Introducing Java Server Faces (JSF) to 4GL Developers

An Oracle White Paper
November 2006

Introducing Java Server Faces (JSF) to 4GL Developers

INTRODUCTION

IBM, BEA, Sun and others have also committed to using JSF as their UI technology.

The past couple of years have seen the emergence of JSF (Java Server Faces) as the technology of choice for building the user interface (UI) layer of Java web applications. Oracle's own business applications are committing to JSF as the UI technology in the "Fusion" stack.

The component based architecture gives business developers simple, yet powerful, UI components with which to build applications; while being based on standards means that the larger development community can collaborate to build richer and more advanced components.

This paper will introduce JSF to 4GL developers, showing how the components can be used, and behind the scenes, how the components work.

INTRODUCING JSF

The Java Community Process (JCP), which is responsible for defining the standard for JSF – initially released as JSR-127 (Java Specification Request 127)- states one of its goals is to: "… *establish a standard API for creating Java Web application GUIs, which will eliminate the burden of creating and maintaining GUI infrastructure for developers*."

Which means that JSF is about components in the traditional UI sense of widgets like buttons, text fields and check boxes. It is not slaved to the traditional mark-up centric view of previous web technologies: a button is a button, not a tag. It has behavior, you can listen for it being pressed and the infrastructure to support what happens when it is pressed is taken care for you. Which is exactly what you have been used to in the Oracle Forms, Visual Basic and PeopleTools world.

A Brief History Lesson

From its formative years in 2001, the first release in 2003, up to the current release (JSF 1.2 JSR-252) in 2006 the JCP has brought together resources from the community, including Oracle, to define the specification and produce a reference implementation of the specification.

Before JSF

It is difficult to see how far JSF has raised the bar for Java web application UIs without first being aware of the development experience (or lack of) that was the catalyst for the simpler UI component based view that JSF typifies.

Static HTML

In the formative years of the web, static HTML (Hyper Text Markup Language) pages made up the bulk of any content you would see on the web. The overriding limitation was that as static pages it became difficult to display any sort of dynamic data like a list of products held in a database. Furthermore, if you wanted to do more than simply serve up a series of static pages, you also had to code your own communication to and from the server using the HTTP Post and Get messages as the carrier.

Servlets

Strictly speaking servlets do not have to generate HTML; they can generate any markup language dependant on the end user device: e.g. a mobile phone or PDA.

To overcome the limitations of static pages, servlets were introduced. Simply a Java program that would do some processing and output a string of HTML tags as its result. Thus, the development of a web page required the developer to code a program that would output the tags to be rendered by the browser. Again, developers would also still have to manage the Post and Get communication highlighted above.

Java Server Pages

In order to move away from this programmatic view of web page development, the next step in the evolution brought us to Java Server Pages (JSP). These were "special" tags that could be embedded into a web page. These tags were used by the JSP container to generate servlets, that in turn generated the markup. So, you had the concepts of developers who could develop JSP tags, and possibly a different set of developers who could use the tags via JSTL (JSP Standard Tag Libraries).

However, this still represented a "tag" based view of UI development and still required the UI developer to understand the intricacies of HTTP protocols and to manage state.

Thus, JSF finally brings the development of web UIs to the stage where developers can work with visual controls that support an event model and allow visual editing through IDEs such as Oracle JDeveloper.

JSF In the Community

So, being a ratified standard, anyone can develop their own set of JSF components for use, and anyone could implement their own JSF runtime on which those components run. And that is what we are seeing in the wider community.

Oracle's contribution

Oracle is one of the leading expert groups involved in the JSF specification, contributing their experiences from their own web framework used by the Oracle Applications developers: UIX (User Interface XML). As a result of the work on the specification, the library of UIX components was redeveloped as standard JSF components. This set of components is called ADF Faces and can be used on any runtime implementation of JSF.

Being committed to the open source community, Oracle has also donated their ADF Faces implementation to the Apache community, and within this community this offering is called Trinidad.

Contribution from others

Sun, who are also involved in the JCP, provide a reference implementation for the JSF framework (the environment on which the JSF components run) and that now appears in the Java Enterprise Edition (EE) 5.

MyFaces, from the Apache community, provide both a 1.1 reference implementation as well as a component library. Other offerings include Glassfish, which is an open source implementation of Java EE 5 application server and contains a JSF 1.2 implementation.

The bottom line is components are standardized within JSF and you can use any component from any vendor and mix and match them as required.

UNDERSTANDING JSF

For the remainder of this paper we will study and use the ADF Faces set of components; although the general points could apply to any JSF implementation.

So, now with an understanding of the need for a component based solution, the next step is to look at these components and what features they offer.

Introducing the component

Figure 1 shows a small sample of some of the ADF Faces components. Each web page in your application will be constructed from components. Components will typically be nested inside other components (called layout components or containers) to provide the required layout.

