CHAPTER 3

Advanced JavaServer Faces

The JSF framework allows developers to build applications utilizing a series of views, and each view consists of a series of components. The framework is kind of like a puzzle in that each piece must fit into its particular place in order to make things work smoothly. Sprinkled into those pieces of the puzzle are advanced capabilities that are used for helping to create a seamless user interface experience.

Components are just one piece of the puzzle. Components are the building blocks that make up JSF views. One of the strengths of using the JSF framework is the abundance of components that are available for use within views. To developers, components can be tags that are placed within the XHTML views. Components resemble standard HTML tags; they contain a number of attributes, an opening tag and a closing tag, and sometimes components that are to be embedded inside of others. Components can also be written in Java code, and their tags can be bound to Java code that resides within a JSF CDI controller.

A number of components come standard with the JSF framework. Some of the recipes in this chapter will cover some widely used standard components in detail, and the chapter will provide examples that will allow you to begin using components in your applications right away.

Another important piece of the JSF user interface puzzle is seamless integration with the backend business logic. A task that can be run in the background, independent of other running tasks, is known as an *asynchronous* task. JavaScript is the most popular modern browser language that is used to implement asynchronous tasking in web applications. Ajax is a set of technologies that allows you to perform asynchronous tasks using JavaScript in the background, sending responses from the client browser to the server and then sending a response back to the client. That response is used to

J. Juneau, *Jakarta EE Recipes*, https://doi.org/10.1007/978-1-4842-5587-2_3

update the page's Document Object Model (DOM). Enhancing an application to make use of such asynchronous requests and responses can greatly improve the overall user experience. The JSF framework allows developers to create rich user experiences via the use of technologies such as Ajax and HTML5. Much of the implementation detail behind these technologies can be abstracted away from the JSF developer using JSF components. As such, the developer needs to worry only about how to use a JSF component tag and relate it to a server-side property.

This chapter delves into using Ajax with the JSF web framework. Along the way, you will learn how to spruce up applications and make the user interface richer and more user-friendly so that it behaves more like that of a desktop application. You'll also learn how to listen to different component phases and system events, allowing further customization of application functionality.

Note This chapter contains examples using the third-party component library PrimeFaces. It is recommended to use the most recent releases of third-party libraries in order to ensure that your application contains stable and secure sources.

Before tackling the recipes, though, the following section provides a brief overview of the standard JSF components and associated common component tags. This will help you get the most out of the recipes.

Component and Tag Primer

Table 3-1 lists the components that are available with a clean install of the JSF framework.

Table 3-1. *JSF HTML Components*

Component	Tag	Description
UIColumn	h:column	Represents a column of data in the dataTable component
UICommand	h:commandButton	Submits a form
	h:commandLink	Links pages or actions
	h:commandScript	Provides ability to call an arbitrary server-side method via Ajax from a JSF view
UIData	h:dataTable	Represents a table used for iterating over collections of data
UIForm	h:form	Represents an input form
UIGraphic	h:graphicImage	Displays an image
UIInput	h:inputHidden	Includes a hidden variable in a form
	h:inputSecret	Allows text entry without displaying the actual text
	h:inputText	Allows text entry
	h:inputTextarea	Allows multiline text entry
UIOutcomeTarget	h:link	Links to another page or location
UIMessage	h:message	Displays a localized message
UIMessages	h:messages	Displays localized messages
UIOutput	h:outputFormat	Displays a formatted localized message
	h:outputLabel	Displays a label for a specified field
	h:outputLink	Displays text and links to another page or location
UIPanel	h:panelGrid	Displays a table
	h:panelGroup	Groups components

(*continued*)

Table 3-1. (*continued*)

Component	Tag	Description
UISelectBoolean	h:selectBooleanCheckbox	Displays a (Boolean) checkbox choice
UISelectItem	h:selectItem	Represents one item in a list of items for selection
UISelectItems	h:selectItems	Represents a list of items for selection
UISelectMany	h:selectManyCheckbox	Displays a group of checkboxes that allow multiple user choices
	h:selectManyListbox	Allows a user to select multiple items from a list
	h:selectManyMenu	Allows a user to select multiple items from a drop-down menu
UISelectOne	h:selectOneListbox	Allows a user to select a single item from a list
	h:selectOneMenu	Allows a user to select a single item from a drop-down menu
	h:selectOneRadio	Allows a user to select one item from a set

JSF provides a number of core tags that can be used to provide more functionality for the components. For example, these tags can be embedded inside JSF component tags and specify rules that can be used to convert the values that are displayed or used as input for the component. Other uses of the core tags are to provide a list of options for a select component, validate input, and provide action and event listeners. Table 3-2 describes the JSF core tags.

Table 3-2. *JSF Core Tags*

Tag	Function
f:actionListener	Registers an action listener method with a component
f:phaseListener	Registers a PhaseListener to a page
f:setPropertyActionListener	Registers a special form submittal action listener
f:valueChangeListener	Registers a value change listener with a component
f:converter	Registers an arbitrary converter with a component
f:convertDateTime	Registers a DateTimeConverter instance with a component
f:convertNumber	Registers a NumberConverter with a component
f:facet	Adds a nested component to particular enclosing parents
f:metadata	Registers a particular facet with a parent component
f:selectItem	Encapsulates one item in a list
f:selectItems	Encapsulates all items of a list
f:websocket	Provides ability to receive messages into a view via WebSockets
f:validateDoubleRange	Registers a DoubleRangeValidator with a component
f:validateLength	Registers a LengthValidator with a component
f:validateLongRange	Registers a LongRangeValidator with a component
f:validator	Registers a custom validator with a component
f:validateRegex	Registers a RegExValidator with a component (JSF 2.0)
f:validateBean	Delegates validation of a local value to a BeanValidator (JSF 2.0)
f:validateWholeBean	Delegates validation of an entire bean or class
f:validateRequired	Ensures that a value is present in a parent component

Note The common sources and the completed classes to run the application for this chapter are contained within the `org.jakartaeerecipes.chapter03` package, and one or more recipes throughout this chapter will utilize classes contained within that package.

Common Component Tag Attributes

Each standard JSF component tag contains a set of attributes that must be specified in order to uniquely identify it from the others, register the component to a controller class, and so on. There is a set of attributes that are common across each component tag, and this section lists those attributes, along with a description of each. *All attributes besides `id` can be specified using JSF EL:*

- `binding`: A controller class property can be specified for this attribute, and it can be used to bind the tag to a component instance within a controller class. Doing so allows you to programmatically control the component from within the controller class.
- `id`: This attribute can be set to uniquely identify the component. If you do not specify a value for the `id` attribute, then JSF will automatically generate one. Each component within a view must have a unique `id` attribute, or an error will be generated when the page is rendered. *I recommend you manually specify a value for the `id` attribute on each component tag, because then it will be easy to statically reference the tag from a scripting language or a controller class if needed. If you let JSF automatically populate this attribute, it may be different each time, and you will never be able to statically reference the tag from a scripting language.*
- `immediate`: This attribute can be set to `true` for input and command components in order to force the processing of validations, conversions, and events when the request parameter values are applied.
- `rendered`: The rendered attribute can be used to specify whether the component should be rendered onscreen. This attribute is typically specified as a JSF EL expression that is bound to a controller class property yielding a Boolean result. The EL expression must be an `rvalue` expression, meaning that it is read-only and cannot set a value.
- `style`: This attribute allows a CSS style to be applied to the component. The specified style will be applied when the component is rendered as output.

- `styleClass`: This attribute allows a CSS style class to be applied to the component. The specified style will be applied when the component is rendered as output.
- `value`: This attribute identifies the value of a given component. For some components, the `value` attribute is used to bind the tag to a CDI property. In this case, the value specified for the component will be read from, or set within, the CDI property. Other components, such as the `commandButton` component, use the `value` attribute to specify a label for the given component.

Common JavaScript Component Tags

Table 3-3 lists a number of attributes that are shared by many of the components, which enable JavaScript functionality to interact with the component.

Table 3-3. *Common Component Attributes*

Attribute	Description
`onblur`	JavaScript code that should be executed when the component loses focus
`onchange`	JavaScript code that should be executed when the component loses focus and the value changes
`ondblclick`	JavaScript code that should be executed when the component has been clicked twice
`onfocus`	JavaScript code that should be executed when the component gains focus
`onkeydown`	JavaScript code that should be executed when the user presses a key down and the component is in focus
`onkeypress`	JavaScript code that should be executed when the user presses a key and the component is in focus
`onkeyup`	JavaScript code that should be executed when key press is completed and the component is in focus
`onmousedown`	JavaScript code that should be executed when the user clicks the mouse button and the component is in focus

(*continued*)

Table 3-3. (*continued*)

Attribute	Description
`onmouseout`	JavaScript code that should be executed when the user moves mouse away from the component
`onmouseover`	JavaScript code that should be executed when the user moves mouse onto the component
`onmousemove`	JavaScript code that should be executed when the user moves mouse within the component
`onmouseup`	JavaScript code that should be executed when the mouse button click is completed and the component is in focus
`onselect`	JavaScript code that should be executed when the component is selected by the user

Binding Components to Properties

All JSF components can be bound to controller class properties. Do so by declaring a property for the type of component you want to bind within the CDI controller class and then by referencing that property using the component's binding attribute. For instance, the following `dataTable` component is bound to a CDI property and then manipulated from within the bean:

In the view:

```
<h:dataTable id="myTable" binding="#{myBean.myTable}" value="#{myBean.
myTableCollection}"/>
```

In the controller:

```
// Provide getter and setter methods for this property
private javax.faces.component.UIData myTable;
...
myTable.setRendered(true);
...
```

Binding can prove to be very useful in some cases, especially when you need to manipulate the state of a component programmatically before re-rendering the view.

3-1. Creating an Input Form

Problem

You want to add input fields to a form within your application.

Solution

Create an input form by enclosing child input components within a parent `form` component. There are four JSF components that will allow for text entry as input. Those components are `inputText`, `inputSecret`, `inputHidden`, and `inputTextarea`. Any or all of these components can be placed within a `form` component in order to create an input form that accepts text entry.

In the example for this recipe, you will create an input form that will be used to sign up for the Acme Bookstore newsletter. The user will be able to enter their first and last names, an email address, a password, and a short description of their interests.

The View: recipe03_01.xhtml

The following code is for the view `recipe03_01.xhtml`, which constructs the layout for the input form:

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

    <body>

        <ui:composition template="layout/custom_template_search.xhtml">
            <ui:define name="content">
                <h:messages globalOnly="true"  errorStyle="color: red"
                infoStyle="color: green"/>
                <h:form id="contactForm">
                    <h1>Subscribe to Newsletter</h1>
```

```
                    <p>
                        Enter your information below in order to be added
                        to the Acme Bookstore newsletter.
                    </p>

                    <br/>
                    <label for="first">First: </label>
                    <h:inputText id="first" size="40"
                    value="#{contactController1.current.first}"/>
                    <br/>
                    <label for="last">Last: </label>
                    <h:inputText id="last" size="40"
                    value="#{contactController1.current.last}"/>
                    <br/>
                    <label for="email">Email: </label>
                    <h:inputText id="email" size="40"
                    value="#{contactController1.current.email}"/>
                    <br/>
                    <label for="password">Enter a password for site
                    access:</label>
                    <h:inputSecret id="password" size="40"
                    value="#{contactController1.current.password}"/>
                    <br/><br/>
                    <label for="description">Enter your book interests
                    </label>
                    <br/>
                    <h:inputTextarea id="description" rows="5" cols="100"
                    value="#{contactController1.current.description}"/>
                    <br/>
                    <h:commandButton id="contactSubmit"
                    action="#{contactController1.subscribe}" value="Save"/>
                </h:form>
            </ui:define>
        </ui:composition>

    </body>
</html>
```

Note As you can see from the example, HTML can be mixed together with JSF component tags. An HTML label tag is used to specify a label for each input component in this recipe. In Recipe 3-3, you will learn about the JSF component that is used to render a label.

To learn more about how the `commandButton` component works, please see Recipe 3-2.

Controller Class: ContactController.java

Each view that contains an input form needs to have an associated controller class, right? The controller class in this case is `RequestScoped`, and the name of the class is `ContactController`. An excerpt from the listing for the `ContactController` class is as follows:

```
import java.util.*;
import javax.enterprise.context.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;
import javax.faces.validator.ValidatorException;
import javax.inject.Inject;
import javax.inject.Named;

@RequestScoped
@Named(value = "contactController")
public class ContactController implements java.io.Serializable {
    private Contact current;

    /**
     * Creates a new instance of ContactController
     */
    public ContactController() {

    }
```

```
    /**
     * Obtains the current instance of the Contact object
     * @return Contact
     */
    public Contact getCurrent(){
        if (current == null){
            current = new Contact();
        }
        return current;
    }

    /**
     * Adds a subscriber to the newsletter
     * @return String
     */
    public String subscribe(){
        // No implementation yet, will add to a database table in Chapter 7
        FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_
        INFO,
                "Successfully Subscribed to Newsletter for " +
                getCurrent().getEmail(), null);
        FacesContext.getCurrentInstance().addMessage(null, facesMsg);
        return "SUBSCRIBE";
    }

