HELLO WORLDWIDE WEB: YOUR FIRST JSF IN JDEVELOPER

Peter Koletzke, Quovera

There are three things I always forget.
Names, faces, and—the third I can't remember.
—*Italo Svevo* (1861–1928)

The purpose of frameworks and of new development technologies in general is to make development within a complex environment easier. Technologies evolve so that development can be faster, cheaper, and more accurate. It is one thing to develop a technology and create guidelines about how best to use it. It is another to actually apply that theory to a real-world application development project. The proof of the technology and its methods comes from applying it. Therefore, as a natural course in the evolution of a technology or framework, features and methods are added that improve development productivity. At some point, needs arise that require core changes to the original technology. At this point, the technology's authors must decide to evolve it with a rewrite or to replace it with another technology and end the life cycle of the original technology.

This kind of evolution has occurred with Java EE web technology. Java EE web technology evolved from Java applets running Java code in a Java runtime in the browser to HTTP servlet code written in pure Java and running on a web server. This evolution was a major shift in development and deployment environments because it meant that applications could run in a Java runtime environment on a centralized server—as opposed to the local desktop—and send HTML from those applications to the client's web browser (instead of requiring the application to be deployed to the client's desktop).

The next major technology shift in application styles was from servlets to JavaServer Pages (JSP) technology, which mixes HTML and Java tags. JSP pages are translated into and run as servlets, so they share the deployment environment, but the code is different; instead of the pure Java coding of servlets, JSPs use a mixture of HTML tags and JSP tags that can contain calls to Java class files. Therefore JSPs require less hands-on work with Java.

The next major evolution in Java EE web application technology was from JSP pages to JavaServer Faces (JSF) pages. JSF technology was ratified as a standard in March 2004 through the *Java Community Process (JCP),* which Sun Microsystems (now Oracle Sun) used to obtain the participation of Java experts when introducing new technologies and existing technology upgrades (jcp.org).

Material about JSF architecture and design principles abounds on the Web. This white paper does not attempt to repeat all of this information. Instead it focuses on what you, as a developer who may be new to this technology environment, need to know to be productive in creating your first JSF. The white paper answers the following questions:

- Why was JSF developed?
- What JSF concepts do I really need to know now?
- Where can I find additional information about JSF?

Note

This white paper provides code examples using the JSF components provided with JDeveloper 11*g*—ADF Faces Rich Client (ADF Faces RC, or just "ADF Faces"). It also refers to Application Development Framework (ADF) tools and techniques used within JDeveloper to create application code. Other Java tools will offer different facilities to create JSF code but the basic concepts of the files and code you create is the same between tools.

Why Was JSF Developed?

The popularity of JSP technology grew rapidly because it was easier to use than the preceding technologies. Instead of building up an HTML page using print statements and coding low-level details of the HTTP communication in a servlet file, developers could concentrate on creating the HTML page using HTML tags and JSP (Java-oriented) tags. They could use HTML visual editors for the visual design and Java tools for the required Java logic. In the early days of JSP technology, developers used the core JSP tags with Java *scriptlets* (small bits of Java code embedded in tags within the JSP file) added for custom functionality. The scriptlets supplied logic to conditionally display items or to retrieve data. They would also supply logic that determined page navigation (page flow).

However, developers found themselves writing the same scriptlets in each application; this made the code difficult to maintain, so they started creating their own custom tags to perform the same function as those scriptlets (for example, drawing an HTML table containing rows and columns of data). They then stored the tags in tag libraries and referenced these tags in each application.

This served the Java community up to a point, but, even though frameworks supplied many of the custom tag libraries required for an application, the Java platform standards (Java EE) did not regulate the standards on which these libraries were built. In addition to problems with design consistency, developers found JSP custom tag libraries lacking consistency in a number of areas. JSF grew out of the need to address inconsistencies and inefficiencies in areas such as these:

Components

The days of writing JSP files that included large amounts of JSP scriptlet tags and JSP expression tags came to an end when developers found that approach nonproductive for code modifications and framework usage. Wrapping complex functionality and display characteristics into a single tag—the *UI component* (or just *component*)—became a goal for most tag libraries. As mentioned before, a single component could query a database table and present an HTML table structure, with data from the query arranged into rows and columns in the table. JSF supplies similar, high-level controls and supports them with a standard. Developers spend little or no time coding HTML layout tags and spend more time on the arrangement of components and how they are used to solve business requirements. They define the functionality and display characteristics of the component by setting property (attribute) values and arranging them inside layout containers.

Layout Containers

Layout containers (also called *layout components, container components,* or *containers*) are components that provide a structure for the components nested inside them (using the tag hierarchy). When you add components to a JSF page, you plug them into areas inside the layout container. The child components will maintain their relative position if the user resizes the window at runtime. For example, an ADF Faces layout container, `af:panelGroupLayout`, can display child objects in a vertical or horizontal arrangement.

Another benefit of using layout containers is that you do not need to worry about coding HTML tables or CSS selectors to position components in specific locations in the page. The HTML and CSS code is created for you based on the functionality and areas defined for the container. Another benefit of layout containers is that they abstract the component arrangements from the specific markup language. For example, you can stack components into an `af:panelFormLayout` layout container. If the JSF page appears in a web browser, the components will appear inside an HTML table. If the display is in a wireless device, the components will appear inside the corresponding table structure specific to the wireless markup language.