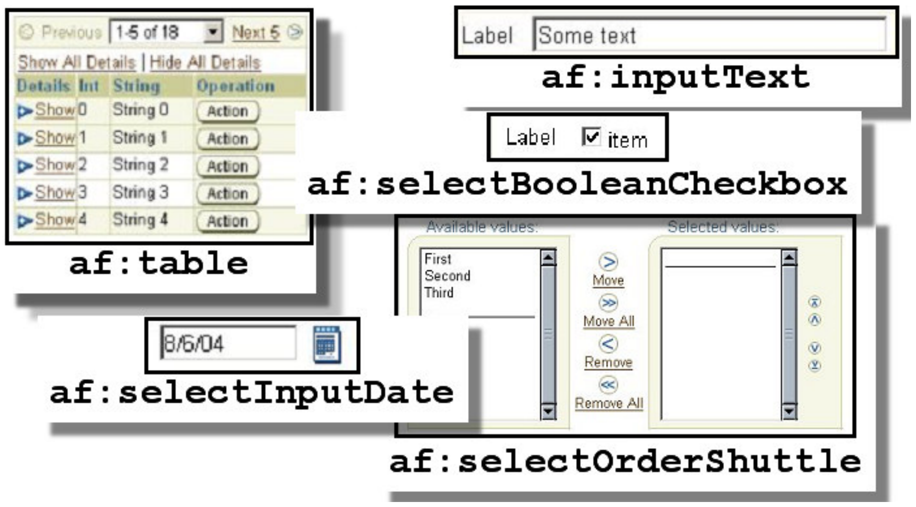


Figure 1 – ADF Faces components

The component architecture

Each component is typified by three parts: attributes, behavior and renderers.

Attributes

Each component has a number of attributes associated with the component. For example, a button will have an attribute *text* that defines the text label that appears on the button; or a *background-color* attribute to define the background color.

Behavior

The component defines a number of actions that control the behavior of that component. For example, a button provides a *valueChange* event on which a developer associates code.

Renderers

The component itself has attributes and behavior as indicated above but that says nothing about how the component will actually look. That is the responsibility of the renderer. It is the job of the renderer to realize the component as defined as a concrete screen instruction – e.g. in the case of a browser application – a fragment of HTML mark-up.

JSF UI components are not limited to generating markup. For example UI Components can also render binary data such as an image streams or different document types such as SVG, PDF and Word.

Having separate (or pluggable) rendering is one of the most compelling features of JSF. The fact that the component behavior is de-coupled from the way it is displayed means that the same UI component can be displayed differently – for example, depending on the device to which it is rendered.

Supporting the component

However, the component does not live in isolation. The JSF runtime provides a number of features to support the components.

Managed beans

The managed bean is simply a Java object that provides integration between the UI component and the data object the component is displaying. You can define the scope of these objects by indicating that they exist for the lifetime of the application, when only requesting a web page, or a session covering a number of web pages.

Expression Language

Expression Language (EL) is a scripting language that is part of the JSF runtime and provides simplified expressions for accessing the data, managed beans and the various properties of the components. EL is a very powerful feature of JSF. For example, a component attribute (such as *required*) can be set by an EL expression depending on the value of data in a backing bean:

```
#{emp.MaritalStatus =='single'}
```

So if the value of MaritalStatus is "single" (and so the expression evaluates as true) then the *required* attribute will be set to true.

Oracle ADF uses EL as the language to bind the UI components to the underlying data.

Navigation case

The JSF framework allows developers to define a set of rules that define the navigation between pages. By default, this is defined in a file called *faces-config.xml* and is covered later in this paper.

Lifecycle

The JSF Request Processing Lifecycle (more commonly referred to as JSF lifecycle) is, as the name suggests, the various stages of processing in the life of a specific JSF request. At certain points in the JSF lifecycle specific actions will happen; whether it is the component retrieving its current state, or rendering the component.

A simple component example

So, in order to understand how each of these "parts" works as part of the whole, lets look at a simple case of a JSF text field component on a web page.

For simplicity, some tags have been omitted.

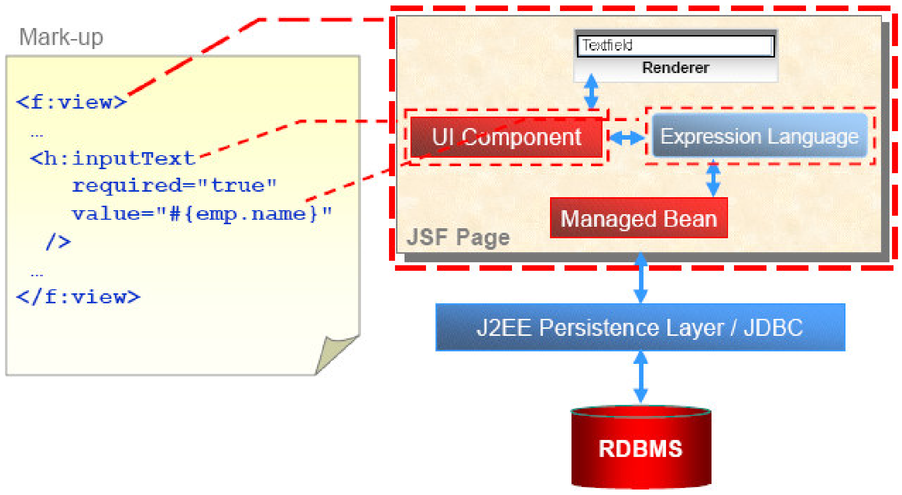


Figure 2 – Simple example

If you change from the design view of the page to the code view in the JDeveloper visual editor, you can see the source JDeveloper has created for you. There are two tags in the source code:

- <f:view>
- <h:inputText>

The first is the top-level container in which all the UI components will appear. JSF holds the definition of the page in a component tree (which can be manipulated at runtime) and <*f:view*> encapsulates this components tree.

<*h:inputText*> is a UI component in this tree and represents the input text field.

Today the implementations of JSF happens to use JSP as a carrier technology. That is, the page definitions are JSP pages and the JSF components are expressed as JSP tag libraries. This however is a convenience, it means that conventional HTML / JSP editors and IDEs can be used to design JSF pages. JSF itself makes no assumptions about the format of the page definition.

Hiding the implementation details

Note that the input field has a property *required* which is set to true, and a property *value*, which is being resolved by an EL expression via a managed bean. However, there is absolutely no detail of how the component implements the properties or in fact, how the component actually renders itself.

So this component could render itself using HTML markup up if displaying in a browser, WML (Wireless Markup Language) if displaying on a mobile device, or possibly even some hybrid of markup and JavaScript (refer to Ajax, covered later).

And that is the beauty of JSF to the business developer, the implementation detail of the UI control is hidden. The component simply exposes functionality and features (as a UI component would in Visual Basic or Oracle Forms).

The event model

The next step is to consider how the UI component initiates and reacts to UI actions or events such as pressing a button or selecting a value from a list.

JSF provides an event model that supports three types of events.

Action events

An action event in analogous to a When-Button-Pressed trigger in Oracle Forms.

An action event occurs when a JSF UI component, like a button or link, is clicked.

Figure 3 shows a simple JSF page consisting of an input field, a button and an output field, which will display the data from the input field concatenate with the string "hello".

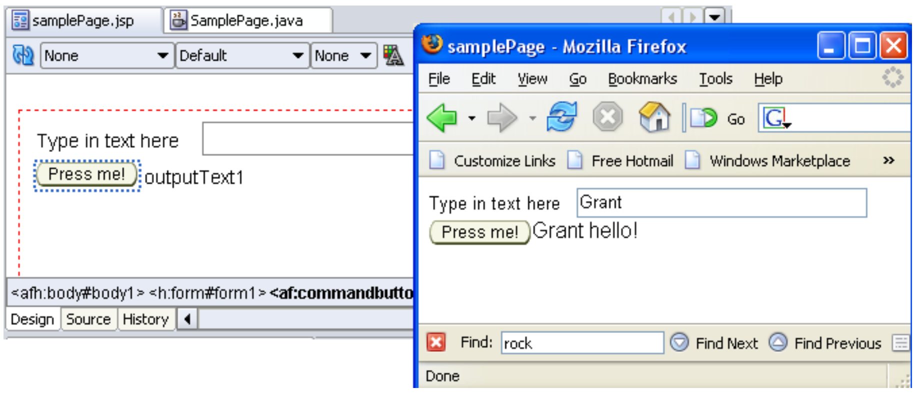


Figure 3 – Simple event example

By double-clicking on the button, JDeveloper will automatically create a placeholder for your code. JDeveloper will also create a backing bean for you and accessor methods for getting and setting the value of the UI components.

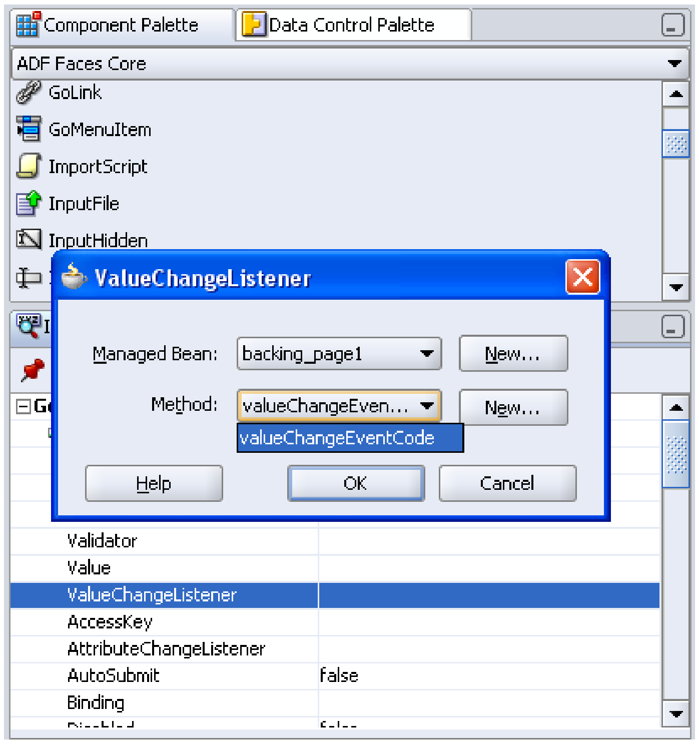
JDeveloper allows you to add a method to the *Action* attribute, which will execute the following code (held in a managed bean)

```
public String buttonPressed_action() {

    // Add event code here...

    String inputText = (String)getInputText1().getValue();

    getOutputText1().setValue(inputText + " hello!");;

    return null;

}
```

