



Step 7 of 12: Designing Better User Interfaces

Mastering J2EE Application Development

Learn how to simplify the J2EE application development life cycle in 12 easy steps

Designing and Implementing Web Application Interfaces

by Hans Bergsten

Learn how to create functional and easy to use web application interfaces.

Downloads for this article

- ✖ [Code Sample](#)
- ✖ [ADF Faces](#)
- ✖ [Oracle JDeveloper 10g](#)



Designing a good user interface is never easy, but designing a web application interface is especially challenging. In this article, I first give you some general advice about design considerations for web application interfaces and about how to pick the appropriate implementation technology. I then dive into the details about one aspect of a web application interface using JavaServer Faces, specifically, how to customize the standard error messages.

Design Considerations

There are many things to get right in order to end up with a functional and easy to use web application interface:

- **Navigation.** Since a web application runs in the browser, users expect to be able to use the browser's buttons for navigation, such as using the Back button to return to the previous page. However, handling this type of navigation correctly is tricky, so it's important to design an interface that encourages the user to use other means for navigation. I recommend that you design the user interface to look as much as possible like a traditional graphical user interface (GUI) — using common GUI widgets, such as trees for selecting items, tabs to show different aspects of selected items, and menu-bar links at the top. Also, keep in mind that a web application is task-oriented, comprised of pages that must be entered in a specific sequence, rather than free-form set of links as in a traditional web site. Users must be allowed to jump directly to specific pages only, such as the main pages for different tasks.
- **Bookmarks.** Bookmarking is related to navigation and can also be hard to support in a web application; you don't want a user to bookmark a page that should only be accessed as a result of submitting a form, for instance. Although I dislike HTML frames on a regular web site, frames can be useful in web applications because they prevent users from bookmarking individual pages.
- **Web application limitations.** Let's face it: a web application can never be as interactive as a traditional GUI application, at least not with the current browser technologies. In a GUI, it's easy to let the user select multiple rows in a table and delete them all in a single click. In a web application, on the other hand, you must handle this differently, for instance by placing a checkbox on each row that the user can check to select a row. A traditional GUI also makes it easy to dynamically enable or disable input widgets based on user input, such as when a radio button or a checkbox is clicked. Implementing dynamic interface components in a web application typically requires JavaScript code, so users whose browsers have JavaScript disabled won't be able to use the application. Unless you have complete control over your user base, you should provide other ways for presenting different options to users, such as combining radio buttons with a button to activate the new

choice, or using links for the choices.

- **Page size.** Although people are accustomed to scrolling around the page to read a complete article online, say, users find Web applications easier to use when all information related to each task is contained in a single readable page, or if scrolling is contained in just part of the page (with the help from internal frames), or by using Next/Previous buttons for a large table, for example.

In my experience, a great way to verify as early as possible that the user interface makes sense is to work with mockups and User's Guide drafts. Before writing a single line of code, I create plain HTML pages and use them for screenshots in an early version of the User's Guide that I ask teammates and, ideally, end-users to review. If you haven't tried this tactic before, you'll be surprised at just how well it works to uncover not only pure interface design issues, but also misunderstood requirements, missing functionality, and many other issues.

"A great way to verify as early as possible that the user interface makes sense is to work with mockups and User's Guide drafts."

Implementation Considerations

Once you're happy with the user interface design, you must then decide how to implement the user interface. With Java, you have many choices.

If your application requires a highly interactive interface, you may want to develop a rich GUI application, rather than a web application. Deploying and maintaining full-fledged GUI applications is almost as easy as for a web application, thanks to Sun's Java Web Start. However, many Internet users still consider Java Web Start too high an entry barrier, so web applications definitely have a place.

Until recently, most Java web applications were implemented using JavaServer Pages (JSP) or one of the comparable open source frameworks, such as Apache Velocity. These technologies are ideal for web sites with dynamically generated content based on very limited user input.

However, for user interfaces with complex user interaction, template page technologies like these start to show their limitations. For instance, even the simple act of displaying a set of checkboxes with checkmarks to represent current selections requires extensive conditional logic in the template page itself, whether it's implemented with Java scriptlets, JSP Standard Tag Library (JSTL) actions and the Expression Language (EL), or the Velocity Template Language (VTL). Nonetheless, in many cases, JSP or similar technologies is the right choice.

