## Article

# Developing Web Applications with JavaServer Faces



*By Qusay H. Mahmoud, August 2004*

Articles Index

JavaServer Faces (JSF) is a standardized specification for building User Interfaces (UI) for server-side applications. Before JavaServer Faces, developers who built web applications often relied on building HTML user interface components with servlets or JavaServer Pages (JSP pages). This is mainly because HTML user interface components are the lowest common denominator that web browsers support. The implication, of course, is that such web applications do not have rich user interfaces, compared with standalone fat clients, and therefore less functionality and/or poor usability. While applets can be used to develop rich user interfaces, web application developers don't always know what clients will be accessing the application and/or they may have no access to the client device.

In addition, if you have taken part in developing a large-scale web system, you may have come across technical challenges, such as how to implement custom components like a query builder or a table viewer for building database queries. Building such custom components requires expertise and a significant amount of time to build and test the new libraries. In an ideal environment, developers would be able to use pre-built, tested, and highly configurable components to integrate into their application development environment.

JavaServer Faces (JSF) is a server-side technology for developing web applications with rich user interfaces. With JSF, you can resolve such technical challenges as creating custom user interface components. This is because JSF technology is a user interface framework for building Java-based web applications that run on the server side, and render the user interface back to the client. That's right! The user interface code runs on the server, responding to events generated on the client.

Rich user interfaces consist of a set of rich components. There are a number of options out there today for building rich server-side user interfaces, such as Flash, Swinglets, and Jade. However, these solutions are proprietary; the tools and runtimes that support their development are usually only available through a single vendor. JSF is, above all, a standard, which means that the developer does not get locked into a single vendor. The specification expert group is actually made up of representatives from all the major tool vendors in the Java community. Thus, developers will have no shortage of choices for tools, and probably will be able to use an updated version of a tool they are using today. While the tool vendors are cooperating on the specification, they will compete with each other in their implementations. This will benefit the developer community in terms of features and in the choice of custom components each vendor makes available. Tool Vendors provide an Integrated Development Environment (IDE) to help you build and deploy your JSF applications. The following vendors offer tools to support JSF development.

- Sun Java Studio Creator
- Borland JBuilder
- IBM Websphere
- Oracle JDeveloper

Component Vendors provide reusable building blocks for application development and integration into JSF-based tools. Components cover a wide range of uses including reporting, charts and graphs, fields, and navigational components. Here are some of the vendors who offer components that support the JavaServer Faces specification.

- Otrix WebMenu
- ILOG JViews
- Quest Software JClass

- Software FX Chart FX

JavaServer Faces technology is based on the Model View Controller (MVC) architecture for separating logic from presentation, so if you have been practicing this, you'll feel at home with JSF.

This article provides a fast-track, code-intensive tutorial to get you started with JSF. The article also:

- Provides an introduction to JSF
- Describes the benefits of JSF
- Describes the internals of JSF
- Describes the genesis of JSF applications
- Gives you a taste of the effort involved in developing JSF applications
- Provides sample code that you can adapt for your own applications
- Discuss the JSF support in the Sun Java Studio Creator IDE

If you are looking for more advanced topics, please see Chapters 17 - 21 of the J2EE 1.4 Tutorial.

**Overview of JavaServer Faces**

JavaServer Faces (JSF) is a technology that is being led by Sun Microsystems as JSR 127 under the Java Community Process (JCP). The objective is to create a standard framework for user interface components for web applications. As mentioned earlier, JSF lets you build web applications that run on a Java server and render the user interface back to the client. This technology provides web application lifecycle management through a controller servlet, and a rich component model with event handling and component rendering.

To get started using JavaServer Faces, the first thing you should do is download a JavaServer Faces compatible implementation. All of the tool vendors listed above provide a JavaServer Faces compatible development and runtime environment. You can get a FREE JavaServer Faces Compatible runtime environment by downloading the J2EE 1.4 SDK, which includes support for JavaServer Faces. you can also get a copy of Java Studio Creator from Sun as part of your subscription to the Sun Developer Network.

Note that as a minimum requirement, JSF runs on Servlet 2.3 and JSP 1.2. Figure 1 depicts the high-level architecture of JSF.