Note that this method returns a String, in this case null. The importance of this string is discussed later in the navigation section

Value change events

Input UI components such as input text fields have two events that are triggered when changing data. A *ValueChangeListener* fires when a value in the field has changed and *Validator* fires to validate the data entry. Again, JDeveloper provides a simple dialog to define the name of the method, the managed bean in which the method is placed and will generate the correct method signature (Figure 4)



Managed Bean: backing_page1

Method: valueChangeEven...

valueChangeEventCode

AutoSubmit false

Figure 4 – Creating a valueChangeListener

Phase events

Phase events execute at specific instances in the JSF lifecycle, allowing you to add your own code into the JSF lifecycle at points such as validation phase, or updating the model with values phase.

The significant point for any of these events is not what you can code, but instead what you DON'T have to code.

- There is no need to process the HTTP request to check what event is raised.
- There is no need to process the HTTP request to find out what value was typed into the input text field
- There is no processing the HTTP response to set the value of the output field. This is done be setting an attribute on the output text field.

The JSF framework provides a layer of abstraction to deal with the "plumbing" of associating an action with a piece of code and allowing to code to access attributes of the components.

The page flow

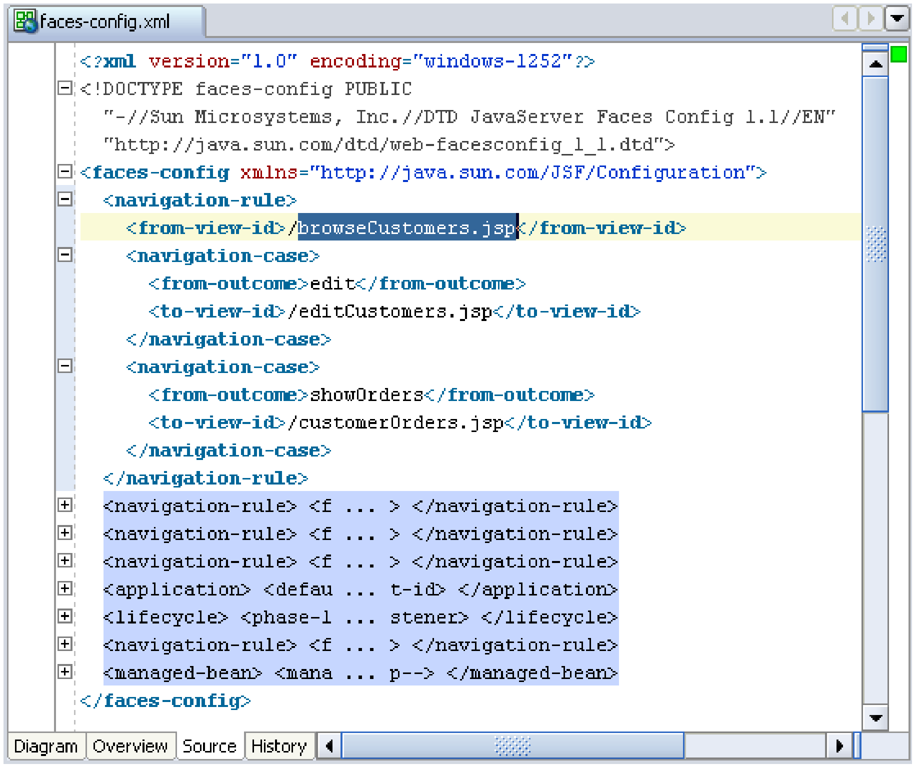
As well as defining a framework for components on a page, JSF also defines a framework for managing the flow between application pages. So, performing a particular action on a page results in navigation to another web page.

Introducing faces-config.xml

Note that the faces-config.xml file contains more information than just the navigation rules. As an introduction to JSF, your primary involvement with this file will be defining navigation but it also used to specify managed beans, render kits, converters and validators.