Support for Different Display Devices

Before JSF, running an application on different display devices, such as a web browser or a PDA, usually required rewriting the View layer code to use a different technology. Tag libraries that supported multiple display devices were, again, not standard. JSF standardizes the View layer components from their display using a render kit specific to a certain style of code.

Controller

The MVC design pattern specifies that a separate code layer (the Controller) defines page flow and data communications. In addition, the Front Controller design pattern describes a single handler for all requests in an application. The patterns are clear, but in the past, the developer was responsible for implementing them. Open-source controller frameworks, the most popular being Apache Struts, removed the need for developers to implement custom controllers and provided the infrastructure needed to manipulate page flow. However, these controller frameworks did not tie the Controller layer to View layer tag libraries in a standard way. JSF includes a Controller layer that is integrated with the JSF View layer tag libraries and defined in the Java EE specifications.

Event Processing

User events, such as a button press, had no standard handling mechanism in web technologies before JSF. JSF defines an event model for its tags using listeners and actions.

Note

JSF was a new addition to the Java EE specification in release 5, Java EE 5—now usually shortened to "Java EE."

What JSF Concepts Do I Really Need to Know Now?

Fortunately, just as with most other modern frameworks, you only need to know a subset of everything there is to know about JSF concepts to start creating JSF pages. However, as is true of any technology, the more you know, the more creative you can be when faced with a development challenge. The resources mentioned in the later section "Where Can I Find Additional Information About JSF?" will continue your JSF education.

What Is JSF?

JSF consists mainly of specifications that resulted from the JCP (jcp.org), focused on Java Specification Request (JSR) 127. The specification was revised as JSR 252 and is included in the current Java platform specifications, Java EE 5. In principle, JSF technology supports any type of client device and programming code markup language. However, Oracle offers a *Reference Implementation (RI)*—code libraries that prove that the standard supports real code—for servlets (as class libraries) and JSP technology (as tag libraries). These libraries contain classes and tags for the components and functionality described later in this section. Since RI libraries are tested and proven implementations of the standard, you can use them as a basis for your own code. The RI also includes a *render kit* (a code layer that translates JSF tags into a specific output format) for HTML output from JSP components.

Note

MyFaces, an open-source, JSF implementation is available at the open-source Apache website page myfaces.apache.org. An earlier version of ADF Faces (10.1.3) was donated to the public domain to kick off the Trinidad project of MyFaces.

Understanding JSF requires knowing a little about each of these concepts:

- Runtime architecture
- JSF-oriented files
- Components
- Designing cross-platform interfaces

Runtime Architecture

JSF code runs in a Web Tier container in the same way as servlet code. Figure 1 shows the main actors in the request-response round-trip communication for a JSF file. JSP files are the standard view technology used for JSF components running on the Web; you usually code JSF inside a JSP file (although JSF, theoretically, supports multiple-view layer technologies). In this example, the JSF code is contained within a JSP file. The browser issues an HTTP request to the web server. The web server finds the URL's virtual directory name in the web.xml file and passes the request to the Faces Servlet that is running (or that will be started) on the web server. The Faces Servlet handles the request by passing control to an object created using the life cycle class (`javax.faces.lifecycle.Lifecycle`). This `Lifecycle` class processes the JSF file using a set of steps called the *JSF Life Cycle* (described later in this section).

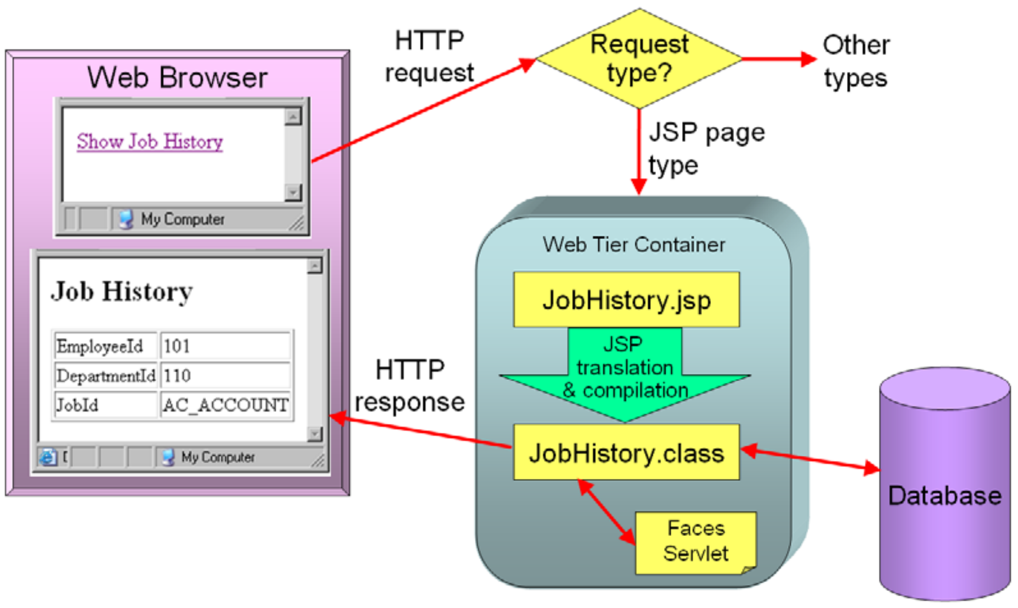


Figure 1. JSF JSP runtime

Note

The Faces Servlet initializes an object typed from the `FacesContext` class that stores state information and data pertaining to the user interface elements, as well as information about the request session. You can tap into the `FacesContext` object in your code.