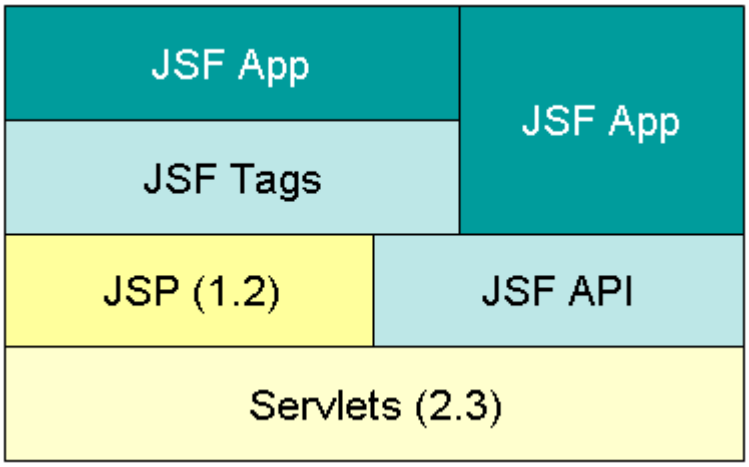


Figure 1: High-level architecture of JSF

The JSF technology consists of two main components:

1. Java APIs to represent UI components, manage state, handle events, and validate input. The API has support for internationalization and accessibility.
2. Two JSP custom tag libraries for expressing user interface (UI) components within a JSP page, and for wiring components to server-side objects. Page authors can easily add UI components to their pages.

Figure 2 shows the relationship between the client, the server, and JSF.

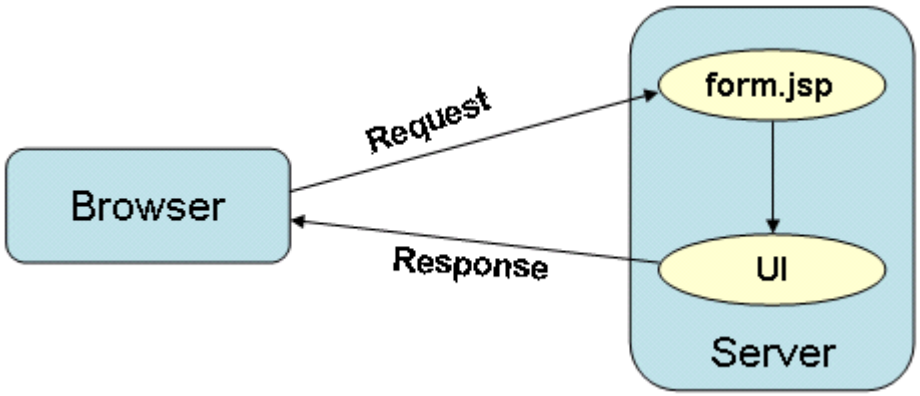


Figure 2: The UI runs on the server

Here, the JSP page represents the user interface components using the JSF custom tag library, rather than hard coding them with a markup language. The UI for the application manages the objects rendered by the JSP page.

Several types of users can benefit from this technology, including:

- *Page authors* who use markup languages, such as HTML, will use the JSP tag library for expressing JavaServer Faces rich user interface components.
- *Application developers* who write the model objects and event handlers.
- *Component developers* who will create custom components based on the JSF components.
- *Tools vendors* who will provide tools that incorporate JSF technology into a new generation of tools that simplify the development of multi tier, web-based applications.
- *Application Server vendors* who provide a runtime environment that incorporates JSF technology into a new generation of Application Servers that can deploy multi tier, web-based applications that use JSF technology.

This technology also opens up the market for reusable web user interface components. Developers and vendors can use JSF as the building blocks for developing custom faces.

One of the advantages of JSF is that it is based on the Model View Controller (MVC) architecture, to offer a clean separation between presentation and logic. This may ring a bell for those who are using existing web frameworks such as Struts. However, note that JSF and Struts are not competing technologies, and in fact, they interoperate together. JSF, however, does have some advantages over Struts. For example, in Struts there is only one way to render an element, while JSF provides several mechanisms for rendering an individual element. It is up to the page designer to pick the desired representation, and the application developer doesn't need to know which mechanism was used to render a component. (Here's a link for more information on the integration of JSF and Struts.) Readers may notice that the author of Struts, Craig McClanahan, is also the co-specification lead for JSF, as well as an employee of Sun Microsystems.

### Genesis of a JSF Application

A JSF application is just like any other Java technology-based web application; it runs in a Java servlet container, and contains:

1. JavaBeans components (or model objects) containing application-specific functionality and data
2. Event listeners
3. JSP pages
4. Server-side helper classes
5. A custom tag library for rendering UI components
6. A custom tag library for representing event handlers and validators
7. UI components represented as stateful objects on the server
8. Validators, event handlers, and navigation handlers. (Validators are used to validate data on individual components before the server-side data is updated.)
9. Application configuration resource file for configuring application resources

