

UI development with JavaServer Faces

Skill Level: Introductory

[Jackwind Li Guojie](#)

Author

02 Sep 2003

This tutorial provides an overview of JavaServer Faces (JSF) and walks you through the basics for developing Web applications using the technology. With Java developer and consultant Jackwind Li Guojie as your guide, you will examine the JSF life cycle, input validation, event handling, page navigation, and internationalization -- all of which are illustrated through a sample application.

Section 1. Before you start

About this tutorial

This tutorial provides an overview of JavaServer Faces (JSF) and presents the basics for developing Web applications using the technology. JSF is a standard UI framework and is designed to ease the burden of developing applications that run on a Java application server and render UIs back to a target client. Development of the technology is being led by Sun Microsystems as JSR 127 under the Java Community Process.

As you work through the tutorial you will examine the JSF life cycle, input validation, event handling, page navigation, and internationalization -- all of which are illustrated through a sample application.

This tutorial is intended for Java server-side developers with a good understanding of the Java language and a working knowledge of JSP technology.

After completing this tutorial, you should have a solid understanding of JavaServer Faces and will be able to develop robust Web applications that use JSF.

Setting up JSF

In order to build, deploy, and run JavaServer Faces applications, you need the Java Web Services Developer Pack (Java WSDP) or Apache Tomcat (version 4.0 or above) with JSF taglibs.

To proceed with this tutorial, you should download and install [Java WSDP 1.2](#). This distribution bundles JavaServer Faces 1.0 Early Access Release (EA4), the Tomcat 5 development container, and many other components. You also need J2SE SDK version 1.3 or above.

Set your environment variables as follows:

- Set `JAVA_HOME` to your J2SE SDK installation directory.
- Set `JWSDP_HOME` to your Java WSDP 1.2 installation directory.
- Set `ANT_HOME` to `$JWSDP_HOME/apache-ant` (under Unix or Linux) or `%JWSDP_HOME%\apache-ant` (under Windows).
- Set `JSF_HOME` to `$JWSDP_HOME/jsf` (under Unix or Linux) or `%JWSDP_HOME%\jsf` (under Windows).

Checking your JSF installation

On a UNIX or Linux platform, run the script `$JWSDP_HOME/bin/startup.sh` to start Java WSDP. On a Windows platform, run `%JWSDP_HOME/bin/startup.bat` or select **Programs > Java Web Service Developer Pack 1.2 > Start Tomcat** from the **Start** menu to bring up the server.

After the server is running, point your browser to `http://localhost:8080/` or `http://127.0.0.1:8080`. You should see the Java WSDP home page. Now, try to execute some JSF examples by following the links under the heading "JSF Examples." If there are no errors, you have successfully set up JSF.

Section 2. JavaServer Faces overview

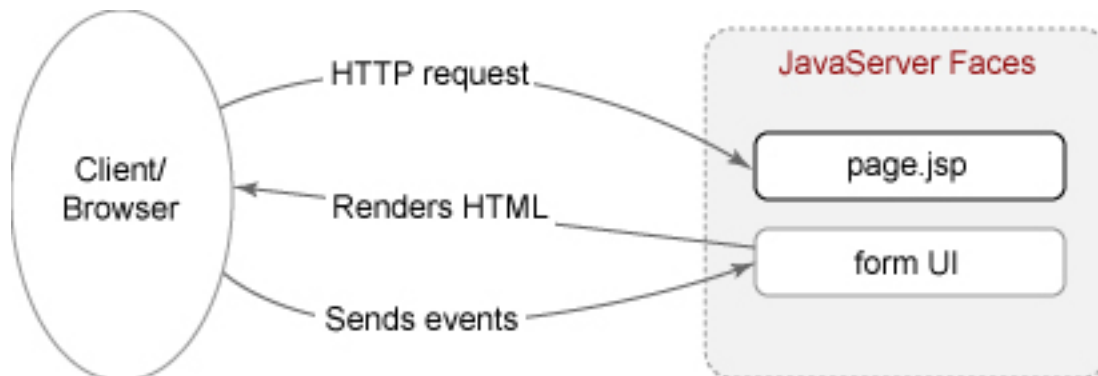
User interfaces for server-side applications

For years, we developers have been using servlets and JavaServer Pages (JSP) technology to build Web-based user interfaces. As our applications become more and more complex, we face many challenges. If we mix logic and presentation code, we'll find it extremely difficult to develop large applications consisting of hundreds or even thousands of Web pages. The construction of custom components is another big challenge: a simple table viewer requires a significant amount of time to develop and test. In addition, there is no easy way to port those HTML-based user interfaces onto other platforms, like handheld devices. Struts, a popular Web framework, and some other proprietary technologies solve some but not all of the problems.

A promising solution is JavaServer Faces (JSF) technology. Aiming to make Java Web application development easy, Sun Microsystems initiated JSF technology as JSR 127 in 2001.

JSF is a user interface framework for Web applications using Java technology. Designed to ease the burden of developing and maintaining applications that run on Java application servers and render their UIs back to a target client, JSF leverages existing, standard UI and Web-tier concepts without limiting developers to a particular markup language, protocol, or client device.

As illustrated in the following figure, a UI that has been created with JSF runs on the server side and renders back to the target client.



JSF components

JSF includes two main components:

- Java APIs for UI component management, UI state management, event handling, input validation, page navigation, internationalization, and accessibility
- A JSP custom tag library for expressing the JSF interface within a JSP page

JSF benefits

JSF simplifies UI development in several ways:

- Makes it easy and fast to create a UI from a set of predefined UI components.
- Automatically manages UI state.
- Facilitates object-oriented development. JSF is based on the Model-View-Controller (MVC) architecture, which offers a clean separation between logic and presentation.
- Enables developers to create custom UI components and reuse them.

JSF and Struts

Struts is an open source MVC framework based on servlet and JSP technology. Struts combines Java servlets, JSP technology, custom tags, and message resources into a unified open framework. Some may think that JSF and Struts are competing technologies, because both are UI frameworks based on MVC. However, an integration library will soon be available to connect JSF and Struts seamlessly. Using this library, you'll be able to use JSF components in your Web user interface along with Struts controller, actions, and business logic.

JSF has some advantages over Struts. For one thing, JSF provides several mechanisms to render an individual component; in Struts, however, there is only one way to do this. JSF also provides a clear separation of roles involved in Web application development. The author of Struts, Craig McClanahan, is one of the leads for developing JSF technology.

For more information on Struts, see [Resources](#).