As you would expect, JSF handles the flow of your application through the web pages with minimum of coding on the developer's part. Instead, an XML file defines what action on a specific page results in navigation to another specific page. This XML file is called *faces-config.xml.*

Figure 5 shows an example faces-config.xml file with a navigation case on the *browserCustomers* page. An *edit* action would result in navigation to the *editCustomers* page, while a *showOrders* action would result in navigation to the *customerOrders* web page.



```
<?xml version="1.0" encoding="windows-1252"?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">
  <navigation-rule>
    <from-view-id>/browseCustomers.jsp</from-view-id>
    <navigation-case>
      <from-outcome>edit</from-outcome>
      <to-view-id>/editCustomers.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>showOrders</from-outcome>
      <to-view-id>/customerOrders.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  <navigation-rule> <f ... > </navigation-rule>
  <navigation-rule> <f ... > </navigation-rule>
  <navigation-rule> <f ... > </navigation-rule>
  <application> <defau ... t-id> </application>
  <lifecycle> <phase-l ... stener> </lifecycle>
  <navigation-rule> <f ... > </navigation-rule>
  <managed-bean> <mana ... p--> </managed-bean>
</faces-config>
```

Figure 5 – Faces-config.xml

However, JDeveloper makes management of this file even easier by providing a visual editor for defining and visualizing the page navigation rules.

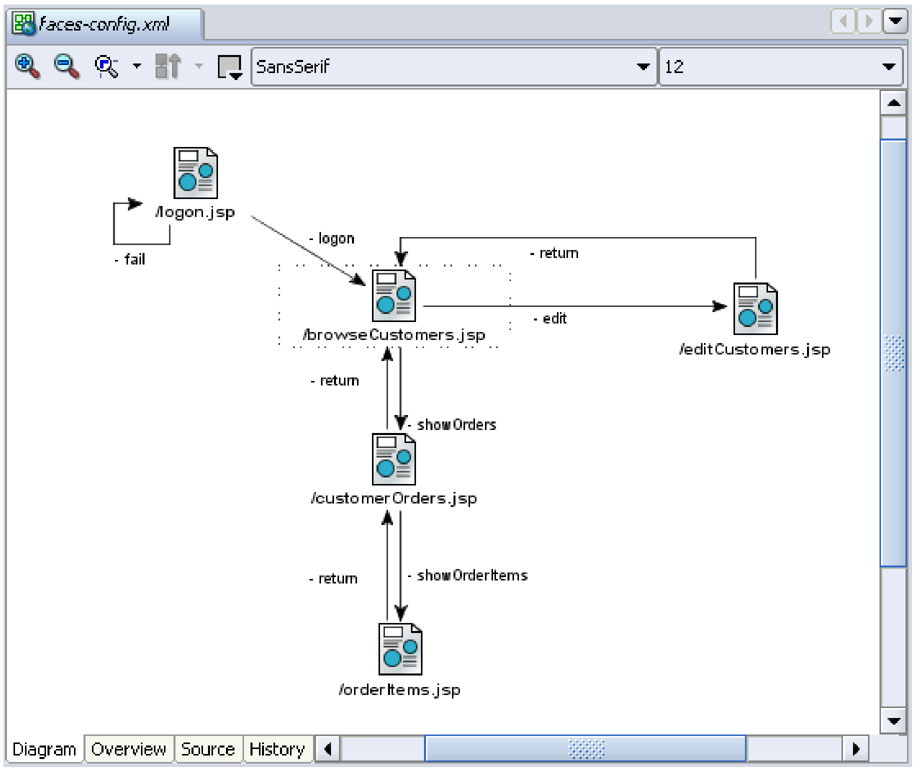


Figure 6 – Faces-config.xml

The action attribute

In order to define which UI component actually raises the action you simply set the *action* property on the component. This is the same property that was discussed earlier for defining the method to be executed on pressing a button. The *action* property can take the name of a navigation case on the diagram or the name of a method – which is why the action method discussed earlier, returns a String, which defines the name of the JSF navigation flow to be followed.

The managed bean

The role of the managed bean is actually quite simple. It is a place to put code relating to UI actions (like a button being pressed) and to hold state information about the UI (for example, the name of the logged on user). More specifically, a managed bean contains code generally shared between many pages.

The term Inversion of Control (IoC) is often used in describing managed beans. This simply means that the management of the object (in this case the managed bean) is handed over to the framework instead of being left to the application developer.

The lifecycle of a managed bean is handled by JSF – when the bean is first referenced, JSF will create an instance of the relevant Java class and keep it around as long as is specified by the bean definition.

You also have the concept of a backing bean. A backing bean is a specialized managed bean created and managed by JDeveloper and is tied to one specific JSF page. Typically this bean provides accessor methods for each UI component on the page it backs.

You don't need a backing bean for every page. A page can happily live without any backing bean at all, or a backing bean can only contain references to a few of the components on the page. In most cases the backing bean will only contain action code and no references to the UI Components – unless you need to programmatically manipulate them.

In the example above (figure 3) a backing bean was used to reference the components created on the page and to provide a place to put action code.

Managed bean definition

Managed beans are registered in the *faces-config.xml* file; however, JDeveloper can aid you in the creation and referencing of these beans. When a JSF application starts up it reads the *faces-config.xml* file to make the beans available. There are a number of configuration options for managed beans.

