SAP NetWeaver J2EE Preview: User Interfaces with JSF

Applies to:
SAP NetWeaver J2EE Preview

Summary
Learn how to develop JSF-based front end.

Author(s): SAP NetWeaver Product Management
Company: SAP AG
Created on: April 2006
JSF Technology Overview

JavaServer Faces (JSF) is one of the most advanced Java technologies for the development of user interfaces. JSF can be used for creating any type of UI but is mainly aimed at Web programming. As its name suggests, this is a server-side technology, meaning that it is suitable for providing dynamically generated content.

Some of the key strengths of JSF are:

- **Standard-based**

  Being a standard creates a non-proprietary environment for controlled development of the framework. On the one hand, such an environment guarantees to application developers that their applications can be executed on multiple JSF implementations. On the other hand, it creates the necessary conditions for adding future improvements to the framework.

- **Extendable**

  JSF is designed to be highly extensible in various ways. Each JSF vendor has to supply a basic set of UI components but additional components can be easily developed in the cases when those components cannot satisfy the application requirements to the UI. Another example of JSF’s extensibility is the possibility of using multiple render kits. UI components can have different representations, such as HTML, WML, SVG or XUL. Non-markup representation is just another option.

- **Customizable**

  The JSF runtime is highly flexible. Runtime aspects such as lifecycle management, response generation mechanisms and state saving approaches can be easily tailored to the specifics of an application.

With regard to Web programming, the JSF technology integrates well with JSP, servlet, HTML, Java Script and other technologies. Web pages are designed with JSF tags plugged into JSP pages.

JSF is especially useful if you want to enable dynamic page reloading or user input validation. Of course, you can provide these features even if you are using solely JSP, servlet or any other technology. The advantage of JSF, however, is that it has built-in support for such tasks, simplifying the development process of Web applications. JSF has a specific request processing lifecycle, an extendable UI component model, and strong
event handling mechanism. These features allow the rapid application development (RAD) principles, which are typical to traditional UI development, to be applied for Java EE based applications.

JSF technology provides a perfect separation of components according to the Model, View and Controller (MVC) principle. Pure servlet and JSP technologies, for example, mix the HTML presentation (View) logic with the programming (Model and Controller) logic. This means that the person who designs the HTML layout of the application also has to be a programmer, or vice-versa. With JSF, the designer works with XML-like tags, most of which are equivalent to HTML tags and some of their attributes. The Java programmer writes the underlying Model and Controller logic in the form of Java beans. The programmer only provides a set of bean properties (variables in the source code), which the designer binds to in the designed pages.

This tutorial guides you through the basic aspects of developing Web UI using JSF. It uses as a basis the source code of the Java EE 5 Demo. Deploying the demo is described in the Deploying and Running the Samples guide.

### Developing Web UI with JSF

Switching from pure JSP and Servlet technology to JSF development is not difficult. Developing Web UI using the JSF technology involves two aspects:

- Writing the Web pages - You develop the JSF pages in standard JSP files. This task involves the standard description of the page layout, which can be handled by a Web designer, for example.

  For more information about writing the pages, see Standard JSF Tags and faces-config.xml.

- Developing the underlying logic - This involves developing Java classes called backing beans, which will provide the underlying logic of the Web pages. This can be handled by a Java developer.

  For more information about developing the underlying logic, see Backing Beans.

The rest of the topics in this guide describe in parallel both aspects of creating the corresponding JSF component.

#### faces-config.xml

Each JSF application needs a faces-config.xml configuration file. It describes the application properties, such as navigation rules between the JSF pages, default bean instances, default values of some variables, message bundles, and so on.

