Using JSF technology for XForms applications

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

This tutorial explains and demonstrates the use of JavaServer Faces (JSF) technology to develop XForms applications. The JSF API (JSR-127) allows enterprise Java developers to build user interfaces for their server-side applications. XForms (a W3C recommendation) is an XML application that provides a framework for designing XML-based data entry forms.

This tutorial covers XForms authoring requirements in a server-side Java application. It provides the inside-out picture of the JSF technology and shows how XForms authoring requirements fit into the JSF architecture. It also lists the tasks required if developers are going to use JSF technology for XForms authoring and demonstrates the development of a JSF tag library capable of authoring XForms markup.

We will also develop a sample application (XForms-JSF shopping cart) so you can see the concepts put to work in a real-world application.

JSF technology provides the API for authoring user interfaces, and XForms defines the markup that needs to be produced. Therefore, it is natural to expect that many Java application developers will need to author XForms markup using the JSF API. That's why we have chosen XForms as our sample application in this tutorial. However, the concepts presented are applicable to all types of JSF applications, so we've provided sample code to help answer your JSF technology questions.

Should I take this tutorial?

To take this tutorial, you should be a Java programmer with some knowledge of JavaBeans and the JavaServer Pages (JSP) technologies. This knowledge is necessary to understand most of the coding details and examples. You also need to be familiar with XML.

This tutorial will help you:

- Understand the JSF architecture
- Know how the JSF framework internally works
- Know what is happening inside a JSF application
- Comprehend the different phases of a JSF application
- Develop your own JSF tag library

This tutorial is not for readers who simply want to use the existing JSF core and HTML tag libraries. If you'd like to learn more about the existing JSF tag

libraries, there are some excellent resources available on *developerWorks*, as detailed in Resources on page200 .

Tutorial topics

This tutorial is divided into the following sections:

° Section 1 is the introduction and overview.

° Section 2 covers the XForms authoring requirements in server-side Java applications. To cover the authoring requirements, we'll use a shopping cart application scenario, and present the flow of data and user interactions in an XForms application. then we provide a list of requirements to fulfill if you are to use XForms in a server-side Java application.

° Section 3 offers a high-level view of the JSF architecture. This section lists the components required to build a JSF application. It covers many of the core JSF technology concepts without going into the low-level details of each concept.

° Section 4 provides low-level details of JSF technology concepts.

° Section 5 discusses the two important components you will build in almost all your JSF applications: model beans and event handlers. This section demonstrates the development of model beans to hold application data and event handlers to handle JSF events. By the end of Section 5, you should have enough knowledge about JSF technology to start building your own JSF tag libraries.

° Section 6 demonstrates how you can use JSF technology to fulfill the server-side requirements of XForms applications. This section will also present the strategy for XForms-JSF technology integration and demonstrate this strategy by developing three XForms-JSF components.

° Section 7 walks you through the development of many more XForms-JSF components using the concepts we learned in Section 6. At the end of this section, we will demonstrate how to build Java Archive (JAR) files to distribute your own tag libraries.

° Finally, in sections 8 and 9, you'll put your JSF and XForms technology knowledge to the test by building a comprehensive real-world application.

Code samples and installation requirements

To run the sample applications, you will need the following:

- J2SE V1.4.2
- J2EE V1.4.0 Update 1
- An application server
- An XForms browser

I tested the sample applications using J2SE V1.4.2, J2EE V1.4 (Update 1), Sun Java System Application Server Platform Edition V8, and Forms Player V1.0 (XForms browser) on Microsoft Windows 2000 Professional Edition Service Pack 2. You should be able to use any J2EE 1.4-compliant application server.

The sample applications are available in the source code download for this tutorial. This download contains three zip files: section6.zip, section7.zip, and section9.zip (see Resources on page 200). These files contain code for the sample applications developed in their respective sections. The steps to try the sample applications in an individual zip file are discussed in their relevant sections.

About the author

Faheem Khan is an independent software consultant specializing in enterprise application integration (EAI) and B2B solutions. You can reach him at *fkhan872@yahoo.com*.

Section 2. A sample XForms application

How is XForms different from HTML forms?

Almost every Web application contains forms to interact with its users and gather information. For this purpose, Web applications use HTML forms.

XForms is an advanced XML-based version of HTML forms. The World Wide Web Consortium (W3C) has prepared the official specification of XForms, which is currently a W3C recommendation. (See Resources on page 200 for the official XForms information.)

XForms is based on years of experience using HTML forms. You can say that XForms is the next-generation Web forms.

The major advantages of using XForms instead of traditional HTML forms:

- XForms separate the design of the user interface from the application data, which means the same application data can be gathered from Web-site users who are using different UIs. This feature lets us develop the UI independent of the data model. In HTML forms, there is no concept of this feature. Our sample shopping cart application will demonstrate this feature.

- XForms browsers wrap data in the form of XML structures. XML data flows from the browser to the server and back. For example, if a user enters data in an XForms input box, the text entered in the input box will be wrapped in an XML tag and sent to the server. In HTML, data is sent to the server in name value pairs. Data is then structured as soon as it is generated on the client end. It is easier to manage structured data on the server side.

- These days, major database vendors support XML interfaces. Therefore, you can map XML data from an XForms browser directly into your database design.

- XForms uses XML Schema Definition to validate user data on the client side. On the other hand, data entered in HTML forms is validated on the server side or by implementing client-side scripts. XForms has almost entirely eliminated the need for server-side processing or client-side scripting for data validation.

We are not going to discuss the basic details of XForms syntax and format in this tutorial. For that information, refer to Resources on page200).

In this section, we will explain how a typical XForms-based application works. The details of this section will help you recognize the server-side processing requirements of a typical application.

We have chosen the shopping cart as our sample application because they are

inherently form-rich and session-based applications. Therefore, while discussing a shopping cart, we will have a good opportunity to demonstrate XForms features in a session-based application.

While you go through the details of the shopping cart, you will get a clear idea about the XML authoring and processing requirements of a typical XForms application. The last topic of this section will summarize and list the server-side processing requirements of an XForms application. Later, when we implement the same concepts using the JSF framework, we will demonstrate how to use JSF technology to fulfill these requirements.

Features of the sample XForms-based shopping cart

Now let's discuss our fictitious shopping cart application. The shopping cart will have the following features:

- The startup page presents a catalog, which consists of categories and products. We call this startup page the *catalog view*.

- The application allows you to browse through the catalog. Each category in the catalog can have products and further subcategories. You can browse through the structure of categories to reach the product of choice.

- If the user clicks a product, a new page opens. This page shows the details of the product selected. Product details include its name, price, description, features, and optional features. We call this page the *product-specification view*. The product-specification view also contains an Add to cart button, which the user can click to place this product in the shopping cart. It also contains a Back to catalog view button to go back to the catalog view.

- Optional features will be displayed as check boxes.

- When the user clicks the Add to cart button, a new page will open. This page will show the cart status (that is, the list of products already in the cart). We call this the *cart view*.

- There are also Edit and Remove buttons associated with each individual product in the cart view.

- If the user clicks Remove, the product will be removed from the cart.

- If the user clicks the Edit button associated with some product in the cart view, another page will appear. This page allows the user to select or deselect optional product features. We call this page the *edit product view*.

- The cart view also contains a Buy button, which will confirm the order with all

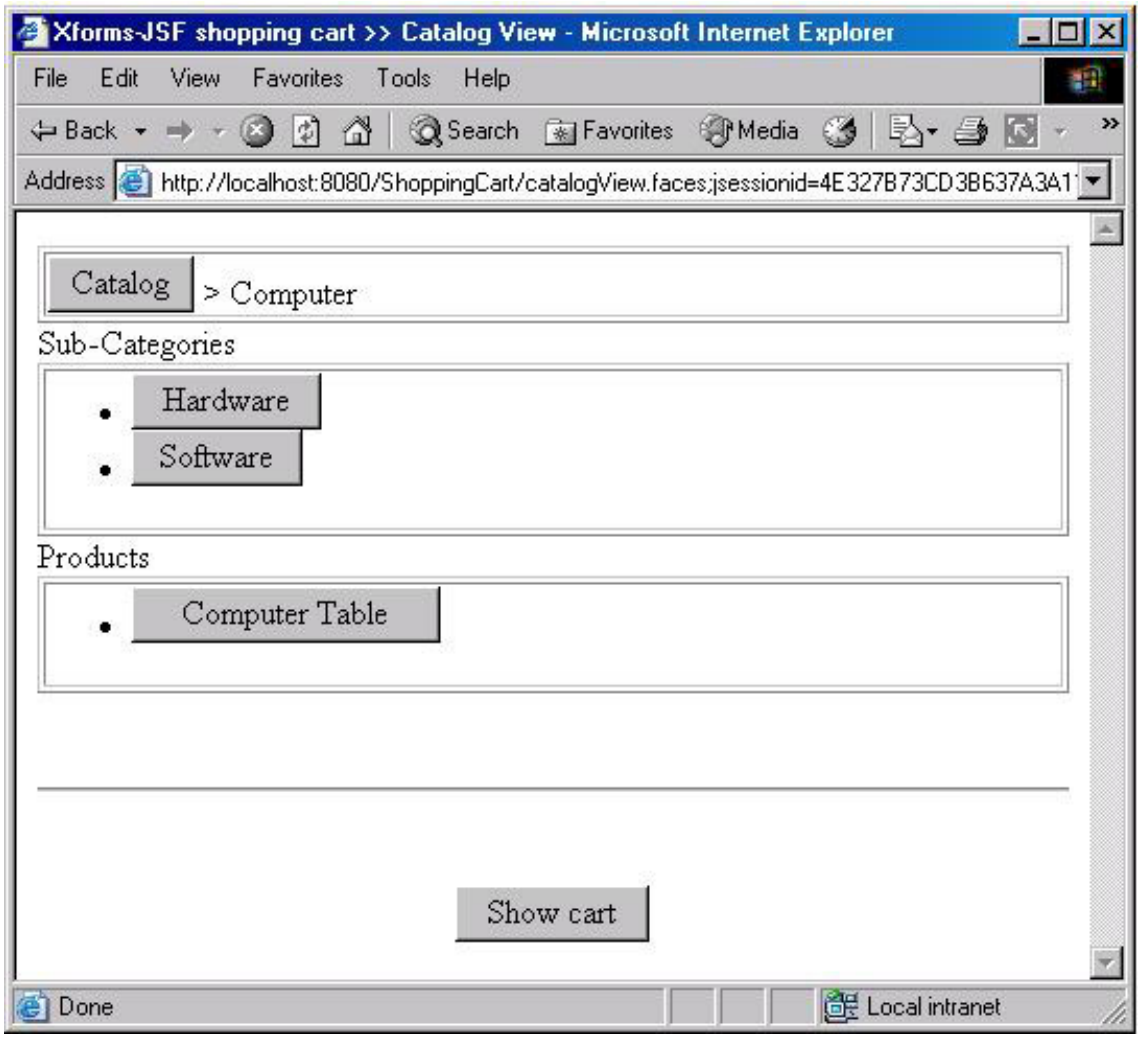
the products in the cart.

Views of the shopping cart

Our shopping cart will include the following views:

- Catalog view
- Product-specification view
- Cart view
- Edit product view

The catalog view shows a list of categories and products:



Note the following points from the catalog view screenshot shown above:

- The screenshot has three boxes: an ancestors box, a subcategories box, and a products box, along with a Show Cart button.

- The ancestors box shows the list of ancestors of the category shown in the catalog view. Each ancestor in the list is shown as a button. The list of ancestors starts with the top-level ancestor in the catalog and ends at the category displayed in the catalog view. You might be wondering why we are using buttons instead of anchors. Actually, XForms V1.0 (W3C recommendation) does not have an element to render anchors. But the XForms V1.0 specification says we can use styles to render buttons as anchors. Most of the XForms browsers available have problems displaying anchors, so the screenshot shows buttons instead of anchors.

  In the screenshot, the ancestors box contains two categories: Catalog and Computer. The Catalog category is the top-level category in our sample catalog, and the Computer category is its immediate child. The catalog view shown above is showing the ancestors, subcategories, and products of the Computer category, so we can say that the Computer category is the currently selected category. While browsing through the catalog, the user can click any category to browse deeper into the catalog. The user can then come back to the parent category by clicking it in the ancestor box.

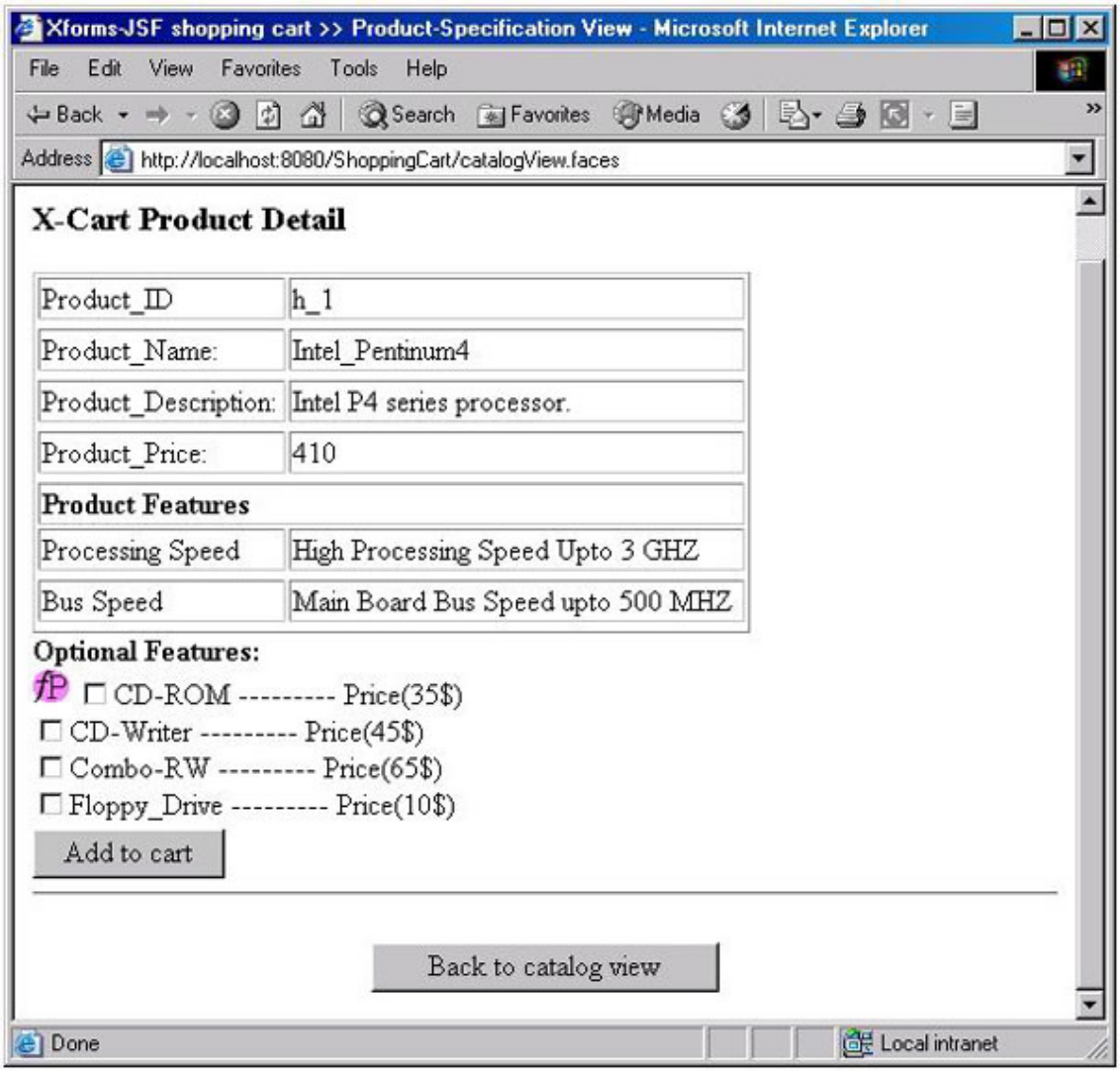
- The subcategories box shows a list of all the subcategories in the selected category. Each subcategory is rendered as a clickable button to view its details. For example, in the screenshot above, the Computer category contains Hardware and Software subcategories.

- The products box shows the entire list of products in the currently open category. In the above screenshot, the Computer category contains only one product: Computer Table.

- The Show cart button is a handy way to display the cart view anytime while browsing through the catalog.

If the user clicks on any of the categories shown on the catalog view, a new catalog view will appear, showing the subcategories and products in the requested category.

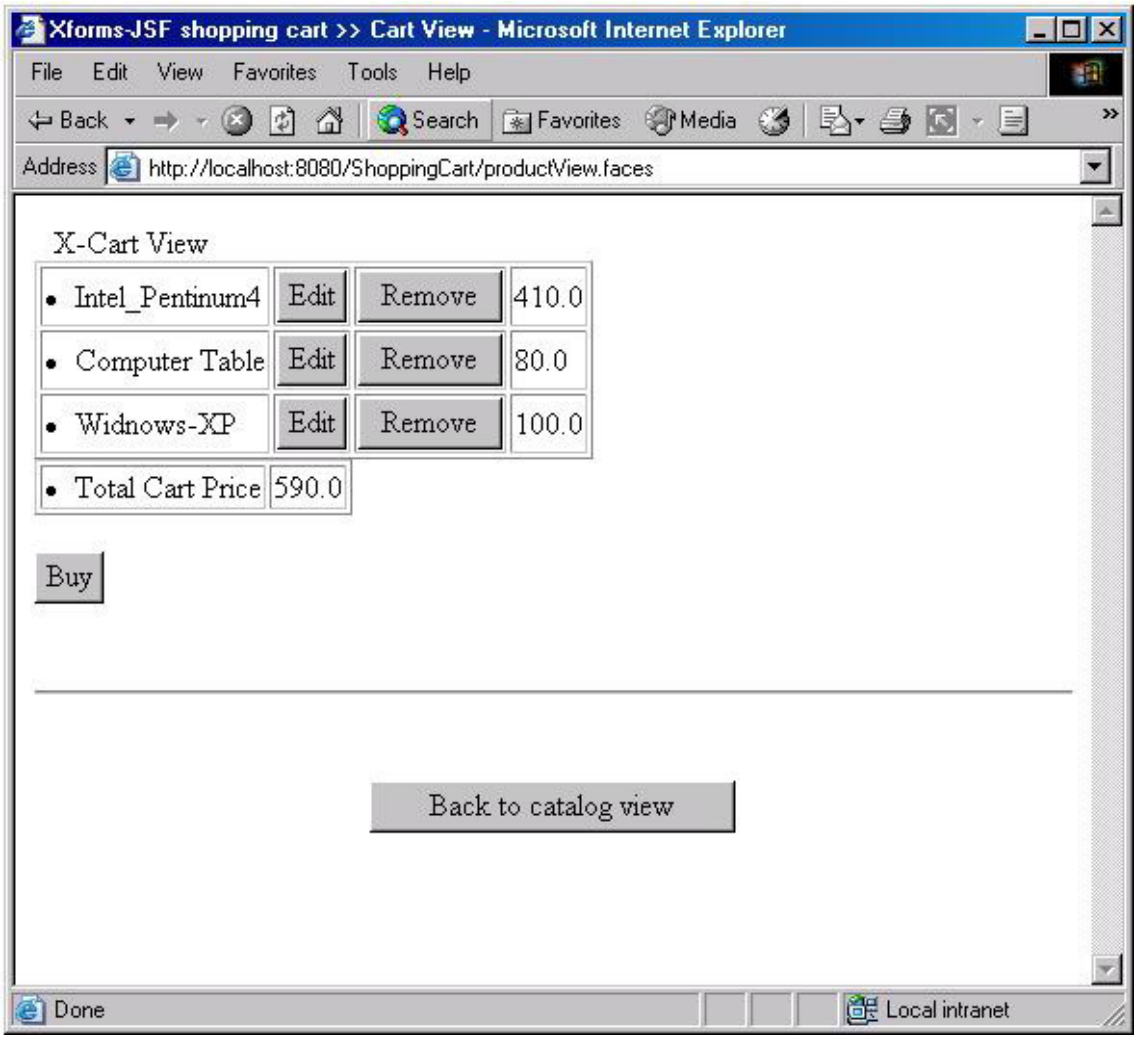
If the user clicks on a product, the details of that product will appear on the product-specification view, as shown in the following screenshot:



The product-specification view renders the specifications, price, description, and default, as well as optional features of a product. The product-specification view allows the user to select from optional features of the product.

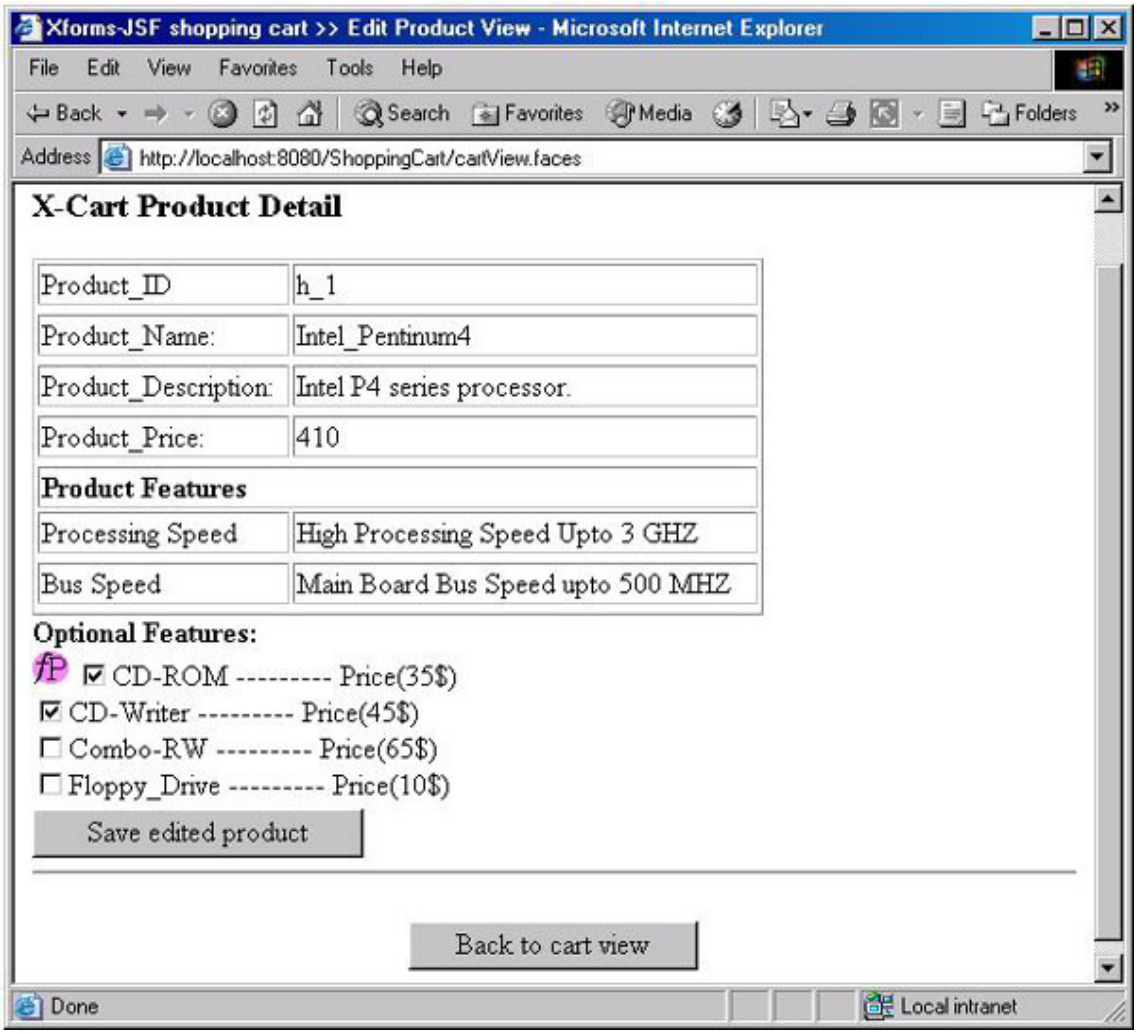
The product-specification view contains an Add to cart button, which will add the product to the cart and display the status of the cart in the cart view. The catalog view button in the screenshot above is a convenient method to go back to the catalog view.

The cart view is shown in the following screenshot:



The cart view shows a list of all products in the cart. This view will show the Edit and Remove buttons with each product in the cart view. If the user clicks Remove, the product associated with the button will be removed from the cart and the cart view will be updated.

Clicking edit will take the user to the edit product view:



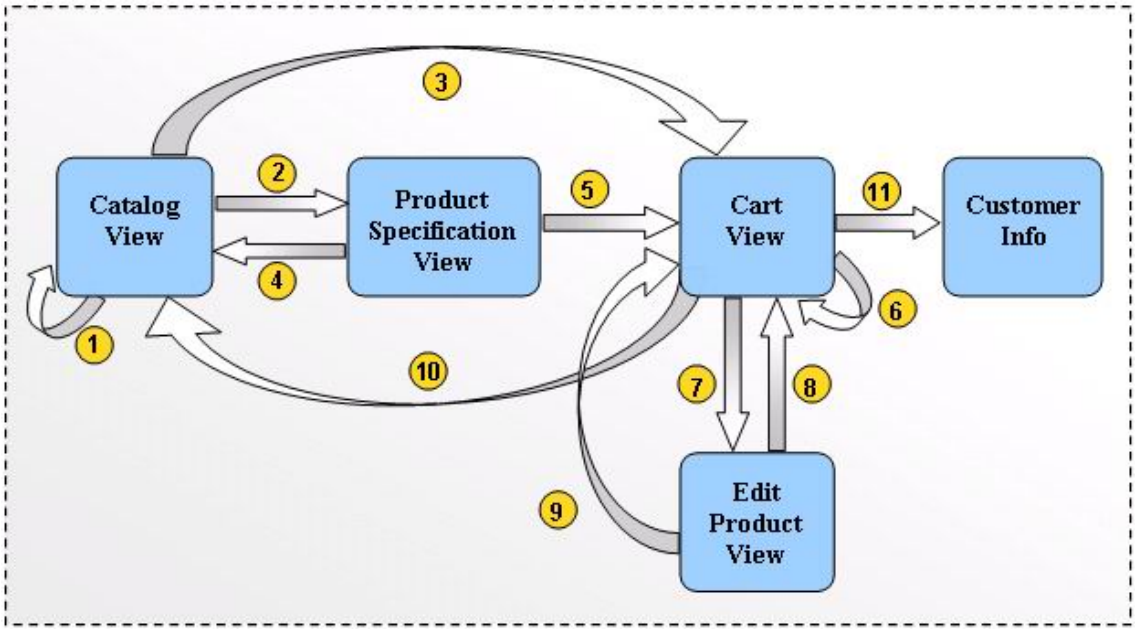
The edit product view is similar to the product-specification view. Differences between the product-specification view and edit product view:

- In the edit product view, the optional features that were selected while adding the product to the cart will appear selected.

- The edit product view has a Save edited product button, instead of Add to Cart.

- If the user clicks Save edited product, the changes in the optional features against the edited product will be saved in the cart and the cart view will be updated.

The following diagram shows the interactions between different views of our shopping cart application:



Map the numbers (associated with arrows in the figure shown above) with the following points:

1. Clicking a category in the catalog view
2. Clicking a product in the catalog view
3. Clicking the Show cart button in the catalog view
4. Clicking the Back to catalog view button in the product-specification view
5. Clicking the Add to cart button in the product-specification view
6. Clicking the Remove button of a particular product in the cart view
7. Clicking the Edit button of a particular product in the cart view
8. Clicking the Back to cart view button in the edit product view; in this case, his changes in the edit product view will be lost
9. Clicking the Save edited product button in the edit product view
10. Clicking the Back to catalog view button in the cart view
11. Clicking the Buy button in the cart view

XML structures for the shopping cart

Because XForms works on the idea of XML, our XForms-based shopping cart will need XML structures. We are going to design the product XML structure, which wraps the details of an individual product; and the category XML structure, which wraps the details of a category in the catalog.

To see the details of these XML structures, look at the following XML, which shows the specification of a product:

```
<product id="h_1" name="Intel_Pentinum4" catId="1.1">
  <description>Intel P4 series processor.</description>
  <price>410</price>
  <features>
   <feature>
    <name>20 GB HD</name>
    <description> A Hard Drive of 20 Giga
Byte</description>
    </feature>
    . . . . .
  </features>
  <optional-features>
    <feature>
      <name>CD-ROM</name>
      <price>35</price>
      <description>optional feature</description>
    </feature>
    . . . . .
  </optional-features>
</product>
```

Notice the following points from the XML structure shown above:

- The root element in the XML structure listed above is `product`, which wraps the details of a product like its ID, name, category (to which this product belongs), description, price, features, and optional features.

- The ID is an attribute of the `product` element. Similarly, the product name and ID of the category to which the product belongs (`catId`) are also attributes of the product element.

- A product has two types of features: fixed (or standard) features and optional features (which cost extra).

- The `features` element wraps all the fixed features for a product. Each feature is wrapped inside a `feature` element, which in turn wraps the name and description of the feature. For example, in the XML shown above, the string "20 GB HD" represents the name of a fixed feature for the product "Intel Pentium4." The description of the feature is "A Hard Drive of 20 Giga Byte."

- The `optional-features` element wraps all the optional features for a product. Individual optional features are represented by `feature` child elements of the `optional-features` element. Each `feature` element wraps the name, price, and description of the optional feature.

A `category` element represents an individual category shown in the catalog view. A `category` element wraps products and subcategories. All the top-level `category` elements reside inside the root `categories` element. For example, have a look at the following XML:

```
<categories>
 <category id="1" name="Computer">
   <category id="1.1" name="Hardware" catId="1">
     <description>Hardware products for PC.</description>
     <product id="h_1" name="Intel_Pentinum4" catId="1.1">
       . . . . . .
     </product>
     <!-- other products and subcategories -->
   </category>
   <!-- other subcategories -->
 </category>
 <!-- other categories -->
</categories>
```

Notice the following points from the XML structure shown above:

- A `categories` tag wraps all the categories in the catalog.
- The information of a particular category is wrapped in a `category` element.
- The name, ID, and ID of the parent category (to which this category belongs) are attributes of the category element.
- The subcategories in a category are wrapped in the `category` children of the `category` element.
- The products in a category are wrapped in `product` children of the `category` element. We have already discussed the structure of the `product` element.

Next, we'll explain the product specification page of the shopping cart. We have chosen to explain that page before explaining the other pages of the shopping cart because this will help you understand some important XForms tags and the subsequent discussion on how the shopping cart works.

Using XForms

Now we will explain the use of XForms elements in an HTML file by taking the markup that generates the product specification view. The XML markup that generates the product-specification view is shown here:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:xforms="http://www.w3.org/2002/xforms">
 <head>
  <title>Xforms-JSF shopping cart</title>
  <xforms:model xmlns:xforms="http://www.w3.org/2002/xforms"
    id="myModel">
    <xforms:submission action="/XCart/faces/productView.jsp"
      method="post" id="submit"/>
    <xforms:instance>
      <data>
        <action-performed></action-performed>
        <selectedFeatures></selectedFeatures>
      </data>
    </xforms:instance>
  </xforms:model>
 </head>
 <body>
  <h3> X-Cart Product Detail </h3>
  <table width="100%">
  <tr>
   <table colspan="3" border="1" width="70%">
     <!-- product details in tabular form-->
   </table>
  </tr>
  </table>
  <xforms:select ref="options" appearance="full"
   xmlns:xforms="http://www.w3.org/2002/xforms">
   <xforms:label>Optional Features: </xforms:label>
   <xforms:item>
    <xforms:label>CD-ROM ------- Price(35$) </xforms:label>
    <xforms:value>CD-ROM</xforms:value>
   </xforms:item>
   <!-- other xforms:item instances-->
  </xforms:select>
  <xforms:submit submission="submit"
    xmlns:xforms="http://www.w3.org/2002/xforms"
    xmlns:ev="http://www.w3.org/2001/xml-events">
    <xforms:action ev:event="DOMActivate">
     <xforms:setvalue
ref="action-performed">_id1</xforms:setvalue>
    </xforms:action>
    <xforms:label>Add to cart</xforms:label>
  </xforms:submit>
   <xforms:submit submission="submit"
     xmlns:xforms="http://www.w3.org/2002/xforms"
     xmlns:ev="http://www.w3.org/2001/xml-events">
     <xforms:action ev:event="DOMActivate">
      <xforms:setvalue
ref="action-performed">id2</xforms:setvalue>
     </xforms:action>
     <xforms:label>Back to catalog view</xforms:label>
   </xforms:submit>
</body>
</html>
```

This XML markup is actually an HTML page, which contains some XForms elements. You can easily identify the following components of this product-specification markup:

- The header that contains an XForms element named `xforms:model`
- The `xforms:select` element in the `body` of the HTML page
- The two `xforms:submit` elements, which are included in the `body` of the HTML page

Now let's see each component of the product-specification page one by one. The next section explains the XForms `model` element.

The model element

Look at the `xforms:model` element:

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms"
  id="myModel">
  <xforms:submission action="/XCart/faces/productView.jsp"
  method="post" id="submit"/>
  <xforms:instance>
    <data>
      <action-performed></action-performed>
      <selectedFeatures></selectedFeatures>
    </data>
  </xforms:instance>
</xforms:model>
```

The `model` element of XForms behaves like the `form` tag of HTML. The `model` element:

- Wraps the structure of application-specific XML data. In our case, the application-specific data structure is the `data` element. The application-specific XML data is the XML structure that will be used for data interchange between the client and server. And this application-specific XML structure always comes in the `instance` element of the XForms `model` element.

- Provides the submission logic of data, which involves various bits of information, such as the method of submission (for instance, `post`, `get`, etc.), the URL used for the submission (that is, the address of the server), what data to submit (that is, the `ref` attribute), etc.

First, let's see how an `xforms:model` element wraps the structure of application-specific XML data.

Look at the `xforms:instance element`:

```
<xforms:instance>
  <data>
    <action-performed></action-performed>
    <selectedFeatures></selectedFeatures>
  </data>
</xforms:instance>
```

Notice the following points:

- There should be just one child of the `instance` element. For example, the `instance` element shown above has just one child named `data`, which wraps the complete application-specific data.

- The application-specific XML structure in the `instance` element wraps the user's data. For example, if the `selectedFeatures` child of the data element is associated with some user interaction element (such as a selection list), the `selectedFeatures` element will wrap the user's selection during data transfer from the client browser to the server. Later, you'll see how to associate user interaction components with different elements in application-specific XML.

Here's how the `xforms:model` specifies how to submit (or send) user data to the server. Have a look at the `xforms:submission` element:

```
<xforms:model>
  .. .. ..
  <xforms:submission action="/XCart/faces/productView.jsp"
    method="post" id="submit"/>
  .. .. ..
</xforms:model>
```

Notice the following points:

- An XForms `submission` element is used.

- This element has three attributes: `action`, `method`, and `id`.

- The `action` attribute specifies the URL for submitting data (the HTTP address of a Web server).

- The `method` attribute contains information about the way data will be sent back to the URL mentioned in the `action` attribute. For example, if the `method` is specified as "POST," data will be sent back to the server using the HTTP POST method.

- The string specified in the `id` attribute value works as an identifier for the `submission` element. There can be more than one `submission` child of a `model` element. Each `submission` element will be identified using its `id` attribute value.

In The submit element on page 20, you'll learn how to associate a user interface

button with a `submission` element.

The select element

We have said that the `instance` element defines the structure of application-specific XML data. We have seen the structure of XML data, but where does the data come from?

Naturally, it's the user's data, so it has to come from the user. Some user interface XForms elements (such as a selection list) will fetch the user's data and wrap it inside some application-specific child of the `xforms:instance` element before sending the user's data to the server.

To understand how this works, look at the `select` element inside the `body` of the product specification markup:

```
<xforms:select ref="options" appearance="full"
 xmlns:xforms="http://www.w3.org/2002/xforms" model="myModel">
 <xforms:label>Optional Features:
<br/></xforms:label>
 <xforms:item>
  <xforms:label>CD-ROM --------- Price(35$)
<br/></xforms:label>
  <xforms:value>CD-ROM</xforms:value>
 </xforms:item>
 <!-- other xforms:item instances-->
</xforms:select>
```

The output of the above XForms markup is shown in the following screenshot:

Let's see the behavior of the `select` element shown above, then we'll see how it will map the user's data to specific elements or attributes of the application-specific XML data.

The following points depict the behavior of the `select` element:

- The behavior of the `select` element is similar to the `<input type="checkbox">` tag of HTML.

- The `select` element provides a list of options (a selection list) to the user, who can select one or more options from it.

- The `label` child of the `select` element is used to display a common label for all the options in the selection list. We have used "Optional Features," which appears as the label for the selection list in the screenshot.

- The `item` child of the `select` element represents one option from the list of options in the selection list.

- The `label` child of the `select` element wraps the label for the option.

- The `value` child of the `select` element wraps a string value, which will be copied into the application-specific XML if the user selects this option.

Now we'll map a user interface element (such as the `select` element discussed above) to application-specific XML data. Take a look at the following XML:

```
<html
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:xforms="http://www.w3.org/2002/xforms">
 <head>
   <xforms:model xmlns:xforms="http://www.w3.org/2002/xforms"
     id="myModel">
     <xforms:submission action="/XCart/faces/productView.jsp"
       method="post" id="submit"/>
     <xforms:instance>
      <data>
       <action-performed></action-performed>
       <selectedFeatures></selectedFeatures>
      </data>
     </xforms:instance>
   </xforms:model>
 </head>
 <body>
  . . . . . .
  <xforms:select ref="options" appearance="full"
    xmlns:xforms="http://www.w3.org/2002/xforms" model="myModel">
    <xforms:label>Optional Features:
<br/></xforms:label>
    <xforms:item>
      <xforms:label>CD-ROM --------- Price(35$)
<br/></xforms:label>
       <xforms:value>CD-ROM</xforms:value>
    </xforms:item>
    <xforms:item>
      <xforms:label>CD-Writer ---------
Price(45$)<br/></xforms:label>
      <xforms:value>CD-Writer</xforms:value>
    </xforms:item>
    <xforms:item>
      <xforms:label>Combo-RW ---------
Price(65$)<br/></xforms:label>
      <xforms:value>Combo-RW</xforms:value>
    </xforms:item>
```

```
    <xforms:item>
      <xforms:label>Floppy_Drive -------
Price(10$)<br/></xforms:label>
      <xforms:value>Floppy_Drive</xforms:value>
    </xforms:item>
  </xforms:select>
  . . . . . .
</body>
</html>
```

Because a single HTML document can contain a number of `xforms:model` elements, we have to identify which `model` a particular `select` element refers to. The `model` attribute of the `select` element is used to establish an association between the `select` and `model` elements. The `model` attribute value of the `select` element matches the `id` attribute value of the `model` element. The `model` attribute of the `select` element in our product-specification page indicates that the `select` element is associated with the `model` whose `id` attribute has the value "myModel."

In addition to specifying which `model` element wraps the application-specific XML data associated with a particular `select` element, the `select` element also needs to specify which element or attribute of the application-specific XML wraps the user's choices. The `ref` attribute value of the `select` element establishes this association. Notice from the `select` element shown above that its `ref` attribute value is "selectedFeatures." This is actually an XPath query, which specifies that the `selectedFeatures` element within the application-specific XML data wraps the user's choice.

Let's use an example to elaborate on these ideas. If the user selects the first and third option from the list, the application-specific XML will look like the following code:

```
<data>
  <action-performed></action-performed>
  <selectedFeatures>Combo-RW CD-ROM</selectedFeatures>
</data>
```

The above XML wraps the value of the first and third elements (specified in the `value` elements of their respective items). The content of the `value` elements is concatenated together with a space between them in the application-specific XML.

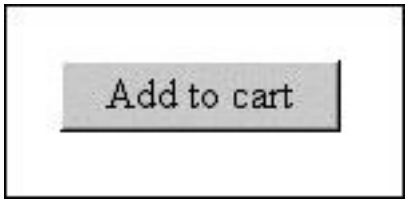
The submit element

Let's discuss the `xforms:submit` element that immediately follows the `xforms:select` element in the product-specification page:

```
<xforms:submit submission="submit">
  <xforms:action ev:event="DOMActivate">
    <xforms:setvalue
```

```
ref="action-performed">_id1</xforms:setvalue>
  </xforms:action>
  <xforms:label>Add to cart</xforms:label>
</xforms:submit>
```

This markup will generate the following view, which resembles a button:



We explained the purpose of the `xforms:submission` element while discussing the XForms `model` element, but when will this submission be initiated? To initiate the submission process, XForms provides the `submit` element, whose graphical view resembles a button.

The `submit` element:

- Behaves like the `<input type="submit">` tag in HTML.
- Renders a button with a label on it that is specified in the label child of the `submit` element.
- Is used to submit data (wrapped in an XML structure) to the URL mentioned in the `action` attribute of the `submission` element.
- Can include different types of submissions in the same `model` element. For each type of submission, there will be one `submission` element in the `model`. Therefore, we need some mechanism in the `submit` element to specify which submission this submit button will invoke. Notice the `submission` attribute in the `submit` element in the markup shown above. The value of the `submission` attribute identifies the XForms `submission` element (a child of the `model` element), which should be invoked by clicking a button.

When the user clicks the button, an `xforms-submit` event is fired that initiates the XForms `submission` element defined in the XForms `model` element. The XForms `submission` element will send the user's data wrapped in application-specific XML to the URL mentioned in the `action` attribute of the XForms `submission` element using the method specified in the `method` attribute.

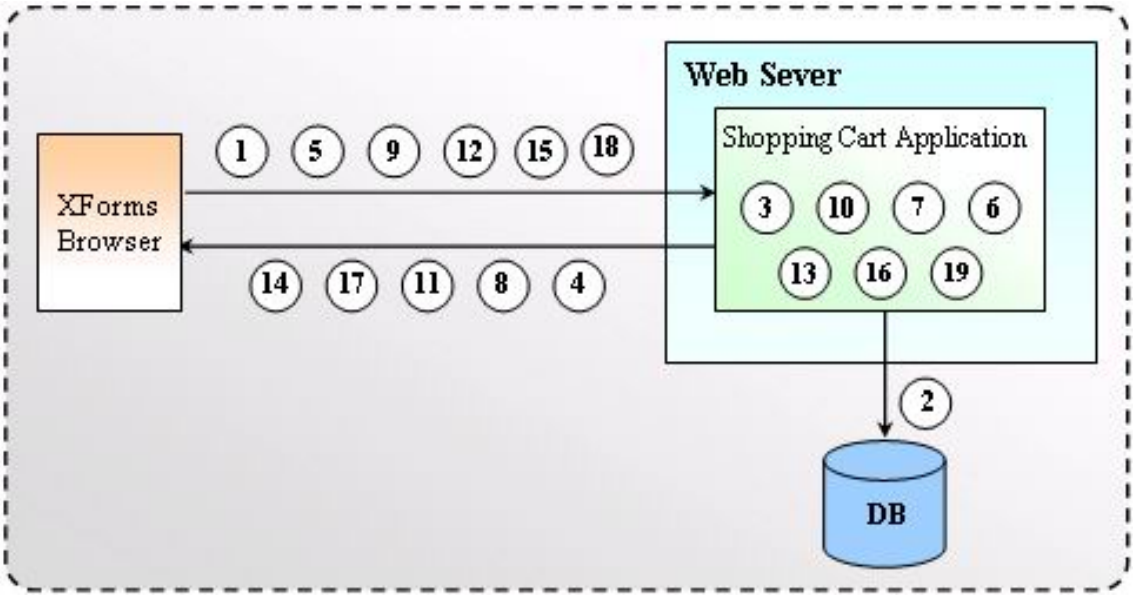
We have provided detailed information about the `xforms:model`, `xforms:select`, and `xforms:submit` elements. This information applies to the different UI elements of XForms. Later, we will introduce many more UI elements (like XForms `input`, `select`, and `select1`). These UI elements use the same concept of associating the user's data with application-specific XML.

User interaction and flow of data in our shopping cart application

Now we will discuss the sequence of events that occurs when a user accesses the shopping cart. In particular, we will focus on the XML data interchange that will take place between the shopping cart application and client.

For now, we won't worry about how the application performs all authoring tasks. We are only concerned with what it authors. Later, we'll demonstrate how the application uses JSF technology to fulfill the authoring tasks.

Look at the following figure, which shows the sequence of events that occurs when a user visits our shopping cart (different events are marked as numbers in the figure):



The events of the figure are explained below:

1. Suppose that a user requests the following URL:

   ```
   www.aFictiousShoppingCart.com
   ```

2. On receipt of this request, the shopping cart application loads the XML structure from the database that contains the catalog data.