    /**
     * Navigational method
     * @return String
     */
    public String add(){
        return "ADD_SUBSCRIBER";
    }
}
```

Note At this time, nothing happens when the submit button is clicked other than a nice "Success" message being displayed on the screen. Later in the book, you will revisit the subscribe method and add the code for creating a record within an underlying database. The input screen should look like Figure 3-1 when rendered.

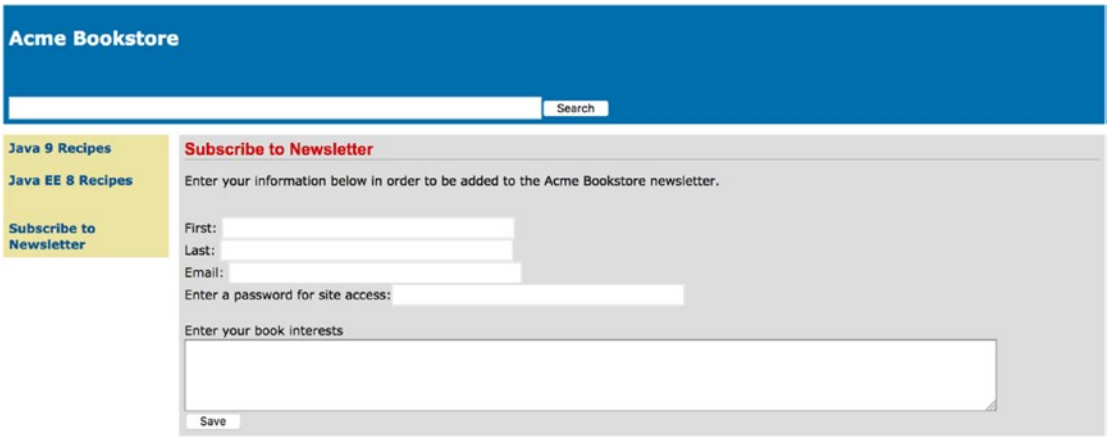

Figure 3-1. *JSF input form for subscribing to the Acme Bookstore newsletter*

How It Works

The JavaServer Faces framework ships with a slew of standard components that can be utilized within JSF views. There are four standard components that can be used for capturing text input: `inputText`, `inputSecret`, `inputHidden`, and `inputTextarea`. These component tags, as well as all of the other standard JSF component tags, share a common set of attributes and some attributes that are unique to each specific tag. To learn more about the common attributes, please see the related section in the introduction to this chapter. In this recipe, I will go over the specifics for each of these input components. The form component, specified via the `h:form` tag, is used to create an input form within a JSF view. Each component that is to be processed within the form should be enclosed between the opening and closing `h:form` tags. Each `form` typically contains at least one command component, such as a `commandButton`. A view can contain more than one `form` component, and only those components that are contained within the `form` will be processed when the `form` is submitted.

Each of the input tags supports the list of attributes that is shown in Table 3-4, in addition to those already listed as common component attributes in the introduction to this chapter.

Table 3-4. *Input Component Tag Attributes*

Attribute	Description
`converter`	Allows a converter to be applied to the component's data.
`converterMessage`	Specifies a message that will be displayed when a registered converter fails.
`dir`	Specifies the direction of text displayed by the component. *(*`LTR` *is used to indicate left-to-right, and* `RTL` *is used to indicate right-to-left.)*
`immediate`	Flag indicating that, if this component is activated by the user, notifications should be delivered to interested listeners and actions immediately (i.e., during the Apply Request Values phase) rather than waiting until the Invoke Application phase.
`label`	Specifies a name that can be used for component identification.
`lang`	Allows a language code to be specified for the rendered markup.
`required`	Accepts a Boolean to indicate whether the user must enter a value for the given component.
`requiredMessage`	Specifies an error message to be displayed if the user does not enter a value for a *required* component.
`validator`	Allows a validator to be applied to the component.
`valueChangeListener`	Allows a controller class method to be bound for event-handling purposes. The method will be called when there is a change made to the component.

The `inputText` component is used to generate a single-line text box within a rendered page. The `inputText` component `value` attribute is most commonly bound to a controller class property so that the values of the property can be retrieved or set when a form is processed. In the recipe example, the first `inputText` component is bound to the controller class property named `first`. The EL expression `#{contactController.current.first}`

is specified for the component `value`, so if the controller class's `first` property contains a value, then it will be displayed within the `inputText` component. Likewise, when the form is submitted, then any value that has been entered within the component will be saved within the `first` property in the controller class.

The `inputSecret` component is used to generate a single-line text box within a rendered page, and when text is entered into the component, then it is not displayed; rather, asterisks are displayed in place of each character typed. This component makes it possible for a user to enter private text, such as a password, without it being displayed on the screen for others to read. The `inputSecret` component works identically to the `inputText` component, other than hiding the text with asterisks. In the example, the value of the `inputSecret` component is bound to a controller class property named `password` via the `#{contactController.current.password}` EL expression.

The `inputTextarea` component is used to generate a multiline text box within a rendered page. As such, this component has a couple of additional attributes that can be used to indicate how large the text area should be. The `inputTextarea` has the `rows` and `cols` attributes, which allow a developer to specify how many rows (height) and how many columns (wide) of space the component should take up on the page, respectively. Other than those two attributes, the `inputTextarea` component works in much the same manner as the `inputText` component. In the example, the `value` attribute of the `inputTextarea` component is specified as `#{contactController.current.description}`, so the `description` property will be populated with the contents of the component when the form is submitted.

The input component I have not yet discussed is the `inputHidden` component. This component is used to place a hidden input field into the form. It works in the same manner as the `inputText` component, except that it is not rendered on the page for the user to see. The value for an `inputHidden` component can be bound to a controller class property in the same way as the other components. You can use such a component for passing a hidden token to and from a `form`.

As you can see, the days of passing and receiving request parameters within JSP pages are over. Utilizing the JSF standard input components, it is possible to bind values to controller class properties using JSF EL expressions. This makes it much easier for developers to submit values from an input form for processing. Rather than retrieving parameters from a page, assigning them to variables, and then processing, the JSF framework takes care of that overhead for you.

3-2. Invoking Actions from Within a Page

Problem

You want to trigger a server-side method to be invoked from a button or link on one of your application pages.

Solution

Utilize the `commandButton` or `commandLink` component within your view to invoke action methods within a controller class. The command components allow for the user invocation of actions within controller classes. Command components bind buttons and links on a page directly to action methods, allowing developers to spend more time thinking about the development of the application and less time thinking about the Java servlet-processing life cycle.

In the example for this recipe, a button and a link are added to the newsletter page for the Acme Bookstore. The button that will be added to the page will be used to submit the input form for processing, and the link will allow a user to log into the application and manage their subscription and bookstore account.

Note This recipe will not cover any authentication or security features; it focuses only on invoking actions within controller classes. For more information regarding authentication, please see Chapter 16.

The View: recipe03_02.xhtml

The following code is for the newsletter subscription view including the command components. The sources are for the file named `recipe03_02.xhtml`:

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

```
<body>

    <ui:composition template="layout/custom_template_search.xhtml">
        <ui:define name="content">
            <h:messages globalOnly="true"  errorStyle="color: red"
            infoStyle="color: green"/>
            <h:form id="contactForm">
                <h1>Subscribe to Newsletter</h1>
                <p>
                    Enter your information below in order to be added
                    to the Acme Bookstore newsletter.
                </p>

                <br/>
                <label for="first">First: </label>
                <h:inputText id="first" size="40"
                value="#{contactController.current.first}"/>
                <br/>
                <label for="last">Last: </label>
                <h:inputText id="last" size="40"
                value="#{contactController.current.last}"/>
                <br/>
                <label for="email">Email: </label>
                <h:inputText id="email" size="40"
                value="#{contactController.current.email}"/>
                <br/>
                <label for="password">Enter a password for site
                access:</label>
                <h:inputSecret id="password" size="40"
                value="#{contactController.current.password}"/>
                <br/><br/>
                <label for="description">Enter your book interests
                </label>
                <br/>
                <h:inputTextarea id="description" rows="5" cols="100"
                value="#{contactController.current.description}"/>
```

```
                    <br/>
                    <h:commandButton id="contactSubmit"
                    action="#{contactController.subscribe}" value="Save"/>
                    <br/><br/>
                    <h:commandLink  id="manageAccount"
                    action="#{contactController.manage}" value="Manage
                    Subscription"/>
                </h:form>
            </ui:define>
        </ui:composition>

    </body>
</html>
```

Controller Class: ContactController.java

The controller class that contains the action methods is named `ContactController`, which was created in Recipe 3-1. The following code excerpt is taken from the `ContactController` class, and it shows the updates that have been made to the methods for this recipe:

Note The complete implementation of `ContactController` resides within the package `org.jakartaeerecipes.chapter03`.

```
...
    /**
     * Adds a subscriber to the newsletter
     * @return String
     */
    public String subscribe(){
        // Using a list implementation for now,
        // but will add to a database table in Chapter 7
```

```
        // Add the current contact to the subscription list
        subscriptionController.getSubscriptionList().add(current);
        FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_
        INFO,
                "Successfully Subscribed to Newsletter for " +
                getCurrent().getEmail(), null);
        FacesContext.getCurrentInstance().addMessage(null, facesMsg);
        return "SUBSCRIBE";
    }

    /**
     * Navigational method
     * @return String
     */
    public String add(){
        return "ADD_SUBSCRIBER";
    }

    /**
     * This method will allow a user to navigate to the manageAccount view.
     * This method will be moved into another controller class that focuses
       on
     * authentication later on.
     * @return
     */
    public String manage(){
        return "/chapter03/manageAccount";
    }
...
```

When the view is rendered, the resulting page looks like Figure 3-2.

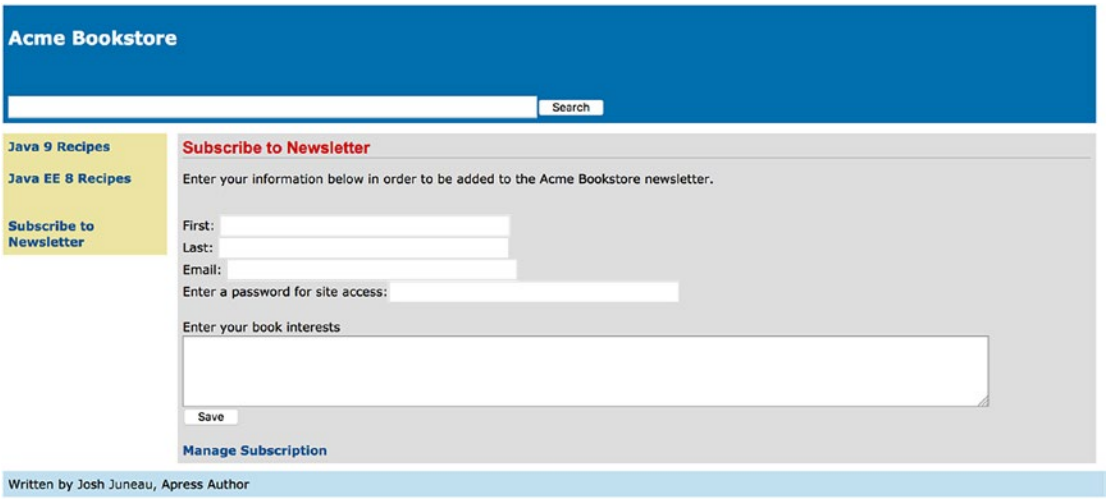

Figure 3-2. *Utilizing command components within a view*

How It Works

The command components make working with JSF vastly different from using JSP technology. In many of the other technologies, form actions are used to handle request parameters and perform any required business logic with them. With the JSF command components, Java methods can be bound directly to a button or a link and invoked when the components are activated (button or link clicked). In the example for this recipe, both the `commandButton` and `commandLink` components are utilized. The `commandButton` component is used to submit the form request parameters for processing, and the `commandLink` component is bound to an action method that performs a redirect to another application page.

The command components have a handful of attributes that are of note. Those attributes, along with a description of each, are listed in Table 3-5 and Table 3-6.

Table 3-5. *commandButton Component Additional Attributes*

Attribute	Description
`action`	EL that specifies a controller class action method that will be invoked when the user activates the component.
`actionListener`	EL that specifies a controller class action method that will be notified when this component is activated. The action method should be public and accept an `ActionEvent` parameter, with a return type of `void`.
`class`	CSS style class that can be applied to the component.
`dir`	Direction indication for text (`LTR`, left-to-right; `RTL`, right-to-left).
`disabled`	A Boolean to indicate whether the component is disabled.
`image`	Absolute or relative URL to an image that will be displayed on the button.
`immediate`	Flag indicating that, if this component is activated by the user, notifications should be delivered to interested listeners and actions immediately (i.e., during the Apply Request Values phase) rather than waiting until the Invoke Application phase.
`label`	Name for the component.
`lang`	Code for the language used for generating the component markup.
`readonly`	Boolean indicating whether the component is read-only.
`rendererType`	Identifier of renderer instance.
`tabindex`	Index value indicating the number of tab button presses it takes to bring the component into focus.
`title`	Tooltip that will be displayed when the mouse hovers over the component.
`transient`	Boolean indicating whether the component should be included in the state of the component tree.
`type`	Indicates type of button to create. Values are `submit` (default), `reset`, and `button`.

Table 3-6. *commandLink Component Additional Attributes*

Attribute	Description
`action`	EL that specifies a controller class action method that will be invoked when the user activates the component.
`accessKey`	Access key value that will transfer the focus to the component.
`cords`	Position and shape of the hotspot on the screen.
`dir`	Direction indication for text (`LTR`, left-to-right; `RTL`, right-to-left).
`disabled`	Specifies a Boolean to indicate whether the component is disabled.
`hreflang`	Language code of the resource designated by the hyperlink.
`immediate`	Flag indicating that, if this component is activated by the user, notifications should be delivered to interested listeners and actions immediately (i.e., during the Apply Request Values phase) rather than waiting until the Invoke Application phase.
`lang`	Code for the language used for generating the component markup.
`rel`	Relationship from the current document to the anchor specified by the hyperlink.
`rev`	Reverse anchor specified by this hyperlink to the current document.
`shape`	Shape of the hotspot on the screen.
`tabindex`	Index value indicating the number of tab button presses it takes to bring the component into focus.
`target`	Name of a frame where the resource retrieved via the hyperlink will be displayed.
`title`	Tooltip that will be displayed when the mouse hovers over the component.
`type`	Indicates type of button to create. Values are `submit` (default), `reset`, and `button`.
`charset`	Character encoding of the resource designated by the hyperlink.

The commandButton and commandLink components in the example for this recipe specify only a minimum number of attributes. That is, they both specify id, action, and value attributes. The id attribute is used to uniquely identify each of the components. The action attribute is set to the JSF EL, which binds the components to their controller class action methods. The commandButton component has an action attribute of #{contactController.subscribe}, which means that the ContactController class's subscribe method will be invoked when the button on the page is clicked. The commandLink has an action attribute of #{contactController.manage}, which means that the ContactController class's manage method will be invoked when the link is clicked. Each of the components also specifies a value attribute, which is set to the text that is displayed on the button or link when rendered.

As you can see, only a handful of the available attributes are used within the example. However, the components can be customized using the additional attributes that are available. For instance, an actionListener method can be specified, which will bind a controller class method to the component, and that method will be invoked when the component is activated. JavaScript functions can be specified for each of the attributes beginning with the word on, providing the ability to produce client-side functionality.

Command components vastly changed the landscape of Java web application development. They allow the incorporation of direct Java method access from within user pages and provide an easy means for processing request parameters.

3-3. Displaying Output

Problem

You want to display text from a controller class property within your application pages.

Solution

Incorporate JSF output components into your views. Output components are used to display static or dynamic text on a page, as well as the results of expression language arithmetic. The standard JSF component library contains five components that render output: outputLabel, outputText, outputFormat, outputLink, and link. The Acme Bookstore utilizes each of these components within the bookstore newsletter application façade.

The View: recipe03_03.xhtml

In the following example, the newsletter subscription view has been rewritten to utilize some of the output components:

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

    <body>

        <ui:composition template="layout/custom_template_search.xhtml">
            <ui:define name="content">
                <h:messages globalOnly="true"  errorStyle="color: red"
                infoStyle="color: green"/>
                <h:form id="contactForm">
                    <h1>Subscribe to Newsletter</h1>
                    <p>
                        <h:outputText  id="newsletterSubscriptionDesc"
                                       value="#{contactController.
                                       newsletterDescription}"/>
                    </p>

                    <br/>
                    <h:outputLabel for="first" value="First: "/>
                    <h:inputText id="first" size="40"
                    value="#{contactController.current.first}">
                        <f:validateRequired/>
                        <f:validateLength minimum="2" maximum="40"/>
                    </h:inputText>
                    <br/>
                    <h:outputLabel for="last" value="Last: "/>
                    <h:inputText id="last" size="40"
                    value="#{contactController.current.last}">
                        <f:validateRequired/>
                        <f:validateLength minimum="2" maximum="40"/>
                    </h:inputText>
```

```
                    <br/>
                    <h:outputLabel for="email" value="Email: "/>
                    <h:inputText id="email" size="40"
                    value="#{contactController.current.email}">
                        <f:validateRequired/>
                        <f:validateRegex pattern=""/>
                    </h:inputText>

                    <br/>
                    <h:outputLabel for="password" value="Enter a password
                    for site access: "/>
                    <h:inputSecret id="password" size="40"
                    value="#{contactController.current.password}">
                        <f:validateRegex pattern=""/>
                    </h:inputSecret>
                    <br/><br/>
                    <h:outputLabel  for="description" value="Enter your
                    book interests"/>
                    <br/>
                    <h:inputTextarea id="description" rows="5" cols="100"
                    value="#{contactController.current.description}"/>
                    <br/>
                    <h:commandButton id="contactSubmit"
                    action="#{contactController.subscribe}" value="Save"/>
                    <br/><br/>
                    <h:commandLink id="manageAccount"
                    action="#{contactController.manage}" value="Manage
                    Subscription"/>
                    <br/><br/>
                </h:form>
            </ui:define>
        </ui:composition>

    </body>
</html>
```

Controller Class: ContactController.java

The `ContactController` controller class has been modified throughout the recipes within this chapter to incorporate new functionality as the recipes move forward. In this recipe, a new property has been added to the `ContactController` that contains the description of the newsletter.

