An Oracle White Paper
April 2009

Oracle ADF for Apache Beehive Developers Part I: The View and Controller

Introduction 1
Comparing Apples to Oranges 2
Keep it Simple! 2
Architecture 3
Form Beans 5
Page Flow 9
Navigation in Apache Beehive 9
Navigation in Oracle ADF 13
Component events in ADF 17
Exception handling 19
Lifecycle 21
Apache Beehive 21
Oracle ADF 21
NetUI form controls vs. ADF Faces Rich Client 22
NetUI 22
ADF Faces Rich Client and DVT 24
UI Data binding 25
Business Service integration (controls) 26
Apache Beehive to Oracle ADF Migration 26
Tools 27
Summary 27

Introduction

Apache Beehive was the web application development framework of choice within the BEA developer community. Apache Beehive is a collection of open source technologies that together aimed to simplify Java EE programming. With Beehive, applications are developed based on JavaServer Pages (JSP) and an enhanced implementation of Apache Struts.

The Oracle Application Development Framework (ADF) is Oracle's strategic framework for building all classes of Java Enterprise applications, from the desktop to Rich Enterprise Applications (REA) running in a browser. ADF is a meta-framework that encompases a choice of technologies for building the view- and business service layer of an application. The possible technlogy choices on offer include Oracle ADF Faces Rich Client—a set of AJAX enabled JavaServer Faces components, ADF Business Components—a declarative object-relational mapping framework and the ADF Controller—an enhanced page flow controller for JavaServer Faces. This particular combination of technologies under the ADF umberalla have been selected by Oracle to build the next generaton of its own Oracle Business Applications software, also known as "Oracle Fusion". This paper is based around the same combination of technologies as we believe that this is the most productive and richest development stack for most Apache Beehive users to move to.

This whitepaper marks the start of a mini series of papers that each cover a selected focus area of the Apache Beehive to Oracle ADF developer skill transition.

In the series, this first paper specifically compares the view and controller layers of the Apache Beehive and the Oracle ADF frameworks to outline similarities and differences. The goal is to help developers that use Apache Beehive today to become familiar with the Oracle Application Development Framework and its associated technologies. Part II of this series covers the topic of comparing Apache Beehive Controls and Oracle ADF Data Controls for the general use case of accessing business services such as Web Service APIs.

Comparing Apples to Oranges

For this paper we assume the reader to be an experienced JavaServer Pages and Struts developer in the context of Apache Beehive. On the Oracle side, the Fusion developer stack of technologies within ADF is used for this comparison. Though we could have shown how Oracle ADF can be used to build web applications based on JavaServer Pages and Apache Struts, we decided against illustrating this option as technology has moved on considerably since Beehive was conceived. Instead we like to encourage Apache Beehive developers to adopt the current Java Enterprise Web UI standard, which is JavaServer Faces.

The goal of this paper is not to judge one suite of technologies better than the other, but to show Apache Beehive developers how and where their development skills fit into the Oracle ADF development process and what the options are for migrating existing Apache Beehive applications to ADF. In this context it is fair to compare the technologies despite their apparent differences.

Keep it Simple!

Both application development frameworks, Apache Beehive and Oracle ADF, follow the same mission: Simplicity. Java and Java EE development has become powerful over the last decade of its existence and while you can build everything, knowing everything has become harder with no end in sight. Service Oriented Architecture (SOA) and the new Rich Enterprise Applications (REA) paradigm add to the complexity of requirements voiced by the end users of modern web based business applications. To increase productivity, developers are looking

for frameworks, open source, vendor owned or those contained in the Java EE standard itself, to hide complexity without loosing flexibility and agility. The strategy for increased productivity in web application development is abstraction, reducing complexity by hiding implementation details from the application developer. Apache Beehive uses custom and system Java controls to abstract Java EE resources like EJB, JMS and JDBC, exposing a simplified JavaBean API to the application developer. In Oracle ADF, the same functionality is provided by the generic Data Controls mechanism. Another example of abstraction is the use of Java annotations in Apache Beehive and the metadata binding layer in JSR-227. Both approaches provide developers with a handle to the resources that they need to read and write to without exposing them to too many implementation details. The same theme continues on the view layer with the Apache NetUI taglib project and the Oracle ADF Faces Rich Client components.

From the perspective of a Beehive developer today, Oracle ADF seems to be doing exactly what Apache Beehive was good at: turning complex tasks into easy development steps. It is the technology used to accomplish the mission that is different between the two frameworks, which is the next topic to cover in this article.

Architecture

Apache Beehive uses the following core technology components to build web applications

- Apache Struts as the underlying, though extended, controller that handles page navigation and keeps the controller state
- NetUI, a JavaServer Page tag library to bind user interface components to their data without coupling the presentation to the navigation controls. NetUI provides a page navigation model, an enhaced contoller based on Struts, that also allows to build reusable page flows that can be nested within other flows.
- Java and system controllers to access Java EE resources
- Java annotations, as defined by JSR 175, that are used to identify Java methods in the controller classes as actions or exception handlers.
- JavaServer Faces support as an alternative component renderer using the NetUI controller

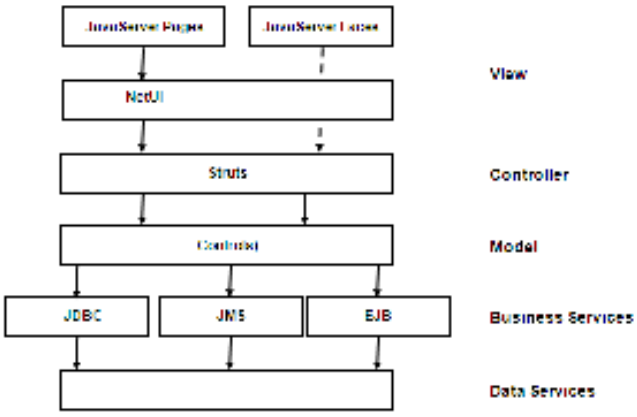


Figure 1: Apache Beehive architecture

In Oracle ADF, using the recommended "Fusion Development" stack, the core technology components are

- ADF Data Controls, the control that implements the business service access for a particular Java EE service, including web services
- ADF Bindings, the API exposed to the application developer, which is used to bind JavaServer Faces UI components to the business data
- ADF Faces Rich Client, an Ajax enabled JavaServer Faces component framework and tag library to build sophisticated web user interfaces
- ADF Faces Rich Client Data Visualization Tools (DVT), a first class JavaServer Faces UI component set to display data in graphs, to implement map views and to display gantt charts
- ADF mobile, a JavaServer Faces component set to display ADF Fusion applications on mobile devices like phone and PDA
- ADF Controller, an extension to the JavaServer Faces controller that enables reuse of an existing page and task flow, declarative transaction control and sub views.