For example, here are some extracts of the faces-config.xml of our Java EE 5 Demo:

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

<faces-config>
  <application>
    <message-bundle>com.sap.engine.examples.EDMAppMessages</message-bundle>
    <locale-config>
      <default-locale>en</default-locale>
    </locale-config>
    <navigation-rule>
      <!-- Navigation rules between JSF pages -->
    </navigation-rule>
  </application>
</faces-config>
```
The `<application>` part describes configuration information for the entire application. In our case, we placed the description of the message bundle implementation and the locales information.

The `<navigation-rule>`s specify under what conditions one page will lead to another. The values of the `<from-outcome>` tags are associated with the actions of Command Buttons.
The `<managed-bean>` sections describe the names of the default instances of the **Backing Beans**. They will be created initially by JSF, and can be used by all JSF pages. For example, this `faces-config.xml` describes that an instance of class `com.sap.engine.examples.backingbeans.DepartmentListBeans` will be created, and will have the name `departmentList`.

### Standard JSF Tags

There are 43 standard JSF tags, which can be divided into two basic types:

- **Equivalents to HTML tags** - Represent the most essential HTML tags, such as `<table>`, `<form>`, input elements, submit and reset buttons, links, and so on.

  JSF tags have a limited set of attributes corresponding to some of the HTML tag attributes. We recommend that you use stylesheets instead of hardcoded formatting.

  JSF provides an additional tag for dynamic data, which is not available in pure HTML, namely the `dataTable` tag. It actually constructs a standard `<table>` component where the row number is not fixed. This is useful, for example, if you want to display information about employees taken from a database, and you cannot know the number of employees in advance.

- **Core JSF tags** - Provide core UI actions, such as validation, conversion, displaying error messages and handling events.

A typical Web page with JSF tags looks like this:

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

<f:view>

<h:form>

</h:form>

</f:view>
```

The directive `<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>` specifies that HTML-equivalent tags will have the "h" prefix in the source code - for example, `<h:form>` or `<h:panelGrid>`. You can specify any prefix instead of "h" and use it in your source code. In this case, your tags will look like this: `<myprefix:form>`, `<myprefix:commandButton>`, and so on.

The directive `<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>` describes the prefix for code tags. In this case, it is "f" - for example, `<f:view>` or `<f:facet>`. You can also use any other prefix instead of "f".

### Backing Beans

Backing beans provide the underlying logic of JSF applications - for example, when you want to display information about customers, the backing beans obtain and process the database data.

The user interface is bound to the backing bean(s) via the **bean properties**. A bean property is a normal variable in the bean source code with a setter and/or getter method. If a variable is to be read-only from the UI, create a getter method. If it is to be write only, create a setter method.

A backing bean is a normal Java class. It does not need to implement specific interfaces or extend specific classes.

© 2006 SAP AG
For example, it may look like this:

```java
public class MyBackingBean {
    private String propA;
    private String propB;
    ...
    public getPropA() {
        return propA;
    }
    public setPropA(String _propA) {
        this.propA = _propA;
    }
    public getPropB() {
        return propB;
    }
    public setPropB(String _propB) {
        this.propB = _propB;
    }
    ...
}
```

In the JSF source code, the binding to the propA and propB properties can look like this:

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

<f:view>
    <h:form>
        <h:panelGrid columns="1">
            <h:outputText value="Submit Data"/>
            <h:outputText value="Property A:"/>
            <h:outputText value="#{myBean.propA}"/>
            <h:outputText value="Property B:"/>
            <h:inputText value="#{myBean.propB}"/>
            <h:commandButton value="Set Properties"
            action="set_props"/>
        </h:panelGrid>
    </h:form>
</f:view>
```

These code samples would create a simple HTML form where the user can submit data that will be set as values of the propA and propB properties.

For example, in the Java EE 5 Demo the com.sap.engine.examples.backingbeans.DepartmentListBean enables access to all departments available in the company database. It provides two properties to the JSF user interface:

- allDepartments - represents the collective property of all departments
- selectedDepartments - represents the collective property of the currently selected departments
Because these properties are read-only, they only have getter methods.

```java
package com.sap.engine.examples.backingbeans;
import java.util.ArrayList;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.sap.engine.examples.ejb3.edm.beans.entity.Department;
import com.sap.examples.hrservices.HRServices;

public class DepartmentListBean {

    private HRServices hrServices;

    private List<DepartmentBean> allDepartments;
    private List<DepartmentBean> selectedDepartments =
        new ArrayList<DepartmentBean>();

    public DepartmentListBean() {
        try {
            InitialContext ctx = new InitialContext();
            hrServices = (HRServices) ctx.lookup(HRServices.class.getName());
        } catch (NamingException e) {
            throw new RuntimeException(e);
        }
    }

    public List<DepartmentBean> getAllDepartments() {
        if (allDepartments == null) {
            readDepartments();
        }
        return allDepartments;
    }

    public List<DepartmentBean> getSelectedDepartments() {
        return selectedDepartments;
    }

