Pretty Java Server Faces

Lucas Jellema, AMIS Services B.V.

JSF: the long awaited Joyful Successor to Forms

Developing Web Applications using Java/J2EE technology has not been a picnic over the last few years. It was often hard work, complex, not very productive and the results were disappointing too. Especially for developers with an Oracle Forms background. Now, at last there is a Joyful Successor to Forms: JSF or Java Server Faces. It is component based, it supports an event-trigger model, it has declarative validation and binding to a backend model, programmatic manipulation of page elements, it is productively supported by IDEs that provide for a largely drag & drop WYSISYG development style and it is based on an industry wide standard. JSF finally brings Java Web Application Development to the level of Oracle Forms.

This article introduces Java Server Faces to developers with some understanding of the development of HTML Web Applications using Java technology. We will discuss the core JSF concepts and create a simple application in a number of steps, using JDeveloper 10.1.3 as our IDE. The article concludes with overviews of both Oracle ADF Faces - a very rich library of JSF components - and JHeadstart – a tool for generating database based ADF Faces applications. Note that this paper will cut some corners when discussing the theory – we cannot of course cram material discussed in rather thick books in this single paper.

A little history

The development of dynamic HTML Web Applications using Java technology has gone through an almost 10 year long evolution. Initially CGI was used to have a Web Server invoke a standalone Java program to generate HTML. In 1997 this approach was superseded by the Servlet. Even though the Servlet improved performance, ease of development and allowed for much easier interaction with the end user, it did not provide a very productive or way of working. With the advent of the JSP in 1999, Java WebApp development rid itself of a lot of the repetitive lines of code found in Servlets. It also promoted mixing HTML markup code with simple Java snippets.

It turned out that the direct mixture of Java and HTML could become unwieldy. Through Custom Tag Libraries it was possible to replace much of the Java in JSPs with special tags that represent those pieces of Java code. However, things could still be improved upon. Every vendor and open source project and many a developer as well created their own custom tags. Many tags appeared that did similar things, though in slightly different way. Attributes fed into tags differed just a little. The year 2002 brought the JSTL (JSP Standard Tag Library), That streamlined the use of the custom tags for a lot of very common operations and made it available in J2EE compliant application servers. It also introduced Expression Language (EL) that made it possible to dynamically evaluate the value of custom tag properties.

Struts – the Controller

Parallel to this evolution in the View side of the MVC architecture, we saw similar development on the Controller side of things. Many clever developers realized that JavaServer Pages AND servlets could be used **together** to deploy web applications. The servlets could help with the control-flow, and the JSPs could focus on the nasty business of writing HTML. In due course, using JSPs and servlets together became known as Model 2 (meaning, presumably, that using JSPs alone was Model 1). In 2000 the Struts framework was launched by Craig R. McClanahan to provide a standard MVC framework to the Java community. It quickly grew to be the de-facto standard Controller, even though there always have been a number of contenders such as Spring MVC and Tapestry (see http://bdn.borland.com/article/borcon/files/6000/paper/6000.html for a comparison).

JSR-127 – introducing a new standard for View and Controller

Early 2004 – when we were still suffering from a lot of complexity, a plethora of frameworks to choose from, often poor productivity and disappointing results, despite all developments described above – the first release of Java Server Faces was published by the Java Community Process as JSR-127. Led by the father of Struts - Craig R. McClanahan – all major vendors in the Java industry participated in this specification, including IBM, Borland, Sun, BEA and of course Oracle. JSF aims to set the new standard for both the View and the Controller in the MVC architecture pattern, leveraging all the work done in the past with Struts, JSP, EL and custom tag technology as well as the latest developments in Java technology.

Java Server Faces is part of the J2EE specification. That means that all J2EE application servers will offer support for JSF, just like they do for JSP. It also means that at last there is a standardized way of designing UI components, in a Controller framework that integrates with a backend Model. This results in easy adoption and mixing of JSF components from various vendors and advanced support from IDEs for JSF development. That support includes drag & drop layout and page navigation development, shortcuts for binding component attributes to bean methods and property palettes with lists of values and hints.

In 2006 we see a number of IDEs with substantial support for JSF, including Sun's Java Studio Creator, NetBeans, WSAD and JDeveloper. There are a number of JSF Component Libraries that we can pick and choose from. The most important ones are: the Reference Implementation, Apache MyFaces and Oracle ADF Faces. Of course developers can create their JSF own components - even though that is not a trivial task to undertake. It has taken some time since 2004 but the buzz and the momentum of JSF seem to be rapidly growing. Oracle's release of JDeveloper 10.1.3 and the availability of the ADF Faces library of JSF components are important elements in this development.

The heart of the JSF matter

We will get to know the essential pieces of the JSF puzzle in a number of steps. These pieces are: the UI Components, the Model including Managed Beans and Binding properties and methods, the Events-and-Listeners infrastructure, Converters and Validators and Navigation.

UI Components

At the very core of JSF are the UI components. These are represented through Custom JSP Tags when using JSP and HTML for rendering the Web application. Note that JSF is not restricted to HTML even though that is still the most common usage; the JSF UI components can have renderers for any technology, for example XUL, HTC, WML, Telnet, Flash, SVG. In this article we will focus on JSF rendering HTML and deployed through JSPs.