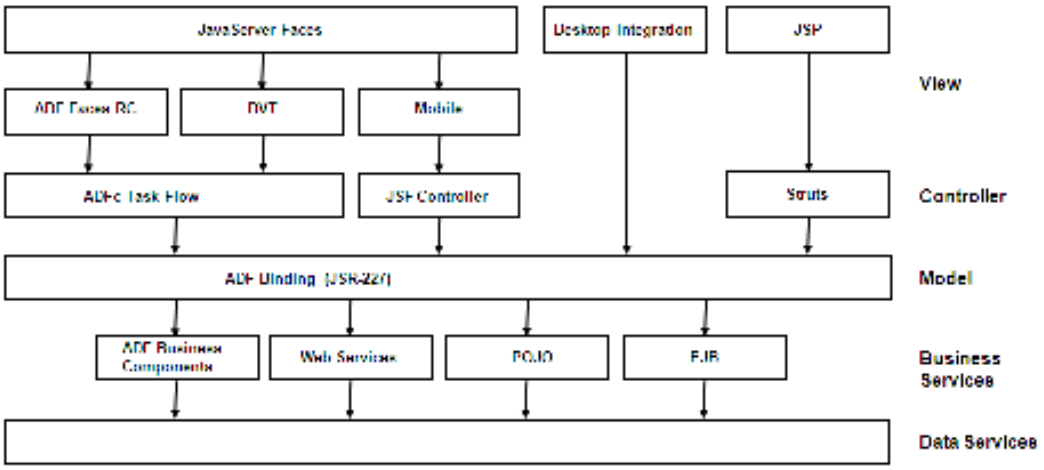


Figure 2: Oracle ADF architecture

Form Beans

A common usecase for application users that work within an application is to read, select, delete and edit data. Apache Beehive uses a form bean to hold form data and to write it back to the business service upon submit. A form bean is a Plain Old Java Object (POJO) that exposes getter and setter methods for its properties, which then are referenced from the input components.

Beehive Usecase

In the example below, a form bean represents a search form that exposes a single property for the search string. The form bean may be developed as a static class so it can be included in the Apache Beehive Controller class, or alternatively as a stand-alone Java class.

```
@Jpf.FormBean()
public static class SearchForm implements Serializable
{
  static final long serialVersionUID = 1L;
```

```
  private String _searchText;
  @Jpf.ValidatableProperty(
      validateRequired=@Jpf.ValidateRequired()
  )
  public String getSearchText()
  {
    return _searchText;
  }
  public void setSearchText(String st)
  {
    _searchText = st;
  }
}
```

The bean above contains a Java annotation that enforces an exception to be thrown when the input field is left empty. As you can see, the form field determines the constraints that exist for an input form.

The JSP page content to invoke the search and fill in the form, looks as follows

```
<netui:form action="doSearch">
  Search: <netui:textBox dataSource="actionForm.searchText"/>
  <br/>
    <netui:button>OK</netui:button>
    <netui:button action="cancelSearch">cancel</netui:button>
</netui:form>
```

The doSearch action is configured in the controller class and expects an argument of type SearchForm. This then also is how the framework accesses the form data to execute in the query. A major difference between application development in JavaServer Faces and Apache Beehive that you spot from this example is the integration of HTML markup in the layout definition.

The Oracle ADF and ADF Faces way

Using JavaServer Faces and the ADF Faces Rich client components, form input fields can be bound to managed beans or the ADF binding layer. Managed beans are POJO classes that are manage by the standard JavaServer Faces framework. In a similar way to Apache Beehive, the form input field is bound to the managed bean with Expression Language (EL). However, the other, more common option in Oracle ADF is to bind user

input components to attribute bindings of the ADF binding container. The ADF binding container is a runtime instance of the metadata binding layer and exposes a consistent programming interfaces for only those attributes and methods of a data control that are used on the current page. This matches the function of the form bean in Apache Beehive. The primary difference in this case is that the developer defines the required meta data through a drag and drop operation in the visual page editor, rather than through the writing of code.

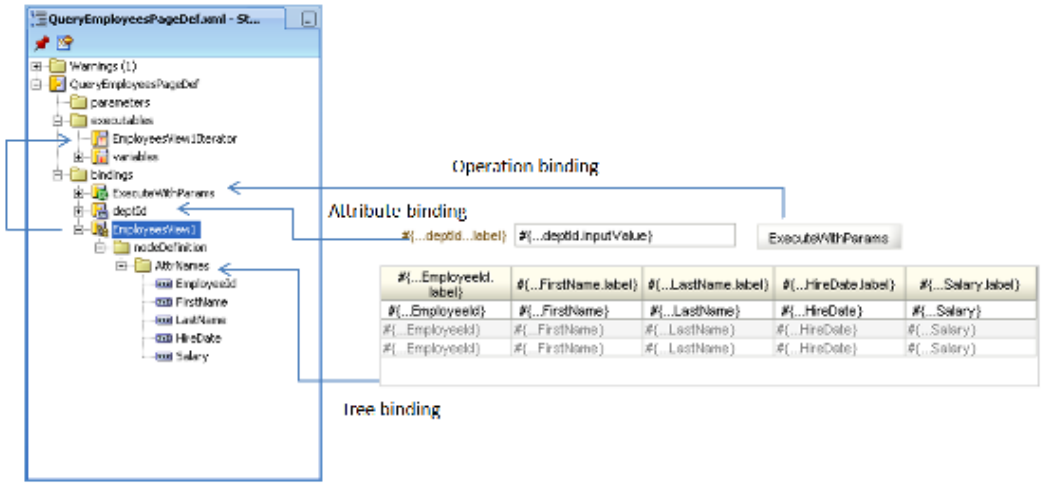


Figure 3: ADF binding referenced from an ADF Faces Rich Client page

Using the "bindings" object reference that exists at runtime to access the ADF binding definition of a page, user input components work with ADF as if working with a managed bean – or form bean in Apache Beehive. The difference is that the ADF binding layer doesn't hold the data but references the business service model.