Note The hard-coded newsletter description is not a good idea for use in a production application. It is used in this example for demonstration purposes only. For a production application, utilization of resource bundles or database storage would be a more viable approach for storing `String`s of text.

The following source excerpt from the `ContactController` class shows the code that is of interest in this example:

```
...
    private String newsletterDescription;

    public ContactController() {
        current = null;
        newsletterDescription = "Enter your information below in order to
        be " + "added to the Acme Bookstore newsletter.";
    }
...
    public String getNewsletterDescription() {
        return newsletterDescription;
    }

    public void setNewsletterDescription(String newsletterDescription) {
        this.newsletterDescription = newsletterDescription;
    }
...
```

The resulting page looks like Figure 3-3. Note that the text is the same, because it is merely reading the same text from a controller class property. Also note that there is now an additional link added to the bottom of the page, which reads Home.

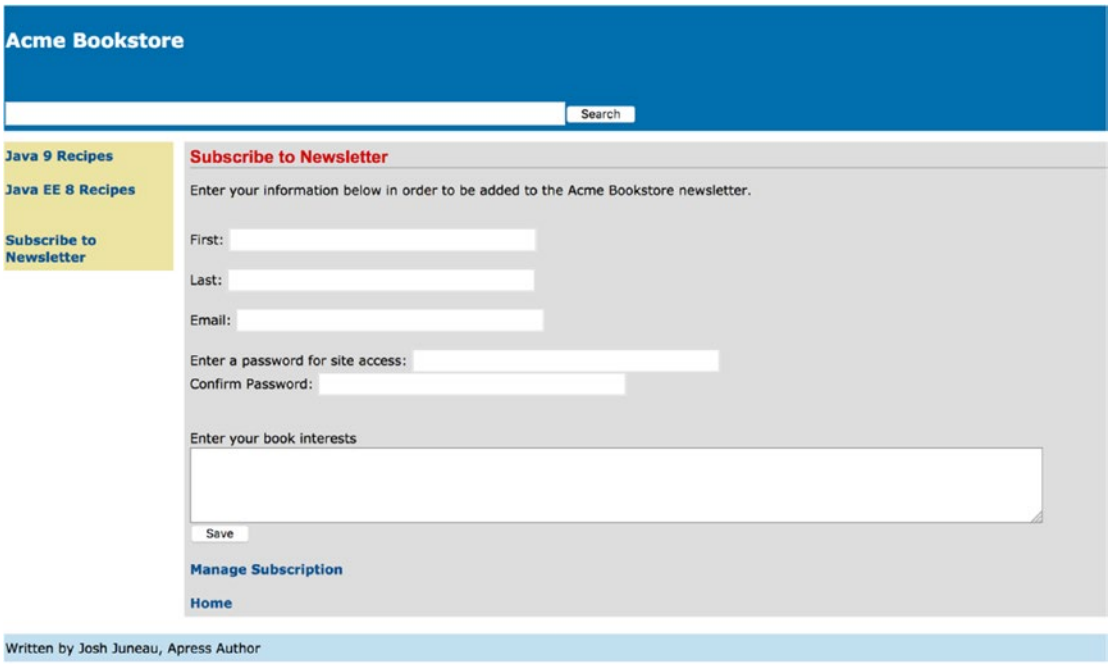

Figure 3-3. *Utilizing output components within a view*

How It Works

Output components can be used to display output that is generated within a controller class or to render a link to another resource. They can be useful in many cases for displaying dynamic output to a web view. The example for this recipe demonstrates three out of the five different output component types: `outputText`, `outputLink`, and `outputLabel`. Each of the components shares a common set of attributes, which are listed in Table 3-7.

Note The `outputText` component has become a bit less important since the release of JSF 2.0 because the Facelets view definition language implicitly wraps inline content with a similar output component. Therefore, the use of the `outputText` tag within JSF 2.0 is necessary only if you want to utilize some of the tag attributes for rendering, JavaScript invocation, or the like.

Table 3-7. *Common Output Component Attributes (Not Listed in Introduction)*

Attribute	Description
`class`	CSS class for styling
`converter`	Converter that is registered with the component
`dir`	Direction of text (`LTR`, left-to-right; `RTL`, right-to-left)
`escape`	Boolean value to indicate whether XML- and HTML-sensitive characters are escaped
`lang`	Code for language used when generating markup for the component
`parent`	Parent component
`title`	Tooltip text for the component
`transient`	Boolean indicating whether the component should be included in the state of the component tree

The `outputText` component in the example contains a value of `#{contactController.newsletterDescription}`, which displays the contents of the `newsletterDescription` property within `ContactController`. Only the common output component attributes can be specified within the `h:outputText` tag. Therefore, an attribute such as `class` or `style` can be used to apply styles to the text displayed by the component. If the component contains HTML or XML, the `escape` attribute can be set to `true` to indicate that the characters should be escaped.

The `outputFormat` component shares the same set of attributes as the `outputText` component. The `outputFormat` component can be used to render parameterized text. Therefore, if you require the ability to alter different portions of a `String` of text, you can do so via the use of JSF parameters (via the `f:param` tag). For example, suppose you wanted to list the name of books that someone has purchased from the Acme Bookstore; you could use the `outputFormat` component like in the following example:

```
<h:outputFormat value="Cart contains the books {0}, {1}, {2}"/>
    <f:param value="Java 9 Recipes"/>
    <f:param value="JavaFX 2.0: Introduction by Example"/>
    <f:param value="Java EE 8 Recipes"/>
</h:outputFormat>
```

The outputLink and outputLabel components can each specify a number of other attributes that are not available to the previously discussed output components. The additional attributes are listed in Table 3-8 (outputLink) and Table 3-9 (outputLabel). The outputLink component can be used to create an anchor or link that will redirect an application user to another page when the link is clicked. In the following example, the outputLink component is used to redirect a user to a view named home.xhtml. The value for the outputLink component can be set to a static page name, as per the example, or it can contain a JSF EL expression corresponding to a controller class property. It is also possible to pass parameters to another page using the outputLink component by nesting f:param tags between opening and closing h:outputLink tags as follows:

```
<h:outputLink id="homeLink" value="home.xhtml">
    <h:outputText value="User Home Page"/>
    <f:param name="username" value="#{contactController.current.email}"/>
 </h:outputLink>
```

The previous example would produce a link with the text *User Home Page* when rendered on the page. It would produce the following HTML link, where emailAddress corresponds to the EL expression of #{contactController.current.email}:

```
<a href="home.xhtml?username=emailAddress">Home Page</a>
```

Similarly, rather than displaying a link as text on the page, an image can be used by embedding a graphicImage component.

The outputLabel component renders an HTML <label> tag, and it can be used in much the same way as the outputText component. In the example, the outputLabel component values are all using static text, but they could also utilize JSF EL expressions to make use of controller class property values if that is more suitable for the application.

Table 3-8. *outputLink Additional Attributes*

Attribute	Description
`accessKey`	Access key value that will transfer the focus to the component.
`binding`	`ValueExpresssion` linking this component to a property in a backing bean.
`charset`	The character encoding of the resource designated by this hyperlink.
`cords`	Position and shape of the hotspot on the screen.
`dir`	Direction indication for text (`LTR`, left-to-right; `RTL`, right-to-left).
`disabled`	Specifies a Boolean to indicate whether the component is disabled.
`fragment`	Identifier for the page fragment that should be brought into focus when the target page is rendered.
`hreflang`	Language code of the resource designated by the hyperlink.
`lang`	Code for the language used for generating the component markup.
`rel`	Relationship from the current document to the anchor specified by the hyperlink.
`rev`	Reverse anchor specified by this hyperlink to the current document.
`shape`	Shape of the hotspot on the screen.
`tabindex`	Index value indicating the number of tab button presses it takes to bring the component into focus.
`target`	Name of a frame where the resource retrieved via the hyperlink will be displayed.
`title`	Tooltip that will be displayed when the mouse hovers over the component.
`type`	Type of button to create. Values are `submit` (default), `reset`, and `button`.

Table 3-9. *outputLabel Additional Attributes*

Attribute	Description
`accessKey`	Access key value that will transfer the focus to the component.
`binding`	`ValueExpresssion` linking this component to a property in a backing bean.
`dir`	Direction indication for text (`LTR`, left-to-right; `RTL`, right-to-left).
`escape`	Flag indicating that characters that are sensitive in HTML and XML markup must be escaped.
`for`	Client identifier of the component for which this element is a label.
`lang`	Code for the language used for generating the component markup.
`tabindex`	Index value indicating the number of tab button presses it takes to bring the component into focus.
`title`	Tooltip that will be displayed when the mouse hovers over the component.
`type`	Type of button to create. Values are `submit` (default), `reset`, and `button`.

The last output component that I'll cover in this recipe is the link component. It was introduced to JSF in release 2.0, and it makes the task of adding links to a page just a bit easier. Both the `outputLink` and `link` components produce similar results, but `link` has just a couple of different attributes that make it react a bit differently. The `value` attribute of the `h:link` tag specifies the label or text that should be used when the link is rendered on the page, and the `outcome` attribute specifies the page that should be linked to. The following example of the link component produces the same output as the `outputLink` component in the example for this recipe:

```
<h:link id=""homeLink"" value=""Home"" outcome=""home""/>
```

Parameters and images can also be embedded within the `h:link` tag, in the same manner as with `outputLink`. The `link` component also contains some custom attributes, as listed in Table 3-10.

Table 3-10. *link Component Additional Attributes*

Attribute	Description
`charset`	Character encoding of the resource that is designated by the hyperlink.
`cords`	Position and shape of the hotspot on the screen, usually used when generating maps or images containing multiple links.
`disabled`	Flag to indicate that the component should never receive focus.
`fragment`	Identifier for the page fragment that should be brought into focus when the link is clicked. The identifier is appended to the # character.
`hreflang`	Language of the resource designated by this link.
`includeviewparams`	Boolean indicating whether to include page parameters when redirecting.
`outcome`	Logical outcome used to resolve a navigational case.
`rel`	Relationship from the current document to the resource specified by link.
`rev`	Reverse link from the anchor specified from this link to the current document.
`shape`	Shape of the hotspot on the screen.
`target`	Name of the frame in which the resource linked to is to be displayed.
`type`	Content type of resource that is linked to.

This recipe provided a high-level overview of the JSF standard output components. In JSF 2.0+, it is important to note that you can simply include a JSF EL expression without using an output component to display text within a page. However, these components can still be quite useful under certain circumstances, making them an important set of components to have within your arsenal.

3-4. Adding Form Validation

Problem

To ensure that valid data is being submitted via your form, you need to incorporate some validation on your input fields.

Solution #1

Utilize prebuilt JSF validator tags on the view's input components where possible. JSF ships with a handful of prebuilt validators that can be applied to components within a view by embedding the validator tag within the component you want to validate. The following code excerpt is taken from a JSF view that defines the layout for the newsletter subscription page of the Acme Bookstore application. The sources can be found in the view named `recipe03_04.xhtml`, and the excerpt demonstrates applying prebuilt validators to some `inputText` components:

```
...
<h:outputLabel for="first" value="First: "/>
<h:inputText id="first" size="40" value="#{contactController.current.first}">
    <f:validateLength minimum="1" maximum="40"/>
 </h:inputText>
<br/>
<h:message id="firstError"
                   for="first"
                   errorStyle="color:red"/>
<br/>
<h:outputLabel for="last" value="Last: "/>
<h:inputText id="last" size="40" value="#{contactController.current.last}">
    <f:validateLength minimum="1" maximum="40"/>
</h:inputText>
<br/>
<h:message id="lastError"
                    for="last"
                    errorStyle="color:red"/>
<br/>
...
```

In the preceding code excerpt, you can see that the `f:validateLength` validator tags have been embedded in different `inputText` components. When the form is submitted, these validators will be applied to the values within the `inputText` component fields and will return an error message if the constraints have not been met.

Solution #2

Utilize JSF bean validation by annotating controller class fields with validation annotations. It is possible to perform validation from within the controller class by annotating the property field declaration with the validation annotations that are needed. When the form is submitted, then the bean validation will be performed.

Note An `f:validateBean` tag can be embedded within the component in the view if making use of `validationGroups` in order to delegate the validation of the local value to the Bean Validation API. If using `f:validateBean`, the `validationGroups` attribute will serve as a filter that instructs which constraints should be enforced.