The servlet accesses any necessary Model layer code to obtain values from the data source (such as a database). The servlet instantiates the bean if a component property value refers to an element in a backing bean (described in the sidebar "About Managed Beans and Backing Beans"). The Faces Servlet then assembles the page (shown here with a .jsp extension identifying it as a JSP page). The file runs through the normal JSP translate and compile steps. The application then assembles an HTTP response using a render kit (in this case, the HTML render kit), which converts the display elements to an HTML page. It then sends the response to the web browser. The process for other display types, such as for a PDA or cell phone client, is similar, although their render kits are different.

About Managed Beans and Backing Beans

A bean in Java (also called a "JavaBean") is a class file that conforms to a standard format. A managed bean (MBean) is a Java file containing code used to handle operations and data for a resource (in this case, for components on the page). In JSF work, a managed bean is manipulated (instantiated and removed) by the JSF controller. It can contain code for one or more pages or code that is shared between pages (for example, a logout action available on many pages). The JSF framework controls the instantiation and subsequent calling of this managed bean. The general convention is to use one managed bean for each JSF page, although some prefer a single managed bean for more than one page in the application.

The term backing bean is used to denote a managed bean assigned to a single page. A JSF page may not require special code for its components; in this case, you would not create a managed bean for that page. Often, the term "backing bean" and "managed bean" is used interchangeably.

JSF Life Cycle

The `FacesServlet` and `Lifecycle` objects work together to process the HTTP request and assemble an HTTP response using the six steps described in this section (represented in Figure 1). A JSF application can support requests to non-JSF components, but processing of a non-JSF request follows an abbreviated version of the life cycle. This description focuses on the process for requests that address JSF components.

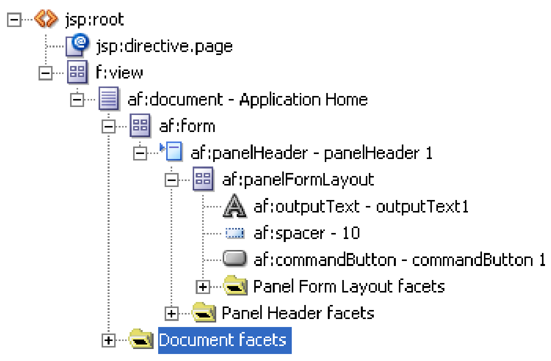
The rough idea is that the steps proceed in the order indicated by their numbers. However, as the servlet processes events it may bypass one or more steps. For example, if there is an error in step 2, an error message would normally be queued for display later in the life cycle. However, the JSF code can be written to jump directly to step 6 (Render Response) and bypass the intermediate steps.

Note

Various sources, including some from Oracle, refer to the "JSF life cycle" as the "JSF lifecycle." This white paper refers to the term "life cycle" but the two forms are synonymous.

1. Restore the Component Tree

The components used in a JSF application are arranged in the file in a hierarchical way. This hierarchy, the *component tree* (also called the *UI tree*), corresponds to the tag hierarchy you would see in an HTML file (plus any other files or components included on the page at runtime). For example, the illustration on the right shows the JDeveloper Structure window view of a JSF home page.



```
jsp:root
  jsp:directive.page
  f:view
    af:document - Application Home
      af:form
        af:panelHeader - panelHeader 1
          af:panelFormLayout
            af:outputText - outputText1
            af:spacer - 10
            af:commandButton - commandButton 1
            Panel Form Layout facets
          Panel Header facets
      Document facets
```

Two types of page requests affect JSF processing: the *initial request*, where the user requests the page for the first time; and a *postback*, where the user submits a page that was previously loaded into the browser. For a postback, the component tree is already created in the `FacesContext` object. It is stored there when the page is first rendered (in step 6). For an initial request, the component tree has not been created in the `FacesContext`, so processing passes from this step directly to step 6 to render the content.

Note

The root component in a JSF file is f:view. All other JSF tags will appear within this tag. A JSP file may have element tags outside the f:view tag (such as taglib directives), but these are used to assist in processing the JSP file.

2. Apply Request Values

In this step, the `Lifecycle` instance populates the components in the component tree with data from the request parameters. These values are held in a buffer area and the process does not validate the assigned datatypes yet. Events that apply to this step are also processed.

3. Process Validations

This step applies *conversions* (defined in *converter* components) that translate the plain-string value loaded into the buffer for each component in the preceding step to the datatype assigned to that component.