3. The shopping cart application gets the information for products and categories from the XML structure, and generates the following markup that renders the catalog view to the user:

```
<html
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
```

```
  xmlns:xforms="http://www.w3.org/2002/xforms">
<head>
   <xforms:model>
    <xforms:submission action="/myXCart/faces/catalogView.jsp"
       method="post" id="submit" />
    <xforms:instance>
        <data>
          <action-performed></action-performed>
          <selectedFeatures>Combo-RW CD-ROM</selectedFeatures>
        </data>
    </xforms:instance>
   </xforms:model>
</head>
<body>
  <table width="100%" >
   <tr>
    <table border="1" width="100%">
     <tr>
      <td>Catalog</td>
     </tr>
    </table>
    <table border="1" width="100%">Sub-Categories
     <tr>
      <td><ul><li>
       <xforms:submit submission="submit" >
         <xforms:label>Computer</xforms:label>
         <xforms:action ev:event="DOMActivate">
           <xforms:setvalue ref="action-performed">
             _id1@1
           </xforms:setvalue>
         </xforms:action>
       </xforms:submit>
      </li></ul></td>
     </tr>
     <!-- remaining catalog data markup in same pattern -->
    </table>
   </tr>
  </table>
</body>
</html>
```

The above markup will render a list of categories and products (catalog view) in the form of buttons as shown in Views of the shopping cart on page 7.

4. The shopping cart application sends back the above markup to the XForms browser.

5. Now suppose the user clicks a product with ID five. The browser will send the request to the Web server with following XML structure:

```
<data>
  <action-performed>_id1@5</action-performed>
  <selectedFeatures>Combo-RW CD-ROM</selectedFeatures>
</data>
```

The data wrapped in the `action-performed` element has two parts separated by an "@" symbol. The prefix of "@" (`_id1`) indicates the ID of the server-side Java component (that authored the markup for the catalog

view). The postfix of "@" (5) represents the ID of the selected product.

6. On receipt of the request, the shopping cart application parses the XML structure in the request and extracts the product's ID from it.

7. The shopping cart application then extracts the product's XML structure against the ID retrieved in step 6 from the XML structure extracted from the database in step 2. The product XML structure contains the details of the product.

8. In response, the shopping cart application generates the following markup and sends it to the XForms browser:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:xforms="http://www.w3.org/2002/xforms">
<head>
  <title>JSF_n_Xforms shopping cart</title>
  <xforms:model xmlns:xforms="http://www.w3.org/2002/xforms"
    id="myModel">
    <xforms:submission action="/XCart/faces/productView.jsp"
      method="post" id="submit"/>
    <xforms:instance>
      <data>
        <action-performed></action-performed>
        <selectedFeatures></selectedFeatures>
      </data>
    </xforms:instance>
  </xforms:model>
</head>
<body>
<h3> X-Cart Product Detail </h3>
 <table width="100%">
 <tr>
  <table colspan="3" border="1" width="70%">
  <tr>
    <td>Product_ID</td>
    <td>h_1</td>
   </tr>
   <!-- other product details -->
  </table>
 </tr>
 </table>
 <xforms:select ref="options" appearance="full"
  xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:label>Optional Features: </xforms:label>
  <xforms:item>
   <xforms:label>CD-ROM --------- Price(35$)
</xforms:label>
   <xforms:value>CD-ROM</xforms:value>
  </xforms:item>
  <xforms:item>
   <xforms:label>
     CD-Writer --------- Price(45$)
   </xforms:label>
   <xforms:value>CD-Writer</xforms:value>
  </xforms:item>
     <!-- other xforms:item instances-->
 </xforms:select>
```

```
 <xforms:submit submission="submit"
   xmlns:xforms="http://www.w3.org/2002/xforms"
   xmlns:ev="http://www.w3.org/2001/xml-events">
   <xforms:action ev:event="DOMActivate">
    <xforms:setvalue ref="action-performed">
      _id1
    </xforms:setvalue>
   </xforms:action>
   <xforms:label>Add to Cart</xforms:label>
 </xforms:submit>
 <xforms:submit submission="submit"
   xmlns:xforms="http://www.w3.org/2002/xforms"
   xmlns:ev="http://www.w3.org/2001/xml-events">
   <xforms:action ev:event="DOMActivate">
    <xforms:setvalue ref="action-performed">
      _id2
    </xforms:setvalue>
   </xforms:action>
   <xforms:label>Back To Catalog View</xforms:label>
 </xforms:submit>
</body>
</html>
```

The graphical view of the above markup is already shown in Views of the shopping cart on page 7.

9. Now suppose that the user selects first and second optional features of the product shown on the product-specification view, and clicks the Add to cart button. On clicking Add to cart, the browser wraps the values corresponding to the selected features ("CD-ROM" and "CD-Writer") and the ID of the button clicked ("_id2") in the request, then forwards the request to the shopping cart application. The XML structure that carries the user's data to the shopping cart application:

```
<data>
  <action-performed>_id2</action-performed>
  <options>CD-ROM CD-Writer</options>
</data>
```

Notice from this XML that the `action-performed` tag wraps the "ID" of the button, and the `options` tag contains all the features the user selected.

10. The shopping cart application receives the request, parses the incoming XML structure, extracts data about the selected features, and saves the data in server-side objects.

11. The shopping cart application will respond with the following markup, which renders the cart view on the user's browser:

```
<html xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:xforms="http://www.w3.org/2002/xforms">
<head>
  <title>JSF_n_Xforms shopping cart >> Cart View
</title>
  <xforms:model>
    <xforms:submission action="/myXCart/faces/cartView.jsp"
      method="post" id="submit"/>
```

```
      <xforms:instance>
        <modelXML>
          <action-performed></action-performed>
        </modelXML>
      </xforms:instance>
    </xforms:model>
  </head>
  <body>
    <table border="1">
     <ol type="1"><tr>
       <td><li>Intel_Pentinum4</li></td>
       <td><xforms:submit submission="submit" >
         <xforms:label>Edit</xforms:label>
         <xforms:action ev:event="DOMActivate">
          <xforms:setvalue ref="action-performed">
            _id1@0@edit
          </xforms:setvalue>
         </xforms:action>
       </xforms:submit></td>
       <td><xforms:submit submission="submit" >
         <xforms:label>Remove</xforms:label>
         <xforms:action ev:event="DOMActivate">
          <xforms:setvalue ref="action-performed">
            _id1@0@remove
          </xforms:setvalue>
         </xforms:action>
       </xforms:submit></td>
     </tr></ol>
    </table>
    <xforms:submit submission="submit">
      <xforms:label>Remove</xforms:label>
      <xforms:action ev:event="DOMActivate">
        <xforms:setvalue ref="action-performed">
           _id2
        </xforms:setvalue>
      </xforms:action>
    </xforms:submit>
  </body>
  </html>
```

The graphical view of the above markup is shown in Views of the shopping cart on page 7.

12. Recall from Views of the shopping cart on page 7 that each product in the cart will have a set of edit and Remove buttons. If the user clicks the Edit button, the XForms browser forwards the following XML:

```
<data>
  <action-performed>_id1@0@edit</action-performed>
</data>
```

The data wrapped in the `action-performed` tag consists of three parts separated by an "@" symbol. The prefix of the first "@" (`_id1`) symbol (from the left) indicates the ID of the server-side Java component. The postfix of "@" (`0`) represents the ID of the selected product. The last string (`edit` identifies the button clicked.

13. The shopping cart application parses the XML structure, extracts the "ID" of

the product for editing, and generates the markup for the edit product view. The markup generated for the edit product view is similar to the markup shown in step 8.

14. The shopping cart application sends the markup for the edit product page to the user's browser.

15. If the user clicks Remove, the XForms browser forwards the following XML to the shopping cart application:

```
<data>
  <action-performed>_id1@0@remove</action-performed>
</data>
```

16. On receipt of the request, the shopping cart application parses the XML structure, extracts the "ID" of the product, removes the product from the cart, and updates the cart view.

17. The shopping cart application responds with the updated cart view.

18. When the Buy button in the cart view is clicked, the browser forwards the following XML to the shopping cart application:

```
<data>
  <action-performed>_id2</action-performed>
</data>
```

19. On receipt of the XML above, the shopping cart application parses the XML structure, extracts the ID of the button from XML, then removes the product from the cart and updates the cart view.

In this section, we mostly we discussed the XML structures, but we did not mention who generates these markups and how. XForms-JSF integration strategy on page 80 andXForms-JSF tag library on page 118 will address these issues. They will show how we can use the JSF architecture to generate the different shopping cart views discussed here. We will explain the server-side responsibilities for generating these markups while demonstrating the whole XForms-JSF shopping cart application.

Server-side processing requirements of an XForms application

We have already discussed the details of XML formats and data interchange in our shopping cart. Keeping in mind the XML flow of data we described, you'll see how our shopping cart application will have the following responsibilities on the server side:

- **Render XForms markup**

Most importantly, our Shopping cart application will have to render XForms markup, so that the XForms browser can properly display XForms components.

- **Author the XML formats**

  Our shopping cart application will have to author XML formats that will be used to track the user interactions on the client side. It is the application-specific XML wrapped inside the XForms `model` element.

- **Parse the XML request**

  In an XForms application, user data is sent to the server wrapped in an XML structure. Therefore, our shopping cart application will have to parse the XML request to extract user data from the request.

- **User interaction with the XForms components**

  In addition to all the above, our shopping cart application will have to implement a mechanism to track the user's interactions with XForms components on the client side. For example, if a user clicks an XForms button, the server-side application needs to know which button was clicked.

- **Talk to a database**

  The XForms shopping cart application will have to interact with some data source to get application-specific data. For the sake of simplicity, our sample shopping cart implementation in Implementing the XForms-JSF shopping cart on page 161 will get the application data from an XML file.

In the next three sections, we will explain the JSF architecture to demonstrate how to use JSF technology to fulfill the server-side authoring and processing requirements stated above.

Section 3. The JSF architecture

Components of a JSF application

In the previous section, we saw how an XForms application works. Our ultimate goal in this tutorial is to explain and demonstrate the use of JSF technology to fulfill the server-side authoring and processing requirements of an XForms application. But before we can do that, we have to dig deep inside JSF technology and see what's there.

First, we'll provide a high-level view of JSF technology to acquaint you with all the components of a JSF application. You will learn how it works, what a JSF tag library is, and how an application uses the JSF tag library.

Because this section is designed as a high-level view, we will not go into the details of individual concepts. Subsequent sections will explore further details of the concepts discussed here.

Let's start our discussion by dividing a JSF application into different types of modules. If you are developing a JSF application, you will find that it is roughly divided into the following four types of modules:

- Java classes that belong to the JSF framework
- The JSF tag library
- JSPs that author the UI of your application by using JSF tags
- JavaBeans that comprise the business logic and application data of the JSF application

The first item is the basic framework that enables the other three types of modules to work properly. There are various Java classes that form the JSF framework. You'll explore these classes further in How does a JSF component tree work? on page 42, which explains the purpose and methods in each class.

The second item (the JSF tag library) allows the JSF applications to run as JSPs. The concept of tag libraries is part of the extensibility framework of JSP technology and is not new with JSF technology. The JSF tag libraries are just like normal JSP tag libraries; the only difference is that tags in a JSF library work according to the JSF framework.

The last two items in the list (JSPs and JavaBeans) are application-specific modules, which means you will develop them once for each JSF application.

In this tutorial, we'll demonstrate the use of the JSF framework. While doing so, we will develop two tag libraries and several JSPs, as well as many JavaBeans.

We will use the terms "tag library-specific" and "application-specific" often. Tag library-specific means that something applies to the entire tag library, no matter which application this tag library is used in. Application- specific means that something is specific to the application and may not be applicable if the same

tag library is used in another JSF application.

JSF tags and JSF components

The JSF tag library is a collection of reusable components required to develop a JSF application. When you develop a JSF application, you will use these components in your application without changing them.

To understand a JSF tag library, we need to know how the *JSF tags* and *JSF components* work. A JSF tag is what you will write inside a JSP page. A JSF component is the back-end class that wraps the functionality of a JSF tag.

Each JSF tag has a *tag handler* class. This tag handler class basically associates the JSF tag with a JSF component. We will discuss the tag handler class in detail in The UIComponentTag class on page43 .

JSF tags are declared in a *tag library descriptor* (TLD) file. This TLD file specifies the tag handler class associated with a particular JSF tag. It also declares the attributes that a JSF tag can have. We will discuss TLD files in detail in Associating JSF tags with tag classes on page45 .

For example, look at the following JSF tag:

```
<jsf:selectManyCheckbox/>
```

In a JSP page, this tag provides the user with an option to select from a list of given choices, just like check boxes in HTML. The behavior of the `selectManyCheckbox` tag is shown in the following points.

- The JSF tag above renders the following HTML markup:

```
<table border="0">
  <tr><td><input type="checkbox" name="_id0"
value="Red">Red</td></tr>
</table>
```

  The JSF component will generate the response markup for the JSF tag by a process called *rendering* or *encoding*. The rendering process is basically done in the encode methods of the JSF component class (the back-end class). In The UIComponentBase class on page 55, we will discuss the JSF component class in detail.

- The HTML markup above provides the user with an option to select choices from a list of given choices. Suppose the user selects some options and submits it to the Web server in the form of a JSF request. When the request reaches the server side, the JSF component class associated with the JSF tag parses the request and detects the options selected by the user. The JSF component class contains a method named `decode()`, which parses all incoming JSF requests. The parsing of JSF requests is technically known

as the *decoding* process. We will describe the decoding process in The decoding process on page56 .

- The options retrieved from the JSF request (during the decoding process) need to be saved somewhere. Normally, a JSF application uses server-side Java objects for this purpose. These server-side Java objects are called *model beans*, which are basically Java classes used to manage application-specific data. The JSF component class contains a method named `updateModel()`, which stores the options in the model bean. We will discuss the model beans in detail in Model bean wrappers for application's business logic and data on page 33. We'll discuss the `updateModel()` method in Updating model beans on page63 .

You will include the `jsf:selectManyCheckbox` tag in a JSP page and configure your application in such a way that a corresponding JSF component class will become associated with the tag. The JSF component class will provide the behavior stated in the above points.

Whenever you develop a tag library, make sure that your component classes provide the required behavior. Later, when an application developer uses your tag library in a JSF application, your JSF components will provide the required functionality to the application.

As we will see later, an application developer can associate JSF tags with different JSF components while developing a JSF application. This process enables the application developer to decide more easily what functionality should be included in an application.

The JSF framework simply enables the use of JSF tags and components in a JSF application so that reusable JSF components can be simply plugged into any JSF application, thereby providing their functionality on its own.

We would like to mention that the JSF technology reference implementation from Sun Microsystems comes with a tag library you can use in your applications. You can also develop your own JSF tag libraries. XForms-JSF integration strategy on page 80 andXForms-JSF tag library on page 118 demonstrate the development of a custom JSF tag library.

Using JSF tags in JSPs

Now we will look at how to use JSF tags in a JSP page.

You will use JSPs in a JSF application the way you use JSPs in any server-side Java application. You will put your JSF tags in a JSP page and pass model beans to JSF components.

For example, look at the following JSP code:

```
<html>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <f:view>
      <h:form formName="myForm">
        <h:selectManyCheckbox
          value="{productData.selectedOptionalFeature}">
          <f:selectItems
            value="{productData.optionalFeatures}"/>
        </h:selectManyCheckbox>
        <h:commandButton label="Submit" >
          <f:actionListener type="jsf.DemoActionListener"/>
        </h:commandButton>
      </h:form>
    </f:view>
  </body>
</html>
```

Notice the following points:

- We have used the `taglib` directives of JSP technology to include the JSF tag library. The `uri` attribute of the `taglib` directive specifies the location of the tag library. In the code above, we have included two tag libraries. One is the core JSF tag library (`http://java.sun.com/jsf/core`), and the other is the JSF HTML tag library (`http://java.sun.com/jsf/html`). These two libraries come with the reference implementation.

- The `prefix` attribute of the `taglib` directive specifies a short prefix to access a particular tag from the tag library in the JSP page. The prefix is defined once and used in the JSP page at many places.

- All the JSF tags are wrapped inside the `f:view` tag. The component class associated with the `f:view` tag is the `UIViewRoot` class. Because the `f:view` tag wraps all the JSF tags in a JSP page, the `UIViewRoot` class will be the parent of all the JSF components in a JSP page. We can also say that a `UIViewRoot` object will always be the root of all JSF component classes in a JSP page.

- The JSPs consist of JSF and non-JSF tags. Each of the JSF tags in a JSP page wraps its own behavior. In the above markup, we used the JSF tags like `f:view`, `h:form`, `h:selectManyCheckbox`, `f:selectItems`, and `h:commandButton`. We also used HTML tags like `head`, `title`, and `body`.

The JSF framework provides a separation between the behavior and presentation of a component. If we develop a non-JSF application using JSPs, we write a tag's behavior and presentation on the same page. The JSF framework has separated both things by providing presentation on the JSP page and by encapsulating the behavior of the tag inside the JSF component class, enhancing reusability of components.

Model bean wrappers for application's business logic and data

Model beans are application-specific Java classes that wrap the business logic of a JSF application. The purpose of model beans is to provide application-specific data to JSF components.

The JSF components use the data provided by model beans and update the model beans with fresh data coming from the user.

For example, look at the following use of the `selectManyCheckbox` tag from our shopping cart application:

```
<jsf:selectManyCheckbox
  value="{productData.selectedOptionalFeatures}">
  <f:selectItems value="{productData.optionalFeatures}"/>
</jsf:selectManyCheckbox>
```

The value of the "value" attribute refers to a model bean and its property. The value attribute of the `selectManyCheckBox` tag above has a string value "{productData.selectedOptionalFeatures}." This string specifies a model bean and its property. In this case, `productData` is an application-specific model bean and `selectedOptionalFeatures` is one of its properties that stores the options selected by the user.

Similarly, you can check the value of the value attribute of the `f:selectItems` child of the `selectManyCheckbox` element. `productData` is the same model bean whose `optionalFeatures` property contains the list of all the options available for the user.

Recall Views of the shopping cart on page 7, in which we provided a screenshot for the product-specification view. That screenshot contains optional features; the JSF component class against the `selectManyCheckbox` tag will interact with the model bean and access the optional features from model beans property mentioned in the "value" attribute above. We will discuss the interaction of a JSF component with model beans in Associating model beans with JSF components on page67 .

Before we detail the concepts introduced in the above section, you should understand the JSF framework, which is explained in the next section.

The JSF framework

To explain how the JSF framework operates, we have to discuss the following classes, which play an important role in the JSF framework:

- The `FacesServlet` class
- The `Lifecycle` class
- The `FacesContext` class

The `FacesServlet` class initiates the JSF framework. As you know, any class that wants to become a servlet has to implement the Servlet interface. The `FacesServlet` class implements the Servlet interface to become a servlet.

The JSF framework revolves around request-response life cycles. We call this request-response life cycle the JSF processing life cycle. The `Lifecycle` class manages the complete JSF processing life cycle.

A JSF request goes through six phases, which we'll introduce in Life-cycle processing phases of a JSF application on page 77. The`Lifecycle` class executes each phase of the JSF framework.

The `FacesContext` class wraps all the information about a JSF request. We will discuss the `FacesContext` class in detail in The FacesContext class on page36 .

How to make FacesServlet handle your requests

When a server-side Java application receives a request from a browser, it needs to invoke a servlet. If the request is for a JSF application, the `FacesServlet` should be the servlet to handle the incoming request.

To instantiate the `FacesServlet` class, you need to tell your servlet container to use `FacesServlet` to process JSF requests. For this purpose, you need to make the following entries in your web.xml file:

```
 <servlet>
   <servlet-name>Faces Servlet</servlet-name>

<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>*.faces</url-pattern>
 </servlet-mapping>
```

Note the following points in the markup above:

- The servlet element wraps three types of information in three different elements: servlet name, servlet class, and load on startup.

- The `servlet-name` element specifies the servlet name, which the JSF framework will use to access the servlet. The name of the servlet is "Faces Servlet."

- The `servlet-class` element wraps the fully qualified name of the servlet class. Above, the `servlet-class` element contains `javax.faces.webapp.FacesServlet`.

- The content of the `load-on-startup` element indicates whether the servlet should be loaded when an application starts.

- The `servlet-mapping` element contains `servlet-name` and `url-pattern` child elements.

- The `servlet-name` element wraps the name of the servlet that should be loaded on finding a specific pattern in the request URL.

- The `url-pattern` element wraps the pattern string, which, if found in the request URL, will result in the invocation of the servlet specified by the `servlet-mapping` element.

The `servlet-mapping` element above tells the servlet container that whenever the string ".faces" comes in the URL, the `javax.faces.webapp.FacesServlet` class should handle the request. For example, look at the following:

```
http://localhost:8080/XCart/index.faces
```

When the container sees the ".faces" string in the URL, it loads the `FacesServlet` as described above.

Now, let's see how the `FacesServlet` will handle a JSF request.

Methods in the FacesServlet class

Like all servlets, the `FacesServlet` implements the `init()` and `service()` methods.

After selecting the `FacesServlet`, the servlet container calls the `init()` method of the `FacesServlet` class to instantiate the servlet. The `init()` method must complete successfully before the servlet can receive any requests.

The `init()` method of the `FacesServlet` class creates an instance of the `Lifecycle` object. The `Lifecycle` object is common across all JSF sessions. This instance of the `Lifecycle` class will be used to execute different phases of the JSF processing life cycle. We will discuss the `Lifecycle` class in detail in the next section.

Whenever the container receives a JSF request, it will call the `service()`

method of the `FacesServlet` class, passing the request along with the method call. The `service()` method does the following:

1. It uses appropriate factory classes to instantiate a `FacesContext` object. We'll discuss the `FacesContext` class in detail in The FacesContext class on page36 .

2. It then populates the `FacesContext` object with request data.

3. Next, it calls the `execute()` method of the `Lifecycle` class, passing the `FacesContext` object along with the method call. The call for the `execute()` method starts the JSF processing life cycle. We will discuss the `Lifecycle` class in detail in the next section.

4. After the `execute()` method call, it calls the `render()` method of the `Lifecycle` class, passing the `FacesContext` object along with the method call. The `render()` method generates the response markup against the requested JSP page.

Now, let's see how the `Lifecycle` class manages the JSF request life cycle.

The Lifecycle class

The `Lifecycle` class executes and manages the JSF request processing life cycle for a JSF request.

The `Lifecycle` class contains two important methods: `execute()` and `render()`. We said that the `service()` method of the `FacesServlet` class calls the `execute()` and `render()` methods and passes it the `FacesContext` object.

The `execute()` and `render()` methods are of special importance because all JSF request processing is accomplished by these two methods.

Now let's see the `FacesContext` class.

The FacesContext class

All the information regarding a JSF request and response resides inside a context named `FacesContext`. The object of `FacesContext` was instantiated by the `service()` method of `FacesServlet` after receiving a request from the client.

The `FacesContext` class contains the methods you can use to fetch information regarding the application context. For example, when any component requires application-specific information, it calls the `getApplication()` method of the `FacesContext` class. We will use the `getViewRoot()` method to get the root JSF component of a JSP page.

We will use many methods of the `FacesContext` class extensively throughout this tutorial. You will learn more about the `FacesContext` class as we proceed.

In the next section, we will explain the sequence of events that occurs when a user requests a JSF application.

The URL processing life cycle in a JSF application

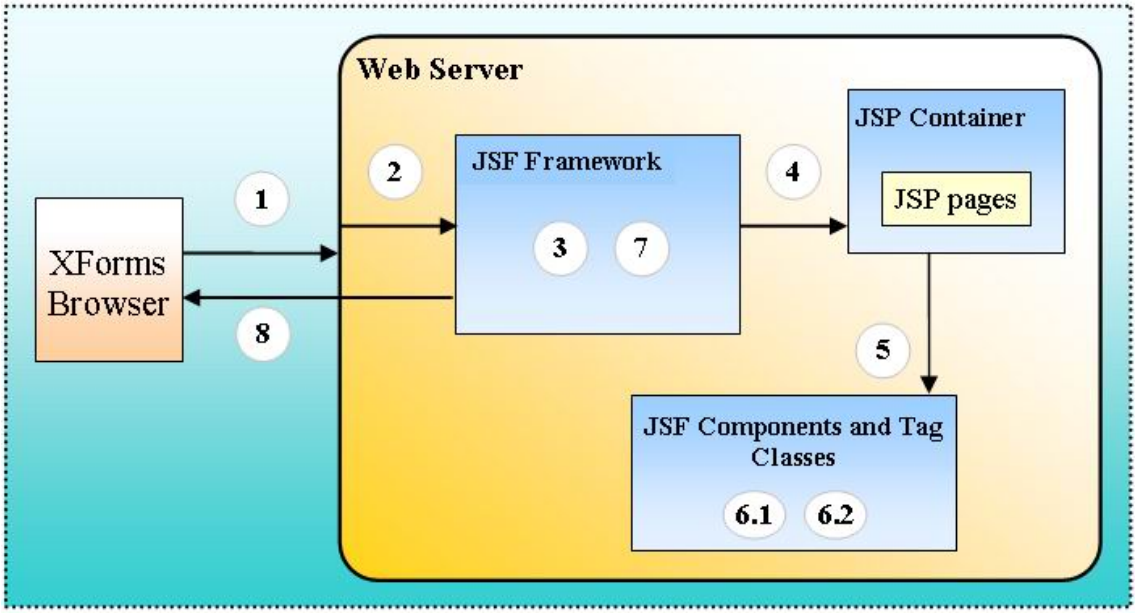
We will explain the URL processing life cycle for a JSF application in two scenarios.

The first scenario (Scenario I) will discuss the sequence of events that occurs when a user makes a request for a JSF page by entering the URL in the address bar of a browser. This scenario takes place when a user requests the first page of a JSF application.

The second scenario (Scenario II) will discuss the sequence of events that occurs when a user makes a request by interacting with a JSF component (such as a button or hyperlink) rendered in response to the first request. This scenario normally takes place when a user visits subsequent pages of a JSF application.

In the following sequence of events, we will only discuss the high-level details. For example, we will say that the JSF framework calls the `decode()` method, but we will not go into the implementation detail of it.

Scenario I



The events of the figure:

1. Suppose the user requests the following URL:

   ```
   www.aFictitiousShoppingCart.com
   ```

2. When the Web server receives this request, it invokes the `FacesServlet` class. The Web server calls the `init()` method of the `FacesServlet` class to load the servlet. Then it calls the `service()` method (by passing its request and response details) of the `FacesServlet` object. The `service()` method creates an instance of the `FacesContext` object and calls the `execute()` and `render()` methods of the `Lifecycle` class one by one, passing the `FacesContext` object along with each method call.

3. The `execute()` method checks whether the root of the JSF component (the `UIViewRoot` object) exists. Because it is the first request for the page, the root does not exist. So, it creates a new `UIViewRoot` object and stores it in the `FacesContext` object.

4. The `render()` method dispatches the JSF request to the JSP container that executes the JSP page.

5. The JSP container goes through the JSP page. When it comes across any JSF tag (such as the `f:view` tag), it invokes its tag handler class (as we said, each JSF tag has a tag handler class associated with it).

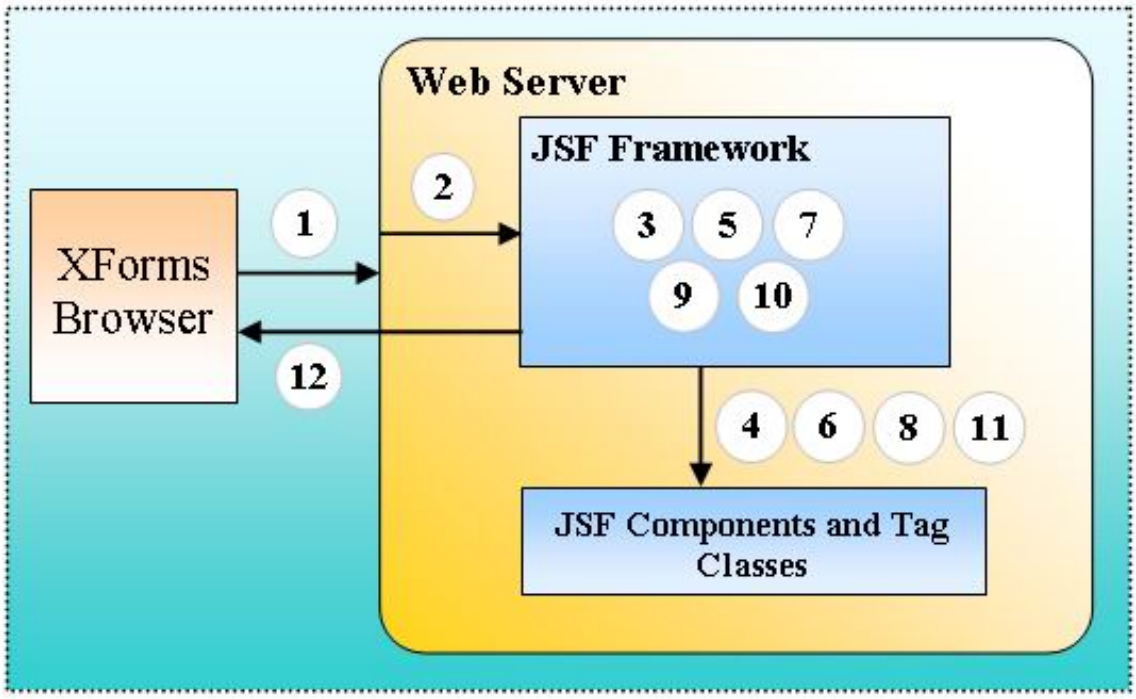
6. The tag handler class does two important functions:
   1. It adds the JSF component associated with the tag to the root component (`UIViewRoot`). As a result, all the components will be added in a tree (hereafter referred to as a component tree), where the `UIViewRoot` object sits as the root of the component tree.

2. It calls the encode methods of the JSF component to render its markup.

7. Next, the JSF framework saves the view in the session.

8. Finally, it sends the generated markup back to the user.

Scenario II



The events of the figure:

1. Suppose the user clicks some button in response to the first request. The following URL will be generated:

```
www.aFictitiousShoppingCart.com?id1=Submit
```

2. The Web server receives the request and initializes the `FacesServlet` class (refer to step 2 in Scenario I for the initialization of `FacesServlet`).

3. The JSF framework checks whether the root of the component tree exists. This time, it finds the `UIViewRoot` object and sets the `UIViewRoot` component in the `FacesContext` object as the root of the component tree. Recall from step 6 of Scenario I that we have already added all components into the component tree.

4. Next, the JSF framework iterates through the JSF component tree and calls the `decode()` method of each JSF component. The `decode()` method parses the request and extracts the user data from the request to be stored

in model beans. It also detects which events occurred on the client side (such as which button was clicked), and fires appropriate action events.

5. The JSF framework checks for the events fired during the decoding. If any action event is added during decoding, the JSF framework calls the appropriate application-specific event handler classes.

6. After decoding and processing the event, the JSF framework calls the `validate()` method of each JSF component in the component tree. The `validate()` method validates the format of the user's data, and compares the old and new value of the component. If the two values differ, it fires a value-change event.

7. At the end of the validation, the JSF framework checks for unhandled events. If any event is added during validation, the JSF framework calls the appropriate application-specific event handler.

8. After validation and event processing, the JSF framework calls the `updateModel()` method for each component in the tree one by one. As a result, the new values, which the `decode()` method parsed in step 4 above, will be stored in the appropriate model beans.

9. The `updateModel()` method call can generate more events. After the `updateModel()` call, the JSF framework checks for unhandled events in the `FacesContext` class. If an event is added during the update, the JSF framework executes the event handler class associated with the component.

10. At this point, we may need to invoke a new JSP page, depending on some navigation rules. (We'll explain navigation rules in detail in Navigation process on page 75.) The JSF framework will check whether we need to invoke a new JSP page.

11. If we don't need to invoke a new JSP page, the JSF framework simply calls the encode methods of all components in the component tree.

12. Finally, the JSF framework sends the generated markup back to the user.

If a new JSP page needs to be invoked in step 10, Scenario I will be repeated starting from step 4.

In the next section, we will explain some special handling required when an XForms-JSF application is requested by the user.

The URL processing life cycle in an XForms-JSF application

Scenario I for an XForms-JSF application is the same as for a JSF application. The difference between processing XForms-JSF and JSF requests will be explored in Scenario II.

Recall step 9 of User interaction and flow of data in our shopping cart application on page 22, where we explained that when the user clicks an XForms button, a request is made with application-specific XML in the body of the request.

When the user clicks a button, the following XML is sent to the server in the body of the request:

```
<data>
  <action-performed>_id1</action-performed>
</data>
```

The `action-performed` tag wraps the ID of the button clicked by the user. This ID tells the button component that the user clicked the button. The special handling we need for XForms-JSF applications is to process XML. We will explain the `action-performed` tag and this ID concept in detail in Implementing the xforms-jsf:model component on page83 .

We have discussed only the high-level view of the JSF framework. We did not go into the details of individual concept. In the next section, we'll explore further details of concepts discussed here.

Section 4. How does a JSF component tree work?

The JSF component tree

We introduced the idea of the JSF component tree in The JSF architecture on page 29. Now we'll explain and demonstrate how the tree works.

First, we'll present a JSP page and graphically show the tree of components in the JSP page. We'll then discuss how JSF tags and components are associated with each other, and how they work together. Finally, we'll discuss the methods of JSF component classes.

Look at the following JSP page:

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<head>
  <title>JSF Demos</title>
</head>
<body>
  <f:view>
    <h:selectOneRadio id="colorsList"
      value="#{dataStore.selectedColor}">
      <f:selectItems id="colors" value="#{dataStore.colors}"/>
    </h:selectOneRadio>
    <h:selectManyCheckbox id="productsList"
      value="#{dataStore.selectedProducts}">
      <f:selectItems id="products"
value="#{dataStore.products}"/>
    </h:selectManyCheckbox>
    <h:commandButton label="Add to cart" id="cartView">
      <f:actionListener type="cart.DemoActionListener"/>
    </h:commandButton>
    <h:commandButton label="Back to product view" id="prodView">
      <f:actionListener type="cart.DemoActionListener"/>
    </h:commandButton>
  </f:view>
</body>
</html>
```

When the request for this JSF page is made, the JSF framework constructs a tree of all the components on the page. Now let's see how this tree is constructed.

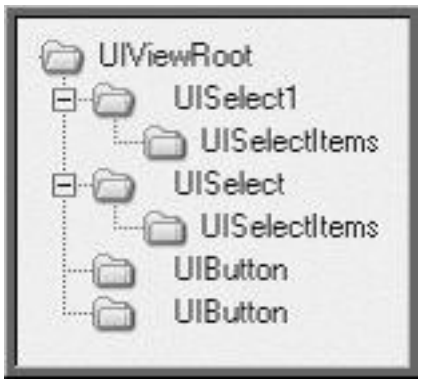
When the first request for a JSF page (see Scenario I in The URL processing life cycle in a JSF application on page 37) is made by a user, the JSF framework does the following steps to construct the component tree:

1. The JSF framework first creates the `UIViewRoot` object, which corresponds to the `f:view` tag and forms the default root of the component tree.

2. Then it goes through the requested JSP page.

3. When it comes across a JSF tag in the JSP page, it inserts the JSF component object associated with the tag into the component tree.

Diagrammatically, the component tree for the JSP page above looks like the following figure:



All the components are under the `UIViewRoot` component. The root component contains the `UISelect1` component class (which corresponds to the `h:selectOneRadio` tag), the `UISelect` class (which corresponds to the `h:selectManyCheckbox` tag), and the `UIButton` class (which corresponds to the `h:commandButton` tag) child components. The `UISelect1` and `UISelect` components in turn contain the `UISelectItems` class (which corresponds to the `f:selectItems` tag).

The UIComponentTag class

In the previous section, we saw that the JSF tags and components are associated with each other. Now we'll see how this association works.

In the JSP page, each JSF tag has a tag handler class associated with it. This tag handler tracks the attributes specified in the JSF tag and wraps the type of the JSF component and renderer classes associated with the tag. We have already introduced the concept of the JSF component class. The renderer class generates the markup for the component on the response stream. A JSF tag may or may not have a renderer class associated with it because a component class can do all the functionality that a renderer class does. In Rendering a component on page 60, we will discuss the renderer classes in detail.

The JSF framework provides a `UIComponentTag` class that implements an interface named Tag. When you develop a component, you only have to extend from the `UIComponentTag` class or one of the `UIComponentTag` children classes. The `UIComponentTag` is an abstract class, so you cannot instantiate it. You always extend this class. The `UIComponentTag` class has two abstract methods: `getComponentType()` and `getRendererType()`. Your tag handler class should implement both.

For example, in the JSP page above, there is an `xforms-jsf:selectOneRadio` JSF tag. The tag handler class associated with the `xforms-jsf:selectOneRadio` tag looks like the following:

```
public class Select1RadioTag extends UIComponentTag{
  ......
  public String getComponentType() { return "Select1"; }
  public String getRendererType() { return null; }
  ......
  public void setProperties(UIComponent component){}
}//Select1RadioTag
```

Notice the following points:

- The `Select1RadioTag` class extends the `UIComponentTag` class.
- It implements the `getComponentType()` and `getRendererType()` abstract methods of the `UIComponentTag` class. Both methods return a `String` object.
- The `getComponentType()` method returns a string that represents the type of the JSF component associated with the JSF tag.
- Similarly, the `getRendererType()` method returns the string that represents the renderer type associated with the JSF tag. The `getRendererType()` method in the `Select1RadioTag` class returns `null`, indicating that there is no renderer associated with this JSF tag. We will discuss the details of renderer classes in Rendering a component on page60 .
- The `Select1RadioTag` class also overrides the `setProperties()` method of the `UIComponentTag` class. The `setProperties()` method is used to pass on the values of the properties from the tag handler class to the corresponding component class. These properties are actually the attributes passed by the JSP author in the JSF tag while using the JSF tag in the JSP page. We will explain the `setProperties()` method in the Setting properties of a component on page 51 section.

The JSF framework calls the `getComponentType()` method of each JSF tag in a JSP page to know the JSF component associated with a particular JSF tag. In Associating a tag with a component on page 54, we will discuss how this component type is used to extract the JSF component class from a JSF configuration file named faces-config.xml.

The JSF framework also calls the `getRendererType()` method of each JSF tag in a JSP page to know the renderer associated with the tag.

But how will the JSF framework know which tag handler class is associated with a particular JSF tag? The next section explains the association between the

JSF tag and the tag handler class.

Associating JSF tags with tag classes

In this section you'll learn how:

- A JSP compiler knows that a JSF tag is associated with a particular tag handler class
- A tag library declares the JSF tags and tag handler classes for a particular JSF tag
- The JSP author includes a tag from a tag library in a JSP page

To explain how a JSP compiler knows that a JSF tag is associated with a particular tag handler class, we'll use a tag library descriptor (TLD) file. A TLD file contains information like the library version, general description, and declaration of all the tags that a tag library supports. The declaration of each tag contains the tag name, tag handler class, and attributes a tag can support.

The TLD file is not a JSF-specific file. You can have your own non-JSF tag libraries. We call them generally as custom tag libraries. All JSF tags are custom JSP tags, and a TLD file is required in all custom JSP libraries.

The TLD file also specifies that a particular tag is associated with a particular tag handler class. To define a JSF tag, you have to declare it in the TLD file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib>
  ......
  <tag>
      <name>selectOneRadio</name>
      <tag-class>xforms_jsf.Select1RadioTag</tag-class>
      <attribute>
        <name>id</name>
        <required>false</required>
      </attribute>
      <attribute>
        <name>ref</name>
        <required>true</required>
      </attribute>
      ......
  </tag>
  <tag>
    <name>commandButton</name>
    <tag-class>xforms_jsf.ButtonTag</tag-class>
    <attribute>
      <name>id</name>
      <required>false</required>
    </attribute>
    <attribute>
```

```
      <name>action</name>
      <required>false</required>
    </attribute>
    ......
  </tag>
  ......
</taglib>
```

Notice the following points in the above TLD file:

- All the `tag` elements of a TLD file are enclosed within a single `taglib` element.
- A `tag` element wraps the information of a single JSF or custom tag.
- The name of the JSF tag is defined within the `name` element.
- The fully qualified name of the tag handler class is specified inside the `tag-class` element.
- The entry of the `name` and `tag-class` elements is mandatory in a tag declaration.
- The `attribute` element declares a single attribute for the particular tag. There can be any number of `attribute` elements in a `tag` element.
- The `attribute` element has different subelements to define a JSF tag's attributes properly, such as name, required, type, etc. The `name` element is necessary for the `attribute` element; the rest are optional.
- The `name` element wraps the name of the attribute.
- The `required` element indicates whether this attribute is mandatory while this tag is used in the JSP page. Its value is specified as "true" or "false" (by default, it's false).
- The `type` element indicates the data type of the attribute (by default, it's string).

For example, in the `selectOneRadio` tag declaration, we have a `ref` attribute. We declare this attribute in a TLD file like this:

```
<attribute>
  <name>ref</name>
  <required>true</required>
</attribute>
```

The `required` element wraps true, which indicates that whenever the `selectOneRadio` tag is used in the JSP page, the `ref` attribute must come

within the tag.

Each attribute declared in the TLD file for a particular JSF tag should have a setter method in its respective tag handler class. The JSF framework calls these setter methods to pass on the values from the JSF tag (in the JSP file) to the tag handler class.

In the above listed TLD file, different attributes are declared for the `xforms-jsf:selectOneRadio` JSF tag. All these attributes should have setter methods in the tag handler class. For example, look at the following tag handler class:

```
public class Select1RadioTag extends UIComponentTag{
  ......
  public void setId(String id){
    this.id = id;
  }
  public void setRef(String ref){
    this.ref = ref;
  }
  ......
}//Select1RadioTag
```

The attributes declared for the `selectOneRadio` tag in the above TLD file have setter methods in the tag handler class. For example, the `ref` attribute declared for the `selectOneRadio` tag has a `setRef()` method in the `Select1RadioTag` class.

Now let's see how the JSP author includes a tag from a tag library in a JSP page. The `taglib` directive of the JSP page is used to include a tag library in the JSP page. The tags declared in the tag libraries are accessed with the prefix defined in the `prefix` attribute of the `taglib` directive. We have already discussed the details of including the tag library in the JSP page in Using JSF tags in JSPs on page 31.

Now let's see how the JSP container maps a JSF tag to its corresponding tag handler class. The following sequence of events enables this mapping:

1. The JSP container sees a JSF tag with a particular prefix.

2. It maps the prefix in the `taglib` directive to identify the right TLD file.

3. It opens the TLD file.

4. It searches the tag name inside the `name` element of each `tag` in the TLD file.

5. After finding the right tag definition, it loads the tag handler class specified in the `tag-class` element of the tag identified in step 4.

You will place the TLD file in the `\WEB-INF` directory of the application. For

example, look at the sample code folder named Demo in the section4.zip file available in the source code download of this tutorial; see Resources on page ?. The Demo folder only shows the directory structure and a few files. We'll add more files in later sections as we develop more code for the XForms-JSF tag library.

At the moment, the folder hierarchy and arrangement of files in the sample code folder looks like this:

Notice that the TLD file for the XForms-JSF tag library resides inside the `\WEB-INF` folder.

Before going further, we would like to explain two classes named `ValueBinding` and `MethodBinding`. The introduction of these classes is important in understanding how to pass application data from the tag handler to the corresponding component classes.

The ValueBinding class

The `ValueBinding` class is used to get application-specific data from model beans. A JSF tag handler or component class can invoke the `ValueBinding` class when it needs to get or set the value of a property from a bean.

To get or set the value of a model bean property, we need to:

1. Get an instance of the `ValueBinding` class
2. Use the instance to get or set the property value

Look at the following code segment used to get an instance of the `ValueBinding` class:

```
Application application = facesContext.getApplication();
ValueBinding binding = application.createValueBinding(value);
```

In the first statement, we simply fetched an instance of the `Application` class from the `FacesContext` object.

Notice that the `FacesContext` object we have used to fetch the `Application` object represents the context in which you want to find the model bean property.

The `Application` class holds all information related to a complete JSF application (for instance, it can return a component object corresponding to a particular component type or it can return any model bean object in the application context by using the `ValueBinding` class).

In the second statement, we have called the `createValueBinding()` method of the `Application` object. The `createValueBinding()` method call takes a string value, which represents the name of a model bean and its property. The call to the `createValueBinding()` method returns an instance of the `ValueBinding` class. For example, if we pass the "#{myModelBean.property}" string to the `createValueBinding()` method, it returns a `ValueBinding` object that contains a reference to the `getProperty()` method of the `myModelBean` object present in the application context.

Now we simply call the `getValue()` method of the `ValueBinding` object to get the value of the model bean property that was passed to the `createValueBinding()` method call:

```
Object value = binding.getValue(context);
```

The `getValue()` method takes an instance of the `FacesContext` class and returns the value of the property.

The return value from this method is `Object` type, so you have to cast it into the appropriate application-specific class.

Recall the `value` attribute, which we discussed in Model bean wrappers for application's business logic and data on page 33. If we pass the value of the `value` attribute (which is in fact a reference of the model bean property) to the `createValueBinding()` method, it returns the `ValueBinding` object for the model bean property. For example, the following code shows the declaration of a JSF tag:

```
<jsf:selectOneRadio value="#{productData.products}"/>
```

The following code (written anywhere in the tag handler class) returns the object corresponding to the `productData.product` property:

```
String value = getValue();
Application application = facesContext.getApplication();
ValueBinding binding = application.createValueBinding(value);
Object value = binding.getValue(context);
```

The `ValueBinding` class has another method named `setValue()`, which sets the value of the model bean property. The `setValue()` method takes two parameters: a `FacesContext` object and the value we want to set in the model bean property:

```
String value = getValue();
Application application = facesContext.getApplication();
```

```
ValueBinding binding = application.createValueBinding(value);
//In this line you will create a new object named newValue.
binding.setValue(context, newValue);
```

Next, we'll discuss the `MethodBinding` class.

The MethodBinding class

The concept behind the `MethodBinding` class is the same as we discussed for the `ValueBinding` class. The `ValueBinding` class was used to bind or fetch the value of a property of a model bean. On the other hand, the `MethodBinding` class is used to bind or call a specific method of a model bean.

You may use the `MethodBinding` class whenever you need to call a particular method of some model bean from your tag handler or component class.

To call a method of a model bean:

1. Get the attribute value of a JSF tag, which specifies the method of a model bean you want to call.

2. Create an array of Objects. Each object in the array represents a parameter in the method call, so the array of Objects represents the signature of the method you want to call.

3. Get an `Application` object from the context.

4. Get an instance of the `MethodBinding` class from the application.

5. Finally, use the method binding object to call or invoke the method.

For example, suppose the following code is the declaration of the JSF tag:

```
<jsf:selectOneRadio method="#{myModelBean.myMehod}"/>
```

The method attribute above refers to `myModelBean.myMethod()`, for which we have to create a binding. Furthermore, suppose the following code is the signature for `myMethod()` in the model bean:

```
public void myMethod(Class1 c1, Class2 c2)
```

Now we'll perform the above five steps to invoke `myMethod()`. First we fetch the value of the `method` attribute:

```
String method = getMethod();
```

Then we'll create an array of Objects containing `Class1` and `Class2` objects. In the method signature above, the `myMethod()` call takes these two objects as arguments.

