ADF Code Corner

Oracle JDeveloper OTN Harvest 11 / 2010

twitter.com/adfcodecorner

Abstract:

The Oracle JDeveloper forum is in the Top 5 of the most active forums on the Oracle Technology Network (OTN). The number of questions and answers published on the forum is steadily increasing with the growing interest in and adoption of the Oracle Application Development Framework (ADF).

The ADF Code Corner "Oracle JDeveloper OTN Harvest" series is a monthly summary of selected topics posted on the OTN Oracle JDeveloper forum. It is an effort to turn knowledge exchange into an interesting read for developers who enjoy harvesting little nuggets of wisdom.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
30-NOV-2010

Oracle ADF Code Corner OTN Harvest is a monthly blog series that publishes how-to tips and information around Oracle JDeveloper and Oracle ADF.

Disclaimer: ADF Code Corner OTN Harvest is a blogging effort according to the Oracle blogging policies. It is not an official Oracle publication. All samples and code snippets are provided "as is" with no guarantee for future upgrades or error correction. No support can be given through Oracle customer support.

If you have questions, please post them to the Oracle OTN JDeveloper forum:
http://forums.oracle.com/forums/forum.jspa?forumID=83

November 2010 Issue – Table of Content

How-to skin a UI specific to a browser Type and Version ?

Browsers are different and there is not much you can do against this nugget of wisdom. Sometimes it happens, that the screen browsers render are different in a very minor point, like the padding of a table or the space left between two components. Using skinning, you can define specific style (CSS) definitions based on the browser type and – if the browser is inconsistent between versions – the browser version. The example below shows what you need to put into a CSS file used by a skin definition to render the colors or size of an ADF Faces quick query component differently from the rest if the browser type is IE and the version on 7.

```
@agent ie (version: 7) {
  af|quickQuery::label{ ... css here ... }
}
```

How-to define localized access keys for command components

Command components in ADF Faces have a "textAndAccessKey" property that developers can use to specify access keys, a character that when pressed in combination with the alt-key invokes the component action. Similar, input components have a "labelAndAccessKey" property defined to specify an access key that puts the focus onto the component when used. To define an access key, you use "&" within the label or text string, just in front of the character to become the access key character.

For example, "&Ok" defines the access key to be alt+o. "Su&bmit", defines alt+b as the access key for a command button with the text string Submit.

The same notation "&" can be used when the text or label is read from a resource properties file, in which case the entry is

mypage.command.submitKey=&Submit;

mypage.command.cancelKey=&Cancel;

If you created a German translation of this resource file, the key definition may look as shown below

mypage.command. submitKey =&Bestaetigung;

mypage.command.cancelKey=&Abbrechen;

Note the difference however. The "C" in Cancel does not show in "Abbrechen". Therefore the access key will change for the translation. To ensure consistent access keys to be used, the only option developers have is to choose characters as access keys that exist in all translations.

If the internationalized strings and labels are read from a Java resource bundle, then, to define the access key, you use the ampersand "&" character only. The "&" string needs to be used in properties files only, not in Java.

How does auto-ppr work?

Auto-PP is a functionality of Active Data Services (ADS) in Oracle ADF Faces. Developers setting the ChangeEventPolicy property of the ADF binding – PageDef.xml – to ppr configure attribute and control

bindings that are used for input text components and trees to refresh when the binding layer changed, e.g in response to users navigating within a table. In fact auto-ppr is a binding based equivalent to setting the PartialTriggers property on a component. So how does it work and how does it know which component to refresh?

FacesCtrl* classes, which are ADF Faces specific extensions to the generic JUCtrl* binding classes have a method defined on its Def classes that check if the underlying iterator (the iterator the components bind to) has auto-ppr configured. If so, then the component is added to the partial target on the FacesContext when a change on the model (like row currency change in response to nagivation) is detected.

Note: In addition to updating the UI in response to users navigating in a collection, using ADF Business Components as a business service, auto-ppr can also be used to notify clients about RDBMS database change events, functionality Oracle customers often ask for.

af:subform, required fields and immediate="true"

The component tag documentation for the af:subform component states that ...

"the rich subform represents an independently submittable region of a page. The contents of a subform will only be validated (or otherwise processed) if a component inside of the subform is responsible for submitting the page or if the default attribute is set to true. This allows for comparatively fine-grained control of which components will be validated and pushed into the model without the compromises of using entirely separate form elements"

A known limitation, which is not yet documented, is for when input components within any of the sub forms of a page have their "immediate" property set to true. In this case validation is performed even if the component is not located in the submitted af:subform. The reason for this behavior is within the JSF lifecycle: Each af:subform tries to determine whether it has been submitted or not in a later part of the apply request phase. So if one of the input field components has its "immediate" property set to true, its validation is processed before the af:subform I is in has a chance to detect whether it has been submitted or not. As a result validation errors are shown. So when using af:subform, ensure none of the contained fields has its immediate property set to true.

How-to efficiently redirect an ADF Faces view using ADFc

ADF Faces developers use facesContex.getExternalContext().redirect(String) to issue a GET request to a JSF view. Using ADFc, the redirect URL should neither be read from the current UIView root directly or provided in the form /faces/<viewId name>. Instead, have the controller generating the redirect String for a specific viewId as shown below:

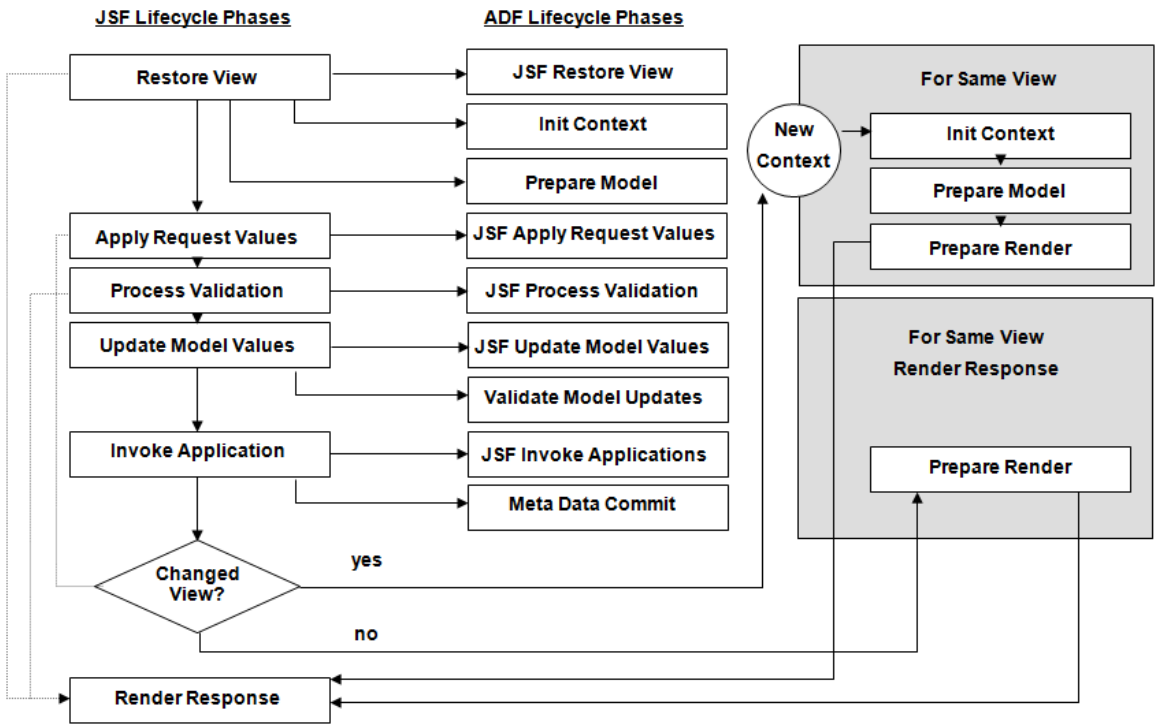
```
FacesContext fctx = FacesContext.getCurrentInstance();
ExternalContext ectx = fctx.getExternalContext();

String viewId = "... add viewId...."
ControllerContext controllerCtx = null;
controllerCtx = ControllerContext.getInstance();
String activityURL = controllerCtx.getGlobalViewActivityURL(viewId);
try{
   ectx.redirect(activityURL);
} catch (IOException e) {
```

```
  //Can't redirect
  e.printStackTrace();
}
```

Why? Because a redirect is a Get request and you don't want ADFc to treat it as a new application request but instead retrieve the internal controller state. For this you need the state toke in the redirect URL, which is what the code line above does

How-to configure an ADF Phase Listener and where to put the file

The Oracle ADF lifecycle integrates with the JavaServer Faces request lifecycle, adding all that it takes to set up the binding context, prepare the binding container, validate and update the ADF model, persist MDS changes and prepare the response.



Developers who need to listen and interact with the request cycle may use an ADF Phase Listener to do so. Unlike the Phase Listener you define in the faces-config.xml file, the ADF Phase Listener allows you to listen to the standard and the ADF phases.

The ADF Phase Listener is defined in Java – of course – and configured in the adf-settings.xml file you need to create. To create an ADF Phase Listener, all it takes is to start from a template like shown below and add your logic.

```
import oracle.adf.controller.v2.lifecycle.PagePhaseListener;

public class MyAdfListener implements PagePhaseListener{
  public MyAdfListener() { }
  public void afterPhase(PagePhaseEvent pagePhaseEvent) {
```

```
    //for example, to listen for the RESTORE_VIEW phase
       int RESTORE_VIEW_ID = PhaseId.RESTORE_VIEW.getOrdinal();
       if (pagePhaseEvent.getPhaseId() == RESTORE_VIEW_ID){
            //...
      }
  }
  public void beforePhase(PagePhaseEvent pagePhaseEvent) {
  }
}
```

As shown in the code sample above, you can listen for any phase you are interested in. Developers who want to globally change the lifecycle context use for example use the after RESTORE_VIEW phase for this.