The JSF specification describes a fairly basic set of UI Components. The set contains components like HtmlForm, HtmlInputText, HtmlSelectOneRadio/ HtmlSelectOneListBox and HtmlDataTable. These render in HTML as a form, an input field, a radio group or a select list and a table.

JSF pages are created in JDeveloper 10.1.3 from the New Gallery, Web Tier, JSF node picking the JSF JSP option. A wizard asks us some details like page name, CSS stylesheets to link in and JSP Tag Libraries to import. For now we will stick to the default taglibs JSF Core and JSF HTML. When the wizard is finished, the WYSIWYG JSP editor is launched.

The JSF page is defined by a tree structure starting at the <f:view> root element and consisting of all UI Components in our page. Note that in the JSP we can freely mix normal HTML, normal JSP and JSF elements. In JDeveloper, we create a page rapidly by dragging components of our choice from the JSF Component Palette onto the JSP editor or the Structure Window. Usually a simple wizard or property editor pops up to help us set some key properties for the component we just added.

In Figure 1 we see a simple page under construction. It is to implement a Search Books page where the user can provide some search details. The page contains an HtmlSelectManyListBox component that renders as a Select item with multiple set to true. Note that we can very easily replace this with a collection of Checkboxes – one for each Topic – by changing selectManyListBox to selectManyCheckBox.

One of the important features of JSF is that this page tree is not only used for rendering the HTML, it is also accessible to our application code at runtime. Just like in Oracle Forms, we can programmatically read the Component Tree, we can also manipulate it. Reorganizing the page, adding components, changing component properties. It can all be done!

Component Properties

UI Components are configured through properties. Some of these properties are generic and apply to all UI Components: id, value, rendered, binding. Others are more specific. All Input Components for example support properties like required, value changeListener, readOnly, validator, converter.

A third category consists of the properties that are not JSF properties at all: the HTML pass-through properties like accesskey, title, maxlength, tabindex, styleClass (for the class property) and the on… event triggers for change, focus, blur, keydown etc.. These are the properties that have a –render technology specific - meaning in HTML and are simply passed through by the JSF component. Note that these properties can be set and read programmatically, just like the real JSF properties.

In the next paragraph we will see how these properties cannot just be set with static values but also with dynamically evaluated expressions.

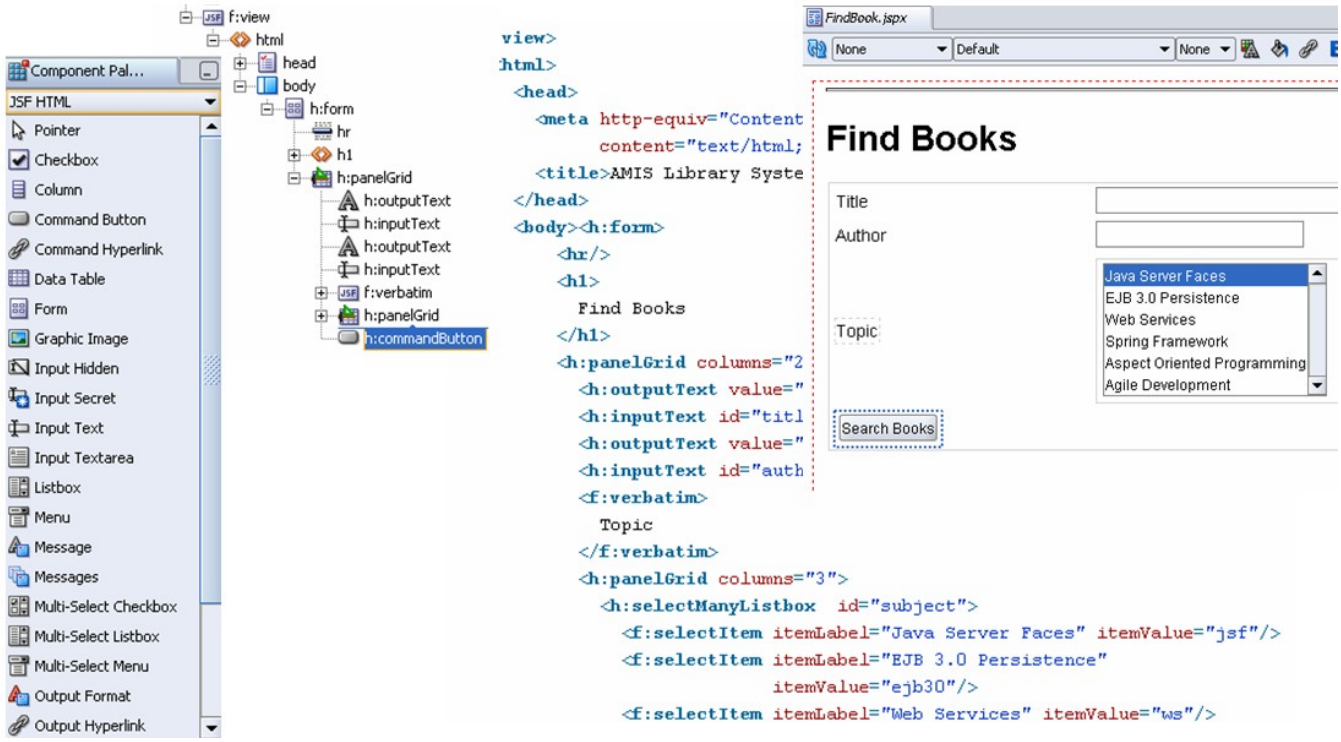

Figure 1 – Creating a JSF page's VIEW tree by dragging UI Components to the Structure Window, the Source View or the WYSIWYG JSP editor

The JSF Model – Expressions Language, Managing and Binding Beans

All properties on JSF Components can be specified using JSF Expression Language (EL) Expressions. The JSF EL is very similar to the JSP Expression Language that is used for example with JSTL. JSF EL is more advanced as it supports two-way (read/write) expressions, expressions referencing not just bean properties but also methods and the ability to create and evaluated expressions programmatically. JSF EL expressions are marked by #{}, while JSP EL uses ${}.