The JavaServer Faces (JSF) specification, released in March 2004, is a better solution for complex web application interfaces. JSF defines a component-based web application development model, enabling vendors and open source projects to create sophisticated user-interface widgets that developers can then use to create easy-to-use web applications, with portability between tools and application servers. (Oracle is an active contributor to the JSF specification and continues to innovate by providing products based on the specification such as the early access version of the ADF Faces component suite. Furthermore, Oracle has promised comprehensive JSF support in an upcoming version of the Oracle JDeveloper IDE.)

With the JSF component model, all logic, such as the conditional code needed to check off checkboxes to represent current selections, is implemented by the component, not by code within the pages. The JSF event model encourages decoupling of components and actions, and lets you develop web application interfaces similar to the way that you develop a GUI application. The ADF Faces component suite and many other component suites, both open source and commercial, gives you most of the components you need, but if you can't find a component that is perfect for your application, you can implement your own by extending JSF classes and implementing JSF interfaces.

If you decide to use JSF, you should be aware of these not so obvious issues:

- **Command buttons or links.** JSF provides two standard components for submitting a form — a command button, and a command link. The command link uses JavaScript code to submit the form, so if you can't ensure that all your users will always have JavaScript enabled, you should stick to the command button rather than the link.
- **Using page fragments.** For a complex user interface, you should use separate files for individual parts and stitch them together by using one master file. This approach makes it easier to develop and maintain the application, and it also lets you reuse the same parts in multiple pages. For example, if you develop your JSF application using JSP, you can create the

master file using either dynamic `<jsp:include>` or `<c:import>` actions, or the static `<%@ include %>` directive, to get the content of the fragments. (I recommend that you use static includes wherever possible to avoid the additional requirements and issues associated with using dynamic includes.)

- **JSP or not.** Although JSP is the only technology for putting together a web application interface with JSF components described completely in the specification, it's not your only choice. The JSF API is flexible enough for you (or someone else) to use other technologies, such as plain XML files. JSP is familiar to many web application developers, but when used with JSF, it throws you a few curveballs. (For more information about the issues, see my *Improving JSF by Dumping JSP* article at <http://www.onjava.com/pub/a/onjava/2004/06/09/jsf.html>.)

Having briefly discussed some of things you need to consider when designing and implementing web application interfaces, I'll now drill down into one aspect of a web application interfaces when using JavaServer Faces, specifically, how to customize the standard error messages. Let's see how Generic Attributes and PhaseListener implementations can help you customize error messages generated by JSF standard converters and validators.

Adding Meaningful Field References

JSF defines a number of converters and validators that you can attach to components to validate the user input. They queue error messages when the input is invalid, and message components then display the messages to the user. For example, this JSP page creates an input component that requires a numeric value between 1 and 10, and two types of message components for detailed and summary messages:

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

<f:view>
  <h:messages layout="table" showDetail="true" showSummary="false" />
  <h:form>
    <h:panelGrid columns="3">
      <h:outputText value="Number of passengers:" />
      <h:inputText id="noOfPass" size="8" required="true">
        <f:convertNumber integerOnly="true" />
        <f:validateLongRange minimum="1" maximum="10"/>
      </h:inputText>
      <h:message for="noOfPass" showDetail="false" showSummary="true"
        style="color: red" />
    </h:panelGrid>
    <h:commandButton value="Submit" />
  </h:form>
</f:view>
```

Validation error messages are commonly displayed together at the top of the page, but this works only if the user can easily match each message with the corresponding invalid field. The standard JSF error messages provide a description of the problem only, without any information about the field to which the message refers, as in this message returned by Sun's JSF 1.0 Reference Implementation (RI) when the validator in the example above is given an out of range value:

```
Validation error: Specified attribute is not between the expected values of 1 and 10
```

To solve this problem, we need two things: a way to associate a user-friendly field reference with a component and a way to add this reference to the message text. Let's use a generic component attribute to specify the field reference for the component:

```
...
<f:view>
  <h:messages layout="table" showDetail="true" showSummary="false" />
```