    . . .
}
```

**Dynamic Data Tables**

The most convenient way to provide dynamic data is in the form of data tables. Data tables are a way of displaying data when we do not know the exact number of elements in a list. For example, if we want to display employee details, we cannot know in advance how many employees will be available in the database. We cannot use the standard panelGrid tag because it creates a table with a fixed number of rows. Instead, we can use a data table (dataTable).

To create a data table, you must supply a "collective" bean property, that is a property of one of the following types:
Array
• An instance of java.util.List
• An instance of java.sql.ResultSet
• An instance of javax.servlet.jsp.jstl.sql.ResultSet
• An instance of javax.faces.model.DataModel

This collective property is the property over which the data table will iterate. The contents of the data table usually visualize the properties of each single instance in the collective property.

The format of a data table is as follows:

```
<h:dataTable value="#{myBean.myCollectiveProp}" var="mySingleProp">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Property One"/>
    </f:facet>
    <h:outputText value="#{mySingleProp.prop1}"/>
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Property Two"/>
    </f:facet>
    <h:outputText value="#{mySingleProp.prop2}"/>
  </h:column>
</h:dataTable>
```

In the above data table, the table iterates over the collective property myCollectiveProp, which consists of single instances named mySingleProp. Accordingly, the two columns of the table display the two properties of each individual instance.

Note: In data tables, elements are described according to the column they belong to, unlike with HTML tables, where components are described according to the row they belong to.

The Java EE 5 Demo uses data tables extensively. For example, to display the available departments, their employees, the available projects and their associated employees, we use data tables.

We used one such data table for the Department Listing page:
## Department Listing

<table>
<thead>
<tr>
<th>Department ID</th>
<th>Department Name</th>
<th>Number of Employees</th>
<th>Department Manager</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPDEV</td>
<td>Application Development</td>
<td>11</td>
<td>Julie Armstrong</td>
<td></td>
</tr>
<tr>
<td>HR</td>
<td>Human Resources</td>
<td>8</td>
<td>Bob Smith</td>
<td></td>
</tr>
<tr>
<td>QM</td>
<td>Quality Management</td>
<td>8</td>
<td>Allisa Glenn</td>
<td></td>
</tr>
<tr>
<td>SALESOPO</td>
<td>Sales and Operations</td>
<td>9</td>
<td>Elena Lopez</td>
<td></td>
</tr>
<tr>
<td>TECHSUPPO</td>
<td>Technical Support</td>
<td>8</td>
<td>Tomoko Akino</td>
<td></td>
</tr>
</tbody>
</table>

The JSP source code for this table is:

```jsp
<h: dataTable value="#{departmentList.allDepartments}" var="dept" styleClass="ftable" columnClasses="tabletxt" headerClass="ftabletxt" width="100%">

<h: column>
    <f: facet name="header">
        <h: panelGroup>
            <f:verbatim>
                <link href="../styles/ejb3demo_styles.css" rel="stylesheet" type="text/css"/>
            </f:verbatim>
            <h: outputText value="Department ID"/>
        </h: panelGroup>
    </f: facet>
    <h: outputText value="#{dept.department.departmentId}"/>
</h: column>

<h: column>
    <f: facet name="header">
        <h: outputText value="Department Name"/>
    </f: facet>
    <h: outputText value="#{dept.department.name}"/>
</h: column>

<h: column>
    <f: facet name="header">
        <h: outputText value="Number of Employees"/>
    </f: facet>
    <h: outputText value="#{dept.numberOfEmployees}"/>
</h: column>

<h: column>
    <f: facet name="header">
        <h: outputText value="Department Manager"/>
    </f: facet>
    <h: outputText value="#{dept.nameOfManager}"/>
</h: column>