While evaluating EL Expressions that reference beans (really just Plain Old Java Objects), JSF automatically looks in all 'context scopes' like the Servlet's ApplicationScope, SessionScope and RequestScope. Thus we can easily access request parameters and application initialization parameters. JSF EL Expressions look like:

```
value="#{param.title}"
rendered="#{param.publicationYear > 1999}"
Action="#{bookSearch.doSearch}"
```

Note that the Action property in the third example is bound to the doSearch() method on the bookSearch bean. When the button or link that this property set is clicked on, this method will be invoked during server side request processing.

Managed Beans

In the not too distant past we were responsible ourselves for instantiating the Java objects our JSPs needed, for example in Servlet Filters or Listeners or using <jsp;useBean> tags. JSF offers more advanced facilities for handling our objects, through the Managed Beans functionality. In the faces-config.xml configuration file in the WEB-INF directory of our web application – automatically created by JDeveloper – we can specify our Managed Beans. Per bean we configure the name and the scope in which it is to be placed as well as the class that is used to instantiate the bean. Note that when we say bean we really only mean an object with a no-argument constructor.

The Managed Bean infrastructure also lets us define the initial values of properties. These initial values can be specified using JSF EL expressions that may refer to Managed Beans! This allows us to construct a substantial object graph during startup of the Web Application, without any programming.

The Managed Bean definitions are written in straightforward if verbose XML. However, JDeveloper has a console for editing the faces-config.xml file that makes life much easier.

```
<managed-bean>
  <managed-bean-name>bookSearch</managed-bea
  <managed-bean-class>nl.amis.als.Book</mana
  <managed-bean-scope>session</managed-bean-
  <managed-property >
    <property-name>title</property-name>
    <property-class>java.lang.String</proper
    <value>Mijn title</value>
  </managed-property>
  <managed-property>
    <property-name>publicationYear</property
    <property-class>java.lang.Integer</prope
    <value>2005</value>
  </managed-property>
</managed-bean>
```

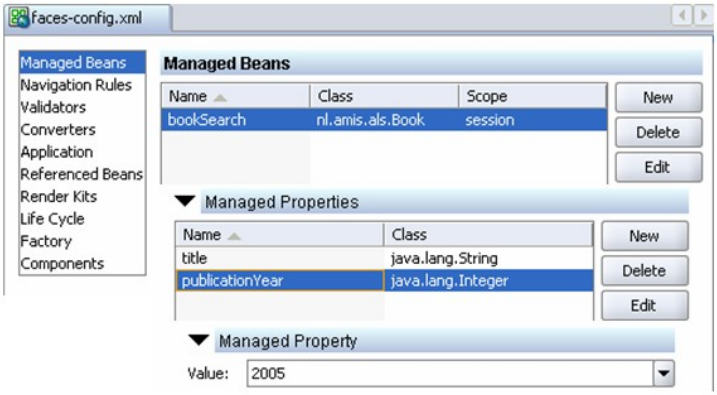

Figure 2 – Managed Beans and the JSF faces-config Console

In the following code snippet, we see how several properties have been defined using EL Expressions that bind them to properties and methods on the managed bean bookSearch:

```
<h:panelGrid columns="2">
  <h:outputText value="#{bookSearch.titlePrompt}" title="#{bookSearch.titleBubble}"/>
  <h:inputText id="title" title="Book Title" value="#{bookSearch.title}"/>
  <h:inputText id="publicationYear" title="Year of First Publication"
               value="#{bookSearch.publicationYear}"/>
  <h:commandButton value="Search Books" action="#{bookSearch.doSearch}"/>
```

Note how pass-through property title has been bound to the titlebubble property on bookSearch. See also that the action property on the command button is associated with the doSearch() method.

Events and Listeners

Your typical Oracle Forms developer can be said to be trigger-happy: it is so convenient to associate little pieces of PL/SQL code with events taking place as a result of the actions of the user with our page elements, such as pressing buttons or changing field values.

JSF has a very similar model: UI Components like Buttons and CommandLink publish Action Events when clicked upon and all Input Components publish ValueChangeEvents when the user has entered a new value. It is important to realize that while these events originate from a user action on the client, it is only on the server side during request processing that the Components find out about the events that have occurred.

Each Input Component has a valueChangedListener property that can be bound to a method in some bean. This method needs to have signature like :

```
public void changeAuthor(ValueChangeEvent valueChangeEvent) {
 FacesContext fc = FacesContext.getCurrentInstance();
 fc.addMessage( valueChangeEvent.getComponent().getClientId(fc)
              , new FacesMessage("Changed value of Author; Old value was: "
                                 + valueChangeEvent.getOldValue())
              );
}
```

This particular valueChange listener method adds a message to the FacesContext whenever the value in the author input item changes, because it is bound to the valueChangeListener property of the author InputText component:

```
<h:inputText id="author" value="#{bookSearch.author}"
 valueChangeListener="#{bookSearch.changeAuthor}"/>
```

Note that using valueChangeListener child elements we can associate multiple listeners with a single input item.