The page source that contains the EL references to the binding layer looks as follows

```
1.  <af:form>
2.    <af:panelFormLayout>
        <af:panelGroupLayout layout="horizontal">
3.        <af:inputText
            value="#{bindings.deptId.inputValue}"
            label="#{bindings.deptId.hints.label}"
```

```
            required="true"
            <f:validator
            binding="#{bindings.deptId.validator}"/>
            <af:convertNumber groupingUsed="false"
              pattern="#{bindings.deptId.format}"/>
          </af:inputText>
4.       <af:commandButton
          actionListener=
            "#{bindings.ExecuteWithParams.execute}"
          text="ExecuteWithParams"
          disabled="#{!bindings.ExecuteWithParams.enabled}"
          partialTriggers="table1"/>
5.      </af:panelGroupLayout>
      </af:panelFormLayout>
6.    <af:table
        value="#{bindings.EmployeesView1.collectionModel}"
        var="row" …
```

How does it work?

1. The *af:form* element in ADF Faces Rich Client demarks an area of form components that is submitted to the server when an action component – such as a button – in the form is pressed
2. The *af:panelFormLayout* component arranges the input fields in columns and rows with the form field label right aligned to the left or on top of the component. Layouts in JavaServer Faces are not defined through markup but component containers. This is similar to how Swing places its component on a Swing panel.
3. The *af:inputText* component renders the HTML input element. In addition it provides properties for the required field evaluation, to set the component label and value, as well as information for the width and height. All properties can be set through Expression Language, which can point to the ADF binding to read property values from the model, or a resource bundle for labels and strings.
4. The *af:commandButton* renders the HTML action component in the generated browser output. It provides options for the developer to specify the size, the enabled and disabled state, the label and many properties more. Most important are the action and actionListener properties that developers use to execute methods on the model and to perform navigation.

5. The `af:table` component iterates over the rows contained in a RowSet and renders as a read only table or editable table. In the example above, the table is bound to the ADF binding layer to access the tree binding used to query the data from the business service.

Page Flow

In traditional web applications such as those created using Apache Beehive, a business process is modeled such that the application user navigates from one page to the next until he or she reaches the end of a task. Once a task is finished, another is started, also requiring the user to navigate from one page to another to update the business model and to query new data to be displayed in the browser.

In modern Rich Enterprise Applications, the term “page flow” no longer properly describes the options that are used for navigation within an application. User interaction within an application can lead to a refresh of small parts of a page only without re-rendering the page as a whole. In addition, navigation, like in a wizard, may be performed on the page, in which case the parent page behaves like a shell for the nested page flow.

In spite of the different requirements for the controllers between the traditional Web model and the modern AJAX model, from the perspective of a developer, the Apache Beehive controller and the Oracle ADF Task Flow controller have a great deal of similarity.

Navigation in Apache Beehive

As mentioned earlier, navigation in Beehive starts on a page and results in the rendering of a target page. The visual page flow development environment in Oracle Workshop for Weblogic allows developers to declaratively configure the controller for a specific flow.

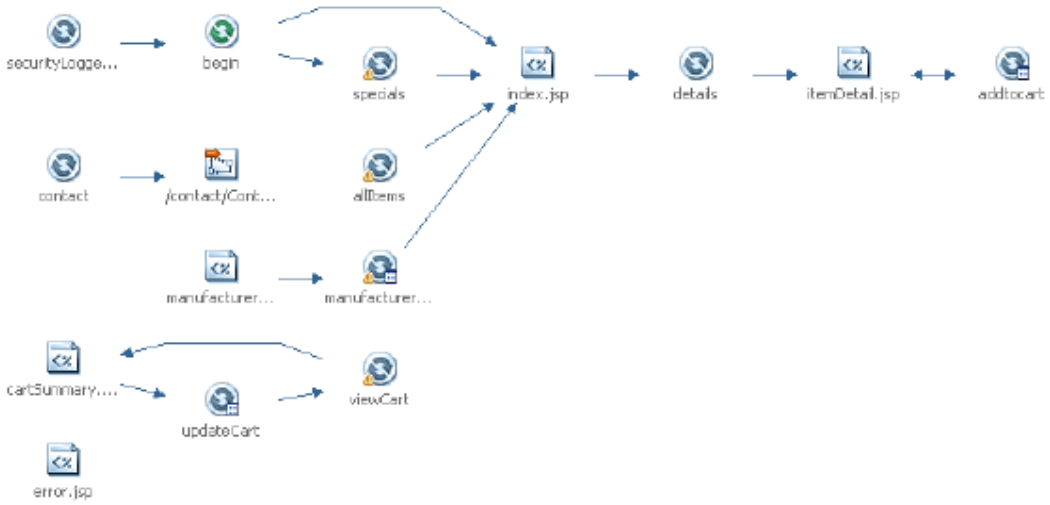


Figure 4: Apache Beehive page flow in Workshop for Weblogic

Page flows in Apache Beehive are defined by controller classes, with a single class managing any number of JSP pages.

The default controller class that is located in the default package is called Controller.java. Like any controller class, Controller.java extends `PageFlowController`, which is located in `the org.apache.beehive.netui.pageflow package`. In Apache Beehive, the NetUI controller performs page navigation and contains the view layer logic.

