the `contacts` and `addressBook` beans from the preceding code are both EJBs. The former is an entity bean, whereas the latter is a stateful session bean. Now we see how it all fits together.

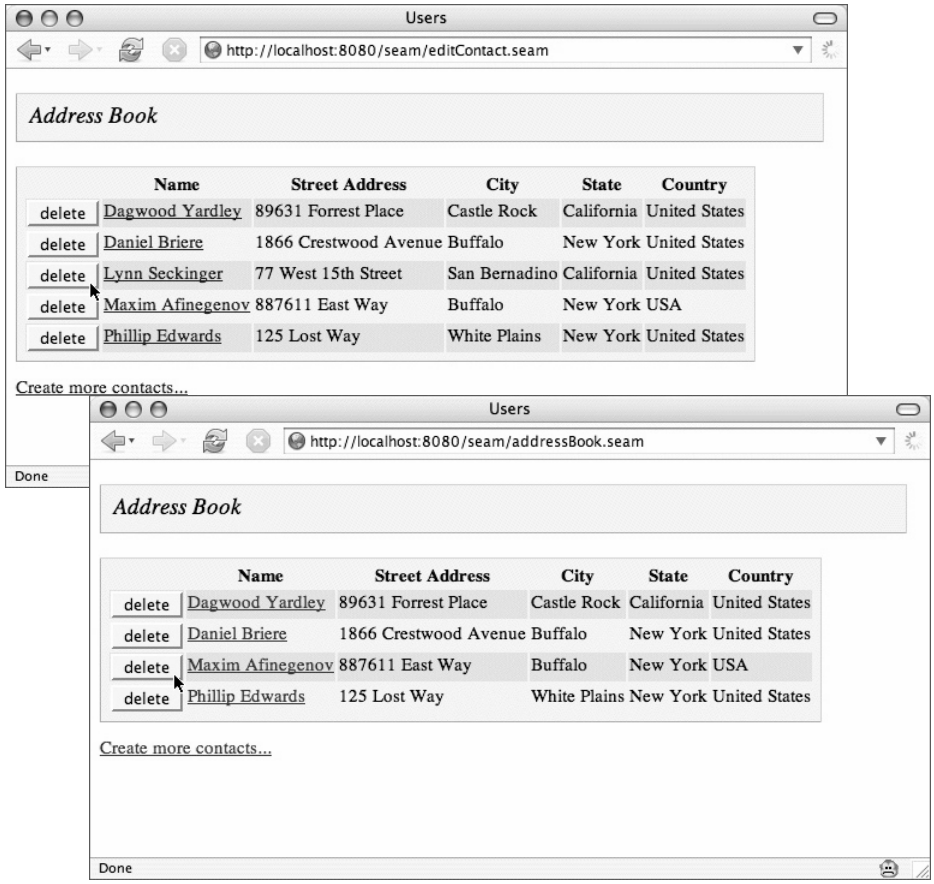

Figure 12–11 Deleting a contact in the address book

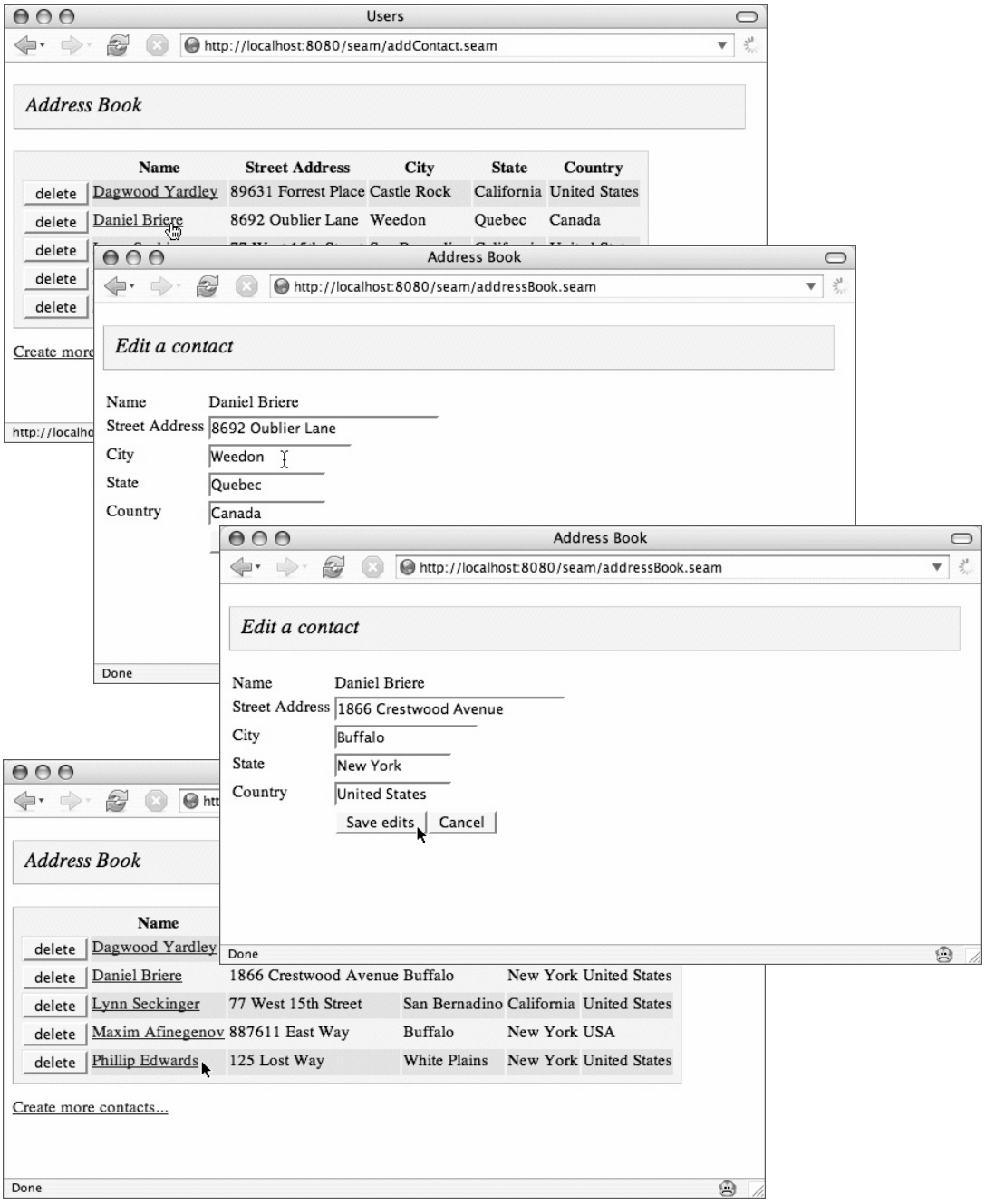

Figure 12–12 Editing a contact in the address book

Configuration

Our application's configuration consists of two XML files: one for JSF and another for persistence. As is typical for Seam applications, our JSF configuration file contains no managed bean declarations—those have been transformed into annotations. In fact, our JSF configuration file consists almost entirely of navigation rules for navigating from one web page to another. The JSF configuration file is so unremarkable that it deserves no further mention.

The persistence XML file is marginally more interesting:

```
<persistence>
   <persistence-unit name="userDatabase">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.dialect"
            value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

In the preceding XML file, we declare our database intentions, including the database name, data source, and Hibernate SQL dialect.

That is essentially all there is for configuration. For our implementation, we used the JBoss embedded EJB server with Tomcat 5.5.

To understand the address book implementation, we start on the ground floor: the database.

Entity Beans

We are storing contacts in a database, so we need an entity bean:

```
@Entity
@Name("contact")
@Scope(ScopeType.EVENT)
@Table(name="contacts")
public class Contact implements Serializable {
   private static final long serialVersionUID = 48L;
   private String name, streetAddress, city;
   private String state, country;
   ...
   public Contact(String name) {
       this.name = name;
    }
```