Buttons and Command Links generate Action events when clicked. Action events can be consumed by ActionListeners. These come in three shapes: ActionListener child elements, the actionListener property that is bound to a method on a bean that takes an ActionEvent as input parameter and the action property that either specifies a static String, an EL Expression that evaluates to a String or an EL Method Binding that links to a method that returns a String and does not take any parameters. We will see in a subsequent section how that String plays an important part in the navigation across pages.

The Search Books button in our sample application is bound to the doSearch() method in our bean:

```
<h:commandButton value="Search Books" action="#{bookSearch.doSearch}"/>
```

This doSearch() method:

```
public String doSearch() {
    FacesContext fc = FacesContext.getCurrentInstance();
    fc.addMessage(null, new FacesMessage("Button Was Pressed!"));
    return "search";
}
```

If we run this extremely simple application, we see a number of interesting things at work:

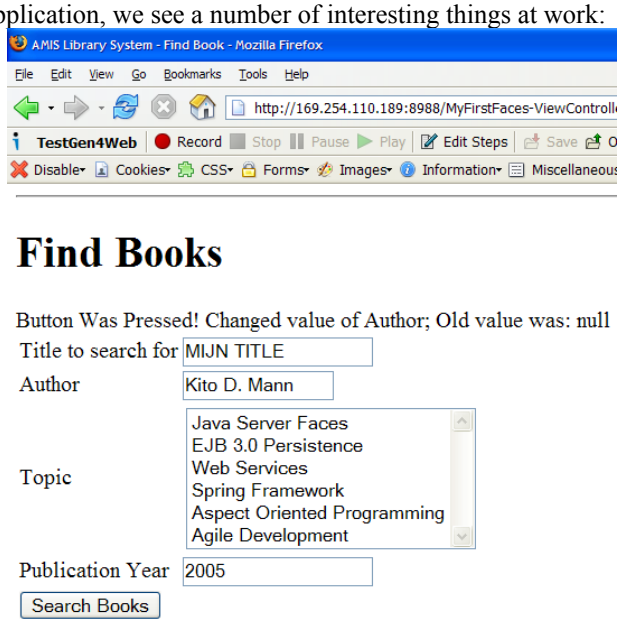

Figure 3 – Running the Find Books page – Property binding and Events Listeners in action

The Title and PublicationYear fields have the default values that were specified in the ManagedBeans section of the faces-config.xml file. If I enter a value in the author field and press the Search Books button, the page is refreshed and two messages are shown – one from the valueChange event on the author field and one from the action event caught by the doSearch() method. The values entered on the form are now stored in the backing bean.

It may sound like nothing spectacular, but in terms of Java Web Applications, we have already accomplished quite a bit!

Converters and Validators

When the user enters data in the HTML form and presses a button, the form is submitted with all the values that were entered. However, no matter if the user entered numeric, date or user defined types: all values are sent by the browser as Strings! At the same time, the backing bean properties are typically strongly typed: Date, Integer, Long or even PostalCode. Somewhere, someone has to convert those String parameters from the HttpRequest into the strongly typed properties on the backing beans. For this task, JSF has the concept of Converters. And vice versa: for generating the display, the strongly typed values are converted to nicely formatted strings, such as dd/mm/yyyy for European style dates.

JSF has built-in converters for common types. For numeric and date properties, the converter accepts format-patterns, very much like the format strings we can use in Oracle with the TO_CHAR, TO_DATE and TO_NUMBER operations. We can also write our own converters for custom types. This is only necessary if we bind UI Components to properties in backing beans that are not covered by the standard converters. Note that if you do not specify a converter, JSF will pick one itself. So for properties of common types like String, Long, Integer, Date you do not have to worry about converters at all.

Validation of user input

Though we all love our end users, we know they cannot be trusted! Or at least they are bound to make mistakes in the values they enter into our web application. To prevent such erroneous data from entering the enterprise database or even the business tier of our application – as well as providing helpful feedback in a patient yet expeditious manner – we typically implement data validation in either the client or the web tier (or both). Java Server Faces offers a validation framework out of the box, using a combination of validator elements and validator methods.

A Validator is a generic component that implements a certain somewhat configurable validation for values entered into the component it is linked to. JSF has a small number of standard Validators like LongRange that enforce numeric values to be within the specified range and Length that ensures that the value is at least x long and/or no longer than y.

In our simple application, we have specified that the Publication Year must be between 1998 and 2006. The Title field should contain at least 5 characters:

```
<h:inputText id="title" title="Book Title" value="#{bookSearch.title}">
  <f:validateLength minimum="5"/>
</h:inputText>
<h:inputText id="publicationYear" title="Year of First Publication"
             value="#{bookSearch.publicationYear}">
  <f:validateLongRange maximum="2006" minimum="1998"/>
</h:inputText>
```

JSF allows us to define our own custom Validators. Alternatively, we can specify Validation method bindings that tie an Input Component to a method on a bean that implement the Validation method signature, like this:

```
public void subject_validator(FacesContext facesContext,
                              UIComponent uiComponent, Object object) {
  if (((String[])object).length > 3) {
    ((UIInput)uiComponent).setValid(false);
    FacesMessage message = new FacesMessage("No more than three topics please!");
    facesContext.addMessage(uiComponent.getClientId(facesContext), message);
  }
}
```

This method is bound to the subject selectManyListBox component:

```
<h:selectManyListbox id="subject"  validator="#{bookSearch.subject_validator}">
```

The validations as defined in the JSF standard are server side only. That means that it takes a server request for the validations to be performed, so there are no true field level validations. JSF implementations like ADF Faces have added instantaneous validation, either through Client Side validation with JavaScript or with AJAX based invocation of the server side validation logic.