To configure the adf-settings.xml file, create the file as shown below in the .adf\META-INF directory of your application

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
   <adfc-controller-config
      xmlns="http://xmlns.oracle.com/adf/controller/config">
    <lifecycle>
    <phase-listener>
        <listener-id>MyAdfListener</listener-id>
        <class>adf.sample. MyAdfListener </class>
   </phase-listener>
    </lifecycle>
  </adfc-controller-config>
</adf-settings>
```

Note: On OTN there has been some confusion of where to put the adf-settings.xml file best because the online documentation on OTN points out to create this file in a META-INF directory in the ViewLayer project SRC directory (which you need to create). This recommendation was a work around to a bug that caused problems with EAR files that contained two WAR files, each having projects that use the ADF Controller. To work properly, MDS – Meta Data Services – needs to have information about the web application root, which could be set in the adf-settings.xml file. Storing the adf-settings.xml file in the View project source directory allows developers to have two adf-settings.xml files with a setting for each application context. The bug causing this work around has been fixed in JDeveloper 11g 11.1.1.2 (PS2), so it should no longer be a problem to have two WAR files deployed in a single EAR file.

#{data} – to use or not to use ?

The #{data} expression allows developers to access the ADF `BindingContext` object that represents the content of the DataBindings.cpx file at runtime. In Oracle JDeveloper 11g, the `BindingContext` class has a static method *getCurrent* that also allows developers to access the `BindingContext` object so there is no need to use the #{data} expression anymore, at least not when access is required from Java in a managed bean.

In Oracle JDeveloper 10.1.3, there exists no static *getCurrent* method and #{data} and #{bindings} are the two options available to access the `BindingContext` at runtime. However, the binding context is not a good starting point for developers to access page bindings or invoke methods on a View Object or Application Module if you want the UI to refresh in response to an operation. For any business service access you want to perform in the context of working with ADF, the recommendation – and best practice is – to use the binding container #{bindings}. In JDeveloper 11g, to access the current active binding container from Java, you use the following code:

```
BindingContext bctx = BindingContext.getCurrent();
DCBindingContainer bindings =
                   (DCBindingContainer) bctx.getCurrentBindingsEntry()
```

In summary: In JDeveloper 10.1.3, use #{data} only to access DataControl data providers – like ADF Business Component Application Modules – for method invocations that are not meant to cause any change in the UI. For all other access to business services methods and attributes, use the #{bindings} expression or the Java code shown above.

An interesting observation of mine while monitoring the OTN forum is that many developers who use the `BindingContext` to invoke operations on ApplicationModules and ViewObjects also tend to cast the access to "Impl" classes, which also is not considered best practices in the context of ADF. Instead casting should be done for the AM or VO interface class.

When to use "createRootApplicationModule" in Oracle ADF

A common pattern observed on OTN is the use of the createRootApplicationModule method on an ADF Business Components Application Module (AM) to access a View Object. Usually a question and associated sample code is posted by developers struggling to see data updates displayed in ADF views. The *Fusion Developer's Guide for Oracle Application Development Framework* product documentation explains the use of the createRootApplicationModule method in the context of creating a command line test case for testing ADF Business Components models. Further it explicitly mentions that "… you typically won't need to write these two lines of code in the context of an ADF-based web or Swing application. The ADF Model data binding layer cooperates automatically with the ADF Business Components layer …"

A question posted on OTN started a discussion about if there is a use case for which the use of createRootApplicationModule in an ADF environment is appropriate. The outcome of the thread is that there doesn't seem to be a use case that requires this method call.

All you – as a developer – want to access in ADF Business Components is accessible through the ADF BC DataControl, which is exposed by the ADF BindingContext object. I think its save to call the use of "createRootApplicationModule" in the context of ADF an anti pattern that causes more problems than it solves. Best practices for accessing the ADF Business Components Application Module to execute, write or read objects are listed below

1. Avoid direct access to the ADF Business Components Application Module and View Object instances. Always work through the ADF binding layer in that you expose the AM functionality as a binding entry in the PageDef file. This handles the case in which managed beans are used to access functionality on the AM

2. If you need to access an Application Module outside of an ADF bound page, for example in a servlet, then make sure the Binding Context is established. For servlets you achieve this by configuring the ADF binding filter to be used when the custom servlet is requested. This configuration can be declaratively defined in the web.xml file. When the servlet is invoked, the ADF binding filter is called and ensures the BindingContext is created.
3. If a servlet needs to access a binding layer, or participate in ADF Security protection, you need to create a Pagedef.xml file for it and edit the Databindings.cpx file to contain a mapping that links the servlet path with the physical PageDef.xml file. Best is to copy and modify and existing entry (carefully though)
4. Invoking a servlet, which invokes the ADF binding filter, from an ADF Faces application creates a new Data Control frame. This however means that the servlet accesses a DataControl state and AM instance that is different from what the ADF Faces application "sees". You should be aware of this and – if you need to access the exact same state – access the Data Control Frame of the calling application from the BindingContext "findDataControlFrame(String)

   …

   ```
   import oracle.adf.model.BindingContext;
   import oracle.adf.model.DataControlFrame;
   import oracle.binding.DataControl;
   …
   BindingContext bctx = BindingContext.getCurrent();
   DataControlFrame dcframe = bctx.findDataControlFrame(name);
   DataControl dc = dcframe.findDataControl(dcName);
   ```

   You access the current Data Control frame in the calling application from the BindingContext in a call to BindingContext.getCurrent().getCurrentDataControlFrame(). The returned string can be saved in a session variable to make it accessible for servlets.

In general it is recommended to always work with the framework and not against it. Therefore I think we can rule out the use of the "createRootApplicationModule" method call in the context of an ADF. I also want to rule out the use of direct access to the DataControl if it is performed from a managed bean. In this case the recommendation is to expose the functionality to work with on the binding layer (unless you need to get information about the Data Control).

Note: Credits to where credits belong. Jan Vervecken and Dimitar Dimitrov did the ground work on the thread leading to this harvest entry

Deleting rows: getRowAtRangeIndex vs. Iterating of RowSet

The requirement in a post on the Oracle internal mailing list was about how to delete multiple rows based on a condition. In the example, the condition is a flag attribute that indicates whether to delete a row or not.

```
ViewObject vo = … < Get ViewObject > …
for (int i = 0; i < vo.getEstimatedRowCount(); i++) {
Row row = dcIterator.getRowAtRangeIndex(i);
```

```
String isSelected = row.getAttribute("IsChecked").toString();
if ((isSelected != null) && Boolean.parseBoolean(isSelected)) {
  row.remove();
}
```

While this code worked in that rows are deleted, it fails for successive row deletions. So if the row at index 11 and the row at index 12 are marked for deletion then only row 11 got deleted but not the row for index 12. The reason for this strange behavior has to do with the index.

When the row of row index 11 is removed from the row set, then row index 12 becomes 11. However, index 11 already has been processed and index 12, which before has been 13, is now on. This leaves the previous index 12 untouched.

A better option, which avoids this problem is to use

```
ViewObject vo = … < Get ViewObject > …
RowSetIterator rsi = vo.createRowSetIterator("deleteRowsRSI");
while (rsi.hasNext()) {
  Row row = rsi.next();
  String isSelected = row.getAttribute("IsChecked").toString();
  if ((isSelected != null) && Boolean.parseBoolean(isSelected)) {
    row.remove();
  }
}
rsi.closeRowSetIterator();
```

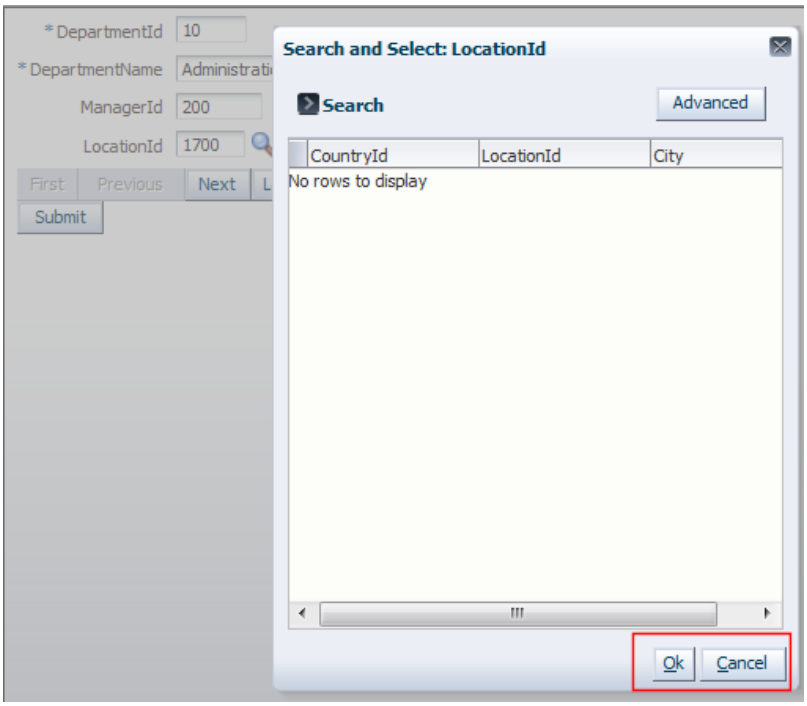
The code line above creates a new `RowSetIterator` that it uses to loop through the accessible rows. For this operation, creating a secondary `RowSetIterator` is not needed. However it appears to be good practice if you want to exclude the risk of disordering the default `RowSetIterator` with the action you perform.

Note: Since the use case is for mass update of rows (in this case mass-deletion) it is recommended to not perform this action from the ADF binding layer. Instead, create a method on the View Object and expose it as a client method. Then from ADF, invoke the method so all operations are performed on the business service.

Note: Credits go to Linda Inci from Oracle in Austria

How-to define access keys to buttons displayed in an LOV dialog

List of Value dialogs that are launched by an "af:inputListOValue" component have a "Ok" and "Cancel" button defined to return a selected value in the LOV list or to cancel the list with no selection. By default, the LOV dialog shows without access keys for the "Ok" and "Cancel" button. However, with a little bit of JavaScript and Java, you can add keyboard access functionality so the dialog shows as below



Note: Credits for this hint go to user "Ramprasadb" on OTN who initially proposed this solution

For the solution outlined in this summary, the assumption is made that the LOV is opened in an English locale, so that the strings to close the LOV display as "Ok" and "Cancel". For internationalized applications, where the button labels change, you need to make sure resource bundles are used to set the correct label with access key.

The process flow to set the access key definition at runtime is as follows:

1. A user clicks the LOV button next to an Input List of Values component. In response to this, the LOV component launch event is fired, which we listen to in a managed bean.
2. The launch event however is handled during the JSF Invoke Application phase, whereas the LOV displays during Render Response. Because of this, it is not possible to use a dialog launch listener to set the access keys for the buttons. Instead, the launch listener is used to invoke client side JavaScript.
3. The client side JavaScript addresses an af:serverListener component to queue a custom event in ADF Faces. The custom event invokes a managed bean method that accesses the list of value component to eventually set the access keys.

The following configuration is added to the ADF Faces page for the List of Value component and the server listener component

```
<af:form>
  <af:inputListOfValues id="lid1"
       ...
       launchPopupListener="#{TestBean.onLOVLaunch}"
       binding="#{TestBean.inputLov}">
       ...
```