```
Object [] args = {new Class1(), newClass2};
```

Next, we'll fetch an instance of the `Application` class from the faces context:

```
Application application = facesContext.getApplication();
```

Notice that the `FacesContext` object we used to fetch the `Application` object represents the context in which you want to find the model bean and invoke its method.

The `Application` class contains a method named `createMethodBinding()`, which returns an instance of the `MethodBinding` class, so we call the `createMethodBinding()` method of the `Application` object.

The `createMethodBinding()` method call takes two parameters: a string value, which represents the model bean and its method name (the value of the method attribute in the JSF tag), and an array of Objects containing parameters for the method call. The call to the `createMethodBinding()` method returns an instance of the `MethodBinding` class, which contains a reference to the `myMethod()` method of the `myModelBean` object present in the application context:

```
MethodBinding mb = application.createMehtodBinding(value, args);
```

The `MethodBinding` class contains a method named `invoke()`. This method takes two parameters: a `FacesContext` object (the same `FacesContext` object that we used to fetch the `Application` object) and an array of Objects that contain the list of parameters you want to pass along with the method call:

```
mb.invoke(context, args);
```

The call to the `invoke()` method results in a call to the method of model bean for which the `MethodBinding` object was created.

The concept of the `MethodBinding` and `ValueBinding` classes will be clearer when we use both concepts in the next section.

Setting properties of a component

We mentioned a `setProperties()` method while discussing the tag handler class in The UIComponentTag class on page 43. You have seen that attributes of a particular tag declared in the TLD file are defined as properties of the tag handler class. This section explains how the `setProperties()` method makes these attributes and properties available for the JSF component class.

In JSF tags and JSF components on page 30, we explained that the component class implements the behavior of a JSF tag. So the component class needs to know all the information specified by the JSP author for a JSF tag in the JSP page. Therefore, the tag handler class needs to set the properties of a tag in the component class.

For example, the tag handler class for the `selectOneRadio` tag implements the `setProperties()` method to set the properties for the associated JSF component class. The following code shows the `setProperties()` method in the `Select1RadioTag` class:

```
public class Select1RadioTag extends UIComponentTag{
  ......
  public void setProperties(UIComponent component){
    super.setProperties(component);
    UISelect1 uis = (UISelect1)component;
    FacesContext fc = FacesContext.getCurrentInstance();
    Application app = fc.getApplication();
    if(value != null){
      if(UIComponentTag.isValueReference(value))
      {
        ValueBinding vb = app.createValueBinding(value);
        uis.setValueBinding("value", vb);
      }
      else
        uis.setValue(value);
    }
    if(valueChangeListener != null){
      if(UIComponentTag.isValueReference(valueChangeListener)){
        Class args[] = {
          javax.faces.event.ValueChangeEvent.class
        };
        MethodBinding mb = app.createMethodBinding
          (valueChangeListener, args);
        uis.setValueChangeListener(mb);
      }
    }//if(valueChangeListener != null)
    if(ref != null){
        uis.getAttributes().put("ref", ref);
    }
    .......
  }
}//Select1RadioTag
```

Notice that the JSF component object (component) is passed as a parameter to the `setProperties()` method. The `setProperties()` method calls its super `setProperties()` method, which sets some default properties for the component (like setting the renderer for the component).

Next, the `setProperties()` method fetches an instance of the `FacesContext` class and gets the instance of the `Application` class from

the `FacesContext` class. This `Application` object does two things. First, we'll use the `Application` object to create a `ValueBinding` object for the `value` attribute. Second, we'll use the `Application` object to create a `MethodBinding` object for the `valueChangeListener` attribute. The `valueChangeListener` attribute is used to specify the handler for value change events, which we'll discuss in detail in Handling value-change events on page71 .

Here's how the `setProperties()` method creates the `ValueBinding` object for the `value` attribute:

1. The `setProperties()` method first checks if the value of the `value` attribute is not `null`. If it's not, it checks whether the value of the `value` attribute is a reference value by calling the static `isValueReference()` method. The `isValueReference()` method internally checks if the `value` attribute specifies a valid reference to a model bean property present in the application context. If it is not a valid reference, the `isValueReference()` method returns false.

2. If the `value` attribute is a reference, the `setProperties()` method calls the `createValueBinding()` method of the `Application` class, passing the `value` property along with the method call.

3. Next, the `setProperties()` method passes the `ValueBinding` object to the component class, so that it can use the binding whenever it's needed.

Here's how the `setProperties()` method creates the `MethodBinding` object for `valueChangeListener`:

1. The `setProperties()` method checks if the `valueChangeListener` property value (passed by the JSP page author as the value of the `valueChangeListener` attribute) is not `null`.

2. It checks whether the `valueChangeListener` value is a reference to a model bean method by calling the static `isValueReference()` method.

3. If the `valueChangeListener` property is a valid reference, the `setProperties()` method calls the `createMethodBinding()` method of the `Application` class, passing the `valueChangeListener` and arguments along with the method call, which returns the `MethodBinding` object.

4. Next, the `setProperties()` method passes the `MethodBinding` object to the component class so that the component class can call the value change listener method specified by the `valueChangeListener` attribute.

5. Finally, the `setProperties()` method checks the value of the `ref` property. The `ref` property is an XForms-specific attribute, which refers to a particular tag in the `model` element. We described the XForms `ref` attribute in The select element on page 18.

If the `ref` attribute is not `null`, the `setProperties()` method calls the `getAttributes()` method of the component, which returns a `Map` object. This `Map` object contains a list of all attributes available in the component class.

We simply call the `put()` method of the `Map` object, passing it the property name ("ref") and value, along with the method call, which results in the addition of another attribute in the list of attributes already available in the component class.

This is another useful technique to pass information from a tag handler class to a component class.

We have discussed three things a `setProperties()` method may do (value binding, method binding, and putting attributes directly into the `Map` object). We will use these techniques often in this tutorial. Now whenever we implement the `setProperties()` method, we'll refer to this section and not go into detail each time.

Associating a tag with a component

We have seen how the JSP container maps a JSF tag to a tag handler class and how the tag handler class manages its attributes. Now let's see how to relate a tag handler class to a JSF component class.

A JSF tag library developer can develop many component classes against a single JSF tag. It is up to an application developer to decide which component he wants to associate with a particular JSF tag.

We need a mechanism that enables us to decide the tag-to-component mapping during application development.

Your JSF application includes a JSF configuration XML file named faces-config.xml, which specifies component classes against component types:

```
<?xml version="1.0"?>
<faces-config>
   ......
   <component>
      <component-type>Select1</component-type>

<component-class>xforms_jsf.UISelect1</component-class>
   </component>
   <component>
      <component-type>Select</component-type>

<component-class>xforms_jsf.UISelect</component-class>
   </component>
   ......
</faces-config>
```

Note the following points from the above:

- The faces-config.xml file contains a number of `component` elements.
- The `component` element in the faces-config.xml file wraps the type of the component and JSF component class against that type.
- It wraps this information with the help of `component-type` and `component-class` elements as shown above. Each `component` element contains `component-type` and `component-class` child elements.
- The `component-type` element specifies the type of the component.
- The `component-class` element wraps the fully qualified name of the JSF component class.

Here's how the JSF framework uses the faces-config.xml file to load the JSF component against a particular JSF tag, as outlined in the following points:

- The JSF framework calls the `getComponentType()` method of a tag handler class, and the method call returns a string value that represents the type of component in the configuration file.
- The JSF framework locates which `component-type` element in the faces-config.xml file wraps the string returned by the `getComponentType()` method of the tag handler class.
- When it finds a matching component, it reads the `component-class` element that forms a pair with the `component-type` element and learns about the JSF component class associated with the JSF tag.

For example, let's suppose the `getComponentType()` method of a `Select1RadioTag` class returns "Select1." In the faces-config.xml file above, the component class whose type matches "Select1" is `xforms_jsf.UISelect1`.

You will write your faces-config.xml files during application development and match the component types with appropriate component classes.

The faces-config.xml file is then placed in the `/WEB-INF` directory of the application.

The UIComponentBase class

We have seen how the JSF tags are associated with the JSF components. Now it's time to see how components in a JSF component tree work.

The `UIComponent` abstract class is the base class for all the components. It represents the functionality or behavior of the JSF component. You can think of `UIComponent` as an abstraction of the functionality of a JSF component. The JSF tag library developers extend the `UIComponent` abstract class, thus allowing the JSF framework to interact with the component.

The `UIComponent` abstract class contains many methods, where each method represents some functionality of the component. Not all components require the whole set of functionality.

The JSF framework provides a `UIComponentBase` class, which already extended the `UIComponent` class. When you are developing your own customized components, you can extend the JSF component class from the `UIComponentBase` class or from its subclasses.

The `UIComponentBase` class contains various methods that the JSF framework calls at appropriate times during a JSF request-processing life cycle. For example, the JSF framework calls the `decode()` method of the `UIComponentBase` class when the user clicks a button or hyperlink in the JSF page (refer to step 4 of Scenario II in The URL processing life cycle in a JSF application on page 37.

The two most important functionalities of a JSF component are decoding and encoding. In the next two sections, we will offer a detailed discussion of these two functionalities of a JSF component. However, note that a JSF component has functions other than decoding and encoding. We'll explain these two first to demonstrate the working of a JSF component tree. Later, we'll demonstrate other functionality of a JSF component.

The decoding process

The JSF framework calls the `decode()` method of each JSF component in the tree one by one. Recall The URL processing life cycle in a JSF application on page 37, where we presented two types of request scenarios. For Scenario I, the `decode()` method was not called, but whenever the request is made according to Scenario II, the `decode()` method is called for each request.

The purpose of calling the `decode()` method is to allow each component in the tree to decode the request value parameters (or data in any form) into a Java object. Look at the following URL generated by clicking a button on the JSF page:

```
www.aFictitiousShoppingCart.com?id0=red;id1=btn1
```

When the JSF framework receives this request, it calls `decode()` methods of each component in the tree one by one. The `decode()` method of each component does the following:

- It checks whether its ID exists in the request. For example, in the URL above, there are two name-value pairs: `_id1=red` and `_id2=btn1`. The `_id1=red` pair may correspond to a check box component, and the `_id2=btn1` pair may correspond to a button component.

- If the component ID the request contains matches the ID of the component whose `decode()` method is called, it fetches the values against this ID from the request. For example, the `decode()` method of the check box component fetches the string "red" as its value.

- Next, it stores this value in the component by calling the `setSubmittedValue()` method of the component class.

- If the `decode()` method is of a button or hyperlink component, instead of fetching the value from the request, it simply fires an action event. An action event represents a user's action (like clicking a button). We will explain the firing and handling of action events in Event-generation mechanism in JSF on page 70 and Handling action events on page74 .

The `decode()` method takes the `FacesContext` object as a parameter. In The FacesContext class on page 36, we explained that the `FacesContext` object contains all the information regarding a JSF request and response.

Look at the following `decode()` method implementation:

```
public void decode (FacesContext context){
    //step-1
    if(fc == null)
       throw new NullPointerException();
    //step-2
    String clientId = getClientId(fc);
    //step-3
    //Gets incoming request and fetches the id and the user data
    //against the id from the request
    //step-4
    if(clientId.equals(id))
    {
      setSubmittedValue(value);
    }
    //step-5
    setValue(true);
}//decode
```

Notice the following steps in the `decode()` method above:

1. The `decode()` method first verifies the `FacesContext` object passed to it. If it is `null`, it throws a `NullPointerException`.

2. It calls the `getClientId()` method of the component, which is implemented by the `UIComponentBase` class. The `getClientId()` method returns an ID for the component that uniquely identifies the component.

3. The `decode()` method gets the incoming request and searches the component ID in the request.

4. If the component ID is found in the request, it fetches the data against this ID and saves it.

5. Finally, it calls the `setValid(true)` method of the component class, which tells the component that the decoding process was successful.

If the `decode()` method belongs to a button component, the method implementation is slightly different:

```
public void decode (FacesContext context){
    //step-1
    if(fc == null)
       throw new NullPointerException();
    //step-2
    String clientId = getClientId(fc);
    //step-3
      //Gets incoming request and fetches the id from the request
    //step-4
    if(clientId.equals(id))
    {
      queueEvent(new ActionEvent(this);
    }
    //step-5 not needed
}//decode
```

The main difference comes in step 4 where the `decode()` method fires an action event (by calling the `queueEvent()` method, which we will discuss in Event-generation mechanism in JSF on page 70) instead of saving the ID value. Also, note that we don't need to call the `setValid()` method in the case of a button component.

The `decode()` methods we will implement for XForms-JSF components will be slightly different from the `decode()` method implementation above. The reason is that the code above works on the idea of name-value pairs in the request URL, while XForms-JSF components work on the idea of having XML markup in the request body. We will demonstrate the difference in XForms-JSF integration strategy on page80 .

The encoding methods

The second major functionality of the JSF component is encoding. In the encoding methods, the component developer writes the markup for the component rendered on the browser. The `UIComponent` has three types of encoding methods:

° `encodeBegin()`

- `encodeChildren()`
- `encodeEnd()`

The `encodeBegin()` method writes the starting tag markup for the JSF tag. The `encodeChildren()` method renders the markup for children of the JSF tag. If the children implement their own encoding, we don't need the `encodeChildren()` method. The `encodeEnd()` method encodes ending tag markup of the JSF tag.

The following points explain the calling sequence of the encode methods:

1. When the JSF framework comes across a JSF tag in the JSP page, it adds the component associated with the JSF tag in the component tree and calls the `encodeBegin()` method.

2. If the JSF tag whose `encodeBegin()` method is called contains any child JSF tag, the JSF framework first adds the associated JSF child component in the component tree and calls the `encodeChildren()` method of the parent component.

3. When the JSF framework comes across the ending JSF tag (whose `encodeBegin()` method is called above), it calls the `encodeEnd()` method of the associated JSF component.

Look at the following `encodeBegin()` method implementation, which encodes the `commandButton` component on the browser:

```
public void encodeBegin(FacesContext context) {
  if(context == null)
    throw new NullPointerException();
  if(!isRendered())
    return;
  ResponseWriter writer = context.getResponseWriter();
  String ref = (String) getAttributes().get("ref");
  //Gets remaining attribute values
  //Writes markup to render the commandButton component using
  //ResponseWriter.write() method
}//encodeBegin
```

Notice the following points:

- The `encodeBegin()` method first verifies the value of the `FacesContext` object passed to it. If it is `null`, it throws a `NullPointerException`.

- After verifying the context, it checks a flag by calling the `isRendered()` method. The `isRendered()` method checks the value of the rendered attribute that the JSP author may have provided in the tag declaration. If the JSP author specified `rendered="false"` in the tag declaration, the JSP author does not want to render this tag. In this case, `isRendered()` returns false and `encodeBegin()` returns without rendering the component.

- Next, it retrieves the `ResponseWriter` object from the `FacesContext` class by calling its `getResponseWriter()` method. The `ResponseWriter` object is used to write the markup in response to the user's request.

- Then it retrieves the values of the different attributes of the tag. It calls the `getAttributes()` method of the component, which returns a `Map` object. (Recall from Setting properties of a component on page 51, where we put the tag attribute values in the same `Map` object.) Now we call the `get()` method of this `Map` object, passing it the name of the attribute. The `get()` method returns the value of the attribute that we will finally use for the encoding.

- Finally, the `encodeBegin()` method writes the markup for the component by calling the `write()` method of the `ResponseWriter` object that we got in step 3. The `write()` method accepts a string, which it simply writes on the `ResponseWriter` object.

Rendering a component

The renderer classes are view-generating (or encoding) classes. We have seen that components can be self-rendering -- they can have encode methods that the JSF framework can call to let the component render itself. The JSF framework also allows you to develop separate renderers, letting you develop separate classes that handle the rendering of a component. It is up to you whether you want a separate renderer class are not.

Normally, you develop a renderer class for your component when you want to have different presentations of the same component.

To write your own renderer, you have to extend your class from a class named `Renderer`. The `Renderer` class is part of the JSF framework and is an abstract class, so you cannot instantiate it. You always extend this class.

Look at the following `Select1Renderer` class:

```
public class Select1Renderer extends Renderer{
  public Select1Renderer (){
  }
  public void decode(FacesContext context, UIComponent component){
  }
  public void encodeBegin(FacesContext context, UIComponent component)
   throws IOException {
  }
  public void encodeChildren(FacesContext context, UIComponent
component)
    throws IOException {
  }
  public void encodeEnd(FacesContext context, UIComponent component)
```

```
      throws IOException {
    }
  }//Select1Renderer
```

Notice that the renderer class contains the decoding and encoding methods. The working of the decoding and encoding methods in a renderer class is similar to the `UIComponentBase` class. The only difference is that the methods in the `Renderer` class have one extra parameter: the `UIComponent` object that represents the component associated with this renderer.

The `UIComponent` instance is passed to these functions because they may need to use some functionalities of the JSF component class. For example, the `decode()` method has to call the `getClientId()` method of the JSF component to check the component's ID.

Associating a tag with a renderer

The association of a renderer class with a specific JSF tag is also done in the JSF configuration (faces-config.xml) file, which means this is also an application-specific task:

```
<?xml version="1.0"?>
<faces-config>
  .......
  <render-kit>
    .......
    <renderer>
       <renderer-type>MyRendererType</renderer-type>

<renderer-class>demo.MyRendererClass.class</renderer-class>
    </renderer>
    <renderer>
       <renderer-type>MyAnotherRendererType</renderer-type>

<renderer-class>demo.MyAnotherRenderer.class</renderer-class>
    </renderer>
    .......
  </render-kit>
  .......
</faces-config>
```

Note the following points:

- The faces-config.xml file contains a `render-kit` element that wraps a number of `renderer` elements. The `renderer` element wraps the type of the renderer and renderer class against that type.

- Each `renderer` element contains `renderer-type` and `renderer-class` child elements.

- The `renderer-type` element specifies the type of the renderer.

- The `renderer-class` element wraps the name of the corresponding renderer class.

Here's how the JSF framework uses the faces-config file to associate a renderer with a JSF tag:

1. The JSF framework calls the `getRendererType()` method of a tag handler class, and the method call returns a string value that represents the type of the renderer in the configuration file. Let's suppose the `getRendererType()` returns the string "MyRendererType."

2. The JSF framework locates which `renderer-type` element in the faces-config.xml file wraps the string returned by the `getRendererType()` method of the tag handler class. Notice that the contents of the `renderer-type` child element of the first renderer element matches with the string returned by the `getRendererType()` method.

3. The JSF framework then returns the renderer class specified in the accompanying `renderer-class` element, which is `MyRenderereClass`.

The Validation process

In step 6 (Scenario II) of The URL processing life cycle in a JSF application on page 37, we said the JSF framework calls the `validate()` method to validate the user data. The JSF framework calls the `validate()` method of each component in the component tree after decoding request and handling the events occurred during decoding.

The `validate()` method compares the model bean property specified in the `value` attribute with the user's data that the `decode()` method stored after decoding the request. If the two values are different, the `validate()` method fires a value-change event. (See Handling value-change events on page 71 for details of value-change events.)

For now, look at the following sample `validate()` method implementation:

```
public void validate(FacesContext context){
  if(context == null)
     throw new NullPointerException();
  Object oldValue = getValue();
  Object newValue = getSubmittedValue();
  if(oldValue != null){
    if(oldValue.equals(newValue))
      return;
    else {
      setValue(newValue);
      queueEvent(new ValueChangeEvent(this, oldValue, newValue));
      return;
    }
```

```
    }//if(oldValue != null)
    else if(newValue != null) {
      setValue(newValue);
      queueEvent(new ValueChangeEvent(this, oldValue, newValue));
    }
  }//validate
```

This method performs the following steps to validate the user data:

1. It calls the `getValue()` method of the component, which returns the value of the model bean property specified in the `value` attribute.

2. Next, it calls the `getSubmittedValue()` method of the component, which returns the component value parsed by the `decode()` method from the request.

3. Finally, it compares the two values. If they are different, the `validate()` method fires a value-change event by calling a method named `queueEvent()`, which is discussed in detail in Event-generation mechanism in JSF on page70 .

The JSF framework also uses the `validate()` method to validate the format of the user's data. But in an XForms application, data validation is the responsibility of the browser. for XForms-JSF applications, we do not need to implement data validation in our components.

Updating model beans

The JSF framework provides component developers with an `updateModel()` method to update the model beans with the user's data. In step 8 of The URL processing life cycle in a JSF application on page 37, we explain that the JSF framework calls the `updateModel()` method to update the model bean with user data. The `updateModel()` method updates only the model bean property specified in the `value` attribute. The JSF framework calls the `updateModel()` method of each JSF component available in the component tree.

The `updateModel()` method calls the `setValue()` method of the `ValueBinding` class to set the model bean property with new data parsed by the `decode()` method from the JSF request.

For example, look at the following sample `updateModel()` method implementation:

```
  public void updateModel(FacesContext context){
    if(context == null)
      throw new NullPointerException();
    if(!isValid())
      return;
    ValueBinding vb = getValueBinding("value");
```

```
      try{
        vb.setValue(context, getLocalValue());
        return;
      }
      catch(Exception e){
        setValid(false);
        throw new FacesException(e);
      }
    }//updateModel
```

This method performs the following steps to update the model value:

1. The `updateModel()` method first calls the `isValid()` method to check whether the `decode()` method processed the request successfully. If this method returns false, the `updateModel()` method stops execution and does not update the model values. Recall from The decoding process on page 56 that we called the `setValid(true)` method at the end of successful decoding.

2. Next, the `updateModel()` method gets the `value` attribute value, which contains the `ValueBinding` object. Recall from Setting properties of a component on page 51 that we created the `ValueBinding` object for the `value` attribute in the `setProperties()` method of the `Select1RadioTag` class and passed it to the component class.

3. Finally, the `updateModel()` method calls the `setValue()` method of the `ValueBinding` object, passing it the context and new value along with the method call. This call to the `setValue()` method results in the model bean property being updated.

Summary

The JSP page includes the TLD file using the `taglib` directive of JSP technology. When the JSP container comes across a JSF tag, it looks for the tag library for that tag to instantiate its tag handler class. Each JSF tag has a tag handler class, which extends from the `UIComponentTag` class, which in turn implements the `Tag` interface. The `UIComponentTag` class has two abstract methods: `getComponentType()` and `getRendererType()`, which every tag handler class implements. The `getComponentType()` method returns a string. The JSF framework uses this string to locate the associated component class from the configuration file. The configuration file is used to associate the tag with a component class.

From the above discussion, we can see that there are two bridges used to complete a JSF application. One of them is the TLD file between the JSP and tag handler class; the second is the configuration file between the tag handler class and main JSF component behind the JSF tag.

The JSF components extend the `UIComponentBase` class, which implements the `UIComponent` abstract class. The component developers implement

customized functionality of their components.

We have now discussed four main functionalities of the component: decoding, encoding, validation, and updating model beans.

Section 5. Model beans and events in JSF

Developing model beans

The JSF framework allows application developers to write data model classes (also called model beans) to hold their application data. Developing model beans is an application-specific task and, therefore, part of the application development process.

This section explains the:

- Development of model beans
- Association of model beans with JSF components
- Updating properties of model beans
- Event-generation mechanism
- Event-handling mechanism

Let's start by explaining why we need model beans in our JSF applications. The simple bean acts as your data model, where any application can call the setter and getter methods to talk to the bean. You can design a simple model bean with public setter and getter methods for properties of the bean.

Let's look at a simple model bean:

```
public class ProductData{
  private Product product;
  private String model = null;
  private String action = null;
  public ProductData(){
  }
  public String getModel(){
  }
  public void setModel(String model){
  }
  public String getAction(){
  }
  public void setAction (String action){
  }
  public Product getProduct(){
  }
  public void setProduct(Product pData){
  }
  public void showCatalogView(ActionEvent ae){
  }
  public void showCartView(ActionEvent ae){
  }
}//ProductData
```

Notice the following points from the `ProductData` model bean class above:

- It has some properties to store application data, and simple setter and getter

methods for the model bean properties.

- The JSF component classes call the setter methods to store the application data in a model bean property. The JSP author passes a reference of this property in the `value` attribute of the associated JSF tag.

- The JSF framework calls the getter method of the model bean property to get application-specific data.

- The JSF framework uses the `ValueBinding` class to call the setter and getter methods of the model bean. Recall Setting properties of a component on page 51, where we created the `ValueBinding` objects for the model bean properties.

- In addition to the setter and getter methods, the model bean also contains a couple of event-handling methods: `showCatalogView()` and `showCartView()`. The JSP author passes the reference of these methods in the `valueChangeListener` or the `actionListener` attributes of the JSF tag.

- The JSF framework uses the `MethodBinding` class to call the event-handling methods of the `model` bean. Recall Setting properties of a component on page 51, where we created a `MethodBinding` object for the `valueChangeListener` property and passed it to the component class.

- Whenever the user interacts with a component that contains the `valueChangeListener` or `actionListener` attributes, the JSF framework calls the event-handling methods, passing the event object along with the method call. We will explain this event-handling capability of the model bean in detail in Handling value-change events on page71 .

Next, we'll discuss the association of model beans with a JSF component.

Associating model beans with JSF components

You can associate a specific property of a model bean with any JSF component during application development. To do this, provide a reference to the name of the bean and its property with the corresponding JSF tag. For example, look at the following JSF tag:

```
<jsf:selectOneRadio value="#{productData.product}">
```

The substring `#{` in the contents of the value attribute indicates that it is a reference value (that is, the reference to a model bean property or method). The name of the model bean and its property are wrapped inside the curly brackets.

The substring (inside the curly brackets) before the dot symbol in the value of the `value` attribute specifies the name of the model bean (`productData`). The string after the dot (`product`) is the name of the property to associate with the JSF component. The name of the bean should match the bean already present in the application.

The component classes interact with the model beans with the help of the `ValueBinding` and `MethodBinding` classes, as discussed in The ValueBinding class on page 48 and The MethodBinding class on page 50, respectively.

We can instantiate the model beans in `application`, `session`, `request`, or `page` scopes.

If we instantiate our model bean in `application` scope, the lifetime of our model bean will be equal to the lifetime of the servlet. We will instantiate a model bean in `application` scope if we want our model bean to preserve its state throughout application execution. Normally, we are not required to specify the scope of a model bean at `application` level. However, if a model bean is maintaining a log throughout the application's existence, we can declare that model bean in `application` scope.

If we specify the scope of our model bean as `session`, on each new session, the Web server creates a new instance of the model bean. We instantiate a model bean in `session` scope if we want to track the interactions of a user with our application in a session or want to process the actions and decide the next move. In most Web applications, application developers declare model beans in `session` scope.

If we specify the scope of our model bean as `request`, on each request, the Web server creates a new instance of the model bean. We instantiate a model bean in `request` scope if we want our bean to instantiate on each request with some new values.

There are a couple of ways to instantiate model beans. The first is by using the `useBean` directive of the JSP; the second is by declaring a bean as managed-bean in the application configuration file.

You should already be familiar with the first way of instantiating model beans. The second method is new with JSF technology, so it's the only one we'll discuss. The JSF framework introduces this managed-bean mechanism of instantiating a model bean.

To declare a model bean as managed-bean you have to make some entries in the faces-config.xml file. The faces-cofig.xml file contains an element named `managed-bean`, which wraps the following information:

- A small description about the model bean
- The name of the model bean
- The fully qualified name of the model bean class
- The scope at which the model bean is declared

Look at the following entry of `managed-bean` in the faces-config.xml file:

```
<?xml version="1.0"?>
<faces-config>
   ......
   <managed-bean>
      <description>Data Bean </description>
      <managed-bean-name>productData</managed-bean-name>
      <managed-bean-class>
          xforms_jsf.ProductData
      </managed-bean-class>
      <managed-bean-scope> session </managed-bean-scope>
      <managed-property>
         <property-name>action</property-name>
         <value>product</value>
      <managed-property>
   </managed-bean>
   ......
</faces-config>
```

Note the following points:

- The `managed-bean-name` element wraps the name of the instance we will use in the application to access the model bean.
- The `managed-bean-class` element specifies the fully qualified name of the model bean class.
- The `managed-bean-scope` element defines the scope of the model bean.
- The `managed-property` element is used to set the value of a property in the model bean when the model bean is first initialized. It contains `property-name` and `value` child elements.
- The `property-name` element contains the name of a property (action) of the model bean. The `value` element contains the actual value that the JSF framework will set in the property specified in the `property-name` element when it initializes the model bean. As you can guess, the JSF framework simply calls the setter method of the property, passing the value as the parameter.

Telling the application that some value has changed or some event has occurred

UI components, like input boxes, radio buttons, check boxes, can re-put changes in model objects. The JSF components tell the application that the user has changed a value. For example, a page can have radio buttons to select an option from a list of options.

Suppose the user has selected a value and submitted it to the server. When the JSF framework receives the request, it does the following to inform the application about the user's selection:

1. Decodes the request
2. Validates the values by comparing the old value and new value from the request; if the values are different, the `validate()` method tells the application by firing a value-change event

We have a detailed discussion of the decoding and validation processes in The decoding process on page 56 and The Validation process on page 62, respectively.

As a result of the event firing, the JSF framework invokes the application-specific event-handling logic. Through this mechanism, our application will learn what values have been changed by the user.

The JSF framework has a comprehensive event-generation and handling mechanism for this purpose, which we'll study in detail.

Event-generation mechanism in JSF

The following events are two types of events a JSF component can generate: value-change events and action events.

A value-change event occurs when the user changes a component value by selecting check boxes, selecting radio buttons, entering text in an input box, etc. The `validate()` method of a component fires value-change events when it finds that the value the user entered is different from the value fetched from the model bean.

Suppose, for example, that after adding a product to a shopping cart, the user wants to change a feature. The user again changes the value of the feature and submits it. On the server side, the component associated with this feature gets the old value from the model bean and compares it with the new value from the user. If the two values are different, it fires a value-change event, passing it the old and new values.

To fire the value-change event, we call the `queueEvent()` method of the component class. The `queueEvent()` method takes an instance of the event to be thrown. In this case, it is the `ValueChangeEvent` class instance. The `queueEvent()` method adds an event in the list of events that the JSF framework handles after the execution of the `updateModel()` method.

Look at the following:

```
    ValueChangeEvent vce = new ValueChangeEvent(this, oldValue,
  newValue);
```

```
queueEvent(vce);
```

In the first statement, we created an instance of the `ValueChangeEvent` class. The `ValueChangeEvent` constructor takes three parameters: a component object, the old value of the component (from the model bean), and the new value (fetched by the `decode()` method from the request).

In the second statement, we called the `queueEvent()` method of the component to fire the value-change event. The `queueEvent()` method takes the `ValueChangeEvent` object (which we created in the first statement) along with the method call.

The execution of these two statements results in firing a new value-change event.

The action event occurs when the user clicks a button or a hyperlink. The action events are fired in the `decode()` method of a component (refer to The decoding process on page 56).

To fire an action event, we call the `queueEvent()` method of the component class. As mentioned above, the `queueEvent()` method takes an instance of the event to be thrown along with the method call. In this case, it is an `ActionEvent` instance.

Look at the following, which is quite similar to the value-change event firing code discussed above:

```
ActionEvent ae = new ActionEvent(this);
queueEvent(ae);
```

In the next couple of sections, we will discuss handling the value-change and action events.

Handling value-change events

Handling events is an application-specific task, so the application developer implements the listeners to handle the value-change events.

The JSF framework provides application developers with two ways to handle the value-change events: implement a value-change event-handling method in the model bean or implement a class, which the JSF framework invokes when a value-change event is fired.

First, we'll demonstrate how to implement a value-change event-handling method in the model beans.

The value-change event-handling method is a simple public method with a return type of `void`. This method should take an instance of the

`ValueChangeEvent` class along with the method call.

The `ValueChangeEvent` class has the record of old and new values of the component. Recall from Event-generation mechanism in JSF on page 70 that while firing the event, we passed both these values in the `ValueChangeEvent` constructor. You can access old and new values of the component by calling the `ValueChangeEvent` class methods named `getOldValue()` and `getNewValue()` respectively.

For example, look at the following sample method:

```
public void valueChanged(ValueChangeEvent vce){
  // Application specific processing goes here......
  .......
  Object  oldValue =  ev.getOldValue();
  Object  newValue =  ev.getNewValue();
  UIComponent component = ev.getComponent();
  // Application specific processing goes here......
  .......
}//valueChanged
```

The `getOldValue()` and `getNewValue()` methods return Object values. Cast these values to application-specific classes. The `getComponent()` method returns the instance of the component that fired this value-change event.

The JSP author passes the name of the value-change event-handling method in the `valueChangeListener` attribute of the JSF tag. We explained the method binding for the `valueChangeListener` property in Setting properties of a component on page 51. When a value-change event is fired by the `validate()` method, the JSF framework uses the `MethodBinding` object to call the `invoke()` method, which in turn calls the model bean method specified in the `valueChangeListener` attribute.

Next, we will discuss the second way to handle value-change events: implementing a value-change event-handling class.

To handle the value-change event in this way, you have to define a class that should implement the `ValueChangeListener` interface. The `ValueChangeListener` interface contains a method named `processValueChange()`. The JSF framework calls this method to handle a value-change event.

The `processValueChange()` method takes a `ValueChangeEvent` object along with the method call. For example, look at the following `DemoValueChangeListener` class, which implements `ValueChangeListener` to catch any value-change event:

```
public class DemoValueChangeListener implements ValueChangeListener{
   public DemoValueChangeListener  (){
   }
   public void processValueChange(ValueChangeEvent ev){
      // Application specific processing goes here......
```

```
        .......
        Object  oldValue =  ev.getOldValue();
        Object  newValue =  ev.getNewValue();
        UIComponent component = ev.getComponent();
        // Application specific processing goes here...
        .......
    }//processValueChange
}//DemoValueChangeListener
```

The JSF framework provides JSF tags that you can use to register value-change handler classes with the components. The tags are:

```
<f:valueChangeListener type="xcart.DemoValueChangeListener"/>
```

The JSP author uses the `valueChangeListener` tag to register a value-change event handler class with a particular component. The JSP author includes the `valueChangeListener` tag as a child of the JSF tag with which he wants to register a `ValueChangeListener` tag. The `type` attribute of this tag specifies the fully qualified name of the value-change handling class.

For example, look at the following tag definition in a JSP page:

```
<jsf:selectOneRadio label="Select One : "
  value="#{productData.selectedFeature}">
  <f:valueChangeListener type="xcart.DemoValueChangeListener"/>
</jsf:selectOneRadio>
```

Handling value-change events vs. updating model beans

Handling the value-change events is an application-specific task. If our application requirement is such that we do not need to process data coming from the user, we do not need to write an event handler. In this case, the JSF framework simply calls the `updateModel()` method of the component to update the model bean. On the other hand, if our application requirement is such that we need to process new data coming from the user, we can process it in the event handlers before updating the model bean.

Application developers can achieve an important task during the handling of a value-change event. For example, if the user sends some invalid data, you can stop the component from updating the model bean with new data. For this purpose, call the `setValid(false)` method of the component class, which stops the component from updating the model bean:

```
public void processValueChange(ValueChangeEvent ev){
  // Application specific processing goes here......
  .......
  UIComponent component = ev.getComponent();
  If(/*some condition to check if user's data is invalid*/)
```

```
    Component.setValid(false);
  // Application specific processing goes here......
  .......
 }//processValueChange
```

Handling action events

As we have seen in Event-generation mechanism in JSF on page 70, the component fires an action event when the user clicks a button or hyperlink. To handle the action event, the JSF framework provides two mechanisms (just like the two mechanisms to handle the value-change events): the implementation of an action event-handling method in the model bean and the implementation of a separate action event-handling class.

The action event-handling method is like the one we implemented for the value-change event previously, except that the action event-handling method takes an instance of the `ActionEvent` class, rather than the `ValueChangeEvent` object.

The `ActionEvent` class has the information about the component class that fired the action event. Recall from Event-generation mechanism in JSF on page 70, while firing the event, we passed the component object to the `ValueChangeEvent` constructor.

For example, look at the following sample method in the model bean:

```
public void actionPerformed(ActionEvent ae){
  UIComponent uic = ae.getComponent();
  // Application-specific code goes here
}//actionPerformed
```

The `actionPerformed()` method above calls the `getComponent()` method of the `ActionEvent` class. The `getComponent()` method simply returns an instance of the component class that fired this action event. You will implement your application-specific processing after getting the component instance. For example, you will call some getter method of the component class to learn what action was performed.

The JSP author passes the model bean's action event-handling method name in the `actionListener` attribute of the JSF tag. When a user clicks a button or hyperlink in the JSP page, the `decode()` method of a component (button or hyperlink) fires an action event.

When the JSF framework finds that the `decode()` method of a component has fired an action event, it checks the value of the `immediate` property of the component that fired the event. If this `immediate` property value is true, it uses the `MethodBinding` object to call the `invoke()` method, which in turn calls the model bean method specified in the `actionListener` attribute. If the `immediate` property value is false, the JSF framework calls the action

event-handling method after the call to the `updateModel()` method of the component.

The default value of the `immediate` property is false. The JSP author specifies the value of the `immediate` property in the `immediate` attribute:

```
<xforms-jsf:commandButton actionListener="dataStore.actionPerformed"
  immediate="true">
</xforms-jsf:commandButton>
```

Next, we will discuss the second way to handle value-change events: implementing a value-change event-handling class.

To handle action through the second method, you have to define a class that implements an interface named `ActionListener`. The `ActionListener` interface contains a method named `processAction()`. The JSF framework calls this method to process the action event. The action event-handling class must implement at the `processAction()` method.

The JSF framework calls the `processAction()` method, passing it the event object (fired by the component) along with the method call.

For example, look at the following `ActionListener`:

```
public class DemoActionListener implements ActionListener{
   public void processAction(ActionEvent event){
      UIComponent uic = ae.getComponent();
   }
}
```

The code given for the `processAction()` method is similar to the code given for the `actionPerformed()` method already discussed. The only difference is in the use of the two event-handling mechanisms.

We use the `actionListener` tag from the JSF core tag library to register an action event handler class with the button tag. For example, look at the following `commandButton` tag:

```
<jsf:command_button label="Submit" >
  <f:actionListener  type="xforms_jsf.DemoActionListener"/>
</jsf:command_button>
```

We have included an `f:actionListener` child tag to the `commandButton` tag. The `type` attribute of the `f:actionListener` tag specifies the name of the action event-handling class.

Navigation process

The JSF framework has defined a mechanism for navigation in a JSF application. The application developer has to write some navigation rules in the configuration file (faces-config.xml). The following faces-config.xml file shows the entry for a navigation rule:

```
<?xml version="1.0"?>
<faces-config>
  .....
  <navigation-rule>
    <from-view-id>/catalogView.jsp</from-view-id>
    <navigation-case>
        <from-outcome>category</from-outcome>
        <to-view-id>/catalogView.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>product</from-outcome>
        <to-view-id>/productView.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  .....
</faces-config>
```

Notice the following points from the above code:

- The faces-config.xml file contains a number of `navigation-rule` elements, where each navigation-rule element defines a single navigation rule from one JSP page to another. Every navigation rule should define three things: the initial JSP page, the action that occurs in the initial JSP page, and the final JSP page to be loaded as a result of the action. Therefore, every navigation rule specifies these three bits of information.

- In the above faces-config.xml file, the `navigation-rule` element contains a `from-view-id` element, which wraps the name of the initial JSP page.

- The `navigation-rule` element also contains two `navigation-case` child elements.

- The first `navigation-case` element in turn contains a `from-outcome` and `to-view-id` child element.

- The `from-outcome` element specifies the action that may occur in the initial JSP page, while the `to-view-id` element wraps the name of the final JSP page that will be invoked as a result of the action specified in the `from-outcome` element.

- Now look at the second `navigation-case` element, which contains different combinations of action and final JSP.

- You can have any number of `navigation-case` elements within a `navigation-rule` element.

We have seen how the navigation rules are defined in the faces-config.xml file. Let's now look at how the JSF framework uses the navigation rule to navigate to a page.

After updating the model beans, the JSF framework enters the invoke application phase, where it performs the following steps for each JSF tag that has an action attribute (such as the `xcart:category` tag in the `catalogView.jsp` page):

1. It calls the model bean method specified in the `action` attribute method call and returns a string value.

2. The JSF framework matches this string value with the contents of each `from-outcome` element in the `navigation-rule` element defined for the current JSP page.

3. When the match is found, the JSF framework invokes the page specified in the `to-view-id` element, which results in automatic invocation of the final JSP page.

After going through the navigation rules, if the JSF framework finds that the same currently loaded page needs to be invoked, nothing special happens and the JSF framework simply calls the encoding methods of all the components in the current component tree.

If the JSF framework finds that a new JSP page needs to be invoked, it goes back to the beginning with the new JSP page.

Life-cycle processing phases of a JSF application

We have learned so many things that happen in a JSF application. Before we start using these concepts to build our XForms-JSF library, however, it is better to summarize our discussion in the form of life-cycle processing phases of a JSF application, as follows.

A JSF application request goes through six phases. Don't worry about the management of these phases. They are managed by the JSF framework.

The `service()` method of the `FacesServlet` class invokes the JSF framework, which checks whether the root component exists. If not, it creates the root component, adds (*stores*) it in the `FacesContext`, and directly goes to the last phase (the Render response phase). If the root component is found, the JSF framework adds (*restores*) the root component, along with all its children in the `FacesContext`, and goes to the next phase (the Apply request values phase). This storing or restoring of the root component in the `FacesContext` is called the *Restore view phase*.

After restoring the components tree, JSF framework calls the `decode()`

method of each component in the component tree. The `decode()` method decodes the client's request according to the component behavior. The `decode()` method can also add action events (for clicking buttons or hyperlinks) for later processing. This phase is called the *Apply request value phase*.

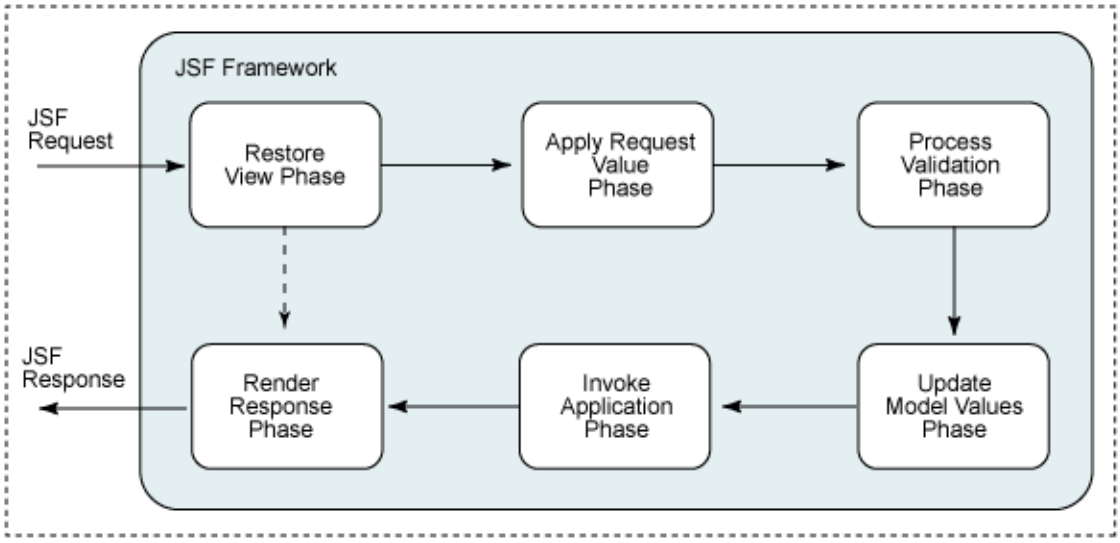
After decoding the request, the JSF framework calls the `validate()` method of each component in the component tree. The `validate()` method verifies the changes made by the user in component values. If a change in component values is found, it fires a value-change event. This validation of data is called the *Process validation phase*.

After validating the user changes in the component, the JSF framework calls the `updateModel()` method of each component in the tree. The `updateModel()` method updates the model bean with the user's new data. This phase is called the *Update model values phase*.

Next, the JSF framework checks the navigation rules to determine whether it needs to invoke another JSP page. If the invocation of new JSP is not required, the component tree, which is currently loaded in the `FacesContext`, remains as it is. Otherwise, a new component tree is created (goes back to the first phase). This phase is called the *Invoke application phase*.

Finally, the JSF framework checks whether the complete component tree of the JSP page that the user requested exists. If it does not, the JSF framework goes through the entire JSP page. When it comes across a JSF tag, it adds its component class as a child of the root component we created in the first phase (forming a component tree). Then it calls the encoding methods of the component. If the component tree exists, the JSF framework simply calls the encoding methods of all the components in the component tree. This phase is called the *Render response phase*.

The phases are shown below:



Summary

Application developers write model beans to hold application-specific data. The component classes call the setter and getter method of the model bean properties to get or store application-specific data.

We can bind a specific property of a model bean or a specific method of a model bean to a component.

A JSF component can fire two types of events: value-change events and action events. The JSF component fires a value-change event when the user changes component values. The JSF component fires an action event when the user clicks a button or clicks a hyperlink component on the page.

Application developers implement event-handling logic in event-handling methods. The JSF framework internally invokes the event-handling methods at the appropriate time to process the event.

Section 6. XForms-JSF integration strategy

XForms-JSF integration requirements

We have explained the server-side programming requirements of an XForms application. We have also demonstrated how the JSF architecture works. We are now ready to demonstrate how we can use JSF to fulfill the server-side requirements of XForms applications.

In this section, we are presenting the strategy for XForms-JSF integration. For this purpose, we are going to develop an XForms-JSF sample application. This section demonstrates the basic integration strategy by developing three components:

- The `xforms-jsf:selectOneRadio` component provides users with a list of choices (a radio button with each choice) to select one of them. It renders the markup for the XForms `select1` element.

- The `xforms-jsf:commandButton` component renders the markup for the XForms `submit` element.

- The `xforms-jsf:model` component renders the markup for the XForms `model` element.

The last subsection provides a small JSP page to demonstrate the use of the three XForms-JSF tags.

In the next section (XForms-JSF tag library on page 118), we will use the same integration strategy to implement the rest of the tags necessary for XForms-JSF integration.

We would like to point out that an XForms-JSF application is like any other JSF application, except that it authors and processes XForms markup, which means that XML is used as the format for data interchange between an XForms browser and a server. A JavaBean component sitting on the server side parses the XML data from the browser and loads it into a Document Object Model (DOM) object. The XForms-JSF application uses this `DOM` object to retrieve data coming from the client.

While trying to fit XForms into JSF, we will try to fulfill the following requirements:

1. We have to design a JSF tag library that works according to the JSF framework discussed in the past three sections.

2. The names of the JSF tags should be the same as the names of corresponding tags in the HTML tag library. While developing a JSF tag, if we can find a similar tag in the JSF's HTML tag library, we will use the same

name. If we cannot find a parallel tag in the JSF's HTML tag library, we will use a different name, which will help in porting existing HTML applications to XForms.

3. XForms-JSF tags should be able to render XForms markup. For example, the `selectOneRadio` tag of the JSF's HTML tag library generates the following markup:

```
<table border="0">
  <tr>
    <td><input type="radio" name="r1"
value="Red">Red</td>
    <td><input type="radio" name="r1"
value="Blue">White</td>
  </tr>
</table>
```

   Our XForms-JSF tag library also contains a tag named `selectOneRadio`. When `selectOneRadio` is used in an XForms-JSF application, it generates the following markup:

```
<xforms:select1 xmlns:xforms="http://www.w3.org/2002/xforms"
  model="optModel" appearance="full" ref="selectedColor" >
  <xforms:label>Choose a color:</xforms:label>
  <xforms:item>
    <xforms:label>Red</xforms:label>
    <xforms:value>Red</xforms:value>
  </xforms:item>
  <xforms:item>
    <xforms:label>White</xforms:label>
    <xforms:value>White</xforms:value>
  </xforms:item>
</xforms:select1>
```

   From the HTML and XForms markup above, the `selectOneRadio` tags in the JSF's HTML tag library and our XForms-JSF tag library are different from each other, and that our XForms-JSF tags should be able to render XForms markup.

4. XForms markup contains some attributes specific to XForms and do not have a parallel in HTML, so the XForms-JSF tag library has to handle such attributes.

   As an example, let's look at the JSF tags used in both cases. Here's the JSF tag used to generate HTML markup:

```
<f:selectOneRadio value="#{dataStore.selectedColor}">
   < f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectOneRadio>
```

   Here's the JSF tag used to generate XForms markup:

```
<f:selectOneRadio value="#{dataStore.selectedColor}"
   model="optModel" ref=" selectedColor" label="Choose a Color:">
   <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectOneRadio>
```

You can observe from the JSF code segments above that the difference between the HTML and XForms tags is that the `xforms-jsf:selectOneRadio` tag contains two extra attributes named `ref` and `model`. In The select element on page 18, we explained the purpose of these two attributes in detail. The XForms-JSF tags we are going to develop will render XForms markup containing XForms-specific attributes.

5. The XForms-JSF tag library should be able to handle the way an XForms browser submits data to a Web server, which is quite different from the way an HTML browser submits data to a Web server. XForms submit XML data, while HTML forms submit data in the form of name-value pairs. This means our XForms-JSF tag library needs to author and process XML data. We will implement the XML authoring and processing logic in the XForms-JSF tag library so an XForms-JSF application developer does not have to worry about low-level details.

6. We will design the XForms-JSF tag library to be independent of the existing JSF's HTML tag library. You can use either or both in a JSF application, which means that XForms and HTML tags should be able to co-exist in one application.

Integration steps

We are going to list the programmatic steps we need to follow to implement the XForms-JSF tag library. Some steps are required to be implemented just once for the entire tag library, while some will be done separately for each tag in the library.

The following steps are common for the entire library, so we will need to implement them just once for the library:

1. Develop a JSF tag and its accompanying JSF component, which renders the application-specific XML data wrapped inside the XForms `model` element. We will call this JSF tag an `xforms-jsf:model` tag. This tag handles all aspects of rendering an XForms `model` element in an XForms page. It also allows an application to supply application-specific XML data to the `xforms-jsf:model` tag, so that the `xforms-jsf:model` component can render the same on the XForms page.

   There is no equivalent of this in the present JSF framework, so we will build the `xforms-jsf:model` tag from scratch.

2. Write a JavaBean component that parses an incoming XForms request and author a `DOM` object that wraps the XML instance data from an XForms browser. The name of this JavaBean will be

`IncomingXMLInstanceRequest`.

3. Create a TLD file for the library.

4. Write a faces-config.xml file for the tag library.

Complete the following steps for each tag in the library:

1. Create a tag handler class for each tag in the XForms-JSF tag library.

2. Make entries in the TLD file for each tag in the XForms-JSF tag library.

3. Make entries in the faces-config.xml file for each component in the tag library.

4. Write the `decode()` method for each component that parses the incoming XML instance data.

5. Write the `encodeBegin()` and `encodeEnd()` methods of the components that render XForms markup.

The rest of this section elaborates on each step listed above.

Implementing the xforms-jsf:model component

The `xforms-jsf:model` component renders the markup for the XForms `model` element. There is no equivalent of the `xforms-jsf:model` tag in the present JSF framework, so this `xforms-jsf:model` tag needs to be developed from scratch.

The purpose of the `xforms-jsf:model` tag is to author an XForms `model` element that contains an XForms `instance` child element. The XForms `instance` element in turn wraps the application-specific XML data. In The model element on page 16, we discussed the XForms `model` element in detail.

For example, if the following code is the application-specific XML data:

```
<modelXml>
  <selectedColor></selectedColor>
</modelXml>
```

Then the `xforms-jsf:model` component would author the following markup:

```
<xforms:model id="myModel">
  <xforms:submission action="/xforms-
    jsf/faces/Demo.jsp;jsessionid=B30A73E86F22A6BB68A36B348CF97D6B"
    method="post" id="submit" />
  <xforms:instance>
    <modelXml>
      <action-performed></action-performed>
      <selectedColor></selectedColor>
    </modelXml>
  </xforms:instance>
</xforms:model>
```

Compare the application-specific XML data with the markup that the `xforms-jsf:model` component delivers, and you will find the following:

- The application-specific XML is wrapped in the `instance` element of the XForms `model` element.

- The application-specific XML generated by the `xforms-jsf:model` tag contains an extra tag named `action-performed`.

- This modification in the application-specific XML is required to track the user interaction in the page. We are required to insert an extra tag because if your application's page contains more than one button or hyperlink, components cannot verify which button or hyperlink the user clicked.

- This `action-performed` tag wraps the ID of the button or hyperlink component clicked by the user on the page. The `decode()` methods of the component classes extract the ID of the button from the `action-performed` tag to identify the component (button or hyperlink) the user clicked.

The JSP author provides this component with a `value` attribute, whose value points to a String type property of the model bean. For example, look at the following use of the `xforms-jsf:model` tag:

```
<xforms-jsf:model value="#{dataStore.xformsModelData}"/>
```

The `xforms-jsf:model` component implementation calls the getter method of the `dataStore.xformsModelData` property, which returns the application-specific XML. The component class associated with the `xforms-jsf:model` tag wraps the application-specific XML inside the XForms `model` and `instance` elements.

The next subsection explains how to develop the model bean that provides the application-specific XML model data. But first, let's see how to implement the `xforms-jsf:model` tag.

The first step toward the development of a component is to make an entry in the TLD file. The TLD file entry for `xforms-jsf:model` looks like the following code:

```
<tag>
  <name>model</name>
  <tag-class>xforms_jsf.ModelTag</tag-class>
  <body-content>JSP</body-content>
  <!-- JSF attributes -->
  <attribute>
      <name>value</name>
      <required>true</required>
      <rtexprvalue> false </rtexprvalue>
  </attribute>
  <!-- XForms Attributes -->
  <attribute>
      <name>id</name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
      <name>schema</name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
      <name>functions</name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
  </attribute>
</tag>
```

In Associating JSF tags with tag classes on page 45, we explained the TLD file in detail. You can refer to this for the details of the above TLD file.

In The UIComponentTag class on page 43, we explained that each JSF tag has a tag class associated with it, whose name we have to wrap in the `tag-class` element in the TLD file. Let's see the implementation of the `xforms_jsf.ModelTag` class mentioned in the `tag-class` element in the above TLD file entry:

```
public class ModelTag extends UIComponentTag{
   private String id        = null;
   private String schema    = null;
   private String value     = null;
   private String functions = null;
   public String getRendererType() {
      return null;
   }
   public String getComponentType() {
      return "Model";
   }
   public String getValue() {
      return value;
   }
   public void setId(String id) {
      this.id = id;
   }
   public void setSchema(String id) {
      this.id = id;
   }
   public void setFunctions(String id) {
```

```
        this.id = id;
    }
    public void setValue(String value) {
        this.value = value;
    }
    public void setProperties(UIComponent component){
        super.setProperties(component);
        UIModel uim = (UIModel) component;
        FacesContext fc = FacesContext.getCurrentInstance();
        Application app = fc.getApplication();
        if(id != null) {
          if(UIComponentTag.isValueReference(id)) {
            ValueBinding vb = app.createValueBinding(id);
            uim.setValueBinding("xid", vb);
          }
          else{
            uim.getAttributes().put("xid", id);
          }
        }
        if(schema != null){
            uim.getAttributes().put("schema", schema);
        }
        if(functions != null){
            uim.getAttributes().put("functions", functions);
        }
        if(value != null){
          if(UIComponentTag.isValueReference(value)) {
            ValueBinding vb = app.createValueBinding(value);
            uim.setValueBinding("value", vb);
          }
          else{
            uim.setValue(value);
          }
      }
    }//setProperties
}//ModelTag
```

The `ModelTag` class above contains the `getComponentType()`, `getRendererType()`, and `setProperties()` methods, in addition to the simple setter and getter methods for the properties. We have already covered the details of these methods. You can refer to The UIComponentTag class on page 43 for the details of the `getComponentType()` and `getRendererType()` methods. Refer to Setting properties of a component on page 51 for the `setProperties()` method.

Here's how to implement the `xforms-jsf:model` component class. Look at the following entry in the faces-config.xml file:

```
<?xml version="1.0"?>
<faces-config>
    .....
    <component>
        <component-type>Model</component-type>
        <component-class>xforms_jsf.UIModel</component-class>
    </component>
    <!-- other component instances-->
</faces-config>
```

The `component-type` element wraps the type of the component. The value

returned by the `getComponentType()` method of `ModelTag` -- "Model" -- is matched with the content wrapped in the `component-type` element. The `component-class` element specifies the fully qualified name of the component class: `xforms_jsf.UIModel`.

Now we will implement the component class `UIModel`.

The following code shows the properties and methods of the `UIModel` class:

```
public class UIModel extends UIComponentBase{
  private String value = null;
  public void encodeBegin(FacesContext fc)
    throws IOException{
  }
  public void setValue(String value){
  }
  public Object getValue(){
  }
  private String insertElement(String xmlData){
  }
  private String getContextPath(FacesContext fc){
  }
}//UIModel
```

The `UIModel` class contains only one property (`value`), two public methods (`setValue()` and `encodeBegin()`), and two private methods (`insertElement()` and `getContextPath()`).

The `UIModel` component does not need to implement the `decode()` method because the `UIModel` is not required to parse the incoming request from the user. It only renders the markup for the XForms `model` element. That is why `UIModel` only overrides the `encodeBegin()` method of `UIComponentBase`.

Let's discuss the methods of the `UIModel` class one by one.

The `setValue()` method is used to set the `value` property of the component. We called this method in the `setProperties()` method of the `ModelTag` class to set the value of the `value` attribute passed by the JSP author:

```
public void setValue(String value)  {
  this.value = value;
}
```

The `insertElement()` method of `UIModel` inserts the `action-performed` tag in the application-specific XML (we have already explained the purpose of this extra tag in the application-specific XML):

```
private String insertElement(String xmlData){
  if(xmlData == null) {
    xmlData = "<dataWraper><action-performed>
               </action-performed></dataWraper>";
  }
  else{
```

```
      String temp = xmlData;
      int index = xmlData.indexOf(">");
      xmlData = xmlData.substring(0,index+1);
      temp = temp.substring(index+1);
      xmlData += "<action-performed></action-performed>";
      xmlData += temp;
    }
    return xmlData;
  }//insertElement
```

The `getContextPath()` method returns a string value that is used in the `action` attribute of the XForms `submission` element. For details about the XForms `action` attribute, see The model element on page 16.

```
  private String getContextPath(FacesContext fc){
    String viewId = fc.getViewRoot().getViewId();
    Application app = fc.getApplication();
    String temp = app.getViewHandler().getActionURL(fc, viewId);
    return fc.getExternalContext().encodeActionURL(temp);
  }//getContextPath
```

Now let's discuss the implementation of the `encodeBegin()` method:

```
  public void encodeBegin(FacesContext fc)
    throws IOException{
    if(fc == null)
      throw new NullPointerException();
    if(!isRendered())
      return;
    ResponseWriter out = fc.getResponseWriter();
    String id        = (String) getAttributes().get("xid");
    String schema    = (String) getAttributes().get("schema");
    String functions = (String) getAttributes().get("functions");
    String xmlData   = null;
    ValueBinding vb = getValueBinding("value");
    if (vb != null)
      xmlData = (String) vb.getValue(fc);
    //inserting action-performed
    else
      xmlData = getValue();
    xmlData = insertElement(xmlData);
    //retrieving path from external context
    String contextPath = getContextPath(fc);
    //writing model data
    out.write("<xforms:model");
    if(id != null)
      out.write(" id=\""+ id + "\"");
    if(schema != null)
      out.write(" schema=\""+ schema + "\"");
    if(functions != null)
      out.write(" functions=\""+ functions + "\"");
    out.write(" xmlns:xforms=\"http://www.w3.org/2002/xforms\">");
    out.write("<xforms:submission action=\"");
    out.write(contextPath);
    out.write("\" ");
    out.write("method=\"post\" id=\"submit\" />");
    out.write("<xforms:instance>");
    out.write(xmlData);
    out.write("</xforms:instance>");
```

```
    out.write("</xforms:model>");
  }//encodeBegin
```

The `encodeBegin()` method:

1. First checks the value of the `rendered` attribute of the JSF tag. If it is false, then it will not render the component. Instead it returns without rendering the component.

2. Retrieves the `ResponseWriter` object from the `FacesContext` object. All encoding methods use this `ResponseWriter` object to write their markup.

3. Retrieves the attribute values (set by the `setProperties()` method of `ModelTag`) by calling the `get()` method and passing it the name of the attribute.

4. Uses the `ValueBinding` object passed by the `ModelTag` class to fetch the application-specific XML from the model bean.

5. Inserts an `action-performed` element in the application-specific XML (to identify the button that the user clicked) by calling the `insertElement()` method.

6. Writes the markup for the component by calling the `write()` method of the `ResponseWriter` object.

The `UIModel` class is now ready to be used in a JSF application to render the XForms `model` element. In the last subsection of this section, we will demonstrate the usage of the `UIModel` component in a sample XForms-JSF application.

Writing the model bean that represents application-specific XML data

The model bean that authors application-specific XML data is like any other application-specific bean that you may need and develop in a server-side Java application. The only special thing that our `xforms-jsf:model` tag demands from the model bean is that the bean should have a string-type property. The `xforms-jsf:model` tag assumes that the string-type property contains the application-specific XML data it needs to wrap inside the XForms `instance` element. Refer to The model element on page 16 for details of the application-specific XML data.

The JSP author passes the reference of the string property as the value of the `value` attribute of the `xforms-jsf:model` tag. The component class (`UIModel`) uses the `value` attribute to get application-specific data.

For example, look at the following model bean named `DataStore`, whose `xformsModelData` property is a good candidate to act as the value attribute of an `xforms-jsf:model` tag:

```
public class DataStore{
   private String action          = null;
   private ArrayList colorsList   = null;
   private String selectedColor   = null;
   private String xformsModelData = null;
   public DataStore(){
      String [] colors = new String[2];
      colors [0] = "Red";
      colors [1] = "Green";
      selectedColor = null;
      colorsList = new ArrayList(colors.length);
      for (int i = 0 ; i < colors.length ; i++)
        colorsList.add(new SelectItem(colors[i], colors[i],
          colors[i]));
      xformsModelData =

"<modelXml><selectedColor></selectedColor></modelXml>";
   }
   public void setSelectedColor(Object selColor) {
      selectedColor = (String) selColor;
   }
   public Object getSelectedColor(){
      return selectedColor;
   }
   public void setColorsList (Collection colorsList) {
      this.colorsList = new ArrayList();
   }
   public Collection getColorsList(){
      return colorsList;
   }
   public String getXformsModelData (){
      return xformsModelData;
   }
   public void setXformsModelData (String xformsModelData) {
      this.xformsModelData = xformsModelData;
   }
   public void setAction(String newAction){
      this.action = newAction;
   }
   public String getAction(){
      return action;
   }
   public void btnPressed(ActionEvent event) {
     if(this.selectedColor.equals("Red"))
       setAction("redSelected");
     else if(this.selectedColor.equals("Green"))
       setAction("greenSelected");
   }//btnPressed
}//DataStore
```

Notice the following points from the `DataStore` class implementation above:

- The `xformsModelData` property of the `DataStore` class wraps the application-specific XML, which the JSP author can pass to an `xforms-jsf:model` tag in the `value` attribute.

- We hard-coded the application-specific XML for the sake of simplicity. While developing your application-specific bean, you can read this from a file or database. These days, nearly all DB vendors provide XML interfaces. The only requirement is that your bean should have a property that represents the application-specific XML.

- The `DataStore` bean contains a few more properties (`colorsList`, `selectedColor`, and `action`) along with their setter and getter methods. The other JSF tags in the JSP page use these properties to pass on application-specific data to the components.

- The public method named `btnPressed()` is actually an action event handler method, which we will use for action event-handling in our sample application.

- Similarly, the public method named `valueChanged()` is actually a value-change event handler method, which we will use for value-change event-handling in our sample application.

Parsing the incoming XML instance data

The XML that the `xforms-jsf:model` tag renders serves to wrap the user's data he wants to submit back to the server. The XML rendered by the `xforms-jsf:model` tag wraps the user's changes and carries the data to the server. Our server-side application needs to parse the XML received from the user to extract the user data.

We'll explain this concept with the following example. Suppose the request body contains the following XML:

```
<modelXML>
   <action-performed>id1</action-performed>
   <selectedColor>Red</selectedColor>
</modelXML>
```

In the XML, the `modelXML` tag carries information in two tags: `action-performed` and `selectedColor`. The `action-performed` tag wraps the ID of the button that the user clicked, and the `selectedColor` tag wraps the color name that the user selected. The `decode()` method of each XForms-specific JSF component has to parse the XML request to fetch the information sent by the user.

To make this work independently from component development, we will implement a class that parses the incoming XForms request and author a `DOM Document` object for the incoming XML. The name of this JavaBean component is `IncomingXMLInstanceRequest`.

The following code shows the properties and methods in the `IncomingXMLInstanceRequest` JavaBean component:

```
public class IncomingXMLInstanceRequest{
   protected Document DOMDocument;
   public IncomingXMLInstanceRequest(){
   }
   public Document getDOMDocument()
    throws org.xml.sax.SAXException,
    javax.xml.parsers.ParserConfigurationException{
   }
}//IncomingXMLInstanceRequest
```

The `IncomingXMLInstanceRequest` class contains a constructor, single property (`DOMDocument`), and public method `getDOMDocument()`.

The `DOMDocument` property contains the contents of the incoming client request. The `decode()` method of each component in the XForms-JSF tag library fetches the value of this property to learn its contents.

Now let's discuss the methods of the `IncomingXMLInstanceRequest` class one by one.

The `IncomingXMLInstanceRequest` constructor simply initializes the `DOMDocument` property with `null`:

```
public IncomingXMLInstanceRequest(){
  DOMDocument = null;
}
```

The `getDOMDocument()` method returns an instance of the `DOM Document` object, which contains the application-specific XML request.

```
public Document getDOMDocument()
  throws org.xml.sax.SAXException,
  javax.xml.parsers.ParserConfigurationException
{
  if(DOMDocument == null){
     FacesContext facesContext = FacesContext.getCurrentInstance();
     ExternalContext eContext = facesContext.getExternalContext();
     ServletRequest request= (ServletRequest)eContext.getRequest();
     ServletInputStream sis = request.getInputStream()
     try{
        DocumentBuilderFactory dbf =
                   DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        DOMDocument    = db.parse();
        return DOMDocument;
     }
     catch(java.io.IOException e){
        e.printStackTrace();
     }
  }//if(DOMDocument == null)
  return DOMDocument;
}//getDOMDocument
```

When the `getDOMDocument()` method is called by any component, it first checks the value of the `DOMDocument` property. If the value is `null`, it does the following steps:

1. It creates an instance of the `FacesContext` object.

2. It calls the `getExternalContext()` method of the `FacesContext` object, which returns an instance of the `ExternalContext` class. This `ExternalContext` object is necessary to fetch the request.

3. It calls the `getRequest()` method of the `ExternalContext` class, which returns an instance of the `ServletRequest` class. This `ServletRequest` object wraps all the details about the request from the client.

4. Then it calls the `getInputStream()` method of the `ServletRequest` object, which returns an instance of the `ServletInputStream` object.

5. This `ServletInputStream` object wraps the input stream from the client in the binary form. This binary data is actually the XML that the user sent.

6. After fetching the input stream, the `getDOMDocument()` method creates an instance of the `DocumentBuilder` class.

7. Then it calls the `parse()` method of the `DocumentBuilder` class, passing it the `ServletInputStream` object retrieved in step 4.

8. This `parse()` method authors the XML from the `ServletInputStream` and returns a `Document` object, which contains the XML the user sent to the Web server.

9. The `getDOMDocument()` method assigns this `Document` object to the `DOMDocument` property.

Finally, the `getDOMDocument()` method returns the `DOMDocument` property.

We will normally declare the `IncomingXMLInstanceRequest` model bean in the faces-config.xml file to make this bean available for the whole application. We said in Associating model beans with JSF components on page 67 that the application-level declarations are made in the faces-config.xml file.

Look at the following declaration of the `IncomingXMLInstanceRequest` bean in the faces-config.xml file:

```
<managed-bean>
   <description>Incoming XML Request</description>

<managed-bean-name>incomingXMLInstanceRequest</managed-bean-name>
```

```
    <managed-bean-class>
       xforms_jsf.IncomingXMLInstanceRequest
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
```

The `managed-bean-name` element wraps the name of the bean: `incomingXMLInstanceRequest`. This name acts like an instance for the class. The application classes access the bean through this name.

The `managed-bean-class` element specifies the qualified name of the bean class: `xforms_jsf.IncomingXMLInstanceRequest`.

The `managed-bean-scope` element wraps the scope for the bean. The scope for the `IncomingXMLInstanceRequest` bean is specified as `request`, which tells that this bean is instantiated on each request from a client.

As you know, the `IncomingXMLInstanceRequest` class loads incoming requests, so this bean is instantiated on each request from the client.

Now let's discuss how the `decode()` method of different components uses this class to access the request.

The `decode()` method of each component in the component tree does the following:

1. Gets an instance of the `incomingXMLInstanceRequest` JavaBean component from the application (as mentioned above, it's a bean defined in the faces-config.xml file)

2. Calls the `getDOMDocument()` method of the `incomingXMLInstanceRequest` bean

3. Iterates through the document that contains XML nodes and retrieves data related to the component

---

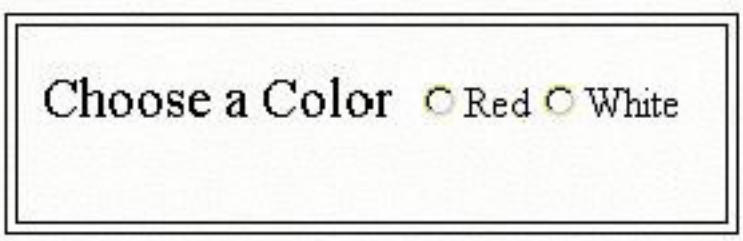
Implementing the xforms-jsf:selectOneRadio tag

The `xforms-jsf:selectOneRadio` tag we are going to implement here renders the following markup:

```
<xforms:select1 ref="selectedColor" model="optModel"
  appearance="full" xmlns:xforms= "http://www.w3.org/2002/xforms">
  <xforms:label>Choose a Color:</xforms:label>
  <xforms:item>
    <xforms:label>Red</xforms:label>
    <xforms:value>Red</xforms:value>
  </xforms:item>
  <xforms:item>
```

```
      <xforms:label>White</xforms:label>
      <xforms:value>White</xforms:value>
    </xforms:item>
  </xforms:select1>
```

In the XForms browser, the above markup looks like the following figure:



The tag as used in a JSP page looks like the following code:

```
<xforms-jsf:selectOneRadio value="#{dataStore.selectedColor}"
  model="optModel" ref=" selectedColor" label="Choose a Color:"
  valueChangeListener="#{dataStore.valueChanged}">
  <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectOneRadio>
```

There are two things to be explained regarding the above code: the `xforms-jsf:selectOneRadio` and `f:selectItems` tags.

The three most important attributes of the `xforms-jsf:selectOneRadio` tag are:

The `value` attribute: As explained in the previous section, the `value` attribute is used to specify the model bean and its property associated with a JSF tag. In the above example, the name of the bean (`dataStore`) and its property (`selectedColor`) is passed to the component in `value`. This property tracks the user's selection that is made from a list of choices.

The `model` attribute: The `model` attribute of the `xforms-jsf:selectOneRadio` element is used to establish an association between the `select1` and `model` elements. The `model` attribute value of the `select` element matches the `id` attribute value of the `model` element. The `model` attribute of the `select1` element in the `xforms-jsf:selectOneRadio` tag indicates that the XForms `select1` element is associated with the XForms `model` element whose `id` attribute has the value "optModel."

The `ref` attribute: This attribute refers to a tag in the XForms `model` element. In the `xforms-jsf:selectOneRadio` tag above, the `ref` attribute contains selectedColor as a value, which is the name of a tag in the XForms `model` element that we discussed in Implementing the xforms-jsf:model component on page 83. In other words, the value of the `ref` attribute always contains the name of the tag from the application-specific XML in the XForms `model` element.

The XForms `select1` element binds itself with the XForms `model` element through this `ref` attribute. We have already explained the purpose of having `ref` attributes in XForms elements in The select element on page 18.

Now let's discuss the `f:selectItems` -- the child tag of the `xforms-jsf:selectOneRadio` tag. Actually, we used this tag from the JSF's core tag library where it provides components with a list of values from model beans to render on the browser.

In the above JSP code, the `f:selectItems` tag has a `value` attribute (whose value is "#{dataStore.colorsList}"). Here, the `value` attribute carries the name of the model bean property, which holds the list of choices that will be rendered in response to the `xforms-jsf:selectOneRadio` tag, as shown in the above screenshot.

Now let's start implementing the `selectOneRadio` tag. The tag entry in the TLD file for the `selectOneRadio` tag looks like the following code:

```
<tag>
  <name>selectOneRadio</name>
  <tag-class>xforms_jsf.Select1RadioTag</tag-class>
  <body-content>JSP</body-content>
  <!-- JSF Specific Attributes -->
  <attribute>
    <name>value</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>valueChangeListener</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>rendered</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>id</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>binding</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>immediate</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>required</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
```

```
    <!-- Xforms Specific attributes -->
    <attribute>
      <name> model </name>
      <required> false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> label </name>
      <required> false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> navindex </name>
      <required> false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> accesskey </name>
      <required>false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> bind </name>
      <required> false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> ref </name>
      <required> true </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> selection  </name>
      <required> false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> incremental </name>
      <required> false </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <!-- JSF Specific attributes -->
    <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name> value </name>
      <required> true </required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>rendered</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
```

The above TLD file entry declares two types of attributes: XForms-specific attributes and JSF-specific attributes.

In the above TLD file entry, the name of the tag class is `xforms_jsf.Select1RadioTag`, which we have implemented in the following code:

```
public class Select1RadioTag extends UIComponentTag{
   private String value              = null;
   private String required           = null;
   private String immediate          = null;
   private String valueChangeListener = null;

   private String ref            = null;
   private String bind           = null;
   private String model          = null;
   private String label          = null;
   private String navindex       = null;
   private String accesskey      = null;
   private String selection      = null;
   private String incremental    = null;
   public String getRendererType(){
      return null;
   }
   public String getComponentType(){
      return "Select1";
   }
   public void setValue(String valueRef) {
      this.value = valueRef;
   }
   public void setRequired(String required) {
      this.required = required;
   }
   public void setImmediate(String immediate) {
      this.immediate = immediate;
   }
   public void setValueChangeListener(String valueChangeListener) {
      this.valueChangeListener = valueChangeListener;
   }
   public void setRef(String ref){
      this.ref = ref;
   }
   public void setBind(String bind){
      this.bind = bind;
   }
   public void setModel(String model){
      this.model = model;
   }
   public void setLabel(String label){
      this.label = label;
   }
   public void setNavindex(String navindex){
      this.navindex = navindex;
   }
   public void setAccesskey(String accesskey){
      this.accesskey = accesskey;
   }
   public void setSelection(String selection){
      this.selection = selection;
   }
   public void setIncremental(String incremental){
      this.incremental = incremental;
   }
   public void setProperties(UIComponent component) {
```

```
    super.setProperties(component);
    UISelect1 uis = (UISelect1)component;
    FacesContext fc = FacesContext.getCurrentInstance();
    Application app = fc.getApplication();
    if(required != null) {
      if(UIComponentTag.isValueReference(required)) {
          ValueBinding vb = app.createValueBinding(required);
          uis.setValueBinding("required", vb);
      }
      else{
        boolean boolRequired =
             (new Boolean(required)).booleanValue();
        uis.setRequired(boolRequired);
      }
    }
    if(valueChangeListener != null) {
      if(UIComponentTag.isValueReference(valueChangeListener)) {
          Class args[] = {
             javax.faces.event.ValueChangeEvent.class
          };
          MethodBinding mb =
             app.createMethodBinding(valueChangeListener, args);
          uis.setValueChangeListener(mb);
      }
    }
    if(immediate != null) {
      if(UIComponentTag.isValueReference(immediate)) {
        ValueBinding vb = app.createValueBinding(immediate);
        uis.setValueBinding("immediate", vb);
      }
      else{
        boolean boolImmediate =
             (new Boolean(immediate)).booleanValue();
        uis.setImmediate(boolImmediate);
      }
    }
    if(value != null) {
      if(UIComponentTag.isValueReference(value)) {
        ValueBinding vb = app.createValueBinding(value);
        uis.setValueBinding("value", vb);
      }
      else{
        uis.setValue(value);
      }
    }
    if(ref != null)
       uis.getAttributes().put("ref", ref);
    if(bind != null)
       uis.getAttributes().put("bind", bind);
    if(label != null)
       uis.getAttributes().put("label", label);
    if(model != null)
       uis.getAttributes().put("model", model);
    if(navindex != null)
       uis.getAttributes().put("navindex", navindex);
    if(accesskey != null)
       uis.getAttributes().put("accesskey", accesskey);
    if(selection != null)
       uis.getAttributes().put("selection", selection);
    if(incremental != null)
       uis.getAttributes().put("incremental", incremental);
    uis.getAttributes().put("appearance", "full");
  }//setProperties
```

```
}//Select1RadioTag
```

This code is exactly like the code used in Implementing the xforms-jsf:model component on page 83, so we won't explain it here.

Here's how to implement the component class associated with the `xforms-jsf:selectOneRadio` tag. Look at the following entry in the faces-config.xml file:

```
<?xml version="1.0"?>
<faces-config>
   .....
   <component>
      <component-type>Select1</component-type>

<component-class>xforms_jsf.UISelect1</component-class>
   </component>
   <!-- other component instances-->
</faces-config>
```

In the next section, we will implement the component class mentioned in the `component-class` element of the faces-config.xml file above (`UISelect1`).

Implementing the UISelect1 component

The `UISelect1` component class lets users select one choice from a list of given choices. If the user changes the component's value, the `UISelect1` component class fires a value-change event and automatically updates the application-specific model bean with the user's new selection.

The `UISelect1` component informs application-specific event-handlers about the user's selection of a new value by firing a value-change event.

The behavior of the `xforms-jsf:selectOneRadio` component is quite similar to the behavior of its equivalent component provided by the JSF's HTML tag library. However, there are slight differences in the implementation. The JSF's HTML tag library component renders HTML markup while its equivalent XForms-JSF tag library component (`UISelect1`) renders XForms markup.

While implementing the `UISelect1` component, we extended our component from the `UIInput` class, which is part of the JSF's HTML tag library. The difference is between encoding the markup and decoding the request. And while implementing, we will override these two methods in `UISelect1`.

The methods in the `UISelect1` class are shown in the following code:

```
public class UISelect1 extends UIInput{
  private Iterator getItems(FacesContext context) {
  }
```

```
    public void encodeEnd(FacesContext fc) throws IOException {
    }
    private String getNewValue(Document doc, String tag) {
    }
    public void decode(FacesContext fc) throws IOException {
    }
    public void validate(FacesContext context){
    }
  }//UISelect1
```

Now let's discuss the implementation of the methods shown one by one.

The `getItems()` helper method returns an instance of the `Iterator` object, which contains the list of choices that the `encodeEnd()` method renders. The following code shows the implementation of the `getItems()` method:

```
  private Iterator getItems(FacesContext context) {
    Object value = null;
    ArrayList list = new ArrayList();
    for(Iterator kids = this.getChildren(); kids.hasNext();)
    {
      UIComponent kid = (UIComponent)kids.next();
      if(kid instanceof UISelectItems)
      {
        value = ((UISelectItems)kid).currentValue(context);
        if(value instanceof Collection)
        {
          for(Iterator item = ((Collection)value).iterator();
            item.hasNext();)
            list.add((SelectItem)item.next());
          return list.iterator();
        }
      }
    }
    return new Iterator(){
      public void remove(){}
      public boolean hasNext(){return false;}
      public Object next(){throw new NoSuchElementException();}
    };
  }//getItems
```

Notice the following points:

- The `getItems()` method first gets the child component of `UISelect1`.
- Then it gets the model bean property value that the JSP author specified in the `value` attribute of the `f:selectItems` tag.
- Then the `getItems()` method iterates through the values fetched in step 2 and adds them to an `ArrayList` object.
- Next, it converts the `ArrayList` into an `Iterator` object and returns the `Iterator` object.
- If there is no child component, this method returns an empty `Iterator`.

Now let's discuss the implementation of the `encodeEnd()` method:

```
public void encodeEnd(FacesContext fc) throws IOException {
  if(fc == null)
    throw new NullPointerException();
  if(!isRendered())
    return;
  String ref         = (String) getAttributes().get("ref");
  String bind        = (String) getAttributes().get("bind");
  String model       = (String) getAttributes().get("model");
  String label       = (String) getAttributes().get("label");
  String navindex    = (String) getAttributes().get("navindex");
  String accesskey   = (String) getAttributes().get("accesskey");
  String selection   = (String) getAttributes().get("selection");
  String appearance  = (String) getAttributes().get("appearance");
  String incremental = (String) getAttributes().get("incremental");
  ResponseWriter out = fc.getResponseWriter();
  out.write("<xforms:select1 ref=\"" + ref +"\"");
  if(bind != null)
     out.write(" bind=\""+ bind + "\"");
  if(model != null)
     out.write(" model=\""+ model + "\"");
  if(navindex != null)
     out.write(" navindex=\""+ navindex + "\"");
  if(selection != null)
     out.write(" selection=\""+ selection + "\"");
  if(accesskey != null)
     out.write(" accesskey=\""+ accesskey + "\"");
  if(appearance != null)
     out.write(" appearance=\""+ appearance + "\"");
  if(incremental != null)
     out.write(" incremental=\""+ incremental + "\"");
  out.write(" xmlns:xforms=\"http://www.w3.org/2002/xforms\">");
  if(label != null)
    out.write("<xforms:label>" + label +
"</xforms:label>");
  else
    out.write("<xforms:label>" + "Select One" +
"</xforms:label>");
  Iterator items = null;
  items = getItems(fc);
  SelectItem si = null;
  if(items == null)
    throw new NullPointerException();
  while(items.hasNext()){
     si = (SelectItem)items.next();
     out.write("<xforms:item>");

     String itemLabel = (String)si.getLabel();
     String strValue  = (String)si.getValue();
     if(itemLabel != null){
       out.write("<xforms:label>" + itemLabel +
"</xforms:label>");
       out.write("<xforms:value>" + strValue +
"</xforms:value>");
     }
     else
        out.write("<xforms:label>Option</xforms:label>");
     out.write("</xforms:item>");
  }//while()
  out.write("</xforms:select1>");
```

```
}//encodeEnd
```

The `encodeEnd()` method above performs the following steps:

1. It first verifies the value of the `FacesContext` object. If it is `null`, it throws a `NullPointerException`.

2. After verifying the context, it retrieves the `ResponseWriter` object from `FacesContext` by calling its `getResponseWriter()` method.

3. Then it retrieves the values of attributes set by the `setProperties()` method of `Select1RadioTag` by calling the `get()` method and passing it the name of the attribute.

4. Next, it calls the `getItems()` method as already explained.

5. Finally, the `encodeEnd()` method writes the markup to render the list of choices returned by the `getItems()` method call.

The `decode()` method calls a helper method named `getNewValue()`, passing it the document that contains the request and the XML tag that wraps the user's selected value along with the method call. The `getNewValue()` method returns the user's selected value after retrieving it from the document. The code for the `getNewValue()` method:

```
private String getNewValue(Document doc, String tag) {
  String newValue = null;
  NodeList nl = doc.getElementsByTagName(tag);
  if (nl != null){
     Node tags = null;
     int length = nl.getLength();
     for(int i=0; i<length ; i++){
        tags = nl.item(i);
        NodeList children = tags.getChildNodes();
        if(children == null)
           break;
        if(children.item(i) == null)
           break;
        newValue = (String)children.item(i).getNodeValue();
     }
  }//if(nl != null)
  return newValue;
}//getNewValue
```

The `decode()` method needs another helper method named `getModelBeanObject()`. The `getModelBeanObject()` method takes a string value (the name of the model bean property that contains the request) and the `FacesContext` object along with the method call. This method returns the actual value of the model bean property:

```
  public Object getModelBeanObject(String ref, FacesContext fc){
    Application app = fc.getApplication();
```

```
    ValueBinding vb = app.createValueBinding(ref);
    return vb.getValue(fc);
  }//getModelBeanObject
```

The `decode()` method is implemented to parse the incoming request. In an XForms-JSF application, the request is in XML. We demonstrated the parsing mechanism in Parsing the incoming XML instance data on page 91.

The code for the `decode()` method:

```
public void decode(FacesContext fc){
  if(fc == null)
    throw new NullPointerException();
  String tag      = null;
  String newValue = null;
  String ref      = (String) getAttributes().get("ref");
  String bind     = (String) getAttributes().get("bind");
  String clientId = getClientId(fc);
  if(bind == null)
    tag = ref;
  else
    tag = bind;
  Document doc = (Document) getModelBeanObject
           ("#{IncomingXMLInstanceRequest.DOMDocument}", fc);
  if(doc != null){
    newValue = getNewValue(doc, tag);
  }
  setSubmittedValue(newValue);
  setValid(true);
}//decode
```

Notice the following points from the `decode()` method above:

1. It retrieves the values of the attributes set by the `setProperties()` method of `Select1RadioTag` by calling the `get()` method and passing it the name of the attribute.

2. The `decode()` method calls the `getModelBeanObject()` method, passing it the "incomingXMLInstanceRequest.DOMDocument" model bean and the `FacesContext` instance along with the method call. This method call returns a `Document` object, which contains the XML request.

3. If the `Document` object fetched in step 2 is `null`, the `decode()` method stops processing and returns.

4. If the document is not `null`, it calls the `getNewValue()` method, which returns the value selected by the user.

5. Next, `decode()` calls the `setSubmittedValue()` method to store the user's selected values.

Now the `UISelect1` class is ready to be used with any JSF application. In the last subsection of this section (Trying out the sample application on page 116),

we will explain the usage of this component in a sample XForms-JSF application.

Next, we will implement the `xforms-jsf:commandButton` tag.

Implementing the xforms-jsf:commandButton tag

The `xforms-jsf:commandButton` tag we are going to implement here renders the following markup:

```
<xforms:submit submission="submit"
   xmlns:xforms="http://www.w3.org/2002/xforms">
   <xforms:action ev:event="DOMActivate"
      xmlns:ev="http://www.w3.org/2001/xml-events">
      <xforms:setvalue
ref="action_performed">id1</xforms:setvalue>
   </xforms:action>
   <xforms:label>Submit</xforms:label>
</xforms:submit>
```

In the XForms browser, the above markup looks like the following figure:

The tag as used in a JSP page looks like the following code:

```
  <xforms-jsf:commandButton  label="Submit"
    actionListener="#{dataStore.actionPerformed}">
  </xforms-jsf:commandButton >
```

The `xforms-jsf:commandButton` tag renders the markup for the XForms `submit` element.

The tag entry in the TLD file for the `commandButton` tag looks like the following code:

```
<tag>
  <name>commandButton </name>
  <tag-class>xforms_jsf.ButtonTag</tag-class>
  <!-- Xforms Specific attributes -->
  <attribute>
    <name> model </name>
```

```
    <required> false </required>
    <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
    <name> appearance </name>
    <required> false </required>
    <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
    <name> navindex </name>
    <required> false </required>
    <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
    <name> accesskey </name>
    <required>false </required>
    <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
    <name> bind </name>
    <required> false </required>
    <rtexprvalue> false </rtexprvalue>
  </attribute>
  <attribute>
    <name> ref </name>
    <required> true </required>
    <rtexprvalue> false </rtexprvalue>
  </attribute>
  <!-- JSF Specific attributes -->
  <attribute>
    <name>id</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>action</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>actionListener</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>immediate</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>value</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>binding</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>rendered</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
```

```
    </attribute>
  </tag>
```

In the above TLD file, we mentioned `xforms_jsf.ButtonTag` as the tag handler class. The following code shows the implementation of the `ButtonTag` class:

```
public class ButtonTag extends UIComponentTag{
  private String value = null;
  private String action = null;
  private String immediate = null;
  private String actionListener = null;

  private String ref = null;
  private String bind = null;
  private String label = null;
  private String model = null;
  private String navindex = null;
  private String accesskey = null;
  private String appearance = null;

  public String getRendererType(){
    return null;
  }
  public String getComponentType(){
    return "Submit";
  }
  public String getValue(){
   return value;
  }
  public void setValue(String newValue) {
    value = newValue;
  }
  public String getAction(){
    return action;
  }
  public void setAction(String newAction) {
    action = newAction;
  }
  public String getImmediate(){
    return immediate;
  }
  public void setImmediate(String newImmediate) {
    immediate = newImmediate;
  }
  public String getActionListener(){
    return actionListener;
  }
  public void setActionListener(String newActionListener) {
    actionListener = newActionListener;
  }
  public String getRef(){
    return ref;
  }
  public String getBind(){
    return bind;
  }
  public String getModel(){
    return model;
  }
```

```
public String getLabel(){
  return label;
}
public String getNavindex(){
  return navindex;
}
public String getAccesskey(){
   return accesskey;
}
public String getAppearance(){
   return appearance;
}
public void setRef(String newRef){
  ref = newRef;
}
public void setBind(String newBind){
  bind = newBind;
}
public void setLabel(String newLabel){
  label = newLabel;
}
public void setModel(String newModel){
  model = newModel;
}
public void setNavindex(String newNavindex){
  navindex = newNavindex;
}
public void setAccesskey(String newAccesskey){
  accesskey = newAccesskey;
}
public void setAppearance(String newAppearance) {
  appearance = newAppearance;
}
protected void setProperties(UIComponent component) {
  super.setProperties(component);
  UIButton uib = (UIButton)component;
  FacesContext fc = FacesContext.getCurrentInstance();
  Application app = fc.getApplication();
  if(action != null) {
    if(UIComponentTag.isValueReference(action)) {
       MethodBinding mb = app.createMethodBinding(action, null);
       uib.setAction(mb);
    }
    else{
      MethodBinding mb = new ConstantMethodBinding(action);
      uib.setAction(mb);
    }
  }
  if(actionListener != null) {
    if(UIComponentTag.isValueReference(actionListener)) {
      Class args[] = {
        javax.faces.event.ActionEvent.class
      };
      MethodBinding mb=app.createMethodBinding(actionListener,args);
      uib.setActionListener(mb);
     }
  }
  if(immediate != null) {
    if(UIComponentTag.isValueReference(immediate)) {
      ValueBinding vb = app.createValueBinding(immediate);
      uib.setValueBinding("immediate", vb);
    }
    else{
```

```
          boolean boolImmediate =
            (new Boolean(immediate)).booleanValue();
          uib.setImmediate(boolImmediate);
        }
      }
      if(value != null) {
        if(UIComponentTag.isValueReference(value)) {
          ValueBinding vb = app.createValueBinding(value);
          uib.setValueBinding("value", vb);
        }
        else{
          uib.setValue(value);
        }
      }
      if(ref != null)
        uib.getAttributes().put("ref", ref);
      if(bind != null)
        uib.getAttributes().put("bind", bind);
      if(label != null)
        uib.getAttributes().put("label", label);
      if(model != null)
        uib.getAttributes().put("model", model);
      if(navindex != null)
        uib.getAttributes().put("navindex", navindex);
      if(accesskey != null)
        uib.getAttributes().put("accesskey", accesskey);
      if(appearance != null)
        uib.getAttributes().put("appearance", appearance);
     }//setProperties
  }//ButtonTag
```

This code is exactly like the code used in Implementing the xforms-jsf:model component on page 83, so we won't explain it here.

Here's how to implement the component class associated with the `xforms-jsf:commandButton` tag. Look at the following entry in the faces-config.xml file:

```
<component>
  <component-type>Submit</component-type>
  <component-class>xforms_jsf.UIButton</component-class>
</component>
```

In the next section, we will implement the component class mentioned in the `component-class` element of the faces-config.xml file above (`UIButton`).

Implementing the UIButton component

The `UIButton` component renders an XForms `submit` element. The `UIButton` component provides users with a facility to submit data back to the application after making changes in the components.

The behavior of the `xforms-jsf:commandButton` component is quite similar to the behavior of its equivalent component in the JSF HTML tag library. The

difference comes while performing the same task with two different technologies. The `xforms-jsf:commandButton` component renders XForms markup and works according to the XForms requirements (such as parsing a request that is in XML).

While implementing the `UIButton` component class, we extended the class from the `UICommand` class, which is the component class for the `h:commandButton` tag in the JSF HTML tag library. The difference is between encoding and decoding so, while implementing, we override these methods in the `UIButton` class.

The following code shows the method in the `UIButton` component class:

```
public class UIButton extends UICommand{
  public void encodeBegin(FacesContext context)
    throws IOException{
  }
  public void decode(FacesContext fc) {
  }
  private Object getModelBeanObject(String valueRef, FacesContext fc){
  }
  private String getButtonValue(Document doc, String tag) {
  }
}//UIButton
```

The `UIButton` class implements two public methods (`encodeBegin()` and `decode()`) and two private methods (`getModelBeanObject()` and `getButtonValue()`).

The following code shows the implementation of the `encodeBegin()` method:

```
public void encodeBegin(FacesContext context)
  throws IOException
{
  if(context == null)
     throw new NullPointerException();
  if(!isRendered())
    return;
  String ref        = (String) getAttributes().get("ref");
  String bind       = (String) getAttributes().get("bind");
  String model      = (String) getAttributes().get("model");
  String label      = (String) getAttributes().get("label");
  String navindex   = (String) getAttributes().get("navindex");
  String accesskey  = (String) getAttributes().get("accesskey");
  String appearance = (String) getAttributes().get("appearance");
  ResponseWriter writer = context.getResponseWriter();
  writer.write("<BR/><xforms:submit");
  writer.write(" submission=\"submit\"");
  if(bind != null)
     writer.write(" bind=\""+ bind + "\"");
  if(model != null)
     writer.write(" model=\""+ model + "\"");
  if(navindex != null)
     writer.write(" navindex=\""+ navindex + "\"");
  if(accesskey != null)
     writer.write(" accesskey=\""+ accesskey + "\"");
```

```
    if(appearance != null)
       writer.write("appearance=\""+ appearance + "\"");
    writer.write(" xmlns:xforms=\"http://www.w3.org/2002/xforms\"");
    writer.write(" xmlns:ev=\"http://www.w3.org/2001/xml-events\">");
    writer.write("<xforms:action ev:event=\"DOMActivate\">");
    writer.write("<xforms:setvalue ref=\"action-performed\">");
    writer.write(getClientId(context));
    writer.write("</xforms:setvalue>");
    writer.write("</xforms:action>");
    writer.write("<xforms:label>");
    if(label != null)
       writer.write(label);
    else
       writer.write("Submit");
    writer.write("</xforms:label>");
    writer.write("</xforms:submit><BR/>");
  }//encodeBegin
```

The following points explain the `encodeBegin()` method:

1. It checks the value of the rendered attribute. If it is false, it does not render the markup for the component and returns.

2. It retrieves the `ResponseWriter` object from the `FacesContext` object.

3. It fetches the attribute values (set by the `setProperties()` method of `ButtonTag`) by calling the `get()` method and passing it the name of the attribute.

4. The `encodeBegin()` method writes the markup for the component by calling the `write()` method of the `ResponseWriter` object.

The `decode()` method calls a private helper method named `getButtonValue()`, which returns the ID of the button the user clicked:

```
  private String getButtonValue(Document doc, String tag){
    String value = null;
    NodeList nl = doc.getElementsByTagName(tag);
    if (nl != null){
      Node tags = null;
      int length = nl.getLength();
      for(int i=0; i<length ; i++){
        tags = nl.item(i);
        if (tags.getNodeName().equals(tag)){
          NodeList children = tags.getChildNodes();
          value = children.item(i).getNodeValue();
        }
      }//for
    }//if (nl != null)
    return value;
  }//getButtonValue
```

The following code shows the implementation of the `decode()` method:

```
public void decode(FacesContext fc){
  if(fc == null)
    throw new NullPointerException();
  String value    = null;
  String clientId = getClientId(fc);
  Document doc    = (Document) getModelBeanObject
    ("#{IncomingXMLInstanceRequest.DOMDocument}", fc);
  if(doc != null){
    value = getButtonValue(doc, "action-performed");
    if(value == null)
      return;
  }
  if(clientId.equals(value)){
    queueEvent(new ActionEvent(this));
  }
}//decode
```

Notice the following points in the `decode()` method above:

1. The `decode()` method retrieves the client ID by calling the `getClientId()` method.

2. The `decode()` method calls the `getModelBeanObject()` helper method, passing it the "incomingXMLInstanceRequest.DOMDocument" model bean and `FacesContext` instance, along with the method call. This method call returns a `Document` object, which contains the XML request.

3. If the `Document` object is `null`, the `decode()` method stops processing and returns.

4. If the `Document` object is not `null`, it calls the `getButtonValue()` private method, which returns the ID of the button clicked by the user.

5. It compares this ID with the client ID retrieved in step 1.

6. If the two values are the same, it fires an action event. Otherwise, it returns without firing any event.

After implementing the `UIButton` class, this component is ready to be used in JSF applications to render the markup for the `submit` element of XForms.

In the next section, we will explain the usage of this component in an XForms-JSF sample application.

Using the model, selectOneRadio, and commandButton tags

We're almost ready to show you how to use all the XForms-JSF components we developed. Before we do, however, we would like to explain the behavior of

the sample JSP page named Demo.jsp. This JSP page provides an option to select one color from a given list of colors and redirect you to the selected color's JSP page.

This page uses the following components:

1. `xforms-jsf:selectOneRadio` to render radio buttons
2. `xforms-jsf:commandButton` to submit the selected option to the application server
3. `xforms-jsf:model` to render the XForms `model` element
4. `DataStore` JavaBean that acts as an application-specific model bean and implements event-handlers

The JSP author has to take care when mapping components to the XForms `model`. For example, XForms can have more than one `model` elements and identifies each `model` with an `id` attribute, which a UI component (such as a `select1` element) refers to using its `model` attribute. Observe the `id` attribute in the `xforms-jsf:model` element and the `model` attribute in the `jsf:selectOneRadio` element in the JSP code below:

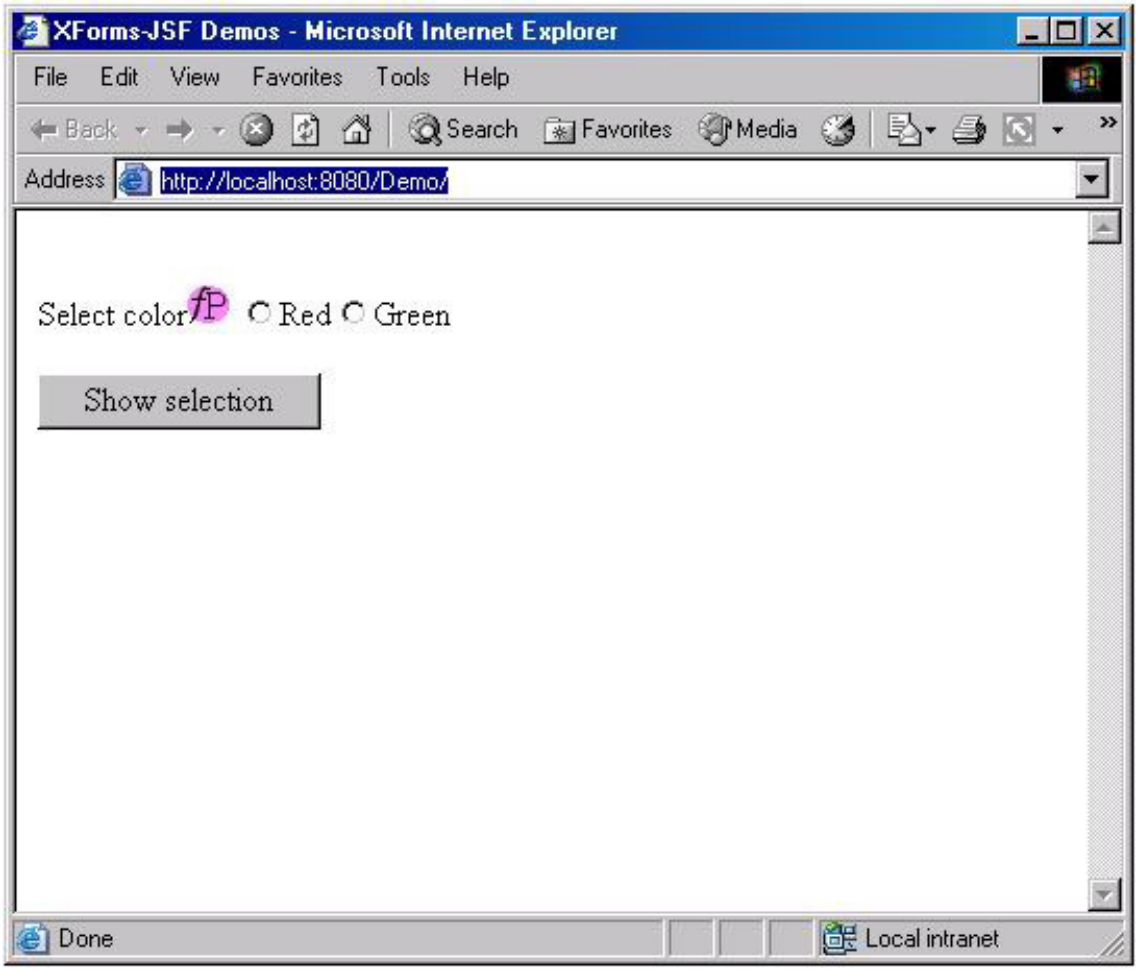
```
<?xml version="1.0" encoding="iso-8859-1"?>
<html>
 ... ...
 <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 <%@ taglib uri="/WEB-INF/xforms-jsf.tld" prefix="xforms-jsf"%>
 <view>
  <head>
     <xforms-jsf:model value="#{dataStore.xformsModelData}"
       id="optModel"/>
  </head>
  <body>
    <xforms-jsf:selectOneRadio ref="selectedColor" label="Select
color"
       value="#{dataStore.selectedColor}">
       <f:selectItems value="#{dataStore.colorsList}"/>
    </xforms-jsf:selectOneRadio>
    <xforms-jsf:commandButton label="Show selection"
immediate="false"
       action="#{dataStore.getAction}"
       actionListener="#{dataStore.btnPressed}">
    </xforms-jsf:commandButton>
  </body>
 </view>
</html>
```

This JSP page renders the following markup:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns:xforms="http://www.w3.org/2002/xforms">
  <head>
    <xforms:model xmlns:xforms="http://www.w3.org/2002/xforms">
      <xforms:submission
action="/xforms-jsf/faces/Demo.jsp;jsessionid=B30A73E86F22A6BB68A36B348CF97D6B"
        method="post" id="submit" />
      <xforms:instance>
```

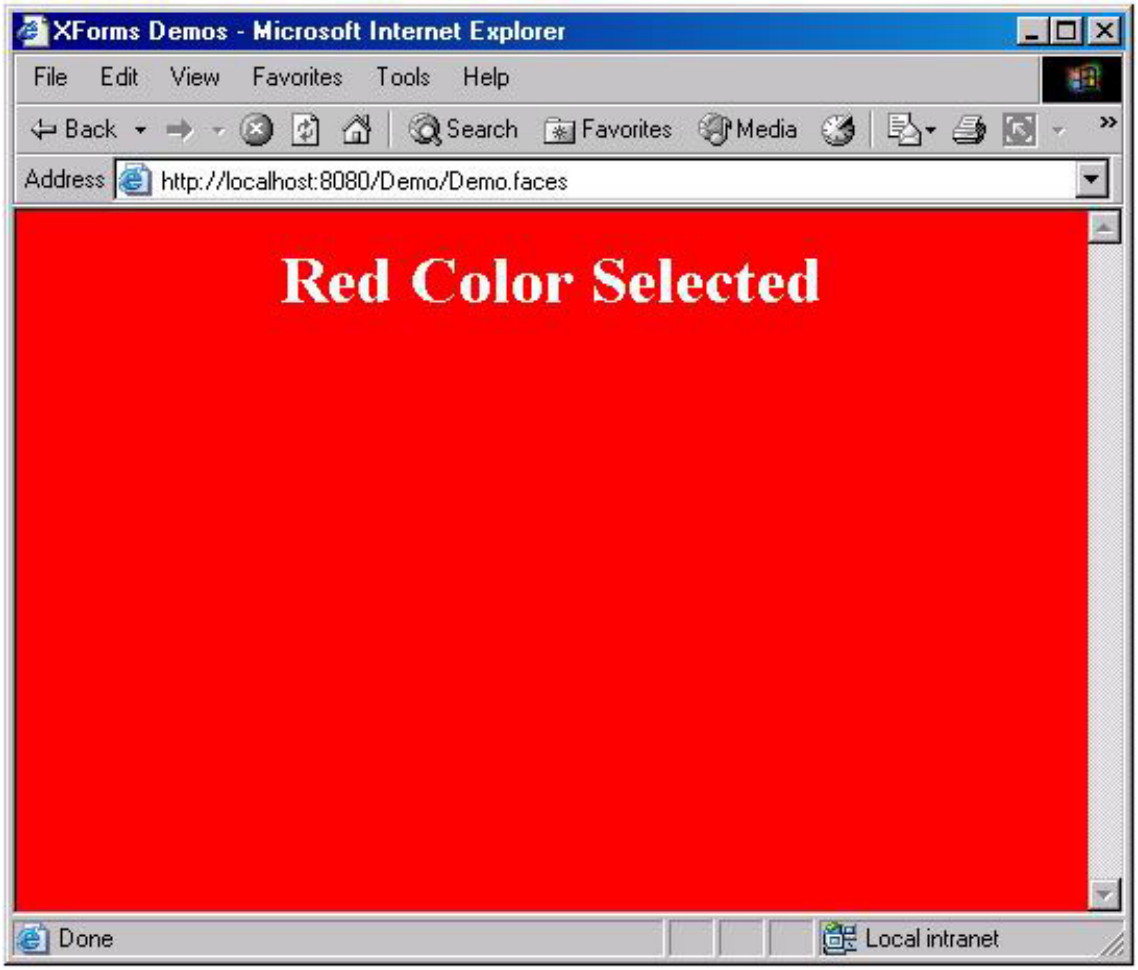
```
        <modelXml>
          <action-performed></action-performed>
          <selectedColor></selectedColor>
        </modelXml>
      </xforms:instance>
    </xforms:model>
  </head>
  <body>
    <xforms:select1 ref="selectedColor" appearance="full"
      xmlns:xforms="http://www.w3.org/2002/xforms">
      <xforms:label>Select a color:</xforms:label>
      <xforms:item>
        <xforms:label>Red</xforms:label>
        <xforms:value>Red</xforms:value>
      </xforms:item>
      <xforms:item>
        <xforms:label>Green</xforms:label>
        <xforms:value>Green</xforms:value>
      </xforms:item>
    </xforms:select1>
    <BR/>
    <xforms:submit submission="submit"
      xmlns:xforms="http://www.w3.org/2002/xforms"
      xmlns:ev="http://www.w3.org/2001/xml-events">
      <xforms:action ev:event="DOMActivate">
        <xforms:setvalue
ref="action-performed">id0</xforms:setvalue>
      </xforms:action>
      <xforms:label>Show Selection</xforms:label>
    </xforms:submit>
    <BR/>
  </body>
</html>
```

The output of this markup is shown in the following screenshot:



This page shows two colors: red and green. You can select one of them and send it to the application server.

Suppose you selected red and submitted it. On the server side, the component checks the selected value (as explained earlier in this tutorial) and redirects it according to the selection -- in this case, to the red.jsp shown below.



All the compiled and uncompiled code of this sample application resides inside the section6.zip file; see Resources on page 200. The next section explains how you can try this code.

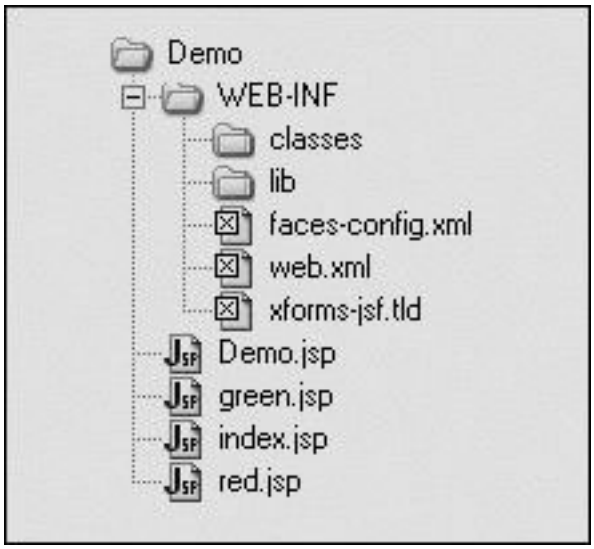
Trying out the sample application

I tested the sample application on the Sun Java System Application Server Platform Edition V8 with J2EE V1.4 Update 1 to run the sample application, but you can use any J2EE 1.4-compliant application server to deploy it.

I've provided a WAR file (Demo.war) so you can deploy the application directly in your application server (be sure to enter the URL `http://localhost:8080/Demo` in the address bar of your XForms browser to run the sample application). The WAR and the Demo folder are in the section6.zip file available in the source code download of this tutorial; see Resources on page200 .

The Demo folder contains the hierarchy of folders and files that the WAR file requires: the JSP pages of our sample application (discussed in the previous section) and a WEB-INF folder. The WEB-INF folder contains a TLD,

faces-config.xml, and web.xml file, as well as a folder named classes (containing the compiled classes for the sample application and the three XForms-JSF tag library-specific components we developed in this section). The arrangement of these files and folders in the Demo folder maps with the arrangement of folders and files in a WAR file, as shown in the screenshot below:



Section 7. XForms-JSF tag library

Completing the XForms-JSF tag library

In this section, we will complete the XForms-JSF tag library and demonstrate the development of each JSF component in it.

In the previous section, we demonstrated the development of `xforms-jsf:model`, `xforms-jsf:commandButton`, and `xforms-jsf:selectOneRadio` components in detail. Now, without going into too much detail, we will discuss the development of the remaining components. We will only stick to the behavior and implementation specific to the components we are developing.

The following components are XForms-JSF tag library components we will develop:

- `xforms-jsf:selectOneMenu`
- `xforms-jsf:selectOneListbox`
- `xforms-jsf:selectManyListbox`
- `xforms-jsf:selectManyCheckbox`
- `xforms-jsf:selectManyMenu`
- `xforms-jsf:inputText`
- `xforms-jsf:inputSecret`
- `xforms-jsf:inputTextarea`

We will explain these components in detail shortly, but first, we'll give a brief description of each component. We'll also discuss the markup generated by the component and the expected interpretation by the XForms browser for the markup, how the JSP author uses the component in the JSP page, and the possible entries for the JSF tag in the TLD file. Finally, we'll discuss the implementation of the tag handler class associated with the tag. After the development of the tag handler class, we will implement the component class for the tag.

Implementing the xforms-jsf:selectOneMenu component

The `xforms-jsf:selectOneMenu` component is similar to the `xforms-jsf:selectOneRadio` component we developed in Implementing the xforms-jsf:selectOneRadio tag on page 94. The only difference is in the graphical appearance of the components in the XForms browser. The remaining features are the same as the `xforms-jsf:selectOneRadio` tag.

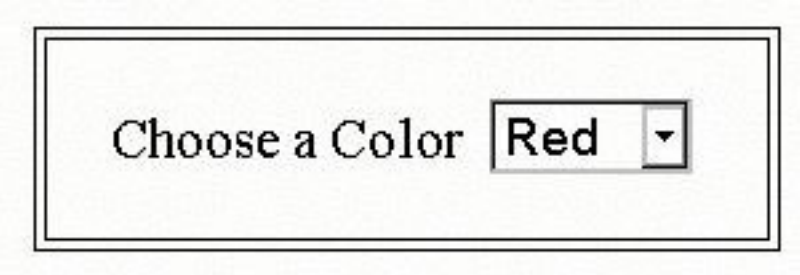
In the case of the `xforms-jsf:selectOneRadio` tag, each choice appears with a radio button, while the `xforms-jsf:selectOneMenu` tag choices appear in a menu list.

The `xforms-jsf:selectOneMenu` component renders the following markup:

```
<xforms:select1 ref="selectedColor" model="optModel"
   appearance="minimal"
xmlns:xforms="http://www.w3.org/2002/xforms">
   <xforms:label>Choose a Color</xforms:label>
   <xforms:item>
     <xforms:label>Red</xforms:label>
     <xforms:value>Red</xforms:value>
   </xforms:item>
   <xforms:item>
     <xforms:label>White</xforms:label>
     <xforms:value>White</xforms:value>
   </xforms:item>
</xforms:select1>
```

Notice the value of the `appearance` attribute in the above markup ("minimal"). In the markup of `xforms-jsf:selectOneRadio`, the value of the `appearance` attribute was "full."

In the XForms browser, the above markup looks like the following:



The JSP author codes the following elements to use the component in the JSP page:

```
<xforms-jsf:selectOneMenu value="#{dataStore.selectedColor}"
  ref="selectedColor" model="optModel">
  <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectOneMenu>
```

The explanation of this JSP code is the same as for the `selectOneRadio` tag in Implementing the xforms-jsf:selectOneRadio tag on page 94.

Let's start implementing the `selectOneMenu` tag.

The first step is to make an entry in the TLD file. The entries in the TLD files are the same as for the `selectOneRadio` tag. The only difference is between the entry in the name element and in the `tag-class` element. Look at the TLD file for `selectOneMenu`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib>
  .....
  <tag>
    <name>selectOneMenu</name>
    <tag-class>xforms_jsf.Select1MenuTag</tag-class>
    <!-- remaining is same as listed for selectOneRadio -->
  </tag>
  <!-- other tag instances-->
</taglib>
```

In this TLD file, we mentioned that the `tag-class` element contains the `xforms_jsf.Select1MenuTag` class. We will implement this tag handler class.

Except for the appearance property of the component, most of the features are the same as for `selectOneRadio` in Implementing the xforms-jsf:selectOneRadio tag on page 94. We will extend the `Select1MenuTag` class from `Select1RadioTag` and override only the `setProperties()` method. The following code shows the implementation of the `Select1MenuTag` class:

```
public class Select1MenuTag extends Select1RadioTag{
  public void setProperties(UIComponent component){
    super.setProperties(component);
    UISelect1 uis = (UISelect1)component;
    uis.getAttributes().put("appearance", "minimal");
  }//setProperties
}//Select1MenuTag
```

We first make a call to the same method of its base class that will perform the default functionality of the `setProperties()` method defined in the base class. Then we override the `appearance` property of the component with the "minimal" value.

The JSF component class we will associate with the `selectOneMenu` is `UISelect1`. It is the same component class we developed for the `selectOneRadio` component in Implementing the UISelect1 component on page 100.

Implementing the xforms-jsf:selectOneListbox component

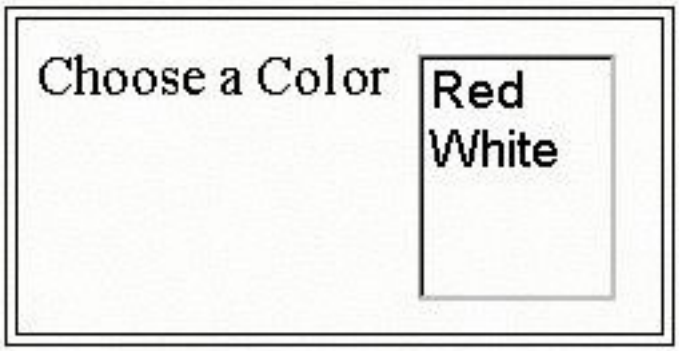
The behavior of the `xforms-jsf:selectOneListbox` tag is similar to the `xforms-jsf:selectOneRadio` component we developed in Implementing the xforms-jsf:selectOneRadio tag on page 94. The only difference is in the graphical appearance of the components in the browser. The remaining aspects are similar to the `xforms-jsf:selectOneRadio` component. In the case of `xforms-jsf:selectOneRadio`, each choice appears with a radio button. In the case of `xforms-jsf:selectOneMenu`, the choices appear in a list box.

The `xforms-jsf:selectOneListbox` component we are going to implement renders the following markup:

```
<xforms:select1 ref="selectedColor" model="optModel"
  appearance="compact" xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:label>Choose a Color</xforms:label>
  <xforms:item>
    <xforms:label>Red</xforms:label>
    <xforms:value>Red</xforms:value>
  </xforms:item>
  <xforms:item>
    <xforms:label>White</xforms:label>
    <xforms:value>White</xforms:value>
  </xforms:item>
</xforms:select1>
```

Notice the value of the `appearance` attribute in the above markup ("compact"). In the markup of `xforms-jsf:selectOneRadio`, the value of the `appearance` attribute was "full."

In the XForms browser, the above markup looks like the following:



The JSP author codes the following elements to use the component in the JSP file:

```
<xforms-jsf:selectOneListbox value="#{dataStore.selectedColor}"
  ref="selectedColor" model="optModel">
  <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectOneListbox>
```

The TLD file entry for the `selectOneListbox` tag is similar to the `selectOneRadio` tag:

```
<tag>
  <name>selectOneListbox</name>
  <tag-class>xforms_jsf.Select1ListTag</tag-class>
  <!-- remaining is same as listed for selectOneRadio -->
</tag>
```

In the above TLD file entry, the `tag-class` element contains the `Select1ListTag` class, which is the name of the tag handler class for the

selectOneListbox tag. We will now implement the Select1ListTag class.

Like the selectOneMenu tag, the selectOneListbox tag is similar to the selectOneRadio tag, except for the appearance property. We will extend the tag handler class for selectOneListbox from Select1RadioTag, as we did earlier for the selectOneMenu tag. The following code shows the implementation of the Select1ListTag class:

```
public class Select1ListTag extends Select1RadioTag{
  public void setProperties(UIComponent component){
    super.setProperties(component);
    UISelect1 uis = (UISelect1)component;
    uis.getAttributes().put("appearance", "compact");
  }//setProperties
}//Select1ListTag
```

The JSF component class we will associate with the selectOneListbox is UISelect1. It is the same class that we developed for the selectOneRadio component in Implementing the UISelect1 component on page 100. There is no need to develop a separate component class, as the behavior of both components is the same.

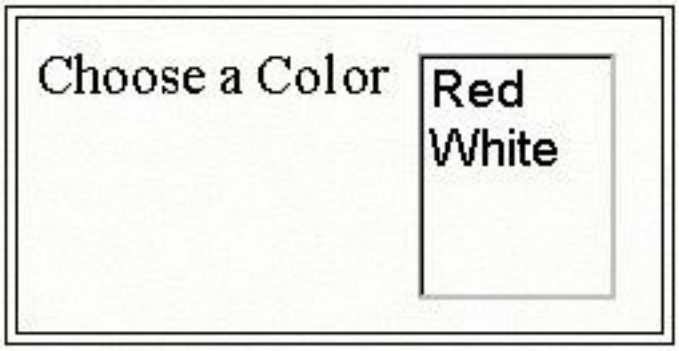
Implementing the xforms-jsf:selectManyListbox component

The xforms-jsf:selectManyListbox component provides users with an option to select multiple choices from a list of given choices in a list box. It renders the XForms select element's markup.

The xforms-jsf:selectManyListbox component renders the following markup:

```
<xforms:select ref="selectedColors" model="optModel"
  appearance="compact" xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:label>Choose a Color</xforms:label>
  <xforms:item>
    <xforms:label>Red</xforms:label>
    <xforms:value>Red</xforms:value>
  </xforms:item>
  <xforms:item>
    <xforms:label>White</xforms:label>
    <xforms:value>White</xforms:value>
  </xforms:item>
</xforms:select>
```

In the XForms browser, the above markup looks like this:



The JSP author codes the following elements to use the component in the JSP file:

```
<xforms-jsf:selectManyListbox ref="selectedColors" model="optModel"
  value="#{dataStore.selectedColors}">
  <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectManyListbox>
```

The details of the attributes of this component are the same as the `xforms-jsf:selectOneRadio` component in Implementing the xforms-jsf:selectOneRadio tag on page 94. The only difference is between the `value` attribute of `selectManyListbox`. As we explained, that value is used by the component to set or get the JavaBeans property. The property that the JSP author passes in the `value` attribute for the `selectManyListbox` tag should be an array of the string object in the model bean (while in the case of the `selectOneRadio` tag, it was simply a string type property).

Now we will implement the `selectManyListbox` tag by making an entry in the TLD file to declare it. The entry looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib>
  ......
  <tag>
    <name>selectManyListbox</name>
    <tag-class>xforms_jsf.SelectListBoxTag</tag-class>
    <!-- JSF Specific Attributes -->
    <attribute>
      <name>value</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>valueChangeListener</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>rendered</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>id</name>
```

```
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>binding</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>immediate</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>required</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <!-- XForms Attributes -->
    <attribute>
      <name> model </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>label</name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> navindex </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> accesskey </name>
      <required>false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> bind </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> ref </name>
      <required> true </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> selection  </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> incremental </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
  </tag>
  <!-other tag instances-->
</taglib>
```

In the above TLD file, we mentioned the tag handler `xforms_jsf.SelectListBoxTag` class in the `tag-class` element. Now we will implement the `SelectListBoxTag` class:

```
public class SelectListBoxTag extends UIComponentTag{
  private String value               = null;
  private String required            = null;
  private String immediate           = null;
  private String valueChangeListener = null;

  private String ref            = null;
  private String bind           = null;
  private String model          = null;
  private String label          = null;
  private String navindex       = null;
  private String accesskey      = null;
  private String selection      = null;
  private String incremental    = null;
  public String getRendererType(){
    return null;
  }
  public String getComponentType(){
    return "Select";
  }
  public void setValue(String valueRef) {
    this.value = valueRef;
  }
  public void setRequired(String required) {
    this.required = required;
  }
  public void setImmediate(String immediate) {
    this.immediate = immediate;
  }
  public void setValueChangeListener(String valueChangeListener) {
    this.valueChangeListener = valueChangeListener;
  }
  public void setRef(String ref) {
    this.ref = ref;
  }
  public void setBind(String bind) {
    this.bind = bind;
  }
  public void setModel(String model) {
    this.model = model;
  }
  public void setLabel(String label) {
    this.label = label;
  }
  public void setNavindex(String navindex) {
    this.navindex = navindex;
  }
  public void setAccesskey(String accesskey) {
    this.accesskey = accesskey;
  }
  public void setSelection(String selection) {
    this.selection = selection;
  }
  public void setIncremental(String incremental) {
    this.incremental = incremental;
  }
  public void setProperties(UIComponent component) {
```

```
      super.setProperties(component);
      UISelect uis = (UISelect)component;
      FacesContext fc = FacesContext.getCurrentInstance();
      Application app = fc.getApplication();
      if(required != null){
        if(UIComponentTag.isValueReference(required)){
          ValueBinding vb = app.createValueBinding(required);
          uis.setValueBinding("required", vb);
        }
        else{
          boolean boolRequired = (new Boolean(required)).booleanValue();
          uis.setRequired(boolRequired);
        }
      }
      if(valueChangeListener != null){
        if(UIComponentTag.isValueReference(valueChangeListener)) {
          Class args[] = {
            javax.faces.event.ValueChangeEvent.class
          };
          MethodBinding mb =
            app.createMethodBinding(valueChangeListener, args);
          uis.setValueChangeListener(mb);
        }
      }
      if(immediate != null) {
        if(UIComponentTag.isValueReference(immediate)) {
          ValueBinding vb = app.createValueBinding(immediate);
          uis.setValueBinding("immediate", vb);
        }
        else{
          boolean boolImmediate =
            (new Boolean(immediate)).booleanValue();
          uis.setImmediate(boolImmediate);
        }
      }
      if(value != null) {
        if(UIComponentTag.isValueReference(value)) {
          ValueBinding vb = app.createValueBinding(value);
          uis.setValueBinding("value", vb);
        }
          uis.setValue(value);
      }
      if(ref != null)
        uis.getAttributes().put("ref", ref);
      if(bind != null)
        uis.getAttributes().put("bind", bind);
      if(label != null)
        uis.getAttributes().put("label", label);
      if(model != null)
        uis.getAttributes().put("model", model);
      if(navindex != null)
        uis.getAttributes().put("navindex", navindex);
      if(accesskey != null)
        uis.getAttributes().put("accesskey", accesskey);
      if(selection != null)
        uis.getAttributes().put("selection", selection);
      if(incremental != null)
        uis.getAttributes().put("incremental", incremental);
      uis.getAttributes().put("appearance", "compact");
    }//setProperties
  }//SelectListBoxTag
```

This code is exactly like the code used in Implementing the xforms-jsf:model component on page 83, so we won't explain it here.

Now we'll discuss the JSF component class associated with the `xforms-jsf:selectManyListbox` tag.

`UISelect` is the component class we will associate with the `xforms-jsf:selectManyListbox` tag. The `UISelect` component is similar to the `UISelect1` component discussed in Implementing the UISelect1 component on page 100. We extended the `UISelect` class from `UISelect1`, which is part of the JSF's HTML tag library.

The following code shows the methods in the UISelect class:

```
public class UISelect extends UISelect1{
  public void encodeEnd(FacesContext fc)
    throws IOException{
  }
  private String[] getNewValues(Document doc, String tag){
  }
}//UISelect
```

The `UISelect` class overrides the `encodeEnd()` method and implements a new helper method named `getNewValues()`.

The `encodeEnd()` method is similar to the same method in the `UISelect1` class, except that it renders a different markup.

The `getNewValues()` helper method returns the values selected by the user, after fetching from the XML request. We need this method because the values selected by the user come as an array of strings in which each individual string is separated from the other by a space. The `getNewValues()` method parses the user's data and returns it in the form of a `String[]`.

The `decode()` method calls the `getNewValues()` method, passing it the `Document` object (that contains the request) and the name of the XML tag that wraps values selected by the user:

```
private String[] getNewValues(Document doc, String tag){
  String[] newValues = null;
  NodeList nl = doc.getElementsByTagName(tag);
  if (nl != null){
    Node tags = null;
    int length = nl.getLength();
    for(int i=0; i<length ; i++){
      tags = nl.item(i);
      NodeList children = tags.getChildNodes();
      if (children == null)
        break;
      if(children.item(i) == null)
        break;
      String values = children.item(i).getNodeValue();
      if(values == null)
        break;
```

```
        int spaceCounter = 0;
        int j = -1;
        while(true) {
          j = values.indexOf(" ", j + 1);
          if(j == -1){
            spaceCounter++;
            break;
          }
          else
            spaceCounter++;
        }
        newValues = new String[spaceCounter];
        int index = 0;
        for(int k = 0 ; values.length() > 0 ; k++){
          k = values.indexOf(" ");
          if(k != -1){
            newValues[index++] = values.substring(0,k);
            values = values.substring(k+1);
          }
          else{
            newValues[index] = values.substring(0);
            break;
          }
        }//for(int k = 0 ; values.length() > 0 ; k++)
      }//for(int i=0; i<length ; i++)
    }//if (nl != null)
    return newValues;
  }//getNewValues
```

Notice the following points in this code:

1. It retrieves the node (from the `Document`) that wraps the new values.

2. It retrieves the contents of the node fetched in step 1 (the list of choices selected by the user).

3. Because this component lets users select more than one choice at a time, a space separates each choice in the node fetched in step 2.

4. The `getNewValues()` method extracts each choice and places it in an array of string type, then returns the array.

Now look at the following code, which shows the decode method implementation:

```
  public void decode(FacesContext fc) {
    if(fc == null)
       throw new NullPointerException();
    String tag      = null;
    String[] newValue = null;
    String ref      = (String) getAttributes().get("ref");
    String bind     = (String) getAttributes().get("bind");
    String clientId = getClientId(fc);
    if(bind == null)
      tag = ref;
    else
```

```
      tag = bind;
    Document doc = (Document) getModelBeanObject
       ("#{incomingXMLInstanceRequest.DOMDocument}", fc);
    if(doc != null) {
      newValue = getNewValues(doc, tag);
    }
    if(newValue == null)
      setSubmittedValue(new String[0]);
    else
      setSubmittedValue(newValue);
    setValid(true);
  }//decode
```

Now you can use this component with any JSF application after making an entry in the faces-config.xml file:

```
<component>
  <component-type>Select</component-type>
  <component-class>xforms_jsf.UISelect</component-class>
</component>
```

Implementing the xforms-jsf:selectManyCheckbox component

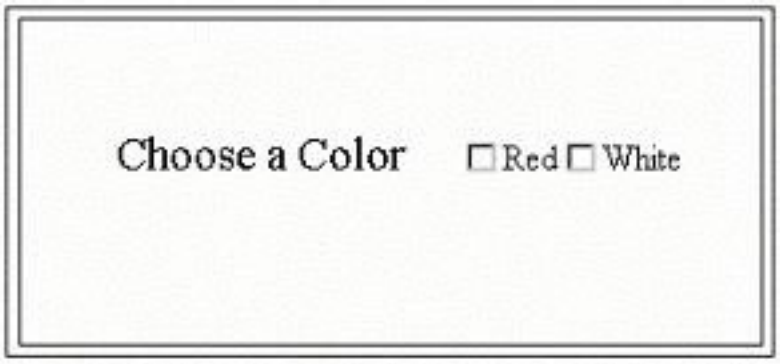
The behavior of the `xforms-jsf:selectManyCheckbox` component is similar to `selectManyListbox`. The only difference is between the appearances of both the components. This component provides users with an option to select one or more check boxes from a list of check boxes (each check box represents a choice).

The `xforms-jsf:selectManyCheckbox` tag renders the following markup:

```
<xforms:select ref="selectedColors" model="optModel"
  appearance="full" xmlns:xforms=  "http://www.w3.org/2002/xforms">
  <xforms:label>Choose a Color</xforms:label>
  <xforms:item>
    <xforms:label>Red</xforms:label>
    <xforms:value>Red</xforms:value>
  </xforms:item>
  <xforms:item>
    <xforms:label>White</xforms:label>
    <xforms:value>White</xforms:value>
  </xforms:item>
</xforms:select>
```

The only difference between the markups of the `xforms-jsf:selectManyCheckbox` and `xforms-jsf:selectManyListbox` tags is the `appearance` attribute.

In the XForms browser, the above markup looks like the following:



The JSP author codes the following elements to use the tag in the JSP file:

```
<xforms-jsf:selectManyCheckbox ref="selectedColors"
  value="#{dataStore.selectedColors}" model="optModel">
  <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectManyCheckbox>
```

Now let's start implementing the `selectManyCheckbox` tag. Its TLD entry:

```
<tag>
  <name>selectManyCheckbox</name>
  <tag-class>xforms_jsf.SelectCheckListTag</tag-class>
  <!-- remaining is same as listed for selectManyListbox -->
</tag>
```

In the above TLD file, we mentioned `<tag-class>xforms_jsf.SelectCheckListTag</tag-class>`. We will implement this tag handler class.

As explained earlier, the `selectManyCheckbox` and `selectManyListbox` components are similar. The only difference is the appearance of the components, which are controlled by the `appearance` property of the component.

We extend the tag handler class for this component from `SelectListBoxTag`, which is the tag handler class for the `selectManyListbox` component (explained in the previous section), and override the `setProperties()` method, which only sets the `appearance` property for the component.