The controller navigation cases, the description of how the application flow navigates from one page to the next, are defined through Java annotations and Java methods in the controller class. Controllers also decide which resources are accessed within an application.

As shown in Figure 4, the development environment for Apache Beehive provides a visual development environment for composing page, sub-flow and method navigation. The visual designtime is synchronized with the controller Java code view so that changes applied to one update the other.

A third view is the Page Flow Explorer view that shows a hierarchical representation of the controller definition (Figure 5).

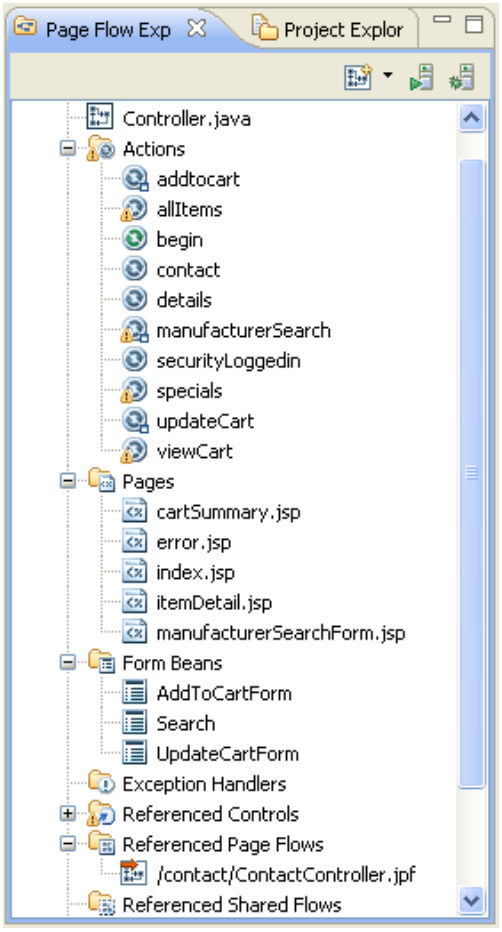


Figure 5: Page Flow Explorer view in Workshop for Weblogic

Apache Beehive: Simple Action

Apache Beehive defines two types of navigation cases, *simple action* and *method action*. To define a simple action, developers use the @Jpf.SimpleAction annotation on top of the controller class definition.

```
@Jpf.Controller(
    simpleActions={
        @Jpf.SimpleAction( name="someName",
            path="somePage.jsp", [...other properties...] )
    }
)
public class Controller
{
  ...
}
```

Simple actions can handle navigation, form submission, and form validation. Simple actions can also have conditions expressed in JSTL to continue navigation based on the outcome of an expression, for example a request parameter evaluation. This allows a declarative routing, a concept that also exists in ADF in the form of explicit router activities.

The definitions of simple actions are written to the struts-config.xml file at compile time. So in the end, all comes together in this standardized form of metadata.

Apache Beehive: Action

Method actions are similar to simple actions, except that they are associated with a method that returns a Forward to determine the next navigation stop in the Beehive page flow.

```
@Jpf.Action(forwards = {
   @Jpf.Forward(name = "success",
                path = "viewCart.do")

    })
    protected Forward updateCart(Controller.UpdateCartForm
form)
   {
      for (int i = 0; i < form.getId().length; i++)
      {
         Integer qty = new Integer(form.getQuantity()[i]);
         globalApp.shoppingCart.updateItem(
                    form.getId()[i],
```

```
                    form.getColor()[i],
                    form.getSize()[i], qty);
        }
    return new Forward("success");
}
```

Upon successful completion of the view layer logic navigation continues with the viewCart action. Multiple navigation targets can be defined using the `@Jpf.Forward` annotation.

Navigation in Oracle ADF

Navigation in Oracle Fusion web applications is performed by the ADF Controller (ADFc), which extends the default JavaServer Faces navigation model, offering a more complete and modular approach for designing page flows in JavaServer Faces and ADF Faces Rich Client applications. In contrast to the standard JavaServer Faces page navigation model, ADFc allows developers to build reusable application flows that are declaratively added to any consuming page flow. Because of its ability to navigate to non-visual targets like method calls or router activities and the ability to leverage sub flows, ADFc uses the term Task Flow instead of page flow to describe its navigation model.

In Apache Beehive NetUI there is a strong coupling between the view layer logic and the definition of page navigation cases since both share the same source file. In ADFc, and in JavaServer Faces in general, there is a clean separation between the navigation and its supporting business logic, the former in XML metadata and the latter in Java code.

The Oracle ADF Controller supports two types of Task Flows bounded and unbounded.

- **Unbounded Task Flow** the root application flow that is used to define entry points into an application. Pages on an unbounded Task Flow are directly accessible from a browser URL, are bookmarkable and can be used to automatically generate hierarchical page menus. View layer logic is stored in managed beans that are instantiated and dismissed by the JavaServer Faces dependent as required.
- **Bounded Task Flow** as in Apache Beehive, bounded Task Flows are used to build sub flows that are called from other flows. The role of bounded Task Flows is to enable reuse and modularization in application development by defining clearly

defined boundaries for a process to run in and to allow transactional isolation within the flow. As in Beehive, there is no difference in the flow setup between the two types of flow. Bounded Task Flows also have the additional capability of being able to orchestrate page fragments as well as whole pages, in which case they execute within a region of a containing page.