```
    </af:inputListOfValues>
  …
  <af:serverListener type="inputLovAccessKeySetter"
                                method="#{TestBean.defineDialogAccessKey}"/>
</af:form>
```

Note the location of the af:serverListener component within the af:form component. The JavaScript client code will look up the form component to queue the server listener.

The managed bean has two method and a component binding defined.

The **component binding** is created for the af:inputListOfValues component's "binding" property and allows us to reference theLOV in the managed bean as "inputLOV", or – using its getter method – getInputLOV.

The **onLOVLaunch** method is called from the af:inputListOfValue "launchPopupListener" property and will use the MyFaces Trinidad Extended Renderkit Service class to send JavaScript to the client. Using this approach ensures the LOV dialog is displayed when the managed bean is called from the serverListener to set the access keys.

The **defineDialogAccessKey** method accesses the LOV component reference to search for the af:dialog instance within. When the dialog instance is found, setting the access key definitions is just a matter of two API calls. The managed bean code is shown below

```
import javax.faces.context.FacesContext;
import oracle.adf.view.rich.component.rich.RichDialog;
import oracle.adf.view.rich.component.rich.RichPopup;
import oracle.adf.view.rich.component.rich.input.RichInputListOfValues;
import oracle.adf.view.rich.context.AdfFacesContext;
import oracle.adf.view.rich.event.LaunchPopupEvent;
import oracle.adf.view.rich.render.ClientEvent;
import org.apache.myfaces.trinidad.render.ExtendedRenderKitService;
import org.apache.myfaces.trinidad.util.Service;

public class LovAccessKeyBean {

  private RichInputListOfValues inputLov = null;
  private FacesContext fctx = FacesContext.getCurrentInstance();

public LovAccessKeyBean() { }

//method called from the list of value dialog launch listener property.
public void onLOVLaunch(LaunchPopupEvent launchPopupEvent) {
 ExtendedRenderKitService erks = Service.getService(
                                              fctx.getRenderKit(),
                                              ExtendedRenderKitService.class);
 //create the JavaScript and invoke it on the client. The af:form id is "f1"
 StringBuffer scriptBuf = new StringBuffer();
 scriptBuf.append("var afForm = AdfPage.PAGE.findComponentByAbsoluteId(\"f1\");");
 scriptBuf.append("AdfCustomEvent.queue(afForm,\"inputLovAccessKeySetter\",{},true);");
 erks.addScript(fctx, scriptBuf.toString());
```

```
}

//method called from the server listener
public void defineDialogAccessKey(ClientEvent ce){
 //get the LOV component binding reference
  RichInputListOfValues lov = this.getInputLov();
   String id = lov.getClientId(fctx);
   //the LOV popup ID is defined by the LOV component ID plus "lovPopupId". Reading the LOV
   //component client Id from the component directly ensures the code also works when the LOV field
   //is located in a naming container
   String popupId = id+"lovPopupId";
   RichPopup popup = (RichPopup) lov.findComponent(popupId);
   RichDialog lovDialog = (RichDialog) popup.getChildren().get(0);

   //the ampersand '&' marks the character that users will be able to use in combination with the
   //alt key to invoke the command button. So pressing alt+o invokes the OK button
   lovDialog.setCancelTextAndAccessKey("&Cancel");
   lovDialog.setAffirmativeTextAndAccessKey("&Ok");
   //refresh the LOV to show the buttons
   AdfFacesContext.getCurrentInstance().addPartialTarget(lovDialog);
 }

 //component bindings
 public void setInputLov(RichInputListOfValues inputLov) {
    this.inputLov = inputLov;
 }

  public RichInputListOfValues getInputLov() {
   return inputLov;
  }
}
```

Note: To make this solution work with pages that have multiple LOV's defined, you need to pass the component Id of the LOV with the JavaScript function. For example, the JavaScript invoked by the Extended Renderkit Service may have a line scriptBuffer.append("var compId ="+ this.getInputLov().getClientId()); This then can be added to the AdfCustomEvent.queue event call as {lovcomp:compId}. The "lovcomp" parameter is accessible from the clientEvent object received by the defineDialogAccessKey method.

How-to filter user data input in a text field

To filter user entries in an input text field - e.g. allowing numeric entries only - you can use JavaScript called from an af:clientListener

```
<af:inputText id="it25" simple="true"
    value="...">
   <af:clientListener method="numbersOnly"
    type="keyDown"/>
```

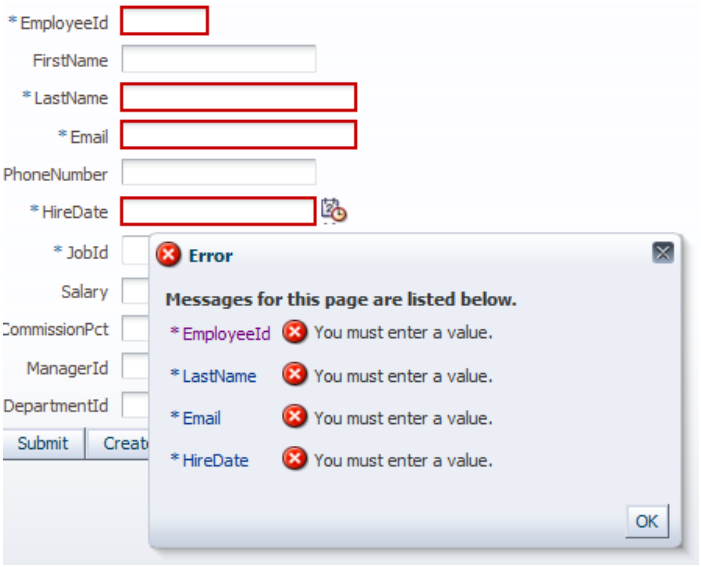
```
</af:inputText>
```

The JavaScript function then needs to be added to the page as shown below. Adding a JavaScript function is through the af:resource tag, which can contain the JavaScript code in its tag body, or reference it in an external JS file

```
function numbersOnly(evt){
      var _keyCode = evt.getKeyCode();
      var _filterField = evt.getCurrentTarget();
      var _oldValue = _filterField.getValue();

      //check for characters
      if (_keyCode > 64 && _keyCode < 91){
        _filterField.setValue(_oldValue);
        evt.cancel();
      }
}
```

Using JavaScript to clear validation error messages

ADF Faces provides client validation for component constraints like required fields. If input fields fail validation, then form data is not submitted to the server and instead an error message is displayed for the fields causing the error.



The error messages are displayed until the user corrects the input field values and re-submits the form, which could be irritating to users. To get rid of error messages of a failed form submit, application developers can use JavaScript as shown below

```
<af:resource type="javascript">
  function clearMessagesForComponent(evt){
    AdfPage.PAGE.clearAllMessages();
    evt.cancel();
```

```
  }
</af:resource>
```

The JavaScript client function is called from the focus event of a client listener added to the input fields in a form

```
<af:panelFormLayout id="pfl1">
  <af:inputText value="#{bindings.EmployeeId.inputValue}" …>
     …
    <af:clientListener method="clearMessagesForComponent"
                       type="focus"/>
  </af:inputText>
  <af:inputText value="#{bindings.FirstName.inputValue}" … >
    …
    <af:clientListener method="clearMessagesForComponent"
                       type="focus"/>
  </af:inputText>
  …
</af:panelFormLayout>
```

Note: The *AdfPage.js* class provides APIs to access the component messages and the changed components in list objects to individually handle component error message instead of clearing them all.

How-to share skin definition files across applications

Skinning defines the look and feel of an ADF Faces web application. On OTN and on internal help lists, the question came up of how to share a single skin definition file across ADF Faces applications to ease skin administration and modification. The preferred solution among customers is a URL reference to a remote server that downloads the skin definition with the start of an application. Though it is possible to setup the "style-sheet-name" element in the trinidad-skins.xml configuration file to reference the skin CSS source file using an absolute URL to a remote host, I am not convinced this is best good practice:

- The remote host connection may fail, in which case all applications start with the simple look and feel, which is far away from any corporate design I am aware of
- In the past it happened that failed configuration on a router – or a defect – lead to bad response time of servers. If one of these servers hosts the remote CSS file, then most likely the application renders with the simple look and feel
- Hard coded URL references are hard to maintain, and even if the trinidad-skins.xml file allowed the use of EL to dynamically resolve the URL, the URL had to be configured for and deployed with the application
- Skin inheritance is from a base skin definition that is referenced by its skin family name and the ".desktop" or ".pda" extension. If both skin definitions reference remote servers for their CSS, then the original motivation to simplify administration is traded in for a more complex and error prone approach.

The better solution, which addresses all the problems listed for the URL reference approach, while still meeting the goal of simplified administration and configuration is to use skin definitions that are defined in and deployed as shared libraries to WLS.

- Shared libraries can contain a single skin definition or related definitions. For example, a base skin file can be shipped with skin definitions hat inherit from it
- Shared libraries contain a copy of trinidad-skins.xml in it, which simplifies skin detection and configuration
- Changing a skin only requires changing the deployed shared library. WLS takes care of making sure all servers in a cluster are equipped with the library
- Different application point to exactly the same skin definition, so that consistency is guaranteed
- No network latency between the application using a skin and the server hosting the skin
- Skin definitions are protected from unapproved changes. This ensures that a corporate skin does not degenerate over time by developers applying ad-hoc fixes

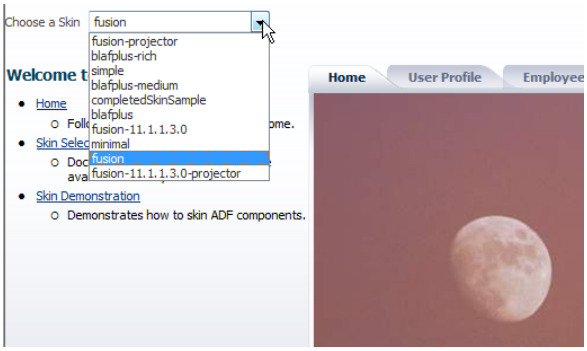
To implement the shared library approach, developers need to change their existing skin definition so it can be deployed in a JAR file. The steps for this include

- Creating a META-INF directory
- Creating a trinidad-skins.xml file that defines the skins deployed with the JAR file
- Creating an META-INF/adf sub directory for images and icons served from the JAR file
- Changing the image reference in the CSS to include the "adf" directory, which makes sure images and icons are handled by the ADF Faces resource loader, which can read resources from JAR files
- JAR the META-INF directory to create the library file

The "how-to deploy skin definitions in a JAR file" is explained in the "Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework" product documentation, which you can access from here:

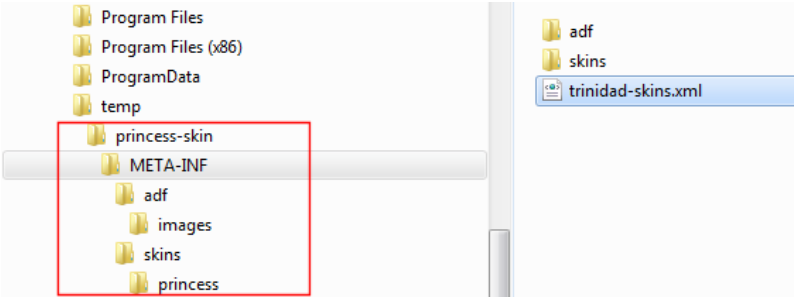
http://download.oracle.com/docs/cd/E15523_01/web.1111/b31973/af_skin.htm#CHDBEDHI

In the following, the steps to deploy the JAR file are listed in brief – though detailed enough to succeed.

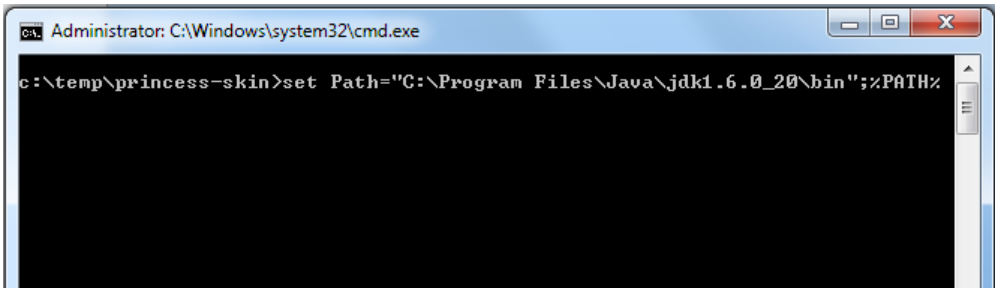


The sample above uses dynamic skin detection to show available skin definitions. The skin definitions you see in the list are those configured with the deployed application, or those that are contained in the ADF Faces product JAR files. A skin that is missing is "princess", which we are going to add in a shared library.

First, create a folder structure as explained in the Oracle product documentation to host images, and CSS files. The trinidad-skins.xml file itself is located directly under the META-INF directory you created.



Make sure the JAR command is in the command path of your OS. If not, set the path and add a reference to the bin directory of a Java JDK 6 or 5. If you have no stand alone JSK installed, use the one within the Oracle JDeveloper installation.

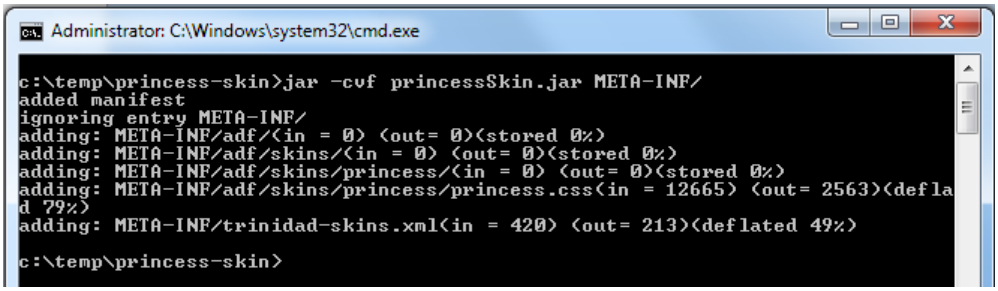


Next, issue the JAR command shown below

```
jar -cvf <name of skin jar>.jar META-INF/
```

The command is expected to be performed from the parent folder of the META-INF folder you created to hold the skin sources and the trinidad-skins.xml file. In the example I built, I created a pricessSkin.jar file, using the following command

```
jar -cvf princessSkin.jar META-INF/
```



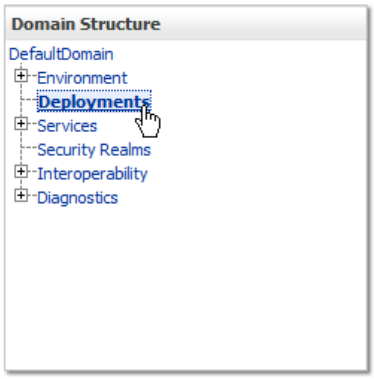
To configure the JAR file as a shared library, startup WebLogic server (either the integrated WLS in Oracle JDeveloper for testing, or a stand-alone server for production) and connect to the WLS console

http://host:7101/console or http://host:7001/console

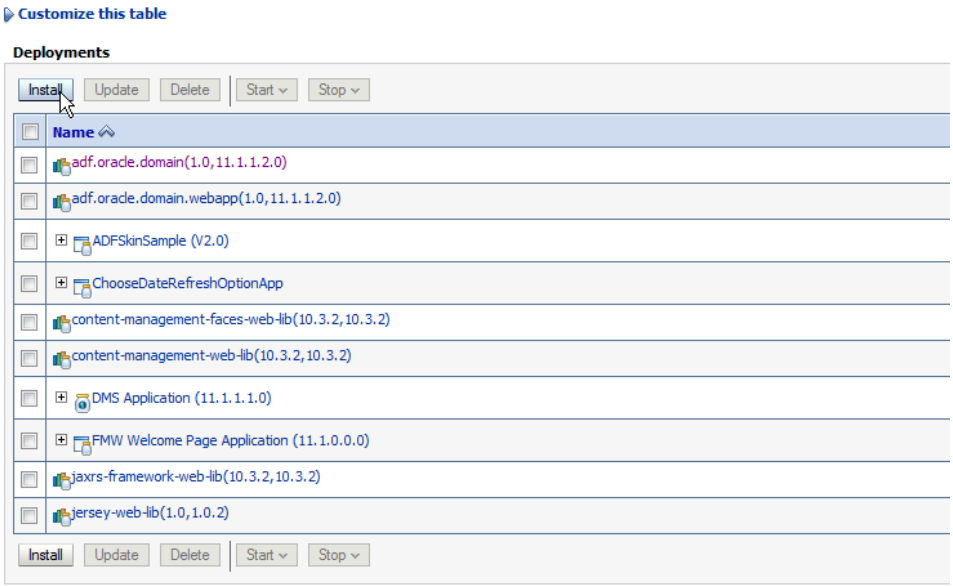
Connect as weblogic/weblogic1 to connect as an administrator

Note: If you are in a production environment and weblogic/weblogic1 still allows you to authenticate to the server then you have a serious security problem

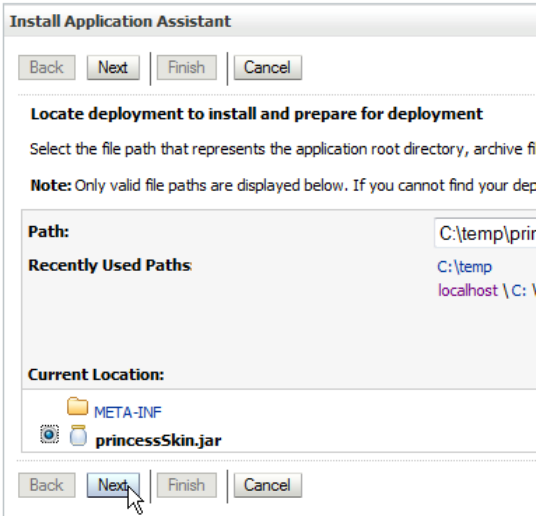
In the administration console, select the "Deployment" entry in the "Domain Structure"



Press the "install" button on top of the deployments table to register a new library

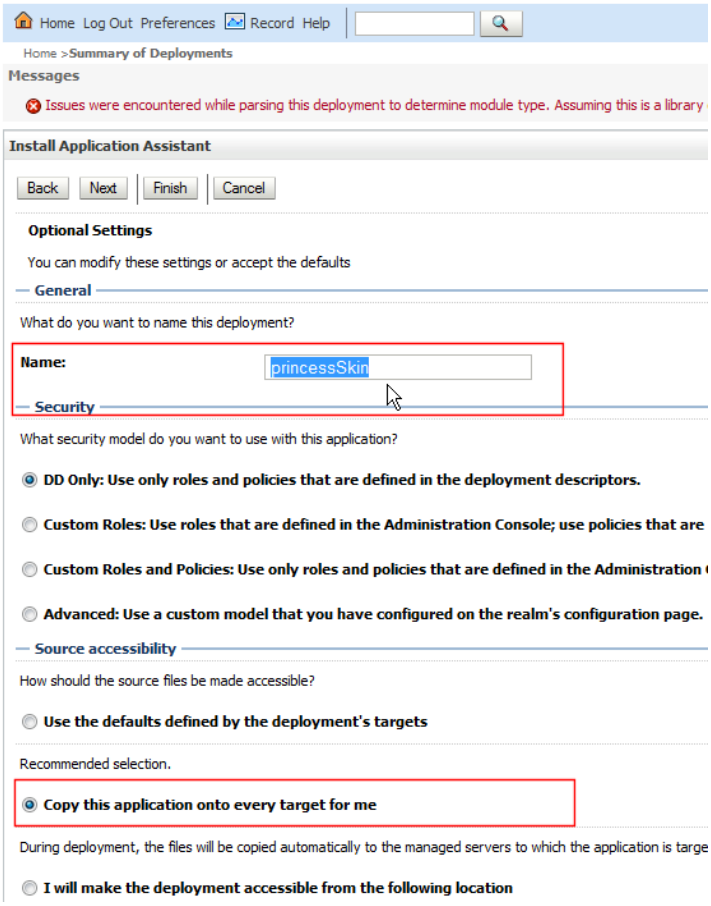


Browse to the location on the local server that has a copy of the skin definition JAR file and select it before pressing "Next"

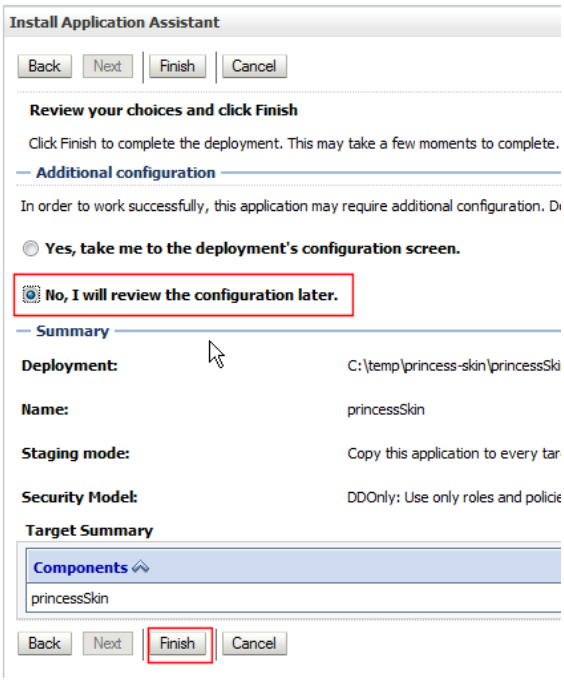


The next screen shows an error message that you can ignore. Still on this screen however, you should pay attention to the "Name" field that shows the suggested share library name. You can keep the default name, or change it. However, it is important to remember the name as we need it later.

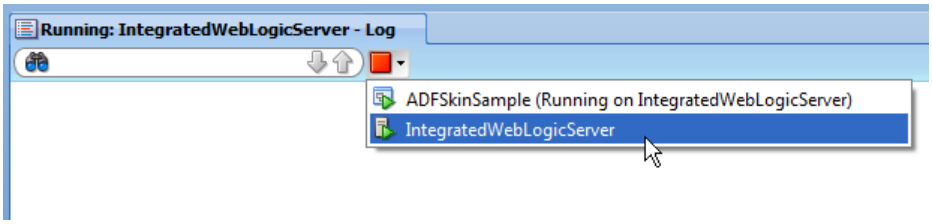
Still on the same screen, select the "*Copy this application onto every target for me*" option, to ensure all servers within a cluster are updated. Then press "Next"



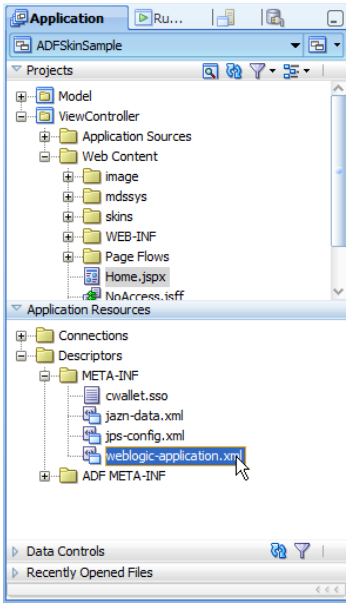
You can then decide whether or not you want to review the configuration before pressing "Finish" on the next screen



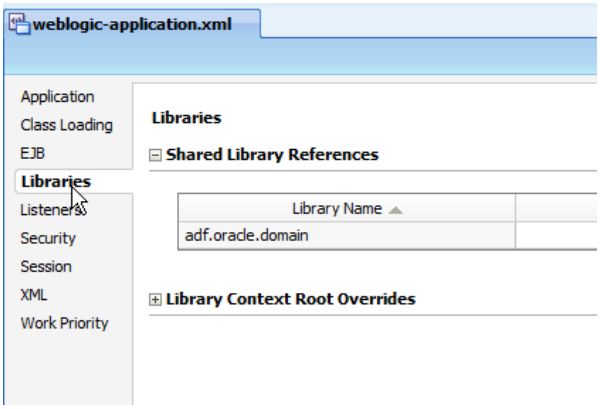
Note: The WLS console may or may not ask you to activate the applied changes. What you should always do though is to re-start WLS server – even if the server says that this is not needed.



Open the application(s) that you want to use the skin definition in the shared library file in Oracle JDeveloper. Expand the "Application Resources" accordion in the Application Navigator and select the "weblogic-application.xml" file. Double click onto the file to open it.

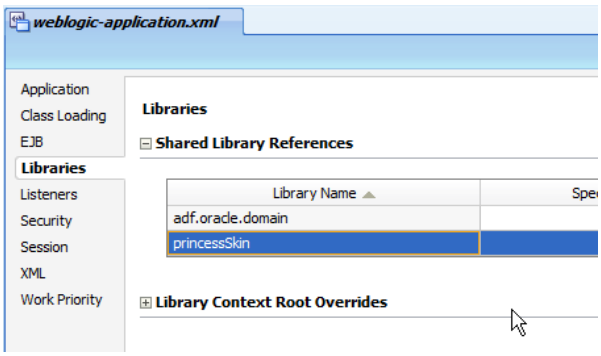


With the "weblogic-application.xml" file opened in the visual editor, select the "Libraries" category

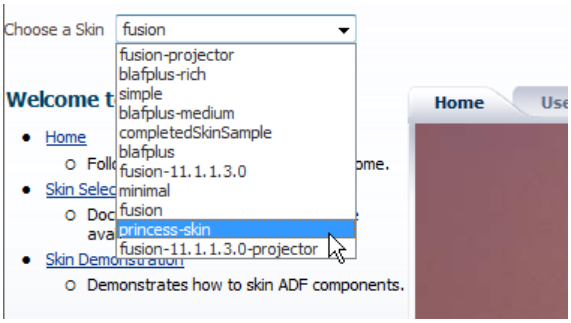


Press the "green plus icon" to create a new library reference for the application

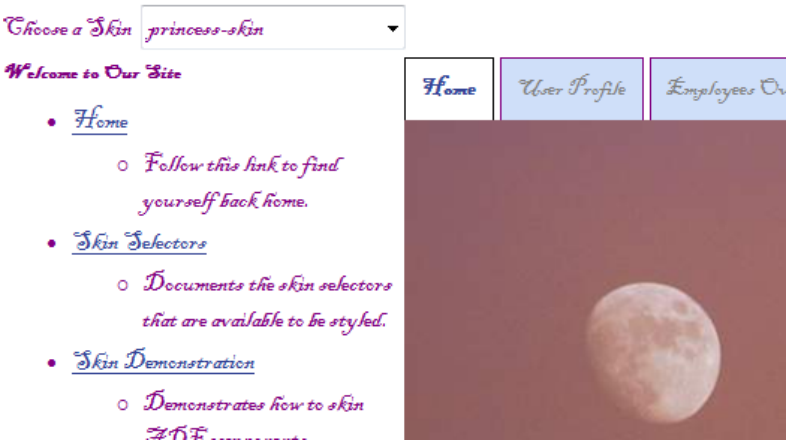
In the library name filed, type the name of the shared library you just created. In the example I used, the name was "princessSkin"



Next time the application deploys, the shared library and its skin definitions become available for the application to use



Note: If you deploy the application to a server that does not have the shared library defined, then the deployment will fail because of the unresolved library reference



How-to dynamically detect available skin definitions

Because I mentioned that skins can be dynamically looked up from available application configurations and the class path (which includes libraries). Find below a code listing that shows available skin definitions in a selectOneChoice component.

```
//managed bean class
public class SkinManager {
  public SkinManager() {
    ADFContext adfctx = ADFContext.getCurrent();
    Map sessionScope = adfctx.getSessionScope();
    Object skinFamily = sessionScope.get("skinFamily");
      if(skinFamily == null){
        sessionScope.put("skinFamily", "fusion");
      }
  }

  //method referenced from a selectOneChoice f:selectItems component
  public List getSkinChoices() {
    List choices = new ArrayList();
```

```
    String skinFamily = null;
    String skinLabel = null;
    SkinFactory sf = SkinFactory.getFactory();
    FacesContext context = FacesContext.getCurrentInstance();
    for (Iterator i = sf.getSkinIds(); i.hasNext(); ) {
      String skinID = (String)i.next();
      Skin skin = sf.getSkin(context, skinID);
      skinFamily = skin.getFamily();
      skinLabel = skinFamily;
      if (skin.getRenderKitId().indexOf("desktop") > 0) {
         choices.add(new SelectItem(skinFamily, skinLabel));
     }
   }
  return choices;
 }

  //method referenced from a selectOneChoice valueChangeListener
  public void onNewSkinSelection(ValueChangeEvent valueChangeEvent) {
   ADFContext adfctx = ADFContext.getCurrent();
   Map sessionScope = adfctx.getSessionScope();
   sessionScope.put("skinFamily", (String)
   valueChangeEvent.getNewValue());
   redirectToSelf();
  }
  //redirect that needs to be performed in case a selected
  //skin shouldbe applied
  private void redirectToSelf() {
   FacesContext fctx = FacesContext.getCurrentInstance();
   ExternalContext ectx = fctx.getExternalContext();
   String viewId = fctx.getViewRoot().getViewId();
   ControllerContext controllerCtx = null;
   controllerCtx = ControllerContext.getInstance();
   String activityURL = controllerCtx.getGlobalViewActivityURL(viewId);
   try {
     ectx.redirect(activityURL);
     fctx.responseComplete();
   } catch (IOException e) {
    //Can't redirect
    e.printStackTrace();
    fctx.renderResponse();
   }
  }
}
```

As select one choice to display the skin definitions would look as shown below:

```
<af:selectOneChoice label="Choose a Skin"
       value="#{sessionScope.skinFamily}" id="soc1"
       autoSubmit="true"
       valueChangeListener="#{SkinManager.onNewSkinSelection}">
       <f:selectItems value="#{SkinManager.skinChoices}" id="si1"/>
</af:selectOneChoice>
```

Note: Feel free to use the managed bean code for other use cases as well.

To make sure the trinidad-config.xml file reads the selected skin definition upon redirecting the page, the following entry needs to be defined

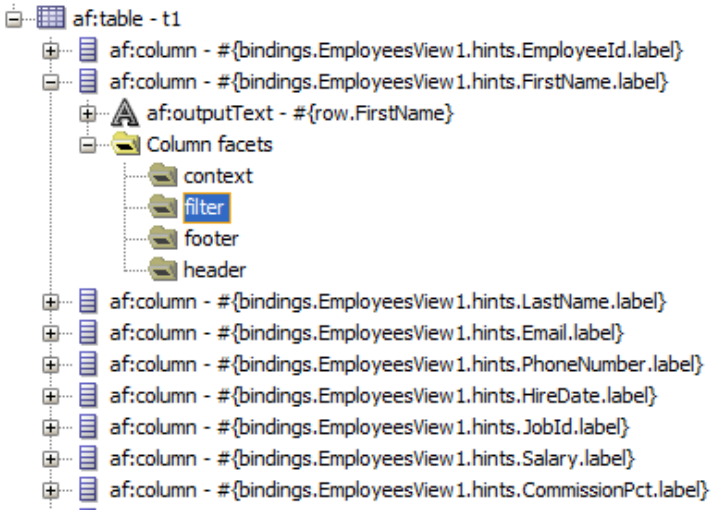
```
<skin-family>#{sessionScope.skinFamily!=null?sessionScope.skinFamily : 'fusion'}</skin-family>
```

The skin family is read from the "skinFamily" name in session scope. If the attribute is not set, then the skin falls back to use the Oracle Fusion skin. If the selectOneChoice described above is used on the initial application home page, then the session attribute is always set with the instantiation of the managed bean.

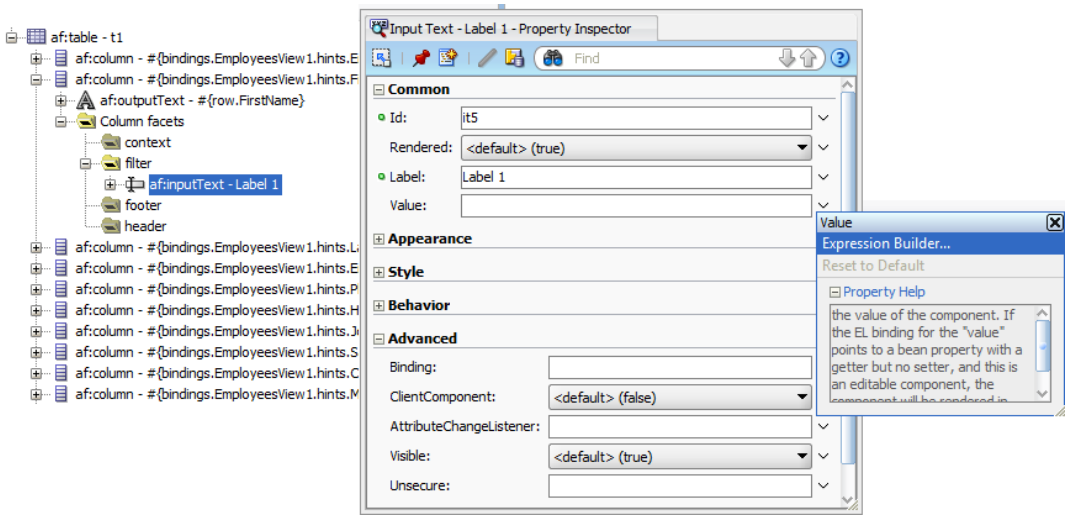
How-to define a tooltip for an ADF Faces table filter field

By default, the ADF Faces table does not show tooltips for its column data filters. However, only because it is not there by default does not mean it cannot be done.

With the page opened in the visual editor, select the table and open the Structure window. In the Structure window, select the af:column for which you want to define a tool tip.



Expand the "column facets" node and select the "filter" facet. From the ADF Faces component palette, drag and drop an af:inputText component into the "filter" facet as shown below.

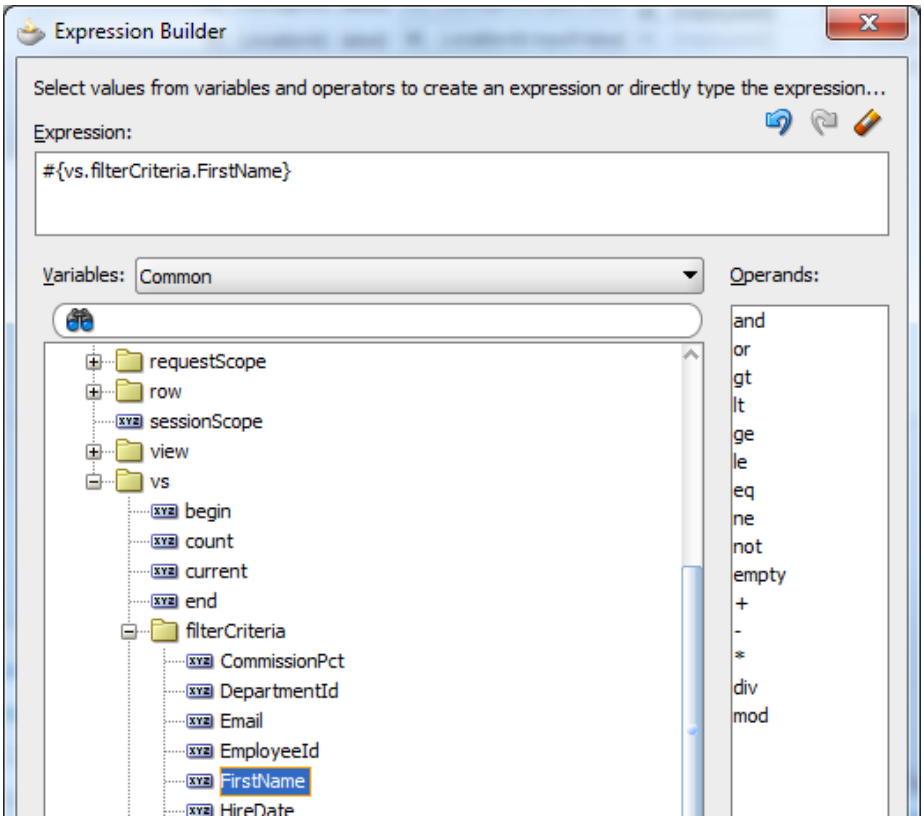


Set the input text field's value property to …

#{vs.filterCriteria.<attribute Name>}

… for the filter to query the column attribute it does query without customizing the filter component.

The correct syntax for the "filterCriteria" can be looked up from the Expression Builder dialog in Oracle JDeveloper as shown below.



The custom tool tip can now be set on the "shortDesc" property of the af:inputText field

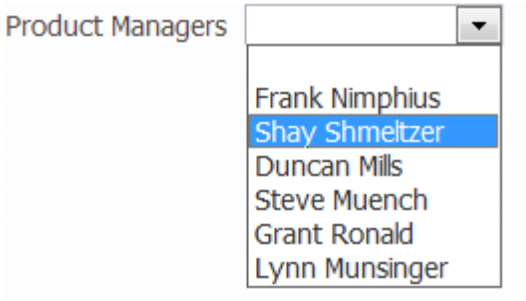
```
<af:table varStatus="vs" ...>
```

```
  <af:column ...>
    <f:facet name="filter">
      <af:inputText value="#{vs.filterCriteria.attributeNameXyz}"
       shortDesc="<...your tooltip...>"/>
    </f:facet>
    ...
  </af:column>
</af:table>
```

How-to read the selected label for a selectOneChoice selection

The af:selectOneChoice component allows you to display a label different from the value selected. But what if you want to show a summary page for which you need to display the selected label instead of the value you use to update the model?

In the sample below, a single select one choice is used to display Oracle JDeveloper Product Managers. Selecting an entry should print the selected label, not the value of the chosen person.




```
<af:selectOneChoice label="Product Managers" id="soc1"
                    valuePassThru="true"
                    valueChangeListener="#{selectionHandler.onSelectionChange}"
                    autoSubmit="true">
  <af:selectItem label="Frank Nimphius" value="1" id="si6"/>
  <af:selectItem label="Shay Shmeltzer" value="2" id="si4"/>
  <af:selectItem label="Duncan Mills" value="3" id="si3"/>
  <af:selectItem label="Steve Muench" value="4" id="si5"/>
  <af:selectItem label="Grant Ronald" value="5" id="si1"/>
  <af:selectItem label="Lynn Munsinger" value="6" id="si2"/>
</af:selectOneChoice>
```

The managed bean code is shown below.

```
public void onSelectionChange(ValueChangeEvent valueChangeEvent) {
  RichSelectOneChoice rsoc =
              (RichSelectOneChoice)valueChangeEvent.getSource();
  List childList = rsoc.getChildren();
  String newVal = (String)valueChangeEvent.getNewValue();
  for (int i = 0; i < childList.size(); i++) {
    if (childList.get(i) instanceof RichSelectItem) {
```

```
        RichSelectItem csi = (RichSelectItem)childList.get(i);
        if (((String)csi.getValue()).equals(newVal)) {
           //TODO store the label or print it
           System.out.println(csi.getLabel());
        }
      }
    }
}
```

A navigation case defined in ADFc does not work

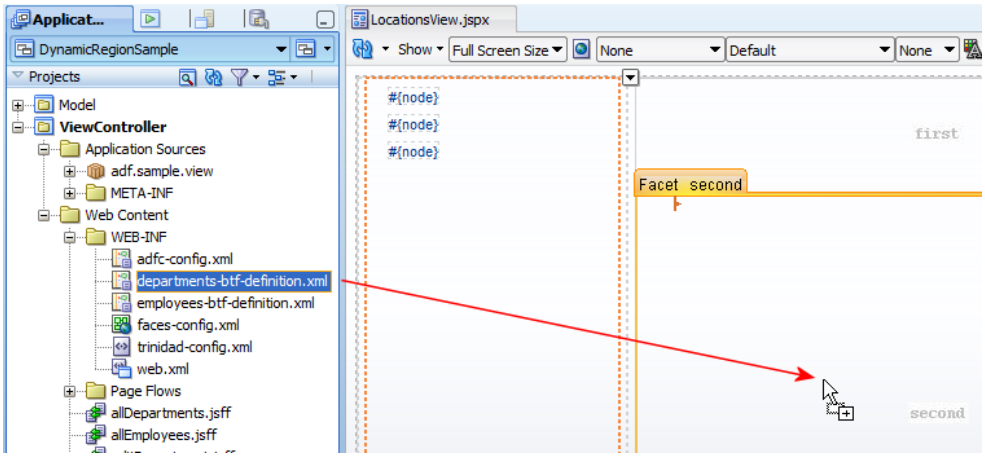
It may happen that a navigation case defined between two JSPX pages does not work when referencing it from the action property of a command component in ADF Faces. If you define the same navigation case in faces-config.xml then the same navigation works. The reason for this behavior is within how you start the application. For ADF controller to take the lead on handling navigation, you call a view activity, not the physical file. So if your request URL shows faces/<filename>.jspx instead of faces/<activityname> then you actually bypassing the ADF controller. Thus the faces-config.xml file is looked up to resolve the navigation case.

What is the _afrLoop URL parameter for?

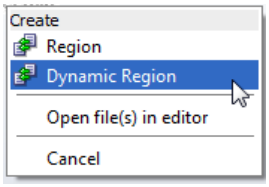
When running ADF applications using the ADF Faces you see the _afrLoop token added to the URL looking like: _afrLoop=1039335694640519 . A frequent question in regard to this token is "can I get rid of the _afrLoop token in the URL?". The answer is no and has to do with why this token is needed. The _afrLoop param is an ADF internal parameter that is used to detect new browser windows to be opened – e.g. using ctrl+N - and also helps detecting PPR navigation. Removing this token will disable this functionality and lead to the framework to malfunction. So don't attempt removing this.

How-to create dynamic regions

Dynamic regions in ADF expose a bounded task flow referenced from a managed bean.



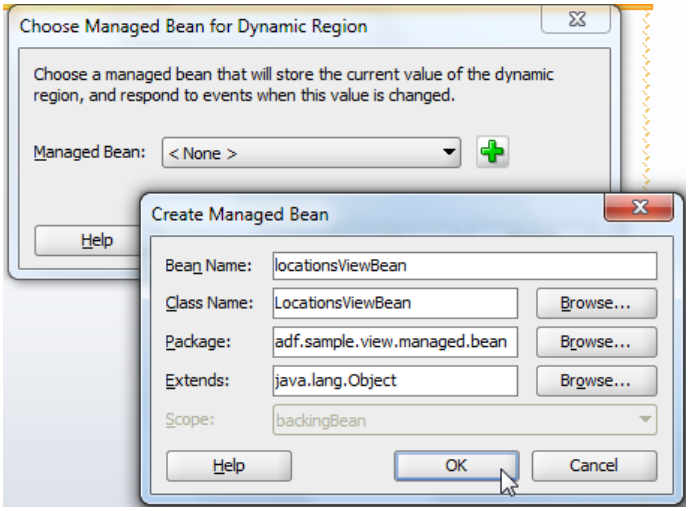
When dragging a bounded task flow that uses page fragments from the Application Navigator to a page, a menu dialog is opened for the developer to choose the kind of region he/she wants to create.



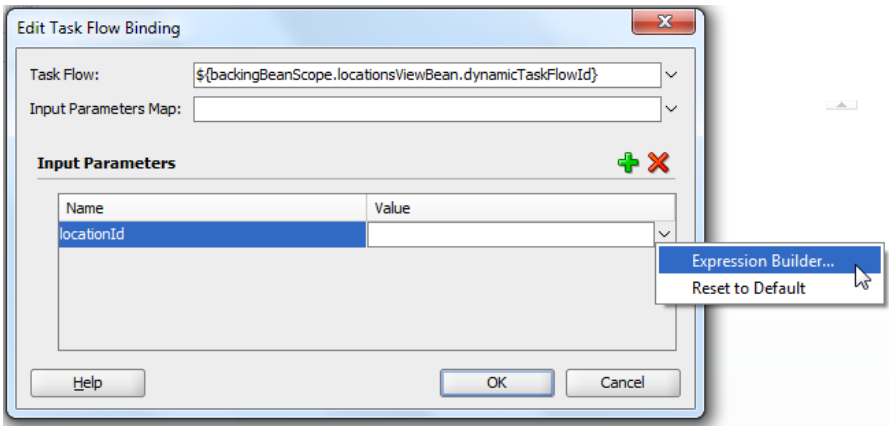
Using the static **Region** options, creates a Region binding in the drop page's PageDef file that points to the document location and task flow id of the task flow to display

Choosing **Dynamic Regions**, creates a region binding that references a managed bean method to obtain the task flow document and Id. Changing the managed bean task flow document reference it returns, thus changes the task flow displayed in the region. Choosing the dynamic region option, a dialog is opened that allows you to create the managed bean to hold the task flow configuration. The only "key to success" you need to be aware of is that the managed bean by default is configured in backingBean scope.

The backingBeanScope is okay to use if you only want to display a single task flow and never intend to allow users to switch between task flows. If you want to switch between task flows, then the backing bean scope must be changed to a larger scope, at least viewScope or pageFlowScope



Once the managed bean is created – though yet in the wrong scope – a dialog is opened for you to define task flow parameters.

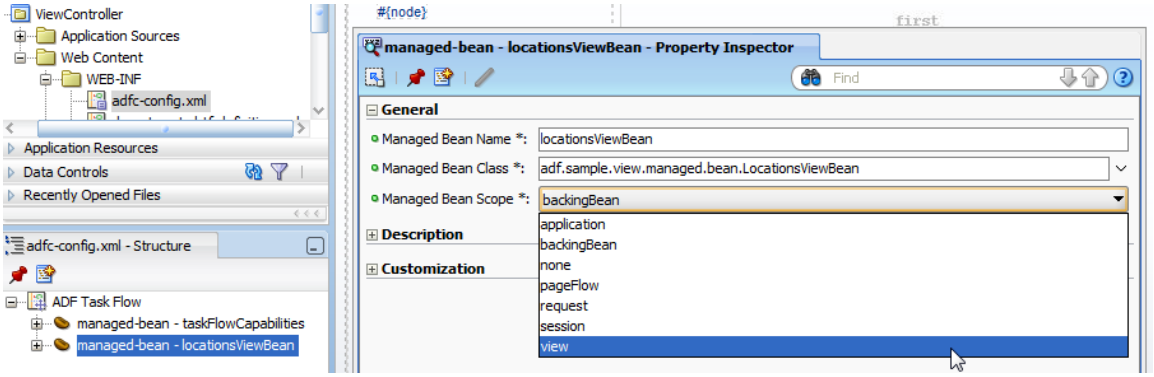


There are two options to provide input parameters that you can choose from

Input Parameters – are named input parameters, where the parameter name matches the input parameter name defined on the bounded task flow. If the task flows you switch between have different input parameter, then you can provide them all in the Input Parameter list. If a parameter is passed to a bounded task flow for which the task flow has no input parameter defined itself, then this parameter is ignored.

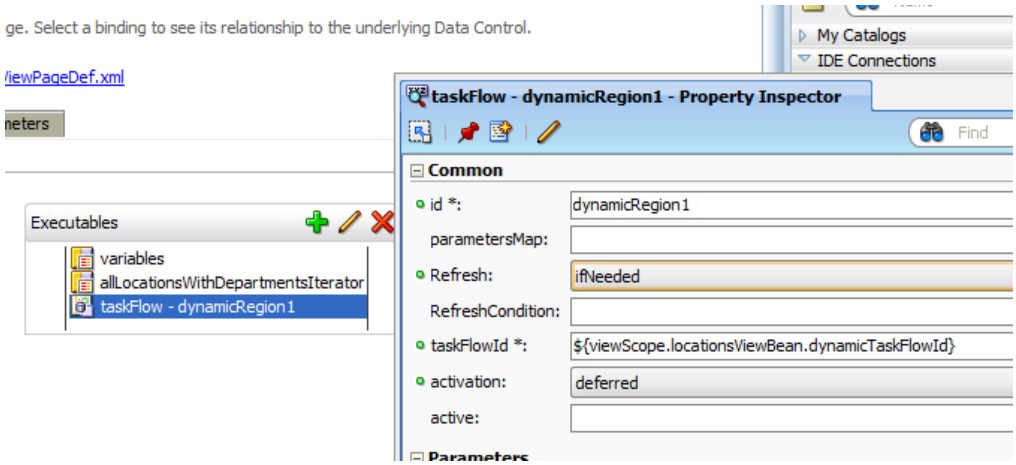
Input Parameter Map – Especially when you switch between more than two task flows, or when task flows have lots of input parameters, providing a static definition of name / value pairs is cumbersome. In this case you use a HashMap to hold the name/value pairs and ensure the map is refreshed according to the task flow you switch to.

To change the scope of the managed bean that got created to reference the task flow, select the task flow configuration file with the managed bean definition in the JDeveloper Application Navigator. Open the Structure Window (ctrl+shift+S) and select the managed bean definition.

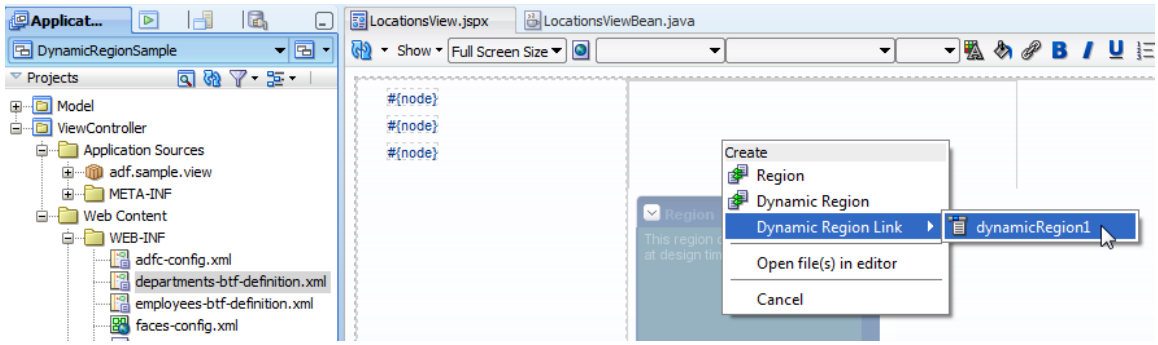


Use the Property Inspector to change the scope of the bean.

Next, you need to change the reference to this bean that is defined in the parent page's PageDef file.

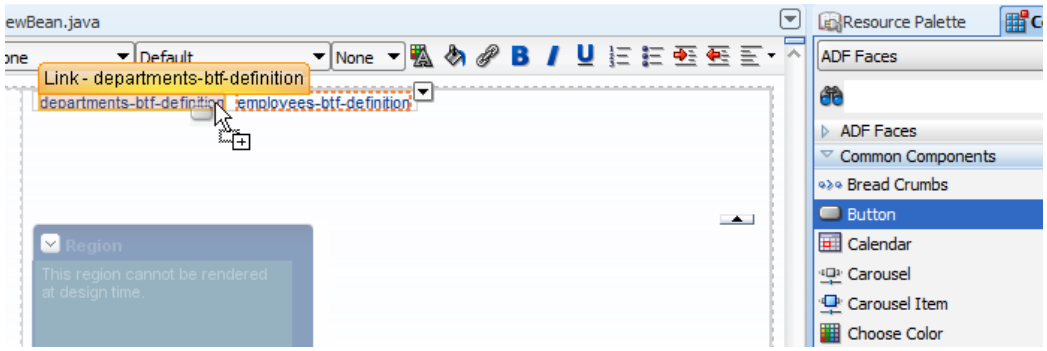


With this change you can now create the command links to switch between task flows, which implicitly create new task flow references and methods exposed by the managed bean.



Drag and drop each bounded task flow definition that you want to show in the ADF Region from the Application Navigator to the page view and choose Dynamic Region Link | <dynamic region name>.

To convert links into command buttons, or menu items you e.g. select the command button component in the JDeveloper Component Palette using the right mouse button. Keep the right mouse button pressed and move the command button onto the command link you want to change and drop it. A menu opens that allows you to choose the convert option.



When are ADF Regions and Dynamic Regions getting refreshed?

The refresh behavior of an ADF Region is influenced by the setting defined on the Region binding "Refresh" and "RefreshCondition" property, or when the refresh method is called on the RichRegion instance. David Giammona from the ADFc development team blogged about the lifecycle phases of the refresh as follows:

"Regions and dynamic regions are initially refreshed in the ADF Lifecycle prepareModel phase when the binding container of the containing page is refreshed. On subsequent requests, the region or dynamic region will be refreshed during the ADF Lifecycle prepareRender phase depending on its task flow binding Refresh and RefreshCondition attributes and parameter values. Task flow definitions associated with dynamic regions are only switched during the ADF Lifecycle prepareRender phase."

How-to navigate in bounded task flows

A frequent question on OTN is of how to programmatically navigate within bounded task flows. The use case usually is to perform navigation in form of a non-action event, like e.g. a value change event.

Three options I am aware of that developers can use to perform navigation in bounded task flows. It is important however to be conscious not to fight the ADF controller framework.

A controller not only navigates between pages – or views as it is referred to in ADFc – but it also keeps track of states, which is especially important if bounded task flows are used. The three options to navigate in bounded task flows are

- Replacing the current view port's viewId
- Queue an action on a command component
- In the case of a region, queue an event on the region component

The first of the listed options is to change the displayed viewId for a view port. A view port in ADFc refers to the area displaying a view. This can be a region, or the browser window. The following code below changes the viewId for the current view port

```
ControllerContext ccontext = ControllerContext.getInstance();
//set the viewId - the name of the view activity to
//go to - to display
String viewId = "EmployeesView";
ccontext.getCurrentViewPort().setViewId(viewId);
```

Be aware that this approach however bypasses the JSF lifecycle. So you need to make sure – when using it – that not submitted data gets submitted before changing the viewId as all changes are getting dismissed when changing the viewId. So be conscious when using this code snippet in the context of an event that is invoked by a component having its immediate property set to true. Another restriction of this solution is that you can only navigate to viewable targets. You cannot navigate to method or router activities. If you need to have a non-viewable targets addressed then one of the following options will work for you.

The second option to programmatically perform navigation is to queue an action event on a navigation component that is displayed or hidden (displayed = "false" *not* rendered="false") on the view.

```
private void navigateByQueueAction() {
   FacesContext fctx = FacesContext.getCurrentInstance();
   UIViewRoot root = fctx.getViewRoot();
   //client Id of button includes naming container like id of region.
   RichCommandButton button =
           (RichCommandButton) root.findComponent("r1:cb3");
   ActionEvent actionEvent = new ActionEvent(button);
   actionEvent.queue();
}
```

The command button that the action event is queued for has the navigation case defined in its action property. If you want t be flexible, then you define wild card navigation rules to the activities you want to access programmatically. The command button would be defined as hidden and act as a proxy. If you follow a coding pattern in that wild card navigation rules are constructed like <activityId>WCRule, then all target control cases become predictable. E.g. to navigate to the "browseEmployees" view activity, developers would know that the wild card navigation rule is "browseEmployeesWCRule".

The third option is if you don't like the idea of adding a proxy command component on views, though you could try a command button in a page template to not have to add it on all pages. In this case you queue the action event on the region containing the bounded task flow. As you guessed, this option is not available for task flows that are not displayed in a region.

```
Private void navUsingQueueAction(ActionEvent actionEvent) {
 // the task flow is contained in a region, so let's go and find it
  UIComponent  comp = actionEvent.getComponent();
  while(!(comp instanceof RichRegion)){
      comp = comp.getParent();

  }
  RichRegion rr = ((RichRegion) comp);
   rr.queueActionEventInRegion(
     createMethodExpressionFromString("browseEmployeesWCRule"),
     null, null, false, 0, 0,PhaseId.INVOKE_APPLICATION);
 }
```

The code above queues the wild card event on the region, which means there is no need for you to add hidden command components to the views to act as a proxy. Instead you can – safely – assume that there exists an instance of Rich Region on the page that you can find from the component that needs to invoke navigation. The queueActionEventInRegion method has the following signature, so you understand the API

```
public void queueActionEventInRegion(
     javax.el.MethodExpression actionExpression,
```

```
      javax.el.MethodExpression launchListenerMethodExpression,
      javax.el.MethodExpression returnListenerMethodExpression,
      java.lang.Boolean useWindow,
      java.lang.Integer windowWidth,
      java.lang.Integer windowHeight,
      javax.faces.event.PhaseId phaseId
)
```

The little helper method "createMethodExpressionFromString" used in the sample, is listed below

```
private MethodExpression createMethodExpressionFromString(String s){
    FacesContext fctx = FacesContext.getCurrentInstance();
    ELContext elctx = fctx.getELContext();
    ExpressionFactory exprFactory =
           fctx.getApplication().getExpressionFactory();
    MethodExpression methodExpr = exprFactory.createMethodExpression(
                   elctx,
                   s,
                   null,
                   new Class[]{});
    return methodExpr;
 }
```

Using option 2 and 3 actually has the ADFc controller performing the navigation, ensuring all state is preserved and data is getting submitted to update the model. Option 1 is if you take care of necessary model updates or ensure the update model phase is processed.

Naming conventions to consider when using ADF Libraries

You use ADF Libraries to reuse ADF artifacts like page templates, bounded task flows, ADF Business Components and declarative components. However, be cautious when developing reusable components to avoid name clashes in Java classes and packages, as well as duplicate metadata file names. Section 33.1.1 "Creating Reusable Components" of the "Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework" suggests development teams to agree on a naming convention

"You and your team should decide on the type of repository needed to store the library JARs, where to store them and how to access them. You should consider how to organize and group the library JARs in a structure that fits your organizational needs. You should also consider creating standardized naming conventions so that both creators and consumers of ADF Library JARs can readily identify the component functionality."

It then adds a note about what you can do if the problem of missing naming conventions shows in the middle of a development project.

"Note: If, in the midst of development, you and your team find a module that would be a good candidate for reuse, you can use the extensive refactoring capabilities of JDeveloper to help eliminate possible naming conflicts and adhere to reusable component naming conventions."

Though this note is correct and refactoring is an option. The note is very moderate as I would consider refactoring a worst case scenario. Better is to consider naming conventions up front, for which this post is a heads up.

See the link below for naming conventions proposed by Oracle

http://download.oracle.com/docs/cd/E15523_01/web.1111/b31974/reusing_components.htm#BEIJIGDG

How-to convert user input into uppercase or lowercase strings

This use case actually depends on when you need the conversion to happen. Trigger points for changing the case of user input data are

- Form submit
- Focus put out of a field
- Key press

The form submit and out of focus case is what you can handle with ADF Faces native means using a managed bean in combination with a ValueChangeListener defined on the field. For the third option, and of course the focus change case, JavaScript can be added to the ADF Faces page as shown below

```
<af:inputText value="...">
  ...
  <af:clientListener type="keyUp" method="toUpper"/>
</af:inputText>
```

The JavaScript function referenced from the af:clientListener, the component that listens to client side component and DOM events, is defined as shown below

```
<af:resource type="javascript">
  function toUpper(event) {
    var txtField = event.getCurrentTarget();
    txtField.setValue(txtField.getSubmittedValue().toUpperCase());
  }
</af:resource>
```

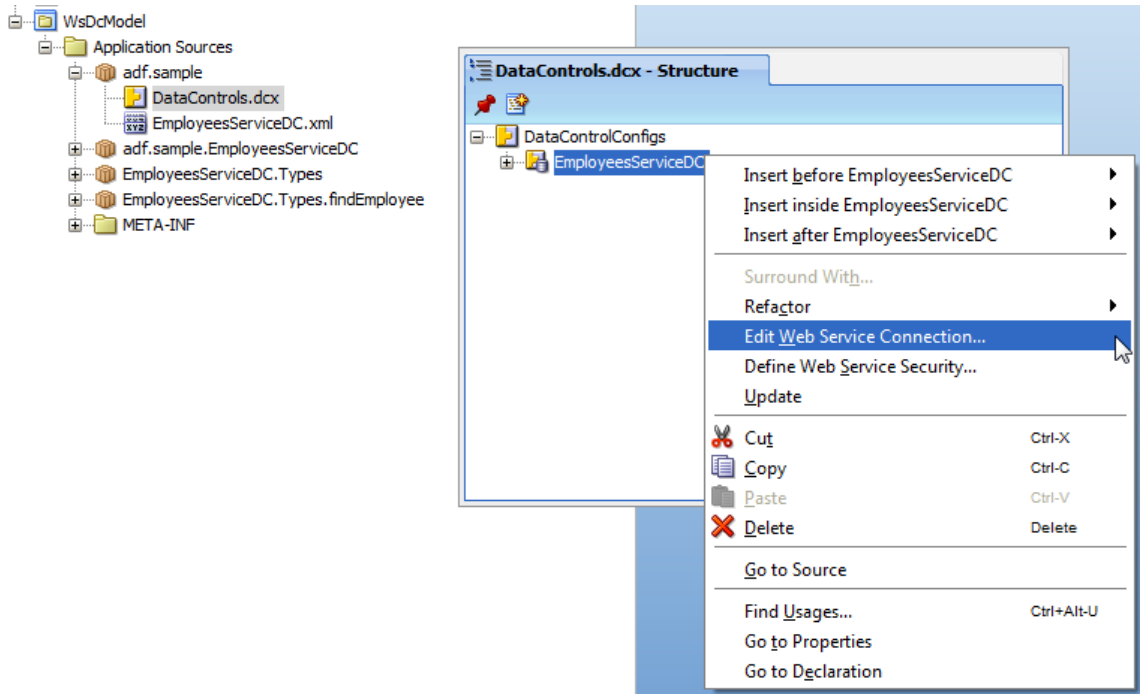
Adding ADF bound page fragments using jsp:include tag

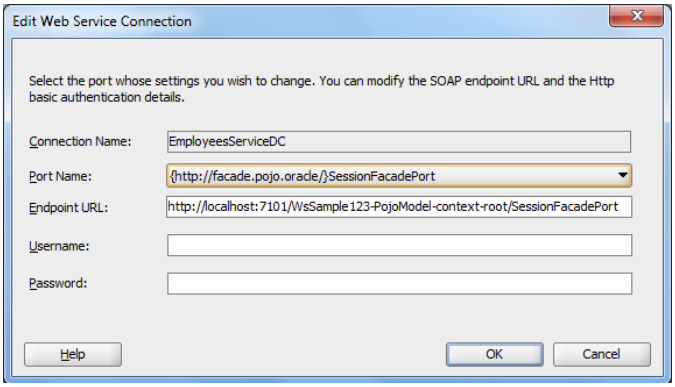
An "old hat" for experienced ADF users, but still a gotcha for new ADF developers: ADF bound page fragments that are added to an ADF Faces page using the **jsp:include** tag fail at runtime. To solve the issue, copy the ADF binding content of the page fragments PageDef.xml file to the PageDef.xml binding file of the parent page.

The ADF binding definitions of a page fragment are not linked from the parent page's PageDef file when referencing the page fragment from a jsp:include. To avoid problems because of missing bindings caused by this, either copy the binding content to the parent PageDef file or use a bounded task flow containing the page fragment.

How-to change the WS Data Control WSDL URL references

To change the WSDL reference of a Web Service DataControl, select the DataControls.dcx file and open the Structure Window (ctrl+shift+S). In here, select the Data Control entry and click on the "Edit Web Service Connection" entry.





RELATED DOCOMENTATION

☒	
☒	
☒	