```
    public Contact() {}

    @Id @NotNull @Length(min=5, max=25)
    public String getName() {
       return name;
    }

    public void setName(String name) {
       this.name = name;
    }

    // Standard JavaBeans setters and getters for the
    // remaining variables are ommitted.

    public String toString() {
       return "Contact(" + name + ")";
    }
}
```

The annotations before the `class` declaration state that the `Contact` class represents an entity bean. We access that bean, named `contact`, in `addContact.jsp`:

```
<h:panelGrid columns="2">
   <h:outputText value="Name"/>
   <h:panelGroup>
      <h:inputText id="name"
         value="#{contact.name}"
         size="20"/>
      <h:message for="name"/>
   </h:panelGroup>

   <h:outputText value="Street Address"/>
   <h:panelGroup>
      <h:inputText id="streetAddress"
         value="#{contact.streetAddress}"
         size="25"/>
      <h:message for="streetAddress"/>
   </h:panelGroup>

   ...

   <h:outputText value=""/>
   <h:commandButton value="Add contact"
      action="#{addressBook.addToBook}"/>
   <h:commandButton value="Cancel"
      action="#{addressBook.cancel}"/>
</h:panelGrid>
```

By virtue of the annotations in the `Contact` class, when Seam first encounters the name `contact` in a value expression, it creates an instance of `com.corejsf.Contact` and places it in request scope (Seam refers to request scope as *event scope*). We also specify the name of the database table—`contacts`—that corresponds to our entity bean.

The `@Id` annotation designates the `name` property as the primary key for the `contacts` table. The `@NotNull` and `@Length` annotations specify that the `name` property cannot be `null` and must contain between five and 25 characters.

NOTE: In the preceding code fragment, we used a total of seven annotations to add persistence to a plain old Java object (POJO) and to transform it into a JSF managed bean. In fact, we used three distinct types of annotations:

- EJB3
- JSF
- Hibernate validator framework

The `@Entity`, `@Table`, and `@Id` annotations are EJB3 annotations, whereas the `@Name` and `@Scope` annotations are JSF-related. Finally, the `@NotNull` and `@Length` annotations are for the Hibernate validator framework, which can be used with either Hibernate or EJB3.

NOTE: Seam performs validation at the model level, not the view level, as is typical for JSF applications.

Stateful Session Beans

Now we have a relatively simple-minded entity bean that we can persist to the database, so it is time to look at the class where the real action is: the stateful session bean that harbors JSF action methods called from JSP pages. First, we declare a local interface:

```
package com.corejsf;

import javax.ejb.Local;

@Local
public interface AddressBook {
   public String addToBook();
   public String delete();
   public String beginEdit();
```

```
   public String edit();
   public void findContacts();
}
```

Next, we implement the stateful session bean:

```
// NOTE: this is not a complete listing. Parts of this class are purposely ommitted
// pending further discussion.

@Stateful
@Scope(ScopeType.SESSION)
@Name("addressBook")
public class AddressBookAction implements Serializable, AddressBook {
   @In(required=false) private Contact contact;
   ...
   @PersistenceContext(type= PersistenceContextType.EXTENDED)
   private EntityManager em;
   ...

   @IfInvalid(outcome=Outcome.REDISPLAY)
   public String addToBook() {
      List existing = em.createQuery("select name from Contact where name=:name")
         .setParameter("name", contact.getName())
         .getResultList();

      if (existing.size()==0) {
         // save to the database if the contact doesn't
         // already exist
         em.persist(contact);
         ...
         return "success";
      }
      else {
         facesContext.addMessage(null,
            new FacesMessage("contact already exists"));
         return null;
      }
   }
   ...