The following code excerpt is taken from the JSF view that defines the layout for the newsletter subscription page of the Acme Bookstore application. The sources can be found in the view named `recipe03_04.xhtml`:

```
...
<h:outputLabel for="email" value="Email: "/>
<h:inputText id="email" size="40" value="#{contactController.current.
email}"/>
<br/>
<h:message id="emailError"
                    for="email"
                    errorStyle="color:red"/>
...
```

Next is an excerpt from the `ContactController` controller class that demonstrates applying a validator annotation to the `email` property field declaration:

```
...
@Pattern(regexp = "[a-zA-Z0-9]+@[a-zA-Z0-9]+\\.[a-zA-Z0-9]+", message =
"Email format is invalid.")
    private String email;
...
```

When the form is submitted, the validation on the `email` field will occur. If the value entered into the `inputText` component does not validate against the regular expression noted in the annotation, then the error message will be displayed within the corresponding `messages` component.

Solution #3

Create a custom validator method within a controller class, and register that method with an input component by specifying the appropriate EL for the component's `validator` attribute. In this scenario, the controller class does not need to implement the `Validator` interface. The following code excerpt is taken from the JSF view that defines the layout for the newsletter subscription page of the Acme Bookstore application. The sources can be found in the view named `recipe03_04.xhtml`, and the excerpt demonstrates a custom validator method to a component by specifying it for the `validator` attribute:

```
 ...
<h:outputLabel for="password" value="Enter a password for site access: "/>
<h:inputSecret id="password" size="40" redisplay="true"
value="#{contactController.current.password}"/>
<br/>
<h:outputLabel for="passwordConfirm" value="Confirm Password: "/>
<h:inputSecret id="passwordConfirm" size="40" redisplay="true"
                        validator="#{contactController.
                        validatePassword}"/>
<br/>
<h:message id="passwordConfirmError"
                     for="passwordConfirm"
                     style="color:red"/>
...
```

Note If you are thinking outside of the box, you'll see that the previous code fragment would be an excellent choice for creating into a composite component! If a composite component is created, then it would be as simple as adding a tag such as `<custom:passwordValidate>` to your form.

The validator attribute specifies the validatePassword method within the ContactController controller class. The following excerpt is taken from ContactController, and it shows the validator method's implementation:

```
...
/**
     * Custom validator to ensure that password field contents match
     * @param context
     * @param component
     * @param value
     */
    public void validatePassword(FacesContext context,
                                 UIComponent component,
                                 Object value){
        Map map = context.getExternalContext().getRequestParameterMap();
        String passwordText = (String) map.get(("contactForm:password"));
        String confirmPassword = value.toString();

        if (!passwordText.equals(confirmPassword)) {
            throw new ValidatorException(new FacesMessage("Passwords do not
            match"));
        }
    }
...
```

When the form is submitted, the validatePassword method will be invoked during the Process Validations phase. The method will read the values of both the password and passwordConfirm fields, and an exception will be thrown if they do not match. For example, if the input form for the newsletter subscription page is submitted without any values, then the page should be re-rendered and look like Figure 3-4.

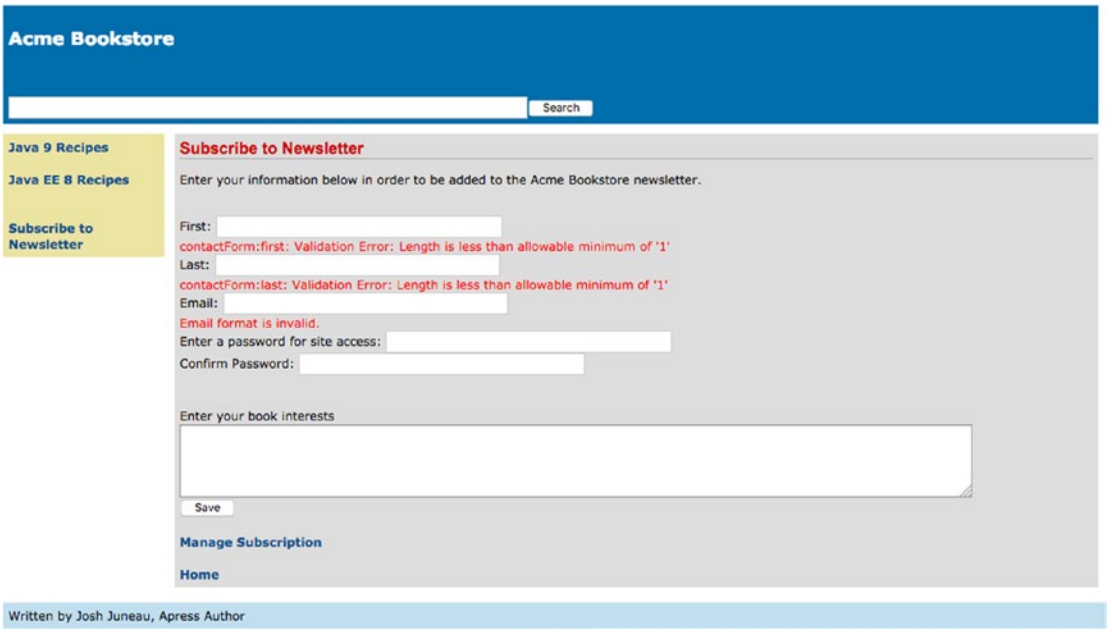

Figure 3-4. *Validation errors on input fields*

How It Works

There are a few different ways in which to apply validation to form input fields. The easiest way to apply validation to an input component is to utilize the prebuilt validator tags that ship with JSF. There are prebuilt tags for validating data for a specified length, range, and so on. Please see Table 3-2 in the introduction to this chapter for the complete list of validator tags. You can also choose to apply validation to input components using bean validation. Bean validation requires validation annotations to be placed on the property declaration within the controller class. Yet another possible way to perform validation is to create a custom validation method and specify the method within the input component's `validator` attribute. This section will provide a brief overview of each prebuilt validation tag, cover the basics of bean validation, and demonstrate how to build a custom validation method.

Note It is possible to create a class that implements the `Validator` interface to perform validation.

No matter which validation solution you choose to implement, the validation occurs during the Process Validations phase of the JSF life cycle. When a form is submitted, via a command component or an Ajax request, all validators that are registered on the components within the tree are processed. The rules that are specified within the attributes of the component are compared against the local value for the component. At this point, if any of the validations fails, the messages are returned to the corresponding `message` components and displayed to the user.

To utilize the prebuilt validation tags, they must be embedded between opening and closing input component tags and specify attributes according to the validation parameters you wish to set. In Solution #1 for this recipe, you learned how to use the `f:validateLength` validator tag, which allows validation of component data for a specified length. The `minimum` and `maximum` attributes are set to the minimum string length and maximum string length, respectively.

The `f:validateLongRange` validator can be used to check the range of a numeric value that has been entered. The minimum and maximum attributes of `f:validateLongRange` are used to determine whether the value entered falls within the lower and upper bounds, respectively.

Similar to `f:validateLongRange` is the `f:validateDoubleRange` validator, which is used to validate the range of a floating-point value. Again, the minimum and maximum attributes of `f:validateDoubleRange` are used to determine whether the value entered falls within the lower and upper bounds, respectively.

The `f:validateRequired` validator is used to ensure that an input field is not empty. No attributes are needed with this validator; simply embed it within a component tag to ensure that the component will not contain an empty value.

Another validator that ships with JSF is the `f:validateRegex` validator. This validator uses a regular expression pattern to determine whether the value entered matches the specified pattern. The validator's pattern attribute is used to specify the regular expression pattern, as shown in the example for Solution #1 to this recipe.

In Solution #2, JSF bean validation is demonstrated. Bean validation allows you to annotate a controller class field with constraint annotations that indicate the type of validation that should be performed. The validation automatically occurs on the annotated fields when a form that contains input components referencing them is submitted. A handful of standard constraint annotations can be applied to bean fields, as listed in Table 3-11. Each annotation accepts different attributes; please see the online documentation at `https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/` for more details.

Table 3-11. *Constraint Annotations Used for Bean Validation*

Annotation	Description
`@AssertFalse`	The annotated element must be false.
`@AssertTrue`	The annotated element must be true.
`@DecimalMax`	The annotated element must be a decimal that has a value less than or equal to the specified maximum.
`@DecimalMin`	The annotated element must be a decimal that has a value greater than or equal to the specified minimum.
`@Digits`	The annotated element must be a number within the accepted range.
`@Email`	The annotated element must adhere to the format of an email address.
`@Future`	The annotated element must be a date in the future.
`@Max`	The annotated element must be a number that has a value less than or equal to the specified maximum.
`@Min`	The annotated element must be a number that has a value greater than or equal to the specified minimum.
`@Negative`	The annotated element must be a negative number.
`@NotBlank`	The annotated element must not be `null` or blank after removing any trailing or leading whitespace.
`@NotEmpty`	The annotated element must not be `null` or empty.
`@NotNull`	The annotated element must not be `null`.
`@Null`	The annotated element must be `null`.
`@Past`	The annotated element must be a date in the past.
`@Pattern`	The annotated element must match the pattern specified in the annotation's regular expression.
`@Positive`	The annotated element must be a positive number.
`@Size`	The annotated element must be between the specified boundaries.

When using bean validation, the input component that references an annotated bean field can contain an `f:validateBean` tag to customize behavior. The `f:validateBean` tag's `validationGroups` annotation can be used to specify validation groups that can be used for validating the component. For instance, such a solution may resemble something like the following:

```
<h:inputText id="email" value="#{contactController.email}">
    <f:validateBean validationGroups="org.jakartaeerecipes.validation.
    groups.EmailGroup"/>
</h:inputText>
```

Note Validation groups define a subset of constraints that can be applied for validation. A validation group is represented by an empty Java interface. The interface name can then be applied to annotation constraints within a bean class in order to assign such constraints to a particular group. For instance, the following field that is annotated with `@Size` specifies a group of `EmailGroup.class`:

```
@Size(min=2, max=30, groups=EmailGroup.class)
private String email;
```

When utilizing the `f:validateBean` tag, any constraint annotations that are contained within the specified group will be applied to the field for validation.

When using bean validation, a custom error message can be displayed if the validation for a field fails. To add a custom message, include the `message` attribute within the annotation, along with the error message that you want to have displayed. As a best practice, error messages should be pulled from a message bundle (https://docs.oracle.com/javase/tutorial/i18n/resbundle/concept.html) so that they can be updated without the need to change code.

The example for Solution #3 demonstrates the use of a custom validator method in order to perform validation on an input component. The input component's `validator` attribute can reference a controller class method that has no return type and accepts a `FacesContext`, a `UIComponent`, and an `Object`. The method can utilize the parameters to gain access to the current `FacesContext`, the `UIComponent` that is being validated, and the current value that is contained in the object, respectively. The validation logic can throw a `javax.faces.validator.ValidatorException` if the value does not pass validation and then return a message to the user via the exception. In the example, the

method named `validatePassword` is used to compare the two password field contents to ensure that they match. The first two lines of code within the method are used to obtain the value of the component with the `id` of `password` and save it into a local variable. The actual validation logic compares that value against the incoming parameter's `Object` value, which is the current value of the component being validated, to determine whether there is a match. If not, then a `ValidationException` is thrown with a corresponding message. That message will then be displayed within the `messages` component that corresponds to the component being validated.

As mentioned at the beginning of this recipe, there are a few ways to validate input. None of them is any better than the other; their usage depends upon the needs of your application. If you are going to be changing validation patterns often, then you may want to stick with the prebuilt validator tags so that you do not need to recompile code in order to change the validation. On the other hand, if you know that your validation will not change, then it may be easier for you to work with the bean validation technique.

3-5. Validating Input with Ajax

Problem

You want to validate the values that are entered into text fields of a form, but you want them to be evaluated immediately, rather than after the form is submitted.

Solution

Perform validation on the field(s) by embedding the `f:ajax` tag within each component whose values you want to validate. Specify appropriate values for the `event` and `render` attributes so that the Ajax validation will occur when the field(s) loses focus, and any validation errors will be identified immediately. The following listing is the JSF view for the newsletter subscription page of the Acme Bookstore application. It has been updated to utilize Ajax validation so that the validation occurs immediately, without the need to submit the form before corresponding errors are displayed:

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
                xmlns:f="http://xmlns.jcp.org/jsf/core"
                xmlns:h="http://xmlns.jcp.org/jsf/html"
```

```
            template="layout/custom_template_search.xhtml">
    <ui:define name="content">
        <h:messages globalOnly="true"  errorStyle="color: red"
        infoStyle="color: green"/>
        <h:form id="contactForm">
            <h1>Subscribe to Newsletter</h1>
            <p>
                <h:outputText id="newsletterSubscriptionDesc"
                              value="#{ch3ContactController.
                              newsletterDescription}"/>
            </p>

            <br />
            <h:panelGrid columns="2" bgcolor="" border="0">
                <h:panelGroup>
                    <h:outputLabel for="first" value="First: "/>
                    <h:inputText id="first" size="40"
                    value="#{ch3ContactController.current.first}">
                        <f:validateLength minimum="1" maximum="40"/>
                        <f:ajax event="blur" render="firstError"/>
                    </h:inputText>
                </h:panelGroup>
                <h:panelGroup>

                    <h:outputLabel for="last" value="Last: "/>
                    <h:inputText id="last" size="40"
                    value="#{ch3ContactController.current.last}">
                        <f:validateLength minimum="1" maximum="40"/>
                        <f:ajax event="blur" render="lastError"/>
                    </h:inputText>
                </h:panelGroup>

                <h:message id="firstError"
                           for="first"
                           errorStyle="color:red"/>
```

```
<h:message id="lastError"
           for="last"
           errorStyle="color:red"/>
<h:panelGroup>
    <h:outputLabel for="email" value="Email: "/>
    <h:inputText id="email" size="40"
    value="#{ch3ContactController.current.email}">
        <f:ajax event="blur" render="emailError"/>
    </h:inputText>
</h:panelGroup>
<h:panelGroup/>
<h:message id="emailError"
           for="email"
           errorStyle="color:red"/>
<h:panelGroup/>

<h:selectOneRadio title="Gender" id="gender"
value="#{ch3ContactController.current.gender}">
    <f:selectItem  itemValue="M" itemLabel="Male"/>
    <f:selectItem itemValue="F" itemLabel="Female"/>
</h:selectOneRadio>
<h:panelGroup>
    <h:outputLabel for="occupation" value="Occupation: "/>
    <h:selectOneMenu id="occupation"
    value="#{ch3ContactController.current.occupation}">
        <f:selectItems value="#{ch3ContactController.
        occupationList}"/>
    </h:selectOneMenu>
</h:panelGroup>
<h:message id="genderError"
           for="gender"
           errorStyle="color:red"/>
```

```
</h:panelGrid>
<br />
<h:outputLabel for="description" value="Enter your book
interests"/>
<br />
<h:inputTextarea id="description" rows="5" cols="75"
value="#{ch3ContactController.current.description}"/>

<br />
<h:panelGrid columns="2">
    <h:outputLabel for="password" value="Enter a password for
    site access: "/>
    <h:inputSecret id="password" size="40"
    value="#{ch3ContactController.current.password}">
        <f:validateRequired/>
        <f:ajax event="blur" render="passwordError"/>
    </h:inputSecret>