```

<h:form>
  <h:panelGrid columns="3">
    <h:outputText value="Number of passengers:" />
    <h:inputText id="noOfPass" size="8" required="true">
      <f:convertNumber integerOnly="true" />
      <f:validateLongRange minimum="1" maximum="10" />
      <f:attribute name="fieldRef" value="Number of passengers" />
    </h:inputText>
    ...
  </h:panelGrid>
  <h:commandButton value="Submit" />
</h:form>
</f:view>

```

A generic attribute is a named custom value for a component that other pieces of code with access to the component can use. The `<f:attribute>` action element sets a generic attribute named `fieldRef` to a value that matches the label for the input field the end user sees.

Next we need something that can get the generic attribute and insert it into the message. The best fit for this task is a custom `PhaseListener`. An application can register one or more `PhaseListener` implementations:

```

<faces-config>
  <lifecycle>
    <phase-listener>
      com.mycompany.jsf.listeners.MessageListener1
    </phase-listener>
  </lifecycle>
</faces-config>

```

JSF invokes a `PhaseListener` before and after the request-processing lifecycle phase it's interested in. This `PhaseListener` implementation handles our message-customization needs:

```

package com.mycompany.jsf.listeners;

import java.util.Iterator;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

public class MessageListener1 implements PhaseListener {

    public PhaseId getPhaseId() {
        return PhaseId.RENDER_RESPONSE;
    }

    public void beforePhase(PhaseEvent e) {
        FacesContext fc = e.getFacesContext();
        UIViewRoot root = fc.getViewRoot();
    }
}

```

```

    Iterator i = fc.getClientIdsWithMessages();
    while (i.hasNext()) {
        String clientId = (String) i.next();
        UIComponent c = root.findComponent(clientId);
        String fieldRef =
            (String) c.getAttributes().get("fieldRef");
        if (fieldRef != null) {
            Iterator j = fc.getMessages(clientId);
            while (j.hasNext()) {
                FacesMessage fm = (FacesMessage) j.next();
                fm.setDetail(fieldRef + ": " + fm.getDetail());
            }
        }
    }
}