Managed bean properties

As well as the place to put UI action code, you can also add variables (like package variables in PL/SQL) to the managed bean. Consider the case where you would like to define a value that would be used as the default value for, one or many, UI fields. You can achieve this by using a managed property, which allows you define a property, its type and its value. This could then be accessed using EL to populate the value of a field. So the entry the faces-config.xml file would include:

```
<managed-property>
  <property-name>defValue</property-name>
  <property-class>java.lang.String</property-class>
  <value>hello</value>
</managed-property>
```

And that property could be access from a JSF page by:

```
#{backing_page1.defValue}
```

Managed Bean Scope

Each managed bean has a "lifespan". JSF will instantiate the bean automatically and that bean will be available for the defined *scope*. The scope of the managed bean is set in the faces-config.xml file. There are four possibilities:

- Application – the bean is available for the duration of the web application. This is useful for any application global data.
- Session – the bean is available for the duration of the client's session. For example, a shopping cart application would keep the session information across many web pages.
- Request – the bean is available from the time it is instantiated until a response is sent back to the client. For example, from the point a button is pressed to when the response is received back at the client.
- None – the bean is instantiated each time it is referenced.

Referencing the managed bean

Now that you have defined information and functionality within your managed bean, you want to be able to access that information.

Referencing from EL

Using EL you can reference managed properties from your managed bean.

```
<af:inputText id="fieldA" value="#{backing_page1.defValue}"/>
```

Thus, on realizing this field, the value of the field would be set to the defValue managed property in the backing_page1 managed bean.

Referencing from Java

You can also access managed properties from Java as well. To do this you get the current instance of the FacesContext. This object contains all of the state information for the components within the JSF lifecycle. By getting a handle to the current instance of the FacesContext you can access the elements of the JSF lifecycle.

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

ValueBinding expr =

fc.getApplication().

createValueBinding("#{backing_page1.defValue}");