Here is a simple example using the tag libraries. For a list of component tags support and other information on JSF, please

refer to the JavaServer Faces specification and Chapter 17 - 21 of the J2EE 1.4 Tutorial.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<body bgcolor="white">
<f:view>
<h:form id="helloform">
<h2>What is your name?</h2>
     <h:inputText id="username" value="#{UserNameBean.userName}"
        validator="#{UserNameBean.validate}"/>
     <h:commandButton id="submit" action="success" value="Submit"/>
</h:form>
</f:view>
```

The JSF component architecture is designed in such a way that the component classes maintain a component's state. A renderkit defines how component classes map to component tags appropriate for a particular client. The JSF reference implementation, which is discussed later, includes a standard RenderKit for rendering to an HTML client. Each JSP component in the HTML RenderKit is composed of the component functionality defined by the `UIComponent` class, and the rendering attributes defined by the `Renderer`. For example, the tags `commandButton` and `commandHyperlink` both represent a `UIComponent`, but they are rendered in two different ways. The button component is rendered as a button and the hyperlink is rendered as a hyperlink.

When a JSP page is created using JSF components, a component tree or a `view` is built into memory on the server with each component tag corresponding to a `UIComponent` instance in the tree. The component tree is used by the JSF framework to handle your application request and create a rendered response. When an event is generated (for example, user clicks on a button), the JSF lifecycle handles the event and generates an appropriate response. Note that the entry point into the JSF framework is the `FacesServlet`. It acts as the front controller and handles request processing lifecycle.

**The JSF reference implementation 1.1** comes with the following:

- `html_basic.tld`: A JSP custom tag library for building JSF applications that render to an HTML client
- `jsf_core.tld`: A JSP custom tag library for representing core actions independent of a particular render kit
- APIs that provide user interface components, model object management, pluggable rendering, server-side validation, data conversion, event processing, page flow management, and custom extensibility for all of these features
- A set of demos located in the `samples` directory of the installation

The JSF Reference Implementation 1.1 can be downloaded here. When you unzip the archive, you'll get a directory structure like this (under Windows):