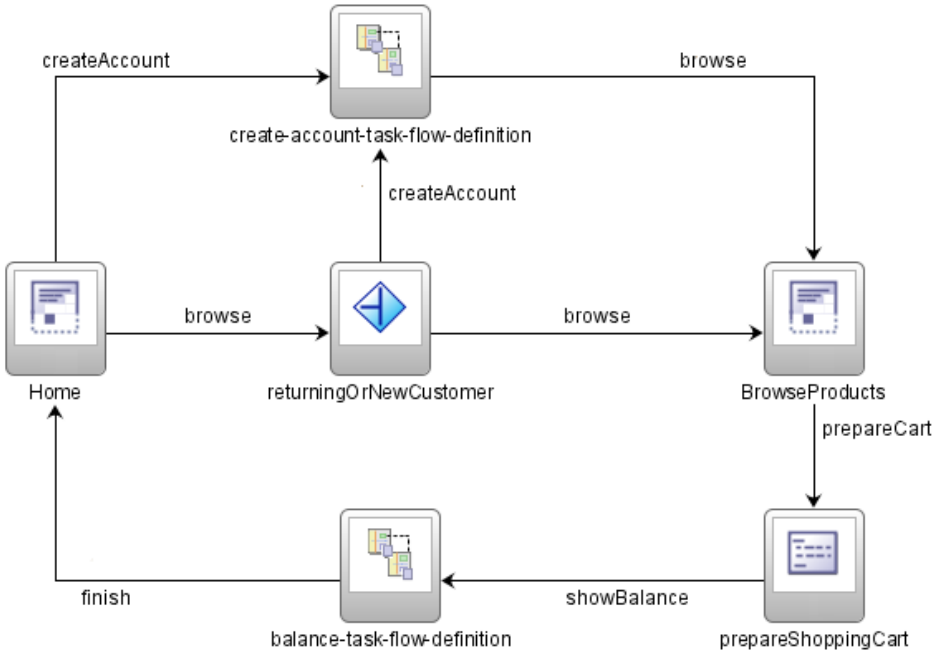


Figure 6: Control flow in Oracle ADF

As an Apache Beehive developer you usually don't see where and how navigation is defined at runtime. At design time you use the controller class to define navigation rules in Java through annotations. At compile time these controller definitions are written as actions into the struts-config.xml file, where they are referenced at runtime. In ADF, the metadata used at designtime is the same data used at runtime, there is no intermediate compile or generation step.

The Task Flow shown in Figure 6 contains method invocations, router decision and sub flow calls. Method invocations use Expression Language to reference JavaServer Faces managed bean methods. Using the method activity element, input parameters can be declarative passed in to the method and, if it exists, a return value written back to a bean or memory attribute of the calling flow. Both, references to input arguments and output objects use Expression Language making it easy for developers to read and understand the diagram. Unlike Apache Beehive, in which the full understanding of the page flow comes by reading the controller Java class, ADF provides this information in property dialogs.

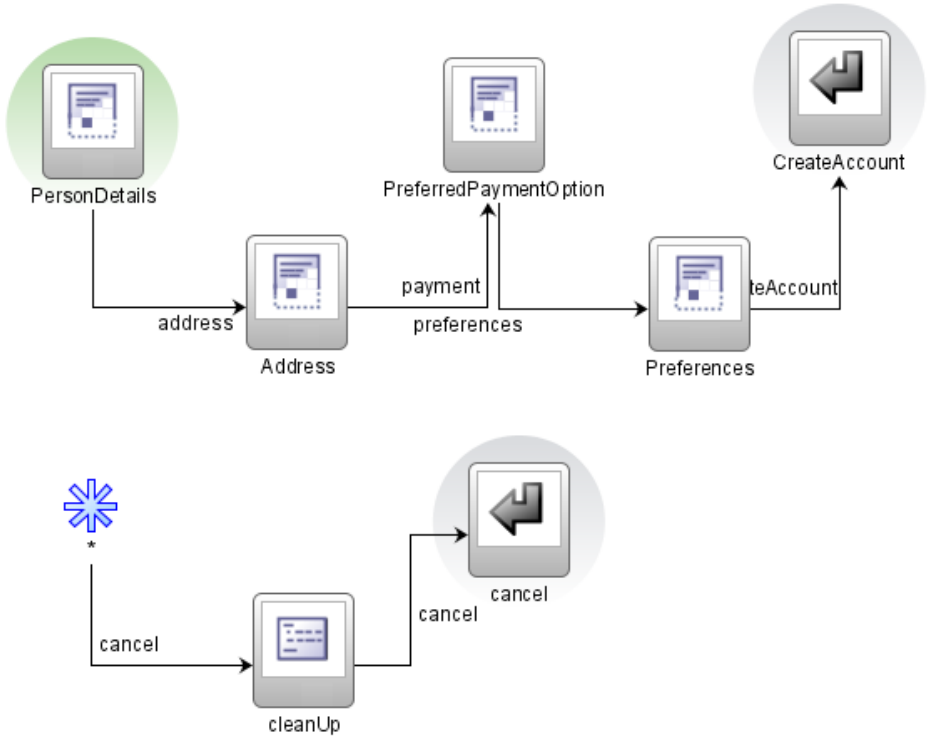


Figure 7: Create account bounded sub-flow

The metadata example below describes the control flow shown in Figure 6, in which the navigation from the "Home" page to the "returningOrNewCustomer" router activity occurs when the action outcome is returned as "browse".

```
<control-flow-rule>
  <from-activity-id>Home</from-activity-id>
    <control-flow-case>
    <from-outcome>browse</from-outcome>
    <to-activity-id>returningOrNewCustomer</to-activity-id>
  </control-flow-case>
```

The router decision in ADF Task Flow replaces the Apache Beehive `@Forward` annotation. A router uses Expression Language to access context parameter or managed bean methods to decide which branch of a navigation to follow. Though the option exists in JavaServer Faces to use a managed bean reference in a command button action to evaluate the navigation target a router offers a more declarative, readable and less error prone alternative.

The Task Flow call activity element that is added to the Task Flow diagram, references a sub flow. Double clicking onto a nested Task Flow in the diagram opens the nested Task Flow definition.

Sub-flows in ADF (see Figure 7) have a defined entry point that, at designtime, is highlighted with a green circular halo. In the example of an account creation sub-flow, the user is routed from one wizard page to the next until he or she confirms the new account. At any time, the user may cancel the account creation process. To handle this, a wildcard case navigation rule is defined that navigates the request to the designated exit component for this usecase. A method activity may be used to send a confirmation mail to the user or to clean up non-transactional system entries. The return activity can be configured such that upon leaving the Task Flow, the existing user transaction is automatically rolled back.