    <h:outputLabel for="passwordConfirm" value="Confirm
    Password: "/>
    <h:inputSecret id="passwordConfirm" size="40"
    value="#{ch3ContactController.passwordConfirm}"
                validator="#{ch3ContactController.
                validatePassword}">
        <f:ajax event="blur" render="passwordConfirmError"/>
    </h:inputSecret>
</h:panelGrid>
<h:message id="passwordError"
           for="password"
           style="color:red"/>
<br />
<h:message id="passwordConfirmError"
           for="passwordConfirm"
           style="color:red"/>
<br />
<hr/>
<br />
```

```
<h:panelGrid columns="3">
    <h:panelGroup>
        <h:outputLabel for="newsletterList"
        value="Newsletters:" style=" "/>
        <h:selectManyListbox id="newsletterList"
        value="#{ch3ContactController.current.newsletterList}">
            <f:selectItems value="#{ch3ContactController.
            newsletterList}"/>
        </h:selectManyListbox>
    </h:panelGroup>
    <h:panelGroup/>
    <h:panelGroup>
        <h:panelGrid columns="1">
            <h:panelGroup>
                <h:outputLabel for="notifyme" value="Would you
                like to receive other promotional email?"/>
                <h:selectBooleanCheckbox id="notifyme"
                value="#{ch3ContactController.current.
                receiveNotifications}"/>
            </h:panelGroup>
            <h:panelGroup/>
            <hr/>
            <h:panelGroup/>
            <h:panelGroup>
                <h:outputLabel for="notificationTypes"
                value="What type of notifications are you
                interested in receiving?"/>
                <br />
                <h:selectManyCheckbox id="notifyTypes"
                value="#{ch3ContactController.current.
                notificationType}">
                    <f:selectItems value="#
                    {ch3ContactController.notificationTypes}"/>
```

```
                        </h:selectManyCheckbox>
                    </h:panelGroup>
                </h:panelGrid>
            </h:panelGroup>
        </h:panelGrid>
        <hr/>
        <br />

        <h:commandButton id="contactSubmit" action="#{ch3Contact
        Controller.subscribe}" value="Save"/>
        <h:panelGrid  columns="2" width="400px;">
            <h:commandLink id="manageAccount" action="#{ch3Contact
            Controller.manage}" value="Manage Subscription"/>

            <h:outputLink id="homeLink" value="home.xhtml">Home</
            h:outputLink>
        </h:panelGrid>
    </h:form>
  </ui:define>
</ui:composition>
```

Once the input components have been "Ajaxified" by embedding the `f:ajax` tag within them, then tabbing through the fields (causing the `onBlur` event to occur for each field) will result in a form that resembles Figure 3-5.

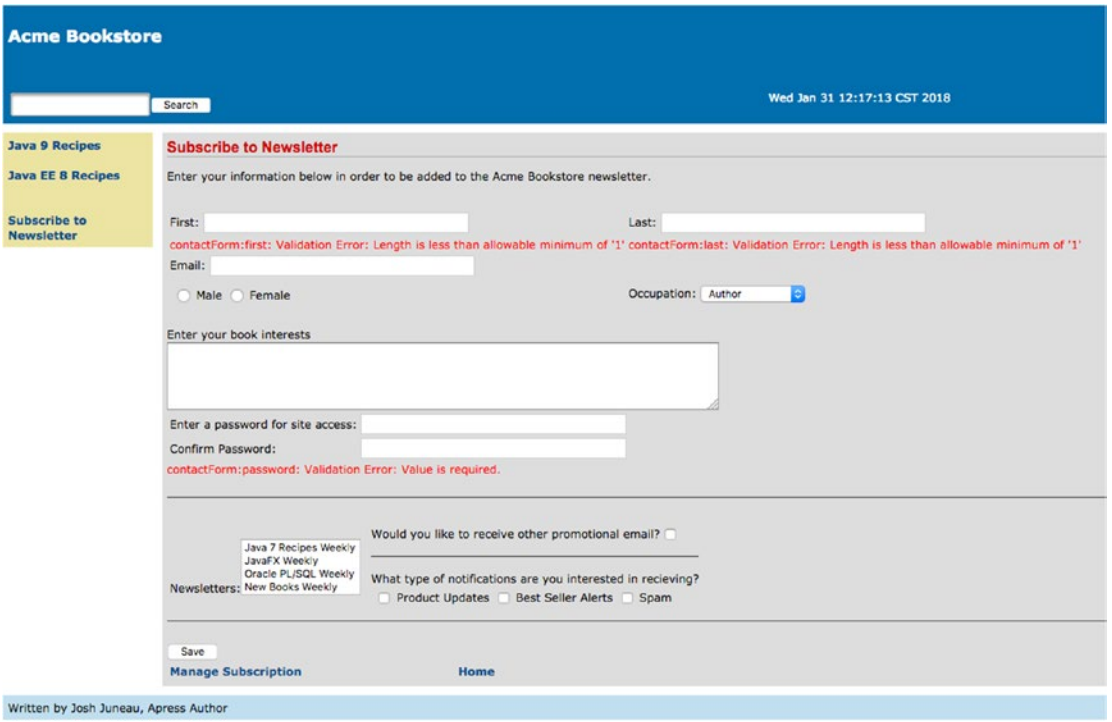

Figure 3-5. *Ajax validation using the f:ajax tag*

How It Works

In releases of JSF prior to 2.0, performing immediate validation required the manual coding of JavaScript or a third-party component library. The `f:ajax` tag was added to the JSF arsenal with the release of 2.0, bringing with it the power to easily add immediate validation (and other asynchronous processes) to JSF views using standard or third-party components. The `f:ajax` tag can be embedded within any JSF input component in order to immediately enhance the component, adding Ajax capabilities to it. This provides many benefits to the developer in that there is no longer a need to manually code JavaScript to perform client-side validation. It also allows validation to occur on the server (in Java code within a JSF controller class) asynchronously, providing seamless interaction between the client and server and generating an immediate response to the client. The result is a rich modern web application that behaves in much the same manner as a native desktop application. Validation can now occur instantaneously in front of an end user's eyes without the need to perform several page submits in order to repair all of the possible issues.

To use the `f:ajax` tag, simply embed it within any JSF component. There are a number of attributes that can be specified with `f:ajax`, as described in Table 3-12. If an attribute is not specified, then the default values are substituted. It is quite possible to include no attributes in an `f:ajax` tag, and if this is done, then the default attribute values for the component in which the `f:ajax` tag is embedded will take effect.

Table 3-12. *f:ajax Tag Attributes*

Attribute	Description
`delay`	A value that is specified in milliseconds, corresponding to the amount of delay between sending Ajax requests from the client-side queue to the server. The value `none` can be specified to disable this feature.
`disabled`	Boolean value indicating the tag status. A value of `true` indicates that the Ajax behavior should not be rendered, and a value of `false` indicates that the Ajax behavior should be rendered. The default value is `false`.
`event`	A `String` that identifies the type of event to which the Ajax action shall apply. If specified, it must be one of the supported component events. The default value is the event that triggers the Ajax request for the parent component of the Ajax behavior. The default event is `action` for `ActionSource` components and is `valueChange` for `EditableValueHolder` components.
`execute`	A collection that identifies a list of components to be executed on the server. A space-delimited `String` of component identifiers can be specified as the value for this attribute, or a `ValueExpression` (JSF EL) can be specified. The default value is `@this`, meaning the parent component of the Ajax behavior.
`immediate`	Boolean value indicating whether the input values are processed early in the life cycle. If `true`, then the values are processed, and their corresponding events will be broadcast during the Apply Request Values phase; otherwise, the events will be broadcast during the Invoke Application phase.
`listener`	Name of the listener method that is called when an `AjaxBehaviorEvent` has been broadcast for the listener.
`onevent`	Name of the JavaScript function used to handle UI events.

(*continued*)

Table 3-12. (*continued*)

Attribute	Description
`onerror`	Name of the JavaScript function used to handle errors.
`resetValues`	If true, then this particular Ajax transaction will reset the values.
`render`	Collection that identifies the components to be rendered on the client when the Ajax behavior is complete. A space-delimited `String` of component identifiers can be specified as the value for this attribute, or a `ValueExpression` (JSF EL) can be specified. The default value is `@none`, meaning that no components will be rendered when the Ajax behavior is complete.

The `execute` and `render` attributes of the `f:ajax` tag can specify a number of keywords to indicate which components are executed on the server for the Ajax behavior or which are rendered again after the Ajax behavior is complete, respectively. Table 3-13 lists the values that can be specified for both of these two attributes.

Table 3-13. *f:ajax Tag execute and render Attribute Values*

Attribute Value	Description
`@all`	All component identifiers are executed on the server, and all component identifiers are re-rendered once Ajax behavior is complete.
`@form`	The form that encloses the component.
`@none`	No component identifiers (default for the `render` attribute).
`@this`	The Ajax behavior parent component.
`@child(n)`	The nth child of the base component.
`@composite`	Closest composite component ancestor of the base component.
`@id(id)`	All component descendants of the base component with the specified id.
`@namingcontainer`	Closest NamingContainer ancestor of the base component.
`@next`	Next component in view after the base component.

(*continued*)

Table 3-13. (*continued*)

Attribute Value	Description
`@parent`	Parent of the base component.
`@previous`	Previous component to the base component.
`@root`	UIViewRoot.
Component IDs	Space-separated list of individual component identifiers.
JSF EL	Expression that resolves to a collection of string identifiers.

In the example for this recipe, an `f:ajax` tag has been embedded inside many of the input components within the form. Each of those components has been Ajaxified, in that the data entered as the values for the components will now have the ability to be processed using the JavaScript resource library associated with JSF. Behind the scenes, the `jsf.ajax.request()` method of the JavaScript resource library will collect the data for each component that has been Ajaxified and post the request to the JavaServer Faces life cycle. *In effect, the data is sent to the controller class property without submitting the page in a traditional fashion.* Notice that the `event` attribute specifies a JavaScript event that will be used to trigger the Ajax behavior. The JavaScript events that can be specified for the `event` attribute are those same JavaScript `event` attributes that are available on the parent component's tag, but the `on` prefix has been removed. For instance, if you want to perform an Ajax behavior on an `inputText` component when it loses focus, you would specify `blur` for the `f:ajax` event attribute rather than `onBlur`. Applying this concept to the example, when a user leaves the first or last name field, they will be validated using their associated `f:validate` tags immediately because the `f:ajax` tag has been embedded in them and the event on the `f:ajax` tag is specified as `blur`. When the Ajax behavior (the validation in this case) is complete, then the components whose identifiers are specified in the `f:ajax` render attribute will be re-rendered. In the case of the first and last `inputText` fields, their associated message components will be re-rendered, displaying any errors that may have occurred during validation.

UTILIZING AN ACTION LISTENER

It is possible to bind an action listener to an `f:ajax` tag so that when the invoking action occurs, the listener method is invoked. Why would you want to bind an action listener? There are any reasons to do so. For instance, suppose you wanted to capture the text that a user is typing into a text field. You could do so by binding an action method within a controller class to the listener attribute of an `inputText` field's corresponding `f:ajax` tag and then obtaining the current component's value from the `AjaxBehaviorEvent` object within the action method. For instance, suppose that you wanted to test a password for complexity and display a corresponding message indicating whether a password was strong enough. The `inputSecret` component for the password could be modified to include an `f:ajax` tag with an event specification of `keyup` and a listener specified as `#{ch3ContactController.passwordStrength}`, as the following listing demonstrates:

Within the view:

```
<h:outputLabel for="password" value="Enter a password for site access: "/>
<h:inputSecret id="password" size="40"
       value="#{ch3ContactController.current.password}">
    <f:validateRequired/>
    <f:ajax event="keyup" listener="#{ch3ContactController.passwordStrength}"
                                          render="passwordStrengthMessage"/>
</h:inputSecret>
...
```

Within the controller:

```
. . .
private String passwordStrengthMessage;
. . .
public void passwordStrength(AjaxBehaviorEvent event){
        UIInput password = (UIInput) event.getComponent();
        boolean isStrong = false;
        String input = password.getValue().toString();

        if(input.matches("((?=.*\\d)(?=.*[a-z])(?=.*[A-Z]).{6,})")) {
            isStrong = true;
        }
```

```
            if(isStrong == true){
                setPasswordStrengthMessage("Password is strong");
            } else {
                setPasswordStrengthMessage("Password is weak");
            }
        }
```

The code in this example would create a listener event that, when a user types a value, would check the present entry to determine whether it met the given criteria for a secure password. A message would then be displayed to the user to let them know whether the password was secure.

Using the `f:ajax` tag makes it easy to add Ajax behavior to a JSF component. Before the `f:ajax` tag, special third-party JavaScript libraries were often used to incorporate similar behaviors within JSF views. `f:ajax` adds the benefit of allowing the developer to choose between using Ajax behaviors, without the need for coding a single line of JavaScript.

3-6. Submitting Pages Without Page Reloads

Problem

You want to enable your input form to have the ability to submit input fields for processing without reloading the page. In essence, you want your web application input form to react more like that of a desktop application rather than navigating from page to page in order to process data.

Solution

Embed an `<f:ajax/>` tag within the command component in the view so that the CDI controller class action is invoked without the page being submitted. Enable `f:ajax` to update the `messages` component in the view so that any errors or success messages that result from the processing can be displayed. In this example, the newsletter subscription page for the Acme Bookstore will be changed so that the form is submitted using Ajax, and the `commandButton` component is processed without submitting the form in a

traditional manner. The following excerpt from the newsletter subscription form sources from `recipe03_06.xhtml`, which demonstrates how to add Ajax functionality to the `action` components within the form:

```
<h:commandButton id="contactSubmit" action="#{ch3ContactController.
subscribe}"
                 value="Save">
    <f:ajax event="action" execute="@form" render="@all"/>
</h:commandButton>
<h:panelGrid  columns="2" width="400px;">
```

When the button or link is clicked, JavaScript will be used in the background to process the request so that the results will be displayed immediately without needing to refresh the page.

How It Works

The user experience for web applications has traditionally involved a point, click, and page refresh mantra. While this type of experience is not particularly a bad one, it is not as nice as the immediate response that is oftentimes presented within a native desktop application. The use of Ajax within web applications has helped create a more unified user experience, allowing a web application the ability to produce an "immediate" response much like that of a native desktop application. Field validation (covered in Recipe 3-5) is a great candidate for immediate feedback, but another area where immediate responses work well is when forms are being submitted.

The `f:ajax` tag can be embedded in an `action` component in order to invoke the corresponding action method using JavaScript behind the scenes. The `f:ajax` tag contains a number of attributes, covered in Table 3-12 (see Recipe 3-5), that can be used to invoke Ajax behavior given a specified event and re-render view components when that Ajax behavior is complete. Please refer to Table 3-13 to see the values that can be specified for the `execute` and `render` attributes of the `f:ajax` tag.

In the example for this recipe, the `commandButton` component with an identifier of `contactSubmit` contains an `f:ajax` tag that specifies the `event` attribute as `action`, the `execute` attribute as `@form`, and the `render` attribute as `@all`. This means that when the button is invoked, the `ch3ContactController.subscribe` method will be called asynchronously using JavaScript, and it will send all the input component values

from the form to the server (controller class) for processing. When the Ajax behavior (`subscribe` method) is complete, all of the components within the view will be re-rendered. By re-rendering all the components in the view, this allows those message components to display any messages that have been queued up as a result of failed validation or a successful form submission. It is possible to process or render only specified components during an Ajax behavior; to learn more about doing so, please see Recipe 3-7.

Note The `event` attribute has a default value of `action` when the `f:ajax` tag is embedded within a `UICommand` component. However, it is specified in the code for this example for consistency.

3-7. Making Partial-Page Updates

Problem

You want to execute only a section of a page using an Ajax event and then render the corresponding section's components when the Ajax behavior is complete.

Solution

Use the `f:ajax` tag to add Ajax functionality to the components that you want to execute and render when the Ajax behavior is completed. Specify only the component identifiers corresponding to those components, or `@form`, `@this`, or one of the other execute keywords, for the `f:ajax` tag `execute` attribute. Likewise, specify only the component identifiers for the corresponding `message` components within the `render` attribute.

Suppose that the Acme Bookstore wants to execute the submission of the newsletter subscription form values and update the form's global message only when the submission is complete. The following `commandButton` component would execute only the form in which it is placed and the component corresponding to the identifier `newsletterSubscriptionMsgs`:

```
<h:commandButton id="contactSubmit" action="#{ch3ContactController.
subscribe}" value="Save">
    <f:ajax event="action" execute="@form" render="newsletterSubscription
    Msgs"/>
</h:commandButton>
```

When the button is clicked, the current form component values will be processed with the request, and the `ContactController` controller class `subscribe()` method will be invoked. Once the `subscribe()` method is complete, the component within the form that contains an identifier of `newsletterSubscriptionMsgs` (in this case a messages component) will be re-rendered.

Note In the case of the newsletter subscription form for the Acme Bookstore, a partial-page render upon completion is a bad idea. This is because the form will never be submitted if the values within the form do not validate correctly. In this case, if some of the form values do not validate correctly, then nothing will be displayed on the page when the save button is clicked because the `subscribe` method will never be invoked. If the `f:ajax` tag's `render` attribute is set to `@all`, then all of the components that failed validation will have a corresponding error message that is displayed. This example should demonstrate how important it is to process the appropriate portions of the page for the result you are trying to achieve.

How It Works

The `f:ajax` tag makes it simple to perform partial-page updates. To do so, specify the identifiers for those components that you want to execute for the `f:ajax execute` attribute. As mentioned in the example for this recipe, suppose you want to execute only a portion of a page, rather than all of the components on the given page. You could do so by identifying the components that you want to execute within the view, specifying them within the `f:ajax execute` attribute, and then rendering the corresponding message components when the Ajax behavior was completed. If nothing is specified for an `f:ajax` execute attribute, then the `f:ajax` tag must be embedded inside a component, in which case the parent component would be executed. Such is the default behavior

for the `f:ajax execute` attribute. In the example, the `execute` attribute of the `f:ajax` tag specifies the `@form` keyword, rather than a specific component `id`. As mentioned previously, a number of keywords can be specified for both the `execute` and `render` attributes of the `f:ajax` tag. Those keywords are listed in Table 3-13, which describes that the `@form` keyword indicates that all components within the same form as the given `f:ajax` tag will be executed when the Ajax behavior occurs. Therefore, all fields within the newsletter subscription form in this example will be sent to the controller class for processing when the button is clicked.

The same holds true for the `render` attribute, and once the Ajax behavior has completed, any component specified for the `render` attribute of the `f:ajax` tag will be re-rendered. Thus, if a validation occurs when a component is being processed because of the result of an `f:ajax` method call, a corresponding validation failure message can be displayed on the page after the validation fails. Any component can be rendered again, and the same keywords that can be specified for the `execute` attribute can also be used for the `render` attribute. In the example, the `newsletterSubscriptonMsgs` component is rendered once the Ajax behavior is completed.

3-8. Applying Ajax Functionality to a Group of Components

Problem

You want to apply Ajax functionality to a group of input components, rather than to each component separately.

Solution

Enclose any components to which you want to apply Ajax functionality within an `f:ajax` tag. The `f:ajax` tag can be the parent to one or more JSF components, in which case each of the child components inherits the given Ajax behavior. Applying Ajax functionality to multiple components is demonstrated in the following code listing. In the following example excerpt, the newsletter subscription view of the Acme Bookstore application is adjusted so that each of the `inputText` components that contains a validator is enclosed by a single `f:ajax` tag. Given that each of the `inputText`

components is embodied within the same f:ajax tag, the f:ajax render attribute has been set to specify the message component for each of the corresponding inputText fields in the group:

```
<ui:define name="content">
    <h:form id="contactForm">
        <h1>Subscribe to Newsletter</h1>
        <p>
            <h:outputText id="newsletterSubscriptionDesc"
                          value="#{ch3ContactController.
                          newsletterDescription}"/>
        </p>

        <br/>
        <h:messages id="newsletterSubscriptionMsgs" global Only="true"
        errorStyle="color: red" infoStyle="color: green"/>
        <br/>
        <f:ajax event="blur" render="firstError lastError emailError
        genderError passwordError passwordConfirmError">
            <h:panelGrid columns="2" bgcolor="" border="0">
                <h:panelGroup>
                    <h:outputLabel for="first" value="First: "/>
                    <h:inputText id="first" size="40"
                    value="#{ch3ContactController.current.first}">
                        <f:validateLength minimum="1" maximum="40"/>

                    </h:inputText>
                </h:panelGroup>
                <h:panelGroup>

                    <h:outputLabel for="last" value="Last: "/>
                    <h:inputText id="last" size="40"
                    value="#{ch3ContactController.current.last}">
                        <f:validateLength minimum="1" maximum="40"/>

                    </h:inputText>
                </h:panelGroup>
```

```
            <h:message id="firstError"
                       for="first"
                       errorStyle="color:red"/>

            <h:message id="lastError"
                       for="last"
                       errorStyle="color:red"/>
            <h:panelGroup>
                <h:outputLabel for="email" value="Email: "/>
                <h:inputText id="email" size="40"
                value="#{ch3ContactController.current.email}">

                </h:inputText>
            </h:panelGroup>
            <h:panelGroup/>
            <h:message id="emailError"
                       for="email"
                       errorStyle="color:red"/>
            <h:panelGroup/>

            <h:selectOneRadio title="Gender" id="gender"
            value="#{ch3ContactController.current.gender}">
                <f:selectItem  itemValue="M" itemLabel="Male"/>
                <f:selectItem itemValue="F" itemLabel="Female"/>
            </h:selectOneRadio>
            <h:panelGroup>
                <h:outputLabel for="occupation" value="Occupation: "/>
                <h:selectOneMenu id="occupation"
                value="#{ch3ContactController.current.occupation}">
                    <f:selectItems value="#{ch3ContactController.
                    occupationList}"/>
                </h:selectOneMenu>
            </h:panelGroup>
            <h:message id="genderError"
                       for="gender"
                       errorStyle="color:red"/>

        </h:panelGrid>
```

```
<br/>
<h:outputLabel for="description" value="Enter your book
interests"/>
<br/>
<h:inputTextarea id="description" rows="5" cols="75"
 value="#{ch3ContactController.current.description}"/>

<br/>
<h:panelGrid columns="2">
    <h:outputLabel for="password" value="Enter a password for
    site access: "/>
    <h:inputSecret id="password" size="40"
    value="#{ch3ContactController.current.password}">
        <f:validateRequired/>
        <f:ajax event="keyup" listener="#{ch3ContactController.
        passwordStrength}" render="passwordStrengthMessage"/>
    </h:inputSecret>

    <h:outputLabel for="passwordConfirm" value="Confirm
    Password: "/>
    <h:inputSecret id="passwordConfirm" size="40"
    value="#{ch3ContactController.passwordConfirm}"
                    validator="#{ch3ContactController.
                    validatePassword}">

    </h:inputSecret>
</h:panelGrid>
<h:panelGroup>
    <h:outputText id="passwordStrengthMessage"
    value="#{ch3ContactController.passwordStrengthMessage}"/>
    <h:message id="passwordError"
               for="password"
               style="color:red"/>
</h:panelGroup>
<br/>
```

```
<h:message id="passwordConfirmError"
           for="passwordConfirm"
           style="color:red"/>
<br/>
<hr/>
<br/>

<h:panelGrid columns="3">
    <h:panelGroup>
        <h:outputLabel for="newsletterList"
        value="Newsletters:" style=" "/>
        <h:selectManyListbox id="newsletterList"
        value="#{ch3ContactController.current.newsletterList}">
            <f:selectItems value="#{ch3ContactController.
            newsletterList}"/>
        </h:selectManyListbox>
    </h:panelGroup>
    <h:panelGroup/>
    <h:panelGroup>
        <h:panelGrid columns="1">
            <h:panelGroup>
                <h:outputLabel for="notifyme" value="Would you
                like to receive other promotional email?"/>
                <h:selectBooleanCheckbox id="notifyme"
                value="#{ch3ContactController.current.
                receiveNotifications}"/>
            </h:panelGroup>
            <h:panelGroup/>
            <hr/>
            <h:panelGroup/>
            <h:panelGroup>
                <h:outputLabel for="notificationTypes"
                value="What type of notifications are you
                interested in recieving?"/>
                <br/>
```

```
                        <h:selectManyCheckbox id="notifyTypes"
                        value="#{ch3ContactController.current.
                        notificationType}">
                            <f:selectItems value="#{ch3Contact
                            Controller.notificationTypes}"/>
                        </h:selectManyCheckbox>
                    </h:panelGroup>
                </h:panelGrid>
            </h:panelGroup>
        </h:panelGrid>
        <hr/>
        <br/>
    </f:ajax>
    <h:commandButton id="contactSubmit" action="#{ch3ContactController.
    subscribe}" value="Save">
        <f:ajax event="action" execute="@form" render="@all"/>
    </h:commandButton>
    <h:panelGrid  columns="2" width="400px;">
        <h:commandLink id="manageAccount" action="#{ch3Contact
        Controller.manage}" value="Manage Subscription">
            <f:ajax event="action" execute="@this" render="@all"/>
        </h:commandLink>
        <h:outputLink id="homeLink" value="home.xhtml">Home
        </h:outputLink>
    </h:panelGrid>
</h:form>
</ui:define>
```

When the page is rendered, each component will react separately given their associated validations. That is, if validation fails for one component, only the `message` component that corresponds with the component failing validation will be displayed, although each component identified within the `f:ajax render` attribute will be re-rendered.

Note As a result of specifying a global `f:ajax` tag, the password component can now execute two Ajax requests. One of the Ajax requests for the field is responsible for validating to ensure that the field is not blank, and the other is responsible for ensuring that the given password `String` is strong.

How It Works

Grouping multiple components with the same Ajax behavior has its benefits. For one, if the behavior needs to be adjusted for any reason, one change can now be made to the Ajax behavior, and each of the components in the group can benefit from the single adjustment. However, the `f:ajax` tag is smart enough to enable each component to still utilize separate functionality, such as validation or actions, so each can still have their own customized Ajax behavior. To group components under a single `f:ajax` tag, they must be added to the view as subelements of the `f:ajax` tag. That is, any child components must be enclosed between the opening and closing `f:ajax` tags. All of the enclosed components will then use Ajax to send requests to the server using JavaScript in an asynchronous fashion.

In the example for this recipe, a handful of the `inputText` components within the newsletter subscription view have been embodied inside an `f:ajax` tag so that their values will be validated using server-side bean validation when they lose focus. The `f:ajax` tag that is used to group the components has an `event` attribute set to `blur`, and its `render` attribute contains the `String`-based identifier for each of the `message` components corresponding to the components that are included in the group. The space-separated list of component `id`s is used to re-render each of the message components when the Ajax behavior is complete, displaying any errors that occur as a result of the validation.

3-9. Custom Processing of Ajax Functionality

Problem

You want to customize the Ajax processing for JSF components within a view in your application.

Solution

Write the JavaScript that will be used for processing your request, and utilize the `jsf.ajax.request()` function along with one of the standard JavaScript event-handling attributes for a JSF component. The following example is the JSF view for the newsletter subscription page for the Acme Bookstore application. All of the `f:ajax` tags that were previously used for validating `inputText` fields (Recipe 3-1) have been removed, and the `onblur` attribute of each `inputText` component has been set to use the `jsf.ajax.request()` method in order to Ajaxify the component. The following excerpt is taken from the view named `recipe03_09.xhtml`, representing the updated newsletter subscription JSF view:

```
...
                    <h:outputScript name="jsf.js" library="javax.faces"
                    target="head"/>
                    <h1>Subscribe to Newsletter</h1>
                    <p>
                        <h:outputText id="newsletterSubscriptionDesc"
                                      value="#{ch3ContactController.
                                      newsletterDescription}"/>
                    </p>

                    <br/>
                    <h:messages id="newsletterSubscriptionMsgs"
                    globalOnly="true" errorStyle="color: red"
                    infoStyle="color: green"/>
                    <br/>

                    <h:panelGrid columns="2" bgcolor="" border="0">
                        <h:panelGroup>
                            <h:outputLabel for="first" value="First: "/>
                            <h:inputText id="first" size="40"
                            value="#{ch3ContactController.current.first}"
                                        onblur="jsf.ajax.request(this,
                                        event, {execute: 'first', render:
                                        'firstError'});
```

```
 return false;">
        <f:validateLength minimum="1"
        maximum="40"/>
    </h:inputText>
</h:panelGroup>
<h:panelGroup>
    <h:outputLabel for="last" value="Last: "/>
    <h:inputText id="last" size="40"
    value="#{ch3ContactController.current.last}"
                onblur="jsf.ajax.request(this,
                event, {execute: 'last', render:
                'lastError'});
 return false;">
        <f:validateLength minimum="1"
        maximum="40"/>
    </h:inputText>
</h:panelGroup>

<h:message id="firstError"
           for="first"
           errorStyle="color:red"/>

<h:message id="lastError"
           for="last"
           errorStyle="color:red"/>
<h:panelGroup>
    <h:outputLabel for="email" value="Email: "/>
    <h:inputText id="email" size="40"
    value="#{ch3ContactController.current.email}"
                onblur="jsf.ajax.request(this,
                event, {execute: 'email', render:
                'emailError'});
 return false;"/>
</h:panelGroup>
<h:panelGroup/>
<h:message id="emailError"
```