```
c:\jsf-1_1>
     \docs
     \javadocs
     \lib
     \metadata
     \renderkitdocs
     \samples
     \tlddocs
     some-other-files
```

The `example` directory contains WAR and source files for sample applications. The `lib` directory contains the JAR files that JSF depends on. The JAR files are:

- `commons-beanutils.jar`: Utilities for defining and accessing JavaBeans component properties
- `commons-collections.jar`: Extensions of the J2SE Collections Framework
- `commons-digester.jar`: For processing XML documents
- `commons-logging.jar`: A general purpose, flexible logging facility to allow developers to instrument their code with logging statements
- `jsf-api.jar`: Contains the `javax.faces.*` API classes
- `jsf-impl.jar`: Contains the implementation classes of the JSF Reference Implementation

Note that the JSF framework uses the JavaServer Pages Standard Tag Library (JSTL), and therefore it is assumed that the web container you use provides the necessary JAR files for JSTL.

The JSF RI can be easily set up to be used with virtually any web container. The RI comes with installations instructions on how to use it with Apache Tomcat, Java WSDP 1.3, and Sun Java System Application Server Platform Edition 8. The installation process is usually a matter of copying some JAR files, such as `jsf-api.jar` and `jsf-impl.jar` from the `lib` directory of the JSF RI to your web container's `lib` directory. For this article, however, the Sun Java Application Server Platform Edition 8 that comes with the Sun Java Studio Creator was used. The Sun Java Application Server Edition 8 can be downloaded from here and this already includes support for JSF.

If you are using the Sun Java Application Server Edition 8, you can easily get started by deploying the sample applications that come with the JSF RI by following these two simple steps:

1. Start the application server
2. Deploy the `.war` files from the `samples` directory of your JSF RI onto the application server. This can be done using the `deploytool` or by running `asant deploy`, but the easiest way is to copy the `.war` files to the `autodeploy` directory (on my machine, this directory is `C:\Sun-Studio-Creator\SunAppServer8\domains\creator\autodeploy`.

Now, you can start your favorite web browser and enter a URL to run a sample JSF application: `http://localhost:18080/`**`jsf-guessNumber`** or any other sample application that starts with `jsf-`. Here is a sample output:

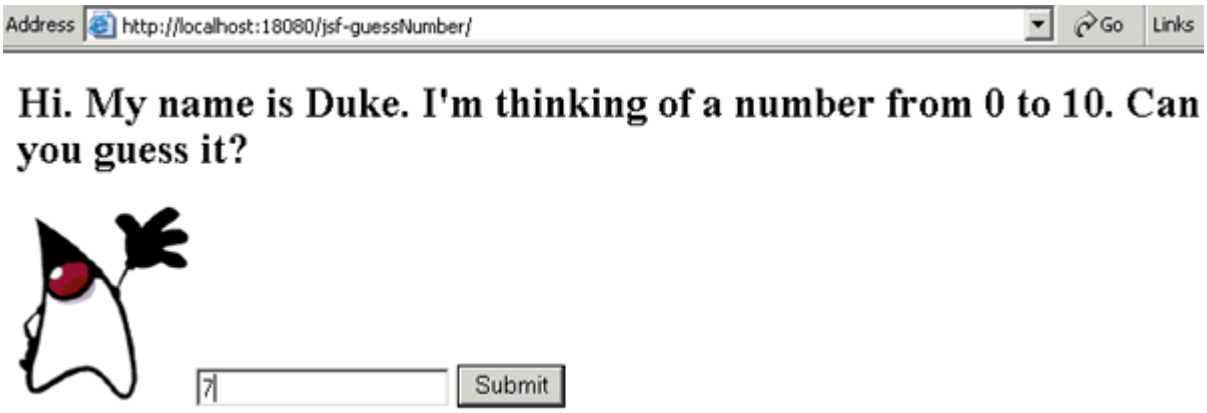


Figure 3: JSF GuessNumber Application in Action

**Understanding JavaServer Faces**

This section describes a sample application, a guessing number game, that comes with the JSF RI. The application, as shown in Figure 3 above, asks you to guess a number between 0 and 10, inclusive. If you enter the wrong number, you'll see something similar to Figure 4, and when you finally enter the right number, you'll see something similar to Figure 5. This application will give you a taste of the effort involved in developing web applications using the JSF technology.

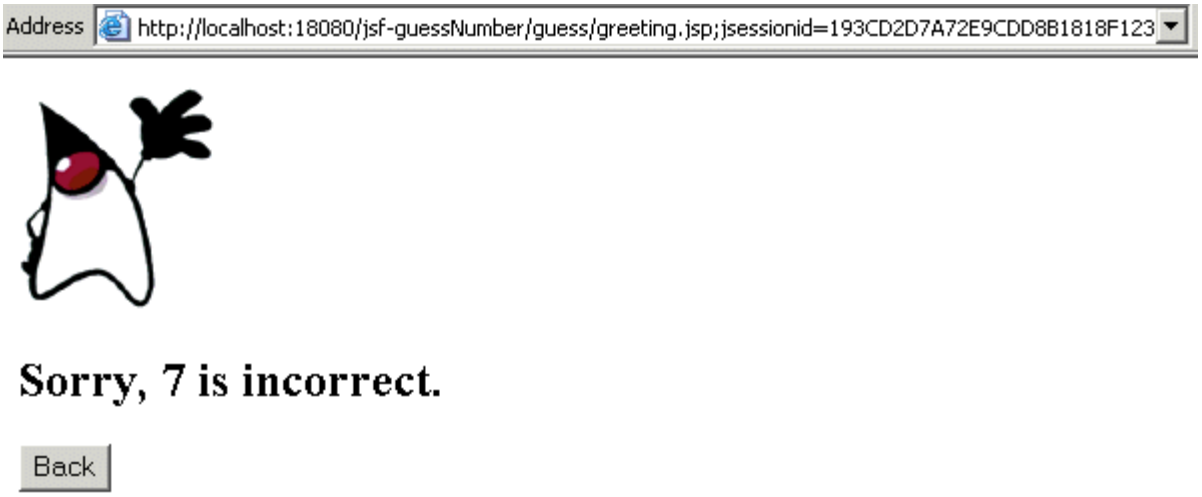


Figure 4: Wrong Answer


Figure 5: Correct Answer

A JSF application consists of the following files:

- JSP pages with JSF components representing the UI
- JavaBeans to hold the model data
- Application configuration files specifying the JSF controller servlet, managed beans, and navigation handles

### The JSP Pages

In the guess number sample application, three JSP pages are used: `index.jsp`, `greeting.jsp`, and `response.jsp`. Let's start with the `index.jsp` page.

**The index.jsp page**: This page is shown in Code Sample 1.

**Code Sample 1**: index.jsp

```
<html>
<head>
</head>
<body>
  <jsp:forward page="guess/greeting.jsp" />
```

```
</body>
</html>
```

As you can see, this page simply forwards the user to the main page, `greeting.jsp`, and therefore in your application you can do without it.

**The greeting.jsp page**: This is the main page presented to the user, which is shown in Figure 3 above. Code Sample 2 shows the content of this page.

**Code Sample 2**: greeting.jsp