Navigation

Our application so far had only a single page. In real life, it will not be uncommon to have applications with multiple pages. One of the pretty important tasks in multi-page applications is controlling the navigation between these pages. How is navigation initiated and where should the user be taken. Following the MVC pattern, we want to ensure that pages are unaware of each other. Navigation logic is not defined in individual pages but centrally in the controller component of our application.

The central repository of Navigation Rules is the faces-config.xml file. Here we can indicate where the application should take the user given a certain situation. Navigation rules specify where, coming from a certain page (the from-view-id) given a certain condition (from-outcome) to which page (to-view-id) we go. Figure 4 shows the navigation rules for our application – extended with a Welcome page and a Search Results page – both in the faces-config.xml file and in the Faces Config console in JDeveloper.

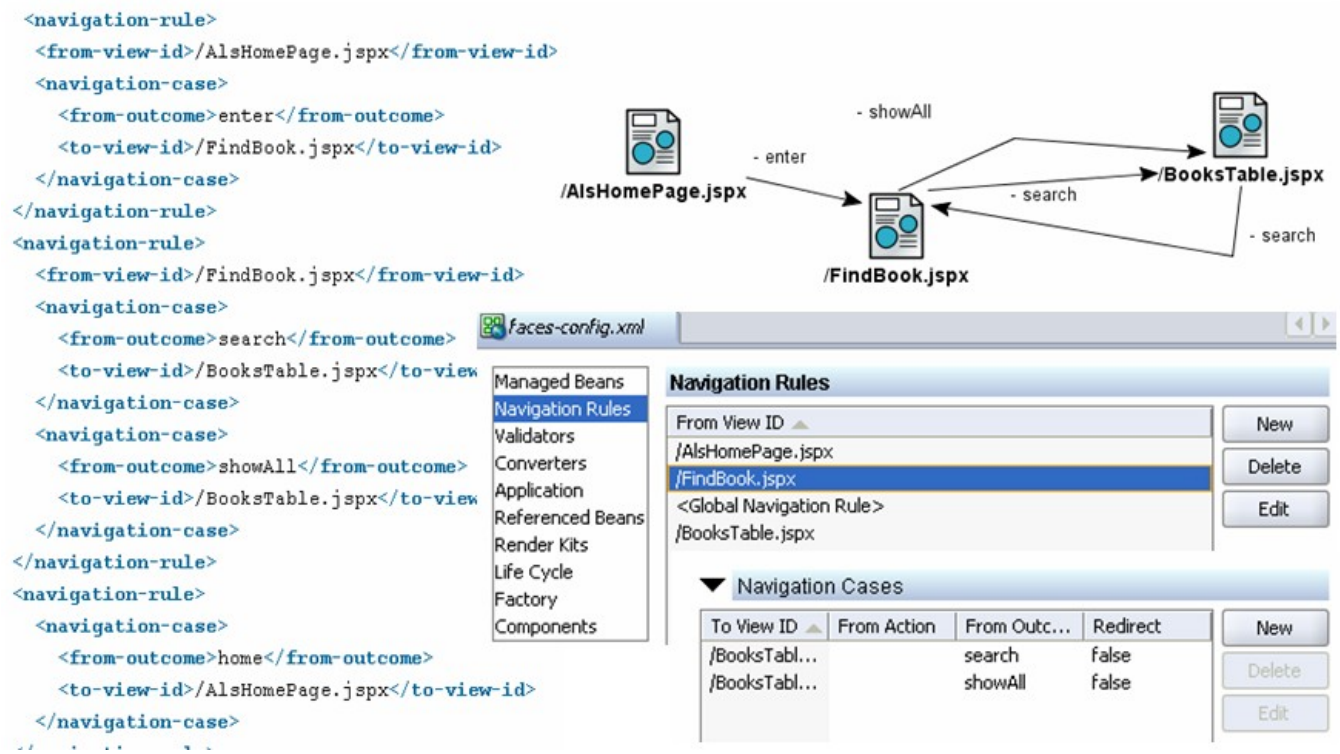

Figure 4 – Navigation Rules for the Library Application

The one question you should have right now is: where does the from-outcome value come from? Where is it set? Well, maybe you have guessed it: the action property we have set on a button or a command link specifies this value, either directly or by invoking an action method like the doSearch() method we have seen before – that returns the string "search".