```
                    for="email"
                    errorStyle="color:red"/>
                <h:panelGroup/>
...
```

Note The `<h:panelGroup/>` tag is used to add a placeholder panel group to the grid for spacing purposes.

Using this technique, the `inputText` components that specify Ajax behavior for the `onblur` event will asynchronously have their values validated when they lose focus. If any custom JavaScript code needs to be used, it can be added to the same inline JavaScript call to `jsf.ajax.request()`.

Note Method calls to CDI controllers cannot be made using the `jsf.ajax.request()` technique, so it is not possible to invoke a listener explicitly with the Ajax request.

How It Works

The JavaScript API method `jsf.ajax.request()` can be accessed directly by a Facelets application, enabling a developer to have slightly more control than using the `f:ajax` tag. Behind the scenes, the `f:ajax` tag is converted into a call to `jsf.ajax.request()`, sending the parameters as specified via the tag's attributes. To use this technique, you must include the `jsf.js` library within the view. A JSF `outputScript` tag should be included in the view, specifying `jsf.js` as the script `name` and `javax.faces` as the `library`. The `jsf.js` script within this example will be placed in the head of the view, which is done by specifying `head` for the `target` attribute of the `outputScript` tag. The following excerpt from the example demonstrates what the tag should look like:

```
<h:outputScript name="jsf.js" library="javax.faces" target="head"/>
```

Note To avoid nested IDs, it is a good idea to specify the `h:form` attribute of `prependId="false"` when using `jsf.ajax.request()` manually. For instance, the form tag should look as follows:

```
<h:form prependId="false">
```

The `jsf.ajax.request()` method can be called inline, as is the case with the example for this recipe, and it can be invoked from within any of the JavaScript `event` attributes of a given component. The format for calling the JavaScript method is as follows:

```
jsf.ajax.request(component, event,{execute:'id or keyword', render:'id or
keyword'});
```

Usually when the request is made using an inline call, the `this` keyword is specified for the first parameter, signifying that the current component should be passed. The `event` keyword is passed as the second parameter, and it passes with it the current event that is occurring against the component. Lastly, a map of name-value pairs is passed, specifying the `execute` and `render` attributes along with the component identifiers or keywords that should be executed and rendered after the execution completes, respectively. For a list of the valid keywords that can be used, please refer to Table 3-2 within the introduction to this chapter.

Note You can also utilize the `jsf.ajax.request` method from within a controller class by specifying the `@ResourceDependency` annotation (https://jakarta.ee/specifications/faces/2.3/apidocs/javax/faces/application/ResourceDependency.html) as follows:

```
@ResourceDependency(name="jsf.js" library="javax.faces"
target="head")
```

3-10. Listening for System-Level Events

Problem

You want to invoke a method within your application whenever a system-level event occurs.

Solution

Create a system event listener class by implementing the `SystemEventListener` interface and overriding the `processEvent(SystemEvent event)` and `isListenerForSource(Object source)` methods. Implement these methods accordingly to perform the desired event processing. The following code listing is for a class named `BookstoreAppListener`, and it is invoked when the application is started up or when it is shutting down:

```
public class BookstoreAppListener implements SystemEventListener {

    @Override
    public void processEvent(SystemEvent event) throws
    AbortProcessingException {
        if(event instanceof PostConstructApplicationEvent){
            System.out.println("The application has been constructed...");
        }

        if(event instanceof PreDestroyApplicationEvent){
            System.out.println("The application is being destroyed...");
        }
    }

    @Override
    public boolean isListenerForSource(Object source) {
        return(source instanceof Application);
    }

}
```

Next, the system event listener must be registered in the `faces-config.xml` file. The following excerpt is taken from the `faces-config.xml` file for the Acme Bookstore application:

```
...
<application>

              <system-event-listener>
                            <system-event-listener-class>
                                          org.jakartaeerecipes.
                                          chapter03.recipe03_10.
                                          BookstoreAppListener
                             </system-event-listener-class>
                             <system-event-class>
                                          javax.faces.event.
                                          PostConstructApplicationEvent
                             </system-event-class
              </system-event-listener>

              <system-event-listener>
                            <system-event-listener-class>
                                          org.jakartaeerecipes.
                                          chapter03.recipe03_10.
                                          BookstoreAppListener
                            </system-event-listener-class>
                            <system-event-class>
                                          javax.faces.event.
                                          PreDestroyApplicationEvent
                            </system-event-class
        </system-event-listener>

    </application>
...
```

When the application is started, the message "The application has been constructed…" will be displayed in the server log. When the application is shutting down, the message "The application is being destroyed…" will be displayed in the server log.

How It Works

The ability to perform tasks when an application starts up can sometimes be useful. For instance, let's say you'd like to have an email sent to the application administrator each time the application starts. You can do this by performing the task of sending an email within a class that implements the `SystemEventListener` interface. A class that implements `SystemEventListener` must then override two methods, `processEvent(SystemEvent event)` and `isListenerForSource(Object source)`. The `processEvent()` method is where the real action occurs, because it is the method into which your custom code should be placed. Whenever a system event occurs, the `processEvent()` method is invoked. In this method, you will need to perform a check to determine what type of event has occurred so that you can process only those events that are pertinent. To determine the event that has occurred, perform an `instanceof()` check on the `SystemEvent` object. In the example, there are two `if` statements used to determine the type of event that is occurring and to print a different message for each. If the event type is of `PostConstructApplicationEvent`, then that means the application is being constructed. Otherwise, if the event type is of `PreDestroyApplicationEvent`, the application is about to be destroyed. The `PostConstructApplicationEvent` event is called just after the application has been constructed, and `PreDestroyApplicationEvent` is called just prior to the application destruction.

The other method that must be overridden within the `SystemEventListener` class is named `isListenerForSource()`. This method must return `true` if this listener instance is interested in receiving events from the instance referenced by the source parameter. Since the example class is built to listen for system events for the application, a `true` value is returned if the source parameter is an instance of `Application`.

After the system event listener class has been written, it needs to be registered with the application. In the example, you want to listen for both the `PostConstructApplicationEvent` and the `PreDestroyApplicationEvent`, so there needs to be a `system-event-listener` element added to the `faces-config.xml` file for each of these events. Within the `system-event-listener` element, specify the name of the event listener class within a `system-event-listener-class` element and the name of the event within a `system-event-class` element.

3-11. Listening for Component Events

Problem

You want to invoke a listener method when a specified component event is occurring. For instance, you want to listen for a component render event.

Solution

Embed an `f:event` tag within the component for which you want to listen for events. The `f:event` tag allows components to invoke controller class listener methods based upon the current component state. For instance, if a component is being rendered or validated, a specified listener method could be invoked. In the example for this recipe, an `outputText` component is added to the book view of the Acme Bookstore application to specify whether the current book is in the user's shopping cart. When the `outputText` component is being rendered, a component listener is invoked that checks the current state of the cart to see whether the book is contained within it. If it is in the cart, then the `outputText` component will render a message stating so; if not, then the `outputText` component will render a message stating that it is not in the cart.

The following excerpt is taken from a view named `recipe03_10.xhtml`, a derivative of the book view for the application. It demonstrates the use of the `f:event` tag within a component. Note that the `outputText` component contains no `value` attribute because the value will be set within the event listener:

```
...
<h:outputText id="isInCart" style="font-style: italic; color: ">
    <f:event type="preRenderComponent" listener="#{ch3CartController.
    isBookInCart}"/>
</h:outputText>
...
```

The `CartController` class contains a method named `isBookInCart(ComponentSystemEvent)`. The `f:event` tag in the view references this listener method via the `CartController` controller name `ch3CartController`. The listener method is responsible for constructing the text that will be displayed in the `outputText` component:

```
public void isBookInCart(ComponentSystemEvent event) {
        UIOutput output = (UIOutput) event.getComponent();
        if (cart != null) {
            if (searchCart(authorController.getCurrentBook()
            .getTitle()) > 0) {
                output.setValue("This book is currently in your cart.");
            } else {
                output.setValue("This book is not in your cart.");
            }
        } else {
            output.setValue("This book is not in your cart.");
        }
    }
```

How It Works

Everything that occurs within JSF applications is governed by the JSF application life cycle. As part of the life cycle, JSF components go through different phases throughout their lifetimes. Listeners can be added to JSF components to perform different tasks when a given phase is beginning or ending. There are two pieces to the puzzle for creating a component listener: the tag that is embedded within the component for which your listener will perform tasks and the listener method itself. To add a listener to a component, the `f:event` tag should be embedded within the opening and closing tags of the component that will be interrogated. The `f:event` tag contains a handful of attributes, but only two of them are mandatory for use: `type` and `listener`. The `type` attribute specifies the type of event that will be listened for, and the `listener` attribute specifies the controller class listener method that will be invoked when that event occurs. The valid values that could be specified for the `name` attribute are `preRenderComponent`, `postAddToView`, `preValidate`, and `postValidate`. In addition to these event values, any Java class that extends `javax.faces.event.ComponentSystemEvent` can also be specified for the `name` attribute.

The listener method must accept a `ComponentSystemEvent` object. In the example, the listener checks to see whether the shopping cart is `null`, and if it is, then a message indicating an empty cart will be set for the `outputText` component's value. Otherwise, if the cart is not empty, then the method looks through the `List` of books in the cart to

see whether the currently selected book is in the cart. A message indicating whether the book is in the cart is then added to the value of the `outputText` component. Via the listener, the actual value of the component was manipulated. Such a technique could be used in various ways to alter components to suit the needs of the situation.

3-12. Developing a Page Flow

Problem

You want to develop a flow of pages within your application that share information with one another.

Solution

Define a page flow using the faces flow technology, a solution that allows a defined set of views to be interrelated with one another to share a common set of data, and views outside of the flow do not have access to the flow's data. Flows also have their own set of navigational logic, so they are almost like a subprogram within an application. To enable an application to utilize faces flow, a `<flow-definition>` section should be added to the `faces-config.xml` file. The section can be empty, because the navigational logic can instead reside in a separate configuration file for the flow. The following `faces-config.xml` file demonstrates how to enable faces flow for an application:

```
<faces-config version="2.3"
       xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_3.xsd">
 ...
<flow-definition>
    </flow-definition>
     ...
</faces-config>
```

The views belonging to a flow should be separated from the rest of the application views and placed into a folder at the root of the application's web directory. The folder containing the flow views should be named the same as the flow identifier. Navigation and configuration code is contained within a separate XML configuration file that resides within the flow view directory, and the file is named `flowname-flow.xml`, where `flowname` is the flow identifier. The following configuration file demonstrates the configuration for a very basic flow identified by `exampleFlow`. You can find more information regarding the different elements that can be used within the flow configuration in the "How It Works" section:

```
<faces-config version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="
        http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">

    <flow-definition id="exampleFlow">

    </flow-definition>

</faces-config>
```

The views belonging to the flow should reside within the flow folder alongside the flow configuration file. Each of the views can access a controller class that is dedicated to facilitating the flow. The flows share a context that begins when the flow is accessed and ends when the flow exits. The following view demonstrates the entry point to a flow named `exampleFlow`. This example view can be found in the book sources in the file `recipes03_12.xhtml`:

```
<ui:composition xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
                xmlns:f="http://xmlns.jcp.org/jsf/core"
                xmlns:h="http://xmlns.jcp.org/jsf/html"
                template="layout/custom_template_search.xhtml">
    <ui:define name="content">
        <h:messages globalOnly="true"  errorStyle="color: red"
        infoStyle="color: green"/>
```

```
        <h:form id="flowForm">
            <p>
                Faces Flow Example
            </p>
            <h:commandButton value="Begin Flow" action="exampleFlow"/>
            <h:commandButton value="Stay Here" action="stay"/>

        </h:form>
    </ui:define>
</ui:composition>
```

Next, let's take a look at a view that is accessing the controller class that is dedicated to the flow. In the following view, the controller class named `FlowBean` is accessed to invoke a method, which will return an implicit navigational `String` directing the application to the next view in the flow. Notice that this view also accesses the `facesContext.application.flowHandler`, which I will discuss more in the "How It Works" section:

```
<h:body>
        <f:view>
            <h:form>
        <p>
            This is the first view of the flow.
            <br/><br/>
            Flow ID: #{facesContext.application.flowHandler.currentFlow.id}
            <br/>
            <h:commandLink value="Go to another view in the flow"
            action="#{flowBean.navMethod()}"/>
        </p>
            </h:form>
        </f:view>
    </h:body>
```

Each subsequent view within the flow can also access the resources of the flow's controller class. Lastly, you'll look at the code that is contained within `org.jakartaeerecipes.chapter03.FlowBean`, which is the controller class that is dedicated to the flow:

```
import javax.faces.flow.FlowScoped;
import javax.inject.Named;

@Named
@FlowScoped("exampleFlow")
public class FlowBean implements java.io.Serializable {

    private String flowValue;
    private String parameter1;
    /**
     * Creates a new instance of FlowBean
     */
    public FlowBean() {
    }

    /**
     * Initializes the flow
     */

    public void initializeIt(){
        System.out.println("Initialize the flow...");
    }
    /**
     * Finalizes the flow
     */

    public void finalizeIt(){
        System.out.println("Finalize the flow...");
    }

    public String navMethod(){
        return "intermediateFlow";
    }
```

```
    public String testMethod(){
        return "intermediate";
    }

    public String endFlow(){
        return "endingFlow";
    }

    /**
     * @return the flowValue
     */
    public String getFlowValue() {
        return flowValue;
    }

    /**
     * @param flowValue the flowValue to set
     */
    public void setFlowValue(String flowValue) {
        this.flowValue = flowValue;
    }

    /**
     * @return the parameter1
     */
    public String getParameter1() {
        return parameter1;
    }