<h: column>
    <f: facet name="header">
        <h: outputText value="Select"/>
    </f: facet>
    <h: selectBooleanCheckbox value="#{dept.selected}"/>
</h: column>
```

© 2006 SAP AG 9
In some cases, it was necessary to use data tables inside other data tables. This is the case with the Department Details page (see the figure below): All selected departments were listed in a root data table, and the details for each department were listed in an internal data table.

<table>
<thead>
<tr>
<th>Department Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department Name</td>
</tr>
<tr>
<td>Department Manager</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>45</td>
</tr>
<tr>
<td>46</td>
</tr>
<tr>
<td>47</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>51</td>
</tr>
<tr>
<td>52</td>
</tr>
<tr>
<td>53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department Name</td>
</tr>
<tr>
<td>Department Manager</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>81</td>
</tr>
<tr>
<td>82</td>
</tr>
<tr>
<td>83</td>
</tr>
<tr>
<td>84</td>
</tr>
<tr>
<td>85</td>
</tr>
<tr>
<td>86</td>
</tr>
<tr>
<td>87</td>
</tr>
<tr>
<td>88</td>
</tr>
</tbody>
</table>

Set Employee Data
Here is the more sophisticated data table source code:

```html
<h:form id="department">
<h:dataTable value="#{departmentList.selectedDepartments}" var="dept" cellspacing="3" width="100%">
    <h:column>
        <h:panelGrid width="100%" columns="1" cellspacing="2">
            <h:panelGrid columns="2" width="100%" styleClass="ftable" cellspacing="1" headerClass="ftabletxt">
                <f:facet name="header">
                    <h:panelGroup>
                        <f:verbatim>Department Details</f:verbatim>
                    </h:panelGroup>
                </f:facet>
            </h:panelGrid>
            <h:column>
                <h:outputText value="Department Name" styleClass="tabletxt"/>
            </h:column>
            <h:column>
                <h:outputText value="#{dept.department.name}" styleClass="tabletxt"/>
            </h:column>
            <h:column>
                <h:outputText value="Department Manager" styleClass="tabletxt"/>
            </h:column>
            <h:column>
                <h:outputText value="#{dept.nameOfManager}" styleClass="tabletxt"/>
            </h:column>
        </h:panelGrid>
        <h: dataTable value="#{dept.department.employees}" var="emp" styleClass="ftable" rowClasses="sht1, tabletxt" width="100%" border="0" headerClass="ftabletxt">
            <f:facet name="header">
                <h:panelGroup>
                    <f:verbatim>Employees</f:verbatim>
                </h:panelGroup>
            </f:facet>
            <h:column>
                <f:facet name="header"><h:outputText value="ID"/></f:facet>
                <h:outputText value="#{emp.employeeId}"/>
            </h:column>
            <h:column>
                <f:facet name="header"><h:outputText value="First Name"/></f:facet>
                <h:outputText value="#{emp.firstName}"/>
            </h:column>
            <h:column>
                <f:facet name="header"><h:outputText value="Last Name"/></f:facet>
                <h:inputText id="LastName" value="#{emp.lastName}" styleClass="inputt" required="true">
                    <f:validateLength maximum="30"/>
                </h:inputText>
            </h:column>
        </h:dataTable>
    </h:column>
</h:form>
```
There are two data tables in this example. The first one iterates over the selected departments from the previous view and contains the second table, which iterates over the employees available in the selected department.

**Command Buttons**

Command buttons are the JSF equivalent to HTML form buttons (submit and reset buttons). A typical command button looks like this:

```html
<h:commandButton value="Press Me" type="submit" action="Some_Action"/>
```

The action attribute specifies the behavior of the page when the button is pressed. There can be two types of action values:

- **String** - A String action specifies that the button redirects to another page, as described in the `faces-config.xml`. This behavior is straightforward, as it does not involve pre-processing of data before loading the next page. In our application, for example, such a button is the command button in the Display Project Details page:
According to the description in the faces-config.xml, the OK outcome that originates from the prjDetails.jsp view, leads to the prjList.jsp view:

```xml
<navigation-rule>
  <from-view-id>/prjDetails.jsp</from-view-id>
  <navigation-case>
    <from-outcome>OK</from-outcome>
    <to-view-id>/prjList.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```
• Binding to a backing bean method that returns a String - When this button is pressed, the specified bean method is called. The method can involve some data processing and the method outcome determines which page is then loaded.