Other stuff

There is a lot more to JSF. In this paper however, we will not discuss the following:

- Resource Bundles and i18n
- Client Side Validation and AJAX (Apart from built-in functionality of ADF Faces)
- Integration with frameworks such as Struts (Shale) and Spring (JSF-Spring)
- File Upload/Download functionality in JSF
- Programmatically manipulating pages (Adding, removing and changing components through Java code)
- Using JSF outside the HTML scope for protocols like WML, PDA, Telnet, XUL, HTC etc.
- Customizing and Extending existing UI Components or Renderers, or even developing new ones
- Other JSF Implementations such as Apache MyFaces, Smile, BackBase Java Server, WebGalileo Faces, ICEFaces, ECruiser, Keel

Pretty (and) Productive

Up until this point we have created simple and sober user interfaces. Of course through interesting images and proper CSS stylesheets, we could have improved upon the looks of the pages, but in terms of advanced user interaction and rich widgets it has been a little on the sober side.

The JSF Specification includes the HtmlDataTable component and that is where things start to get more interesting. The DataTable is used for displaying a Collection of Objects in a grid like structure. It supports Row-banding, column-banding, header, footer, pagination (start with, rows) and (multi-record) edit. The DataTable can be bound to a Java Collection of Beans, an Array and also to a JDBC ResultSet. In JDeveloper 10.1.3, dragging a DataTable to the page will open a wizard that allows us to bind the table to a method that returns a data set. The wizard will do introspection on the objects in the data set and come with a proposal for the columns in the table – based on the properties in those objects. Through only using the wizard, you can create an advanced table in under a minute.

Our Library Application has a managed bean called libraryManager. This bean has a method getBooks() that returns a collection of Book objects. The Book class has been set up with a number of typical book properties like title, isbn, author and publication year. By simply binding our DataTable to this method and binding the columns to the properties of the local book iterator of type Book, we get our DataTable going. Figure 5 illustrates this.

Figure 5 – Binding a DataTable to a Collection of Books and presenting Book properties in columns

Compared to more traditional Web Application development, this IDE supported, wizard driven creation of a DataTable bound to any collection property on a bean, is almost overwhelming. It is like the Oracle Forms data block wizard!

Oracle and Apache ADF Faces

In December 2005 at JavaPolis in Belgium, Oracle announced the donation of its ADF Faces components to the Open Source Community, in particular the Apache MyFaces project. ADF Faces is a library of over 100 components that provide rich client side functionality in a visually appealing and very well designed way – based on the Oracle Usability Lab's Browser Look and Feel Guidelines – also see http://www.oracle.com/technology/tech/blaf/index.html. With ADF Faces, JSF developers have at their disposal a wide variety of UI components that make developing rich and attractive user interfaces in the standard JSF way very real.

ADF Faces components are JSF Components that adhere to all JSF standards. They can be used together with components from other libraries. ADF Faces components are defined through their own Tag Libraries and set of JSP tags. ADF Faces started life as Oracle UIX (User Interface XML) and was first made public in 2001. UIX was created for the development of HTML User Interfaces in the Oracle E-Business Suite (Self Service Apps, CRM modules). In 2004 it was integrated into JDeveloper as ADF UIX and in 2005 it evolved into ADF Faces. As we speak, it is one of the (if not the) most advanced JSF implementation to date. And, like JDeveloper, it is free!

The features offered by ADF Faces include rich set of components (tree, calendar, list of values, shuttle, train, colorchooser, menu, table navigation, …), additional converters and validators – including Client Side validation, support for AJAX or partial page rendering, Skinning, Accessibility - support similar to ADF UIX Accessibility, Bidirectional language support - ADF Faces components automatically render themselves appropriately for bidirectional languages, ADF (binding framework) integration - including support for JSR-227 (Data binding), Rich Client - upcoming rich DHTML client-side renderers.

Using almost the same DataTable wizard that we used for the JSF standard DataTable, we can create an ADF Table. It offers features like built-in sortable headers, pagination (step through records in sets of some specified number at a time) and Detail Disclosure, illustrated in Figure 6 where the somewhat less important fields are moved to a detail block that is dislayed in-line when the Show icon is clicked on.

Figure 6 – Demonstration of the ADF Faces Table Component

We have also included some Menu Components from the ADF Faces library as well as a button or two. Coming from the standard JSF DataTable from figure 5 – with the backing beans in place – it took less than ten minutes to create this page.

ADF Faces and AJAX

The AJAX hype will not have gone unnoticed. Rich Internet Applications with dynamic interaction and partial page refreshes, eliminating hourglasses, full page reloads and annoying flickering of browsers, are all the rage. AJAX is primarily a concept that can be implemented in a number of ways. Key for us developers is that AJAX-style immediate responses to client side user actions should be easy to add to our page.

UIX, the predecessor of ADF Faces, already had Partial Page Rendering – long before the term AJAX was even coined. This functionality lives on in ADF Faces. This means for example that some of the Table features we saw before are 'AJAX enabled' meaning that only a partial page refresh is performed for sorting, paginating or detail show/hide – making for a very smooth end user experience.

In general we can say that most ADF Faces components can trigger an AJAX operation. When the user changes the value of an Input Component and leaves the field or when a Button or Link is clicked upon, we can have the server notified of the current situation in the form, allowing it to go through the standard JSF request lifecycle, including conversion, validation and update of the model as well as returning a partial page response. ADF Faces comes with JavaScript libraries that know how to merge this partial response into the page.

Among the frequently desired functionality we can easily implement in ADF Faces are instantaneous validation of field values using server side logic, maintain summary fields, refresh select-items based on user provided values.