Like Apache Beehive in Workshop for Weblogic, Task Flows in JDeveloper can be developed in multiple ways:

- **Visual diagrammer** allows developers to drag and drop navigation elements from the Oracle JDeveloper component palette to design the flow. Using method

activities, the method is referenced from a managed bean or dragged from the Data Control palette to include functions that are exposed on the business service.

- **Structure Window** shows a hierarchical view of the elements that build the Task Flow definition. Additional elements can be added by dragging them from the component palette or using the right mouse menu
- **Overview dialog** shows the Task Flow definition file in a categorized view that allows quick access to a specific information in the file
- **Property Inspector** all elements that are used to define an element on a Task Flow are available as properties in the Property Inspector. Setting a property adds the configuration element, an XML element or attribute, to the Task Flow.

For an application developer that uses Apache Beehive in the past, the main difference in working with ADF Task Flow is that the metadata configuration files exist at designtime as well. In addition to this, you don't deal with one or many Java classes that hold the controller logic, but with reusable managed bean classes that are loaded on demand by the JavaServer Faces framework.

Note: As an Apache Beehive developer you are used to adding business logic into the controller definition. The same task is handled by managed beans in JavaServer Faces. Managed beans are general purpose beans that can be used for all sorts of client logic. A managed bean in JavaServer Faces is a POJO class with a no-argument constructor that is configured in the faces-config.xml file, or a Task Flow configuration. The JSF framework manages the bean lifecycle, creating it on demand and destroying it as it goes out of scope.

Component events in ADF

Component events are raise by UI components to notify the system about user activities. Component events include events like value change, row selection, component resize, node disclosure and command actions.

In Apache Beehive, actions are raised on the form level and handled in the controller logic, in a coarse grained event notification which is not very suitable for today's very rich application interfaces. Component events in JSF are handled on a component level, which, in the case of ADF Faces Rich Client, doesn't always require a full page submit.

Developers use managed beans to add listener methods that the component references in its action or listener property using EL. The example below shows a command button that raises an action event to the server.

```
<af:commandButton text="Send mail"
                  action="#{myManagedBean.sendMail}"/>
```

The `sendMail` method that is referenced in the managed bean has the following signature

```
Public String sendMail(){
   // add custom code
  return "navigation_case_or_null"
}
```

The action method returns a string value that is used by the navigation handler to determine the next navigation target. If no navigation is needed then the returned value should be set to null.

Listener events like ValueChange or ActionEvent provide additional information about the object that raises the event and have a different method signature. The example below is similar to the first example but uses an ActionListener to respond to the action event.

```
<af:commandButton text="Send mail"
                actionListener="#{myManagedBean.doSendMail}"/>
```

The managed bean method signature contains an argument that takes an object of the event type as the argument

```
Public void doSendMail(ActionEvent event){
   // add custom code
   // access event source from event object
}
```

It goes beyond the scope of this white paper to discuss the different types of listeners and where they apply to. The information to take away from the code examples above is that JavaServer Faces provides a JavaBean event architecture that allows developers to execute client logic in response to a user interaction on a component-by-component rather than a form-by-form basis.

Exception handling

The last thing you want an application user to stare at is a stack trace displayed in the browser screen caused by an unexpected error that occurred.

To handle this in Java EE, Apache Beehive and Oracle ADF all provide facilities to graceful redirect an application. The default error handling facility in Java EE is the error handler that can be specified in the web.xml descriptor file. Usually the container error handling mechanism is used for http errors and Java class exceptions. The downside of having the container taking care of the error handling process is that the container usually knows very little about the application the user runs when hitting the problem.

To allow exceptions to be handled gracefully within the context of the running application, Apache Beehive uses another controller annotation class, `@Jpf.Catch`.

```
@Jpf.Catch(type = Exception.class,
           method = "handleException"),
@Jpf.Catch(type = PageFlowException.class,
           method = "handlePageFlowException"),
@Jpf.Catch(type = NotLoggedInException.class,
           method = "handleNotLoggedInException")
```

Exceptions are either forwarded to a method or a page in Apache Beehive. This allows application developers to use the container managed Exception handling as a last resort.

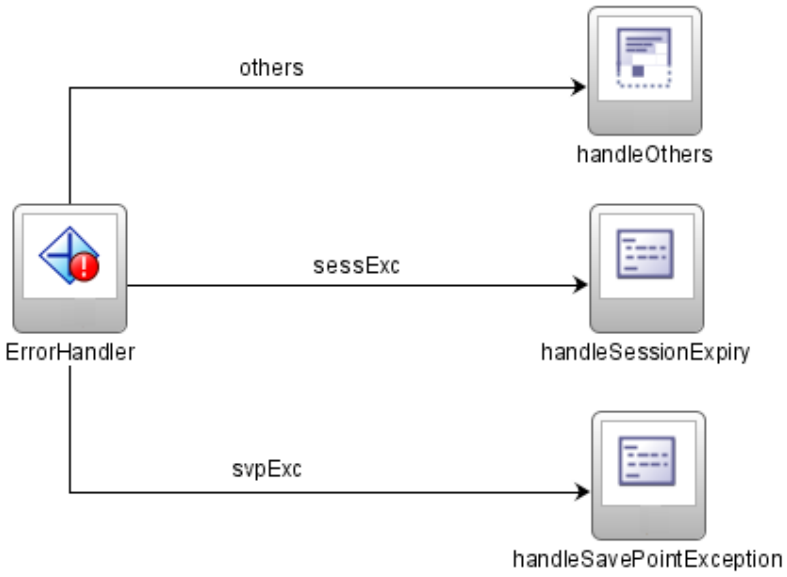


Figure 8: Declarative exception handling

Oracle ADF follows a similar approach and also allows developers to declaratively specify the component to handle exceptions. The main difference is that the Exception information is available from the `ControllerContext` framework class, which knows of all exceptions that the ADF controller catches.