    /**
     * @param parameter1 the parameter1 to set
     */
    public void setParameter1(String parameter1) {
        this.parameter1 = parameter1;
    }
}
```

This solution provided a quick overview of the files that are required for creating a flow within a JSF application. In the next section, I'll cover the features in more detail.

How It Works

The concept of session management has been a difficult feat to tackle since the beginning of web applications. A *web flow* refers to a grouping of web views that are related and must have the ability to share information with each view within the flow. Many web frameworks have attempted to tackle this issue by creating different solutions that would facilitate the sharing of data across multiple views. Oftentimes, a mixture of session variables, request parameters, and cookies are used as a patchwork solution.

Since JSF 2.2, a solution has been adopted for binding multiple JSF views to each other, allowing them to share information among each other. This solution is referenced as *faces flow*; and it allows a group of interrelated views to belong to a *flow instance*, and information can be shared across all the views belonging to a flow instance. Flows contain separate navigation that pertains to the flow itself and not the entire application. As such, flow navigation can be defined in an XML format or via code. A flow contains a single point of entry, and it can be called from any point within an application.

Defining a Flow

As mentioned in the solution to this recipe, the `faces-config.xml` file for a JSF application that will utilize the flow feature must contain a `<flow-definition>` section. This section of the `faces-config.xml` file can contain information specific to one or more flows residing within an application. However, for the purposes of this recipe, the solution utilizes a separate XML configuration file for use with the flow. Either way will work; the syntax does vary just a bit because the XML configuration file that is flow-specific uses a new JSF `taglib` for accessing the flow-specific configuration tags.

Note To learn more about using the `faces-config.xml` file for flow configuration, please refer to the online documentation (https://docs.oracle.com/javaee/7/tutorial/jsf-configure003.htm).

Even if a flow is not using the `faces-config.xml` file for defining the flow configuration, the `<flow-definition>` section must exist to tell the JSF runtime that flows are utilized within the application.

The flow-specific configuration file and all flow-related views should reside within the same folder, at the root of the application's web directory. The name of the folder should be the same as the flow identifier. As mentioned in the solution, the flow configuration file should be named `flowname-flow.xml`, where `flowname` is the same as the flow identifier.

The Flow Controller Class

A flow contains its own controller class annotated as `@FlowScoped`, which differs from `@SessionScoped` because the data can be accessed only by other views (`ViewNodes`) belonging to the flow. The `@FlowScoped` annotation relies upon Contexts and Dependency Injection (CDI), because `FlowScoped` is a CDI scope that causes the runtime to consider classes with the `@FlowScoped` annotation to be in the scope of the specified flow. A `@FlowScoped` bean maintains a life cycle that begins and ends with a flow instance. Multiple flow instances can exist for a single application, and if a user begins a flow within one browser tab and then opens another, a new flow instance will begin in the new tab. This solution resolves many lingering issues around sessions and standard browsers that allow users to open multiple tabs. To maintain separate flow instances, the `ClientId` is used by JSF to differentiate among multiple instances.

Each flow can contain an `initializer` and a `finalizer` (i.e., a method that will be invoked when a flow is entered and a method that will be invoked when a flow is exited, respectively). To declare an initializer, specify a child element named `<initializer>` within the flow configuration `<flow-definition>`. The initializer element can be an EL expression that declares the controller class initializer method, as such:

```
...
<initializer>#{flowBean.initializeIt}></initializer>
...
```

Similarly, a `<finalizer>` element can be specified within the flow configuration to define the method that will be called when the flow is exited. The following demonstrates how to set the finalizer to an EL expression declaring the controller class finalizer method:

```
...
<finalizer>#{flowBean.finalizeIt}></finalizer>
...
```

Flows can contain method calls and variable values that are accessible only via the flow nodes. These methods and variables should be placed within the `FlowScoped` bean and used the same as standard controller class methods and variables. The main difference is that any method or variable that is defined within a `FlowScoped` bean is available only for a single flow instance.

Navigating Flow View Nodes

Flows contain their own navigational rules, which can be defined within the `faces-config.xml` file or the individual flow configuration files. These rules can be straightforward and produce a page-by-page navigation, or they can include conditional logic. There are a series of elements that can be specified within the navigation rules, which will facilitate conditional navigation. Table 3-14 lists the different elements, along with an explanation of what they do.

Table 3-14. *Flow Navigational Elements*

Element	Description
`view`	Navigates to a standard JSF view.
`switch`	Represents one or more EL expressions that conditionally evaluate to `true` or `false`. If `true`, then navigation occurs to the specified view node.
`flow-return`	Outcome determined by the caller of the flow.
`flow-call`	Represents a call to another flow; creates a nested flow.
`method-call`	Arbitrary method call that can invoke a method that returns a navigational outcome.

The following navigational sequence is an example of a flow navigation that contains conditional logic using the elements listed in Table 3-14:

```
<flow-definition>

    <start-node>exampleFlow</j:start-node>

    <switch id="startNode">
        <navigation-case>
            <if>#{flowBean.someCondition}</if>
                <from-outcome>newView</from-outcome>
        </navigation-case>
     </switch>

     <view id="oneFlow">
         <vdl-document>oneFlow.xhtml</vdl-document>
     </view>

     <flow-return id="exit">
         <navigation-case>
             <from-outcome>exitFlow</from-outcome>
         </navigation-case>
     </flow-return>

     <finalizer>#{flowBean.finalizeIt}</finalizer>


 </flow-definition>
```

Flow EL

Flows contain a new EL variable named `facesFlowScope`. This variable is associated with the current flow, and it is a map that can be used for storing arbitrary values for use within a flow. The key-value pairs can be stored and read via a JSF view or through Java code within a controller class. For example, to display the content for a particular map key, you could use the following:

```
The content for the key is:  #{facesFlowScope.myKey}
```

3-13. Broadcasting Messages from the Server to All Clients

Problem

Your organization has constructed a Jakarta EE application, and it is in use by a number of clients. You wish to have the ability to send a message from the server and have that message distributed to all of the clients at once.

Solution

Make use of the `f:websocket` tag, which was new with the release of JSF 2.3, to send a message to all listening clients. The following example includes a client view which contains a text box, a send button, and a `f:websocket` tag. The user can type a message into the text box and click the send button, and the typed message will be sent to all other clients that are currently listening on the same channel:

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

    <body>

        <ui:composition template="layout/custom_template_search.xhtml">
            <ui:define name="content">
                <h:messages globalOnly="true"  errorStyle="color: red"
                infoStyle="color: green"/>
                <h:form id="webSocketForm">
                    <script type="text/javascript">
                        function messageListener(message) {
                            document.getElementById("messageDiv").innerHTML
                            += message + "<br/>";
                        }
                    </script>
```

```
                    <p>
                        Websocket Integration Example
                    </p>
                    <p>
                        Enter text into the box below and press send
                        button.  This will send
                        a message to all connected clients.
                    </p>
                    <h:inputText id="websocketMessageText"
                    value="#{bookstoreController.messageText}"/>
                    <br/>
                    <h:commandButton id="sendMessage"
                    action="#{bookstoreController.sendMessage}"
                    value="Send">
                        <f:ajax/>
                    </h:commandButton>
                    <br/>
                    <f:websocket channel="messagePusher"
                    onmessage="messageListener" />

                    <div id="messageDiv"/>
                </h:form>
            </ui:define>
        </ui:composition>
    </body>
</html>
```

The following code shows the server-side code behind the `messagePusher` channel and the `bookstoreController.sendMessage()` method:

```
import java.util.Date;
import javax.enterprise.context.ApplicationScoped;
import javax.faces.push.Push;
import javax.faces.push.PushContext;
import javax.inject.Inject;
import javax.inject.Named;
```

```
@Named("bookstoreController")
@ApplicationScoped
public class BookstoreController {

    private Date dayAndTime = null;

    private int counter;

    @Inject
    @Push(channel="messagePusher")
    private PushContext push;

    private String messageText;
. . .
/**
     * Initiates a notification to all Websocket clients.  This method is
     used
     * for example 3-12.
     */
    public void sendMessage(){
        System.out.println("sending message");
        push.send(messageText);
        messageText = null;
    }
. . .
}
```

The resulting solution looks like the following. If one types and clicks send, all listening clients (on the same view) will receive the message.

How It Works

Websockets have become a standard protocol for client and server communication. There are a couple of different ways in which to implement Websocket solutions. One can utilize a framework such as Atmosphere to develop Websockets, or since the release of Java EE 7, the native Websocket support can be utilized. Both approaches are supported by the JSF Websocket. The support in JSF 2.3 includes both implementations,

so it provides some flexibility. To enable this support, one must specify the `javax.faces.ENABLE_WEBSOCKET_ENDPOINT` context parameter in the `web.xml` deployment descriptor with a value of `true`, as follows:

```
<context-param>
    <param-name>javax.faces.ENABLE_WEBSOCKET_ENDPOINT</param-name>
    <param-value>true</param-value>
</context-param>
```

The `f:websocket` tag enables support for Websockets within JSF client views. The tag includes a required `channel` attribute, which is a `ValueExpression` used to list the channel on which the Websocket client will listen. The tag also includes a required `onmessage` attribute, which is also a `ValueExpression`, and it is used to list the name of a JavaScript function that is to be executed when the Websocket message is received. In the example, you can see that the channel is set to `messagePusher`, meaning that the server must send message(s) to the channel named `messagePusher` in order to successfully send to this client. The message attribute is set to `messageListener`, and if you look at the JavaScript source that has been added to the view, you can see that it contains a function named `messageListener`. This function is executed when the message is received. In this example, the function merely prints a message to the `div` with an ID of `messageDiv` in the view. The signature of the JavaScript function in this example accepts the message only. However, a JavaScript function could also accept a channel name and event argument, if needed.

The `f:websocket` tag contains a number of other useful attributes as well. While optional, the following parameters may be of use in certain circumstances:

- `onclose`: Specifies a JavaScript function to invoke when the message is closed.
- `scope`: Used to specify a limit as to where messages are propagated. If set to session, this attribute limits the messages to all client views with the same websocket channel in the current session only.
- `port`: Specifies the TCP port number other than the HTTP port, if needed.

Now let's take a look at the server-side implementation. The solution to this recipe uses a new `PushContext`, which is injected into an `ApplicationScoped` bean. This `PushContext` is used to send the message to all listening clients, and it can be injected into any CDI bean by including the `@Push` annotation, along with the context. The name of the channel can be specified via an optional channel attribute on the `@Push` annotation; otherwise, it will assume the same name as the `PushContext` identifier. In the example, the `PushContext` is simply named "push." This is the channel on which all clients must listen.

To send a message, call upon the `send()` method of the `PushContext`, passing the message to be broadcast. The message will be encoded as JSON and delivered to the message argument of the JavaScript function on the client which corresponds to the function named in the `f:websocket onmessage` attribute. The message can be composed of any number of containers, including a plain String, List, Map, Object, and so on.

3-14. Programmatically Searching for Components

Problem

You wish to use Expression Language or Java code to find a particular component or a set of components within a JSF view. There are a number of reasons why you may wish to obtain access to components, such as invoking the component programmatically or referencing them from another component within the view.

Solution #1

Make use of the JSF component search framework via the use of expression language or programmatically from Java code. In the following example, a JSF `panelGrid` component is updated via expression language using key JSF search terms. The `f:ajax` tag contains a render attribute that specifies `@parent`, indicating that the parent component should be re-rendered once the Ajax process is complete:

```
<h:panelGrid columns="2">
                <h:outputLabel for="password" value="Enter a password for
                site access: "/>
```

```
            <h:inputSecret id="password" size="40"
            value="#{ch3ContactController.current.password}">
                <f:validateRequired/>
                <f:ajax event="blur" render="@parent"/>
            </h:inputSecret>
            <h:panelGroup/>
            <h:message id="passwordError"
                   for="password"
                   style="color:red"/>

            <h:outputLabel for="passwordConfirm" value="Confirm
            Password: "/>
            <h:inputSecret id="passwordConfirm" size="40"
            value="#{ch3ContactController.passwordConfirm}"
                           validator="#{ch3ContactController.
                           validatePassword}">
                <f:ajax   event="blur" render="@parent"/>
            </h:inputSecret>
            <h:panelGroup/>
            <h:message id="passwordConfirmError"
                   for="passwordConfirm"
                   style="color:red"/>
        </h:panelGrid>
```

Solution #2

Utilize the programmatic API to search for components from within a server-side CDI controller class. In the following solution, a button from a JSF view is used to invoke an action method in the CDI bean. The action method merely demonstrates the programmatic search expression API. In the action method, a component is looked up by explicit ID:

```
public void findById() {
        FacesContext context = FacesContext.getCurrentInstance();
        SearchExpressionContext searchContext = SearchExpressionContext.
                                                createSearchExpressionCo
                                                ntext(context, context.
                                                getViewRoot());
```

```
        context.getApplication()
                .getSearchExpressionHandler()
                .resolveComponent(
                   searchContext,
                   "passwordConfirm",
                   (ctx, target) -> out.print(target.getId()));
    }
```

How It Works

For years, JSF developers had difficulty referencing JSF components within a view by ID. There are a couple of problems that can be encountered if attempting to simply look up a component by ID. First, if an ID is not explicitly assigned to a JSF component, then the `FacesServlet` assigns one automatically. In this situation, the ID is unknown until runtime, and therefore it is almost impossible to reference the component using EL or from within Java code. Second, even if a JSF component is assigned a static ID, then the nesting architecture of JSF views and the JSF component tree causes the IDs of each parent component to be prepended to the ID of the child component. This can cause for long and sometimes difficult to maintain component IDs. Moreover, even if a specified component is easy to identify by prepending parent IDs, some components, such as those nested in tables, will still have a dynamic ID assigned at runtime.

There have been a number of third-party libraries that have developed solutions to combat this problem. OmniFaces and PrimeFaces are some of the most widely used. The addition of the JSF search expression API to JSF proper significantly reduces the work that needs to be done in order to gain access to JSF components within a view. This is especially the case in the event that a component is nested deep within other components in a view or part of a `dataTable` as mentioned previously. The search expression API allows one to utilize keywords to help search the component tree in a dynamic manner, rather than hard-coding static IDs that may change down the road.

Prior to JSF 2.3, there were four abstract search keywords that could be used to obtain reference to components, those being `"@all", "@this"`, `"@form"`, and `"@none"`. Moreover, one could only perform EL search expressions in the `f:ajax` tag. This was quite a limitation, and JSF 2.3 greatly expands this functionality. Please refer to Table 3-15 for the search keywords. The following features have been added to the search expression API:

- Keywords and search expressions can be used programmatically.
- Many more keywords have been added.
- Keywords accept arguments.
- Keywords are extendible and can be chained.

Table 3-15. *Search Keywords*

Keyword	Description
@child(n)	The nth child of the base component
@composite	Nearest composite component of the base
@id(id)	Nearest descendant of the base component with an id matching a specified value
@namingcontainer	Nearest naming container of the base component
@next	Next component in view following the base component
@parent	Parent of the base component
@previous	Previous component to the base
@root	The UIViewRoot

The solution demonstrates how to find components using the `@parent` keyword, but any of the others can be used and strung together in order to find desired components.

Another new feature with JSF 2.3 is the programmatic search expression API. This makes it possible to gain access to components from within the controller class. The second listing in the solution demonstrates how to use the programmatic API. To use the API, first create a `SearchExpressionContext`, which will later be passed as a parameter

to help find the component. Second, call upon the `FacesContext` to gain reference to the application via getApplication(), and then invoke `getSearchExpressionHandler().resolveComponent()`, passing the `SearchExpressionContext`, the search expression string, and the function to call when the component is found. This can be used to search for any component via a programmatic API.