A quick example: we want to provide instant feedback to the end-user when the Publication Year is changed. If the year is set to 1969, the validator method will raise a protest. Because of the AJAX functionality in ADF Faces, this protest is displayed almost immediately when the user leaves the field. The required setup in the page:

Set the autoSubmit property on the publicationYear inputText component to true:

```
<af:inputText id="publicationYear" value="#{book.publicationYear}"label="The Year of Publication"
              autoSubmit="true" validator="#{findBookBacker.validateYear}">
```

Have the partialTriggers property on the messages component include the publicationYear id:

```
<af:messages partialTriggers="publicationYear"/>
```

The somewhat silly validator method on the findBookBacker bean:

```
public void validateYear(FacesContext facesContext, UIComponent uiComponent, Object object) {
  Integer year = (Integer)object;
  if (year.toString().equalsIgnoreCase("1969")) {
    ((CoreInputText)uiComponent).setValid(false);
    FacesMessage message = new FacesMessage("Please, any year but 1969!");
    facesContext.addMessage(uiComponent.getClientId(facesContext), message);
}
```

As a result, the user is notified of the application's reluctance with regard to 1969 as soon as the user leaves the field:

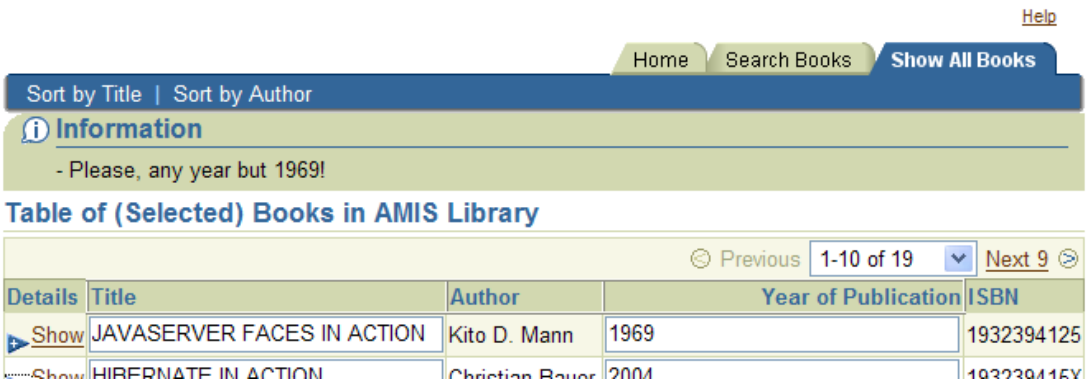

Figure 7 – AJAX style immediate server side validation when Year field is changed

Could you pass me the data please?

What we have *not* discussed so far is only about the most important part of most of our applications: the integration with the database. JSF is not peculiar about where data comes from. JSF components can be bound to backing beans and they do not care where those beans get their data from. That means that we can integrate Object Relational Mapping technologies such as Toplink, Hibernate or more general EJB 3.0 Persistence and of course ADF Business Components, and with the same ease Web Services, flat files, XML data sources or any other backend service. If our beans know how to get to the data, JSF can leverage them.

ADF Binding Framework

Late 2004, Oracle released ADF with at its core the ADF Binding Framework – a JSR-227 implementation avant la lettre. The ADF Binding Framework allows us to register Data Providers such as ADF BC Application Modules, WebServices or POJOs that expose data sets and data operations. The ADF Binding Framework wraps these providers from various technologies in a uniform interface – adding a fair bit of smart generic functionality – and makes them available as Data Controls to our web applications.

JDeveloper makes it possible to drag and drop these ADF Data Controls onto our page as ADF Faces components, that are automatically bound to a generic managed bean called *bindings*. Through this managed bean, the components can access all Data Controls and their data and operations.

Creating a Database backed JSF application consists of basically these two steps:

1. Create the ADF BC layer with Entity Objects, View Objects and an Application Module – this can be done through a simple wizard. The Application Module and its ViewObjects are automatically exposed as Data Controls
2. Create a JSF JSP page and drag one of the ViewObjects onto the page; indicate the way the Data Control should be represented on the page – see figure 8 for an overview of the options for a Data Set. A code snippet for an ADF Faces table linked to an ADF Data Control looks like this:

```
<af:table value="#{bindings.books.collectionModel}" var="row">
  <af:column headerText="#{bindings.books.labels.author}" sortProperty="author" >
    <af:inputText value="#{row.author}" required="#{bindings.books.attrDefs.author.mandatory}"/>
  </af:column>
```

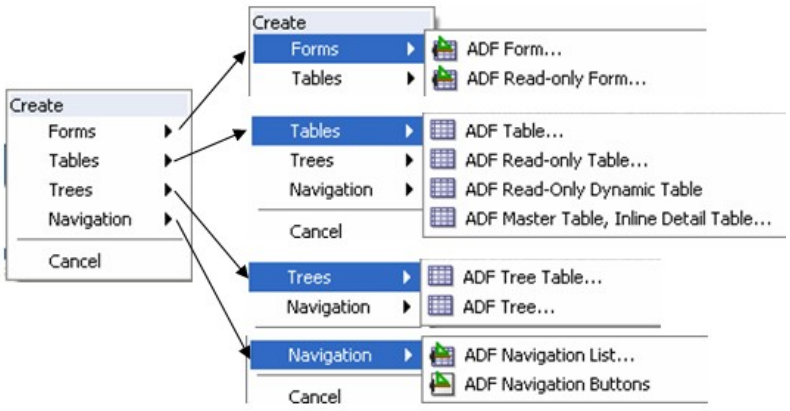

Figure 8 – Options available when dragging an ADF DataControl's DataSet onto a JSF page

This means that in no time at all we can create an advanced, rich user interfaces web application on top of our database.