Any Task Flow element in a Task Flow definition can be assigned to be the exception handler. In Figure 8, a router activity is called whenever an exception is detected by the ADF Task Flow. The router then evaluates the type of exception and the error message provided to route the error handling to a method or page. If an error is caught in a bounded sub-flow, but not handled there, then the exception propagates to the parent Task Flow. This also is the behavior of Apache Beehive exception handling.

Developers who need to handle exceptions where they occur, for example if the exception is caused by a method call in java, use a try-catch block to locally handle the error. In Beehive, since the controller also contains the client logic, the `@Jpf.Catch` annotation would be assigned to the forward action to handle exceptions on the same level.

Lifecycle

The sequence of steps performed in the process of handling a request is referred to as the lifecycle. In traditional web applications, a request goes hand in hand with a page refresh or page navigation. However, this is not necessarily the case in modern, rich Internet applications that use asynchronous request processing. In such applications the request lifecycle still exists but will execute more frequently and only with small subsets of the page.

Apache Beehive

The request cycle in Apache Beehive is an extended version of request-parse-render, which is the default lifecycle in JavaServer Pages applications. The extensions to this model are

- Two call back methods that act as "phase listeners" for the invocation of an action. Developers add the `beforeAction` and `afterAction` methods to the controller class to respond action calls within a page flow
- Two lifecycle methods `onCreate` and `onDestroy` exist that developers use to respond to the controller instantiation and dismiss
- A special `onExitNesting` method callback that notifies the application about the return from a sub-flow

Oracle ADF

The Oracle ADF lifecycle extends the JavaServer Faces lifecycle with the additional phases of the ADF model preparation and processing. Application developers can listen and respond to the lifecycle phases or, for individual page or global, configure their own lifecycle to change the default behavior. The request stages of JavaServer Faces are

- **Restore View** The first phase within the lifecycle receives the page request from the JavaServer Faces servlet and attempts to find and restore the sever side component state, a set of in memory objects that build the server side hierarchical page structure. If the view cannot be found as identified by the requested view Id, a new view is created on the server.

- **Apply Request Values** During this second phase, the request values are read and converted to the data type expected by the underlying model attribute. The values are accessed from the UI component's submitted value and passed to the local value.
- **Process Validations** The components are validated for missing data or data that doesn't conform to the expected component format. At the end of this phase, value change events are raised.
- **Update Model Values** During this phase, the JavaServer Faces model is updated with the validated value of the UI component local value. The local value on the component is discarded right after. The JSF model can be a managed bean or, in the Oracle ADF case, the binding container that is associated with the current page.
- **Invoke Application** During this phase, all application level events, such as form submits, are executed. This phase also executes the view layer logic that is stored in action listeners. If there is a navigation case associated with a command action return value then page navigation is performed to this point in time.
- **Render Response -** This last phase prepares the page view for display on the client device.

Additional ADF lifecycle phases are integrated in the overall JavaServer Faces lifecycle and provide more granular hooks into the request lifecycle.

As in Apache Beehive, application developers can listen and respond to these phases in Java in where they have access to a context object that provides all of the information that they might require about the lifecycle at that point.

NetUI form controls vs. ADF Faces Rich Client

The two tag libraries to look at for building web user interfaces are NetUI for Apache Beehive and ADF Faces Rich client and DVT as the preferred Oracle ADF view layers.

NetUI

The NetUI tag library consists of three parts: a core library, data grid and templates. The most commonly used library is the core library that is prefixed with the “netui” namespace when added to a JavaServer Page. The NetUI core components match to the list of HTML input and output components and the form element.

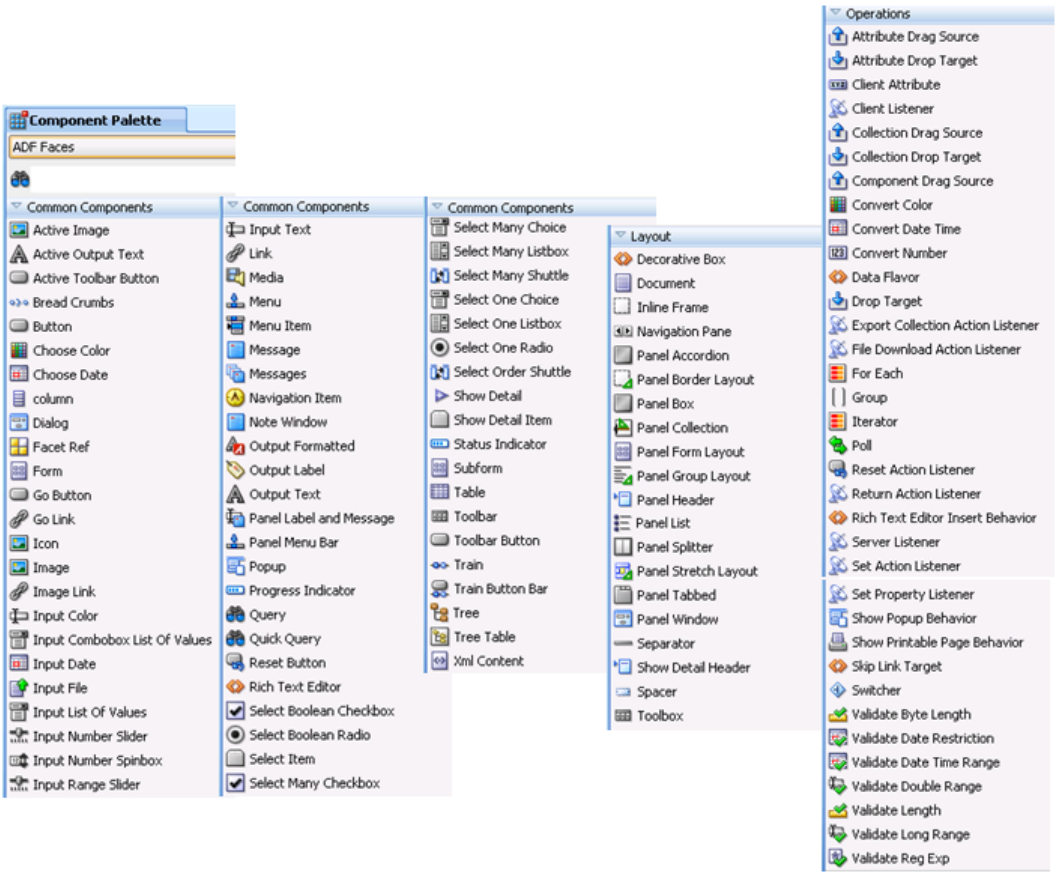
NetUI Form Control	**Binding Data Type**
<netui:anchor>	None
<netui:button>	None
<netui:checkBox>	boolean or java.lang.Boolean or java.lang.String
<netui:checkBoxGroup>	java.lang.String[]
<netui:checkBoxOption>	None, See the checkBoxGroup tag
<netui:fileUpload>	org.apache.struts.upload.FormFile
<netui:form>	None
<netui:hidden>	java.lang.String or java.lang.String[]
<netui:imageButton>	None
<netui:radioButtonGroup>	java.lang.String
<netui:radioButtonOption>	None, See the radioButtonGroup tag
<netui:select>	java.lang.String or java.lang.String[]
<netui:selectOption>	None, See the select tag
<netui:textArea>	java.lang.String[]
<netui:textBox>	java.lang.String[]