The following code shows the implementation of the `SelectCheckListTag` class, which represents the `selectManyCheckbox` tag:

```
public class SelectCheckListTag extends SelectListBoxTag{
  public void setProperties(UIComponent component){
    super.setProperties(component);
    UISelect uis = (UISelect)component;
    uis.getAttributes().put("appearance", "full");
 }
}//SelectCheckListTag
```

In the `setProperties()` method, we first make a call to the same method of

its base class that performs the default functionality of the `setProperties()` method defined in the base class, then we override the `appearance` property of the component with "full."

The component class for this JSF tag is `UISelect`, the same as for `selectManyListbox`.

Implementing the xforms-jsf:selectManyMenu component

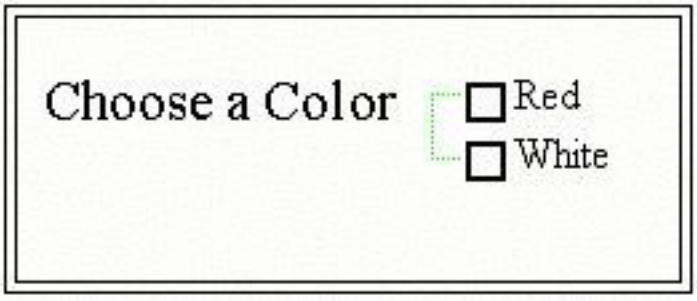
The `xforms-jsf:selectManyMenu` component provides users with an option to select check boxes from a tree of check boxes (in the tree, each node has a check box and a label. Each node represents a choice).

The `xforms-jsf:selectManyMenu` component renders the following markup:

```
<xforms:select ref="selectedColors" model="optModel"
  appearance = "minimal"
xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:label>Choose a Color</xforms:label>
  <xforms:item>
    <xforms:label>Red</xforms:label>
    <xforms:value>Red</xforms:value>
  </xforms:item>
  <xforms:item>
    <xforms:label>White</xforms:label>
    <xforms:value>White</xforms:value>
  </xforms:item>
</xforms:select>
```

If you compare the markups generated by the `xforms-jsf:selectManyMenu` and `xforms-jsf:selectManyListbox` components, you will see that the difference is only in the value of the `appearance` attribute.

In the XForms browser, the above markup looks like the following:



The JSP author codes the following elements to use the component in the JSP file:

```
<xforms-jsf:selectManyMenu  value="#{dataStore.selectedColors}"
  ref="selectedColors" model="optModel">
  <f:selectItems value="#{dataStore.colorsList}"/>
</xforms-jsf:selectManyMenu>
```

The TLD file entry for the `selectManyMenu` tag:

```
<tag>
  <name>selectManyCheckbox</name>
  <tag-class>xforms_jsf.SelectMenuTag</tag-class>
  <!-- remaining is same as listed for selectManyListbox -->
</tag>
```

In the above TLD file, we mentioned `<tag-class>xforms_jsf.SelectMenuTag</tag-class>`. Now we will implement this tag handler class.

The following code shows the implementation of the `SelectMenuTag` class, which represents the `selectManyMenu` tag:

```
public class SelectMenuTag extends SelectListBoxTag{
  public void setProperties(UIComponent component){
    super.setProperties(component);
    UISelect uis = (UISelect)component;
    uis.getAttributes().put("appearance", "minimal");
  }
}//SelectMenuTag
```

In the `setProperties()` method, we first make a call to the same method of its base class, which performs the default functionality of the `setProperties()` method defined in the base class. We then override the `appearance` property of the component with "minimal."

As before, we can use the `UISelect` component class for this tag.

Implementing the xforms-jsf:inputText component

The `xforms-jsf:inputText` component provides the user with an input box to enter text into it. In addition, `xforms-jsf:inputText` renders the following markup:

```
<xforms:input ref="name" model="optModel"
  xmlns:xforms=  "http://www.w3.org/2002/xforms">
  <xforms:label>Name</xforms:label>
  <xforms:hint>Please enter your name.</xforms:hint>
</xforms:input>
```

In the above markup, the XForms `hint` element is used. In the `hint` element, you can wrap a description about the text to enter. For example, in the markup

shown above, the label is "Name." When the user rolls the mouse on the input box, the hint is displayed to the user.

In the XForms browser, the above markup looks like the following:

The JSP author codes the following elements to use the component in the JSP file:

```
<xforms-jsf:inputText value="#{customerData.name}" ref="name"
  model="optModel">
</xforms-jsf:inputText>
```

The component saves the user data in the bean property passed in the `value` attribute.

Let's start implementing the `xforms-jsf:inputText` component. The first step is to make a new tag entry in the TLD file. The tag entry in the TLD file for `xforms-jsf:inputText` looks like the following:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib>
  .....
  <tag>
    <name>inputText</name>
    <tag-class>xforms_jsf.InputBoxTag</tag-class>
    <!-- JSF Specific Attributes -->
    <attribute>
      <name>value</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>valueChangeListener</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>rendered</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
```

```
    </attribute>
    <attribute>
      <name>binding</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>immediate</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>required</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
  <!-- XForms Attributes -->
    <attribute>
      <name> model </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>label</name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> navindex </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> accesskey </name>
      <required>false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> bind </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> hint </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> ref </name>
      <required> true </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> inputmode  </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name> incremental </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
```

```
      <name> appearance </name>
      <required> false </required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
  </tag>
  <!-other tag instances-->
</taglib>
```

In the above TLD file, we mentioned `<tag-class>xforms_jsf.InputBoxTag</tag-class>`. Now let's implement this tag handler class.

The following code shows the implementation of the `InputBoxTag` class:

```
public class InputBoxTag extends UIComponentTag{
  protected String value               = null;
  protected String required            = null;
  protected String immediate           = null;
  protected String valueChangeListener = null;

  protected String ref            = null;
  protected String bind           = null;
  protected String hint           = null;
  protected String model          = null;
  protected String label          = null;
  protected String navindex       = null;
  protected String accesskey      = null;
  protected String inputmode      = null;
  protected String appearance     = null;
  protected String incremental    = null;
  public String getRendererType(){
    return null;
  }
  public String getComponentType(){
    return "InputBox";
  }
  public void setRef(String ref){
    this.ref = ref;
  }
  public void setBind(String bind){
    this.bind = bind;
  }
  public void setHint(String hint){
    this.hint = hint;
  }
  public void setModel(String model){
    this.model = model;
  }
  public void setLabel(String label){
    this.label = label;
  }
  public void setValue(String valueRef){
    this.value = valueRef;
  }
  public void setNavindex(String navindex){
    this.navindex = navindex;
  }
  public void setAccesskey(String accesskey){
    this.accesskey = accesskey;
  }
```

```
public void setInputmode(String inputmode){
  this.inputmode = inputmode;
}
public void setAppearance(String appearance){
  this.appearance = appearance;
}
public void setIncremental(String incremental){
  this.incremental = incremental;
}
public void setProperties(UIComponent component){
  super.setProperties(component);
  UIInputBox uib = (UIInputBox)component;
  FacesContext fc = FacesContext.getCurrentInstance();
  Application app = fc.getApplication();
  if(required != null) {
    if(UIComponentTag.isValueReference(required)) {
      ValueBinding vb = app.createValueBinding(required);
      uib.setValueBinding("required", vb);
    }
    else{
      boolean boolRequired = (new Boolean(required)).booleanValue();
      uib.setRequired(boolRequired);
    }
  }
  if(valueChangeListener != null){
    if(UIComponentTag.isValueReference(valueChangeListener)){
      Class args[] = {
         javax.faces.event.ValueChangeEvent.class
      };
      MethodBinding mb =
         app.createMethodBinding(valueChangeListener, args);
      uib.setValueChangeListener(mb);
    }
  }
  if(immediate != null){
    if(UIComponentTag.isValueReference(immediate)){
      ValueBinding vb = app.createValueBinding(immediate);
      uib.setValueBinding("immediate", vb);
    }
    else{
      boolean boolImmediate =
        (new Boolean(immediate)).booleanValue();
      uib.setImmediate(boolImmediate);
    }
  }
  if(value != null){
    if(UIComponentTag.isValueReference(value)){
      ValueBinding vb = app.createValueBinding(value);
      uib.setValueBinding("value", vb);
    }
    else
      uib.setValue(value);
  }
  if(ref != null)
    uib.getAttributes().put("ref", ref);
  if(bind != null)
    uib.getAttributes().put("bind", bind);
  if(hint != null)
    uib.getAttributes().put("hint", hint);
  if(label != null)
    uib.getAttributes().put("label", label);
  if(model != null)
    uib.getAttributes().put("model", model);
```

```
    if(navindex != null)
      uib.getAttributes().put("navindex", navindex);
    if(accesskey != null)
      uib.getAttributes().put("accesskey", accesskey);
    if(inputmode != null)
      uib.getAttributes().put("inputmode", inputmode);
    if(appearance != null)
      uib.getAttributes().put("appearance", appearance);
    if(incremental != null)
      uib.getAttributes().put("incremental", incremental);
  }//setProperties
}//InputBoxTag
```

Now let's implement the JSF component class associated with `xforms-jsf:inputText`.

The `UIInputBox` is the component class we associate with `xforms-jsf:inputText`. We extend `UIInputBox` from `UISelect1`:

```
public class UIInputBox extends UISelect1{
  public void encodeEnd(FacesContext fc) throws IOException{
    if(fc == null)
      throw new NullPointerException();
    if(!isRendered())
      return;
    String ref         = (String) getAttributes().get("ref");
    String bind        = (String) getAttributes().get("bind");
    String hint        = (String) getAttributes().get("hint");
    String model       = (String) getAttributes().get("model");
    String label       = (String) getAttributes().get("label");
    String navindex    = (String) getAttributes().get("navindex");
    String inputmode   = (String) getAttributes().get("inputmode");
    String accesskey   = (String) getAttributes().get("accesskey");
    String appearance  = (String) getAttributes().get("appearance");
    String incremental = (String) getAttributes().get("incremental");
    ResponseWriter out = fc.getResponseWriter();
    out.write("<xforms:input ref=\"" + ref +"\"");
    if(bind != null)
      out.write(" bind=\""+ bind + "\"");
    if(model != null)
      out.write(" model=\""+ model + "\"");
    if(navindex != null)
      out.write(" navindex=\""+ navindex + "\"");
    if(inputmode != null)
      out.write(" inputmode=\""+ inputmode + "\"");
    if(accesskey != null)
      out.write(" accesskey=\""+ accesskey + "\"");
    if(appearance != null)
      out.write(" appearance=\""+ appearance + "\"");
    if(incremental != null)
      out.write(" incremental=\""+ incremental + "\"");
    out.write(" xmlns:xforms=\"http://www.w3.org/2002/xforms\">");
    if(label != null)
      out.write("<xforms:label>" + label +
"</xforms:label>");
    else
      out.write("<xforms:label>" + "Enter data  " +
"</xforms:label>");
    if(hint != null)
      out.write("<xforms:hint>" + hint + "</xforms:hint>");
```

```
      out.write("</xforms:input>");
    }//encodeEnd
  }//UIInputBox
```

The `UIInputBox` class only overrides the `encodeEnd()` method of the `UISelect1` class to render its own markup. The reason is because the markup rendered is different from the parent class, but the behavior of the tag is not different. It has to manage only one string coming from the user.

Now you can use this component with any JSF application after an entry in the faces-config.xml file:

```
<component>
  <component-type>InputBox</component-type>
  <component-class>xforms_jsf.UIInputBox</component-class>
</component>
```

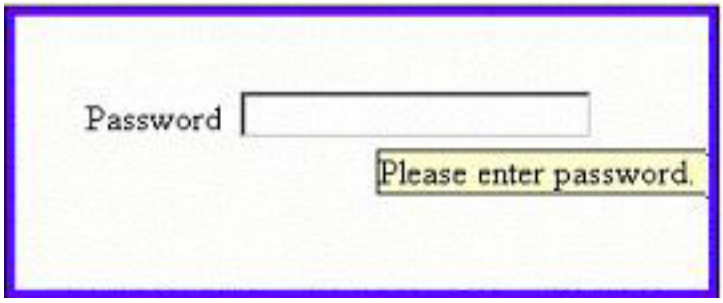
Implementing the xforms-jsf:inputSecret component

The `xforms-jsf:inputSecret` component provides the user with an input box to enter text in a nonreadable form.

The `xforms-jsf:inputSecret` renders the following markup:

```
<xforms:secret ref="password" model="optModel"
  xmlns:xforms=  "http://www.w3.org/2002/xforms">
  <xforms:label>Password</xforms:label>
  <xforms:hint>Pls. Enter password here.</xforms:hint>
</xforms:secret>
```

In the XForms browser, the above markup looks like the following:



The JSP author codes the following elements to use the component in the JSP file.

```
<xforms-jsf:inputSecret value="#{customerData.pwd}" ref="password"
  model="optModel">
</xforms-jsf:inputSecret>
```

The component saves the user's password in the bean property passed in the `value` attribute.

The TLD file for `xforms-jsf:inputSecret` looks like the following:

```
<tag>
  <name>inputSecret</name>
  <tag-class>xforms_jsf.InputSecretTag</tag-class>
  <!-- Remaining is same as implemented for inputText-->
</tag>
```

In the above TLD file, we mentioned `<tag-class>xforms_jsf.InputSecretTag</tag-class>`. Now let's implement this tag handler class.

Next, we'll implement the tag handler class for `xforms-jsf:inputSecret`. The attributes for the `inputSecret` tag are the same as were defined for the `inputText` tag. We extended the tag handler class for `inputSecret` from `InputBoxTag` that is implemented for the `inputText` tag in the previous topic. The following code shows the implementation of the `InputSecretTag` class:

```
public class InputSecretTag extends InputBoxTag{
  public String getComponentType(){
    return "InputSecret";
  }
  public void setProperties(UIComponent component){
    super.setProperties(component);
    UIInputSecret uis = (UIInputSecret)component;
    FacesContext fc = FacesContext.getCurrentInstance();
    Application app = fc.getApplication();
    if(required != null) {
      if(UIComponentTag.isValueReference(required)) {
        ValueBinding vb = app.createValueBinding(required);
        uis.setValueBinding("required", vb);
      }
      else{
        boolean boolRequired = (new Boolean(required)).booleanValue();
        uis.setRequired(boolRequired);
      }
    }
    if(valueChangeListener != null) {
      if(UIComponentTag.isValueReference(valueChangeListener)){
        Class args[] = {
           javax.faces.event.ValueChangeEvent.class
        };
        MethodBinding mb =
          app.createMethodBinding(valueChangeListener, args);
        uis.setValueChangeListener(mb);
      }
    }
    if(immediate != null) {
      if(UIComponentTag.isValueReference(immediate)) {
        ValueBinding vb = app.createValueBinding(immediate);
        uis.setValueBinding("immediate", vb);
      }
      else{
```

```
          boolean boolImmediate =
            (new Boolean(immediate)).booleanValue();
          uis.setImmediate(boolImmediate);
        }
      }
      if(value != null) {
        if(UIComponentTag.isValueReference(value)) {
          ValueBinding vb = app.createValueBinding(value);
          uis.setValueBinding("value", vb);
        }
        else
          uis.setValue(value);
      }
      if(ref != null)
        uis.getAttributes().put("ref", ref);
      if(bind != null)
        uis.getAttributes().put("bind", bind);
      if(hint != null)
        uis.getAttributes().put("hint", hint);
      if(label != null)
        uis.getAttributes().put("label", label);
      if(model != null)
        uis.getAttributes().put("model", model);
      if(navindex != null)
        uis.getAttributes().put("navindex", navindex);
      if(accesskey != null)
        uis.getAttributes().put("accesskey", accesskey);
      if(inputmode != null)
        uis.getAttributes().put("inputmode", inputmode);
      if(appearance != null)
        uis.getAttributes().put("appearance", appearance);
      if(incremental != null)
        uis.getAttributes().put("incremental", incremental);
    }//setProperties
  }//InputSecretTag
```

The `InputSecretTag` class overrides the `getComponentType()` and `setProperties()` methods of the `InputBoxTag` class. The `getComponentType()` method returns the type of component associated with the `xforms-jsf:inputSecret` tag.

Now we will discuss the JSF component class associated with `xforms-jsf:inputSecret`.

`UIInputSecret` is the component class we associated with `xforms-jsf:inputSecret`. We extended the `UIInputSecret` class from the `UIInputBox` class:

```
  public class UIInputSecret extends UIInputBox{
    public void encodeEnd(FacesContext fc) throws IOException{
      if(fc == null)
         throw new NullPointerException();
      if(!isRendered())
        return;
      String ref         = (String) getAttribute("ref");
      String bind        = (String) getAttribute("bind");
      String hint        = (String) getAttribute("hint");
      String model       = (String) getAttribute("model");
      String label       = (String) getAttribute("label");
```

```
      String navindex    = (String) getAttribute("navindex");
      String accesskey   = (String) getAttribute("accesskey");
      String inputmode   = (String) getAttribute("inputmode");
      String appearance  = (String) getAttribute("appearance");
      String incremental = (String) getAttribute("incremental");
      ResponseWriter out = fc.getResponseWriter();
      out.write("<xforms:secret ref=\"" + ref +"\"");
      if(bind != null)
         out.write(" bind=\""+ bind + "\"");
      if(model != null)
         out.write(" model=\""+ model + "\"");
      if(navindex != null)
         out.write(" navindex=\""+ navindex + "\"");
      if(inputmode != null)
         out.write(" inputmode=\""+ inputmode + "\"");
      if(accesskey != null)
         out.write(" accesskey=\""+ accesskey + "\"");
      if(appearance != null)
         out.write(" appearance=\""+ appearance + "\"");
      if(incremental != null)
         out.write(" incremental=\""+ incremental + "\"");
      out.write(" xmlns:xforms=\"http://www.w3.org/2002/xforms\">");
      if(label != null)
         out.write("<xforms:label>" + label +
  "</xforms:label>");
      else
         out.write("<xforms:label>"+"Enter Secret:"+
  "</xforms:label>");
      if(hint != null)
         out.write("<xforms:hint>" + hint +
  "</xforms:hint>");
      out.write("</xforms:secret>");
    }//encodeEnd
  }//UIInputSecret
```

The `UIInputSecret` class extends the `UIInputBox` class and overrides the `encodeEnd()` method, which renders the markup for the XForms `secret` element.

After implementing the `UIInputSecret` class, you can use this component with any JSF application after an entry in the faces-config.xml file:

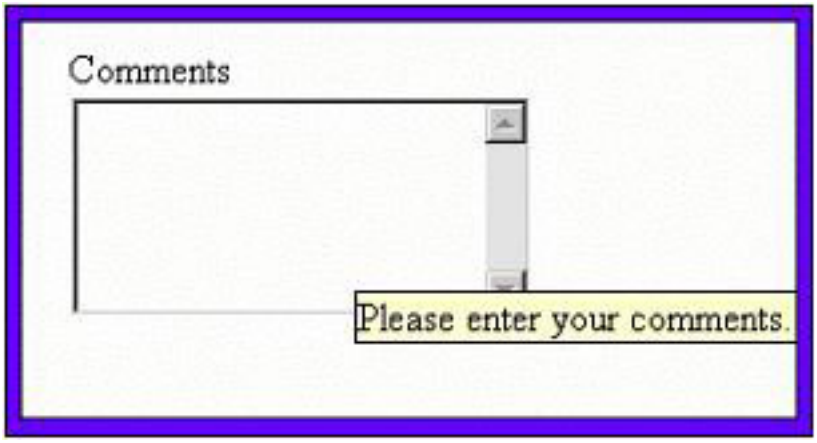
```
<component>
  <component-type>InputSecret</component-type>

<component-class>xforms_jsf.UIInputSecret</component-class>
</component>
```

Implementing the xforms-jsf:inputTextarea component

The `xforms-jsf:inputTextarea` component provides the user with an edit box to enter multiple lines of text and renders the following markup:

```
<xforms:textarea ref="comments" model="optModel"
  xmlns:xforms=  "http://www.w3.org/2002/xforms">
  <xforms:label>Comments</xforms:label>
  <xforms:hint>Please enter your comments.</xforms:hint>
</xforms:textarea>
```

In the XForms browser, the above markup looks like the following:



The JSP author codes the following elements to use the component in the JSP page.

```
<xforms-jsf:inputTextarea value="#{customerData.comments}"
  ref="comments" model="optModel">
</xforms-jsf:inputTextarea>
```

The component saves the text entered by the user in the model bean property passed in the `value` attribute.

The TLD file entry for the `xforms-jsf:inputTextarea` tag looks like the following markup:

```
<tag>
  <name>inputTextarea</name>
  <tag-class>xforms_jsf.InputTextAreaTag</tag-class>
  <!-- Remaining is same as implemented for inputText-->
</tag>
```

In the above TLD file, we mentioned `<tag-class>xforms_jsf.InputTextAreaTag</tag-class>`. Now let's implement this tag handler class.

The attributes for the `inputTextarea` tag are the same as defined for `inputText`. We extended the `InputTextAreaTag` tag handler class from the `InputBoxTag` class implemented for the `inputText` tag. The following code shows the implementation of the `InputTextAreaTag` class:

```
public class InputTextAreaTag extends InputBoxTag{
```

```
public void setProperties(UIComponent component){
  super.setProperties(component);
  UIInputTextArea uita = (UIInputTextArea)component;
  FacesContext fc = FacesContext.getCurrentInstance();
  Application app = fc.getApplication();
  if(required != null) {
    if(UIComponentTag.isValueReference(required)) {
      ValueBinding vb = app.createValueBinding(required);
      uita.setValueBinding("required", vb);
    }
    else{
      boolean boolRequired = (new Boolean(required)).booleanValue();
      uita.setRequired(boolRequired);
    }
  }
  if(valueChangeListener != null) {
    if(UIComponentTag.isValueReference(valueChangeListener)){
      Class args[] = {
         javax.faces.event.ValueChangeEvent.class
      };
      MethodBinding mb =
        app.createMethodBinding(valueChangeListener, args);
      uita.setValueChangeListener(mb);
    }
  }
  if(immediate != null) {
    if(UIComponentTag.isValueReference(immediate)) {
      ValueBinding vb = app.createValueBinding(immediate);
      uita.setValueBinding("immediate", vb);
    }
    else{
      boolean boolImmediate =
       (new Boolean(immediate)).booleanValue();
      uita.setImmediate(boolImmediate);
    }
  }
  if(value != null) {
    if(UIComponentTag.isValueReference(value)) {
      ValueBinding vb = app.createValueBinding(value);
      uita.setValueBinding("value", vb);
    }
    else
      uita.setValue(value);
  }
  if(ref != null)
    uita.getAttributes().put("ref", ref);
  if(bind != null)
    uita.getAttributes().put("bind", bind);
  if(hint != null)
    uita.getAttributes().put("hint", hint);
  if(label != null)
    uita.getAttributes().put("label", label);
  if(model != null)
    uita.getAttributes().put("model", model);
  if(navindex != null)
    uita.getAttributes().put("navindex", navindex);
  if(accesskey != null)
    uita.getAttributes().put("accesskey", accesskey);
  if(inputmode != null)
    uita.getAttributes().put("inputmode", inputmode);
  if(appearance != null)
    uita.getAttributes().put("appearance", appearance);
  if(incremental != null)
```

```
        uita.getAttributes().put("incremental", incremental);
    }//setProperties
  }//InputTextAreaTag
```

Now let's discuss the `UIInputTextArea` component class associated with `xforms-jsf:inputTextarea`:

```
public class UIInputTextArea extends UIInputBox{
  public void encodeEnd(FacesContext fc) throws IOException{
    if(fc == null)
       throw new NullPointerException();
    if(!isRendered())
      return;
    String ref         = (String) getAttributes().get("ref");
    String bind        = (String) getAttributes().get("bind");
    String hint        = (String) getAttributes().get("hint");
    String model       = (String) getAttributes().get("model");
    String label       = (String) getAttributes().get("label");
    String navindex    = (String) getAttributes().get("navindex");
    String inputmode   = (String) getAttributes().get("inputmode");
    String accesskey   = (String) getAttributes().get("accesskey");
    String appearance  = (String) getAttributes().get("appearance");
    String incremental = (String) getAttributes().get("incremental");
    ResponseWriter out = fc.getResponseWriter();
    out.write("<xforms:textarea ref=\"" + ref +"\"");
    if(bind != null)
       out.write(" bind=\""+ bind + "\"");
    if(model != null)
       out.write(" model=\""+ model + "\"");
    if(navindex != null)
       out.write(" navindex=\""+ navindex + "\"");
    if(inputmode != null)
       out.write(" inputmode=\""+ inputmode + "\"");
    if(accesskey != null)
       out.write(" accesskey=\""+ accesskey + "\"");
    if(appearance != null)
       out.write(" appearance=\""+ appearance + "\"");
    if(incremental != null)
       out.write(" incremental=\""+ incremental + "\"");
    out.write(" xmlns:xforms=\"http://www.w3.org/2002/xforms\">");

    if(label != null)
       out.write("<xforms:label>" + label +
"</xforms:label>");
    else
       out.write("<xforms:label>"+" Enter Message
"+"</xforms:label>");
    if(hint != null)
       out.write("<xforms:hint>" + hint +
"</xforms:hint>");
    out.write("</xforms:textarea>");
  }//encodeEnd
}//UIInputTextArea
```

The `UIInputTextArea` class extends the `UIInputBox` class. The `UIInputTextArea` class only overrides the `encodeEnd()` method, which renders the markup for the XForms text area element.

The `UIInputBox` class renders a text box, which takes a single line string as

input. On the other hand, the `UIInputTextArea` class renders a text area, which can take multiple lines of text as input. Whether it is a single line or a number of lines, it is always a string (a multiple line string contains "\r" and "\n" characters at the end of each line). We can treat both types of strings the same. That's why the `UIInputTextArea` class does not override the `decode( )` method of the parent class.

Now we can use the `UIInputTextArea` component with any JSF application after including the component entry in the faces-config.xml file:

```
<component>
  <component-type>InputSecret</component-type>

<component-class>xforms_jsf.UIInputSecret</component-class>
</component>
```

By implementing the `UIInputTextArea` component, our XForms-JSF tab library is complete. In the next section, we will see how to compile a tag library in a JAR file.

How to distribute tag libraries

To distribute a tag library, you have to compile its components in a JAR file. The JAR file (compiled tag library) should contain the following information:

- The compiled tag handler, component, and model bean classes of a tag library in their respective packages

- The configuration files (faces-config.xml) and TLD file in a folder named META-INF

To create the JAR file, place the compiled classes and META-INF folder in a single folder (such as the XForms-JSF folder, which you will find when you unzip the section7.zip file; see Resources on page200 .

Run the following command in the command prompt:

```
d:\XForms-JSF>jar cvf xforms-jsf.jar *.*
```

This command creates a JAR file named xforms-jsf.jar, which contains the META-INF folder (which in turn contains the TLD and configuration files) and the class files available in the XForms-JSF folder.

The xforms-JSF.jar file is available in the section7.zip file; see Resources on page200 .

Now the tag library is ready to be used with any JSF application. Place this JAR

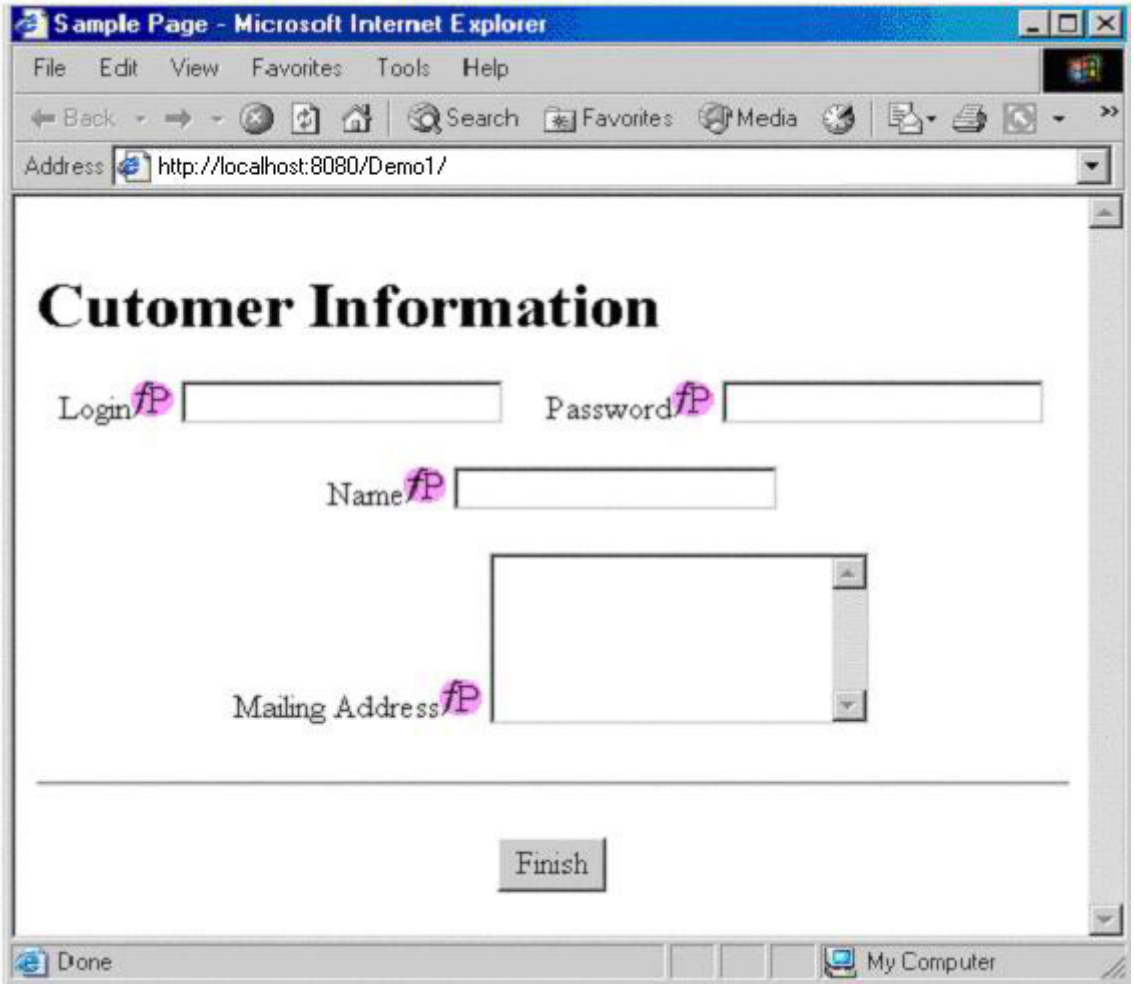
file in the WEB-INF\lib folder of your application. In the next section, we will develop a sample application to show you how to use the JAR file in a JSF application.

A sample JSP page to try the XForms-JSF tag library

To show how to use the tag library in an application, I have written a simple JSP page, which includes different components from our tag library:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html
  xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xforms="http://www.w3.org/2002/xforms">
 <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 <%@ taglib uri=" http://afictitiousshoppingcart.com/XForms-JSF"
  prefix="xforms-jsf" %>
 <f:view>
  <head>
    <xforms-jsf:model value="#{customerData.model}"/>
  </head>
  <body>
   <h1>Customer Information</h1>
   <xforms-jsf:inputText ref="login" value="#{customerData.login}"
     label="Login" hint="Please enter your login name."/>
   <xforms-jsf:inputSecret ref="password"
value="#{customerData.password}"
     label="Password" hint="Please enter your password."/>
   <xforms-jsf:inputText ref="name" value="#{customerData.name}"
label="Name"
     hint="Please enter your full name."/>
   <xforms-jsf:textarea ref="address"
value="#{customerData.address}"
     label="Mailing Address" hint="Please enter your address."/>
   <xforms-jsf:commandButton label="  Finish  ">
   </xforms-jsf:commandButton>
  </body>
 </f:view>
</html>
```

The output of the above JSP page is shown in the following screenshot:



To try the XForms-JSF tag library and JSP page, unzip the section7.zip file in the source code download for this tutorial; see Resources on page200 .

The section7.zip file contains two folders named Demo1 and src, and a couple of files named xforms-jsf.jar and Demo1.war. The Demo1 folder contains the sample JSP page, and the src folder contains the complete source code for our XForms-JSF tag library we created in this section.

Now simply deploy the Demo1.war file in your application server. Be sure to use following URL in the address bar of your XForms browser:

```
http://localhost:8080/Demo1
```

As a result, the JSP page shown in the above screenshot displays in your XForms browser.

Section 8. Designing the XForms-JSF shopping cart

What will our XForms-JSF shopping cart demonstrate?

The XForms-JSF shopping cart application tests the concepts discussed so far. We will demonstrate the architecture of a real-world XForms-JSF application and demonstrate how to use a few tags we have developed in the past two sections. In this section, we will develop a small shopping cart-specific tag library and show how to use the tag library to fit the requirements of a shopping cart.

Recall the XForms-JSF shopping cart application described in A sample XForms application on page 5. Our XForms-JSF shopping cart application is the JSF implementation of exactly the same XForms shopping cart.

Components of the XForms-JSF shopping cart application

As mentioned in Components of a JSF application on page 29, a JSF application can have different types of components. Our shopping cart has the following components:

- **JSF components:**

  Our XForms-JSF shopping cart uses some components from the JSF core tag library (for instance, `f:actionListener`, `f:selectItems`, and so on). We will also use some components from the XForms-JSF tag library developed in XForms-JSF integration strategy on page 80 and XForms-JSF tag library on page 118.

  We will also develop three JSF components (`xcart:category`, `xcart:product`, and `xcart:cart`) for our shopping cart. The `xcart:category` tag renders the list of categories and products as buttons in the catalog view. The `xcart:product` tag renders the details of an individual product in a tabular format. The `xcart:cart` tag displays the details of products in the cart.

- **JSP pages:**

  We will have the following six JSP pages in our shopping cart application:

  - **catalogView.jsp:** The first page with which a visitor starts browsing our shopping cart. It displays the list of categories and products in the catalog view.

- **productView.jsp:** Appears when the user clicks a product in the catalogView.jsp page. It shows the features of the clicked product and a button to add the product in the cart. This page also allows the user to select from available optional features of a product.

- **cartView.jsp:** Appears when the user clicks the Add to cart button in the productView.jsp page. This page shows a list of all the products the user put in the cart, with buttons to remove and edit them. The cartView.jsp page also shows a Buy button to buy the contents of the cart.

- **editProductView.jsp:** Appears when the user clicks the Edit product button in the cart. This page provides the user with a facility to edit a particular product added in the cart.

- **checkout.jsp:** Appears when the user clicks the Buy button in the cartView.jsp page. This page takes the user's bio data and provides a Checkout button.

- **thankyou.jsp:** Appears when the user clicks the Checkout button in the checkout.jsp page. It's the last page for our XForms shopping cart.

- **Model beans to hold application data:**

  Our shopping cart has the following three model beans to store the application data:

  - **CategoryData:** Holds the data of the complete catalog. As the user browses through the catalog view, he clicks on various categories. When he clicks on any category, that category becomes the selected category. The `CategoryData` model bean keeps track of the current category.

  - **ProductData:** Holds the data of a product (name, ID, description, price, and features).

  - **CartData:** Wraps cart data -- the list of all products the user added to the cart.

- **Event handler classes:**

  We also need to implement two action event handler classes to handle the events generated by the `UICategory` and `UICart` components. The action handlers prepare the model beans according to the action performed.

  For example, when the user clicks a product or category in the catalog view, an action event will occur. The `UICategoryActionListener` action listener handles this action event. Similarly, when a user clicks the Edit or Remove button against a product, `UICartActionListener` handles the generated action event.

The interaction of components in our shopping cart

In the next few panels, we will elaborate on the interaction between the different components of our shopping cart application for the following actions:

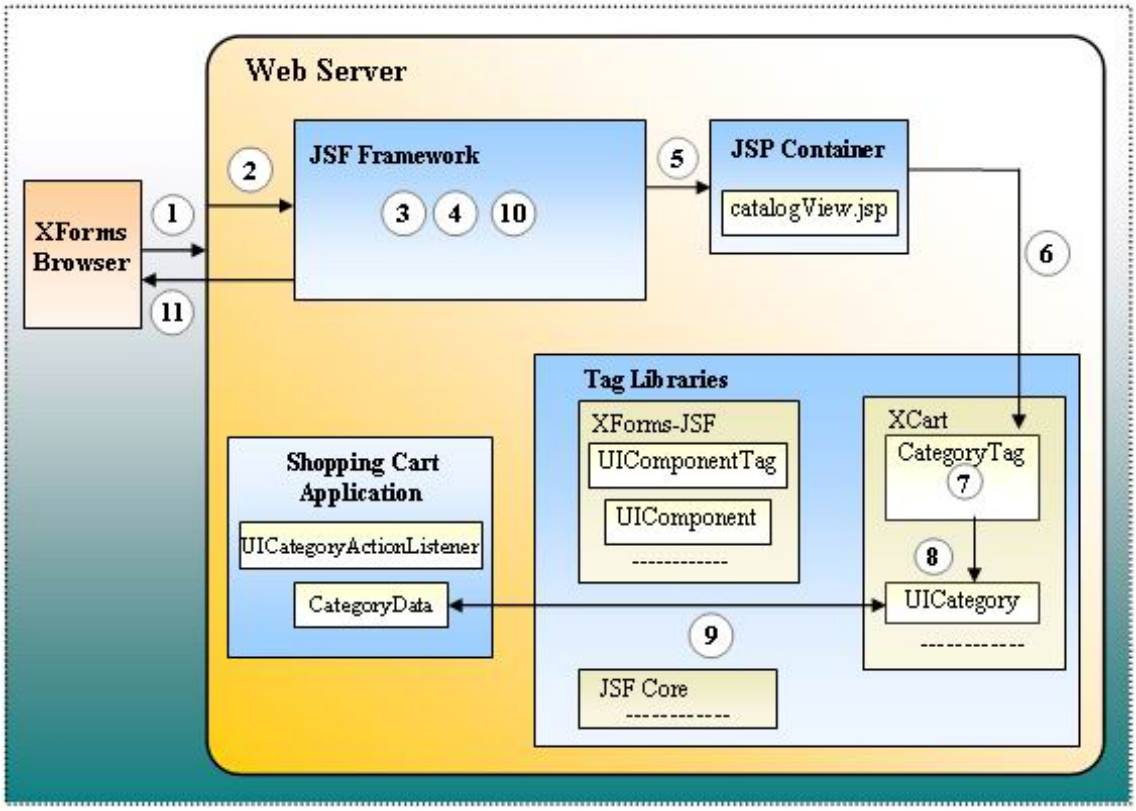
- Rendering the catalog data (categories and products names)
- Rendering product data
- Adding a product to the cart
- Editing a product entry from the cart
- Saving an edited product entry to the cart

Each action represents a sequence of events, which we will explain in the next five sections. The explanations of these actions act as a high-level view of our shopping cart application and will provide a practical example of the concepts we have learned so far.

Rendering the catalog data (catalog view)

The first job a shopping cart application does is generate a list of the products and categories.

Recall the Views of the shopping cart on page 7, where we described the rendering of catalog data as a catalog view. The following diagram shows the sequence of events that occur inside our XForms-JSF shopping cart application while rendering the catalog view:



The events of the figure are explained below. Each step in the following discussion is marked as a number in the figure shown above:

1. Suppose a user requests the following URL:

   ```
   www.aFictitiousShoppingCart.com
   ```

2. The Web server receives the request and invokes the JSF framework.

3. Now the JSF framework initializes the model beans defined in the faces-config.xml file. All the beans in our XForms shopping cart application are in `session` scope, so a separate set of beans is instantiated for every session. The first model bean our shopping cart application uses is the `CategoryData` class. The `CategoryData` constructor is built in such a manner that it prepares itself with the catalog data when it is initialized. Note that the other model beans in our shopping cart application are initialized, but they are not prepared for use with components. The user interactions with our shopping cart prepare the remaining model beans.

4. After initializing the model beans, the JSF framework starts the request processing life cycle of the catalogView.jsp page. It first checks whether the root of the JSF component (the `UIViewRoot` object) exists. Because it is the first request for the page, the root does not exist, so it creates a new `UIViewRoot` object and stores it in the `FacesContext` object.

5. The JSF framework dispatches the request to the JSP container (that executes the JSP page).

6. The JSP container goes through the catalogView.jsp page. When it comes across a JSF tag in the catalogView.jsp, it finds its corresponding tag handler class from the TLD file and instantiates the tag class.

7. The tag handler class extracts the reference of the corresponding JSF component class by reading the faces-config.xml file.

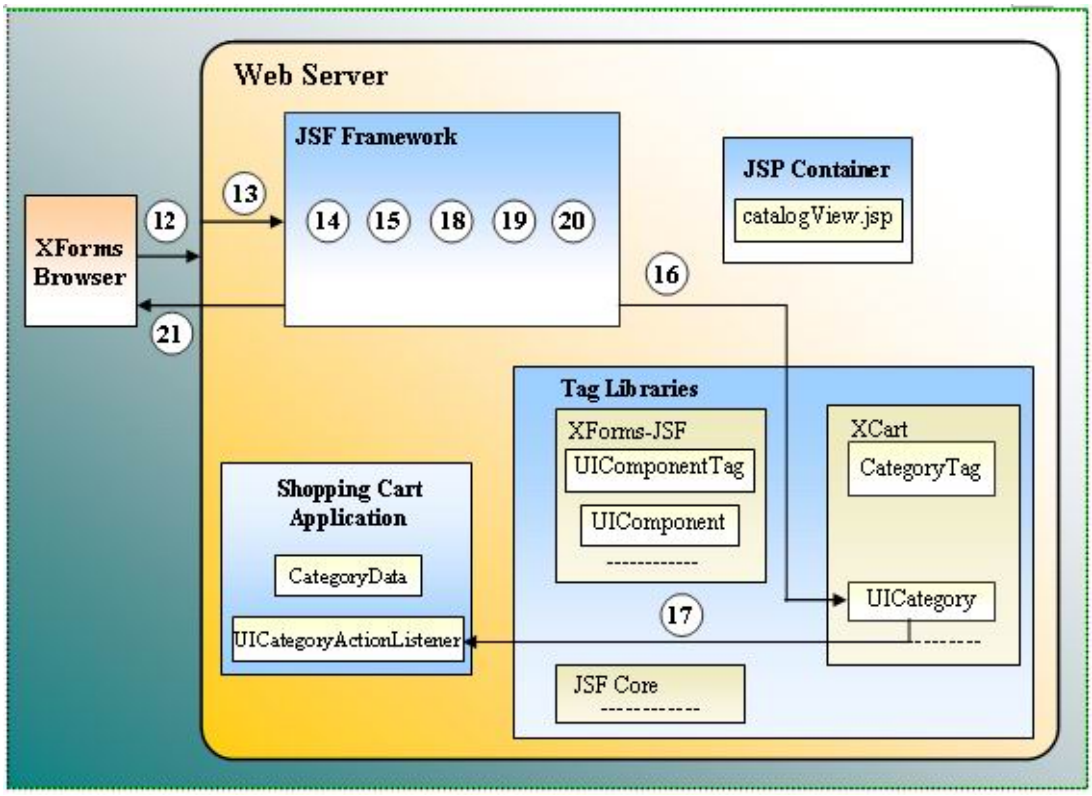
8. The tag handler class adds the component in the JSF component tree as a child of the root component (`UIViewRoot`) and calls the encode methods (`encodeBegin()`, `encodeChildren()`, and `encodeEnd()` methods) of the JSF component just added to the tree.

9. The encode methods extract the application-specific data from the `CategoryData` model bean, which we instantiated in step 3. This application-specific data contains categories and products to be rendered. Steps 6-9 repeat for each JSF tag in the catalogView.jsp page.

10. The JSF framework saves the component tree in the session.

11. Finally, it sends the generated markup back to the user. The output of this markup is already shown in Views of the shopping cart on page 7.

    Now the catalog view is in front of the user, displaying the list of categories and products. The user can click a category or a product. The following diagram shows the sequence of events that occur when the user clicks on a category:



The events of the figure are explained below:

12. The user clicks a category in the catalog view. The XForms browser wraps the information related to the category in the application-specific XML. Next, it wraps the XML structure in the body of a request for the same JSP page (catalogView.jsp) and sends the request to the Web server.

13. The Web server receives the request and invokes the JSF framework.

14. All the model beans in our XForms-JSF shopping cart are in `session` scope, so they will not be initialized this time. All the model beans will maintain their previous states.

15. The JSF framework checks whether the root of the component tree exists, and this time it finds the `UIViewRoot` object. It sets the `UIViewRoot` component in the `FacesContext` object as the root of the component tree. (Recall step 6, where we added all components into the component tree.)

16. The JSF framework iterates through the JSF component tree and calls the `decode()` method of each JSF component in the tree. The `decode()` method of the `xcart:category` component decodes the request and extracts the ID of the clicked item. Then the `decode()` method fires an action event.