   @Remove @Destroy
   public void destroy() {}
}
```

The `@Name` annotation specifies the name of a managed bean. We referenced that `addressBook` bean from `addressBook.jsp`, listed on page 598. We use the `@Scope` annotation to specify the `addressBook` bean's scope.

With the `@In` annotation, we *inject* the `contact` instance, which means that Seam will intercept all `AddressBookAction` method calls, and if a scoped variable named `contact` exists, Seam will inject it into `AddressBookAction`'s `contact` property before invoking the method. For the `contact` property, injection is not required; otherwise, Seam would throw an exception for any method called when there was no `contact` scoped variable for Seam to inject.

In the `addToBook` method, we use the EJB entity manager to save the contact to the database. The `addToBook` method is called from `addContact.jsp`:

```
            <h:form>
               <h:panelGrid columns="2">
                  <h:outputText value="Name"/>
                  <h:panelGroup>
                     <h:inputText id="name"
                        value="#{contact.name}"
                        size="20"/>
                     <h:message for="name"/>
                  </h:panelGroup>
                  ...

                  <h:outputText value=""/>
                  <h:commandButton value="Add contact"
                     action="#{addressBook.addToBook}"/>
               </h:panelGrid>
            </h:form>
         </body>
      </html>
   </f:view>
```

Here is how the scenario unfolds: When you load `addContact.jsp`, Seam encounters the expression `#{contact.name}`. Since the `contact` bean is request-scoped (or event-scoped in Seam-speak), Seam creates an instance of `com.corejsf.Contact` and stores it in request scope under the name `contact`. Then Seam calls `contact.getName()` to populate the `name` text field as the page loads. Subsequently, the `contact` bean is available throughout the rest of the page.

When the user submits the form, assuming all submitted values pass validation, Seam invokes the corresonding setter methods for the `contact` object's properties and invokes `addressBook.addToBook()`.

When Seam intercepts the call to `addressBook.addToBook()`, it first injects the value of the request-scoped `contact` variable into the `addressBook`'s `contact` property; thus, `addToBook()` has access to the `contact` entity bean, and from there it uses the EJB entity manager to drive the changes home to the database.

NOTE: The address book has two EJBs: an entity bean representing a contact and a stateful session bean. The contact entity beans are stored in the database, whereas the stateful session bean contains JSF action methods and maintains the list of contacts in the database. The stateful session bean could just as easily have been implemented as a JavaBean. But we wanted the convenience of database access in our JSF actions, so we opted for a session bean, as is often the case for Seam applications.

JSF DataModel *Integration*

Seam has built-in support for JSF tables. Once again, take a look at a severely truncated listing of the contacts table in addressBook.jsp:

```
<h:dataTable value="#{contacts}"
   var="currentContact"
   styleClass="contacts"
   rowClasses="contactsEvenRow, contactsOddRow">
   ...
</h:dataTable>
```

Now we revisit the stateful session bean, AddressBookAction, that we discussed in "Stateful Session Beans" on page 603. In that discussion, we ommitted some details, which we explore in the next couple of sections. Here, we look at the @DataModel and @DataModelSelection annotations and their corresponding properties. First, we discuss @DataModel:

```
public class AddressBookAction implements Serializable, AddressBook {
   @DataModel
   @Out(required=false)
   private List<Contact> contacts;
   ...

   @Factory("contacts")
   public void findContacts() {
      contacts = em.createQuery("from Contact")
         .getResultList();
   }
   ...
}
```

When Seam comes across <h:dataTable value="#{contacts}"...>...</h:dataTable> in addressBook.jsp, it looks for a scoped variable named contacts. If the contacts variable does not exist, Seam creates it with a call to the variable's *factory* method: AddressBookAction.findContacts(). That method performs a database query to ensare

all the contacts in the database and stores the resulting list in the contacts variable. At the end of the factory method call, Seam exports the contacts variable to page scope, at the behest of the @Out annotation.

As you can see from this example, Seam factory methods let you wire a JSF component to a persistent object; in the preceding example, we wired a list of contacts from the database to a JSF table.

Seam also has special support for handling table selections. In AddressBookAction, we add a @DataModelSelection annotation:

```
public class AddressBookAction implements Serializable, AddressBook {
   @DataModel
   @Out(required=false)
   private List<Contact> contacts;

   @DataModelSelection
   @Out(required=false, scope=ScopeType.CONVERSATION)
   private Contact selectedContact;
   ...