System.out.println(expr.getValue(fc).toString());
```

Converters and validators

When the user enters data into a JSF page and that page is submitted from the browser, all the data on the page is enclosed in the resulting HTTP message as strings. So, regardless if data entered was a date or a number when the data gets to the application server for processing it appears as a string.

The predicament here is that the developer building the "back-end" code will typically be using Java types (like Integer or Date) to manipulate that data.

Without the use of a framework this would require writing code to convert the strings to the correct data type.

However, this functionality is provided to you through JSF.

Converters

As the name would suggest, a converter is responsible for converting String data to a type such as *java.util.Date* or *java.lang.Number*. JDeveloper provides a palette of converters and will also automatically add converters for components that would reasonably require them: e.g. an input field that is bound to date data. Thus, data entered that represents a date, and is transmitted via HTTP as a string, can still be typed in the back end code as a *java.util.Date*.

Validators

Validation, on the other hand, is the process whereby editable component data is validated against rules or conditions before the data is updated into the model layer. Thus, before posting data on the page, validation will check the data is, for example, within defined limits (*<af:validateDateTimeRange>).* As with converters, JDeveloper provides a palette of validators and will automatically include them on certain UI items.

THE JSF LIFECYCLE

We have already come across the phrase "JSF lifecycle" in this paper and it is worth looking at what this means in a little more detail.

Overview

Once a request starts being processed (e.g. as the result of pressing a button), JSF goes through a set series of steps. These steps are called the lifecycle**.** The lifecycle of a JSF page is similar to the lifecycle of most web pages: the client makes an HTTP request, the application server receives and processes the request and responds in HTML. The JSF lifecycle, however, manages a UI component tree on the server and also controls events and handles state changes.

While outside the scope if this paper, the JSF lifecycle can be customized if you require specialized behavior.

The JSF lifecycle uses the component tree to handle application requests and create rendered responses. Depending on the type of request and response, JSF executes different tasks in different phases of the request processing lifecycle, and then creates a view of the page (component tree) for rendering (in the Render Response phase).

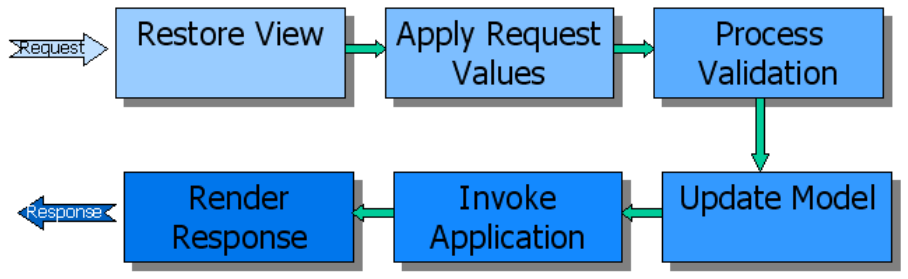


Figure 7 – JSF lifecycle

Looking at the phases

The fine details of each of the phases are beyond the scope of this paper, however, an acknowledgement of their existence and general function is useful. In brief, the JSF lifecycle phases are as follows:

Restore View

An internal view of the page is held in a component tree. In this phase, the component tree of a page is newly built or restored. All the component tags, event handlers, converters, and validators have access to the *FacesContext* instance. If it is a new empty tree, the lifecycle proceeds directly to the Render Response phase.

Apply Request Values

Each component in the tree extracts its new value from the request message and stores it locally. If a component has its *immediate* attribute set to true, then the validation, conversion, and events associated with the component is processed during this phase.

Process Validations

The local values of the components are converted and validated. If there are errors, the lifecycle jumps to the Render Response phase. At the end of this phase, new component values are set, and any conversion error messages and events are queued on FacesContext.

Update Model Values

The component objects' properties are set to the validated local values.

Invoke Application

Application level code is executed such as action events.

Render Response

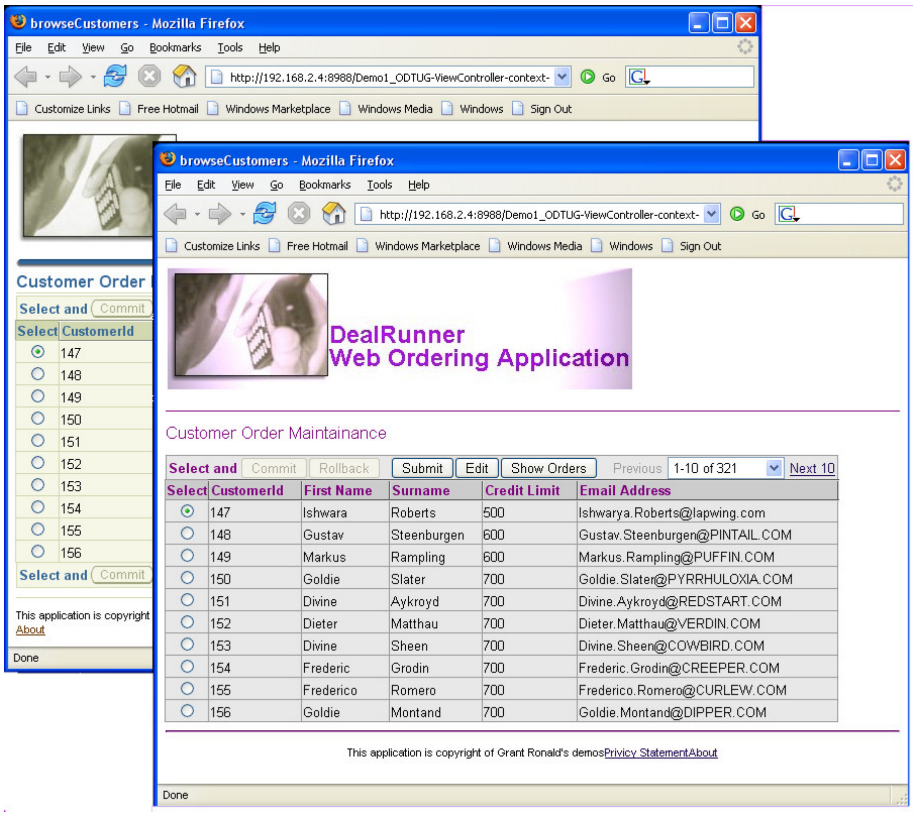
The components in the tree are rendered as the web container traverses the tags in the page. State information is saved for subsequent requests and the Restore View phase.

JSF ADVANCED FEATURES

As an introductory paper, these advanced features are beyond the scope of this paper. However, an acknowledgement of their basics is useful to know.

Skinning

Skinning is the ability to define an overall "look and feel" to an application UI. In the same way that Windows allows you to define a theme for your desktop, ADF Faces provides this feature by using JSF's pluggable rendering technology.



Select	CustomerId	First Name	Surname	Credit Limit	Email Address
●	147	Ishwarya	Roberts	500	Ishwarya.Roberts@lapwing.com
○	148	Gustav	Steenburgen	600	Gustav.Steenburgen@PINTAIL.COM
○	149	Markus	Rampling	600	Markus.Rampling@PUFFIN.COM
○	150	Goldie	Slater	700	Goldie.Slater@PYRRHULOXIA.COM
○	151	Divine	Aykroyd	700	Divine.Aykroyd@REDSTART.COM
○	152	Dieter	Matthau	700	Dieter.Matthau@VERDIN.COM
○	153	Divine	Sheen	700	Divine.Sheen@COWBIRD.COM
○	154	Frederic	Grodin	700	Frederic.Grodin@CREEPER.COM
○	155	Frederico	Romero	700	Frederico.Romero@CURLEW.COM
○	156	Goldie	Montand	700	Goldie.Montand@DIPPER.COM

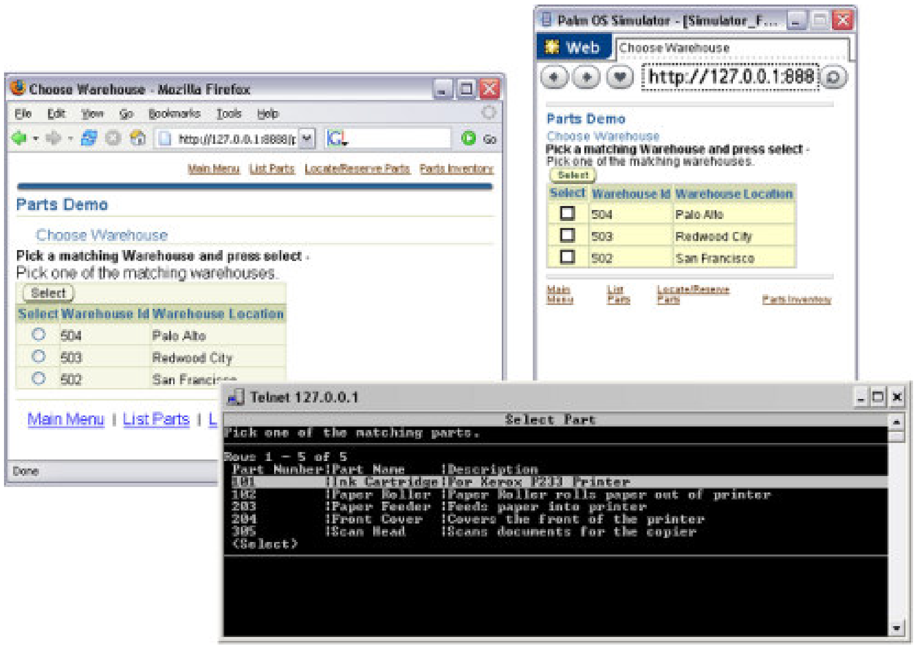
This application is copyright of Grant Ronald's demosPrivicy StatementAbout

Figure 8 – Skinning

Figure 8 shows the same application rendered using different skins. ADF Faces comes with a number of pre-configured skins but you can develop your own custom skins through manipulation of CSS (Cascading Style Sheet) tags.

Render Kits

Each JSF component is responsible for outputting its own markup tags that are used by the end device to physically render the screen. ADF Faces provides render kits for HTML, mobile and telnet.


Choose Warehouse - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://127.0.0.1:8888/
Go
Main Menu List Parts Locate/Reserve Parts Parts Inventory
Done
Palm OS Simulator - [Simulator_F...
Web
Choose Warehouse
http://127.0.0.1:888
Main Menu List Parts Locate/Reserve Parts Parts Inventory
Telnet 127.0.0.1

Parts Demo

Choose Warehouse

Pick a matching Warehouse and press select -
Pick one of the matching warehouses.

Select

Select	Warehouse Id	Warehouse Location
○	504	Palo Alto
○	503	Redwood City
○	502	San Francisco