In our application, for example, the "View Details" button on the "Department Listing" page should lead to the "Department Details" page of the selected departments. Hence, before loading the next page, JSF must receive all selected departments. This is done in the showSelectedDepartments() method of the DepartmentListBean. The View Details button has the following syntax:

```xml
<h:commandButton value="View Details"
    styleClass="form_buttons"
    action="#{departmentList.showSelectedDepartments}"/>
```

The source code of the showSelectedDepartments() method is:

```java
package com.sap.engine.examples.backingbeans;

import com.sap.engine.examples.ejb3.edm.beans.entity.Department;

public class DepartmentListBean {

    private List<DepartmentBean> selectedDepartments = new ArrayList<DepartmentBean>();

    public String showSelectedDepartments() {
        selectedDepartments.clear();
        for (DepartmentBean departmentBean : allDepartments) {
            if (departmentBean.isSelected()) {
                selectedDepartments.add(departmentBean);
            }
        }
        if (selectedDepartments.size() > 0){
            return "departments_selected";
        }
        return "";
    }
}
```

This method determines which departments were selected on the "Department Listing" page. In the best case, the method returns "departments_selected", which, according to the faces-config.xml, leads to the deptDetails.jsp ("Department Details" page):

```xml
<navigation-rule>
    <from-view-id>/deptList.jsp</from-view-id>
    <to-view-id>/deptDetails.jsp</to-view-id>
</navigation-rule>
</navigation-case>
```
Note: Unlike with ordinary HTML buttons, where you provide the form data as parameters of the HTTP request, the JSF buttons send the form data as new values of the specified bean properties.

User Input Validation

In JSF, you can easily implement checking the correctness (validating) of user input in Web forms. This is necessary because incorrect data can mess up your underlying business model. If you validate the user input before submitting the data, you can prevent any incorrect data entering your model.

Basically, your validation can affect the formatting, content and length of any user input. Usually, this applies to dates, numbers, currencies, and so on. You can use the default validators available in JSF or create custom ones.

In our Java EE 5 Demo, we have added validators for the Start Date and End Date fields in the Create Project page. If you do not supply correct dates, an error message is displayed:

- Your input '12-10-2006' is not a valid date format. Specify the date as MM/DD/YY.
- Your input '12-10-2007' is not a valid date format. Specify the date as MM/DD/YY.

If you do not supply start or end date, another error message appears:
The validators for the **Start Date** and **End Date** fields are described with the following source code in the `prjCreate.jsp`:

```html
    <h:outputText value="Start Date:" styleClass="ttw"/>
    <h:inputText id="startDate"
        value="#{projectList.newProject.startDate}"
        styleClass="inputc" required="true">
        <f:convertDateTime type="date" dateStyle="short"/>
    </h:inputText>

    <h:outputText value="End Date:" styleClass="ttw"/>
    <h:inputText id="endDate"
        value="#{projectList.newProject.endDate}"
        styleClass="inputc" required="true">
        <f:convertDateTime type="date" dateStyle="short"/>
    </h:inputText>

    <h:messages styleClass="error_msg"/>
```

This code extract states that the two fields are required, the date and time converter is used as a validator for each of them, and an error message is displayed if you try to supply invalid data or omit one of the fields.

The **Project Name** and **Project Description** fields are also validated according to their length. They are described in the following way:

```html
    <h:outputText value="Project Name:" styleClass="ttw"/>
    <h:inputText id="prjName"
        value="#{projectList.newProject.project.title}"
        styleClass="inputc" required="true">
        <f:validateLength maximum="30"/>
    </h:inputText>

    <h:outputText value="Project Description:" styleClass="ttw"/>
    <h:inputTextarea id="prjDescription"
        value="#{projectList.newProject.project.description}"
        styleClass="inputc" required="true">
        <f:validateLength minimum="0" maximum="1000"/>
    </h:inputTextarea>

    <h:messages styleClass="error_msg"/>
```

- The 'create:StartDate' input field cannot be empty. Please, enter a value in the appropriate format.
- Your input '12-10-2007' is not a valid date format. Specify the date as MM/DD/YY.
This extract indicates that the two fields are required and the length validator is used for both of them. An error message is displayed if you try to supply a longer text for them or if you leave them blank.

Adding custom validators is not described in this tutorial.