   @End
   public String edit() {
       em.persist(selectedContact);
       contacts = em.createQuery("from Contact").getResultList();
       return "edited";
   }
   ...

   // This method is called from addressBook.jsp.
   public String delete() {
       // Deletes the selected contact from the database
       contacts.remove(selectedContact);
       em.remove(selectedContact);
       return "deleted";
   }
   ...
}
```

When the user clicks a button or link from the contacts table, Seam injects the selected contact into the AddressBookAction's selectedContact variable before entering the action method associated with the button or link. For example, when the user clicks a "delete" button on the address book page, Seam invokes AddressBookAction.delete(). But before it does, it injects the selected contact into the AddressBookAction's selectedContact variable. Interestingly, setter and getter

methods are not required for the `selectedContact` variable—it is enough to declare the variable and its annotation, and Seam takes care of the rest.

In addition to injecting the `selectedContact` variable, we also export (or outject, if you must) it to *conversation* scope, so we can access it in `editContact.jsp`. Next, we see what conversation scope is all about.

Conversation Scope

In web applications, we have request scope, which spans a single HTTP request, and session scope, which sticks around indefinitely. Often, when implementing a series of interactions, such as a wizard, for example, it would be nice to have a scope in between request and session. In Seam, that's conversation scope.

When we delete a contact from the address book, it is a one-step process. The user clicks a "delete" button, and Seam invokes `AddressBookAction.delete()`, as outlined in the previous section. Seam takes care to inject the selected contact before making the call (this is discussed in "JSF `DataModel` Integration" on page 606). The `delete` method deletes the contact from the database and updates the list of contacts.

However, editing a contact is a two-step process. It starts when the user clicks the link representing the contact's name:

```
<h:dataTable value="#{contacts}" var="currentContact"
      styleClass="contacts"
      rowClasses="contactsEvenRow, contactsOddRow">
   <h:column>
      <h:commandButton value="delete"
         action="#{addressBook.delete}"/>
   </h:column>
   <h:column>
      <f:facet name="header">
         <h:outputText value="Name"/>
      </f:facet>
      <h:commandLink value="#{currentContact.name}"
         action="#{addressBook.beginEdit}"/>
   </h:column>
</h:dataTable>
```

Seam invokes `AddressBookAction.beginEdit()`, once again injecting the selected contact into the `selectedContact` variable before making the call. Here is how `beginEdit()` is implemented:

```
public class AddressBookAction implements Serializable, AddressBook {
   ...

   @DataModelSelection
   @In(required=false)
   @Out(required=false, scope=ScopeType.CONVERSATION)
   private Contact selectedContact;
   ...
   @Begin public String beginEdit() {
       return "edit";
   }

   @End public String edit() {
       em.persist(selectedContact);
       contacts = em.createQuery("from Contact").getResultList();
       return "edited";
   }
}
```

The `beginEdit` method is a JSF action method that returns a string outcome used by JSF to navigate to the next view. That is unremarkable. What is remarkable is the `@Begin` annotation, which signifies that `beginEdit()` starts a conversation. When we leave `beginEdit()`, Seam exports the `selectedContact` to conversation scope, where we subsequently access it in `editContact.jsp`.

The `@End` annotation, attached to the `edit` method, signifies the end of the conversation. When we exit that method, Seam removes the `selectedContact` from conversation scope.

NOTE: We could have eschewed conversations and instead stored the selected contact in session scope. In the edit method, we could have manually removed the selected contact from session scope, thereby creating a psuedo-conversation scope. In fact, many developers have done just that sort of thing; however, it is tedious and error prone. It is much more convenient to let the framework take care of that bookeeping so you can concentrate on higher-level concerns.