The Generation Gap – Introducing JHeadstart 10.1.3

If JSF is the successor to Oracle Forms, you could say that JHeadstart is the heir of Oracle Designer. JHeadstart was first demonstrated as Project Atlantis at the 2001 ODTUG Conference in San Diego. The application that was generated at that time, using BC4J, UIX and the Oracle home-grown MVC Framework Controller, could be generated today using JHeadstart 10.1.3 using ADF BC and ADF Faces. The same meta-data that was created back in 2001 can still be used to drive generation! It is like CASE 5.0 meta-data, once used for generating SQL*Forms 3.0, that can now be used from Designer 10g to generate Oracle 10g WebForms.

JHeadstart allows us to design an application in a technology agnostic manner, using terms like Master-Detail, Tree, Shuttle, Table or Form layout, very much like we do with Module Definitions in Oracle Designer. JHeadstart will take our meta data and generate JSF JSP pages, the faces-config.xml file, Resource Bundles and the UI Page Model with the ADF Data Binding definitions. The 10.1.3 release has rich facilities for influencing and extending the generator's actions.

Note that JHeadstart also allows us to generate applications based on Module Definitions in Oracle Designer. That means it can be used to migrate Oracle Forms applications to the new world of ADF Faces, Bindings and Business Components.

Conclusion

We have our Joyful Successor to Forms. JSF is a very appealing technology, for at last we achieve 4GL style Web Application development with Java. JSF in combination with Oracle's JDeveloper 10.1.3 gives us strong IDE support, easy, visual and productive UI development, a robust and straightforward model for events and listeners, built in facilities for conversion and validation and simple navigation control.

JSF has clearly benefited tremendously from the experience gained with its predecessors, the shoulders it stands on, like Struts and JSP/JSTL. It takes the View and Controller in our Web Applications to the next level and it does so in an industry wide agreed manner that has gained support from all major vendors. JSF is a technology worth investing in.

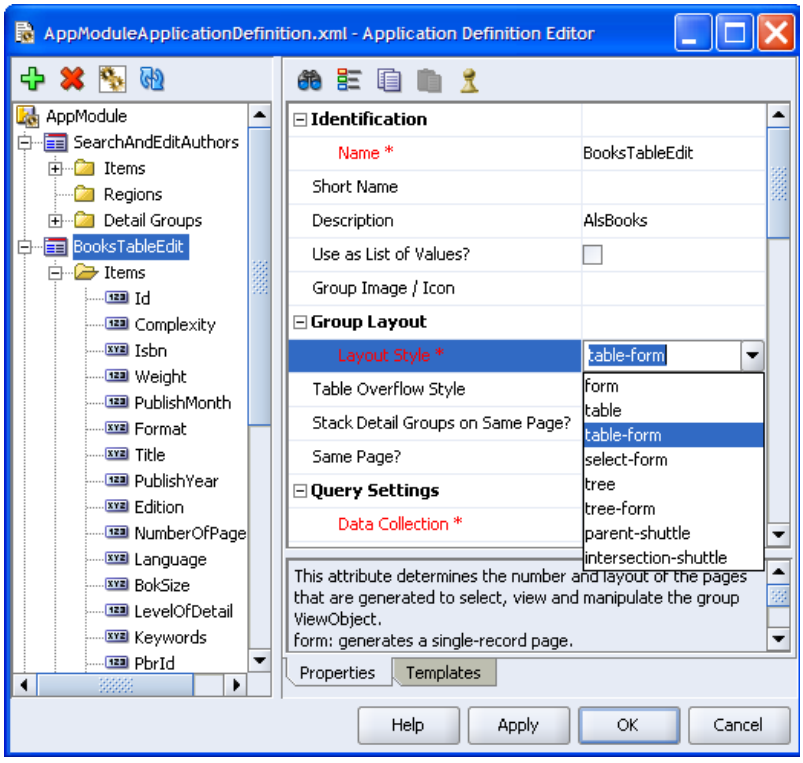

Figure 9 – The JHeadstart Application Definition Editor for specifying the Generator Meta-data

Using Oracle JDeveloper 10.1.3, in combination with ADF Faces and possible ADF Binding Framework and JHeadstart, allows us to develop even more productively and produce even richer user interfaces. The very structured approach we can take when developing using these technologies also means we run less risk with our projects and we create applications that are far easier to maintain than some of our past creations.

Oracle Forms shops who have been contemplating a (partial) move to Java now have a great opportunity to finally do so.

To download the source code for the examples in this paper as well as find a list of other useful resources, please go to: http://technology.amis.nl/blog/?p=1180 on the AMIS Technology Weblog.

About the author: Lucas Jellema is a well known face at ODTUG. He attended all ODTUG Conferences since 1997 and presented more than 25 times on many different topics. After 8 years at Oracle Corporation, in 2002 he joined AMIS Services, a Dutch consulting firm specializing in Oracle and Java technology. In November 2005, he was nominated Oracle ACE for his many contributions to the Oracle community, not least of which is the AMIS Technology Weblog (http://technology.amis.nl/blog) that features over 200 of his articles.