17. The action event handler class associated with the `xcart:category` JSF tag catches the event. The action listener checks whether the ID belongs to a category or a product. In this case, the ID is of a category, so it tells the `CategoryData` model bean which category was clicked. The `CategoryData` model bean stores this information for future use.

18. Next, the JSF framework invokes the model bean method specified in the `action` property of the component. This method call returns a string value. The JSF framework searches this string value in the `navigation-rule` written in the application configuration file (faces-config.xml) for the catalogView.jsp page. If the contents of the `from-outcome` child of any of the `navigation-case` element matched with the string value, the JSF framework returns the JSP page specified in the accompanying `to-view-id` element. In this case, the user clicked a category, which means he just wants to browse deeper into the catalog view. The same categoryView.jsp page handles this request. The event handler classes will not change the view ID.

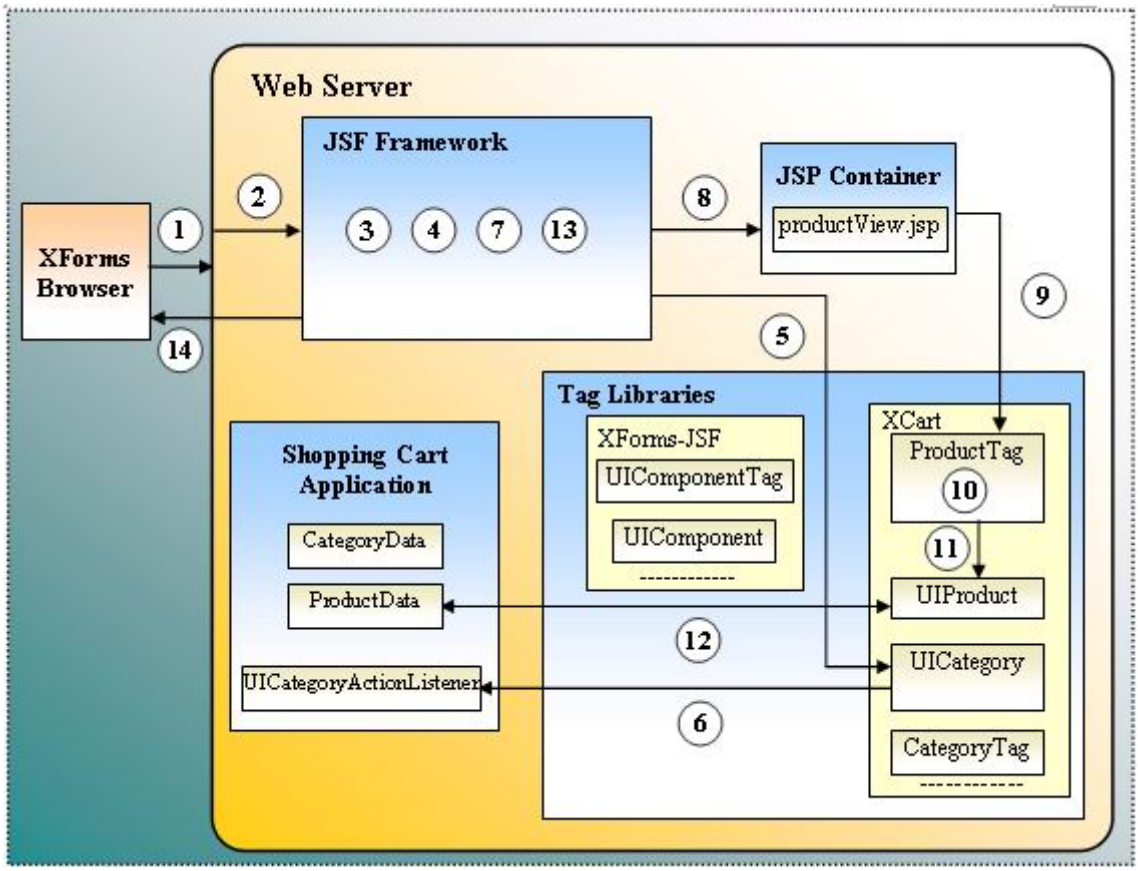
19. Next, the JSF framework calls the encoding method of each component in the tree one by one.

20. It then saves the component tree.

21. Finally, the JSF framework sends back the response to the user.

Rendering the product data (product-specification view)

Clicking a product in the catalog view brings up the product-specification view. The productView.jsp page shows product details like product name, price, description, features, and optional features, and allows the user to select them. The following diagram shows the sequence of events that occur while rendering the product-specification view:



The events of the figure are explained below:

1. The user clicks a product button in the catalogView.jsp page.

2. The Web server receives the request and invokes the JSF framework.

3. As you know, the model beans in our XForms-JSF shopping cart are in `session` scope, so they will not be initialized this time. All the model beans will maintain their previous states.

4. The JSF framework checks whether the root of the component tree exists, and this time it finds the `UIViewRoot` object. It sets the `UIViewRoot` component in the `FacesContext` object as the root of the component tree.

(Recall step 6, where we added all components into the component tree.)

5. The JSF framework iterates through the JSF component tree and calls the `decode()` method of each JSF component in the tree. The `decode()` method of the `xcart:category` component decodes the request and extracts the ID of the clicked item. Then the `decode()` method fires an action event.

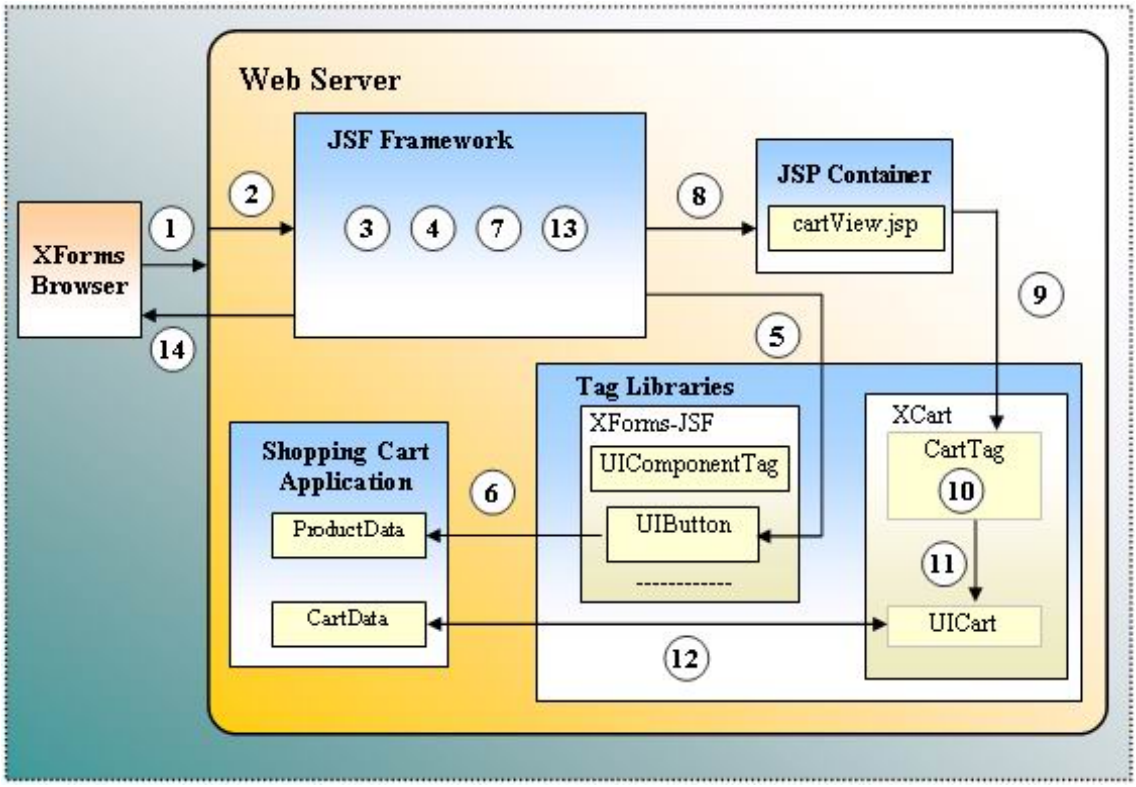
6. The action event handler class associated with the `xcart:category` JSF tag catches the event. The action listener checks whether the ID belongs to a category or a product. In this case, the ID is of a category, so it tells the `CategoryData` model bean which category was clicked. The `CategoryData` model bean stores this information for future use.

7. Similar to step 18 from the previous panel, but this time the JSF framework invokes the productView.jsp page.

8. Steps 8-14 are the same as steps 5-11 discussed in the previous panel, except that the productView.jsp page and `ProductData` model bean are used instead of the catalogView.jsp page and `CategoryData` model bean respectively.

Adding a product to the shopping cart (cart view)

The following diagram shows the sequence of events that occur when a user clicks the Add to cart button in the product-specification view.



The events of the figure are explained below:

1. The user clicks the Add to cart button in the product-specification view. The XForms browser gathers the information about the product to be added to the cart, wraps this information in application-specific XML, and sends the request to the Web server.

2. The Web server receives the request and invokes the JSF framework.

3. As you know, the model beans in our XForms-JSF shopping cart are in `session` scope, so they will not be initialized this time. All the model beans will maintain their previous states.

4. The JSF framework checks whether the root of the component tree exists, and this time it finds the `UIViewRoot` object. It sets the `UIViewRoot` component in the `FacesContext` object as the root of the component tree. (Recall step 6, where we added all components into the component tree.)

5. The JSF framework iterates through the JSF component tree and calls the `decode()` method of each JSF component in the tree. The `decode()` method of the JSF component against the `XForms-JSF:commandButton` tag decodes the request, extracts the ID of the clicked button and fires an action event.

6. The action event handler method passed in the `actionListener` attribute of the `XForms-JSF:commandButton` JSF tag handles the event. The action handler adds the product in the `CartData` model bean, which holds

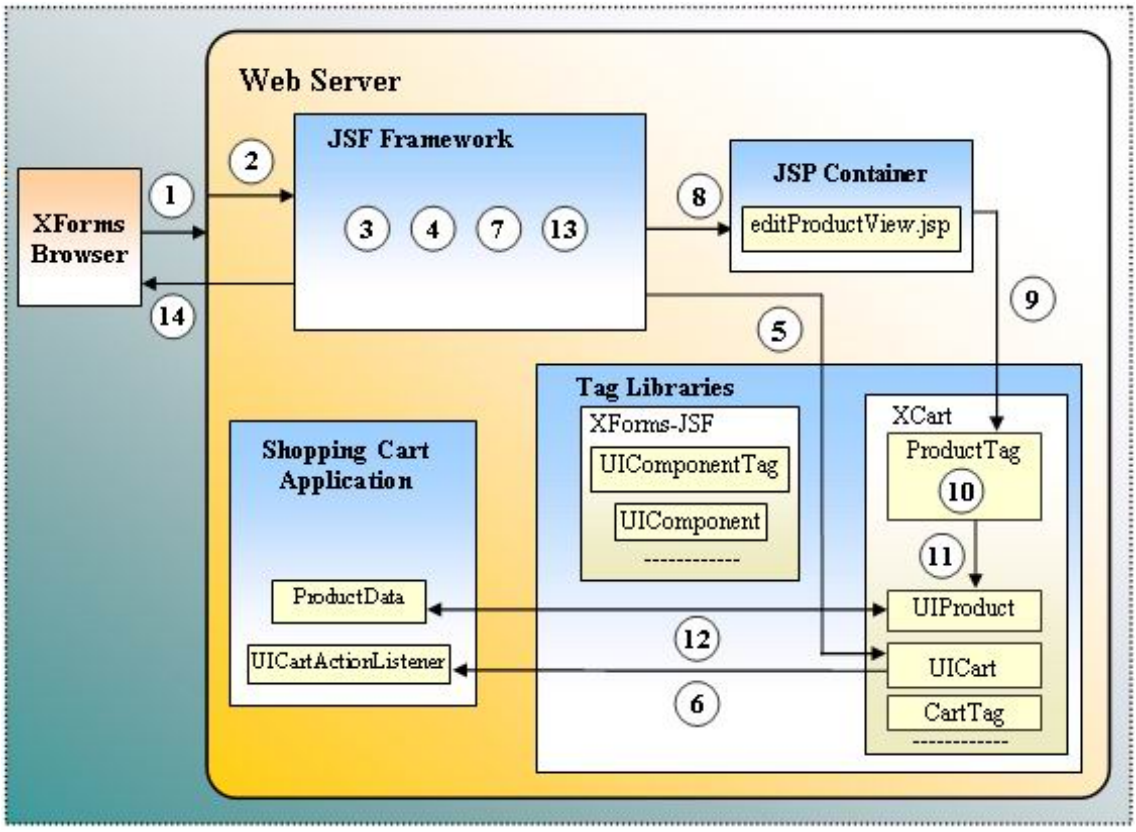
the data of a cart.

7. Now the JSF framework checks the faces-config.xml file and invokes the cartView.jsp page.

8. Steps 8-14 are the same as steps 5-11 discussed in Rendering the catalog data (catalog view) on page 150, except that the cartView.jsp page and `CartData` model bean are used instead of the catalogView.jsp page and `CategoryData` model bean, respectively.

Editing the product entry from cart (edit product view)

The cart view contains an Edit product entry button with each product in the cart. The following diagram shows the sequence of events that occurs when a user clicks that button:



The events of the figure are explained below:

1. The user clicks the Edit button in the cart view. The XForms browser wraps the button-identifying information in the application-specific XML data. The XForms-JSF component sitting on the server side uses this button-identifying information to check which button was clicked by the user.

2. The Web server receives the request and invokes the JSF framework.

3. The model beans in our XForms-JSF shopping cart are in `session` scope, so they will not be initialized this time. All the model beans will maintain their previous states.

4. The JSF framework checks whether the root of the component tree exists, and this time it finds the `UIViewRoot` object. It sets the `UIViewRoot` component in the `FacesContext` object as the root of the component tree. (Recall step 6, where we added all components into the component tree.)

5. When the `decode()` method of the JSF component against the `xcart:cart` JSF tag is called, it decodes the request and extracts the ID of the clicked button and fires an event.

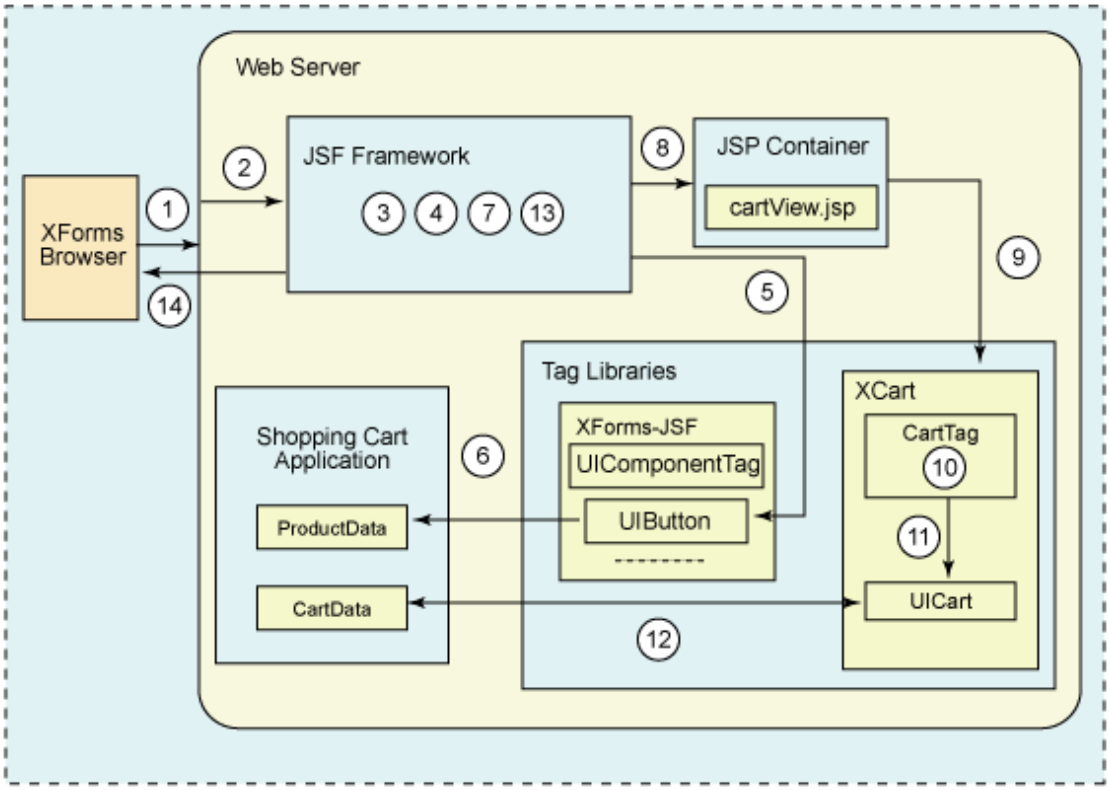
6. The action event handler class (`UICartActionListener`) associated with the `xcart:cart` JSF tag handles the event. The action handler adds the product in the model bean, which holds the data of a cart (`CartData`).

7. The JSF framework checks the `navigation-rule` and invokes the editProductView.jsp page.

8. Steps 8-14 are same as steps 5-11 discussed in Rendering the catalog data (catalog view) on page 150, except that the editProductView.jsp page and `ProductData` model bean are used instead of the catalogView.jsp page and `CategoryData` model bean, respectively.

Saving edited product entries to the cart

The edit product view renders the product for editing, which is accompanied by an Edit cart entry button. The user edits the product entry in the edit cart view. The following sequence of events occurs when saving the edited product:



The events of the figure are explained below. Each step in the following discussion is marked as a number in the figure shown above:

1. A user clicks the Save edited product button in the editProductView.jsp page.

2. The Web server receives the request and invokes the JSF framework.

3. The model beans in our XForms-JSF shopping cart are in `session` scope, so they will not be initialized this time. All the model beans will maintain their previous states.

4. The JSF framework checks whether the root of the component tree exists, and this time it finds the `UIViewRoot` object. It sets the `UIViewRoot` component in the `FacesContext` object as the root of the component tree. (Recall step 6, where we added all components into the component tree.)

5. The JSF framework iterates through the JSF component tree and calls the `decode()` method of each JSF component in the tree. The `decode()` method of the JSF component against the `xforms-jsf:commandButton` tag decodes the request, extracts the ID of the clicked button, and fires an action event.

6. The action event handler method passed in the `actionListener` attribute of the `xforms-jsf:commandButton` JSF tag handles the event. The action handler replaces the product-edited product in the `CartData` model

bean, which holds the data of a cart.

7. The JSF framework checks the faces-config.xml file and invoke the cartView.jsp page.

8. Steps 8-14 are the same as steps 5-11 discussed in Rendering the catalog data (catalog view) on page 150, except that the cartView.jsp page and `ProductData` model bean are used instead of the catalogView.jsp page and `CategoryData` model bean, respectively.

Summary

In this section, we have provided the high-level view of our XForms-JSF shopping cart application. We discussed the components required to develop the application (JSP pages, model beans, event handlers, etc.). We also explained how these components interact with each other.

In the next section, we will implement the components introduced in this section and complete the implementation of our XForms-JSF shopping cart application. The next section also explains all the low-level details of the implementation.

Section 9. Implementing the XForms-JSF shopping cart

Implementing the NameValuePair, Category, and Product classes

In Components of the XForms-JSF shopping cart application on page 148, we discussed three JSF tags: `xcart:category`, `xcart:product`, and `xcart:cart`. These tags form the XForms-based shopping cart tag library. We wish to make this tag library independent of our shopping cart application to ensure that you can integrate our shopping cart tag library into your own shopping cart application.

To make our shopping cart tag library independent of the shopping cart application, we need to implement three classes: `NameValuePair`, `Category`, and `Product`. These classes will be part of the shopping cart tag library. Application-specific model beans initialize and use these classes.

The `NameValuePair` class just wraps a name-value pair. The `Category` class holds the subcategories and products corresponding to a particular category. The `Product` class holds the details of an individual product (name, ID, description, price, etc.).

Let's look at the implementation details of the `NameValuePair`, `Category`, and `Product` classes in detail.

The `NameValuePair` class contains two fields -- name and value -- and their respective setter and getter methods. The following code shows the implementation of the `NameValuePair` class:

```
public class NameValuePair{
  protected String name;
  protected String value;
  public void setName(String name) {
    this.name = name;
  }
  public String getName(){
    return name;
  }
  public void setValue(String value){
    this.value = value;
  }
  public String getValue(){
    return value;
  }
}//NameValuePair
```

Many of the classes in our shopping cart tag library and shopping cart application use the `NameValuePair` class.

Any application that wants its data to be rendered using the `xcart:category` tag instantiates a `Category` object and populates it with application-specific

data. The `xcart:category` component uses this `Category` object to fetch the application-specific data.

The `Category` class holds the category data in `id`, `products`, `ancestors`, and `subcategories` properties.

The `id` property of the `Category` class is a `String` object that holds the ID of the current category. The `products`, `ancestors`, and `subCategories` properties are arrays of `NameValuePair` objects. The `products` property contains the information of each product in the category. The `ancestors` property tracks the ancestors of the current category. The `subCategories` property stores the list of subcategories in the current category.

The following code shows the implementation of the `Category` class:

```
public class Category{
  protected String id;
  protected NameValuePair[] products;
  protected NameValuePair[] ancestors;
  protected NameValuePair[] subCategories;
  public void setAncestors(NameValuePair[] nameId) {
    ancestors = nameId;
  }
  public NameValuePair[] getAncestors(){
    return ancestors;
  }
  public void setSubCategories(NameValuePair[] nameId) {
    subCategories = nameId;
  }
  public NameValuePair[] getSubCategories(){
    return subCategories;
  }
  public void setProducts (NameValuePair[] nameId) {
    products = nameId;
  }
  public NameValuePair[] getProducts(){
    return products;
  }  public void setId(String id) {
    this.id = id;
  }
  public String getId(){
    return id;
  }
}//Category
```

Now let's discuss the implementation of the `Product` class.

The `Product` class holds the product data in seven properties:

- `id`: Stores the ID of a particular product the user clicked in the catalog view
- `name`: Stores the name of the product
- `price`: Stores the price of the product

- `description`: Stores the description of the product
- `features`: Stores the features of a product in name-value pair form
- `optional-features`: Stores the optional features of the product
- `selectedOptionalFeatures`: Stores those optional features the user selected in the product specification view

The following code shows the implementation of the `ProductData` class:

```
public class Product{
  protected String id;
  protected String name;
  protected String price;
  protected String[] options;
  protected String description;
  protected NameValuePair[] features;
  protected NameValuePair[] optionalFeatures;
  protected NameValuePair[] selectedOptionalFeatures;
  public void setName ( String name) {
    this.name = name;
  }
  public String getName (){
    return name;
  }
  public void setId (String id) {
    this.id = id;
  }
  public String getId(){
    return id;
  }
  public void setDescription ( String description) {
    this.description = description;
  }
  public String getDescription (){
    return description;
  }
  public void setPrice( String price) {
    this.price = price;
  }
  public String getPrice(){
    return price;
  }
  public void setFeatures(NameValuePair[] features) {
    this.features = features;
  }
  public NameValuePair[] getFeatures (){
    return features;
  }
  public void setSelectedOptionalFeatures(NameValuePair[]
selOptFeatures){
    this.selectedOptionalFeatures = selectedOptionalFeatures;
  }
  public NameValuePair[] getSelectedOptionalFeatures(){
    return selectedOptionalFeatures;
```

```
    }
    public void setOptionalFeatures (NameValuePair[] optionalFeatures){
      this.optionalFeatures = optionalFeatures;
    }
    public NameValuePair[] getOptionalFeatures (){
      return optionalFeatures;
    }
    public void setOptions (String[] options) {
      this.options = options;
    }
    public String[] getOptions(){
      return options;
    }
  }//Product
```

Now let's start implementing our sample XForms-JSF shopping cart. We have to implement three JSP pages (catalogView.jsp, productView.jsp, and cartView.jsp), three JSF components (`UICategory`, `UIProduct`, and `UICart`), two event handler classes (`UICategoryActionListener` and `UICartActionListener`), and three model beans (`CategoryData`, `ProductData`, and `CartData`).

The catalogView.jsp page, `UICategory` component, `UICategoryActionListener` event handler, and `CartData` model bean form a set. We will explain the working of this set in the next four sections. For the rest of the JSP pages, components, event handlers, and model beans, we will only explain the differences from the first set.

Using the xcart:category tag to generate the catalog view

Look at the following catalogView.jsp page, which generates the catalog view:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html
  xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xforms="http://www.w3.org/2002/xforms">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://afictitiousshoppingcart.com/XCart"
  prefix="xcart" %>
<%@ taglib uri="http://afictitiousshoppingcart.com/XForms-JSF"
  prefix="xforms-jsf" %>
<f:view>
 <head>
   <xforms-jsf:model value="#{categoryData.model" />
 </head>
 <body>
   <xcart:category value="#{categoryData.category}"
     action="#{categoryData.getAction}">
     <f:action_listener type="model.UICategoryActionListener"/>
   </xcart:category>
   <xforms-jsf:commandButton label="Show Cart" immediate="true"
     action="#{categoryData.getAction}"
```

```
      actionListener="#{categoryData.showCartView}">
    </xforms-jsf:commandButton> </body>
  </body>
 </f:view>
 </html>
```

Note the following points:

- We have used the `xforms-jsf:model` tag from the XForms-JSF tag library in the `head` element. We explained this tag in Implementing the xforms-jsf:model component on page 83.

- We have included the `xcart:category` tag in the `body` element, which contains two attributes: `value` and `action`.

- The JSP author provides the `xcart:category` tag with a `value` attribute. The `value` attribute contains the property of the model bean (`categoryData.category`), which contains the application data associated with the catalog view. The `categoryData.category` property is an object of the `Category` class explained earlier. Note that the model bean (categoryData) can be an object of any class, but the property that contains the application data should be an object of the `Category` class.

- The method of the model bean specified in the `action` attribute ("categoryData.getAction") controls navigation. In Navigation process on page 75, we explained how navigation works and how the JSF framework uses the `action` attribute for navigation.

- We have associated an action listener (`UICategoryActionListener`) with the `xcart:category` component. When the user clicks any category or product in the catalog view, the `UICategoryActionListener` class handles the generated action event, which we will discuss in Implementing the UICategoryActionListener class on page176 .

- We have used the `xforms-jsf:commandButton` tag to show the cart view. The `actionListener` attribute of the `xforms-jsf:commandButton` tag refers to the event handling method of the model bean. When the user clicks this button, the JSF framework calls this method to handle the generated action event.

Implementing the UICategory component

The `xcart:category` tag renders the list of categories and products available in the catalog view. Each category or product is rendered as a button. If the user clicks on any category, its subcategories and products are displayed. If the user clicks on a product, the product-specification view appears to show the details of the product.

The first step toward the development of any JSF tag is to make an entry in the TLD file. The TLD entry for the `xcart:category` tag looks like the following code:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib>
  ......
  <tag>
    <name>category</name>
    <tag-class>xcart.CategoryTag</tag-class>
    <attribute>
       <name>value</name>
       <required>false</required>
       <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
       <name>id</name>
       <required>false</required>
       <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
       <name>action</name>
       <required>false</required>
       <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>binding</name>
      <required>false</required>
      <rtexprvalue> false </rtexprvalue>
    </attribute>
    <attribute>
      <name>actionListener</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>immediate</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>rendered</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
  .....
  <!--other tag instances-->
<taglib>
```

In the above TLD entry, the `name` element inside the `tag` element wraps the name of the JSF tag (category), and the `tag-class` element wraps the complete qualified name of the tag handler class (`xcart.CategoryTag`).

The `xcart:category` tag contains two attributes: `value` and `action`. The two attribute elements in the above TLD entry correspond to these two attributes.

Here's how we implement the `xcart.CategoryTag` tag handler class:

```
public class CategoryTag extends UIComponentTag {
  private String value;
  private String action;
  private String immediate;
  private String actionListener;
  public void setValue(Object value){
    this.value = value;
  }
  public Object getValue(){
    return this.value;
  }
  public void setAction(Object action) {
    this.action = action;
  }
  public Object getAction(){
    return this.action;
  }
  public String getComponentType(){
    return "category";
  }
  public String getRendererType(){
     return null;
  }
  public String getImmediate(){
    return immediate;
  }
  public void setImmediate(String newImmediate) {
     immediate = newImmediate;
  }
  public String getActionListener(){
     return actionListener;
  }
  public void setActionListener(String newActionListener) {
    actionListener = newActionListener;
  }
  public void setProperties(UIComponent component) {
    super.setProperties(component);
    UICategory catComp = (UICategory)component;

    FacesContext fc = FacesContext.getCurrentInstance();
    Application app = fc.getApplication();

    if(action != null)    {
      if(UIComponentTag.isValueReference(action)) {
        MethodBinding mb = app.createMethodBinding(action, null);
        catComp.setAction(mb);
      }
      else{
        MethodBinding mb = new ConstantMethodBinding(action);
        catComp.setAction(mb);
      }
    }
    if(value != null) {
      if(UIComponentTag.isValueReference(value))
      {
        ValueBinding vb = app.createValueBinding(value);
        catComp.setValueBinding("value", vb);
      }
      else
        catComp.setValue(value);
    }
    if(actionListener != null) {
```

```
        if(UIComponentTag.isValueReference(actionListener)) {
          Class args[] = {
            javax.faces.event.ActionEvent.class
          };
          MethodBinding mb =
            app.createMethodBinding(actionListener, args);
          catComp.setActionListener(mb);
        }
      }
      if(immediate != null) {
        if(UIComponentTag.isValueReference(immediate)) {
          ValueBinding vb = app.createValueBinding(immediate);
          catComp.setValueBinding("immediate", vb);
        }
        else{
          boolean boolImmediate =
            (new Boolean(immediate)).booleanValue();
          catComp.setImmediate(boolImmediate);
        }
      }
    }//setProperties
  }//CategoryTag
```

The `CategoryTag` class extends the `UIComponentTag` class to act as a tag class. Notice the following points from the `CategoryTag` class implementation above:

- The `getComponentType()` method returns the type of the component (category), which tells the JSF framework about the component class used with this tag. Later, we will implement the matching component class.

- The `getRendererType()` method returns null, which indicates that there is no renderer class associated with this JSF tag.

- There are two setter methods (`setValue()` and `setAction()`) in the tag class, which the JSF framework uses to pass the attribute values from the JSP page to the `CategoryTag` tag handler class.

- The `setProperties()` method is used to pass on the values of `value` and `action` attributes from the tag handler class to the matching component class.

Here's how to implement the component class associated with the `xcart:category` tag. Look at the following entry in the faces-config.xml file:

```
<?xml version="1.0"?>
<faces-config>
  .....
  <component>
    <component-type>category</component-type>
    <component-class>xcart.UICategory</component-class>
  </component>
  <!-- other component instances-->
</faces-config>
```

The `component-type` element wraps the type of the component (category). The name of the component class is specified in the accompanying `component-class` element (`xcart.UICategory`). Now we will implement this `UICategory` component class.

The `UICategory` component renders the categories and products as buttons. An action event is generated whenever the user clicks on a category or a product button. We will extend our `UICategory` component class from the `UICommand` class, which helps us handle action events.

The following code shows the fields and methods of the `UICategory` class:

```
public class UICategory extends UICommand{
  private String btnId = null;
  private void renderValueAsButton (ResponseWriter writer, String name,
     String id, String clientId) {
  }
  public void encodeBegin(FacesContext context)
    throws IOException {
  }
  private String getActionValueFromRequest(FacesContext fc, String ref,
    String tagName) {
  }
  private Object getModelBeanObject(String ref, FacesContext fc){
  }
  public void decode (FacesContext context)
    throws IOException {
  }
  public void setBtnId(String id) {
  }
  public String getBtnId(){
  }
}//UICategory
```

The `UICategory` class contains only one property (`btnId`), its setter and getter methods, two public methods (`encodeBegin()` and `decode()`), and three private helper methods (`renderValueAsButton()`, `getActionValueFromRequest()`, and `getModelBeanObject()`).

The `btnId` property contains the ID of the category or product button that the user clicks. When the `UICategory` component fires an action, the action event handling class calls the getter method of this property to identify the category or product the user clicked in the catalog view.

Let's discuss the methods of the `UICategory` class one by one.

The `renderValueAsButton()` method writes the markup to render a single product or category as a button. The `encodeBegin()` method of the `UICategory` class calls this method once for every category and product available in the catalog view.

The `renderValueAsButton()` method takes the following four parameters:

- **writer:** An object of the `ResponseWriter` class. The `renderValueAsButton()` method writes XForms markup on this

`ResponseWriter` object.

- **name:** The name of a product or category.
- **id:** The ID of a product or category.
- **clientId:** uniquely identifies a JSF component on the client side.

The code for the `renderValueAsButton()` method:

```
private void renderValueAsButton(ResponseWriter writer, String name,
  String id, String clientId){
  try{
    writer.write("<xforms:submit submission=\"submit\" >");

writer.write("<xforms:label>"+name.trim()+"</xforms:label>");
    writer.write("<xforms:action ev:event=\"DOMActivate\">");
    writer.write("<xforms:setvalue ref=\"action-performed\">"+
      clientId+"@"+id+"</xforms:setvalue>");
    writer.write("</xforms:action>");
    writer.write("</xforms:submit>");
  }
  catch (java.io.IOException ie){
    ie.printStackTrace();
  }
}//renderValueAsButton
```

The above code authors the following markup:

```
<xforms:submit submission="submit"
  xmlns:xforms="http://www.w3.org/2002/xforms"
  xmlns:ev="http://www.w3.org/2001/xml-events">
  <xforms:label>Computer</xforms:label>
  <xforms:action ev:event="DOMActivate">
    <xforms:setvalue
ref="action-performed">_id1@1</xforms:setvalue>
  </xforms:action>
</xforms:submit>
```

This markup renders a button. The whole markup is hard-coded in the `renderValueAsButton()` method, except the strings "Computer" and "_id1@1" (boldface in the markup above). The string "Computer" is the value of the second parameter (name). The second string "_id1@1" is the result of concatenating the third and fourth parameters (`id` and `clientId`). This data uniquely identifies a product or category. It consists of two parts separated by an "@" symbol. The first part (_id1, value of the third parameter) is the ID of the `UICategory` component. The second part (1, value of the fourth parameter) identifies a particular category or product in the catalog view.

Recall from Implementing the xforms-jsf:model component on page 83, where we explained the purpose of inserting an extra `action-performed` tag in the application-specific XML. The `renderValueAsButton()` method wraps the component and category (or product) ID in the `action-performed` tag. When

a user clicks the button corresponding to a particular category or product, the data in boldface (_id1@1) is sent back to the server, where it is used to identify which button was clicked.

The `encodeBegin()` method of the `UICategory` class is lengthy, so we will implement this method incrementally.

The `encodeBegin()` method first checks whether the `FacesContext` instance is `null`. If it is `null`, it simply throws an exception.

Then it extracts the component ID by calling the `getClientId()` method. Next, it gets the `ResponseWriter` object so it can render the markup for this component. These steps are shown in the following code:

```
public void encodeBegin(FacesContext context)
  throws IOException
{
if (context == null)
    throw new NullPointerException();
  String clientId = getClientId(context);
  ExternalContext externalContext = context.getExternalContext();
  ResponseWriter writer = context.getResponseWriter();
  .....
}//encodeBegin
```

Now the `encodeBegin()` method calls the `getModelBeanObject()` method. We explained how the `getModelBeanObject()` method works in Implementing the UISelect1 component on page 100. The `getModelBeanObject()` method returns an object of the `Category` class, which contains the data to be rendered, as shown in boldface below:

```
public void encodeBegin(FacesContext context)
  throws IOException
{
  if (context == null)
    throw new NullPointerException();
  String clientId = getClientId(context);
  ExternalContext externalContext = context.getExternalContext();
  ResponseWriter writer = context.getResponseWriter();
  categoryData = (Category) getModelBeanObject(context,
    value);
  .....
}//encodeBegin
```

Now we check if the `Category` instance returned by the `getModelBeanObject()` method is `null`. If it is `null`, it means that we have no data (that is, categories and products) to render in the catalog view.

```
public void encodeBegin(FacesContext context)
  throws IOException
{
  if (context == null)
    throw new NullPointerException();
  String clientId = getClientId(context);
```

```
    ExternalContext externalContext = context.getExternalContext();
    ResponseWriter writer = context.getResponseWriter();
    categoryData = (Category) getModelBeanObject(context,
      value);
    if (categoryData == null){
      writer.write ("catalog is empty...");
      return;
    }
    .....
  }//encodeBegin
```

Recall Views of the shopping cart on page 7, where we discussed that the catalog view contains three boxes: ancestors, subcategories, and products. Now we will code to display these boxes one by one.

The `encodeBegin()` method calls the `getAncestors()` method of the `Category` object to check whether the current category has any ancestors. If the category contains ancestors, the `encodeBegin()` method writes markup to display the ancestors in the ancestors box.

```
  public void encodeBegin(FacesContext context)
    throws IOException
  {
    ......
    if (categoryData == null){
      writer.write ("category data is null...");
      return;
    }
    if (categoryData.getAncestors() != null){
      writer.write("<table border=\"1\" width=\"100%\">");
      writer.write("<tr>");
      writer.write("<td>");
      NameValuePair[] ancestors = categoryData.getAncestors();
      if (ancestors != null){
        for(int x = 0; x < ancestors.length-1; x++){
          renderValueAsButton (writer, ancestors[x].getValue(),
            ancestors[x].getName(), clientId);
          writer.write(" > ");
        }
      }//if (ancestors != null)
      writer.write (ancestors[ancestors.length-1].getValue());
      writer.write("</td>");
      writer.write("</tr>");
      writer.write ("</table>");
    }//if (categoryData.getAncestors() != null)
    .....
  }//encodeBegin
```

Next it calls the `getSubCategories()` and `getProducts()` methods of the `Category` object to check for the subcategories and products in the current category. If it contains any subcategory or product, the `encodeBegin()` method writes markup to display subcategories and products as buttons in their respective boxes. For this purpose, it calls the `renderValueAsButton()` method explained earlier.

If the category does not contain a subcategory or product, the `encodeBegin()` method writes the markup to display that there is no subcategory or product in

the current category.

```
public void encodeBegin(FacesContext context)
  throws IOException
{
  ......
  if (categoryData.getAncestors() != null){
    ......
  }
  if (categoryData.getSubCategories() != null){
    writer.write("<table border=\"1\" width=\"100%\">");
    writer.write("Sub-Categories");
    writer.write("<tr>");
    writer.write("<td>");
    writer.write("<ul>");
    NameValuePair[] categories = categoryData.getSubCategories();
    for (int x =0; x < categories.length; x++){
      writer.write("<li>");
      renderValueAsButton (writer, categories[x].getValue(),
        categories[x].getName(), clientId);
      writer.write("</li>");
    }//for
    writer.write("</ul>");
    writer.write("</td>");
    writer.write("</tr>");
    writer.write("</table>");
  }//if
  else{
    NameValuePair[] ancestors = categoryData.getAncestors();
    writer.write("<table border=\"1\" width=\"100%\">");
    writer.write("Sub-Categories");
    writer.write("<tr>");
    writer.write("</tr>");
    writer.write("<tr><td>");
    writer.write("<h3> There is no sub-category in the ");
    writer.write (ancestors[ancestors.length-1].getValue());
    writer.write(" category. . . </h3> ");
    writer.write("</td></tr>");
    writer.write("<tr>");
    writer.write("</tr>");
    writer.write("</table>");
  }//else
  if (categoryData.getProducts() != null){
    writer.write("<table border=\"1\" width=\"100%\" >");
    writer.write("Products");
    writer.write("<tr>");
    writer.write("<td>");
    writer.write("<ul>");
    NameValuePair[] products = categoryData.getProducts();
    for (int x =0; x < products.length; x++){
      writer.write("<li>");
      renderValueAsButton(writer, products[x].getValue(),
        products[x].getName(), clientId);
      writer.write("</li>");
    }//for
    writer.write("</ul>");
    writer.write("</td>");
    writer.write("</tr>");
    writer.write("</table>");
  }//if
  else{
```

```
      NameValuePair[] ancestors = categoryData.getAncestors();
      writer.write("<table border=\"1\" width=\"100%\">");
      writer.write("Products");
      writer.write("<tr> ");
      writer.write("</tr>");
      writer.write("<tr><td>");
      writer.write("<h3> There is no product in the ");
      writer.write (ancestors[ancestors.length-1].getValue());
      writer.write(" category. . . </h3> ");
      writer.write("</td></tr>");
      writer.write("<tr>");
      writer.write("</tr>");
      writer.write("</table>");
    }//else
    writer.write("</tr>");
    writer.write("</table>");
  }//encodeBegin
```

Now let's see the implementation of the `getActionValueFromRequest()` method. This method parses the incoming request and extracts an identifier from the request (the identifier used to identify which category or product was clicked by the user). The decode method uses this method.

The `getActionValueFromRequest()` method takes the following parameters:

- **fContext:** The `FacesContext` instance.
- **ref:** Contains the model bean (`IncomingXMLInstanceRequest`) name and the property that contains the incoming XML request from the client side. See Parsing the incoming XML instance data on page 91 for details about this model bean.
- **tagName:** The name of the tag (`action-performed`) that contains the information about the button clicked by the user.

```
  private String getActionValueFromRequest(
                FacesContext fContext,
                String ref,
                String tagName){
    Document doc = (Document) getModelBeanObject(fContext, ref);
    String value = new String();
    if(doc != null) {
      NodeList nl = doc.getElementsByTagName(tagName);
      if (nl != null) {
        for (int i=0; i < nl.getLength(); i++){
          if(nl.item(i).getFirstChild().getNodeType() ==
            Node.TEXT_NODE ) {
            value = nl.item(i).getFirstChild().getNodeValue();
            if (value.indexOf('@') != -1) {
              int index = value.indexOf('@');
              String id = value.substring(0,index);
              if (id.equals(getClientId(fContext)))
                return value.substring(index+1);
              else
                return null;
```

```
          }//if (value.indexOf('@') != -1)
        }//if(nl.item(i).getFirstChild()
      }//for
    }//if(nl != null)
  }//if(doc != null)
  return null;
}//getActionValueFromRequest()
```

Notice the following points from the `getActionValueFromRequest()` method above:

1. It first calls the `getModelBeanObject()` method, which returns a `DOM Document` object. The `DOM Document` object contains `DOM` representation of the application-specific XML from the user's request.

2. Then it extracts the `action-performed` tag from the `DOM` document and extracts its contents. The contents of the `action-performed` tag consist of two parts separated by the "@" symbol. The first part identifies the component; the second part identifies the category or the product that was clicked. The `getActionValueFromRequest()` method identifies both the values and returns the product and category ID.

Next, we will discuss the implementation details of the `decode()` method.

```
public void decode (FacesContext context){
  if (context == null)
    throw new NullPointerException();
  String id = getActionValueFromRequest(context,
    "#{incomingXMLInstanceRequest.DOMDocument}", "action-performed");
  if (id != null){
    this.btnId = id;
    queueEvent(new ActionEvent(this));
  }//if
}//decode
```

You are already familiar with the functionality of the `decode()` method. Here, the `decode()` method is performing the following steps:

1. It checks whether the `FacesContext` object is `null`. If it is `null`, the `decode()` method throws an exception.

2. It calls the `getActionValueFromRequest()` method explained above.

3. The call in step 2 returns a string value. This value can tell which product or category was clicked.

4. if the ID returned by the `getActionValueFromRequest()` method is not `null`, the `decode()` method will set this ID in the `btnId` property of the component (which the event handling logic will use to identify the category or product ID the user clicked).

5. Finally, the `decode()` method fires an action event. To fire an action event,

the `decode()` method first instantiates an `ActionEvent` object, passing it the component instance. It then calls the `queueEvent()` method, passing it the `ActionEvent` object along with the method call. Note that the event handler class will extract the component object from the `ActionEvent` object, then call the `getBtnId()` method of the component to identify the category or product the user clicked.

Implementing the UICategoryActionListener class

The `UICategoryActionListener` class is associated with the `UICategory` component. When the user clicks any category or product in the catalog view, the `UICategory` component fires an action event. this action event is handled by `UICategoryActionListener`. The following code shows the methods in the `UICategoryActionListener` class:

```
public class UICategoryActionListener implements
ActionListener {
  public void processAction(ActionEvent event) {
  }
}//UICategoryActionListener
```

`UICategoryActionListener` implements the `processAction()` method of the `ActionListener` interface.