```
                              Select Part
Pick one of the matching parts.

Rows 1 - 5 of 5
 Part Number|Part Name     |Description
 101        |Ink Cartridge|For Xerox P233 Printer
 102        |Paper Roller |Paper Roller rolls paper out of printer
 203        |Paper Feeder |Feeds paper into printer
 204        |Front Cover  |Covers the front of the printer
 305        |Scan Head    |Scans documents for the copier
 <Select>
```

Figure 9 – Render kits

Figure 9 shows samples of an application rendered using different render kits. JSF also has the provision of allowing you to develop your own render kits.

Ajax

Web UIs work on a "click and wait" paradigm. That is to say, a request is only initiated on a UI action (such as a clicking a button) and the UI will wait for a response. Which can be a limiting factor in trying to build a rich experience. This has been a factor in the emergence of so called Ajax technology.

Ajax is based around asynchronous JavaScript and XML. Browsers now support the ability to asynchronously request XML data. Thus, the combination of this asynchronous passing of XML data, and JavaScript on the browser lends itself to producing much richer web UIs.

The beauty of this technology for 4GL developers is that the Ajax functionality can be hidden within JSF components.

Some ADF Faces components already exhibit "Ajax like" behavior, but at the time of writing this paper, work is underway in developing true Ajax functionality within the ADF Faces components.

CONCLUSION

For further reading on how Oracle JDeveloper and ADF aid you in developing JSF applications,see the [1.3 Declarative Development with Oracle ADF and JavaServer Faces](http://download-uk.oracle.com/docs/html/B25947_01/intro003.htm#CHDHDHFJ) section of the ADF Developer's Guide for Forms/4GL Developers.

http://download-uk.oracle.com/docs/html/B25947_01/intro003.htm#CHDHDHFJ.

Java Server Faces is now providing a rich, productive and standards based framework for developer web UIs. By providing a componentized layer of abstraction, it factors away much of the complexity of web UI development, particularly for those developers familiar with 4GL tools such as Oracle Forms, PeopleTools and Visual Basic.

Introducing Java Server Faces (JSF) to 4GL Developers
November 2006
Author: Grant Ronald
Contributing Authors:

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com