public void afterPhase(PhaseEvent e) {
}
}

```

The `MessageListener1` class returns `PhaseId.RENDER_RESPONSE` from the `getPhaseId()` method, so JSF invokes its `beforePhase()` method just before the response is rendered. The `beforePhase()` method first gets the client IDs for all components for which a message is queued. It then locates each such component with the help of the `findComponent()` method and gets the value of the `fieldRef` attribute assigned to the component in the JSP page. Finally, the method gets all messages for the component and adds to the beginning the field reference to the detailed message text. When JSF moves on to render the response, the `<h:messages>` component at the top of the JSP page renders the modified detailed messages, including the user-friendly field reference shown in Figure 1

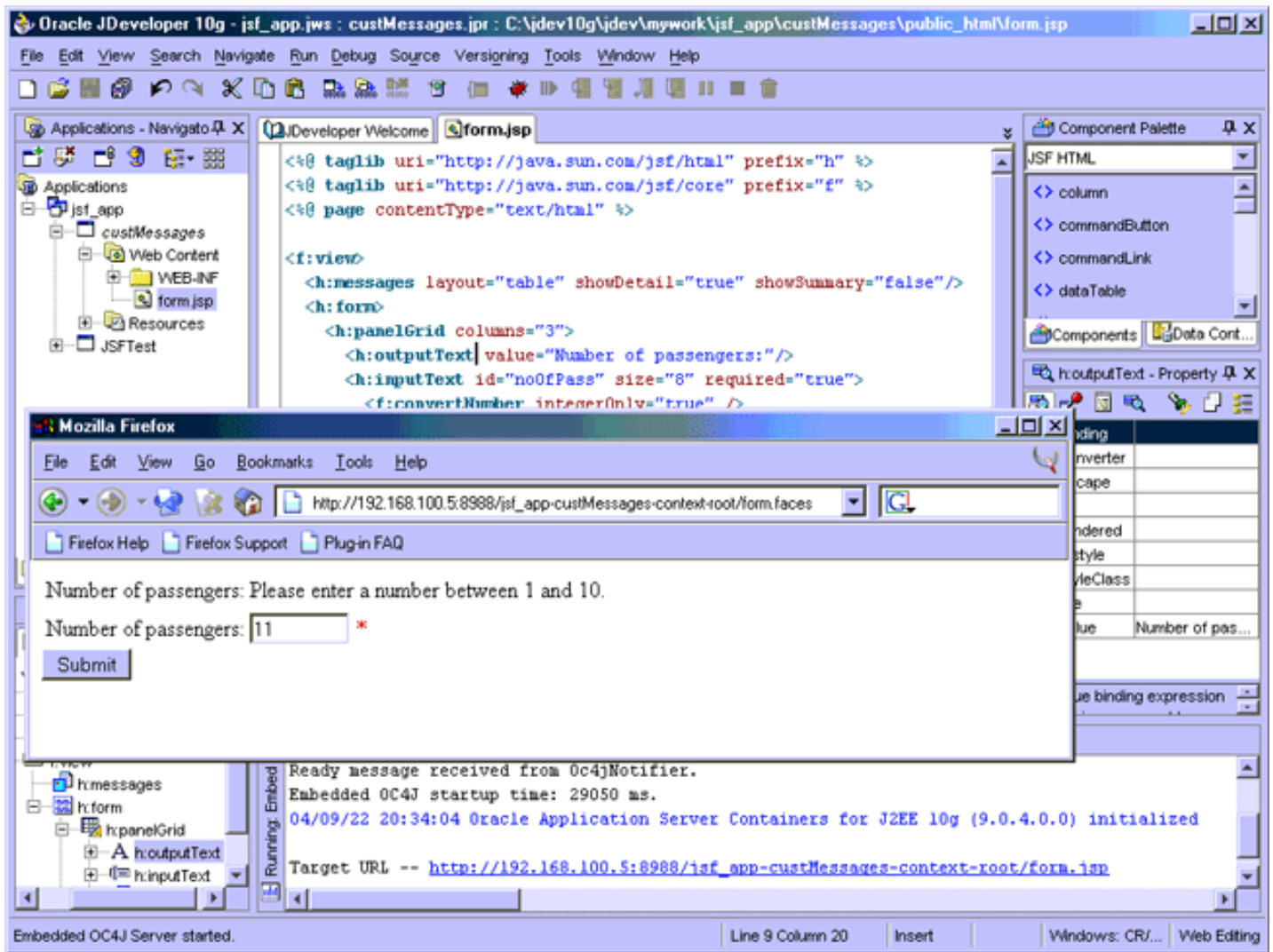


Figure 1: JSP Page being edited in Oracle JDeveloper 10g

Figure 1 shows the JSP page being edited in Oracle's JDeveloper 10g IDE, with a browser launched by asking JDeveloper to run the JSP page. You can of course use a plain text-editor to develop JSF applications, but JDeveloper 10g makes it a bit easier, with code completion for the tag library elements and other nice features.

Be aware, however, that JDeveloper 10g doesn't distinguish between an ordinary JSP page and a JSP page containing JSF components, so it launches the browser with a URL that doesn't match the URL pattern for the JSF servlet. You must manually change the URL in the browser to get it to work; in other words, you must change the ".jsp" extension to a ".faces" extension if the JSF servlet is mapped to "*.faces". (See "How To Use JSF with JDeveloper 10g" for more information)..

I've been a bit fancy in this example and have also placed an asterisk (*) next to the field with the invalid value, using a <h:message> component combined with a customized summary message that holds just the asterisk:

```

...
<f:view>
  <h:messages layout="table" showDetail="true" showSummary="false" />
  <h:form>
    <h:panelGrid columns="3">
      <h:outputText value="Number of passengers:" />
      <h:inputText id="noOfPass" size="8" required="true">
        ...
      </h:inputText>
      <h:message for="noOfPass" showDetail="false" showSummary="true"
        style="color: red" />
    </h:panelGrid>
  </h:form>
</f:view>

```

```

    </h:panelGrid>
    <h:commandButton value="Submit" />
</h:form>
</f:view>

```

A JSF message has both a summary and a detailed text, and you can customize both parts for all standard messages by overriding them in a resource bundle declared in the faces-config.xml file:

```

<faces-config>
  <application>
    <message-bundle>custMessages</message-bundle>
  </application>
</faces-config>

```

The resource bundle file, custMessages.properties in the WEB-INF/classes directory, looks like this, with an asterisk as the text for the "not in range" summary message and custom text for the detail message:

```

javax.faces.validator.NOT_IN_RANGE=*
javax.faces.validator.NOT_IN_RANGE_detail=Please enter a number between {0} and {1}.

```

The message keys for all standard messages are defined in the JSF specification.

If you use Oracle ADF Faces components rather than standard JSF components, you won't need to use a `<h:message>` component to add an asterisk next to the invalid field, since all ADF Faces input components automatically highlight invalid input by placing error icons in front of the fields and the summary message after the field. Nonetheless, the technique for adding user-friendly reference to error message text in this way can be useful, even for ADF Faces, because as with the standard messages, the ADF Faces messages lack a user-friendly reference although, on the other hand, the ADF `<af:messages>` component renders messages as links to the fields to which they refer, so users can click on a link to match a message to a field to avoid confusion.

Adding New Dynamic Elements to Standard Messages

Some JSF standard messages contain dynamic elements, such as the value that is out of range for the range validator message. However, the standard conversion error message doesn't contain any dynamic elements:

```

Conversion error occurred

```

Including the invalid value to the conversion error message makes it easier to understand, so let's do so with an extended version of the listener, along with some additional custom messages. We start with the messages in the resource bundle file:

```

javax.faces.component.UIInput.CONVERSION=*
javax.faces.component.UIInput.CONVERSION_detail=CONV_ERR_MSG
CUST_CONV_ERR_MSG_detail='{0}' is not a valid format for this field

```

These entries define a custom detail text (`CONV_ERR_MSG`) for the standard conversion error message that the listener can recognize, and a brand-new message for the custom conversion error message that contains a placeholder for the invalid value.

The interesting parts of the new version of the listener look like this:

```

...
public class MessageListener2 implements PhaseListener {
    ...
    public void beforePhase(PhaseEvent e) {
        FacesContext fc = e.getFacesContext();
        UIViewRoot root = fc.getViewRoot();
        String mbName = fc.getApplication().getMessageBundle();
    }
}

```

```

Locale locale = root.getLocale();
ResourceBundle rb = ResourceBundle.getBundle(mbName, locale);

Iterator i = fc.getClientIdsWithMessages();
while (i.hasNext()) {
    String clientId = (String) i.next();
    UIComponent c = root.findComponent(clientId);
    String fieldRef =
        (String) c.getAttributes().get("fieldRef");
    if (fieldRef != null) {
        Iterator j = fc.getMessages(clientId);
        while (j.hasNext()) {
            FacesMessage fm = (FacesMessage) j.next();
            String detail = fm.getDetail();
            if ("CONV_ERR_MSG".equals(detail)) {
                String custMsgPattern =
                    rb.getString("CUST_CONV_ERR_MSG_detail");
                Object[] params = new Object[1];
                params[0] =
                    ((EditableValueHolder) c).
                        getSubmittedValue();
                String custMsg =
                    MessageFormat.format(custMsgPattern,
                        params);
                fm.setDetail(custMsg);
            }
            fm.setDetail(fieldRef + ": " + fm.getDetail());
        }
    }
}
}
...
}

```

What's new is that the listener checks messages for the string we defined as the text for the standard conversion error message (CONV_ERR_MSG) and replaces it with the custom error message text, pulled from the application's resource bundle and formatted using the `MessageFormat` class to replace the placeholder with the invalid submitted value.

The EA7 version of ADF Faces doesn't allow you to customize the conversion error message, so you can't use this trick with the ADF Faces input components. On the other hand, the ADF Faces conversion message already includes the invalid value, so this particular customization isn't needed if you use ADF Faces.

Using Per-Component Messages

Although I generally recommend using generic messages that get dynamically populated with component-specific values at runtime, in some cases I want to use distinct messages for each component. You can accomplish this by combining the solutions we've looked at so far.

For example, say you want to specify different texts for the "value required" error message for each component. We can use a generic attribute for the message text, just as we did for the field reference earlier:

```

...
<f:view>
  <h:messages layout="table" showDetail="true" showSummary="false" />
  <h:form>
    <h:panelGrid columns="3">
      <h:outputText value="Number of passengers:" />

```

```

<h:inputText id="noOfPass" size="8" required="true">
  <f:convertNumber integerOnly="true" />
  <f:validateLongRange minimum="1" maximum="10"/>
  <f:attribute name="fieldRef" value="Number of passengers" />
  <f:attribute name="custMsg"
    value="Please enter the number of passengers" />
</h:inputText>
...
</h:panelGrid>
<h:commandButton value="Submit"/>
</h:form>
</f:view>

```

We can then use a recognizable string (VALUE_REQ_MSG) as the detail text for the standard "value required" error message:

```

javax.faces.component.UIInput.REQUIRED=*
javax.faces.component.UIInput.REQUIRED_detail=VALUE_REQ_MSG

```

Finally, we add another piece of logic to the listener:

```

...
public class MessageListener3 implements PhaseListener {
  ...
  public void beforePhase(PhaseEvent e) {
    FacesContext fc = e.getFacesContext();
    UIViewRoot root = fc.getViewRoot();
    String mbName = fc.getApplication().getMessageBundle();
    Locale locale = root.getLocale();
    ResourceBundle rb = ResourceBundle.getBundle(mbName, locale);

    Iterator i = fc.getClientIdsWithMessages();
    while (i.hasNext()) {
      String clientId = (String) i.next();
      UIComponent c = root.findComponent(clientId);
      String fieldRef =
        (String) c.getAttributes().get("fieldRef");
      if (fieldRef != null) {
        Iterator j = fc.getMessages(clientId);
        while (j.hasNext()) {
          FacesMessage fm = (FacesMessage) j.next();
          String detail = fm.getDetail();
          if ("CONV_ERR_MSG".equals(detail)) {
            ...
          }
          else if ("VALUE_REQ_MSG".equals(detail)) {
            String custMsg = (String)
              c.getAttributes().get("custMsg");
            fm.setDetail(custMsg);
          }
          fm.setDetail(fieldRef + ": " + fm.getDetail());
        }
      }
    }
  }
  ...
}

```

This version of the listener replaces the message text with the value from the component's `custMsg` generic attribute when it encounters a "value required" message.

As with the conversion error message, the EA7 version of ADF Faces doesn't allow you to customize the "value required" message, but you can use this technique for validation error messages even with ADF Faces components.

Summary

I am hopeful that some of the consideration points I outlined in this article will help you design and implement better web application interfaces. There are plenty of books, articles and online tutorials for learning about JSF. A good place to start is Sun's [JSF](#) site.

Developers using JSF often ask how to include human readable component labels, such as "First Name," in the error messages generated by standard JSF validators and converters. Although JSF 1.0/1.1 has no direct support for this, it can be done using using generic attributes and a `PhaseListener`, as I've shown in this article. The three different message-customization requirements described in this article should cover the most common needs, but the basic idea presented here can also easily be extended and tweaked to cover other similar issues you may encounter.

Next Steps

UI next steps

- 1) [Download](#) the source code of this article's example Web application
- 2) Get [Early Access Release of Oracle ADF Faces Components](#): a rich set of JSF components.
- 3) [Learn How To Use ADF Faces With JDeveloper 10g](#)

Additional Information:

[Making Faces](#)

JavaServer Faces (JSF) provides a powerful new framework for creating Web GUIs.

[From ADF UIX to JSF](#)

Oracle ADF Faces Components brings a library of reuse to JavaServer Faces.

[List of ADF Faces Components](#)

[Read the Oracle ADF Faces FAQ](#)

[Oracle JDeveloper and JSF](#)

[Roadmap for the ADF UIX technology and JavaServer Faces](#)

[More on JavaServer Faces Technology](#)

[Oracle ADF Development Guidelines](#)

the JavaServer Pages (JSP), JSP Standard Tag Library (JSTL), and Java Servlet specifications.

Copyright © 2005, Oracle. All Rights Reserved.