Application development roles

JSF makes it possible to divide the application development process into distinct roles so that different people can contribute to different parts of the process. Even if the same person plays more than one of these roles, it is still useful to understand the individual perspectives separately. Here are some key roles:

- **Page authors:** A page author is someone who uses markup language to author pages for Web applications. Responsibilities include using markup languages, doing graphic design, assembling content of pages, and using

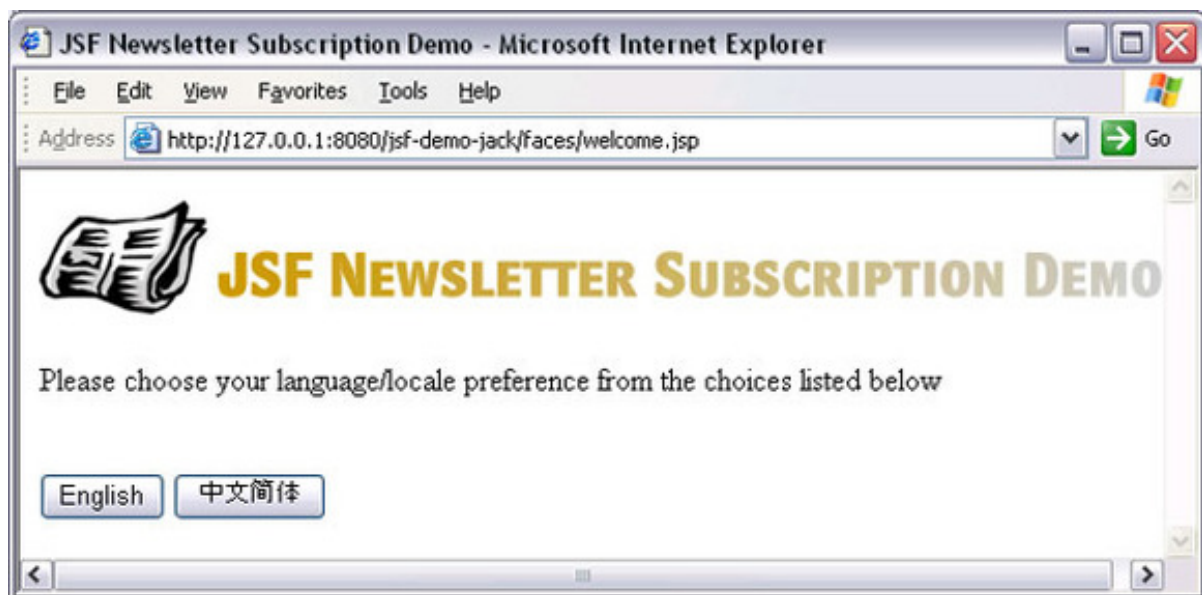
JSP tags. Files like `welcome.jsp` and `subscribe.jsp` in our sample application would be created by a page author.

- **Component writers:** A component writer is someone who creates reusable user interface components. In the sample application, there is a custom UI component for e-mail address representation, `UIOutputEmail`, which is used in the file `thanks.jsp`; this would be the work of a component writer.
- **Application developers:** An application developer provides the server-side functionality -- model objects, validators and event handlers, and so on. In an MVC scenario, the person filling this role provides both model and controller parts. In the sample application, the application developer would provide the two model objects (`SubscriberBean` and `SubscriptionBean`), the validator (`EmailAddressValidator`), and the event handler (`LocaleEventHandler`).

Section 3. Introducing the sample JSF application

JSF application overview

In this section, you'll learn the basic steps required to create a typical JavaServer Faces application. To illustrate JSF's capabilities, we will use this technology to build an e-mail newsletter subscription application.

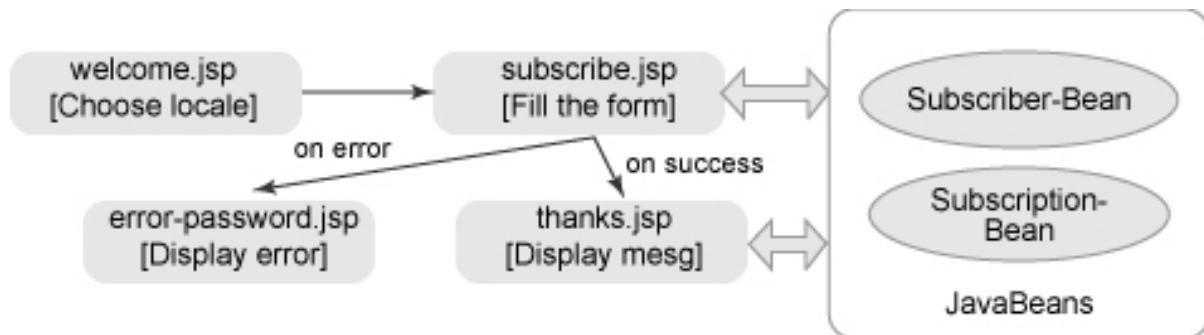


In the sections that follow, we'll use this application to illustrate various aspects of JavaServer Faces technology.

About the sample application

Our sample application allows visitors to subscribe to e-mail newsletters. The application is available in two languages, English and Simplified Chinese. The subscriber needs to provide a valid e-mail address, supply a password, and specify a newsletter delivery format to complete the subscription transaction.

The following figure shows the page flow of the newsletter subscription application.



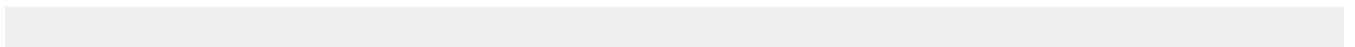
Installing the sample application

Follow these steps to install the sample application:

1. Download [/jsf-demo-jack.zip](#). Unzip the contents to the `JWSDP_HOME\jsf\samples\jsf-demo-jack` directory.
2. Copy the configuration file `JWSDP_HOME\jsf\samples\jsf-demo-jack\jsf-demo-jack.xml` to the `JWSDP_HOME\webapps` directory.
3. Restart the Tomcat server.
4. Test the installation by pointing your browser to `http://127.0.0.1:8080/jsf-demo-jack`.

Files and directories

The following tree shows the file structure of our sample application:



```

jsf-demo-jack/
|-- build.properties      # Apache Ant build configuration
|-- build.xml            # Apache Ant build file
|-- jsf-demo-jack.war    # The deployable war file
|-- jsf-demo-jack.xml    # Webapp descriptor
|-- src                  # Source code folder
|-- web
|   |-- WEB-INF
|   |   |-- classes      # Java class folder
|   |   |-- faces-config.xml # JFS config file
|   |   |-- jsf-demo-jack.tld # Custom taglib descriptor
|   |   |-- lib          # Library folder
|   |   |   |-- demo-components.jar
|   |   |-- web.xml      # Web app. descriptor
|   |-- img              # Resources,
|   |-- index.html       # JSP, HTML scripts ...
|   |-- error-password.jsp
|   |-- subscribe.jsp
|   |-- thanks.jsp
|   |-- welcome.jsp

```

All the source files are in the folder `src`. Most of the configuration files are under the directory `WEB-INF`.