```
<html>
<head><title>Hello</title></head>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<body bgcolor="white">
<f:view>
  <h:form id="helloForm" >
   <h2>Hi. My name is Duke.  I'm thinking of a number from
   <h:outputText value="#{UserNumberBean.minimum}"/> to
   <h:outputText value="#{UserNumberBean.maximum}"/>.
       Can you guess it?</h2>
    <h:graphicImage id="waveImg" url="/wave.med.gif" />
    <h:inputText id="userNo" value="#{UserNumberBean.userNumber}"
               validator="#{UserNumberBean.validate}"/>
    <h:commandButton id="submit" action="success"
             value="Submit" />
    <p>
    <h:message style="color: red; font-family: 'New Century Schoolbook',
              serif; font-style: oblique; text-decoration: overline"
                       id="errors1" for="userNo"/>
   </h:form>
</f:view>
</body>
</html>
```

The `greeting.jsp` page demonstrates several features that you will use in most of your JSF applications. These are:

- In order to use JSF tags, you need to include the `taglib` directives to the `html` and `core` tag libraries that refer to the standard HTML renderkit tag library, and the JSF core tag library, respectively.
- A page containing JSF tags is represented by a tree of components whose root is the `UIViewRoot`, which is represented by the `view` tag. All component tags must be enclosed in the `view` tag. Other content such as HTML and other JSP pages can be enclosed within that tag.
- A typical JSP page includes a form, which is submitted when a button is clicked. The tags representing the form components (such as textfields and buttons) must be nested inside the `form` tag.
- The `outputText` tag represents a label. The `greeting.jsp` page has two such tags that display the numbers 0 and 10. The `value` attribute of the tag gets the values from the `minimum` and `maximum` properties of a bean class, `UserNumberBean`, using value-binding expressions that have the syntax `#{bean-managed-property}`
- The `graphicImage` tag is used to reference an image.
- The `inputText` tag represents an input text field component. The `id` attribute represents the ID of the component object represented by this tag, and if it is missing, then the implementation will generate one. The `validator` attribute refers to a method-binding expression pointing to a backing bean method that performs validation on the component's data.
- The `commandButton` tag represents the button used to submit the data entered in the text field. The `action` attribute helps the navigation mechanism to decide which page to open next.
- The `message` tag displays an error message if the data entered is not valid. The `for` attribute refers to the component whose value failed validation.

**The response.jsp page**: This page, which is similar to Figure 4 and 4 above, is displayed as a response to the user's clicking the `Submit` button. Code Sample 3 shows the content of this page whose tags have already been explained.

**Code Sample 3**: response.jsp

```
<html>
<head> <title>Guess The Number</title></head>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<body bgcolor="white">
<f:view>
  <h:form id="responseForm" >
    <h:graphicImage id="waveImg" url="/wave.med.gif" />
    <h2><h:outputText id="result"
         value="#{UserNumberBean.response}"/></h2>
    <h:commandButton id="back" value="Back" action="success"/><p>
  </h:form>
</f:view>
</body>
</html>
```

### The Model Object

A typical JSF application uses a bean with each page in the application. The bean defines the properties and methods associated with the UI components used on the page. A bean can also define a set of methods that perform functions, such as validating the component's data, for the component. The model object bean is like any other JavaBeans component: it has a set of accessor methods. Code Sample 4 shows a sample JavaBean component, which is referenced in the `greeting.jsp` and `response.jsp` pages.

**Code Sample 4**: UserNumberBean.java

```
package guessNumber;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.LongRangeValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import java.util.Random;
public class UserNumberBean {
    Integer userNumber = null;
    Integer randomInt = null;
    String response = null;
    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(10));
        System.out.println("Duke's number: " + randomInt);
    }
    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
        System.out.println("Set userNumber " + userNumber);
    }
    public Integer getUserNumber() {
        System.out.println("get userNumber " + userNumber);
        return userNumber;
    }
    public String getResponse() {
        if (userNumber != null && userNumber.compareTo(randomInt) == 0) {
            return "Yay! You got it!";
        } else {
            return "Sorry, " + userNumber + " is incorrect.";
        }
    }
    protected String[] status = null;
    public String[] getStatus() {
        return status;
```

```
}
public void setStatus(String[] newStatus) {
    status = newStatus;
}
private int maximum = 0;
private boolean maximumSet = false;
public int getMaximum() {
    return (this.maximum);
}
public void setMaximum(int maximum) {
    this.maximum = maximum;
    this.maximumSet = true;
}
private int minimum = 0;
private boolean minimumSet = false;
public int getMinimum() {
    return (this.minimum);
}
public void setMinimum(int minimum) {
    this.minimum = minimum;
    this.minimumSet = true;
}
public void validate(FacesContext context,
                     UIComponent component,
                     Object value) throws ValidatorException {
    if ((context == null) || (component == null)) {
        throw new NullPointerException();
    }
    if (value != null) {
        try {
            int converted = intValue(value);
            if (maximumSet &&
                (converted > maximum)) {
                if (minimumSet) {
                    throw new ValidatorException(
                        MessageFactory.getMessage
                        (context, component,
                         Validator.NOT_IN_RANGE_MESSAGE_ID,
                         new Object[]{
                             new Integer(minimum),
                             new Integer(maximum)
                         }));
                } else {
                    throw new ValidatorException(
                        MessageFactory.getMessage
                        (context, component,
                         LongRangeValidator.MAXIMUM_MESSAGE_ID,
                         new Object[]{
                             new Integer(maximum)
                         }));
                }
            }
            if (minimumSet &&
                (converted < minimum)) {
                if (maximumset) {
                    throw new validatorexception(messagefactory.getmessage
                        (context, component,
                        validator.not_in_range_message_id,
                        new object[]{
                        new double(minimum),
                        new double(maximum)
                        }));
                } else {
                    throw new validatorexception(
```

```
                            messagefactory.getmessage
                            (context, component,
                             longrangevalidator.minimum_message_id,
                             new object[]{
                                 new integer(minimum)
                             }));
                        }
                    }
                } catch (numberformatexception e) {
                    throw new validatorexception(
                        messagefactory.getmessage
                        (context, component, longrangevalidator.type_message_id));
                }
            }
        }
        private int intvalue(object attributevalue)
            throws numberformatexception {
            if (attributevalue instanceof number) {
                return (((number) attributevalue).intvalue());
            } else {
                return (integer.parseint(attributevalue.tostring()));
            }
        }
    }
```

### Application Configuration Resource File

An application configuration resource file, `faces-config.xml`, is used to define your managed beans, validators, converters, and navigation rules. Code Sample 5 shows the `faces-config.xml` file for the guess number application that defines navigation rules and the mapping of managed beans.

**Code Sample 5**: faces-config.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JSF Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>fr</supported-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
  </application>
  <navigation-rule>
    <description>
        The decision rule used by the NavigationHandler to
        determine which view must be displayed after the
        current view, greeting.jsp is processed.
    </description>
    <from-view-id>/greeting.jsp</from-view-id>
    <navigation-case>
        <description>

            Indicates to the NavigationHandler that the response.jsp
            view must be displayed if the Action referenced by a
            UICommand component on the greeting.jsp view returns
            the outcome "success".
        </description>
      <from-outcome>success</from-outcome>
      <to-view-id>/response.jsp</to-view-id>
```

```
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
   <description>
        The decision rules used by the NavigationHandler to
        determine which view must be displayed after the
        current view, response.jsp is processed.
    </description>
    <from-view-id>/response.jsp</from-view-id>
    <navigation-case>
        <description>
            Indicates to the NavigationHandler that the greeting.jsp
            view must be displayed if the Action referenced by a
            UICommand component on the response.jsp view returns
            the outcome "success".
        </description>
        <from-outcome>success</from-outcome>
      <to-view-id>/greeting.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  <managed-bean>
    <description>
      The "backing file" bean that backs up the guessNumber webapp
    </description>
    <managed-bean-name>UserNumberBean</managed-bean-name>
    <managed-bean-class>guessNumber.UserNumberBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>minimum</property-name>
      <property-class>int</property-class>
      <value>0</value>
    </managed-property>
    <managed-property>
      <property-name>maximum</property-name>
      <property-class>int</property-class>
      <value>10</value>
    </managed-property>
  </managed-bean>
</faces-config>
```

The task of defining navigation rules involves defining which page is to be displayed after the user clicks on a button or a hyperlink. Each `<navigation-rule>` element defines how to get from one page as defined by the `<form-view-id>` to the other pages of the application. A `<navigation-rule>` element can contain any number of `<navigation-case>` elements that define the page to open next using the `<to-view-id>` based on a logical outcome defined by the `<from-outcome>`. This outcome is defined by the `action` attribute of the component that submits the form (such as the `commandButton` in Code Samples 2 and 3).

In addition, the beans need to be configured in the `faces-config.xml` file so that the implementation can automatically create new instances of the beans as needed. The `<managed-bean>` element is used to create a mapping between a bean name and class. The first time the `UserNumberBean` is referenced, the object is created and stored in the appropriate scope.

Finally, in order to use the JSF framework in your web applications, you need to define the `FaceServlet` and its mapping in your deployment descript file, `web.xml`. This servlet acts as the front controller and handles all JSF-related requests. Code Sample 6 shows the `web.xml` file for the guess number application. An important thing to note is the `javax.faces.STATE_SAVING_METHOD` paramter, which is used to specify where the state should be saved (the client in this case). If you want to save the state on the server (this is the default in the JavaServer Faces reference implementation), then specify `server` instead of `client` in the `param-value`. Note that if the state is saved on the client, the state of the entire view is rendered to a hidden field on the page.

**Code Sample 6**: web.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!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>
    <display-name>JSF Guess Number Sample Application</display-name>
    <description>
        JSF Guess Number Sample Application
    </description>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
        <description>
            Set this flag to true if you want the JSF
            Reference Implementation to validate the XML in your
            faces-config.xml resources against the DTD.  Default
            value is false.
        </description>
    </context-param>
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>true</param-value>
        <description>
            Set this flag to true if you want the JSF
            Reference Implementation to verify that all of the application
            objects you have configured (components, converters,
            renderers, and validators) can be successfully created.
            Default value is false.
        </description>
    </context-param>
    <!-- Faces Servlet -->
    <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>/guess/*</url-pattern>
    </servlet-mapping>
    <security-constraint>
          <!-- This security constraint illustrates how JSP pages
             with JSF components can be protected from
             being accessed without going through the Faces Servlet.
             The security constraint ensures that the Faces Servlet will
             be used or the pages will not be processed. -->
         <display-name>Restrict access to JSP pages</display-name>
         <web-resource-collection>
            <web-resource-name>
                 Restrict access to JSP pages
            </web-resource-name>
             <url-pattern>/greeting.jsp</url-pattern>
             <url-pattern>/response.jsp</url-pattern>
         </web-resource-collection>
         <auth-constraint>
             <description>
                 With no roles defined, no access granted
             </description>
         </auth-constraint>
```

```
    </security-constraint>
</web-app>
```

### Creating Your Own Applications

The easiest way to start creating your own JSF applications is by adapting some of the file from the guess number application. Here is how to get started:

1. Create the following directory (for example, `MyApp`) and its structure under the `samples` directory of your JSF RI:

```
c:\jsf-1_1\samples\MyApp
                    \src
                       YourBeans.java
                    \web
                       YourJSP-pages.jsp
                       AnyImages.gif
                    \WEB-INF
                       web.xml
                       faces-config.xml
```

2. Create the JSP pages and the model objects (beans).
3. Modify the `web.xml` and `faces-config.xml` as needed for your application.
4. Copy the file `build.properties.sample` to `build.properties`. This file is located in the `samples` directory, then set the `tomcat.home` property to the location of the application server installation. And finally, set the other self-explanatory properties as described in the file.
5. To build and package your application, create a `build.xml` file such as the one in Code Sample 7, and place it in your directory (for example, `MyApp`).

   **Code Sample 7**: build.xml

```
<project name="MyFirst" default="build.war" basedir=".">
  <property file="../build.properties"/>
  <!-- Configure the context path for this application -->
  <property name="context.path" value="/jsf-MyFirst"/>
  <property name="example" value="jsf-MyFirst" />
  <property name="build"  value="${basedir}/build" />
  <path id="classpath">
    <pathelement location="${commons-beanutils.jar}"/>
    <pathelement location="${commons-collections.jar}"/>
    <pathelement location="${commons-digester.jar}"/>
    <pathelement location="${commons-logging.jar}"/>
    <pathelement location="${jsf-api.jar}"/>
    <pathelement location="${jsf-impl.jar}"/>
    <pathelement location="${jstl.jar}"/>
    <pathelement location="${build}/${example}/WEB-INF/classes"/>
    <pathelement location="${servlet.jar}"/>
  </path>
  <target name="clean" >
    <delete dir="${build}" />
    <delete dir="${context.path}" />
  </target>
  <target name="prepare" description="Create build directories.">
    <mkdir dir="${build}/${example}" />
    <mkdir dir="${build}/${example}/WEB-INF" />
    <mkdir dir="${build}/${example}/WEB-INF/classes" />
    <mkdir dir="${build}/${example}/WEB-INF/lib" />
  </target>
  <!-- Executable Targets -->
  <target name="build" depends="prepare,deploy.copyJars"
     description="Compile Java files and copy static files." >
    <javac srcdir="src" destdir="${build}/${example}/WEB-INF/classes">
```

```
            <include name="**/*.java" />
        <classpath refid="classpath"/>
      </javac>
      <copy todir="${build}/${example}/WEB-INF">
          <fileset dir="web/WEB-INF">
          <include name="*.xml" />
        </fileset>
      </copy>
       <copy todir="${build}/${example}/">
        <fileset dir="web">
          <include name="*.html" />
          <include name="*.gif" />
          <include name="*.jpg" />
          <include name="*.jsp" />
          <include name="*.xml" />
        </fileset>
      </copy>
      <copy todir="${build}/${example}/WEB-INF/classes/${example}" >
        <fileset dir="web" >
           <include name="*properties"/>
        </fileset>
      </copy>
    </target>
    <target name="deploy.copyJars" if="build.standalone">
      <copy todir="${build}/${example}/WEB-INF/lib" file="${commons-beanutils.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${commons-collections.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${commons-logging.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${commons-digester.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${jsf-api.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${jsf-impl.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${jstl.jar}" />
      <copy todir="${build}/${example}/WEB-INF/lib" file="${standard.jar}" />
    </target>
    <target name="build.war" depends="build">
      <!-- create a war file for distribution -->
      <jar jarfile="${example}.war" basedir="${build}/${example}"/>
      <copy todir=".." file="${example}.war" />
      <delete file="${example}.war" />
    </target>
  </project>
```

6. Modify the `build.xml` shown in Code Sample 6 to suit your needs.
7. Go to the `MyApp` directory and run the command `asant`. Your application will be built and a `.war` (for example, `jsf-MyFirst.war`) file will be created in the `samples` directory.
8. To deploy the application, ensure that the application server is running, copy the `.war` file to the `autodeploy` directory o your Sun Java Application Server 8 (such as `C:\Sun-Studio-Creator\SunAppServer8\domains\creator\autodeploy`).
9. Now, your can test your application using the URL: `httpL://localhost:18080/jsf-MyFirst`.

**JSF Support in Sun Java Studio Creator**

The Sun Java Studio Creator is an Integrated Development Environment (IDE) for developing state-of-the-art web application Based on JSF technology, this IDE simplifies writing Java code by providing well-defined event handlers for incorporating business logic, without requiring developers to manage details of transactions, persistence, and other complexities. In additio supports the web applications architecture as defined in the J2EE BluePrints.

Web applications in the Java Studio Creator development environment are supported by a set of JavaBeans components, ca managed beans, which provide the logic for initializing and controlling JSF components and for managing data across page requests (a single round trip between the client and server), user sessions, or the application as a whole. When you add your own code to components and to the application, you will be adding Java code to these beans. Figure 6 shows the JSF Palette the Java Studio Creator.

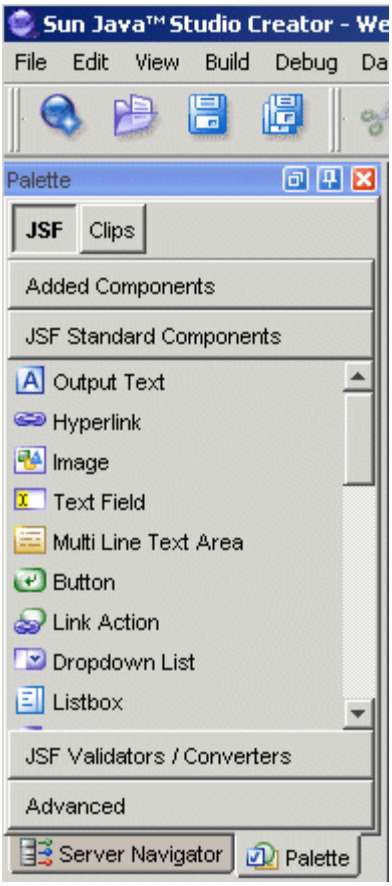


Figure 6: Java Studio Creator's JSF Palette

As you can see, JSF Components are organized into categories:

- Added Components: JSF components you have written or acquired elsewhere and imported into the Java Studio Creator development environment
- JSF Standard Components: The basic set of components including form fields, hyperlinks, labels and text. The JSF components are organized into logical groups by function.
- JSF Validators/Converters: Components that modify or validate data in other components but are not themselves displayed
- Advanced: JSP tags for advanced JSF features. These tags generally do not have a visual appearance and are only useful if you are familiar with both the JSP and JSF technologies.

**Conclusion**

JavaServer Faces (JSF) is a user interface framework for building web applications that run on the server side and render the user interface back to the client. It lets you develop tools that simplify coding web-based Java applications. Sun and other members of the JSF expert group -- which includes Borland, IBM, Macromedia, and Oracle, along with many other companies and individuals -- are evaluating ways to incorporate JSF technology into a new generation of tools that simplify the development of multi tier web-based applications. One of these tools is Sun's Java Studio Creator.

Users of your JSF-based web applications will appreciate the wide range of user actions made available by JSF controls. You can offer more features, more conveniently than you can with a standard HTML front end. And remember, JSF requires little more effort than an ordinary JSP configuration -- but with many more benefits.

**For More Information**

- JSF Technology

- JSF Specification (JSR 127)
- Download the JSF Reference Implementation
- JavaServer Faces Components
- J2EE 1.4 Tutorial (See Chapters 17 - 21 on JSF Technology)
- Struts-Faces (Integration Strategy for Struts and JSF)
- JSF Technology Forum
- JSF FAQ

**Acknowledgments**
Special thanks to Dennis MacNeil, Roger Kitain, and Jim Inscore of Sun Microsystems for their contributions to this article.

.

**Rate and Review**

Tell us what you think of the content of this page.

**Excellent** **Good** **Fair** **Poor**

**Comments:**

**If you would like a reply to your comment, please submit your email address:**

Note: We may not respond to all submitted comments.

Submit »