Table 1: NetUI Form Control (src: beehive.apache.org)

Developers use these tag elements in a markup driven layout to build their web pages. Data is accessed from the components using Expression Language and Action form attribute references.

ADF Faces Rich Client and DVT

The ADF Faces Rich Client and Data Visualization Tools (DVT) tag libraries provide developers with a choice of more than 150 simple and complex UI components that are grouped in three categories: core components, layout containers and operations.



- **Common components** The list of components includes all UI components that are available in HTML plus a list of more complex input controls like shuttle, calendar, slider, table, tree, tree table and many more.
- **Layout container** JavaServer Faces does not promote the use of HTML markup to layout components on a screen and instead uses layout components, which better integrate into the JSF architecture and philosophy of device independence. Layout components have one or many areas in which application developers can add components to and may also have a behavior, which is the way the layout arranges the added components. Like HTML markup elements, layout components can be nested. Layout containers also exhibit geometry management capabilities providing the developer with control over stretching and resizing of user interface elements.
- **Operations** Operation components provide declarative support for actions that otherwise a developer has to code for. Operation elements include drag and drop support, export of collections to Microsoft Excel, printable page behavior, validators and type converters, as well as client and server listeners that support JavaScript calls into the framework.

Working with ADF Faces components is fundamentally the same as working with the NetUI tag library except that more and richer components are available.

UI Data binding

Both, Apache Beehive and Oracle ADF use expressions to bind input data components to the model objects. Beehive uses the JavaServer Pages 2.0 standard EL expressions to bind values, as well as action form attribute references.

Oracle ADF Faces uses the JavaServer Faces 1.2 EL and no implicit object references. Value binding expressions either reference a property exposed on a managed bean, or, as it is more common when working with ADF, the ADF “bindings” object.

Though JavaServer Pages and JavaServer Faces use a unified Expression Language, there is difference in the usage. The most obvious difference between EL used in JSP and JSF is that the EL expression in JSP 2.0 starts with a “$” character whereas EL in JSF starts with a “#”.

EL that uses the "$" syntax executes the expressions eagerly, which means that the result is returned immediately when the page renders. Using the "#" syntax defers the expression evaluation to a point defined by the implementing technology. In general, JavaServer Faces uses deferred EL evaluation because of it multiple lifecycle phases in which events are handled. To ensure the model is prepared before the values are accessed by EL, it must defer EL evaluation until the appropriate point in the life cycle. Amongst other benefits, this allows pages to be lazily populated with data after they have been rendered.

The above doesn't change the way application developers build a web application because the IDE usually takes care of the differences mentioned above. However, as a developer coming from a JSP development background you should be aware of this difference when starting your application development with Oracle ADF and ADF Faces Rich Client components.

Business Service integration (controls)

Controls in Apache Beehive are JavaBean wrappers that expose a simplified API to the tools and application developers, hiding the complexity of the Java EE resource lookup. In the way that controls are architecture, they are comparable to data controls in Oracle ADF. The difference though is in the access, which in Oracle ADF is through the binding layer that exposes a consistent set of APIs that is accessible from Java and Expression Language. The second paper in this series compares Apache Beehive Controls and Oracle ADF Data Controls for the general use case of accessing business services such as Web Service APIs.

Apache Beehive to Oracle ADF Migration

There is no automated migration path from Apache Beehive to Oracle ADF and ADF Faces Rich Client, although the development gestures and required skills map nicely. There are platform specific coding constructs in Apache Beehive applications that don't map with Oracle ADF and ADF Faces. Even with a simplified migration path that uses a POJO model with ADF and ADF Faces Rich Client or JSP with Struts as the view layer, there still is no viable direct migration path.

Tools

The Java IDE for Oracle ADF and Java EE REA is Oracle JDeveloper 11g. Though it is possible to configure Oracle ADF and ADF Faces Rich Client components with the Eclipse design time, for an end-to-end declarative development experience Oracle JDeveloper provides the optimum development environment.

Summary

Apache Beehive deserves all the praise it received for the quality of the framework and the productivity gain it provided to Struts application developers. Oracle ADF is a mature and road tested Java EE development. All functionality available in Apache Beehive also exists in Oracle ADF of Oracle JDeveloper 11*g*. In fact Oracle ADF goes beyond Apache Beehive in many ways. For example, ADF Faces RC and the ADF controller integrate Ajax capabilities to JavaServer Faces for developers to build REA applications in Java EE without having to code JavaScript. Though there is no automated migration path for existing Apache Beehive applications to Oracle ADF, Apache Beehive developers will find the development experience in Oracle ADF to be similar to that which they already know.

Oracle ADF for Apache Beehive Developers

April 2009

Author: Frank Nimphius

Contributing Authors: Duncan Mills

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

Oracle is committed to developing practices and products that help protect the environment