The `web.xml` configuration file

Located at the top level of the `WEB-INF` directory, `web.xml` is used to configure the whole Web application. For JSF applications, we need to configure both the servlet used to process JSF requests and the servlet mapping for that servlet. The following listing is the complete configuration file used with our sample application:

```

<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app character-encoding="UTF8">
  <description>
    JSFNews: Newsletter Subscription Using JavaServer Faces
  </description>
  <display-name>
    JSFNews: Newsletter Subscription Using JavaServer Faces
  </display-name>

  <context-param>
    <param-name>saveStateInClient</param-name>
    <param-value>>false</param-value>
  </context-param>

  <!-- Faces Servlet Request Processor -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup> 1 </load-on-startup>
  </servlet>

  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>

```

```
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Using JSF tags with JSP pages

The following listing shows the contents of the welcome.jsp page in the sample application:

```
<%@ page contentType="text/html; charset=UTF8" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>

<f:use_faces>
<fmt:setBundle basename="net.jackwind.jsf.Resources"
    scope="session" var="myBundle"/>

<html>
<head>
    <title><h:output_text key="title" bundle="myBundle" /></title>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>

<body bgcolor="white">

<h:graphic_image url="img/title.gif" />
<p></p>

<h:output_text key="locale" bundle="myBundle" />
<p></p><br>

<h:form formName="imageMapForm" >

<h:command_button id="US" label="English" commandName="English"
    action="success">
    <f:action_listener type="net.jackwind.jsf.LocaleEventHandler" />
</h:command_button>

<h:command_button id="CN" label="####" commandName="Chinese"
    action="success">
    <f:action_listener type="net.jackwind.jsf.LocaleEventHandler" />
</h:command_button>

</h:form>

</f:use_faces>
</body>
</html>
```

Here are some important features that you should note from the code above:

- JSF tag libraries have been declared. These libraries expose most of the JSF components, validators, event registers, and so on.
- `form` tag represents an input form with which the user can submit data to the server.

- `command_button` represents a button that the user can click to submit data to the server. Some other standard components available are `command_hyperlink`, which represents an HTML hyperlink and is rendered as HTML; `graphic_image`, which displays an image; `input_text`, which accepts a text string of one line; and `output_text`, which displays a text string of one line. See Resources for a link to a complete list of built-in JSF components.
- `action_listener` represents an event listener. When the user clicks the button, data will be submitted to the server. The server then processes the data, and will call the corresponding event listeners.

Defining page navigation

In the last section, you saw our `welcome.jsp` page. After the user clicks one of the buttons, which page should be displayed next? JavaServer Faces applications use *navigation rules* to control page navigation. Each navigation rule specifies how to get from one page to the others within the application. In the MVC architecture, page navigation is one of the responsibilities of the controller. The navigation rules for JSF applications are contained in a file named `faces-config.xml` under the `WEB-INF` directory. The listing below contains the portion of that file that makes up the navigation rule for the `welcome.jsp` page:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<!-- ===== FULL CONFIGURATION FILE ===== -->

<faces-config>

  <!-- ===== Navigation rules ===== -->

  <navigation-rule> <!-- Navigation: From welcome.jsp - The home page. -->
    <from-tree-id>/welcome.jsp</from-tree-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-tree-id>/subscribe.jsp</to-tree-id>
    </navigation-case>
  </navigation-rule>
  ...
</faces-config>
```

When one of the buttons is clicked, the server calls its action listener, which always returns as successful in this case. JSF's `NavigationHandler` will use this outcome to match navigation cases and return the corresponding `from-tree-id`, which identifies the page to be displayed next.

We'll discuss this subject in more detail in [Page navigation](#) .

Model objects

The model objects in JSF applications are no different from the JavaBeans components you use in any other applications. In our sample application, we have two beans. The first, `SubscriptionBean`, represents a newsletter subscription; the second, `SubscriberBean`, represents a subscriber.

You can create JavaBeans components in two different ways. You can either create them in JSP pages directly, or you can have the JSF framework automatically handle bean management for you. In our sample application, we use the latter method. To do this, we use the following entries in the `faces-config.xml` file:

```
<!-- ===== Initialize Beans ===== -->
<managed-bean>
  <description>Subscription Bean</description>
  <managed-bean-name>subscriptionBean</managed-bean-name>
  <managed-bean-class>
    net.jackwind.jsf.SubscriptionBean
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>subject</property-name>
    <property-class>java.lang.String</property-class>
    <value>JSF Weekly</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <description>Subscriber Bean</description>
  <managed-bean-name>subscriberBean</managed-bean-name>
  <managed-bean-class>
    net.jackwind.jsf.SubscriberBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Notice that `managed-property` can be used to initialize bean properties.

Using JavaBeans components

In the last section, we saw how to create JavaBeans components and register them to the JSF framework. Now, we can easily use beans in any JSP files. As an example, take a look at the following JSP code, extracted from `subscribe.jsp`:

```
<f:use_faces>
...
<h:output_text id="title" valueRef="subscriptionBean.subject" />
...
<h:input_text id="email" valueRef="subscriberBean.email">
  <f:validate_required />
</h:input_text>
```