The `processAction()` method first creates an instance of the `FacesContext` object. It then gets an instance of the model bean from the context by calling the `getModelBeanObject()` method. The `getModelBeanObject()` method returns an object of the `CategoryData` class, which contains complete catalog data, as shown in boldface below:

```
public void processAction(ActionEvent event)
{
FacesContext context = FacesContext.getCurrentInstance();
  CategoryData categoryData = (CategoryData) getModelBeanObject
    ("#{categoryData}", context);
  .......
}//processAction
```

Next, the `processAction()` method calls the `getComponent()` method of the event object, which returns the instance of the component that fires the action event. Then it calls the `getBtnId()` method, which returns the ID of the category or product that identifies the category or product button clicked by the user in the catalog view. The `processAction()` method stores the ID in a `String` type variable named `actionId`:

```
public void processAction(ActionEvent event){
  FacesContext context = FacesContext.getCurrentInstance();
  CategoryData categoryData = (CategoryData) getModelBeanObject
    ("#{categoryData}", context);
```

```
UICategory uic = (UICategory) event.getComponent();
  String actionId = (String) uic.getBtnId();
  .......
}//processAction
```

Next, we check whether the user clicked a category or product. If the user clicked a category, there can be two further subcases: The user can click the root category or any subcategory.

If the user clicked the root category, we will set the category ID as `null` (as the root category does not have an ID). If the user clicked any subcategory in the catalog view, we will set the category ID as the `actionId` value. We then call the `setAction()` method of the model bean, passing the string "category" to tell the model bean that the user clicked a category.

```
public void processAction(ActionEvent event){
  .......
  UICategory uic = (UICategory) event.getComponent();
  String actionId = (String) uic.getBtnId();
if(actionId.equals("null"))
    category = true;
  else
    category = categoryData.isCategory(actionId);
  if (category){
    if(actionId.equals("null"))
      categoryData.getCategoryData().setId(null);
    else
      categoryData.getCategoryData().setId(actionId);
    categoryData.setAction("category");
  }//if(category)
  .......
}//processAction
```

Now the `processAction()` method handles the case when the user clicks a product in the catalog view. Clicking a product shows the product-specification view. First, we have to prepare the `ProductData` model bean, then we have to tell the `CategoryData` model bean that the user clicked a product.

So we call the `getModelBeanObject()` method, which returns an instance of the `ProductData` class. The `ProductData` model bean was initialized when the first request was made of the shopping cart, but it does not contains any data. Now we populate the `ProductData` object with the data of the product that the user wants. The product-specification view uses this data to render the specifications of the product.

Finally, we call the `setAction()` method of the `CategoryData` model bean, passing the string "product" to tell the `CategoryData` model bean that the user clicked a product:

```
public void processAction(ActionEvent event){
  .......
  if(actionId.equals("null"))
    category = true;
  else
    category = categoryData.isCategory(actionId);
```

```
    if (category){
      .......
    }
  else {
      ProductData productData = (ProductData) getModelBeanObject
        ("#{ProductData}", context);
      productData.setDocument(categoryData.getDocument());
      productData.setProductId(actionId);
      categoryData.setAction("product");
    }//else
  }//processAction
```

Implementing the CategoryData model bean

The `CategoryData` bean is an application-specific model bean that performs the following actions:

- It parses the XML file, which holds the complete catalog data comprising all products and categories in our shopping cart. Note that we are using a simple XML file for back-end data storage. If you want to use some other type of back-end data storage in your shopping cart application, you will only need to reimplement the `CategoryData` model bean, and the rest of the application will work fine.

- It also instantiates and populates the `Category` class required by the components to render the catalog view.

- The `CategoryData` model bean also implements action event handling methods to handle the events fired by the `xforms-jsf:commandButton` components in the catalog view.

The following code shows the properties and methods of the `CategoryData` class:

```
public class CategoryData{
  private String model = null;
  private String outcome = null;
  private Category categoryData;
  private Document document = null;

  public CategoryData() {}
  public void setModel(String model) {}
  public String getModel() {}
  public void setDocument (Document doc) {}
  public Document getDocument() {}
  public void setAction(String action) {}
  public Action getAction() {}
  public void setCategoryData(Category categoryData) {}
  public Category getCategoryData() {}
  public boolean isCategory(String id) {}
  private Category setCategoryDataValues(String id, Document doc) {}
  private void setSubCategoriesAndProducts (Element element,
```

```
      Category categoryData) {}
    private void setNameValuePair(Element[] elements, NameValuePair[]
      nameValuePair) {}
    private Element[] getElementsByTagName(Element element, String
      tagName) {}
    private Element getElementById(Element element, String tagName,
      String id) {}
    private void loadXML(String fileName) {}

    public void showCartView(ActionEvent ae) {
    }
  }//CategoryData
```

The `CategoryData` class has a constructor, four properties (`document`, `model`, `outcome`, and `categoryData`), setter and getter methods for these properties, some private helper methods, and a public `showCartView()` action event handling method.

The `document` property contains the complete catalog of our shopping cart. The `UICategoryActionListener` class calls the getter method (`getDocument()`) to pass on the document to the `ProductData` model bean.

The `model` property contains the application-specific XML, which is used to track the category or product the user clicked in the catalog view. The `xforms-jsf:model` component renders the application-specific XML, so the `UIModel` component explained in Implementing the xforms-jsf:model component on page 83 calls the getter method of the model property to fetch the application-specific XML.

The `outcome` property specifies the string used to navigate to the next page. This property can have any of the following three values: "category," "product," and "cart." If the value of the `outcome` property is "category" (which tells that the user clicked a category in the catalog view), the next page will be the same: catalogView.jsp. If the value of the `outcome` property is "product" (which tells that the user clicked a product), the next page will be productView.jsp. If the value of the `outcome` property is "cart" (which tells that the user clicked the Show cart button), the next page will be cartView.jsp.

The `categoryData` property is an instance of the `Category` class, which this model bean instantiates and populates for the `UICategory` component.

Let's discuss the methods of the `CategoryData` class one by one.

The `CategoryData` constructor first initializes the `Category` object. It then calls the `loadXML()` method, passing along the name of the XML file that contains the complete catalog data of our shopping cart. The constructor also sets the ID property of the newly created `Category` object to `null`. The `loadXML()` method parses the XML file and then loads it in the `document` property of the bean discussed above.

```
  public CategoryData(){
    categoryData = new Category();
    categoryData.setId(null);
    String path = fc.getApplication().getViewHandler().getResourceURL
```

```
      (fc, "/shoppingcart.xml");
    String url = fc.getExternalContext().encodeResourceURL(path);
    this.loadXML(url);
  }//CategoryData
```

The `CategoryData` class implements the `setAction()` method. The `setAction()` method takes a string parameter and sets it in the `outcome` property of the model bean. The `decode()` method of the `UICategory` class knows the ID of the category or product that the user clicked in the catalog view. Before firing the action event, the `decode()` method sets this ID in the `btnId` property of the component.

The event handler class (`UICategoryActionListener`) calls the getter method of the `btnId` to check if the ID belongs to a category. Then it calls the `setAction()` method and passes it a string value "category."

```
public void setAction(String action){
  this.outcome = action;
}//setAction
```

The `getAction()` method returns a string value, which the JSF framework uses for navigation. In Using the xcart:category tag to generate the catalog view on page 164, we provided the catalogView.jsp page. Notice the `action` attribute in the `xcart:category` tag shown in the JSP page. The `action` attribute specifies the `categoryData.getAction()` method name that decides the navigation. The following code shows the implementation of this method in the model bean:

```
public String getAction(){
  return outcome;
}//getAction
```

The `getAction()` method returns the value of the `outcome` property (which `UICategoryActionListener` has already set by calling the `setAction()` method).

The `setCategoryData()` method simply takes an instance of the `Category` object and assigns it to the `categoryData` property.

```
public void setCategoryData(Category categoryData){
  this.categoryData = categoryData;
}//setCategoryData
```

The `decode()` method extracts the ID of the category or product the user clicked in the catalog view and passes it to the event handler class (the `UICategoryActionListener` class). When `UICategoryActionListener` receives the ID, it instantiates the `Category` class and sets the `id` property of the `Category` object. It then calls the `setCategoryData()` method of the model bean and passes the `Category` object along with the method call.

The `getCategoryData()` method simply calls a helper method named `getCategoryDataValues()`. The `getCategoryDataValues()` method

takes the same ID that the `UICategoryActionListener` supplied to the model bean. The `getCategoryDataValues()` helper method returns another `Category` object that contains the same ID, as well as all the data related to the same category (list of subcategories, ancestors, and products).

```
public Category getCategoryData(){
  Category catData = getCategoryDataValues(categoryData.getId());
  return catData;
}//getCategoryData
```

The `CategoryData` model bean implements an action event handling method named `showCartView()`. Recall the screen shot for the catalog view from Views of the shopping cart on page 7, which contains a Show cart button. When the user clicks this, the JSF framework calls the `showCartView()` method of the model bean, which simply sets the `action` property with a "cart" string and causes the navigation to the cartView.jsp page.

```
public void showCartView(ActionEvent ae){
  this.action = "cart";
}//showCartView
```

In the past four sections, we have seen the implementation of the catalogView.jsp page, `UICategory` component, `UICategoryActionListener`, and the `CategoryData` model bean. These four components together form a set, which renders the catalog view.

In the next two sections, we will implement similar kinds of sets to render the product-specification view and catalog view. But this time, we will not go into low-level details like we did for first set.

Implementing the productView.jsp page, UIProduct component, and ProductData model bean

In the productView.jsp page, we have used three tags from our XForms-JSF tag library (`xforms-jsf:model`, `xforms-jsf:selectManyCheckbox`, and `xforms-jsf:commandButton`) and one tag (`xcart:product`) from our shopping cart tag library.

The JSP author provides the `xcart:product` tag with a `value` attribute, whose value refers to a `Product` class type property of a `ProductData` model bean. For example, look at the following `xcart:product` tag in the JSP page:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html
  xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xforms="http://www.w3.org/2002/xforms">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://afictitiousshoppingcart.com/XCart"
```

```
prefix="xcart" %>
<%@ taglib uri="http://afictitiousshoppingcart.com/XForms-JSF"
  prefix="xforms-jsf" %>
<f:view>
 <head>
  <xforms-jsf:model value="#{productData.model}"/>
 </head>
 <body>
  <xcart:product value="#{productData.product}"/>
  <xforms-jsf:selectManyCheckbox label="Optional Features: "
    value="#{productData.selectedOptionalFeatures}" ref="options">
    <f:selectItems value="#{productData.optionalFeatures}"/>
  </xforms-jsf:selectManyCheckbox>
  <xforms-jsf:commandButton label="Add to cart" immediate="false"
     action="#{productData.getAction}"
     actionListener="#{productData.addProductToCart}">
  </xforms-jsf:commandButton>
  <xforms-jsf:commandButton label="Back to catalog view"
     action="#{productData.getAction}" immediate="true"
     actionListener="#{productData.showCatalogView}">
  </xforms-jsf:commandButton>
 </body>
</f:view>
</html>
```

The `xcart:product` tag takes only one attribute: `value`. You are already familiar with the functionality of the `value` attribute. In the above JSP, the `value` attribute contains the `product` property of the `ProductData` model bean. The `xcart:product` component implementation calls the getter method of the `productData.product` property and gets the `Product` instance that contains the complete details of a single product.

The `xcart:product` tag renders the details of the product the user clicked in the catalog view. These details of a product consist of the name, price, description, and features of a particular product.

Now let's implement the component class mentioned in the `component-class` element of the faces-config.xml file above: the `UIProduct` class.

The following code shows the properties and methods of the `UIProduct` class:

```
public class UIProduct extends UIComponentBase{
  private String value;
  public void encodeBegin(FacesContext context)
    throws IOException  {
  }
  public void setValue(Object value) {
  }
}//UIProduct
```

The `UIProduct` class contains only one property (`value`), two public methods (`encodeBegin()` and `setValue()`), and four private helper methods (`renderProductSpec()`, `renderFeatures()`, `renderNameValue()`, and `getModelBeanObject()`).

The `value` property contains the name of the `Product` object associated with the `UIProduct` component. The `UIProduct` component uses the `value`

attribute to get the `Product` object. It then uses the `Product` object to fetch the application-specific data (name, id, price, and description of a product).

Let's look at the `encodeBegin()` method of the `UIProduct` class.

```
public void encodeBegin(FacesContext context)
  throws IOException
{
  if (context == null)
    throw new NullPointerException();
  Product productData = (Product) getModelBeanObject (context,
    value);
  ResponseWriter writer = context.getResponseWriter();
  if (productData == null)
    writer.write("<br/>product data is null"); //show nothing
  else
  {
    writer.write("<h3> X-Cart Product Detail </h3>");
    writer.write("<table width=\"100%\">");
    writer.write("<tr>");
    writer.write("<table colspan=\"3\" border=\"1\" width=\"70%\"");
    String[] spec = new String[4];
    spec[0] = productData.getId();
    spec[1] = productData.getName();
    spec[2] = productData.getDescription();
    spec[3] = productData.getPrice();
    String[] labels = {"Product_ID","Product_Name:",
      "Product_Description:", "Product_Price:"};
    renderProductSpec(writer, spec, labels);
    NameValuePair[] features = productData.getFeatures();
    if (features != null)
      renderFeatures(writer, "Product Features",features);
    writer.write("</table>");
    writer.write("</tr>");
    writer.write("</table>"); //colspan=\"4\" border=\"1\"
  }//else
}//encodeBegin
```

The following points explain the above code:

- It first verifies the `FacesContext` instance passed to it. If it is `null`, it throws a `NullPointerException`.

- After verifying the `FacesContext` object, it calls the `getModelBeanObject()` method, which returns an instance of the `Product` object.

- It fetches the `ResponseWriter` object.

- It then determines whether the `Product` instance is `null`. If so, it hard-codes the markup to display that the product is `null`.

- If the `Product` instance retrieved in step 2 is not `null`, it uses the `Product` instance to get the name, description, price, and features of the product.

- Finally, it calls the `renderProductSpec()` and `renderFeatures()` private helper methods to display the product specifications (product name, description, price, etc.) and product features.

Now we will implement the `ProductData` model bean.

The `ProductData` model bean is associated with the product-specification view. This model bean basically populates the `Product` class used by `UIProduct` and `UISelect` components to display the product details like name, price, features, and optional features. The `ProductData` model bean also implements the action event handling methods to handle the action events generated in the product-specification view and in the edit product view.

The following code shows the implementation of the `ProductData` model bean:

```
public class ProductData{
  private Product productData;
  private Document document = null;
  private String model = null;
  private String productId = null;
  private boolean isProductSet = false;
  private ArrayList optionalFeatures;
  private String[] selectedOptionalFeatures = null;

  public ProductData() {}
  public String getModel() {}
  public void setModel(String model){}
  public void setDocument(Document doc) {}
  public Document getDocument(){}
  public String getProductId(){}
  public void setProductId(String id) {}
  public Product getProductData(){}
  public void setProductData(Product pData) {}
  public void setOptionalFeatures (String[] optFeatures, Product pd) {}
  public Collection getOptionalFeatures (){}
  public void setSelectedOptionalFeatures (String[]
    selectedOptionalFeatures) {}
  public String[] getSelectedOptionalFeatures(){}
  public void showCatalogView(ActionEvent ae){
  }
  public void showCartView(ActionEvent ae){
  }
  public void addProductToCart(ActionEvent ae){
  }
  public void saveToCart(ActionEvent ae){
  }
}//ProductData
```

The `UIProduct` class contains a constructor, six properties (`document`, `model`, `productData`, `productId`, `optionalFeatures`, and `selectedOptionalFeatures`), setter and getter methods for these properties, and four event handler methods (`showCatalogView()`, `addProductToCart()`, `showCartView()`, and `saveToCart()`).

The `document` property contains the complete catalog of our shopping cart.

`UICategoryActionListener` calls the setter method (`setDocument()`) to pass on the document to the `ProductData` model bean.

The `model` property contains the application-specific XML. The `UIModel` component calls the getter method of the `model` property to fetch the application-specific XML.

The `productId` property contains the ID of the product the user clicked in the catalog view. `UICategoryActionListener` calls the setter method (`setProductId()`) to pass on the product ID the user clicked.

The `optionalFeatures` property of bean contains the optional feature of the product the user clicked in the catalog view. The `UISelect` component calls the `getOptionalFeatures()` method to render the optional features of the product as check boxes.

The `selectedOptionalFeatures` property stores those optional features the user selects in the product-specification view. The `updateModel()` method of the `UISelect` component calls the `setSelectedOptionalFeatures()` method to set the user's selected optional features that its respective `decode()` method retrieved from the request.

The `product` property of the `ProductData` model bean is an instance of the `Product` class. This property only stores the record of a single product. The `UIProduct` and `UISelect` components call the `getProduct()` method to render the product details.

Let's discuss the implementation of the event handling methods in the `ProductData` model bean one by one.

The `showCatalogView()` method takes an instance of the `ActionEvent` class along with the method call. Recall the product-specification view from Views of the shopping cart on page 7, where a Back to catalog view button is shown. When the user clicks this, an action event occurs. The JSF framework calls the `showCatalogView()` method to handle the event. This method simply sets the `action` property with the "catalogView" string:

```
public void showCatalogView(ActionEvent ae){
  this.action = "catalogView";
}//showCatalogView
```

The `showCartView()` method takes an instance of the `ActionEvent` class along with the method call. Recall the edit product view from Views of the shopping cart on page 7, where a Back to cart button is shown. When the user clicks this, an action event occurs. The JSF framework calls the `showCartView()` method to handle the event. This method simply sets the action property with the "cart" string.

```
public void showCartView(ActionEvent ae){
  this.action = "cart";
}//showCartView
```

The `addProductToCart()` event handling method takes an instance of the `ActionEvent` object along with the method call. When the user clicks the Add to cart button in the product-specification page, the JSF calls the `addProductToCart()` method to handle the event. The following code shows the implementation of the `addProductToCart()` method:

```
public void addProductToCart(ActionEvent ae) {
  FacesContext context = FacesContext.getCurrentInstance();
  Application app = context.getApplication();
  ValueBinding vb = app.createValueBinding("#{cartData}");
  CartData cdmb = (CartData) vb.getValue(context);
  Product pd = getProduct();
  String[] selectedFeatures = getSelectedOptionalFeatures();
  if (selectedFeatures != null) {
    NameValuePair[] optionalFeatures =  pd.getOptionalFeatures();
    NameValuePair[] new_optionalFeatures = new
          NameValuePair[selectedFeatures.length];
    for (int x = 0; x < selectedFeatures.length; x++){
      for (int y = 0; y < optionalFeatures.length; y++){
        NameValuePair pair = optionalFeatures[y];
        if (selectedFeatures[x].equals(pair.getName()))
          new_optionalFeatures[x] = pair;
      }//for
    }
    pd.setSelectedOptionalFeatures(new_optionalFeatures);
  }//if(selectedFeatures != null)
  cdmb.addProduct(pd);
  this.action = "cart";
}//addProductToCart
```

Note the following points:

- The `addProductToCart()` method first gets an instance of the `FacesContext` object.
- It fetches an instance of the `CartData` model bean from the application context.
- The `addProductToCart()` method then calls the `getProduct()` method, which returns the `Product` class instance that wraps the details of the product the user clicked in the catalog view.
- Next, the `addProductToCart()` method calls the `getSelectedOptionalFeatures()` method, which returns the list of features the user selected in the product-specification view.
- It checks if the selected optional features are not `null`, then it sets the selected optional features in the `selectedOptionalFeatures` property of the `Product` object.
- After setting the selected features, it adds the product to the cart. It calls the `addProduct()` method of the `CartData` model bean to add the product to the cart.

- Finally, the `addProductToCart()` method sets the `action` property of the model bean with the "cart" string.

When the user clicks the Save edited product button in the edit product view, the JSF calls the `saveToCart()` method to handle the event. The following code shows the implementation of the `saveToCart()` method:

```
public void saveToCart(ActionEvent ae){
  FacesContext context = FacesContext.getCurrentInstance();
  Application app = context.getApplication();
  ValueBinding vb = app.createValueBinding("#{cartData}");
  CartData cd = (CartData) vb.getValue(context);
  Product pd = getProduct();
  String[] selectedFeatures = getSelectedOptionalFeatures();
  if (selectedFeatures == null || selectedFeatures.length == 0)
    pd.setSelectedOptionalFeatures(new NameValuePair[0]);
  else{
    NameValuePair[] optionalFeatures =  pd.getOptionalFeatures();
    NameValuePair[] new_optionalFeatures =
        new NameValuePair[selectedFeatures.length];
    for (int x = 0; x < selectedFeatures.length; x++){
      for (int y = 0; y < optionalFeatures.length; y++){
        NameValuePair pair = optionalFeatures[y];
        if (selectedFeatures[x].equals(pair.getName()))
          new_optionalFeatures[x] = pair;
      }//for(int y = 0; y < optionalFeatures.length; y++)
    }//for(int x = 0; x < selectedFeatures.length; x++)
    pd.setSelectedOptionalFeatures(new_optionalFeatures);
  }//else
  Product[] pData = cd.getProducts();
  for(int i = 0; i< pData.length; i++){
    if( pData[i].getId().equals(pd.getId()) )
      cd.setProductAt(pd, i);
  }//for
  this.action = "cart";
}//saveToCart
```

Note the following code points:

- The `saveToCart()` method first gets an instance of the `FacesContext` object.

- It then fetches an instance of the `CartData` model bean from the application context.

- Then the `saveToCart()` method calls the `getProduct()` method, which returns the `Product` class instance that wraps the details of the product the user clicked in the catalog view.

- The `saveToCart()` method calls the `getSelectedOptionalFeatures()` method, which returns the list of features the user selected in the product-specification view.

- It then checks if the selected optional features are `null` and sets the

`selectedOptionalFeatures` property of the `Product` object as `null`.

- If the selected optional features are not `null`, it sets the selected optional feature in the `selectedOptionalFeatures` property of the `ProductData` model bean class.

- After adding the selected features, it calls the `getProducts()` method of `CartData`, which returns all products in the cart as an array of the `Product` object.

- After iterating through all the products until the current product (edited) is found, it overwrites the edited product.

- Finally, the `saveToCart()` method sets the `action` property of the model bean with the "cart" string.

Implementing the cartView.jsp page, UICart component, UICartActionListener, and CartData model bean

Look at the following cartView.jsp page, which generates the cart view:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html
  xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:xforms="http://www.w3.org/2002/xforms">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://afictitiousshoppingcart.com/XCart"
prefix="xcart" %>
<%@ taglib uri="http://afictitiousshoppingcart.com/XForms-JSF"
  prefix="xforms-jsf" %>
<f:view>
 <head>
   <xforms-jsf:model value="#{cartData.model}"/>
 </head>
 <body>
  <xcart:cart value="#{cartData.products}"
    action="#{cartData.getAction}">
    <f:actionListener type="model.UICartActionListener"/>
  </xcart:cart>
  <xforms-jsf:commandButton label="Buy Cart"
      action="#{cartData.getAction}"
      actionListener="#{cartData.buy}" immediate="true">
  </xforms-jsf:commandButton>
  <xforms-jsf:commandButton label="Back to catalog view"
      action="#{cartData.getAction}"
      actionListener="#{cartData.showCatalogView}" immediate="true">
  </xforms-jsf:commandButton>
 </body>
</f:view>
```

```
</html>
```

Note the following points:

- We have included the `xcart:cart` tag in the body element, which contains two attributes: `value` and `action`.

- The `xcart:cart` tag renders the list of all the products a user added to the cart. The `xcart:cart` tag also renders Remove and Edit buttons with each product in the cart.

- The method of the model bean specified in the `action` attribute (cart.getAction) controls navigation.

- We have associated an action listener (`UICartActionListener`) with the `xcart:cart` component. When the user clicks the edit or Remove button of any product in the cart view, the `UICartActionListener` class handles the generated action event, which we will discuss shortly.

- We have used two `xforms-jsf:commandButton` tags. The first is used to go back to the catalog view; the second is to buy the cart.

Next, we'll implement the component class associated with the `xcart:cart` tag.

The following code shows the properties and methods in the `UICart` component:

```
public class UICart extends UICommand{
  private String btnId = null;
  private String productId = null;
  private void renderValueAsButton (ResponseWriter writer, String name,
    String id, String clientId) {}
  public void encodeBegin(FacesContext context)
    throws IOException  {}
  public void decode (FacesContext context) {}
  private String getActionValueFromRequest(String retValue,
    FacesContext fc, String ref, String tagName) {}
}//UICart
```

The `UICart` class contains two properties (`btnId`, `productId`), setter and getter methods for the properties, two public methods (`encodeBegin()` and `decode()`), and three private helper methods (`renderValueAsButton()`, `getModelBeanObject()`, and `getActionValueFromRequest()`).

The `btnId` property contains the ID of the Edit or Remove button the user clicks in the cart view. When the `UICart` component fires an action, the action event handling class calls the getter method of this property to compare the button the user clicked against a particular product in the cart view.

The `productId` property contains the ID of the product whose Edit or Remove

button the user clicks in the cart view. When the `UICart` component fires an action, the action event handling class calls the getter method of the `productId` property to get the product ID.

Let's discuss the methods of the `UICart` class one by one.

The `renderValueAsButton()` method writes the markup to render a single product as a button. The `encodeBegin()` method of the `UICategory` class calls this method once for every product available in the cart view.

The `renderValueAsButton()` method takes the following four parameters:

- **writer:** An object of the `ResponseWriter` class. Its `write()` method is used to write the response markup.
- **name:** The name of a product.
- **id:** The ID of a product.
- **clientId:** Uniquely identifies a component on the client side.

The code for the `renderValueAsButton()` method:

```
private void renderValueAsButton(ResponseWriter writer, String name,
  String id, String clientId){
  try {
    writer.write("<xforms:submit submission=\"submit\"");
    writer.write("
xmlns:xforms=\"http://www.w3.org/2002/xforms\">");

writer.write("<xforms:label>"+name.trim()+"</xforms:label>");
    writer.write("<xforms:action ev:event=\"DOMActivate\"");
    writer.write("
xmlns:ev=\"http://www.w3.org/2001/xml-events\">");
    writer.write("<xforms:setvalue ref=\"action-performed\">"+
      clientId+"@"+id+"@"+name.trim()+"</xforms:setvalue>");
    writer.write("</xforms:action>");
    writer.write("</xforms:submit>");
  }
  catch (java.io.IOException ie){
    ie.printStackTrace();
  }
}//renderValueAsButton
```

The above code authors the following markup:

```
<xforms:submit submission="submit"
  xmlns:xforms="http://www.w3.org/2002/xforms"
  xmlns:ev="http://www.w3.org/2001/xml-events">
  <xforms:label>Intel_Pentinum4</xforms:label>
  <xforms:action ev:event="DOMActivate">
    <xforms:setvalue ref="action-performed">
 _id1@1@edit
```

```
    </xforms:setvalue>
  </xforms:action>
</xforms:submit>
```

This markup renders a button. The whole markup is hard-coded in the `renderValueAsButton()` method, except the strings "Intel_Pentinum4" and "_id1@1@edit" (boldface in the markup above). The first string "Intel_Pentinum4" is the value of the second parameter (`name`). The second string "_id1@1@edit" uniquely identifies the product whose Remove and Edit button the user clicks. The second string consists of three parts, each of which is separated by an "@" symbol. The first part (_id1) is the ID of the component. The second part (1) identifies a particular product in the catalog view. The third part (`edit`) identifies the button clicked.

Recall Implementing the xforms-jsf:model component on page 83, where we explained the purpose of inserting an extra `action-performed` tag in the application-specific XML. The `renderValueAsButton()` method wraps the component, product ID, and button name (edit or remove) in the `action-performed` tag. When a user clicks the button corresponding to a particular product in the cart view, the string "_id1@1@edit" is sent back to the server.

The `encodeBegin()` method of the `UICart` class is lengthy, so we will implement this method incrementally.

The `encodeBegin()` method first checks whether the `FacesContext` instance is `null`. If it is `null`, it simply throws an exception. Then it extracts the component ID by calling the `getClientId()` method. Next, it gets the `ResponseWriter` object so that it can render the markup for this component. See these steps in the following code:

```
public void encodeBegin(FacesContext context) throws IOException {
if (context == null)
    throw new NullPointerException();
  String clientId = getClientId(context);
  ExternalContext externalContext = context.getExternalContext();
  ResponseWriter writer = context.getResponseWriter();
  double totalPrice = 0;
  .....
}//encodeBegin
```

Now the `encodeBegin()` method calls the `getModelBeanObject()` method. We explained how this method works in Implementing the UISelect1 component on page 100. The `getModelBeanObject()` method returns an array-type object `Product` class, which contains the data to be rendered. See the boldface code below:

```
public void encodeBegin(FacesContext context)
  throws IOException {
  if (context == null)
    throw new NullPointerException();
  String clientId = getClientId(context);
  ExternalContext externalContext = context.getExternalContext();
  ResponseWriter writer = context.getResponseWriter();
```

```
  double totalPrice = 0;
Product[] products = (Product[]) getModelBeanObject(context,
    value);
  .....
}//encodeBegin
```

Now we check if the `Product` instance returned by the `getModelBeanObject()` method is `null`. If it is `null`, it means that we have no data (products) to render in the cart view.

```
public void encodeBegin(FacesContext context)
  throws IOException {
  ......
  Product[] products = (Product[]) getModelBeanObject(context,
    value);
if (products == null){
    writer.write ("Cart is empty......");
    return;
  }//if(products == null)
  .....
}//encodeBegin
```

Now the `encodeBegin()` method writes the markup to display each product and its price in tabular format from the `Product` object, as shown in boldface below:

```
public void encodeBegin(FacesContext context)
  throws IOException {
  ......
  if (products == null){
    writer.write ("<br><h3> Cart is
empty......</h3>");
    return;
  }//if (products == null)
writer.write("\r\n X-Cart View <br/> ");
  writer.write("<table border=\"1\">");
  writer.write("<ol type=\"1\">");
  for (int x = 0; x<products.length; x++){
    Product product = products[x];
    double price = Integer.parseInt(product.getPrice());
    NameValuePair[] pair = product.getSelectedOptionalFeatures();
    if(pair != null ){
      for(int y = 0; y < pair.length; y++){
        price += Integer.parseInt(pair[y].getValue());
      }
    }//if(pair != null )
    totalPrice += price;
    writer.write("<tr>");
    writer.write("<td>");
    writer.write("<li>"+product.getName()+"</li>");
    writer.write("</td>");
    writer.write("<td>");
    renderValueAsButton(writer, " Edit ",
      new Integer(x).toString(), clientId);
    writer.write("</td>");
    writer.write("<td>");
    renderValueAsButton(writer, " Remove ",
      new Integer(x).toString(), clientId);
```

```
      writer.write("</td>");
      writer.write("<td>");
      writer.write("" + price);
      writer.write("</td>");
      writer.write("</tr>");
    }//for (int x = 0; x<products.length; x++)
    writer.write("</ol>");
    writer.write("</table>");
    writer.write("<table border=\"1\">");
    writer.write("<ol>");
    writer.write("<tr>");
    writer.write("<td>");
    writer.write("<li> Total Cart Price </li>");
    writer.write("</td>");
    writer.write("<td>");
    writer.write(""+totalPrice);
    writer.write("</td>");
    writer.write("</tr>");
    writer.write("</ol>");
    writer.write("</table>");
  }//encodeBegin
```

Now we'll implement the `getActionValueFromRequest()` method. The `decode()` method calls this method to get the ID of the product whose Edit or Remove button the user clicked in the cart view.

The `getActionValueFromRequest()` method takes the following four parameters:

- **retValue:** Contains the string value "id" or "action," which decides the return value of the method.
- **fc:** The `FacesContext` instance.
- **ref:** Contains the model bean (that is, `IncomingXMLInstanceRequest`) name and the property that contains the incoming XML request from the client side. See Parsing the incoming XML instance data on page 91 for details about this model bean.
- **tagName:** The name of the tag (that is, `action-performed`) that contains the clicked button information.

```
  private String getActionValueFromRequest(String retValue,
     FacesContext fc, String ref, String tagName){
    String value = new String();
    Document doc = (Document) getModelBeanObject(fc, ref);
    if(doc != null){
      NodeList nl = doc.getElementsByTagName(tagName);
      if (nl != null){
        for (int i=0; i < nl.getLength(); i++){
          if (nl.item(i).getFirstChild().getNodeType() ==
            Node.TEXT_NODE ){
            value = nl.item(i).getFirstChild().getNodeValue();
            if (retValue.equals("id")){
              if (value.indexOf('@') != -1){
                int index = value.indexOf('@');
```

```
              String id = value.substring(0,index);
              if (id.equals(getClientId(fc)))
                return value.substring(index+1,
                  value.lastIndexOf('@'));
              else
                return null;
            }//if(value.indexOf('@') != -1)
          }//if(retValue.equals("id"))
          else
            return value.substring(value.lastIndexOf('@')+1);
        }//if
      }//for
    }//if(nl != null)
  }//if(doc != null)
  return null;
}//getActionValueFromRequest
```

Note the following points from the `getActionValueFromRequest()` method:

° The method first gets the property from the model bean that contains the client request as `DOM Document` object.

° Then it extracts the `action-performed` tag from the `DOM` document and extracts its value.

° Finally, it checks the value of the `retValue` parameter. If it is "id," then it returns the product ID whose Edit or Remove button is clicked. Otherwise, it returns the ID of the button -- edit or Remove -- from the request.

Next, we will discuss the implementation details of the `decode()` method.

```
public void decode(FacesContext context){
  if (context == null)
    throw new NullPointerException();
  String id = getActionValueFromRequest("id", context,
     "#{incomingXMLInstanceRequest.DOMDocument}", "action-performed");
  String action = getActionValueFromRequest("action", context,
     "#{incomingXMLInstanceRequest.DOMDocument}","action-performed");
  if(id != null){
    this.btnId = action;
    this.productId = id;
    queueEvent(new ActionEvent(this));
  }//if(id != null)
}//decode
```

Note the following points in the `decode()` method implementation:

° It first verifies the `FacesContext` instance passed to it by the JSF framework. If it is `null`, it throws a `NullPointerException`.

° If the `FacesContext` instance is not `null`, the `decode()` method calls the `getActionValueFromRequest()` method, passing it the string "id" in the `retValue` parameter, as explained above. This method returns the ID of the product that the user clicked.

- The `decode()` method again calls the `getActionValueFromRequest()` method, passing it the string "action." It returns the name of the button the user clicked (edit or Remove).

- If the product ID returned by the `getActionValueFromRequest()` method is not `null`, it sets the product name ID in the `btnId` property and the product ID in the `productId` property of the component (that the event handling logic uses to identify the Edit or Remove button of a particular product the user clicked).

- Finally, the `decode()` method fires an action event by instantiating an `ActionEvent` object, passing it the component instance. It then calls the `queueEvent()` method, passing it the `ActionEvent` object along with the method call. Note that the event handler class extracts the component object from the `ActionEvent` object and calls the `getBtnId()` method of the component to identify the product the user clicked.

Next, we will implement the `UICartActionListener`.

The `UICartActionListener` class is associated with the `UICart` component. When the user clicks the Edit or Remove button of a particular product in the cart view, the `UICart` component fires an action event. This action event is handled by the `UICartActionListener` class. The implementation of the `UICartActionListener` class:

```
public class UICartActionListener implements ActionListener {
  public void processAction(ActionEvent event)  {
  }
  private Object getModelBeanObject(FacesContext context,String value){
  }
}//UICartActionListener
```

The `UICartActionListener` class contains two methods (`processAction()` and `getModelBeanObject()`). We explained the `getModelBeanObject()` method in Implementing the UISelect1 component on page 100. The `processAction()` method processes the action event fired by the component:

```
public void processAction(ActionEvent event) {
  FacesContext context = FacesContext.getCurrentInstance();
  CartData cd = (CartData) getModelBeanObject("#{cartData}",
    context);
  ProductData pdm = (ProductData)
    getModelBeanObject("#{editProductData}", context);
  UICart uic = (UICart) event.getComponent();
  String productId = uic.getProductId();
  String btn = uic.getBtnId();
  if(btn.equals("Edit")){
    Product pd = cd.getProductAt(productId);
    if (pd != null)
      pdm.setProduct(pd);
    cd.setAction("edit");
  }//if(btn.equals("Edit"))
```

```
    else{
      cd.removeProduct(productId);
      cd.setAction("remove");
    }//else
  }//processAction
```

Note the following points:

- The `processAction()` method first retrieves the `FacesContext` instance.
- It then fetches the `CartData` and `ProductData` model bean instances from the application context.
- Next, it calls the `getComponent()` method of the event object, which returns an object of the component class that fired the event.
- Then it calls the `getBtnId()` and `getProductId()` methods of the component class, which returns the ID of the button the user clicked and the product ID against that button, respectively.
- It then checks if the button ID is Edit. if it is, it fetches the product from the cart against the product ID and sets it in the `ProductData` model bean by calling its `setProduct()` method.
- If the button ID is not Edit, it simply calls the `removeProduct()` method of the `CartData` model bean, passing it the product ID. This method call simply removes the product from the cart.

Now let's implement the `CartData` model bean.

The `CartData` model bean stores all the products added by the user in the cart. The `UICart` component uses this `CartData` model bean to fetch the list of products added to the cart and also to handle some action events that occurred in the cart view.

The following code shows the implementation of the `CartData` class:

```
  public class CartData{
    protected String model = null;
    protected String outcome = null;
    protected Product[] products = null;

    public String getAction(){
    }
    public void setAction (String outcome){
    }
    public String getModel(){
    }
    public void setModel(String model){
    }
    public Product [] getProducts(){
```

```
    }
    public Product getProductAt(String index){
    }
    public void setProductAt(Product pd, int index)
    }
    public void removeProduct(String index){
    }
    public void addProduct(ProductData productData){
    }
    public void showCatalogView(ActionEvent ae){
    }
    public void buy(ActionEvent ae){
    }
  }//CartData
```

The `CartData` class has three properties (`model`, `outcome`, and `products`), setter and getter methods for these properties, some private helper methods, four public methods (`getProductAt()`, `setProductAt()`, `addProduct()`, and `removeProduct()`), and two action event handling methods (`showCatalogView()` and `buy()`).

The `model` property contains the application-specific XML, which is used to track the product the user clicked in the cart view. The `xforms-jsf:model` component renders the application-specific XML, so the `UIModel` component explained in Implementing the xforms-jsf:model component on page 83 calls the getter method of the `model` property to fetch the application-specific XML.

The `outcome` property specifies the string used for navigation. This property can have one of three values: "catalogView," "buy," or "edit." If the value of the `outcome` property is "catalogView" (which indicates that the user clicked the Back to catalog view button in the cart view), the next page will be the catalogView.jsp. If the value is "edit" (which indicates that the user clicked the Edit button of particular product), the next page will be editProductView.jsp. If the value is "buy" (which indicates that the user clicked the Buy button), the next page will be checkout.jsp.

The products property is an array of the `Product` object, which stores all the products added by the user to the cart. The `UICart` component calls its getter method to fetch the list of products added to the cart.

Let's discuss the methods of the `CartData` class one by one.

The `CartData` class implements the `setAction()` method. The `setAction()` method takes a string parameter and sets it in the `outcome` property of the model bean. The action event handling logic calls the `setAction()` method, passing it a string value.

```
public void setAction(String action){
  this.outcome = action;
}//setAction
```

The `getAction()` method returns a string value, which the JSF framework uses for the navigation. In the cartView.jsp page above, notice the `action` attribute in the `xcart:cart` tag. The `action` attribute specifies the

cartData.getAction method name that decides the navigation. The following code shows the implementation of this method in the model bean:

```
public String getAction(){
  return outcome;
}//getAction
```

The `getAction()` method returns the value of the `outcome` property (which the event handling logic has already set by calling the `setAction()` method).

The `getProductAt()` method returns a product from the array of `Product` objects at some specific index. It takes the product index along with the method call. The `UICartActionListener` calls this method to fetch the product at some specific location in the cart.

```
public Product getProductAt(String index) {
  if ( productsVector.size() > 0 )
    return (Product)productsVector.elementAt(Integer.parseInt(index));
  return null;
}//getProductAt
```

The `setProductAt()` method sets a product at some specific index in the array of the `Product` object. It takes the `Product` object and index along with the method call. The `UICartActionListener` calls this method to set the product at a particular location:

```
public void setProductAt(Product pd, int index) {
  if ( pd != null )
    productsVector.setElementAt(pd, index);
}//setProductAt
```

The `addProduct()` method adds the product to the cart that is passed to it along with the method call. The event handling logic behind the Add to cart button in the product-specification view calls the `addProduct()` method, passing it the `Product` object along with the method call.

```
public void addProduct(Product productData) {
  if (productData != null) {
    productsVector.addElement(productData);
  }
}//addProduct
```

The `removeProduct()` method removes the product from the array of `Product` objects at the given index. When the user clicks the Remove button in the cart view, the `UICartActionListener` calls the `removeProduct()` method, passing it the index of the product in the cart.

```
public void removeProduct(String index) {
  productsVector.removeElementAt(Integer.parseInt(index));
}//removeProduct
```

The `CartData` model bean implements an action event handling method named `showCatalogView()`. Recall the screenshot for the cart view from Views of the shopping cart on page 7, which contains a Back to catalog view button. When the user clicks this, the JSF framework calls the `showCatalogView()` method of the model bean, which sets the `action` property with a "catalogView" string and causes the navigation to the catalogView.jsp page.

```
public void showCatalogView(ActionEvent ae){
  this.outcome = "catalogView";
}//showCatalogView
```

The screenshot for the cart view shown in Views of the shopping cart on page 7 contains a Buy button. When the user clicks this, the JSF framework calls the `buy()` method of the model bean, which sets the `action` property with a "buy" string and causes the navigation to the checkout.jsp page.

```
public void buy(ActionEvent ae){
  if(getProducts().length == 0){
    return;
  }
  this.outcome = "buy";
}//buy
```

Trying out the shopping cart

We have placed the source code for our complete XForms-JSF shopping cart application in the section9.zip file available in the source code download of this tutorial; see Resources on page 200. When you unzip the section9.zip file, you will find that it contains an xcart.jar file (that is, the shopping cart-specific tag library developed in this section), ShoppingCart.war file (that is, the XForms-JSF Shopping cart application developed in this section), and a folder named ShoppingCart. The ShoppingCart folder contains the complete source code for our XForms-JSF shopping cart application, including all the JSP pages, model beans, and action listeners developed for our sample shopping cart application.

To try our XForms-JSF shopping cart application, you deploy the ShoppingCart.war file in your application server. Be sure to use the following URL in the address bar of your XForms browser:

```
http://localhost:8080/ShoppingCart
```

Once the catalog view displays, you can browse through our shopping cart, as we discussed in Views of the shopping cart on page 7.

Section 10. Wrap-up and resources

Summary

Throughout this tutorial, our goal has been to give an inside-out picture of the JSF framework, to show how to use JSF for XForms applications, and enable you to develop your own JSF components and tag libraries. For this purpose, we developed tag libraries and many sample applications.

We also discussed the XForms authoring requirements in a server-side Java application and listed the tasks you need to perform to develop an XForms application using JSF technology. We discussed how the JSF framework internally works and how the different modules of the JSF application cooperate with each other. Finally, we developed the XForms-JSF tag library and several XForms applications using the JSF framework.

Resources

- Download the j-jsfx-source.zip file for the complete source code (sample application, WAR, and JAR files) developed in this tutorial.
- Download *J2SE V1.4.2* (http://java.sun.com/j2se/1.4.2/download.html) and *J2EE V1.4* (http://java.sun.com/j2ee/1.4/download.html) (Update 1) reference implementation.
- Visit the official *JavaServer Faces information* at Sun Microsystems.
- For an overview of JSF, and to learn about the existing JSF core and HTML tag libraries, read *UI development with JavaServer Faces* (developerWorks, September 2003), a tutorial by Jackwind Li Guojie.
- Learn more about JSF with Rick Hightower's series, *JSF for non believers* (developerWorks, February 2005).
- Read *Integrating Struts, Tiles, and JavaServer Faces* (developerWorks September 2003) by Srikanth Shenoy and Nithin Mallya, which explains how to use JSF with the Apache Struts framework.
- Visit the official *XForms information* (http://www.w3.org/TR/xforms/) at w3.org.
- To understand XForms in detail, read *Understanding XForms* (developerWorks, December 2002) by Nicholas Chase.

- To get started with XForms, read *Get ready for XForms* (developerWorks, September 2002) by Joel Rivera and Len Taing.

- *XSmiles* (http://www.xsmiles.org/download.html) is an XForms browser to render the XForms markup.

- *formsPlayer* (http://www.formsplayer.com/download/) is an XForms plug-in for Internet Explorer V6.0. We used formsPlayer to test our sample applications in this tutorial.

- Visit the *developerWorks Java technology zone* (http://www-136.ibm.com/developerworks/java/) for hundreds of articles and tutorials about every aspect of Java programming.

- Visit *The Developer Bookstore* (http://devworks.krcinfo.com/) for a comprehensive listing of technical books, including hundreds of *Java-related titles*.

Your feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .