**Final exam notes**

**JSF**

**JSF has these parts:**

• A set of prefabricated UI (user interface) components

• An event-driven programming model

• A component model that enables third-party developers to supply

additional components

#{user.name}

@Named@ Named("user") //

@ManagedBean @ManagedBean(name="user")

A Java ***bean***is a class that exposes properties and events to a framework, such as

JSF. A *property* is a named value of a given type that can be read and/or written.

<h:commandButton value="Login" action="welcome"/>

<h:outputText id="out" value="#{user.greeting}"/>

<h:inputText value="#{user.name}" id="name"/>

The h:inputText, h:inputSecret, and h:commandButton tags correspond to the text

field, password field, and submit button.

A *bean* is a Java class that exposes properties, by following a simple naming

convention for the getter and setter methods. The code is in the file

UserBean.java (see Listing 1–3). Note the @Named or @ManagedBean annotation that

specifies the name by which an object of this class is referenced in the JSF

pages. (For compatibility reasons, there are two alternative annotations

for naming a bean. @Named is the best choice with a Java EE 6 compliant

application server. @ManagedBean is intended for use with legacy application

servers and standalone servlet runners.)

A managed bean is a Java bean that can be accessed from a JSF page. A managed

bean must have a name and a scope. The bean in our example has name user and session

scope.

A JSF page is similar to an HTML form. Note the following differences:

• Your page must be properly formatted XHTML. Unlike a browser, the JSF

implementation is not forgiving of syntax errors.

• You use h:head, h:body, and h:form instead of head, body, and form.

• Instead of using the familiar input HTML tags, use h:inputText, h:inputSecret,

and h:commandButton.

<servlet>

**9.** <servlet-name>Faces Servlet</servlet-name>

**10.** <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>

**11.** </servlet>

**12.** <servlet-mapping>

**13.** <servlet-name>Faces Servlet</servlet-name>

**14.** <url-pattern>/faces/\*</url-pattern>

**15.** </servlet-mapping>

**16.** <welcome-file-list>

**17.** <welcome-file>faces/index.xhtml</welcome-file>

**18.** </welcome-file-list>

**19.** <context-param>

**20.** <param-name>javax.faces.PROJECT\_STAGE</param-name>

**21.** <param-value>Development</param-value>

**22.** </context-param>

**23.** </web-app>

Asynchronous JavaScript with XMLHttpRequest (Ajax) is a technology for

updating a web page in the browser client without submitting a form and rendering

the response.

**The most important services that JSF provides:**

* MVC architecture
* Data conversion
* Validation and error handling
* Internationalization
* Custom components
* Ajax support
* Alternative renderers

The JSF specification defines six distinct *phases*:

1. Restore View

2. Apply Request Values

3. Process Validations

4. Update Model Values

5. Invoke Application

6. Render Response

The ***Restore View***phase retrieves the component tree for the requested page if it

was displayed previously or constructs a new component tree if it is displayed

for the first time.

If there are no request values, the JSF implementation skips ahead to the

***Render Response***phase. This happens when a page is displayed for the first

time.

Otherwise, the next phase is the ***Apply Request Values***phase. In this phase, the

JSF implementation iterates over the component objects in the component tree.

Each component object checks which request values belong to it and stores

them.

The values stored in the component are called “local values”. When you design

a JSF page, you can attach validators that perform correctness checks on the

local values. These validators are executed in the ***Process Validations***phase. If

validation passes, the JSF life cycle proceeds normally. However, when conversion

or validation errors occur, the JSF implementation invokes the Render

Response phase directly, redisplaying the current page so that the user has

another chance to provide correct inputs.

After the converters and validators have done their work, it is assumed that

it is safe to update the model data. During the ***Update Model Values***phase, the

local values are used to update the beans that are wired to the components.

In the ***Invoke Application***phase, the action method of the button or link component

that caused the form submission is executed. That method can carry out

arbitrary application processing. It returns an outcome string that is passed to

the navigation handler. The navigation handler looks up the next page.

Finally, the ***Render Response***phase encodes the response and sends it to the

browser. When a user submits a form, clicks a link, or otherwise generates a

new request, the cycle starts anew.

**The JSF implementation does the following:**

• Creates and discards beans as needed (hence the term “managed beans”)

• Reads bean properties when displaying a web page

• Sets bean properties when a form is posted

***Session Scope***

Recall that the HTTP protocol is *stateless*. The browser sends a request to the

server, the server returns a response, and then neither the browser nor the

server has any obligation to keep any memory of the transaction.

***Request Scope***

The *request scope* is short-lived. It starts when an HTTP request is submitted

and ends after the response is sent back to the client. If you place a managed

bean into request scope, a new instance is created with each request. It is worth

considering request scope if you are concerned about the cost of session scope

storage.

***Application Scope***

The *application scope* persists for the entire duration of the web application. That

scope is shared among all requests and all sessions.

***Conversation Scope***

A conversation scope ranges over a set of related pages. This provides data

persistence until a particular goal has been reached, without having to store the

data for the entire session. A conversation is tied to a particular browser page

or tab. A single session can have multiple conversations in different pages. This

is an important requirement in practice.

@Named

@SessionScoped

public class EditBean {

@Inject private UserBean currentUser;

...

}

@ManagedBean

@SessionScoped

public class EditBean implements Serializable {

**@ManagedProperty(value="#{user}")**

private UserBean currentUser;

public void setCurrentUser(UserBean newValue) { currentUser = newValue; }

. . .

}

**Action**

**ActionListener**

**ValueChangeListener**

**Static Navigation-** clicking a particular

button always selects a fixed JSF page for rendering the response.

<h:commandButton label="Login" action="welcome"/>

**Dynamic Navigation -** To implement dynamic navigation, the submit button must have a *method*

*expression*, such as:

<h:commandButton label="Login" action="#{loginController.verifyUser}"/>

Here is an example of an action method:

String verifyUser() {

if (...)

return "success";

else

return "failure";

}

This is achieved by adding navigation-rule entries into faces-config.xml. Here is a

typical example:

<navigation-rule>

<from-view-id>/index.xhtml</from-view-id>

<navigation-case>

<from-outcome>success</from-outcome>

<to-view-id>/welcome.xhtml</to-view-id>

</navigation-case>

</navigation-rule>

This rule states that the success outcome navigates to /welcome.xhtml if it occurred

inside /index.xhtml.

You can merge navigation rules with the same from-view-id. Here is an example:

<navigation-rule>

<from-view-id>/index.xhtml</from-view-id>

<navigation-case>

<from-outcome>success</from-outcome>

<to-view-id>/welcome.xhtml</to-view-id>

</navigation-case>

<navigation-case>

<from-outcome>failure</from-outcome>

<to-view-id>/newuser.xhtml</to-view-id>

</navigation-case>

</navigation-rule>

Single rule:

rule:

<navigation-rule>

<navigation-case>

<from-outcome>logout</from-outcome>

<to-view-id>/loggedOut.xhtml</to-view-id>

</navigation-case>

</navigation-rule>

Redirection:

If you don’t use navigation rules, add the string

?faces-redirect=true

to the outcome string, for example:

<h:commandButton label="Login" action="welcome?faces-redirect=true"/>

In a navigation rule, you add a redirect element after to-view-id, as follows:

<navigation-case>

<from-outcome>success</from-outcome>

<to-view-id>/success.xhtml</to-view-id>

<redirect/>

</navigation-case>

Most of the core tags represent objects you add to components, such as the

following:

• Attributes

• Parameters

• Facets

• Listeners

• Converters

• Validators

• Selection items

Chapter 7, we set the separator character for credit card digit groups like this:

<h:outputText value="#{payment.card}">

<f:attribute name="separator" value="-" />

</h:outputText>

Facelets

In addition to being a better view handler, Facelets supports a number of tags

for templating and other purposes.

Facelets tags can be grouped in these categories:

• Including content from other XHTML pages (ui:include)

• Building pages from templates (ui:composition, ui:decorate, ui:insert,

ui:define, ui:param)

• Creating custom components without writing Java code (ui:component,

ui:fragment)

• Miscellaneous utilities (ui:debug, ui:remove, ui:repeat)

To use Facelets tags, add the following namespace declaration to your JSF

page:

xmlns:ui=<http://java.sun.com/jsf/facelets>

**Facelet Tags**

**ui:include** Includes content from another XML file.

**ui:composition** When used without a template attribute, a composition is a

sequence of elements that can be inserted somewhere else. The

composition can have variable parts (specified with ui:insert

children).

When used with a template attribute, the template is loaded. The

children of this tag determine the variable parts of the template.

The template contents replaces this tag.

**ui:decorate** When used without a template attribute, ui:decorate specifies a

page into which parts can be inserted. The variable parts are

specified with ui:insert children.

When used with a template attribute, the template is loaded. The

children of this tag determine the variable parts of the template.

**ui:define** Defines content that is inserted into a template with a matching

ui:insert.

**ui:insert** Inserts content into a template. That content is defined inside the

tag that loads the template.

**ui:param** Specifies a parameter that is passed to an included file or a

template.

**ui:component** This tag is identical to ui:composition, except that it creates a

component that is added to the component tree.

**ui:fragment** This tag is identical to ui:decorate, except that it creates a

component that is added to the component tree.

**ui:debug** The ui:debug tag lets users display a debug window, with a

keyboard shortcut, that shows the component hierarchy for the

current page and the application’s scoped variables.

**ui:remove** JSF removes everything inside of ui:remove tags.

**ui:repeat** Iterates over a list, array, result set, or individual object.

The template also inserts a stylesheet into the head of the page. The stylesheet

defines the layout for the heading, the sidebar, and the main content.

You can specify default content inside the body of a ui:insert tag. For example:

<ui:insert name="header">

*Default header goes here*

</ui:insert>

The default is used if no replacement for header is specified when the template is

used.

It is common to use a ui:include tag to include default content from another file:

<ui:insert name="header">

<ui:include src="/sections/planetarium/header.xhtml">

</ui:insert>

To make use of a template, you use a ui:composition tag with a template attribute,

<ui:composition template="/templates/masterLayout.xhtml"

xmlns:ui="http://java.sun.com/jsf/facelets">

<ui:define name="windowTitle">

#{msgs.saturn}

</ui:define>

<ui:define name="content">

Saturn has rings made of ice and dust.

</ui:define>

</ui:composition>

<h:head>

**9.** <title><ui:insert name="windowTitle"/></title>

**10.** <h:outputStylesheet library="css" name="styles.css"/>

**11.** </h:head>

**12.**

**13.** <h:body>

**14.** <div id="heading">

**15.** <ui:insert name="heading">

**16.** <ui:include src="/sections/planetarium/header.xhtml"/>

**17.** </ui:insert>

**18.** </div>

**19.**

**20.**

**21.** <div id="sidebarLeft">

**22.** <ui:insert name="sidebarLeft">

**23.** <ui:include src="/sections/planetarium/sidebarLeft.xhtml"/>

**24.** </ui:insert>

**25.** </div>

**26.**

**27.** <div id="content">

**28.** <ui:insert name="content"/>

**29.** </div>

**30.** <ui:debug/>

The ui:define tags inside the ui:composition tag correspond to the ui:insert tags of

the template shown in Listing 5–1. For example,

<ui:define **name="content"**>

Saturn has rings made of ice and dust.

</ui:define>

in the composition corresponds to

<ui:insert name="content"/>

in the template.

**mars.xhtml**

<ui:composition template="/templates/masterLayout.xhtml">

<ui:define name="windowTitle">

**#{msgs.mars}**

</ui:define>

<ui:define name="content">

**Scientists believe that life may have existed on Mars in the past.**

</ui:define>

</ui:composition>

**186**

**planetarium.xhtml**

<ui:composition template="/templates/masterLayout.xhtml">

<ui:define name="windowTitle">

<ui:define name="content">

**#{msgs.planetariumWelcome}**

</ui:define>

</ui:composition>

**login.xhtml**

<ui:composition template="/templates/masterLayout.xhtml">

<ui:define name="windowTitle">

**#{msgs.loginTitle}**

</ui:define>

**<ui:define name="heading">**

**<ui:include src="/sections/login/header.xhtml"/>**

**</ui:define>**

**<ui:define name="sidebarLeft">**

**<ui:include src="/sections/login/sidebarLeft.xhtml"/>**

**</ui:define>**

<ui:define name="content">

<**h:form>**

**<h:panelGrid columns="2">**

**#{msgs.namePrompt}**

**<h:inputText id="name" value="#{user.name}"/>**

**#{msgs.passwordPrompt}**

**<h:inputSecret id="password" value="#{user.password}"/>**

**</h:panelGrid>**

**<p>**

**<h:commandButton value="#{msgs.loginButtonText}"**

**action="planetarium"/>**

**</p>**

**</h:form>**

</ui:define>

</ui:composition>

<body>

**8.** <ui:composition>

**9.** <div class="welcome">

**10.** #{msgs.loginWelcome}

**11.** <div class="welcomeImage">

**12.** <h:graphicImage library="images" name="Saturn.gif"/>

**13.** </div>

**14.** </div>

**15.** </ui:composition>

**16.** </body>

**Login page content**

<ui:define name="content">

**22.** <h:form>

**23.** <h:panelGrid columns="2">

**24.** #{msgs.namePrompt}

**25.** <h:inputText id="name" value="#{user.name}"/>

**26.** #{msgs.passwordPrompt}

**27.** <h:inputSecret id="password" value="#{user.password}"/>

**28.** </h:panelGrid>

**29.** <p>

**30.** <h:commandButton value="#{msgs.loginButtonText}"

**31.** action="planetarium"/>

**32.** </p>

**33.** </h:form>

**34.** </ui:define>

**Decorators**

Decorators are a more content-centric

approach. You write your pages as usual, but you surround the contents with a

ui:decorate tag that has a template attribute.

**<ui:decorate template="/templates/masterDecorator.xhtml">**

<!-- Contents to be decorated -->

<h:form>

<h:panelGrid columns="2">

#{msgs.namePrompt}

<h:inputText id="name" value="#{user.name}"/>

#{msgs.passwordPrompt}

<h:inputSecret id="password" value="#{user.password}"/>

**194 Chapter 5** ■ **Facelets**

</h:panelGrid>

<p>

<h:commandButton value="#{msgs.loginButtonText}"

action="planetarium"/>

</p>

</h:form>

**</ui:decorate>**

Note the <ui:insert/> tag without a name attribute. It inserts all children of the ui:decorate tag.

With decorators, as with compositions, you can override defaults with ui:define

tags, like this:

<ui:decorate template="/templates/masterDecorator.xhtml">

<ui:define name="heading">*Special Header*</ui:define>

*Body*

</ui:decorate

The difference between the composition and decorator

Facelets simply considers them complementary

constructs: ui:composition trims all surrounding contents, whereas

ui:decorator doesn’t (and therefore requires a ui:composition in the template).

***Parameters(ui:parameter is for getting the data from the bean)***

When you invoke a template, you can supply arguments in two ways: with

ui:define and with the ui:param tag. As you have already seen, ui:define is used to

provide markup that is inserted into the template. In contrast, ui:param sets an

EL variable for use in a template, like this:

<ui:composition template="templates/masterTemplate.xhtml">

<ui:**param** name="currentDate" value="#{someBean.currentDate}"/>

</ui:composition>

In the corresponding template, you can access the parameter with an EL

expression, like this:

...

<body>

Today's date: #{currentDate}"/>

</body>

...

The ui:param tag can also be used as a child of a ui:include tag.

package com.corejsf;

**2.**

**3.** import java.io.Serializable;

**4.** import javax.inject.Named;

**5.** // or import javax.faces.bean.ManagedBean;

**6.** import javax.enterprise.context.RequestScoped;

**7.** // or import javax.faces.bean.RequestScoped;

**8.**

**9.** @Named // or @ManagedBean

**10.** @RequestScoped

**11.** public class Planetarium implements Serializable {

**12.** private String selectedPlanet;

**13.**

**14.** public String getSelectedPlanet() { return selectedPlanet; }

**15.**

**Listing 5–9** planets/web/WEB-INF/tags/corejsf/planet.xhtml

**Listing 5–10** planets/src/com/corejsf/Planetarium.java

**Custom Tags 197**

Implementing a custom Facelets tag with JSF 2.0 is a two-step process:

1. Implement the custom tag (or component) in an XHTML file.

2. Declare the custom tag in a tag library descriptor

**16.** public String changePlanet(String newValue) {

**17.** selectedPlanet = newValue;

**18.** return selectedPlanet;

**19.** }

Tag library custom file

<?xml version="1.0"?>

**2.** <!DOCTYPE facelet-taglib PUBLIC

**3.** "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"

**4.** "http://java.sun.com/dtd/facelet-taglib\_1\_0.dtd">

**5.** <facelet-taglib>

**6.** <namespace>http://corejsf.com/facelets</namespace>

**7.** <tag>

**8.** <tag-name>planet</tag-name>

**9.** <source>tags/corejsf/planet.xhtml</source>

**10.** </tag>

**11.** </facelet-taglib>

Next, specify the location of the tag library file in web.xml:

<context-param>

<param-name>**facelets.LIBRARIES**</param-name>

<param-value>**/WEB-INF/corejsf.taglib.xml**</param-value>

</context-param>

Fragment

<ui:fragment rendered="#{name == planetarium.selectedPlanet}">

*Conditionally included children*

</ui:fragment>

Commenting in facelets:

**<ui:remove>**

<h:commandButton id="loginButton"

value="#{msgs.loginButtonText}"

action="planetarium"/>

**</ui:remove>**

**Data Table**

**How to use Data Table**

<h:dataTable **value="#{items}" var="item"**>

<h:column>

<!-- left column components -->

#{**item.aPropertyName**}

</h:column>

<h:column>

<!-- next column components -->

<h:commandLink value=**"**#{**item.anotherPropertyName**}**"** action="..."/>

</h:column>

<!-- add more columns, as desired -->

</h:dataTable>

<h:head>

**7.** <title>#{msgs.windowTitle}</title>

**8.** </h:head>

**9.** <h:body>

**10.** #{msgs.pageTitle}

**11.** <h:form>

**12.** <h:dataTable value="#{tableData.names}" var="name">

**13.** <h:column>

**14.** #{name.last},

**15.** </h:column>

**16.**

**17.** <h:column>

**18.** #{name.first}

**19.** </h:column>

**20.** </h:dataTable>

**21.** </h:form>

**22.** </h:body>

Column headers and footers are specified with facets, as shown here:

<h:dataTable>

...

<h:column **headerClass**="columnHeader"

**footerClass**="columnFooter">

<f:**facet** name="**header**">

<!-- header components go here -->

</f:facet>

<!-- column components go here -->

<f:**facet** name="**footer**">

<!-- footer components go here -->

</f:facet>

</h:column>

...

</h:dataTable>

**Data Table Bean**

@Named // or @ManagedBean

**13.** @SessionScoped

**14.** public class TableData implements Serializable {

**15.** private static final Name[] names = new Name[] {

**16.** new Name("William", "Dupont"),

**17.** new Name("Anna", "Keeney"),

**18.** new Name("Mariko", "Randor"),

**19.** new Name("John", "Wilson")

**20.** };

**21.**

**22.** public Name[] getNames() { return names;}

**23.** }

Column headers and footers are specified with facets, as shown here:

<h:dataTable>

...

<h:column **headerClass**="columnHeader"

**footerClass**="columnFooter">

<f:**facet** name="**header**">

<!-- header components go here -->

</f:facet>

<!-- column components go here -->

<f:**facet** name="**footer**">

<!-- footer components go here -->

</f:facet>

</h:column>

...

</h:dataTable>

<h:dataTable ... **captionClass**="caption">

<f:facet name="caption">An Array of Names:</f:facet>

...

</h:dataTable>

***Styles by Row***

You can use the rowClasses attribute to specify CSS classes by rows instead of

columns, as illustrated in Figure 6–6. That data table is implemented like this:

<h:dataTable value="#{order.all}" var="order"

styleClass="**orders**"

headerClass="**ordersHeader**"

rowClasses="**oddRow**,**evenRow**">

<table>

<ui:repeat value="#{tableData.names}" var="name">

<tr>

<td>#{name.last},</td>

<td>#{name.first}</td>

</tr>

</ui:repeat>

</table>

The index property can be used for row numbers:

<table>

<ui:repeat value="#{tableData.names}" var="name" varStatus="status">

<tr>

<td>**#{status.index + 1}**</td>

<td>#{name.last},</td>

<td>#{name.first}</td>

</tr>

</ui:repeat>

</table>

Conversion and Validation

The converted values are not immediately transmitted to the beans that make

up the business logic. Instead, they are first stored inside the component

objects as *local values*. After conversion, the local values are *validated*.

The Update Model Values phase starts only if all validations

are successful.

We attach a converter to the text field and tell it to format the current value

with at least two digits after the decimal point:

<h:inputText value="#{payment.amount}">

<f:convertNumber minFractionDigits="2"/>

</h:inputText>

Pattern date/year:

<h:inputText value="#{payment.date}">

<f:convertDateTime pattern="MM/yyyy"/>

</h:inputText>

Currency:

<h:outputText value="#{payment.amount}">

<f:convertNumber type="currency"/>

</h:outputText>

**The converter Attribute**

An alternate syntax for attaching a converter to a component is to add the converter

attribute to the component tag. You specify the ID of the converter like this:

<h:outputText value="#{payment.date}" converter="javax.faces.DateTime"/>

This is equivalent to using f:convertDateTime with no attributes:

<h:outputText value="#{payment.date}">

<f:convertDateTime/>

</h:outputText>

A third way of specifying the converter would be as follows:

<h:outputText value="#{payment.date}">

<f:converter converterId="javax.faces.DateTime"/>

</h:outputText>

All JSF implementations must define a set of converters with predefined IDs:

• javax.faces.DateTime (used by f:convertDateTime)

• javax.faces.Number (used by f:convertNumber)

***Conversion Errors***

When a conversion error occurs, the JSF implementation carries out the following

actions:

• The component whose conversion failed posts a *message* and declares

itself invalid. (You will see in the following sections how to display the

message.)

• The JSF implementation redisplays the current page immediately after

the Process Validations phase has completed. The redisplayed page contains

all values that the user provided—no user input is lost.

**Types of convertors**

<f:convertNumber minFractionDigits="2"/>

<f:convertDateTime pattern="MM/yyyy"/>

<f:convertNumber type="currency"/>

<f: converter converterId=”ClassName”/>

**Displaying Error Messages**

You want your users to see the messages that are caused by conversion

and validation errors. Add h:message tags whenever you use converters and

validators.

Normally, you want to show the error messages next to the components that

reported them (see Figure 7–4). Give an ID to the component and reference that

ID in the h:message tag. As of JSF 1.2, you also need to supply a component label

that is displayed in the error message:

<h:inputText id="amount" label="#{msgs.amount}" value="#{payment.amount}"/>

<h:message for="amount"/>

By default, the h:message tag shows the detail and hides the summary. If you

want to show the summary message instead, use these attributes:

<h:message for="amount" showSummary="true" showDetail="false"/>

Usually, you will want to show error messages in a different color. You use the

styleClass or style attribute to change the appearance of the error message:

<h:messages styleClass="errorMessage"/>

or

<h:message for="amount" style="color:red"/>

The default value of the layout attribute for h:messages is "list", which yields an

unnumbered list whose appearance you can control with a stylesheet. Alternatively,

you can up the messages vertically by using:

<h:messages layout="table"/>

Validating String Lengths and Numeric Ranges

It is easy to use JSF validators within JSF pages—add validator tags to the body

of a component tag, like this:

<h:inputText id="card" value="#{payment.card}">

<f:validateLength minimum="13"/>

</h:inputText>

<h:inputText id="amount" value="#{payment.amount}">

<f:validateLongRange minimum="10" maximum="10000"/>

</h:inputText>

Required

***Checking for Required Values***

To check that a value is supplied, you can nest a validator inside the input component

tag:

<h:inputText id="date" value="#{payment.date}">

<f:validateRequired/>

</h:inputText>

Alternatively, yo

Validators

<h:inputText id="card" value="#{payment.card}">

<f:validator validatorId="javax.faces.validator.LengthValidator">

<f:attribute name="minimum" value="13"/>

</f:validator>

</h:inputText>

<h:inputText id="card" value="#{payment.card}" required="true"

requiredMessage="#{msgs.cardRequired}"

validatorMessage="#{msgs.cardInvalid}">

<f:validateLength minimum="13"/>

</h:inputText>

implement a Cancel button like this:

<h:commandButton value="Cancel" action="cancel" immediate="true"/>

Bean Validation Framework

public class PaymentBean {

**@Size(min=13)** private String card;

**@Future** public Date getDate() { ... }

...

}

<h:inputText value="#{payment.card}">

<f:converter converterId="com.corejsf.Card"/>

</h:inputText>

Or, more succinctly, we can use the converter attribute:

<h:inputText value="#{payment.card}" converter="com.corejsf.Card"/>

Alternatively, if you are confident that your converter is appropriate

***Getting Error Messages from Resource Bundles***

Of course, for proper localization, you will want to retrieve the error messages

from a message bundle.

Doing that involves some busywork with locales and class loaders:

1. Get the current locale.

FacesContext context = FacesContext.getCurrentInstance();

UIViewRoot viewRoot = context.getViewRoot();

Locale locale = viewRoot.getLocale();

**javax.faces.application.FacesMessage**

**javax.faces.convert.ConverterException**

**282 Chapter 7** ■ **Conversion and Validation**

2. Get the current class loader. You need it to locate the resource bundle.

ClassLoader loader = Thread.currentThread().getContextClassLoader();

3. Get the resource bundle with the given name, locale, and class loader.

ResourceBundle bundle = ResourceBundle.getBundle(bundleName, locale, loader);

4. Get the resource string with the given ID from the bundle.

String resource = bundle.getString(resourceId);

Faces message

The com.corejsf.util.Messages class has a static method, getMessage, that returns a

FacesMessage with a given bundle name, resource ID, and parameters:

FacesMessage message

= com.corejsf.util.Messages.getMessage(

"com.corejsf.messages", "badCreditCardCharacter",

new Object[] { new Character(invalidCharacter) });

You can pass null for the parameter array if the message does not contain

placeholders.

When attaching the converter, also nest an f:attribute tag inside the component:

<h:outputText value="#{payment.card}">

<f:converter converterId="CreditCard"/>

<f:attribute name="separator" value="-"/>

</h:outputText>

In the converter, retrieve the attribute as follows:

separator = (String) component.getAttributes().get("separator");

Validation fails

If validation fails, generate a FacesMessage that describes the error, construct a

ValidatorException from the message, and throw it:

if (validation fails) {

FacesMessage message = ...;

message.setSeverity(FacesMessage.SEVERITY\_ERROR);

throw new ValidatorException(message);

}

Alternatively, you can register the validator in a configuration file (such as

faces-config.xml), like this:

<validator>

<validator-id>com.corejsf.Card</validator-id>

<validator-class>com.corejsf.CreditCardValidator</validator-class>

</validator>

You specify the validator ID in the f:validator tag—for example, the following

code fragment uses the credit card validator discussed above:

<h:inputText id="card" value="#{payment.card}" required="true">

<f:converter converterId="com.corejsf.Card"/>

<f:validator validatorId="com.corejsf.Card"/>

</h:inputText>

The f:validator tag uses the validator ID to look up the corresponding class, creates

an instance of that class if necessary, and invokes its validate method.

To carry out this approach, the validator of the last component needs to have

access to the other components. You can achieve that access by giving ID values

to the other components. Then you can use the findComponent method of the

UIComponent class to locate them:

public class BackingBean {

...

public void validateDate(FacesContext context, UIComponent component,

Object value) {

UIInput dayInput = (UIInput) component.findComponent("day");

UIInput monthInput = (UIInput) component.findComponent("month");

int d = ((Integer) dayInput.getLocalValue()).intValue();

int m = ((Integer) monthInput.getLocalValue()).intValue();

int y = ((Integer) value).intValue();

if (!isValidDate(d, m, y)) {

FacesMessage message = ...;

throw new ValidatorException(message);

}

}

...

}

<h:outputText value="#{payment.card}">

<corejsf:convertCreditcard separator="-"/>

</h:outputText>

<tag>

<tag-name>convertCreditCard</tag-name>

<converter>

<converter-id>com.corejsf.CreditCard</converter-id>

</converter>

</tag>

<tag>

<tag-name>validateCreditCard</tag-name>

<validator>

<validator-id>com.corejsf.CreditCard</validator-id>

</validator>

</tag>

</facelet-taglib>

When implementing converters or validators that have state, you need to make

sure that the state can be saved. The easiest way of accomplishing that is to

implement the Serializable interface and follow the usual rules for Java serialization.

<h:inputText id="card" value="#{payment.card}" required="true">

<corejsf:validateCreditCard errorDetail="#{msgs.creditCardError}"/>

</h:inputText>

**Types of validation**

<f:validateLongRange maximum =”100” minimum = “100”>

**<f:convertDateTime pattern=*"dd/MM/yyyy"/>***

**<f:validateRegex pattern=*"(([0-9]{3})/[a-z]{3})" />***

<**f:validateDoubleRange** minimum="3.0" maximum="4.5" for="ustaRankingId"/>

*<!--f:validateRegex pattern="[3-4]\.?[05]" for="ustaRankingId"/-->*

<h:inputText value="#{carInfo.brand}" id="inputBrand" validator="#{carInfo.validateBrand}"/>

<h:message style="color: red" id="brandError" for="inputBrand"/>

@FacesValidator("licenceValidator")

public class LicenceplateValidator implements Validator {

@Override

public void validate(FacesContext fc, UIComponent uic, Object o) throws ValidatorException {

Pattern mask = null;

mask = Pattern.compile("[A-Z]{2}[-][0-9]{2}[-][A-Z]{2}");

String zipField = (String)o;

Matcher matcher = mask.matcher(zipField);

if (!matcher.matches()){

FacesMessage message = new FacesMessage();

message.setDetail("Licence plate is not valid");

message.setSummary("Licence plate is not valid");

message.setSeverity(FacesMessage.SEVERITY\_ERROR);

throw new ValidatorException(message);

}

}

}

public void validateBrand(FacesContext context, UIComponent toValidate, Object value) {

String brand = (String) value;

if (!(brand.equalsIgnoreCase("Mercedes") || brand.equalsIgnoreCase("BMW"))) {

((UIInput) toValidate).setValid(false);

FacesMessage message = new FacesMessage("Invalid Brand");

context.addMessage(toValidate.getClientId(context), message);

}

}

Brand

<h:inputText value="#{carInfo.brand}" id="inputBrand"

validator="#{carInfo.validateBrand}"/>

<h:message style="color: red" id="brandError" for="inputBrand"/>

Licence nr.

<h:inputText value="#{carInfo.licence}" id="inputLicence">

<f:validator validatorId="licenceValidator"/>

</h:inputText>

<h:message style="color: red" id="licenceError" for="inputLicence"/>

**Event handling**

<h:selectOneMenu valueChangeListener="#{form.countryChanged}"...>

...

</h:selectOneMenu>

public void checkAirline(ValueChangeEvent e) {

if(e.getNewValue().equals("All"))

{

alist = new ArrayList<>();

alist = allList;

}

<h:selectOneMenu id = "AirID" valueChangeListener="#{airlineBean.checkAirline}" onchange="submit()">

<h:selectOneMenu id = "AirID" valueChangeListener="#{airlineBean.checkAirline}" onchange="submit()">

<f:selectItem itemValue="All" itemLabel="All" />

<f:selectItem itemValue="KLM" itemLabel="KLM" />

<f:selectItem itemValue="North West" itemLabel="North West" />

<f:selectItem itemValue="United Airlines" itemLabel="United Airlines" />

</h:selectOneMenu>

<br/><br/>

<h:dataTable value="#{airlineBean.alist}" var="thelist" border="1">

<h:column> #{thelist.from}</h:column>

<h:column> #{thelist.to}</h:column>

<h:column> #{thelist.date}</h:column>

<h:column> #{thelist.airline}</h:column>

<h:column> #{thelist.price}</h:column>

</h:dataTable>

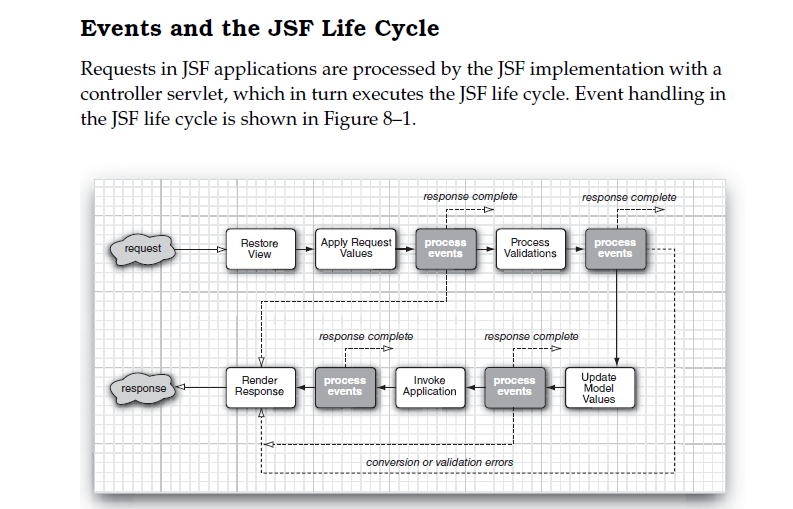
JSF supports four kinds of events:

• Value change events

• Action events

• Phase events

• System events



Value Change Listeners

<h:selectOneMenu value="#{form.country}" **onchange="submit()"**

**valueChangeListener**="#{form.countryChanged}">

<f:selectItems value="#{form.countries}" var="loc"

itemLabel="#{loc.displayCountry}" itemValue="#{loc.country}"/>

</h:selectOneMenu>

Here, #{form.countries} is bound to an array of Locale objects.

When a user selects a country from the menu, the JavaScript submit function is

invoked to submit the menu’s form, which subsequently invokes the JSF life

cycle. After the Process Validations phase, the JSF implementation invokes the

form bean’s countryChanged method. That method changes the view root’s locale,

according to the new country value:

public void countryChanged(ValueChangeEvent event) {

for (Locale loc : countries)

if (loc.getCountry().equals(event.getNewValue()))

FacesContext.getCurrentInstance().getViewRoot().setLocale(loc);

}

UIComponent getComponent()

Returns the input component that triggered the event.

• Object getNewValue()

Returns the component’s new value, after the value has been converted

and validated.

• Object getOldValue()

Returns the component’s previous value.

<h:selectOneMenu value="#{form.country}" onchange="submit()"

**27.** valueChangeListener="#{form.countryChanged}">

**28.** <f:selectItems value="#{form.countries}" var="loc"

**29.** itemLabel="#{loc.displayCountry}" itemValue="#{loc.country}"/>

**30.** </h:selectOneMenu>

@ManagedBean(name="form")

public void countryChanged(ValueChangeEvent event) {

**39.** for (Locale loc : countries)

**40.** if (loc.getCountry().equals(event.getNewValue()))

**41.** FacesContext.getCurrentInstance().getViewRoot().setLocale(loc);

**42.** }

<application>

**8.** <resource-bundle>

**9.** <base-name>com.corejsf.messages</base-name>

**10.** <var>msgs</var>

**11.** </resource-bundle>

**12.** </application>

Properties

windowTitle=Using Value Change Events

**2.** pageTitle=Please fill in your address

**3.**

**4.** streetAddressPrompt=Address

**5.** cityPrompt=City

**6.** statePrompt=State

**7.** countryPrompt=Country

**8.** submit=Submit address

**Action Events**

( page 304-317,320-325, 328-345)

For links and for button we use actionListener

<h:commandLink **actionListener**="#{bean.linkActivated}">

</h:commandLink>

<h:commandButton image="mountrushmore.jpg" action="#{rushmore.navigate}">

**<f:actionListener type="com.corejsf.RushmoreListener"/>**

**<f:actionListener type="com.corejsf.ActionLogger"/>**

</h:commandButton>

<h:inputText value="#{form.streetAddress}" **required="true"**/>

But that results in an error *when we select a country* without

**Immediate Components**

It will fire the event anyway. It will skip even the wrong input.

We specify immediate components with the immediate attribute, which is available

to all input and command components:

<h:selectOneMenu value="#{form.country}" **onchange="submit()" immediate="true"**

valueChangeListener="#{form.countryChanged}">

<f:selectItems value="#{form.countryNames}"/>

</h:selectOneMenu>

To prevent validations for the other

components in the form, we have one more thing to do, which is to call the

renderResponse method of the FacesContext class at the end of our value change

listener, like this:

private static final String US = "United States";

...

public void countryChanged(ValueChangeEvent event) {

FacesContext context = FacesContext.getCurrentInstance();

if (US.equals((String) event.getNewValue()))

context.getViewRoot().setLocale(Locale.US);

else

context.getViewRoot().setLocale(Locale.CANADA);

**context.renderResponse();**

}

**23.** public void handleMouseClick(ActionEvent e) {

**24.** FacesContext context = FacesContext.getCurrentInstance();

**25.** String clientId = e.getComponent().getClientId(context);

**26.** Map<String, String> requestParams

**27.** = context.getExternalContext().getRequestParameterMap();

The f:actionListener and f:valueChangeListener tags are analagous to the actionListener

and valueChangeListener attributes.

<h:selectOneMenu value="#{form.country}" onchange="submit()"

**valueChangeListener="#{form.countryChanged}"**>

<f:selectItems value="#{form.countryNames}"/>

</h:selectOneMenu>

**Event Listener Tags 319**

Alternatively, we could use f:valueChangeListener, like this:

<h:selectOneMenu value="#{form.country}" onchange="submit()">

**<f:valueChangeListener type="com.corejsf.CountryListener"/>**

<f:selectItems value="#{form.countryNames}"/>

</h:selectOneMenu>

<h:selectOneMenu value="#{form.country}" **onchange="submit()" immediate="true"**

valueChangeListener="#{form.countryChanged}">

<f:selectItems value="#{form.countryNames}"/>

</h:selectOneMenu>

The call to renderResponse skips the rest of the life cycle—including validation of

the rest of the input components in the form—up to Render Response. Thus,

the other validations are skipped and the response is rendered normally (in

this case, the current page is redisplayed).

To summarize, you can skip validation when a value change event fires by

doing the following:

1. Adding an immediate attribute to your input tag

2. Calling the renderResponse method of the FacesContext class at the end of your

listener

**Phase Events**

The JSF implementation fires events, called *phase events*, before and after each

life cycle phase. Those events are handled by phase listeners. Unlike value

change and action listeners that you attach to individual components, a phase

listener is attached to the view root. You can specify a phase listener for an individual

page with a tag, such as the following, placed anywhere in that page:

<f:phaseListener type="com.corejsf.PhaseTracker"/>

Alternatively, you can specify global phase listeners in a faces configuration

file, like this:

<faces-config>

<lifecycle>

<phase-listener>com.corejsf.PhaseTracker</phase-listener>

</lifecycle>

</faces-config>

You implement phase listeners by means of the PhaseListener interface from the

javax.faces.event package. That interface defines three methods:

• PhaseId getPhaseId()

• void afterPhase(PhaseEvent)

• void beforePhase(PhaseEvent)

<f:view beforePhase="#{backingBean.beforeListener}">

...

</f:view>

There are four ways in which a class can receive system events:

• With the f:event tag:

<inputText value="#{...}">

<f:event name="postValidate" listener="#{bean.method}"/>

</inputText>

The method must have the signature

public void listener(ComponentSystemEvent) throws AbortProcessingException

This is the most convenient way for listening to component or view

events.

• With an annotation for a UIComponent or Renderer class:

@ListenerFor(systemEventClass=PreRenderViewEvent.class)

We discuss these classes in Chapter 11. This mechanism can be useful for

component developers.

• By being listed as a system event listener in faces-config.xml:

<application>

<system-event-listener>

<system-event-listener-class>*listenerClass*</system-event-listener-class>

<system-event-class>*eventClass*</system-event-class>

</system-event-listener>

</application>

This mechanism is useful for installing a listener to application events.

• By calling the subscribeToEvent method of the UIComponent or Application class.

This method is intended

Multi-Component Validation

**<f:event type="postValidate" listener="#{bb.validateDate}"/>**

#{msgs.day}

<h:inputText id="day" value="#{bb.day}" size="2" required="true"/>

#{msgs.month}

<h:inputText id="month" value="#{bb.month}" size="2" required="true"/>

#{msgs.year}

public void validateDate(ComponentSystemEvent event) {

UIComponent source = event.getComponent();

UIInput dayInput = (UIInput) source.findComponent("day");

UIInput monthInput = (UIInput) source.findComponent("month");

UIInput yearInput = (UIInput) source.findComponent("year");

int d = ((Integer) dayInput.getLocalValue()).intValue();

int m = ((Integer) monthInput.getLocalValue()).intValue();

int y = ((Integer) yearInput.getLocalValue()).intValue();

**if (!isValidDate(d, m, y)) {**

**FacesMessage message = com.corejsf.util.Messages.getMessage(**

**"com.corejsf.messages", "invalidDate", null);**

**message.setSeverity(FacesMessage.SEVERITY\_ERROR);**

**FacesContext context = FacesContext.getCurrentInstance();**

**context.addMessage(source.getClientId(), message);**

**context.renderResponse();**

**}**

Making decisions before rendering the view

<f:view>

<f:event type="preRenderView" listener="#{user.checkLogin}"/>

<h:head>

<title>...</title>

</h:head>

<h:body>

...

</h:body>

</f:view>

<f:view>

8. <f:event type="preRenderView" listener="#{user.checkLogin}"/>

9. <h:head>

10. <title>Welcome</title>

11. </h:head>

12. <h:body>

13. <h3><h:outputText value="Welcome to JavaServer Faces, #{user.name}!" /></h3>

14. <h:form>

15. <h:commandButton value="Logout" action="#{user.logout}" />

16. <h:commandButton value="Continue" action="enterDate" />

17. </h:form>

18. </h:body>

19. </f:view>

**Custom Components**

<util:debug /> - for debugging

You can use any name you want for the namespace, but the namespace’s value

must always start with [http://java.sun.com/jsf/composite**/**](http://java.sun.com/jsf/composite/).

**10.** <composite:implementation>

**11.** <div style="font-size: 1.2em; font-style: italic">

**12.** Request header:

**13.** </div>

**14.**

**15.** <p>#{header}</p>

**16.**

**17.** <div style="font-size: 1.2em; font-style: italic">

**18.** Request parameters:

**19.** </div>

**20.**

**Listing 9–2** simple-composite/web/resources/util/debug.xhtml

**21.** <p>#{param}</p>

**22.** </composite:implementation>

Like all composite components, the debug component has an *interface* and an

*implementation*. A composite component’s implementation is simply its

markup, whereas the component’s interface specifies component attributes so

developers can configure composite components.

<composite:**interface**>

<composite:**attribute** name="**image**"/>

<composite:**attribute** name="**actionMethod**"

**method-signature**="java.lang.String action()" />

</composite:interface>

<composite:implementation>

<h:form>

<h:commandLink action="#{**cc.attrs.actionMethod**}">

<h:graphicImage url="#{**cc.attrs.image**}" styleClass="icon" />

</h:commandLink>

**354 Chapter 9** ■ **Composite Components**

</h:form>

</composite:implementation>

**Attribute Types**

As we did with icons, you can specify attributes with value expressions, like

this:

<util:icon image="**#{resource['images:back.jpg']}**"

actionMethod="**#{user.logout}**" />

<util:icon image="**/composite-login/faces/javax.faces.resource/back.jpg?ln=images**"

actionMethod="#{user.logout}" />

If you want a composite attribute to represent a subclass of java.lang.Object, you

must tell JSF what the attribute’s type is. One way to do that is to use the methodsignature

attribute, of the composite:attribute tag, like we did in Listing 9–1. When

you specify a method signature for an attribute, JSF resolves the attribute’s

value expression to a *method expression*, instead of an *object*.

Another way to specify an attribute’s type is with the composite:attribute’s type

attribute. The type attribute must be specified with a fully qualified Java class

name. So, for example, if you wanted a date attribute whose value was a Date

object, you would do this:

<composite:attribute name ="date" type="java.util.Date" />.

<composite:interface>

**9.** <composite:attribute name="image" required="true" />

**10.** <composite:attribute name="doValidation" default="false" />

**11.** <composite:attribute name="styleClass" default="icon" />

**12.** <composite:attribute name="actionMe

thod"

**13.** method-signature="java.lang.String action()" />

**Listing 9–3** /composite-login/web/resources/util/icon.xhtml

**356 Chapter 9** ■ **Composite Components**

**14.** </composite:interface>

**15.**

**16.** <composite:implementation>

**17.** <h:form>

**18.** <h:commandLink action="#{cc.attrs.actionMethod}"

**19.** immediate="#{not cc.attrs.doValidation}">

**20.**

**21.** <h:graphicImage url="#{cc.attrs.image}"

**22.** styleClass="#{cc.attrs.styleClass}" />

**23.**

**24.** </h:commandLink>

**25.** </h:form>

**26.** </composite:implementation>

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

xmlns:h="http://java.sun.com/jsf/html"

xmlns:ui="http://java.sun.com/jsf/facelets"

xmlns:composite="http://java.sun.com/jsf/composite">

<composite:interface>

<composite:attribute user="**user**"/>

<composite:attribute name="**namePrompt**"/>

<composite:attribute name="**passwordPrompt**"/>

<composite:attribute name="**loginButtonText**"/>

<composite:attribute name="**loginAction**"

method-signature="java.lang.String action()"/>

</composite:interface>

<composite:implementation>

<h:form id="form">

<h:panelGrid columns="2">

#{**cc.attrs.namePrompt**}

<h:inputText id="name" value="#{**cc.attrs.user.name**}"/>

#{**cc.attrs.passwordPrompt**}

<h:inputSecret id="password" value="#{**cc.attrs.user.password**}"/>

</h:panelGrid>

<p>

<h:commandButton id="loginButton" value="#{**cc.attrs.loginButtonText**}"

action="#{**cc.attrs.loginAction**}"/>

</p>

</h:form>

</composite:implementation>

</html>If you want validation to occur when you click an icon, you must specify that

explicitly, like this: <util:icon **doValidation**="**true**".../>.

<util:login namePrompt="#{msgs.namePrompt}"

passwordPrompt="#{msgs.passwordPrompt}"

loginAction="#{user.login}"

loginButtonText="#{msgs.loginButtonText}"

**user="#{user}"**/>

<util:login namePrompt="#{msgs.namePrompt}"

passwordPrompt="#{msgs.passwordPrompt}"

**name="#{user.username}"**

**password="#{user.passwd}"**

loginAction="#{user.login}"

loginButtonText="#{msgs.loginButtonText}"/>

Now, instead of giving the login component a user managed bean, and letting

the component reference the bean’s properties, we specify the bean’s name and

password properties directly. That means that this new version of the login component

will work with managed beans with properties of any name.

You will need to make the following changes to the component definition:

<composite:interface>

<composite:attribute name="**name**"/>

<composite:attribute name="**password**"/>

...

</composite:interface>

<composite:implementation>

...

#{cc.attrs.namePrompt}

<h:inputText id="name" value="#{**cc.attrs.name**}"/>

#{cc.attrs.passwordPrompt}

<h:inputSecret id="password" value="#{**cc.attrs.password**}"/>

...

</composite:implementation>

Now page authors can associate individual properties of a managed bean, in

our case #{user.username} and #{user.passwd}, with inputs created by the login

component.

**Implementing a Component Class**

When you provide a custom component, you need to implement a *component*

*class* with the following responsibilities:

• To maintain the component state (for example, the minimum, maximum,

and current value of a spinner)

• To *encode* the user interface by writing markup (in the case of the spinner,

the HTML code for the input field and buttons)

• To *decode* HTTP requests (such as clicks on the spinner buttons)

You usually subclass one of the following three standard component classes:

• UIOutput, if your component displays a value, but does not allow the user

to edit it

• UIInput, if your component reads a value from the user (such as the

spinner)

• UICommand, if your component produces actions similar to a command

button or link

The UIComponent class manages several important categories of data. These include:

• A list of *child components*. For example, the children of the h:panelGrid component

are the components that are placed in the grid location. However,

a component need not have any children.

• A map of *facet components*. Facets are similar to child components, but

each facet has a key, not a position in a list. It is up to the component how

to lay out its facets. For example, the h:dataTable component has header

and footer facets.

A map of *attributes*. This is a general-purpose map that you can use to

store arbitrary key/value pairs.

• A map of *value expressions*. This is another general-purpose map that

you can use to store arbitrary value expressions. For example, if a spinner

tag has an attribute value="#{cardExpirationDate.month}", then the component

stores a ValueExpression object for the given value expression under the key

"value".

• A collection of *event listeners*. The listeners are notified when broadcasting

an event whose source is this component.

Spinner:

Here is how you use corejsf:spinner:

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

xmlns:h="http://java.sun.com/jsf/html"

**xmlns:corejsf="http://corejsf.com"**>

...

<**corejsf:spinner** value="#{cardExpirationDate.month}"

minimum="1" maximum="12" size="3"/>

...

<**corejsf:spinner** value="#{cardExpirationDate.year}"

minimum="1900" maximum="2100" size="5"/>

Components encode markup with three methods:

• encodeBegin()

• encodeChildren()

• encodeEnd()

The methods are called by JSF at the end of the life cycle, in the order in which

they are listed above

Here is a complete example of the HTML generated by the spinner:

<input type="text" **name="\_id1:monthSpinner"** value="1" size="3"/>

<input type="submit" **name="\_id1:monthSpinner.less"** value="<"/>

<input type="submit" **name="\_id1:monthSpinner.more"** value=">"/>

To understand the decoding process, keep in mind how a web application

works. The server sends an HTML form to the browser. When the user submits

the form, the browser sends back a POST request that consists of name/value

pairs. That POST request is the only data that the server can use to interpret the

user’s actions inside the browser.

The component type is an identifier for the component class that must get

mapped to the actual class. You can set the ID with an annotation of the

component class:

@FacesComponent("com.corejsf.Spinner")

public class UISpinner extends UIInput

You can set the file location in web.xml, like this:

<context-param>

<param-name>javax.faces.FACELETS\_LIBRARIES</param-name>

<param-value>/WEB-INF/corejsf.taglib.xml</param-value>

</context-param>

Place the tag library descriptor file into the META-INF directory.

2. If you need a faces-config.xml file, also place it into the META-INF directory.

3. Place any resources (such as images, scripts, or CSS files) into the

META-INF/resources directory.

4. Avoid name clashes by using an appropriate prefix for the global names,

such as component names, message keys, or loggers, used by your

implementation.

<h:body>

**13.** <h:form id="spinnerForm">

**14.** <h:outputText value="#{msgs.creditCardExpirationPrompt}"

**15.** styleClass="pageTitle"/>

**16.** <p/>

**17.** <h:panelGrid columns="3">

**18.** #{msgs.monthPrompt}

**19.** <corejsf:spinner value="#{cardExpirationDate.month}"

**20.** id="monthSpinner" minimum="1" maximum="12" size="3"/>

**21.** <h:message for="monthSpinner"/>

**22.** #{msgs.yearPrompt}

**23.** <corejsf:spinner value="#{cardExpirationDate.year}"

**24.** id="yearSpinner" minimum="1900" maximum="2100" size="5"/>

**25.** <h:message for="yearSpinner"/>

**26.** </h:panelGrid>

**27.** <p/>

**28.** <h:commandButton value="#{msgs.nextButtonPrompt}" action="next"/>

**29.** </h:form>

For example, here’s how you would attach an Ajax

behavior to a text input:

<h:inputText value="#{someBean.someProperty}">

<f:ajax event="keyup" render="someOtherComponentId"/>

</h:inputText>

**Ajax and JSF**

Conceptually, Ajax is simple. In fact, Ajax requests differ from regular HTTP

requests in only two ways:

1. Ajax *partially* processes forms on the server during the Ajax call.

2. Ajax *partially* renders Document Object Model (DOM) elements on the

client after the Ajax call returns from the server.

*JSF Ajax requests partially process components on the server, and partially render*

*components on the client when the request returns.*

when JSF executes a component on the server, it:

• Converts and validates the component’s value (if the component is an

input)

• Pushes valid input values to the model (if the component is wired to a

bean property)

• Executes actions and action listeners (if the component is an action)

So JSF 2.0 effectively has two life cycles: one that executes components, and one

that renders components. JSF always executes components first, and subsequently

renders them.

Here’s the recipe for using Ajax with JSF 2.0:

1. Associate a component and an event with an Ajax request.

2. Indentify components to execute on the server during the Ajax request.

3. Identify components to render after the Ajax request.

<h:inputText id="name" value="#{user.name}">

**<f:ajax event="blur" execute="@this" render="nameError"/>**

</h:inputText>

We could also execute and render multiple components, like this:

<h:inputText id="nameInput" value="#{user.name}">

<f:ajax event="blur" execute="**@this passwordInput**"

render="**nameError passwordError**"/>

</h:inputText>

<h:outputText id="**nameError**"/>

...

<h:inputText id="**passwordInput**"/>

<h:outputText id="**passwordError**" value="#{user.passwordError}"/>

The key to using Ajax with JSF is to keep in mind what it

means to execute a component on the server during an Ajax call

That default means we can omit execute="@this" from the preceding

markup, like this:

<h:inputText id="name" value="#{user.name}">

**<f:ajax event="keyup" render="echo"/>** <!-- execute="@this" is implicit -->

</h:inputText>

You can use the same keywords (@all, @this, @form, and @none) that

are valid for the execute attribute.

For a successful Ajax

request, JSF invokes the onevent function three times: when the Ajax request

begins, when it completes, and again after completion, for a successful request.

JSF invokes the onerror JavaScript function after an unsuccessful Ajax request.

Default AJAX Events:

Command buttons and links ------------🡪action

Text inputs, text areas, secret inputs, and all of the

select components-----------------------🡪valueChange

AJAX Event

<f:ajax **event="click"**>

<h:form>

...

<h:inputText id="name" value="#{user.name}">

...

<h:inputText id="password" value="#{user.password}"/>

...

<h:commandButton value="Submit" action="#{user.login}"/>

</h:form>

</f:ajax>

Here’s the markup for the name input, and an associated h:message tag:

<h:inputText id="name" value="#{user.name}" validator="#{user.validateName}">

<f:ajax event="keyup" render="nameError"/>

</h:inputText>

<h:message for="name" id="nameError" style="color: red"s/>

You can reduce the number of Ajax requests, by making an Ajax request only

when the name field loses focus, like this:

<h:inputText id="name" value="#{user.name}" validator="#{user.validateName}">

<f:ajax event="**blur**" render="nameError"/>

</h:inputText>

You can monitor Ajax requests with the f:ajax tag’s onevent attribute. That

attribute’s value must be a JavaScript function. JSF calls that function at each

stage of an Ajax request: begin, complete, and success.

<h:outputScript library="javascript" name="prototype-1.6.0.2.js"/>

<script type="text/javascript">

function **showProgress**(data) {

var inputId = data.source.id

var progressbarId = inputId.substring(0, inputId.length - "name".length)

+ "pole";

if (data.status == "begin")

Element.show(progressbarId);

else if (data.status == "success")

Element.hide(progressbarId);

}

<h:panelGrid columns="2">

#{msgs.namePrompt}

<h:panelGroup>

<h:**inputText** id="name" value="#{user.name}"

**validator**="#{user.validateName}">

<**f:ajax** event="blur" render="**nameError**" onevent="**showProgress**"/>

</h:inputText>

<h:graphicImage id="pole"

library="images" name="orange-barber-pole.gif"

**style="display: none"**/>

<h:message for="name" id="**nameError**"

value="#{user.nameError}" style="color: red"/>

</h:panelGroup>

Handling AJAX’s Errors:

<f:ajax onerror="handleAjaxError"/>

For errors, the data object also contains three properties not present for events:

• description

• errorName

• errorMessage

1. // Set session Object in Session
3. FacesContext context = FacesContext.getCurrentInstance();
4. HttpServletRequest request = (HttpServletRequest)context.getExternalContext().getRequest();
5. HttpSession httpSession = request.getSession(**false**);
6. httpSession.setAttribute(attName, reqdObj);
8. // Get session object from Session
10. FacesContext context = FacesContext.getCurrentInstance();
11. HttpServletRequest request = (HttpServletRequest)context.getExternalContext().getRequest();
12. HttpSession httpSession = request.getSession(**false**);
13. RequiredObject reqdObj = (RequiredObject) httpSession.getAttribute(attName);