```
<jack:email_address_validator />
</h:input_text>
```

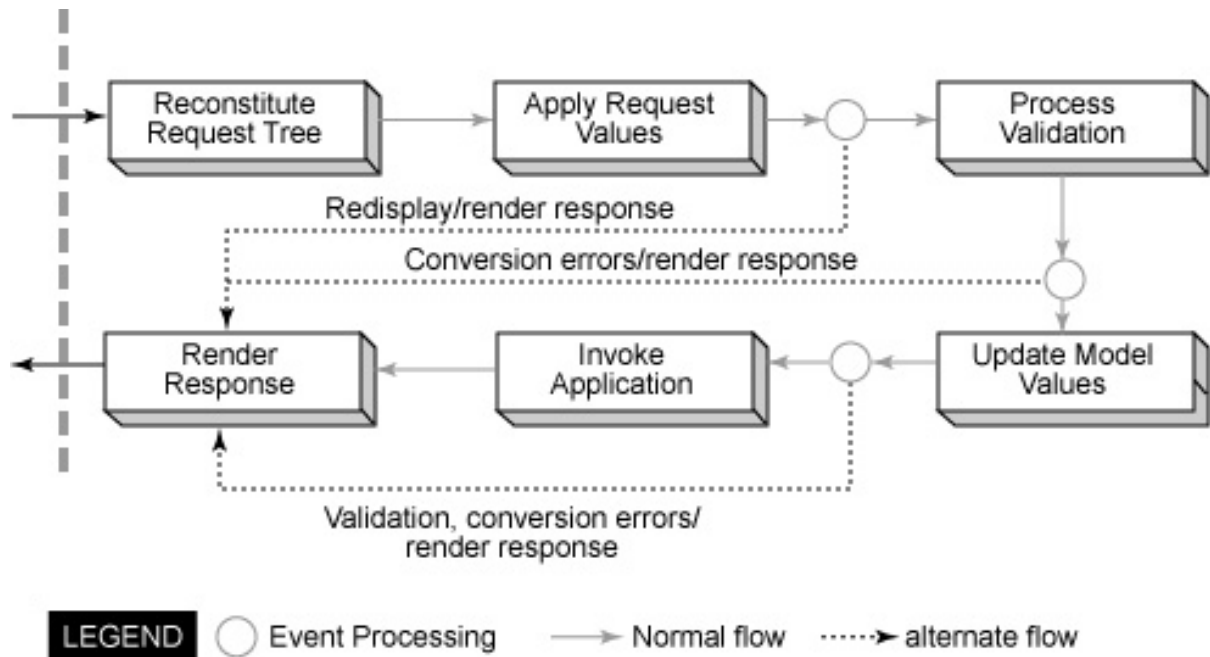
Many JSF components can interact with beans easily. In the above code, `input_text` represents a text field component. Its `valueRef` attribute uses a reference expression to refer to the model object property that holds the data. The first part of the reference expression is `subscriberBean`, which is the name of the `SubscriberBean` defined in `faces-config.xml`; the second part is a property of the bean. We'll discuss this in more detail in [Update Model Values phase](#).

Section 4. The life cycle of a JSF page

JSF page overview

The request processing life cycle of a JSF page is very similar to that of a JSP page. Because JavaServer Faces provides additional services, the life cycle of a JSF page contains some extra phases. In this section, we will use our sample application's `subscribe.jsp` page to illustrate this life cycle, handling each phase in a separate panel.

The figure below illustrates the standard request processing life cycle phases in JSF. Note the instances of event processing in the figure; we'll discuss event processing separately in [Event handling with JSF](#).



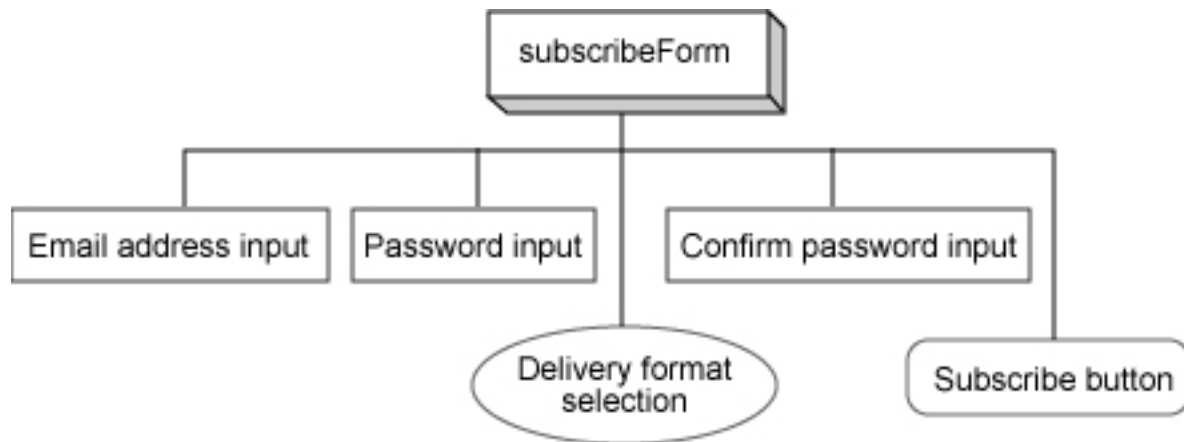
Reconstitute Request Tree phase

When the user clicks a link or a button (for example, when the user clicks the Subscribe button in our subscribe.jsp page), the JSF implementation begins the Reconstitute Request Tree phase.

During this phase, the JSF implementation will try to accomplish the following tasks:

- Building the component tree of the JSF pages
- Wiring up event handlers and validators for components
- Saving the tree in the `FacesContext`

The tree for subscribe.jsp looks like this:



Apply Request Values phase

In the next phase of the life cycle, called the Apply Request Values phase, each component updates its current value based on the information embedded in the current request (request/post parameters, headers, cookies, and so on). If the value extracted cannot be converted to a type that the component accepts, an error will be generated and queued on the `FacesContext`.

In our example, if the user enters his e-mail address into the e-mail input component and presses the Submit button, then the e-mail input component will be set to that value.

When events are queued during this phase, the JSF implementation broadcasts the events to corresponding listeners.

Process Validation phase

In the Process Validation phase, the JSF framework processes all validations that have been registered on the components in the tree. Validators will examine the attributes of each component and perform validations against them. If the value is invalid, an error message will be added to the `FacesContext` and the life cycle jumps to the final phase of the life cycle, Render Response, so that the same page is rendered again with the error messages displayed.

For instance, imagine that our user enters `jack#jackwind.net` into the e-mail text field and submits it. This data will not pass the e-mail address validation, and the same page will be displayed, with error messages. If the e-mail address provided is `jack@jackwind.net`, which is valid, then the life cycle will advance to next phase: Update Model Values.

We'll look at the validation process in more detail in [Input validation](#)

Update Model Values phase

By the time we reach the Update Model Values phase, we are confident that all the values associated with components in the tree are valid. During this phase, input components having `valueRef` expressions are updated. The following code shows the e-mail input component in `subscribe.jsp` page.

```
<h:input_text id="email" valueRef="subscriberBean.email">
```

The `email` property of the model -- that is, the `subscriberBean` -- will be updated using the local value of the e-mail input component.

Errors may occur if the local data cannot be converted to the types specified by the model object properties. If this happens, the life cycle will advance directly to the Render Response phase, and the page is rendered again with errors displayed. If events are queued during this phase, the JSF implementation broadcasts them to the corresponding listeners.

Invoke Application phase

The JavaServer Faces implementation handles application-level events in the Invoke Application phase.

In `subscribe.jsp`, there is one application-level event associated with the `command_button` component:

```
<h:command_button id="submit" key="submit" bundle="myBundle"  
  commandName="subscribe" actionRef="subscriberBean.submitSubscription" />
```

When processing this event, the framework will execute the action specified in the `actionRef` and retrieve the outcome. In the above example, `subscriberBean.submitSubscription` returns an `Action`, which is used to register the subscriber with the information obtained. The outcome `success` will be returned if the registration process has been completed successfully. The `NavigationHandler` will match this outcome to the proper navigation rule defined in the `faces-config.xml` to determine which page needs to be displayed next. In our example, the outcome will match the following rule:

```
<navigation-rule> <!-- Navigation: From subscribe.jsp -->  
  <from-tree-id>/subscribe.jsp</from-tree-id>  
  <navigation-case>  
    <from-outcome>success</from-outcome>  
    <to-tree-id>/thanks.jsp</to-tree-id>
```

```
</navigation-case>
<navigation-case>
  <from-outcome>passwords-not-match</from-outcome>
  <to-tree-id>/error-password.jsp</to-tree-id>
</navigation-case>
</navigation-rule>
```

Obviously, the page to be displayed next is `thanks.jsp`, according to the first `navigation-case` in the navigation rule. The framework then transfers control to the next phase: Render Response.

Render Response phase

Render Response is the final phase in the JSF life cycle. During this phase, the framework renders the component tree saved in the `FacesContext`. The output is usually in a markup language that the client understands; in the sample application, the output is in HTML format.

After the content of the tree is rendered, the tree will be saved so that subsequent requests can access it. This feature can boost the performance of JSF applications significantly.

Section 5. Input validation

Input validation overview

The JavaServer Faces framework provides some standard validation classes that you can use to validate a component's data. Additionally, JavaServer Faces allows you to create custom validators easily.

Validation is performed in the Process Validation phase of the JSF life cycle (see [Process Validation phase](#)), before the model object is updated.

The standard validators

JSF provides the following standard validators:

Validator class	Validator tag	Validation
<code>DoubleRangeValidator</code>	<code>validate_doublerrange</code>	Verifies that the local value of the component is within the

		range specified. The value must be of type <code>floating-point</code> or convertible to <code>floating-point</code> .
<code>LengthValidator</code>	<code>validate_length</code>	Verifies that the length of a component's local value is within the range specified.
<code>LongRangeValidator</code>	<code>validate_longrange</code>	Verifies that the local value of a component is within the range specified. Note that the value must be of type <code>long</code> or convertible to <code>long</code> .
<code>RequiredValidator</code>	<code>validate_required</code>	Verifies that the local value of a component is not null.
<code>StringRangeValidator</code>	<code>validate_stringrange</code>	Verifies that the local value of a component is within the range specified. The value must be a <code>String</code> .

In `subscribe.jsp`, we use the following code to make sure that the password is between 6 and 16 characters long:

```
<h:input_secret id="password" valueRef="subscriberBean.password">
  <f:validate_required />
  <f:validate_length maximum="16" minimum="6" />
</h:input_secret>
```

Displaying validation error messages

If any validation error occurs during the Process Validation phase, the life cycle will advance to the Render Response phase, and the same page is rendered again with an error message displayed. The error messages resulting from both standard and custom validation failures can be output using the `output_errors` tag. Here is how we output the error messages for the password component:

```
<h:input_secret id="password" valueRef="subscriberBean.password">
  <f:validate_required />
  <f:validate_length maximum="16" minimum="6" />
</h:input_secret>
<h:output_errors for="password" />
```

In case of validation errors, the output looks like this:

Password Validation Error: Value is required.

Custom validators

If the standard validators do not meet your validation requirements, you can easily create a custom validator. There are four simple steps to implementing a custom validator:

1. Implement the `javax.faces.validator.Validator` interface.
2. Register the custom `Validator` class.
3. Register error messages for the validator.
4. Create custom tags.

We'll examine each of these steps in turn in the next few panels.

Implement the `javax.faces.validator.Validator` interface

In the sample application, we need to validate the e-mail address provided by the subscriber. Here is our e-mail address validator class:

```
/*
 * $Id: EmailAddressValidator.java,v 1.2 2003/06/20 16:48:49 jack Exp $
 */
package net.jackwind.jsf;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.FactoryFinder;
import javax.faces.application.Application;
import javax.faces.application.ApplicationFactory;
import javax.faces.application.Message;
import javax.faces.component.UIComponent;
import javax.faces.component.UIOutput;
import javax.faces.context.FacesContext;
import javax.faces.context.MessageResources;
import javax.faces.validator.Validator;

/**
 * @author JACK (Jun 16, 2003)
 * @class EmailValidator
 */
public class EmailAddressValidator implements Validator {

    /**
     * The message identifier of the Message to be created if
     * the validation fails.
     */
    public static final String EMAIL_ADDRESS_INVALID =
        "net.jackwind.jsf.EMAIL_ADDRESS_INVALID";
}
```

```

// Email address pattern. Used by regex.
public static String PATTERN_EMAIL =
"[a-zA-Z0-9][\\w\\.\\-]*@[a-zA-Z0-9][\\w\\.\\-]*\\.([a-zA-Z][a-zA-Z\\.]*)";
private Pattern pattern = Pattern.compile(PATTERN_EMAIL);

public void validate(FacesContext context, UIComponent component)
{
    if ((context == null) || (component == null)) {
        throw new NullPointerException();
    }
    if (!(component instanceof UIOutput)) {
        return;
    }

    Object componentValue = ((UIOutput) component).getValue();
    String value =
componentValue == null ? null : componentValue.toString();
    if(value == null)
        return;

    if (validateEmailAddress(value)) {
        component.setValid(true);
    } else {
        component.setValid(false);
        Message errMsg =
            getMessageResources().getMessage(context,
            EMAIL_ADDRESS_INVALID,
            (new Object[] {"Sample: name@company.com"}));
        context.addMessage(component, errMsg);
    }
}

private boolean validateEmailAddress(String emailAddress) {
    if(emailAddress == null) return false;
    Matcher matcher = pattern.matcher(emailAddress);
    return matcher.matches();
}

/**
 * This method will be called before calling
 * facesContext.addMessage, so message can be localized.
 * <p>Return the {@link MessageResources} instance for the message
 * resources defined by the JavaServer Faces Specification.
 */
public synchronized MessageResources getMessageResources() {
    MessageResources emailResources = null;
    ApplicationFactory aFactory =
        (ApplicationFactory)FactoryFinder.getFactory
        (FactoryFinder.APPLICATION_FACTORY);
    Application application = aFactory.getApplication();

    emailResources =
application.getMessageResources("net.jackwind.jsf.EMAIL_ADDRESS_INVALID");
    return (emailResources);
}
}

```

The Validator interface declares only one function to be implemented: `public void validate(FacesContext context, UIComponent component)`. If the e-mail address is correct, `component.setValid(true)` will be called; otherwise, `component.setValid(false)` will be invoked and error messages will be added to the FacesContext.

Registering the custom validator

The custom validator must be available at application startup. The custom validator is registered with the code below, found in the faces-config.xml file:

```
<validator>
  <description>An email address validator</description>
  <validator-id>EmailValidator</validator-id>
  <validator-class>
    net.jackwind.jsf.EmailAddressValidator
  </validator-class>
</validator>
```

At a minimum, you need to supply the `validator-id` and `validator-class` sub-elements. `validator-id` represents a unique identifier with which the `Validator` class should be registered. `validator-class` is your custom `Validator` class. If you wish, you can put a simple description message in the optional `description` sub-element.

Registering error messages

Custom error messages must be available at application startup time. Error messages can be registered with the application configuration, the faces-config.xml file. The code below shows the custom messages for our e-mail address validator:

```
<message-resources>
  <message-resources-id>
    net.jackwind.jsf.EMAIL_ADDRESS_INVALID
  </message-resources-id>
  <message>
    <message-id>
      net.jackwind.jsf.EMAIL_ADDRESS_INVALID
    </message-id>
    <summary xml:lang="en"> Invalid email address
    </summary>
    <summary xml:lang="zh"> #####
    </summary>
  </message>
</message-resources>
```

Notice that we can specify localized message information. We'll discuss localization in more detailed in [Internationalization with JSF](#) .

Creating a custom tag

The tag handler for a custom validator must extend the `javax.faces.webapp.ValidatorTag` class. Here is our tag handler for the

e-mail address validator:

```

/*
 * $Id: EmailAddressValidatorTag.java,v 1.1 2003/06/20 16:48:49 jack Exp $
 */
package net.jackwind.jsf;

import javax.faces.validator.Validator;
import javax.faces.webapp.ValidatorTag;
import javax.servlet.jsp.JspException;

/**
 * @author      JACK (Jun 16, 2003)
 * @class      EmailAddressValidatorTag
 */
public class EmailAddressValidatorTag extends ValidatorTag {

    public EmailAddressValidatorTag() {
        super();
        setId("EmailAddressValidator");
    }

    protected Validator createValidator() throws JspException {
        return new EmailAddressValidator();
    }
}

```

We can also write a tag library description for the tag handler above. The listing below illustrates that description for our sample application; the file itself is located at WEB-INF/email-tag.tld

```

<?xml version="1.0" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>JSF Email Component by Jack Li Guojie.</short-name>
  <display-name>JSF Email Component by Jack Li Guojie.</display-name>
  <description>
    This library contains custom email component for JSF
  </description>

  <tag>
    <name>output_email</name>
    <tag-class>net.jackwind.jsf.UIOutputEmailTag</tag-class>
    <body-content>JSP</body-content>
    <description>
      JSF Email Component by Jack Li Guojie.
    </description>
    <attribute>
      <name>id</name>
      <required>>false</required>
      <rtexprvalue>>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>value</name>
      <required>>false</required>
      <rtexprvalue>>false</rtexprvalue>
    </attribute>
  </tag>

```

```
<attribute>
  <name>valueRef</name>
  <required>>false</required>
  <rtexprvalue>>false</rtexprvalue>
</attribute>
</tag>
</taglib>
```

Using the custom validator

To use the custom validator, we need to declare the custom tag library. The e-mail validator is needed in `subscribe.jsp`. We declare:

```
<%@ taglib uri="/WEB-INF/jsf-demo-jack.tld" prefix="jack" %>
```

Now the e-mail validator is available. We can use it with the e-mail input component:

```
<h:input_text id="email" valueRef="subscriberBean.email">
  <f:validate_required />
  <jack:email_address_validator />
</h:input_text>
<h:output_errors for="email"/>
```

If the e-mail address is invalid, an error message will be displayed, as shown below:

Email Address Invalid email address

Section 6. Event handling with JSF

Introduction

JSF aims to leverage existing models and paradigms so that developers can quickly adopt the technology in their applications. Like the JavaBeans component model, JSF technology defines listener and event classes that applications can use to handle events generated by UI components. An `Event` object stores the source of an event and other information. In order to be notified, a listener needs to register with the component that generates the event. An event will be fired when the user activates a component -- by clicking a button, for example. All listeners listening for the event will then be invoked by the JSF framework.

There are two different kinds of events in JSF: *value-changed* events and *action*

events. A value-changed event will be generated when the local value of a component changes. An action event occurs when a button or a hyperlink is clicked.

To set your application to react to an event, follow these two steps:

1. Implement an event listener (either an action listener or an value-changed listener).
2. Register the event handler on the component.

We'll illustrate a listener used by our sample application in the next panel.

Creating an action listener

To create an event listener, we need to implement `javax.faces.event.ActionListener`. Here is the action listener our sample application uses to handle a locale change event:

```
/*
 * $Id: LocaleEventHandler.java,v 1.2 2003/06/17 01:14:03 jack Exp $
 */
package net.jackwind.jsf;

import java.util.Locale;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;
import javax.faces.event.PhaseId;

/**
 * @author JACK (Jun 13, 2003)
 * @class LocaleEventHandler
 */
public class LocaleEventHandler implements ActionListener {

    public void processAction(ActionEvent event)
        throws AbortProcessingException {

        Locale locale = null;

        String command = event.getActionCommand();
        LogUtil.log("LocaleEventHandler invoked.
            [actionCommand=" + command);

        if(command.equalsIgnoreCase("ENGLISH"))
            locale = Locale.ENGLISH;
        else if(command.equalsIgnoreCase("CHINESE"))
            locale = Locale.CHINESE;
        else
            locale = Locale.ENGLISH;

        FacesContext context = FacesContext.getCurrentInstance();
        context.setLocale(locale);
    }
}
```

```
        System.out.println("Set locale to: " + locale);
    }
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}
```

When the user clicks a button, an action event is fired. The function `public void processAction(ActionEvent event)` will be called. This function will try to set the locale that the user has selected. Because more than one button can generate a locale change event, `event.getActionCommand()` is used to determine which button has been clicked. The value returned by `event.getActionCommand()` is the value of the `commandName` element within the command button component. The other function, `getPhaseId()`, is used for timing purposes; for details, see [Event processing timing](#).

Creating a value-changed listener

The process of implementing a value-changed listener is almost identical to that for implementing an action listener. The only difference is that a value-changed listener implements `javax.faces.event.ValueChangeListener` instead of the `ActionListener` interface.

Event processing timing

The code shown in [Creating an action listener](#) includes the function `public PhaseId getPhaseId()`. This function indicates the phase during which the listener should be invoked for event processing. For our locale change listener, the timing is not important, so `getPhaseId()` returns `PhaseId.ANY_PHASE`. If the listener implementation returns a `PhaseId` of `PhaseId.ANY_PHASE`, then the listener will be invoked for event processing during the Apply Request Values phase, if possible.

Sometimes, proper timing is crucial for event processing. A value-changed event should always return a `PhaseId` of `PhaseId.PROCESS_VALIDATIONS` so that input values pass the validation process before the model objects are updated.

Registering event listeners on components

The page author can register event listeners on components using `valuechanged_listener` and `action_listener` tags. The code below, extracted from `welcome.jsp`, registers an action listener to a command button:

```
<h:command_button id="US" label="English"
  commandName="English" action="success">
  <f:action_listener type="net.jackwind.jsf.LocaleEventHandler" />
</h:command_button>
```

Notice that we specify the event listener class using the `type` attribute.

Section 7. Page navigation

Page navigation overview

Every Web application comprises a set of Web pages. Managing the page navigation is one of the primary tasks of the Web developer.

JSF simplifies page navigation management by using a navigation rule configuration model. With JavaServer Faces, you do not need to write Java code to define navigation. Instead, you define it using navigation rules in the application configuration file, `faces-config.xml`.

The JSF page navigation mechanism

To navigate to another page within a Web application, the user generally clicks a button or a hyperlink. As you saw in the previous section, in JSF a button click fires an action event. The JavaServer Faces implementation provides a default action listener to determine the outcome of the action. There are two ways to define the outcome. The first is to use a string property, like this:

```
<h:command_button id="US" label="English"
  commandName="English" action="success">
```

The outcome is the value of the `action` attribute within the component.

You can use also an `Action` object. When extra processing is needed, an `Action` object can be used to perform desired actions. Here's an example:

```
<h:command_button id="submit" key="submit" bundle="myBundle"
  commandName="subscribe"
  actionRef="subscriberBean.submitSubscription" />
```

After the user clicks the button, an `Action` object referenced by the `actionRef` attribute will be obtained and executed. The outcome will be returned by the `Action` object's `invoke()` method.

After the outcome has been determined, the default action listener then passes this outcome to the `NavigationHandler` instance. Based on the outcome, the `NavigationHandler` will select the appropriate page by matching rules inside the application configuration file.

Setting navigation rules

Navigation rules guide `NavigationHandler` to select the correct page. In our sample application, the following rules are set in the application configuration file:

```
<!-- ===== Navigation rules ===== -->

<navigation-rule> <!-- Navigation: From welcome.jsp - The home page. -->
  <from-tree-id>/welcome.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/subscribe.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>

<navigation-rule> <!-- Navigation: From subscribe.jsp -->
  <from-tree-id>/subscribe.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/thanks.jsp</to-tree-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>passwords-not-match</from-outcome>
    <to-tree-id>/error-password.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>

<navigation-rule> <!-- Navigation: From error-password.jsp -->
  <from-tree-id>/error-password.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>back</from-outcome>
    <to-tree-id>/subscribe.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
```

Each rule defines possible ways to navigate from a certain page within the Web application. For example, the second rule says that the application will navigate to page `thanks.jsp` if the outcome is `success`, and it will navigate to `error-password.jsp` if the outcome is `passwords-not-match`. The `navigation-case`s will be checked against the outcome. When any of these `navigation-case`s are matched, the component defined in `tree-id` is selected.

Associating actions with components

In [The JSF page navigation mechanism](#), you saw two methods to define an outcome. The first method is very straightforward. Let's have a look at how the second method -- using `Action` objects -- works. The listing below is the subscribe button component in `subscribe.jsp`:

```
<h:command_button id="submit" key="submit" bundle="myBundle"
  commandName="subscribe"
  actionRef="subscriberBean.submitSubscription" />
```

After the button is clicked, the default action handler will use the reference provided in the `actionRef` attribute. In this case, the listener will call `subscriberBean`'s `getSubmitSubscription()` method for an `Action` object. The following is the skeleton of the method:

```
public Action getSubmitSubscription() {
    return new Action ...
}
```

Creating Actions

An `Action` is an object that performs a task and returns an outcome in `String` format that describes the result of the task processing. Most of an application's business logic-related code should be put in `Action`s. All `Action`s will be invoked during the `Invoke Application` phase in the JSF life cycle.

An `Action` class must extend the `javax.faces.application.Action` class. There is only one method to be overridden: `String invoke()`. The following code is the complete listing of the `getSubmitSubscription()` method, which is an `Action` class from our sample application:

```
public Action getSubmitSubscription() {
    return new Action(){
        public String invoke() {
            if(!( password != null &&
                confirmPassword != null &&
                password.equals(confirmPassword)))
                return "passwords-not-match";

            // TODO: Store the subscription information
            // and Check the status.
            return "success";
        }
    }
}
```

Notice that we use an anonymous class to create an `Action`. The `invoke()`

method has been overridden. This method checks to see if the password and confirm password fields match, stores information about the subscriber, and, finally, returns the outcome.

Section 8. Internationalization with JSF

Internationalization with JSF overview

Internationalization is the process of implementing applications that support multiple locales. *Localization* is the process of adapting an internationalized application to support a specific locale. JavaServer Faces relies on the JSP standard tag library to fully support internationalization and localization.

Our sample application has been localized for both English and Simplified Chinese. The first page, `welcome.jsp`, allows you to select the locale you prefer. The two different localized versions of the application are illustrated below.



For a comprehensive introduction to internationalization in Java programming, see [Resources](#).

Creating JSF pages with internationalization in mind

The code below is part of subscribe.jsp:

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

<fmt:setBundle basename="net.jackwind.jsf.Resources"
  scope="session" var="myBundle"/>

<html>
<head>
<title>
<h:output_text key="title" bundle="myBundle" />
</title>
</head>
```

```
<body bgcolor="white">

<f:use_faces>
<h:graphic_image url="img/title.gif" />

<p></p>

<h3>
<h:output_text key="prompt" bundle="myBundle" />
<font color=green>
<h:output_text id="title" valueRef="subscriptionBean.subject"/>
</font>
<h:output_text key="newsletter" bundle="myBundle" />
</h3>
```

There are a few internationalization-related aspects of this code that you should notice:

- Besides the tag libraries for `html_basic` and `jsf-core`, the JSTL `fmt` tag library has been declared and imported. (For more on JSTL, see [Resources](#).)
- We use the `fmt:setBundle` tag to set the resource bundle `net.jackwind.jsf.Resources`.
- Most visible components support localization by using `key` and `bundle` attributes.

Creating resource bundles

There are two ways to create resource bundles. You can create one either as a properties file or as a Java class extending `java.util.ListResourceBundle` that defines one abstract method: `getContents()`. In our sample application, we use Java classes as resource bundles. Here's the Java class for our English resource bundle:

```
/*
 * $Id: Resources.java,v 1.1 2003/06/17 01:14:03 jack Exp $
 */
package net.jackwind.jsf;

import java.util.ListResourceBundle;

/**
 * @author JACK (Jun 15, 2003)
 * @class Resources
 */
public class Resources extends ListResourceBundle {

    static final Object[][] contents = new String[][] {
        { "locale", "Please choose your language/locale preference
          from the choices listed below" },
        { "title", "JSF Newsletter Subscription Demo " },
        { "prompt", "You are going to subscribe" },
    };
}
```

```
        { "newsletter", "newsletter" },
        { "email", "E-mail address" },
        { "password", "Password" },
        { "confirmPassword", "Confirm Password" },
        { "format", "E-mail Delivery Format" },
        { "donotuse", "Please do not send me information about
          other offerings." },
        { "submit", "Subscribe" },
        { "url-welcome", "Home" },
        { "url-subscribe", "Subscribe" },
        { "passwords-not-match", "Password and confirm password
          do NOT match. " },
        { "go-back", "Go back" },
        { "thank-you", "Thank you! Your subscription request
          has been submitted. " },
        { "details", "Your subscription details" }

};

public Object[][] getContents() {
    return contents;
}
}
```

A separate Java class, `net.jackwind.jsf.Resources_zh`, serves as the resource bundle for the Chinese language.

Setting locales

With JSTL, we can set the locales using the following code in a JSP page:

```
<fmt:setLocale value='en' scope='session' />
```

In our sample application, however, we use another method to accomplish this. First, the `FacesContext` object is obtained; then the locale for `FacesContext` is set to the one that the user has selected. When the user clicks one of the buttons in the `welcome.jsp` page, an event is fired, which `net.jackwind.jsf.LocaleEventHandler` processes. The following code demonstrates how the locale of a `FacesContext` object can be set:

```
public void processAction(ActionEvent event)
    throws AbortProcessingException {

    Locale locale = null;

    String command = event.getActionCommand();

    if(command.equalsIgnoreCase("ENGLISH"))
        locale = Locale.ENGLISH;
    else if(command.equalsIgnoreCase("CHINESE"))
        locale = Locale.CHINESE;
    else
        locale = Locale.ENGLISH;

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

```
        context.setLocale(locale);
        System.out.println("Set locale to: " + locale);
    }
```

Section 9. Creating custom components

Creating custom components overview

JavaServer Faces technology offers a rich set of standard, reusable UI components that we can use to construct UIs for Web applications easily. Additionally, the flexible architecture of JSF allows us to create custom components.

In our sample application, we created a custom component to display e-mail addresses. The e-mail address output component enables the OS to invoke an e-mail client automatically when the user clicks the e-mail address. Also, this component will hide the e-mail address from Web bots, putting a halt to spam in the early stages.

E-mail address jack@jackwind.net

Creating the component tag handler

Creating a component tag is very similar to creating a custom JSP tag. `javax.faces.webapp.UIComponentTag` is the base class for all JSF component tag handlers. Here is the tag handler class for our e-mail address output component:

```
/*
 * $Id: UIOutputEmailTag.java,v 1.1 2003/06/20 16:48:49 jack Exp $
 */
package net.jackwind.jsf;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentTag;

/**
 * @author      JACK (Jun 19, 2003)
 * @class      UIOutputEmail
 */
public class UIOutputEmailTag extends UIComponentTag {

    private String valueRef;
    private String value;

    protected String getComponentType() {
```

```
        return "UIOutputEmail";
    }

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

    public String getValue() {
        return value;
    }

    public String getValueRef() {
        return valueRef;
    }

    public void setValueRef(String newValueRef) {
        valueRef = newValueRef;
    }

    public String getRendererType() {
        return null;
    }

    protected void overrideProperties(UIComponent component) {
        super.overrideProperties(component);
        UIOutputEmail email = (UIOutputEmail) component;

        if (value != null) {
            email.setValue(value);
        }

        if(email.getValueRef() == null && valueRef != null) {
            email.setValueRef(valueRef);
        }
    }
}
```

There are two abstract methods in `UIComponentTag`: `getComponentType()` and `getRendererType()`. The `getComponentType()` method returns the component type for the component that is bound to the tag. The other method, `getRendererType()`, selects the `Renderer` to be used for rendering the component. In the code above, `getRendererType()` returns `null`, which means that the component manages rendering itself without using external renderers. `overrideProperties()` is used to override properties of the component if the corresponding properties of this tag handler were explicitly set, and the corresponding attribute of the component has not been set. The rest are typical JavaBeans methods that can be used to get and set element values.

Creating the tag library descriptor

The tag library descriptor (TLD) associated with the `output_email` tag is illustrated below:

```
<?xml version="1.0" ?>
<!DOCTYPE taglib
```

```

PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>JSF Email Component by Jack Li Guojie.</short-name>
  <display-name>JSF Email Component by Jack Li Guojie.</display-name>
  <description>
    This library contains custom email component for JSF
  </description>

  <tag>
    <name>output_email</name>
    <tag-class>net.jackwind.jsf.UIOutputEmailTag</tag-class>
    <body-content>JSP</body-content>
    <description>JSF Email Component by Jack Li Guojie.
  </description>
    <attribute>
      <name>id</name>
      <required>>false</required>
      <rtexprvalue>>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>value</name>
      <required>>false</required>
      <rtexprvalue>>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>valueRef</name>
      <required>>false</required>
      <rtexprvalue>>false</rtexprvalue>
    </attribute>
  </tag>
</taglib>

```

This TLD is defined in the file `WEB-INF/email-tag.tld`. There are three attributes: `id` is used to set the ID for this component; the other two attributes are used to set the value -- the e-mail address. If the e-mail address is stored in a bean, then `valueRef` should point to the property of the bean. The `value` attribute can be set to a static e-mail address directly, like so:

```
<jack:output_email id="outemail" value="jack@jackwind.net" />
```

Building the custom component

A component class defines the state and behavior of a UI component. The state information includes the component's type, identifier, and local value. Some examples of behavior defined by component class are decoding (converting the request parameter and other information to the component's local value), encoding (converting a local value to some markup), or updating model objects.

We create the e-mail address output component by extending the `javax.faces.component.UIOutput` class.

```

/*
 * $Id: UIOutputEmail.java,v 1.1 2003/06/20 16:48:49 jack Exp $
 */
package net.jackwind.jsf;

import java.io.IOException;
import java.util.StringTokenizer;

import javax.faces.component.UIOutput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

/**
 * @author      JACK (Jun 19, 2003)
 * @class      UIOutputEmail
 */
public class UIOutputEmail extends UIOutput {
    private String host;
    private String user;

    private void parseEmail(String email) {
        StringTokenizer st = new StringTokenizer(email, "@");
        if (st.countTokens() != 2)
            throw new IllegalArgumentException(
                "Invalid email address: " + email);
        user = st.nextToken();
        host = st.nextToken();
    }

    // This method indicates whether this component renders itself
    // or delegates rendering to a renderer.
    public boolean getRendersSelf() {
        return true;
    }

    // Called during the Render Response phase
    public void encodeEnd(FacesContext context) throws IOException {
        if(user == null || host == null) {
            try {
                parseEmail((String)currentValue(context));
            } catch (Exception e) {
                LogUtil.log(e);
            }
        }
        ResponseWriter writer = context.getResponseWriter();

        // Represent this component as HTML with JavaScripts
        writer.write("<script language='JavaScript'>\n");
        writer.write("function emailAddress(host, user) { \n");
        writer.write("document.write('<a href='\"mailto:'>\n");
        writer.write("document.write(user + '@' + host);\n");
        writer.write("document.write('>');\n");
        writer.write("document.write(user + '@' + host);\n");
        writer.write("document.write('</a>');\n}\n\n");
        writer.write("emailAddress('" + host + "', '" +
            user + "');\n");
        writer.write("</script>\n");
    }

    protected String getComponentType() {
        return "output_email";
    }
}

```

Note that we override the `encodeEnd()` method to render the component into

HTML markup. This method will be called during the Render Response phase of the JSF life cycle.

Registering the component

To register the component, we need to put a component entry into the application configuration file. Here is the code to register the e-mail address output component:

```
<!-- ===== Custom components ===== -->
<component>
  <component-type>UIOutputEmail</component-type>
  <component-class>net.jackwind.jsf.UIOutputEmail</component-class>
</component>
```

Using the custom component

We've now created and registered our custom component. It can be used in the same way as standard components. The code below displays an e-mail address that is stored in the bean (see the file thanks.jsp):

```
<%@ taglib uri="/WEB-INF/jsf-demo-jack.tld" prefix="jack" %>
...
<jack:output_email id="outemail" valueRef="subscriberBean.email" />
```

And here's the HTML output from this component:

```
<script language='JavaScript'>
function emailAddress(host, user) {
document.write('<a href="mailto:');
document.write(user + '@' + host);
document.write('>');
document.write(user + '@' + host);
document.write("</a>");
}

emailAddress('jackwind.net', 'jack');
</script>
```

Section 10. Summary

Summary

This tutorial has provided an introduction to JavaServer Faces technology. You've learned about the JSF life cycle, input validation, event handling, page navigation, internationalization, custom components, and more.

The real-world, code-intensive sample application outlined here provides you a solid foundation to create your own Java Web application with JavaServer Faces technology. Hopefully, this new Java technology will help you spend less time worrying about the UI of your Web-based apps and more time adding value and functionality to the application logic.

Downloads

Description	Name	Size	Download method
Sample code	jsf-demo-jack.zip	200KB	HTTP

[Information about download methods](#)

Resources

Learn

- WebSphere Studio V5.1.1 has introduced JavaServer Faces runtime components along with easy-to-use tools for those components. The [IBM JavaServer Faces Component Catalog](#) showcases some of that tooling. The accompanying WAR file provides a fully interactive application that will let you browse the working components and see how they look and function at runtime.
- Visit the [JavaServer Faces technology home page](#).
- The [JavaServer Faces technology tutorial](#) from Sun provides an in-depth look at JSF.
- For authoritative references, visit [JSR 127](#) 's home page.
- Craig McClanahan explains [how JSF affects Struts](#), and which user interface technology you should choose.
- For a comprehensive introduction JSP technology, see Noel Bergman's tutorial "[Introduction to JavaServer Pages technology](#)" (*developerWorks*, August 2001).
- The book [Mastering JSP Custom Tags and Tag Libraries](#) by James Goodwill (John Wiley & Sons, 2002) is a good reference on JSP tags.
- You may also want to review the following articles for assistance with JSP custom tags:
 - "[Take control of your JSP pages with custom tags](#)" (*developerWorks*, January 2002).
 - "[JSP taglibs: Better usability by design](#)" (*developerWorks*, December 2001).
- JSTL expert Mark Kolb adeptly introduces the technology in his four-part series, "A JSTL primer" (*developerWorks*, February-May 2003):
 - "[Part 1: The expression language](#)"
 - "[Part 2: Getting down to the core](#)"
 - "[Part 3: Presentation is everything](#)"
 - "[Part 4: Accessing SQL and XML content](#)"
- Joe Sam Shirah offers a comprehensive look at internationalization in Java programming in his tutorial "[Java internationalization basics](#)" (*developerWorks*, April 2002).

- Don't miss this complete listing of free [Java programming tutorials](#) from *developerWorks*.
- You'll find hundreds of articles about every aspect of Java programming in the [developerWorks Java technology zone](#).

Get products and technologies

- You need to download [Java Web Services Developer Pack 1.2](#) to run JavaServer Faces applications.

About the author

Jackwind Li Guojie

Contributing developerWorks author.