You can define *validation* (using validator components)—for a component to check the value assigned to it. For example, you can define a minimum and maximum value validator. This step applies these validations.

Validations and conversions are bypassed for a particular component if that component is assigned an *immediate* property value of "true." You would set this value for a component that you would not want to validate, such as a button.

If any validation or conversion fails, the error messages are queued and the process jumps to step 6 for rendering the response. In addition, event code written for this step is executed. Errors or messages from these events are also sent to the message queue.

4. Update Model Values

Just as step 2 sets values for the components in the component tree from the request, this step sets the values for the corresponding components in the Model layer code. As before, if errors occur in validations or conversions, the messages are queued and the process jumps to step 6 for rendering the response. As with all phases, appropriate events are also processed or queued.

5. Invoke Application

This step performs application-level tasks, such as page navigation, as appropriate to the request. Events are handled if the page being restored contains components that fired events. Therefore, this step runs any event code you have written.

6. Render Response

If this is the first request to a page, control is passed from step 1. The `FacesServlet` object builds the component tree as specified in the JSF file and stores the tree in the `FacesContext` object. The framework then requests components in the tree to be rendered; this results in the components using the appropriate render kit to output markup text (such as HTML), which the JSF servlet buffers into the final page definition.

Messages that were queued from previous steps are written to the output as well. In addition, the state of the page (for example, the component data values) is stored so that subsequent requests can use it for the restore view step. This step ends with the Faces Servlet sending an HTTP response to the browser containing the buffered page output.

Working with the JSF Life Cycle

During most JSF development work that you perform, the life cycle is transparent. However, knowing a bit about the life cycle steps should give you an idea about the richness of the environment. It can also assist when debugging or when you need to write specialized code to alter the normal runtime behavior. You can control the JSF runtime process with custom code. You can also customize the life cycle using additional ADF life cycle steps. JDeveloper provides the *ADF Declarative Debugger,* which can display details about the ADF life cycle runtime.

JSF-Oriented Files

When you set up a project for JSF work in JDeveloper by including the JSF technology scope in the project (or using a JSF application template), the web deployment descriptor file (web.xml) is automatically modified so that it contains a definition that the Faces Servlet will load when the application starts as shown here:

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

Additional code in web.xml sets up a pointer for requests that contain a designated virtual directory (such as “/faces/”) as in the following code:

```
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Note

For an ADF application, web.xml will contain references to additional servlets and filters.

You will interact with the following types of files when developing a JSF application: managed beans, faces-config.xml, the message bundle, and the JSF JSP file.

Managed Beans

As mentioned in the section “Runtime Architecture” previously, a managed bean may be associated with the JSF JSP page file. For example, a login.jsp page file would use a managed bean file typically named “Login.java.” The Java file contains the code needed to process events that are outside of the declarative framework.

In addition, the managed bean can contain accessors (getters and setters) for the objects on the page (for example, text fields), so you can manipulate the values of these objects programmatically. These accessors manipulate a private object for each component in the JSF file. This object and its accessors can be used to change the characteristics of the object as well as its behavior.

For example, if your JSF file contains a text input field, lastName, the managed bean would contain a lastName `HtmlTextInput` object as well as a `getLastName()` method to retrieve the value and other properties of the object, and a `setLastName()` method to assign the object's properties, as in the following example.

```
getLastName().setRendered(false);
```

In this code, the getter, `getLastName()`, provides access to the component and `setRendered()` changes the *rendered* property value.

Note

Working with a managed bean is necessary only if you need to change the default behavior or provide special processing for the page. You do not need to create one otherwise.

EVENTS AND ACTION LISTENERS

One of the strengths of the JSF framework is its rich model for event handling. Events occur due to some user interaction, such as a button click or list selection. You can write *action listener* methods in the managed bean to handle these events. For example, an OK button called "okButton" in a JSF page could have a corresponding action listener method called `okButton_action()` written in the managed bean. When the user clicks the OK button, the code in this method would be run. JSF action listener methods are, by convention, named with a prefix of the component they work for. A single-action method can be reused to provide the event handler for many components.

The action listener name is associated with the component using a property, such as *Action* for a button component. Alternatively, you can code a separate listener class (that implements the `ActionListener` interface). You then associate that class with the component using a different property, such as *ActionListener* for a button component.

Note

If JSF components render as HTML controls in a web browser, you can set event properties, such as *onClick,* on JSF components to run JavaScript code you have written. With ADF Faces, use an `af:setClientListener` component under an ADF Faces component to define the JavaScript event.

faces-config.xml

This XML file, called the *application configuration resource file,* contains definitions that specify the following:

- **Navigation rules** for defining flow from one page to another
- **Managed bean definitions** for making JavaBean code accessible in the application
- **Render kits** that are used to display the page in a certain encoding, such as HTML or WML
- **Converters**, mentioned before, which change the type of data from the plain-vanilla string supplied by the HTTP request parameter value to a specific datatype for a component value
- **Validators**, mentioned earlier, which check the value of the data (for example, the range of the value or its length)

Although this file is coded in XML, JDeveloper offers a visual representation of the navigation rules in the JSF Navigation Diagram. You can code other entries in this file using the code editor or a property editor available from the Overview tab.

A NOTE ABOUT STRUTS

The open-source, highly evolved Struts framework has become popular for the Controller layer of a JSP or servlet application. Struts is not supported by the Java EE standards, but due to its widespread use, Struts was a very popular technology choice before JSF technology was released. The JSF controller serves the same purpose for JSF applications as Struts served for JSP applications. Therefore, you do not need to use Struts when creating a JSF application. However, JDeveloper supports working with Struts and provides a visual tool, the *Struts Page Flow Modeler,* which you can use to interact with Struts components defined in the struts-config.xml file.

Message Bundle

JSF offers the ability to store *messages*—text such as prompts, errors and warnings, button labels, and tab labels as text strings inside a *message bundle*, a Java file or properties file that contains key-value pairs defining the text displayed to users (also called a *resource bundle*). You can access this file programmatically in the backing bean (or other Controller) code as well as declaratively within EL expressions. The benefits of message bundles are as follows:

- **They centralize messages**. That way, you can change any message in the application by editing a single file.
- **They allow you to internationalize (localize) the application**. You can create separate message bundle files for each language you need to support. Each message bundle would contain all messages used in the application translated into the local language. These language files share the same file name prefix and a suffix indicating the language. The JSF runtime engine retrieves the locale information from the browser's settings and uses the appropriate file to supply messages to the application.
- **They make it easy to replace default validation messages**. The JSF framework allows you to code validators for each component by embedding a child tag—such as `f:validateLength minimum="8" maximum="20"`—inside the opening and closing tags of the component (for example, `af:inputText`). This validator calls default functionality and displays a default error message under the component (such as a field). You can replace the default error message using message bundles. The sidebar "Using a Message Bundle for JSF Error Messages" describes this a bit more.

Using a Message Bundle for JSF Error Messages

Default JSF messages are not completely user-friendly. For example, the standard JSF message for a required field (and InputText component with the Required property set to "true"), which the user has left blank, is "Validation Error: Value is required." This message does not indicate the problem field name, and the wording may not be as descriptive as you might like.

To replace messages such as these, you create entries with predefined names (keys) in the message bundle file. For example, naming the message "javax.faces.component.UIInput.REQUIRED" will cause the JSF runtime to replace the default error message for required text with whatever you have assigned to that message. You can find the JSF error message names in an NetBeans article "Customizing a Standard Validator Message" (netbeans.org/kb/docs/web/convert-validate.html#08). This technique will only redefine the default message that the framework provides.

JSP Page and JSP Document

The *JSP page* is a file with a .jsp extension that contains both HTML tags and JSP-oriented tags. It is the traditional style used for JSP files. The benefit of this file type is that it allows you to use HTML tags, which you can display and design in an HTML editor such as Dreamweaver. JDeveloper also provides HTML editing and viewing capabilities, including a formatting toolbar that helps you enter HTML tags in a visual way.

The *JSP document* is a file with a .jspx extension that contains only well-formed XML code. In this case, *well-formed code* means that all start tags have corresponding end tags and all tag elements are named with lowercase names.

Although some browsers can interpret HTML tags without closing tags, XML parsers are stricter than browsers about requiring well-formed code. JDeveloper provides a way to view JSP document code in a visual way and to edit it using the Property Inspector and some drag-and-drop functions.

SHOULD I USE A .JSP OR A .JSPX FILE?

You can use either JSP pages or JSP documents for creating JSP code that contains JSF tags. However, since JSF architecture supports multiple display device types, you can choose to use a tag library such as ADF Faces that does not require any HTML. Therefore, the separation of HTML and JSP tags that a JSP page offers is not important. Page design consists more of selecting which tags to use and of setting their properties. Those tags will draw major areas of the page without the requirement for HTML design.

Using a format that is independent of the display device technology can make your application more flexible. Whether your application will support multiple display devices is still not a firm decision point for using either JSP pages or JSP documents because you can code both JSP pages and JSP documents without HTML. However, it is more difficult to write HTML within a JSP document, and selecting the JSP document as a standard may help enforce portability. It is a natural choice for files you will be editing only in JDeveloper, because JDeveloper is built to help you lay out and maintain JSF tags and ADF Faces in a visual way. Other HTML editing tools support layout with JSP pages better, and if you will need to design the page using a tool other than JDeveloper, a JSP page might be a better choice.

Note

Fusion applications are written using JSP documents.

WHY USE JSP DOCUMENTS?

As you may have gathered from the preceding discussion, the choice of whether to use JSP pages or JSP documents for JSF work is almost arbitrary. However, developing an application in a way that is less dependent upon a display device can have strategic advantages. For example, if another display device ever becomes more popular than the web browser, an application written using a portable language (such as ADF Faces inside a JSP document) will have a better chance of running in that new device without being rewritten.

Another strong advantage of developing JSP documents in JDeveloper is code validation. The IDE applies rules for validating syntax, and it shows some syntax errors immediately after navigating off the line of code. The stricter rules for validating XML tags will ensure that your file contains only well-formed code. In addition, and most important, some features of ADF Faces work only when using the XML style enforced by JSP documents. JSP documents are therefore advantageous because of their portability and better support for writing better code.

Components

JSF components for JSP work are contained in two tag reference implementation (RI) libraries:

- **JSF Core** This tag library contains components, such as validators and converters, that are used in conjunction with other components. Components in this library use the "f" prefix, for example, `f:loadbundle`.
- **JSF HTML** This tag library contains HTML user interface components, such as text input, buttons, labels, and radio options. You refer to these components using the "h" prefix, for example, `h:datatable`, `h:column`, and `h:form`.

Note

As mentioned, the JSF standard supports multiple output device styles through the use of render kits, but the initial RI only offers an HTML render kit for JSP files. ADF Faces RC also provides render kits for HTML clients. Its predecessor, ADF Faces offers render kits for TELNET and wireless clients as well.

Facets

Some components may be placed inside of other components as a child object in the component hierarchy, for example, an `af:column` inside an `af:table`. In addition to nesting related components, you can nest components using the *facet*, a named subcomponent. It is up to the component to determine where to render the contents of a particular facet. For example, the ADF Faces `af:panelBorderLayout` component supports top, bottom, start, and end facets. Items in these facets will appear in specific positions within the page, as shown in Figure 2.

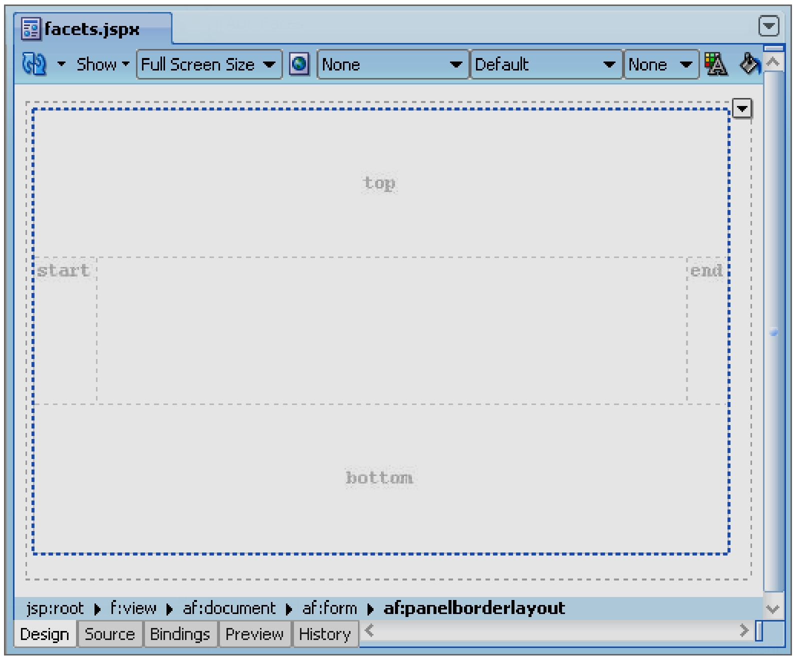


Figure 2. af:panelBorderLayout facets

Note

A facet can only contain one component, but that one component can be a layout container (such as `af:panelForm`), which can contain more than one component.

The JSP/HTML Visual Editor provides special support for facets in a component by displaying *facet drop zones* into which you can place components. These drop zones are shown as dotted boxes in Figure 2. They reflect the relative layout of the components at runtime.

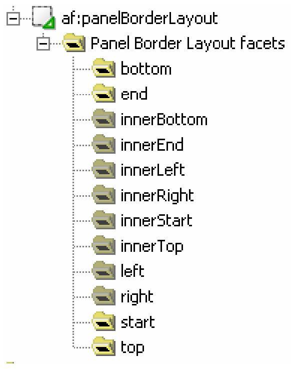
Note

You do not need to use all facets for a component.

Facets are important in controlling and adapting standardized layouts (skins) within JSF applications, because they provide a type of named API for component children that can be changed without having to recode the page.

Tip

To quickly navigate to a component and expand the Structure window hierarchy to a particular component, select that component in a visual editor.



The bottom, end, start, and top facets for `af:panelBorderLayout` display drop zones by default, but you can drop components into the Structure window for other facets (or for the default facets), as shown on the right.

You can control the facets displayed in the JDeveloper visual editor by checking the relevant facet names in the right-click submenu selection, **Facets – <component>**, on the component in the Structure window, as shown in Figure 3.

ADF Faces Rich Client

ADF Faces Rich Client (or just "ADF Faces") is a set of component tags contained in class libraries and tag libraries. ADF Faces (in JDeveloper 10.1.3) evolved to allow access in JSF code to the enhanced component set offered previously as ADF UIX. ADF Faces is designed to be displayed on different devices.

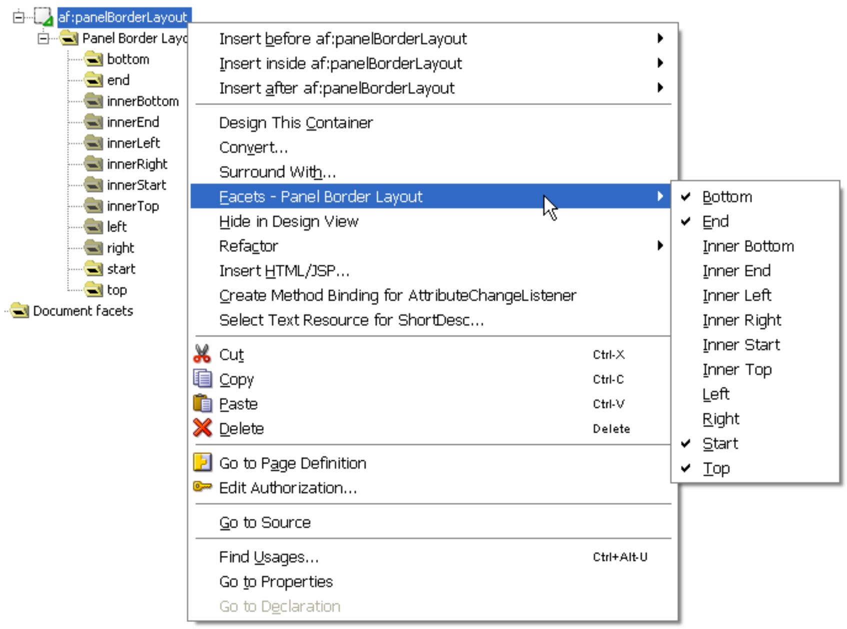


Figure 3. Displaying or hiding facets in the visual editor

Note

ADF Faces RC requires a Java EE container that is capable of running JSF 1.2 code such as Oracle WebLogic Server 10.3 or later. Oracle Application Server's OC4J 10.1.3 is not capable of running JSF 1.2 code. ADF Faces Rich Client also requires a client browser of Internet Explorer 7.0 or later, Mozilla Firefox 2.0 or later, or Safari, version 3.0.

HTML and ADF Faces

When working with ADF Faces, you do not code HTML tags directly. Instead, you use ADF Faces components that will be rendered in a web browser using HTML tags. For example, ADF Faces offers a component, `af:inputText`, which renders an HTML form input item (a text field) when displayed in a web browser.

ADF Faces Layout Container

As mentioned before, layout containers hold other components. The process of creating a file with ADF Faces consists of adding layout containers tags, and then inserting components within the containers. ADF Faces offers containers, such as `af:panelBorderLayout`, that contain a number of predefined areas, as shown in Figure 2. These predefined areas use JSF facets to provide default positions for objects on the page. After dragging this component onto the page, you can drag other components to each of the facet areas you need to use.

All ADF Faces containers provide automatic layout capabilities that reposition or stretch child components when the container is resized.

Note

Mastering the layout containers is a key skill needed to achieve the perfect UI layout.

Tag Libraries

Just as JSF offers Core and HTML tag libraries, ADF Faces offers Core tag libraries (prefixed with "af"). It also offers *Data Visualization Tools (DVT)* tags for graphs and charts. This tag library is represented with a dvt prefix. Some of the ADF Faces components are parallel to JSF components. For example, the JSF RI component `h:inputText` has an ADF Faces equivalent, `af:inputText`, used to present a text entry field. You can distinguish between these components in code, because of their prefixes (“h” for JSF RI and “af” for ADF Faces). Also, the ADF Faces tags offer more properties than the RI tags.

These prefixes for components that you use on the page are defined in a `jsp:root` tag in the beginning of the JSF JSP file as follows:

```
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
    xmlns:af=http://xmlns.oracle.com/adf/faces/rich
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:dvt="http://xmlns.oracle.com/dss/adf/faces">
```

Note

JDeveloper automatically adds the appropriate taglib directive for a tag library the first time you drag a component from that library onto the page. Technically, you can change the prefix in the taglib directive, but it is best to stick with the default prefixes that JDeveloper assigns.

ADF Faces Properties

In general, ADF Faces components offer more properties and built-in functionality than the JSF RI components. For example, the ADF Faces `af:inputText` component includes a *label* property (for a prompt) that is not included with the comparable JSF RI component. Also, unlike the JSF RI component, the ADF Faces component validates items marked as required using JavaScript, which does not require a communications trip to the server.

Binding to Data

ADF offers a technology, ADF Model, which allows an easy method to connect (bind) the UI components to data in a database. An Expression Language (EL) property value ties the component to a binding file that refers to objects you build in an ADF Business Components (ADF BC) project. The ADF BC objects, in turn, define database objects and offload the normally tedious chore of writing Java Database Connectivity (JDBC) code that communicates to the database. For example, an ADF Faces text item component (`af:inputText`) might be coded with EL in the following way:

```
<af:inputText value="#{bindings.FirstName.inputValue}"
              label="#{bindings.FirstName.hints.label}"
              required="#{bindings.FirstName.hints.mandatory}"
              columns="60"
              maximumLength="#{bindings.FirstName.hints.precision}"
              shortDesc="#{bindings.FirstName.hints.tooltip}" id="it1">
</af:inputText>
```

In this example, properties such as *value* are assigned the results of the EL expression (delimited by “#{ }” characters). All EL values in this example point to a *bindings* layer that ultimately refers to properties of elements in ADF BC. The *value* property in this example refers to the FirstName attribute’s *inputValue* (the actual data of the FIRST_NAME column in a row of a table). The ADF technique of dragging a data-aware component onto the JSF visual editor automatically creates these binding expressions and the underlying code required to link them to the ADF BC layer.

Messages

As the application runtime proceeds, messages are written to the `FacesContext` object. They are placed into a queue until the last step of the life cycle renders a response. You can programmatically add messages to a component (`af:messages`) using managed bean code.

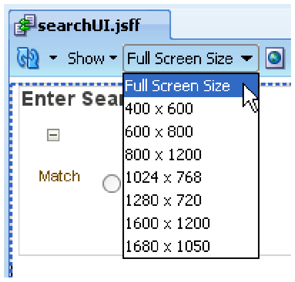
JSF messages support the idea of specific text for specific locales and languages. To implement this feature, you abstract the messages out of the JSF JSP file into message bundle files (described earlier) that hold all text for the application (one message bundle for each location or language). The JSF controller matches the correct language file and uses it to supply messages for the application.

Designing Cross-Platform Interfaces

ADF Faces and JSF support the concept of displaying the user interface on different types of client devices. The render kit used to display the application is specific to each device’s needs, but the same JSF code can display a user interface in a web browser or cell phone or other wireless device. If the application needs to support multiple client platforms, you would work this requirement in from the starting point. You would need to set a standard that code in the application avoids the use of HTML tags. In addition, you would create a standard to avoid use of JavaScript and CSS, both of which are add-ins to HTML browser code that may not work on other display devices.

Tip

The JDeveloper visual editor offers a screen size pulldown (shown next) that you can use to see how the layout you have designed will appear in different screen resolutions.



Where Can I Find Additional Information about JSF?

This white paper describes most of the JSF basics you will need to know to start working with JSF technology. However, you will likely need other resources for learning about JSF. The authors have found the following resources useful.

Websites

Information about JSF concepts abounds on the Internet. The following websites are good places to begin further research:

- **JSF home page**, on Oracle Technology Network (OTN), currently at www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html (or google “oracle technology jsf”); also perform a search on OTN for ADF Faces for articles about that subject.
- **JSF Central**. This website (www.jsfcentral.com) contains many articles and discussions about JSF.
- ***JSF Tutorial***. This is an online book by Marty Hall, currently at www.coreservlets.com/JSF-Tutorial.
- ***JavaServer Faces Resources***, by James Holmes, www.jamesholmes.com/JavaServerFaces.
- ***Java EE 5 Tutorial (Chapter 10)*****.** This is an online book currently at download.oracle.com/javaee/5/tutorial/doc/.
- **JSF Tutorials** at www.jsftutorials.net.
- **OnJava**, at www.onjava.com, by O’Reilly Media, Inc. This website contains articles such as “Introduction to JavaServer Faces”(onjava.com/pub/a/onjava/2004/04/07/jsf.html).
- **Rich Enterprise Applications**, showcases and provides a launchpad for more information about ADF Faces components (rea.oracle.com).

Caution

The website addresses presented are current as this white paper is being written. If your requests to these addresses fail, use a web search engine to look for their new locations.

Printed Books

You can supplement information about JSF from the Web using printed books, such as the following:

- *JavaServer Faces in Action*, by Kito D. Mann (Manning Publications, 2005). This is a good book for overall JSF learning and reference.
- *Core JavaServer Faces, Third Edition*, by David Geary and Cay Horstmann (Sun Core Series, Prentice Hall PTR, 2010). Some chapters of this book are also available online at www.horstmann.com/corejsf.

- *JavaServer Faces 2.0: The Complete Reference*, by Chris Schalk and Ed Burns (McGraw-Hill/Professional, 2009). This book is an in-depth look at JSF technology.

Conclusion

This white paper has introduced the fundamentals of JSF development work. It explained how JSF is accessed and interpreted at runtime and discussed the aspects of code you need to write when creating JSF applications. This information should help you when you are faced with creating your first JSF file. As with the learning process for any new technology or language, it is a good idea to start your first JSF work using a simple application so you can master the basics before trying to take on complexities of a major application. In this way, you should be able to become comfortable with how JSF works, which should enable you with the ability to address non-standard requirements with grace.

About the Author

Peter Koletzke is a technical director and principal instructor for the Enterprise e-Commerce Solutions practice at Quovera, in Mountain View, California, and has 27 years of industry experience. Peter has presented at various Oracle users group conferences more than 280 times and has won awards such as Pinnacle Publishing's Technical Achievement, ODTUG Editor's Choice (twice), ODTUG Best Speaker, ECO/SEOUC Oracle Designer Award, ODTUG Volunteer of the Year, and NYOUG Editor’s Choice (three times). He is an Oracle Certified Master, Oracle ACE Director, and coauthor (variously with Dr. Paul Dorsey, Avrom Roy-Faderman, and Duncan Mills) of eight Oracle Press development tools books including *Oracle JDeveloper 11g Handbook* (from which some of the material in this white paper is taken).

Tip

Try out the techniques for developing a JSF in JDeveloper by following a hands-on practice available with the presentation file and this white paper in the conference papers section of the ODTUG website.