ADF Code Corner

Oracle JDeveloper OTN Harvest 04 / 2011

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.

http://blogs.oracle.com/jdevotnharvest/

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
30-APR-2011

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

April 2011 Issue – Table of Contents

How and where to start learning ADF

Typically developers who join the Oracle JDeveloper and Oracle ADF community have a programming background which may or may not be Java. So to answer the question of where and how to start learning Oracle ADF is not straight forward as it depends on what you already know. In his blog, Shay Shmeltzer outlined a trail of documents and samples that give you a path into ADF.

http://blogs.oracle.com/shay/2010/02/how_do_i_start_learning_oracle_adf_and_jdeveloper.html

If your programming background is not Java, don't worry if this looks like a lot to read up on and keep in mind that Rome wasn't build in a day either. Try some of the Oracle by Example (OBE) tutorials in which we step you through building an ADF application without assuming any Java knowledge on your side.

Two additions to Shay's 2010 list of materials are:

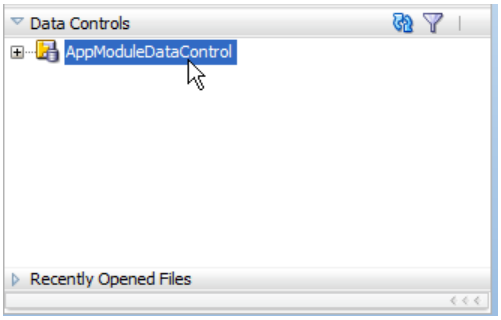
- Grant Ronald's "Quick Start Guide to Oracle Fusion Development" book. The book is written to be the lowest entry level to application development with Oracle ADF. http://www.mhprofessional.com/product.php?cat=112&isbn=0071744290
- ADF Insider Essentials: A video tutorial series that show you some common development tasks and how to conquer: http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adfinsideressentials-337133.html

Most popular mistake when reporting problems on OTN

The topic of "how to ask good questions on a forum" is well covered in blogs and articles. One mistake I quite often find when monitoring the Oracle JDeveloper forum on OTN is complexity that prevents us, meaning everyone who is willing to help, to understand or reproduce the problem.

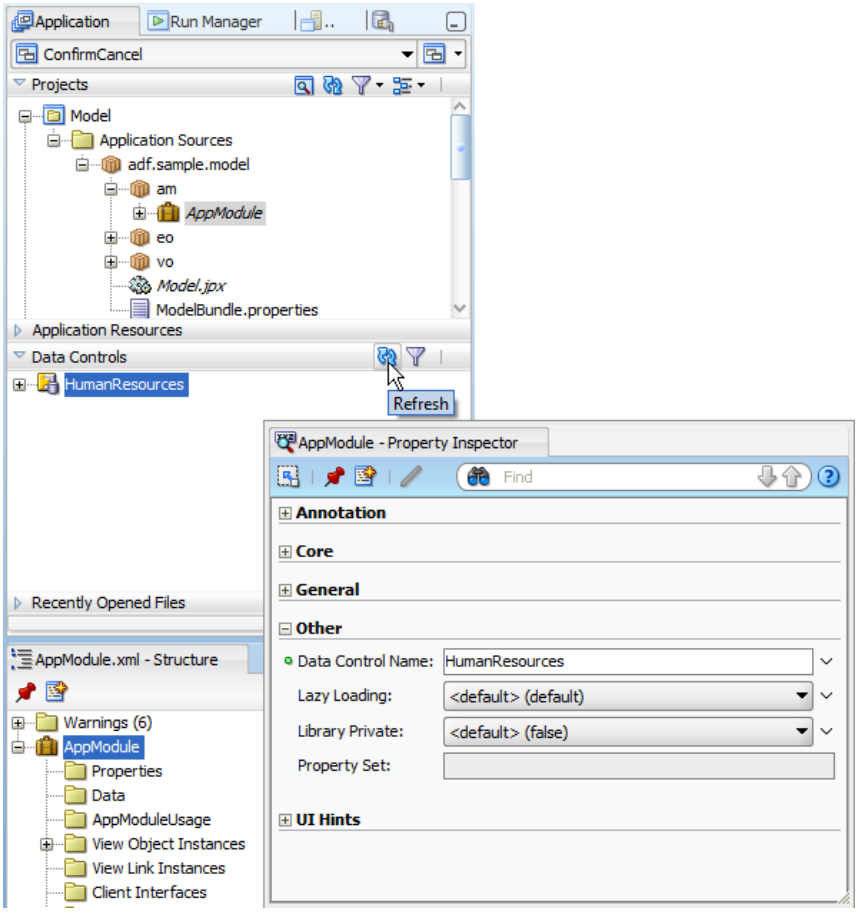
Customizing the ADF BC Data Control Name

When creating an ADF Business Component service and exposing it to ADF, then the Application Module Name is exposed as the Data Control name in the ADF Data Controls panel.



Names that are exposed on the Data Controls panel are a contract defined between the business service developer and the application developer. Those names should be descriptive for intuitive and error free

use during application development. So instead of the technical Application Module name to be exposed in the Data Controls panel, you want something better, as shown below

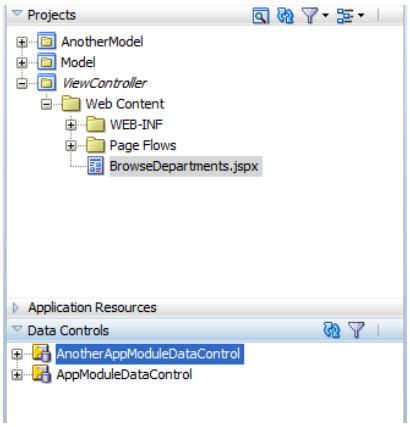


Select the Application Module in the Oracle JDeveloper Application Navigator and open the Structure Window. In here, select the Application Module root node and open the Property Inspector. Change the **Data Control Name** property and save the project. Hit the **refresh** icon on top of the Data Controls panel to show the user friendly name.

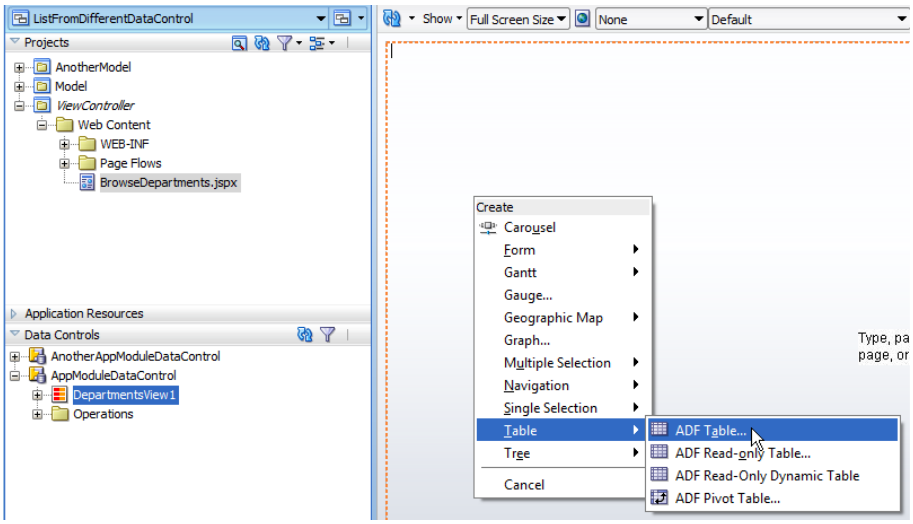
Note: Data Control names are saved in the `DataBindings.cpx` file as the ID for the DataControls configuration reference. To change the Data Controls name for an existing project, rename the ID in the `DataBindings.cpx` file accordingly and search for all references of the old ID. This "user friendly" naming for sure is easier to use for new developments.

Populating select choice components from other DataControls

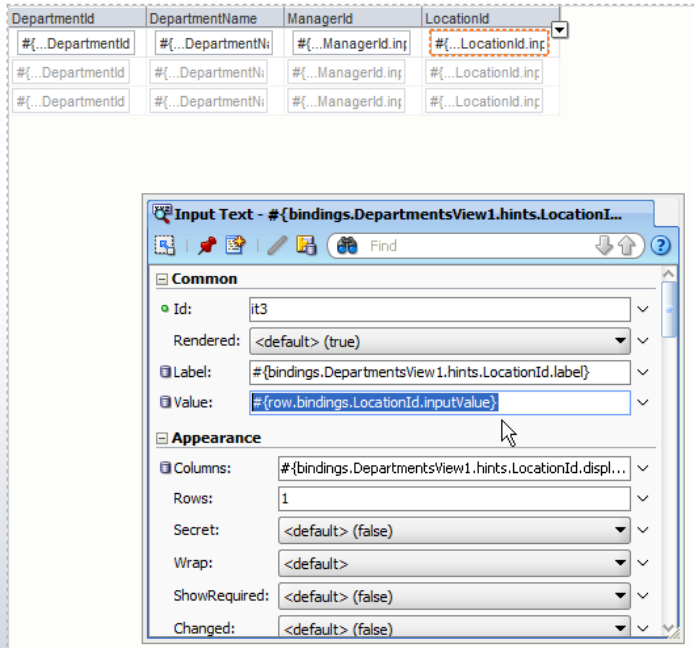
Oracle ADF allows you to reference values from other Data Controls when working in a form. But how do you populate list values of an `af:selectOneChoice` component with data queried form another Data Control?



For simplicity reasons when creating the sample, I used two ADF Business Components Data Controls. Of course, cross referencing Data Controls makes more sense if the Data Controls reference different model technologies, for example when reading list values from a Web Service or an EJB / JPA model. The two ADF Business Components Data Controls are read from different projects. Using different projects is not a requirement for having two separate Data Controls, but help clarifying the use case to show here.



First I created a table from a View Object contained in the Model project. The table is configured to be editable and to support row selection.



Note that the value of the `af:inputText` component that renders the **LocationId** attribute references the table *row* variable.

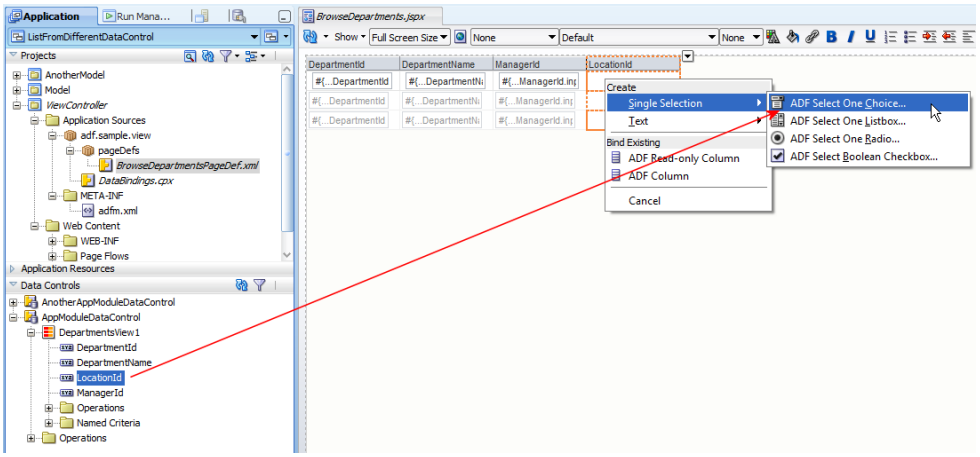
#{row.bindings.LocationId.inputValue}

If the DepartmentsView view object wasn't created as a table but a form, then the EL would reference an attribute binding look as shown below

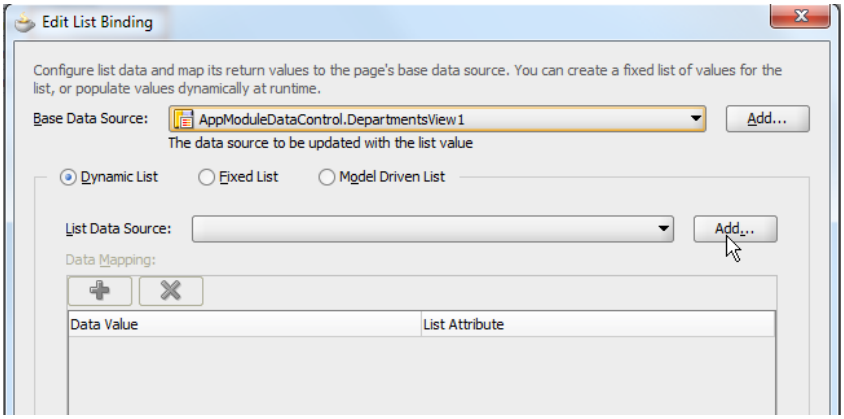
#{bindings.LocationId.inputValue}

I copied the **value** property EL to the clipboard so I can paste it back in later on. This is an important step, so don't forget this.

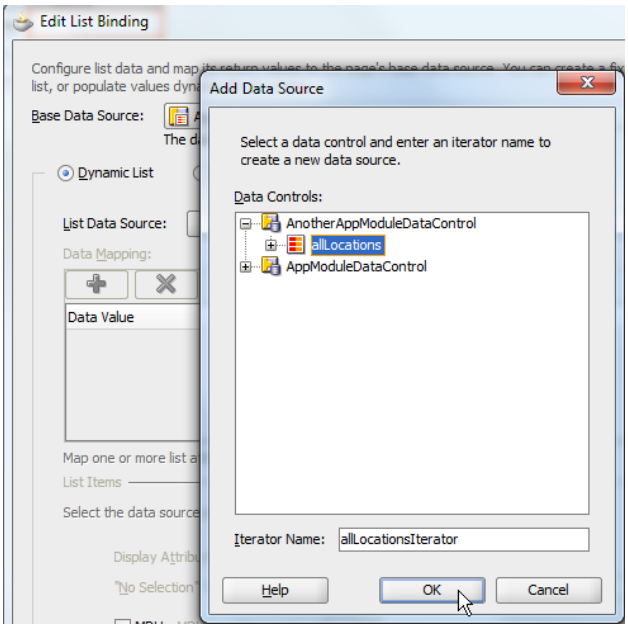
I deleted the `af:inputText` component in the **LocationId** column, but made sure the `af:column` component was kept.



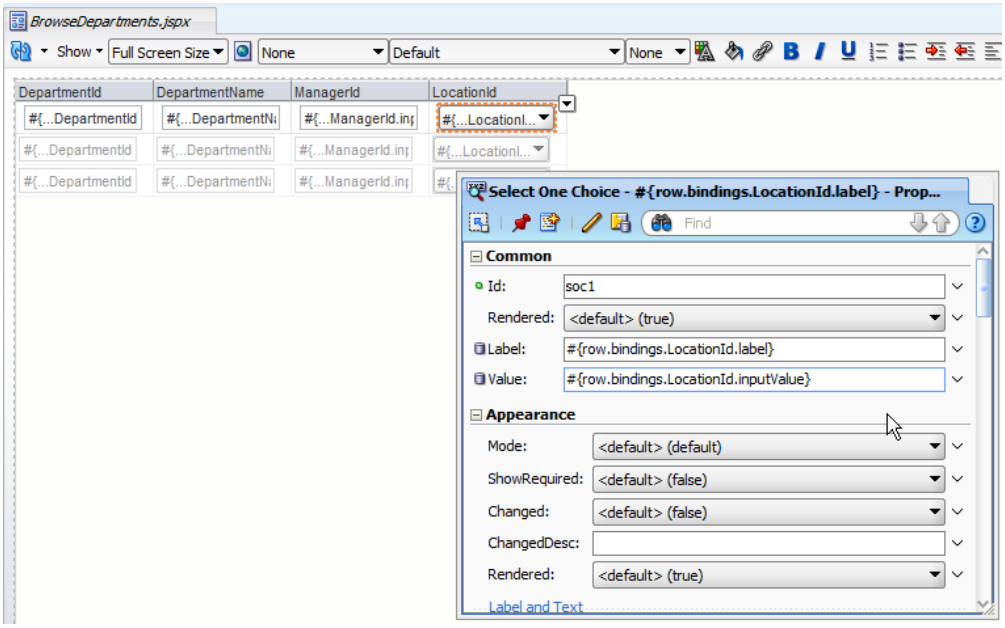
I then expanded the **DepartmentsView** View Object node that I used to build the table and dragged the **LocationId** attribute into the **LocationId** table column. From the choice of rendering options, I chose **ADF Select One Choice** component.



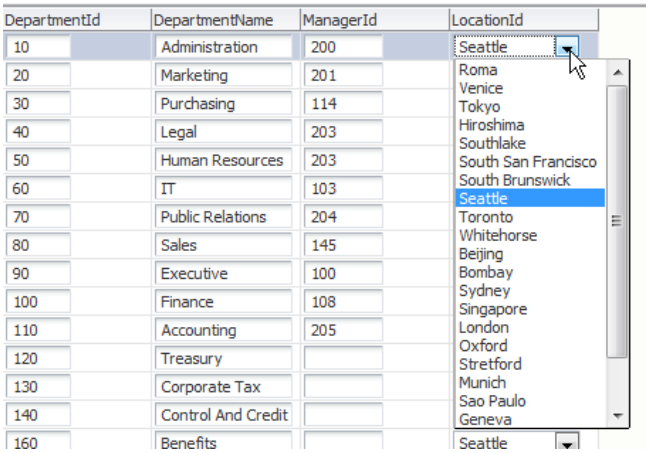
To retrieve the select item values from another Data Control, I set the list type to **Dynamic List** and pressed the **Add** button next to the **List Data Source** list box.



I then chose the collection (View Object) in the other Data Control that provides the list values and pressed OK.



I then selected the `af:selectOneChoice` component in the **LocationId** column and opened the Property Inspector. I pasted the expression string I kept in the clipboard into the **Value** property of the `af:selectOneChoice` component so the list update is for the row in the table.



Note: Don't make it a general practice in your application development to spread dependent information across Data Controls. Only do so if there is a reason. In my example, using two application modules as independent Data Controls will create two database connection. Using two separate Data Controls always makes sense, as mentioned, when business services are implemented with different technologies.

Run ADF Faces applications with IE 9 in IE 8 compatibility mode

MS Internet Explorer 9 is production and developers and users eagerly pick this browser version up for their production environment. Oracle JDeveloper 11.1.1.4 has been released before Internet Explorer 9 and, for this reason is not supported with this version of IE. Developers who don't want to wait for the next JDeveloper 11g patch set to support IE 9, or developers who don't want to upgrade their application

infrastructure to a new version of Oracle Fusion Middleware and Oracle JDeveloper 11g, however may consider running Internet Explorer 9 in IE8 or E7 compatibility or emulation mode.

http://blogs.msdn.com/b/ie/archive/2010/06/16/ie-s-compatibility-features-for-site-developers.aspx

http://expression.microsoft.com/en-us/dd835379.aspx

Since IE 8, Microsoft supports a meta tag that, if added to the document header, enforces the browsers to behave like a previous version

<meta http-equiv="X-UA-Compatible" content="|" />

For example, to force a browser to behave like IE 7, you use

<meta http-equiv="X-UA-Compatible" content="IE=7" />

To add the **X-UA-Compatible** meta tag from your JavaServer Faces application in a way that does not require you to change the source of your pages, you use a `PhaseListener` that you define in the **faces-config.xml** file.

The example below checks the user browser version for IE9. If IE9 is detected, the browser will be forced to run the application in IE 8 compatibility mode. If the browser is a previous version of IE, or if the browser is a different type, then nothing happens. Also note that the browser check is only performed once so that you don't need to worry about performance.

Example **faces-config.xml** file:

```
<?xml version="1.0" encoding="windows-1252"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee">
 <application>
  <default-render-kit-id>oracle.adf.rich</default-render-kit-id>
 </application>
 <lifecycle>
  <phase-listener>adf.sample.view.IECompatibilityPhaseListener</phase-listener>
 </lifecycle>
</faces-config>
```

Example `PhaseListener`:

```
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.myfaces.trinidad.context.Agent;
import org.apache.myfaces.trinidad.context.RequestContext;

public class IECompatibilityPhaseListener implements PhaseListener {
```

```
//defining unique session keys to test browser checking and version
//for user sessions
private final String BROWSER_CHECK_KEY =
                 "____$browser-compatibility-checked-key$____";

private final String IS_IE9_CHECK_KEY =
                 "____$IS-IE9-check-key$____";

public IECompatibilityPhaseListener() {
  super();
}

public void afterPhase(PhaseEvent phaseEvent) {
  //check if browser is IE 9 in RestoreView phase if this
  //hasn't been checked before
  if(phaseEvent.getPhaseId() == PhaseId.RESTORE_VIEW
     && !this.isBrowserChecked()){
     //which browser does the user use
     RequestContext trinidadContext =
                         RequestContext.getCurrentInstance();
     Agent agent = trinidadContext.getAgent();
     String browserName = agent.getAgentName();
     String browserVersion = agent.getAgentVersion();
     //is it IE
     if(browserName.toLowerCase().indexOf("ie") > -1){
        if (browserVersion.equalsIgnoreCase("9.0")){
          this.setIsIE9(true);
        }
        else{
          this.setIsIE9(false);
        }
      }
      else{
        this.setIsIE9(false);
      }
      //browser has been checked
      this.setBrowserChecked(true);
   }
  }

  public void beforePhase(PhaseEvent phaseEvent) {
   //check render response
   if(phaseEvent.getPhaseId() == PhaseId.RENDER_RESPONSE
      && this.isIsIE9()){
     FacesContext fctx = FacesContext.getCurrentInstance();
     ExternalContext ectx = fctx.getExternalContext();
     HttpServletResponse response =
```

```
                    (HttpServletResponse) ectx.getResponse();
      response.addHeader("X-UA-Compatible", "IE=8");
    }
  }
  public PhaseId getPhaseId() {
   return PhaseId.ANY_PHASE;
  }

  private void setIsIE9(boolean isIE9) {
    setBooleanSessionKeyValue(IS_IE9_CHECK_KEY, isIE9);
  }

  private boolean isIsIE9() {
   return getBooleanSessionKeyValue(IS_IE9_CHECK_KEY);
  }

  private void setBrowserChecked(boolean browserChecked) {
    setBooleanSessionKeyValue(BROWSER_CHECK_KEY, new
    Boolean(browserChecked));
  }
  private boolean isBrowserChecked() {
   return getBooleanSessionKeyValue(BROWSER_CHECK_KEY);
  }

  private boolean getBooleanSessionKeyValue(String _key){
    FacesContext fctx = FacesContext.getCurrentInstance();
    ExternalContext ectx = fctx.getExternalContext();
    //get user session
    HttpSession userSession = (HttpSession) ectx.getSession(true);
    Object browserCheckObject = userSession.getAttribute(_key);
    if(browserCheckObject == null){
      return false;
    }
    else{
      return ((Boolean) browserCheckObject).booleanValue();
    }
  }
   private void setBooleanSessionKeyValue(String _key, Object _value){
     FacesContext fctx = FacesContext.getCurrentInstance();
     ExternalContext ectx = fctx.getExternalContext();
     //get user session
     HttpSession userSession = (HttpSession) ectx.getSession(true);
     userSession.setAttribute(_key,_value);
   }
}
```

The nice thing about using a PhaseListener is that once you are on an Oracle JDeveloper and ADF version that supports IE9, you remove the PhaseListener configuration from your application. This is much easier to do than changing the application page sources.

Does this blog post mean Internet Explorer 9 is supported in Oracle JDeveloper 11.1.1.4 when running in IE 8 compatibility mode? No, it doesn't! However, chances are good that issues you experience in Oracle JDeveloper 11.1.1.4 using IE9 don't show in IE8 or IE7 compatibility mode.

If you still experience problems, even though you run IE9 in compatibility mode, you first need to verify this problem to exist in native IE8 before filing a bug.

Note: Thanks to Andrejus Baranovskis for testing this on IE9

Recommended number and size of Application Module(s)

Application Modules in Oracle ADF Business Components expose the data model, the View Object instances that application developers work with when building ADF applications using ADF Business Components as the business service choice. A frequent question is about the ideal number and size of Application Modules to ensure good performance. The answer to this question is that it depends on your application and the requirements you have. Remember that there are different types of Application Modules:

- Root Application Modules
- Nested Application Modules
- Shared Application Modules

Root application modules manage the transaction and therefore require their own database connection. The more root Application Modules an application uses, the more database connections – multiplied by the number of users using the application – are open at a given time. Shared Application Modules are good for sharing data queries, for example to populate list-of-values with data that is not individual to a specific user session and that does not change frequently. Using shared Application Module is like using a singleton in Java. Nested Application Modules share the database connection and transaction with the parent Application Module and are good to tailor the data model to a specific need or use case, making it easy for application developers to split the work in teams and understand functional boundaries.

So the question is not how many Application Modules you have but how many root Application Modules you use. The recommendation is to go with a minimum number of root application modules (a single one at best) and break down your application into uses of nested application modules. This *break down* could be by a one-to-one relation to use cases you identified for your application or more coarse grain logical units of work. The guidance to give therefore is to use many nested Application Modules to modularize large business services into manageable units representing a single use case or a logical unit of work, but to go with the a minimum of root application modules.

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/bcservices.htm#sthref844
http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/bcservices.htm#sm0229

About JSF fragments, ADF regions, declarative components …

Starting application development with Oracle ADF and ADF Faces, some concepts may be hard to grasp at the beginning. Using Oracle ADF and ADF Faces, the following terminologies are used in the context of reuse of components and processes

- **JSF fragments**

  JSF page fragments are page definitions that run embedded in another JSF page. Fragments are like page includes in JavaServer Pages, with the difference that in Oracle ADF Faces they are usually used in the context of ADF regions or dynamic declarative components. You can also reference page fragments directly from a JSP `includes` tag added to a JavaServer Faces document (JSPX). However, in this case, and only if a page fragment has ADF bound content, you need to make sure the content of the page fragments ADF binding file (PageDef) is copied to the PageDef file of the parent page. Otherwise ADF queried data will not show.

- **ADF regions**
  ADF regions consist of an ADF Faces `af:region` tag, an ADF bounded task flow and page fragments. Page fragments that are used in a bounded task flow don't need to copy their ADF binding references to the parent container, which is a huge difference between JSP includes and ADF regions. ADF regions define an interactive area on a view, a JSF document or another JSF page fragment, that developers use to show a single view or a complete, multi-step, process. ADF regions can be statically or dynamically defined. In either way they require a PageDef file and a bounded task flow to reference. ADF regions help building desktop like web applications in which users stay for long on a single page while working on a business task.

- **Declarative components**

  Declarative components allow developers to build a composite component out of existing ADF Faces components. Declarative components exist in two flavors: library driven and dynamic declarative components (ddc). The tag library driven components are declaratively built from the File | New menu option. In the JSF view option you find a declarative component menu option that steps you through building your own ADF F aces component from existing ADF Faces components. You use tag library driven declarative components to build custom components with behavior, like a tool bar or a custom file-upload handler. The goal of building declarative components is to build re-usable components that simplify development and administration by avoiding duplicate page codes. Dynamic declarative components (DDC) are used within the scope of the web application they are defined in and cannot be re-used across applications. Their main usage is to build reusable layout artifacts or page area components. For example, a custom tab canvas is what you would build using DDC components.

- **Page templates**

  Page templates are layout definitions that you use as a starter when building new pages to enforce consistent page layouts throughout applications and enterprises. Best practices are to build a page template using the ADF Faces Quick start templates. You cannot nest page templates, but you can use page templates on parent and child views (page fragments). A page template is the page level equivalent to a DDC component.

- **ADF Library**

  ADF libraries are special Oracle ADF archive files that you use to reuse bounded task flows (regions), page templates and declarative components. They are standard JAR files with extra information in the archive manifest file that allows you to import the library files into the Oracle JDeveloper Resource Palette for declarative reuse.

When designing an application you best start planning reuse of components and page segments. If you have an application wide look and feel that you can define as page template(s) then do this first. If you can identify areas within pages that you may need more often on other pages as well, without the pages to be identical from their layout, you use dynamic declarative components. For functionality like global toolbars or common and composite user interface logic, you build tag library based declarative components, which then can be used across applications too. An ADF region is an interactive and optionally also data centric page are that you use to show complete business processes in place. ADF regions are a friend for building rich Internet application interfaces and business centric web desktops. ADF libraries are the vehicle to deploy your reusable work.

Read more:

DDC

http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_declarativeComponent.html

Reuse

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/reusing_components.htm#BABCHHHJ

Fragments, Templates, Declarative Components

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31973/af_reuse.htm#CHDDECDG

How-to determine a task flow for the ID in dynamic region exists

When working with ADF dynamic regions, developers switch between task flows by changing the internal state of a property in the managed bean referenced from the ADF Region binding.

To read more about ADF regions, read up on this:
http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/taskflows_regions.htm#CHDJHACA

Unless there is a separate method in the managed bean to set the task flow ID for each task flow could be displayed in the region, there is no guarantee that the task flow ID that is passed to the bean matches an existing task flow. A requirement on OTN thus was to tell beforehand if a task flow exist for the task flow id specified at runtime. Unfortunately there is no public API available for this, so that the answer to this requirement is to use an internal packaged framework class that Chris Muir documented in a blog post:

http://chrismuir.sys-con.com/node/1606250

Using a method documented by Chris, you can get to the information you need for the use case discussed in this section.

```
MetadataService metadataService = MetadataService.getInstance();
TaskFlowDefinition taskFlowDefinition =
              metadataService.getTaskFlowDefinition(taskFlowId);
```

Note that starting Oracle JDeveloper 11.1.1.4, the attempt of importing internal packaged classes in customer applications is handled by a new audit rule. The audit rule prevents the class that imports the internal packaged library from compiling. So to continue using internal packaged libraries, you need to disable the audit rule as explained here:

http://blogs.oracle.com/jdevotnharvest/2011/03/internal_package_import_errors_and_how_to_switch_them_off.html

Best practices when dealing with ADF internal framework packages

Oracle is aware of that there may be a need for developers to use internally packaged classes if there is no acceptable other way, or public API to use. However, internally packaged classes and APIs may change without notice, for example for Oracle to implement enhancement requests. When working with internal classes therefore it is recommended that you

- File an enhancement request for Oracle to provide a public API for the internal API you want to use
- Create an abstraction layer, a utility class or managed bean that you use to access the internal class, instead of directly accessing the "forbidden" internal API in your application code.
- Change the internal class access in the abstraction layer to a public class once your enhancement request is implemented.

Note: I filed ER 12345520 for the use case explained in this section. The current status of it is that a fix is provided in the next major release of Oracle JDeveloper

Managed Properties: the forgotten JSF feature

As it is common practice to resolve expression in Java using a `ValueExpression`, the use of managed properties seems to be forgotten. If a managed bean requires access to another managed bean in the same or a larger scope, then instead of code like this …

```
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
ExpressionFactory exprFactory =
                    fctx.getApplication().getExpressionFactory();
ValueExpression ve = null;
ve = exprFactory.createValueExpression(
                   elctx,
                   "#{managedBeanB}",
                    Object.class);
ManagedBeanB managedBeanB = (ManagedBeanB) ve.getValue(elctx);
```

… you can use a managed bean property. For example if ManagedBeanA has needs access to ManagedBeanB, then it will have the following property and methods created

```
ManagedBeanB managedBeanBvar = null;
…
public void setManagedBeanBvar(ManagedBeanB managedBeanB) {
    this.managedBeanBvar = managedBeanB;
 }

 public ManagedBeanB getManagedBeanBvar() {
    return managedBeanBvar;
}
```

The managed bean configuration for ManagedBeanA in `adfc-config.xml`, `faces-config.xml` or bounded task flow definitions then looks as shown below

```
<managed-bean id="__7">
 <managed-bean-name id="__6">managedBeanA</managed-bean-name>
 <managed-bean-class id="__5">sample.ManagedBeanA</managed-bean-class>
 <managed-bean-scope id="__8">request</managed-bean-scope>
 <managed-property id="__16">
  <property-name id="__17">managedBeanBvar</property-name>
  <property-class>sample.ManagedBeanB</property-class>
  <value id="__18">#{viewScope.managedBeanB}</value>
 </managed-property>
</managed-bean>
<managed-bean id="__11">
 <managed-bean-name id="__10">managedBeanB</managed-bean-name>
 <managed-bean-class id="__12">sample.ManagedBeanB</managed-bean-class>
 <managed-bean-scope id="__9">view</managed-bean-scope>
</managed-bean>
```

The above configuration ensures that `ManagedBeanB` is instantiated when ManagedBeanA is instantiated so that it is accessible from code in `ManagedBeanA` through the *managedBeanBvar* variable.

So why would you want to use managed properties if you could use a `ValueExpression` to access a managed bean? There is no simple answer to this other than better readability and ease of maintenance as everything that is put in code is compiled and harder to read and change from the outside.

Whitepaper: JavaScript in ADF Faces

JavaScript stay on top of Oracle ADF customer's interest and often is requested for when it comes to integration with 3rd party applications or enhancing ADF Faces component default functionality. ADF Faces provides a client side JavaScript framework that allows developers to integrate JavaScript in their ADF applications with no need to worry about lifecycle synchronization or browser differences. Though released in January 2011 already, the JavaScript whitepaper I wrote gives you an introduction and some advanced tips for using JavaScript in ADF Faces.

http://www.oracle.com/technetwork/developer-tools/jdev/1-2011-javascript-302460.pdf

In addition, ADF Code Corner sample #71 shows how to use JavaScript to communicate between ADF Faces and a Java Applet on the client. This sample can be modified easily to work with other client side products.

http://www.oracle.com/technetwork/developer-tools/adf/learnmore/71-adf-to-applet-communication-307672.pdf

However, don't worship JavaScript for the sake of it. If there is a solution in Java or JavaServer Faces, for example to integrate with Web Services, then avoid using JavaScript.

Whitepaper: ADF application performance and scalability testing

Stress and performance testing is a question that frequently shows on OTN. This March, Stewart Wilson from Oracle published a whitepaper "Techniques for Testing Performance/Scalability and Stress-Testing ADF Applications" on OTN that I recommend reading:

http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adfloadstresstesting-354067.pdf

"This paper examines some tools and techniques for testing the scalability of ADF Faces applications, based on Oracle internal experience."

Whitepaper Update: ADF Task Flow Design Fundamentals

Duncan Mills published an update version of the ADF Task Flow Design Fundamental whitepaper that outlines proven practices when working with task flows.

http://www.oracle.com/technetwork/developer-tools/jdev/adf-task-flow-design-132904.pdf

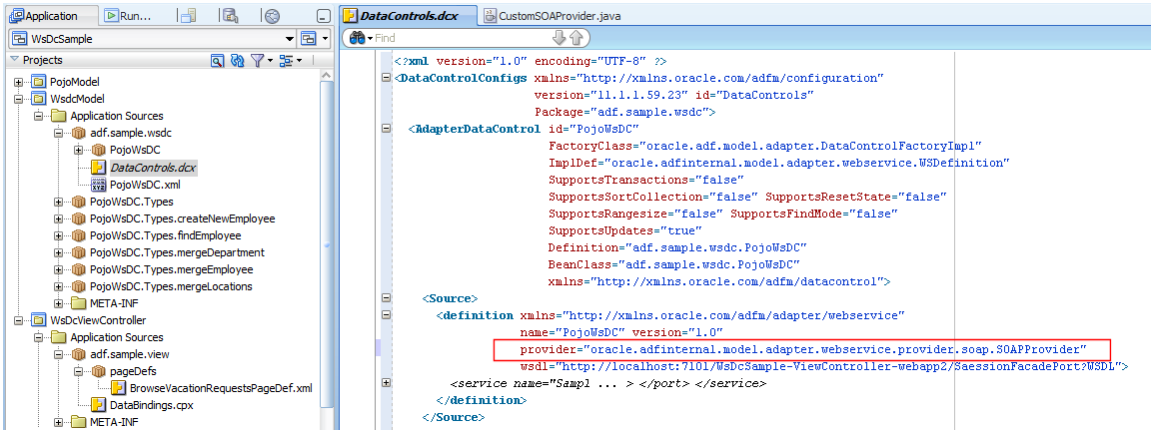
How to access the WS SOAP message using WS DC

A frequent requirement is to access the SOAP message of a service to e.g. set SOAP header information required by a service for passing license keys or authentication information. In Java, the `SOAPMessage` class gives you access to the `SoapPart`, `SOAPEnvenlope`, `SOAPBody` and the `SOAPHeader` using code like shown below:

```
SOAPPart        sp = soapMessage.getSOAPPart();
SOAPEnvelope    se = soapMessage.getEnvelope();
SOAPBody        sb = soapMessage.getBody();
SOAPHeader      sh = soapMessage.getHeader();
```

See: http://download.oracle.com/javaee/1.4/api/javax/xml/soap/SOAPMessage.html

To access incoming and outgoing SOAP messages when using the Oracle ADF WS Data Control, you need to override the default SOAP provider that is configured in the `DataControls.dcx` file of the WS Data Control project.

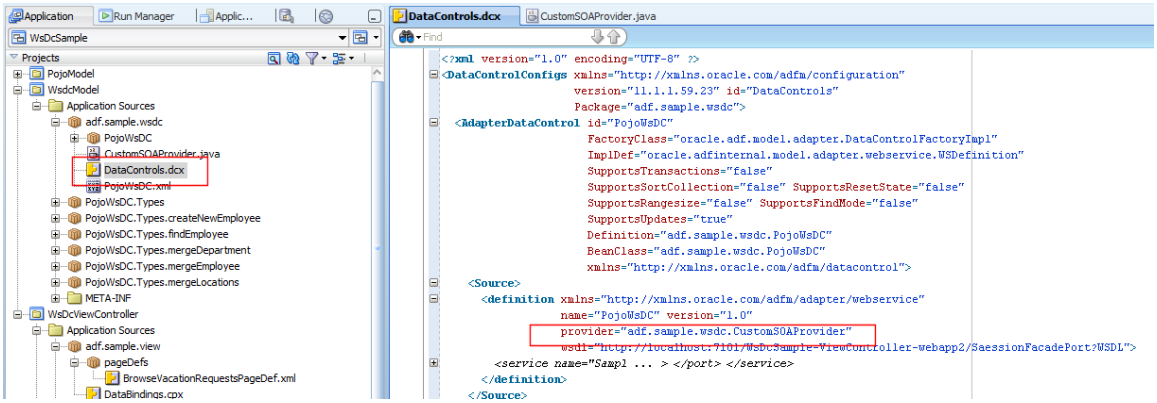


By default, the provider is configured as

```
<Source>
      <definition xmlns=http://xmlns.oracle.com/adfm/adapter/webservice
                  name="PojoWsDC" version="1.0"
provider=
     "oracle.adfinternal.model.adapter.webservice.provider.soap.SOAPProvider"
…
```

To change the provider class, extend the default `SOAPProvider` provider class as shown below

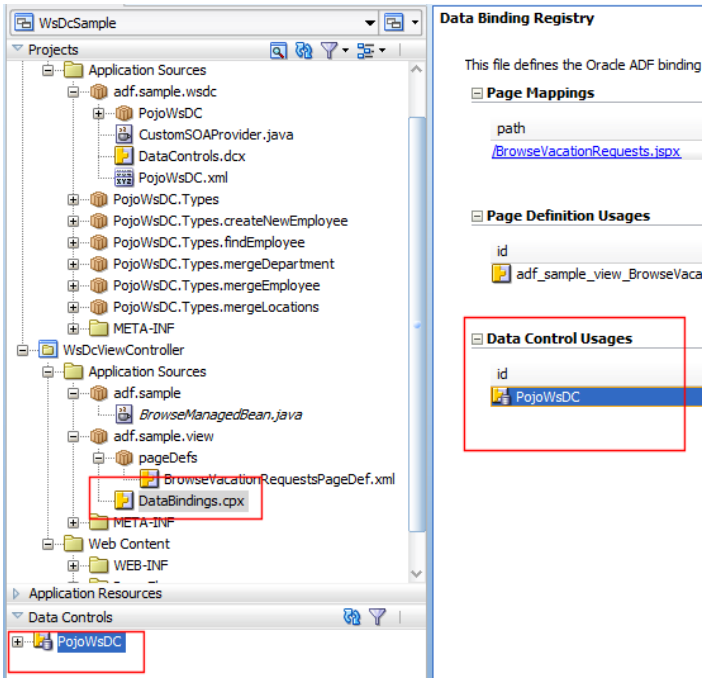
```
public class CustomSOAProvider extends SOAPProvider {
    public CustomSOAProvider() {
        super();
    }

    //expose protected method to public
    public void handleRequest(SOAPMessage soapMessage) throws AdapterException
    {
      super.handleRequest(soapMessage);
    }
    //Expose protetcted method to public
    public void handleResponse(SOAPMessage soapMessage) throws AdapterException
    {
        super.handleResponse(soapMessage);
    }
}
```



Change the DataControls.dcx configuration to use your custom provider, for example

```
<Source>
  <definition xmlns=http://xmlns.oracle.com/adfm/adapter/webservice
              name="PojoWsDC" version="1.0"
               provider="adf.sample.wsdc.CustomSOAProvider"
```

...



You can access the custom SOAP provider from a managed bean, which also allows you to expose its functionality to Expression Language. Before you can access the custom SOAP provider from a managed bean, you need to know the name of the Data Control that is used to access the Web Service. The name of the Data Control is defined in the **`DataControls.cpx`** file where it can be looked up. The file is located in the **ViewController** project.

The Data Control name can also be looked up in the Data Controls palette. This however requires that the name indicates that it accesses a Web Service Data Control, as otherwise it will be hard to tell.

You use the following code to access the custom SOAP provider from Java in a managed bean:

```
BindingContext bctx = BindingContext.getCurrent();
DataControl dc = bctx.findDataControl("PojoWsDC");
WSDataControl wsdc = (WSDataControl) dc.getDataProvider();
CustomSOAProvider customSoapProvider =
                   (CustomSOAProvider)  wsdc.getProvider();
```

If you are using JDeveloper 11.1.1.4 or later, because the `SOAPProvider` class is in an ADF internal package, you need to be aware of the ADF internal Java class audit rule and how to switch it off:

http://blogs.oracle.com/jdevotnharvest/2011/03/internal_package_import_errors_and_how_to_switch_them_off.html

But why do you need to create a custom SOAPProvider extending the default provider?

The default `SOAPProvider` class has the `handleRequest` and `handleResponse` methods defined as protected for security reasons. To make these methods available for the ViewController to use, you need to expose them as public methods, which is what the custom SOAPProvider class primarily is for.

Note: I filed an ER for a public access to the `SOAPProvider` class.

Passing parameters to managed bean methods using EL

No, you cannot pass arguments to a managed bean method using Expression Language. But you can work around this limitation. To make required arguments available to a managed bean method, you either use in memory attributes, the ADF binding layer or a setter/getter method on a managed bean that sets an internal variable.

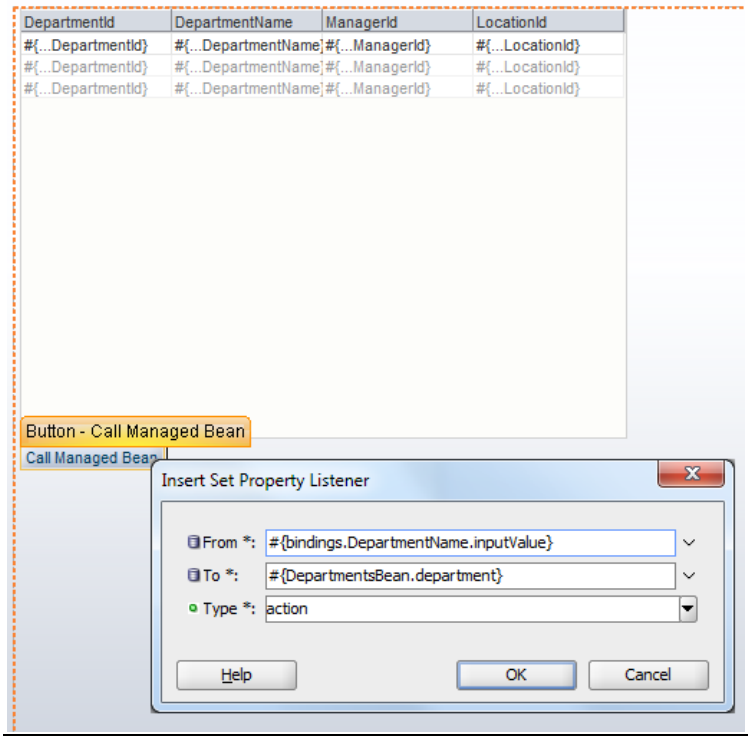
Sample setup: The managed bean method `onCallManagedBean` used in the following is referenced from a command button action property. The page also contains an ADF bound table that I use to access the department name of the current selected row to pass it as an argument to the managed bean method.

Option 1: Call a managed bean method to set properties

In this use case, the managed bean itself has a property *department* defined, which value can be set through a pair of setter/getter methods.

To pass the department name of the selected table row to the managed bean in the sample below, I created an attribute binding for *DepartmentName*.

Then, in an `af:setPropertyListener`, I referenced the department name attribute binding using Expression Language, to pass its value to the managed bean property **department**. The **department** property is a private variable in the managed bean that is exposed by a public setter/getter method pair.



The managed bean code is shown below:

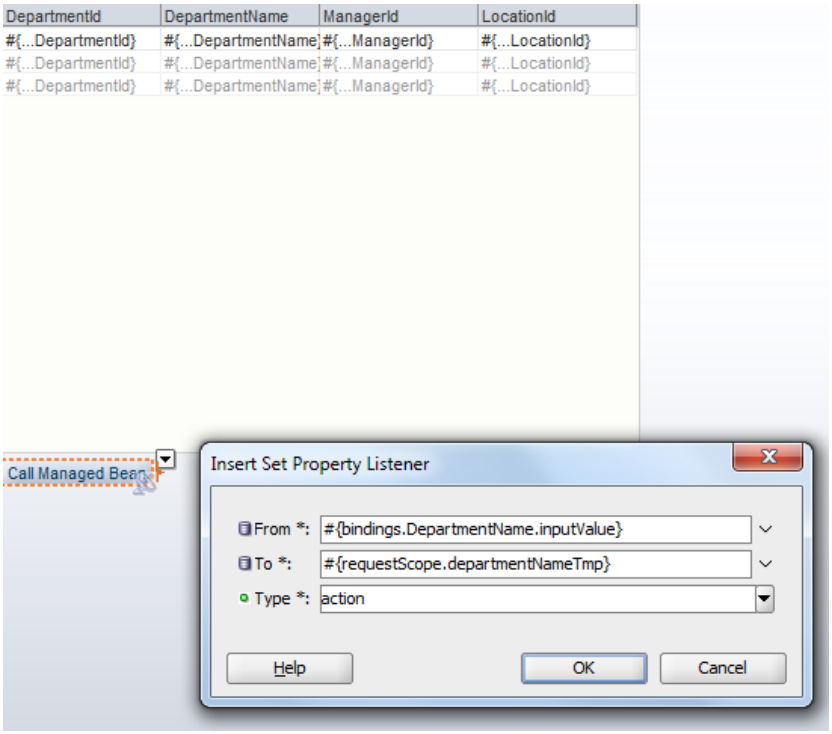
```
 String department = "";
public DepartmentsBean() {}


public String onCallManagedBean() {
    System.out.println("The Department Name is "+department);
    return null;
}

public void setDepartment(String department) {
  this.department = department;
}

public String getDepartment() {
  return department;
}
```

Option 2: Use memory attributes to pass arguments to a managed bean

Option 2 is similar to Option 1, except for that it does not use a property in the managed bean to hold the argument required by the invoked method, but a memory attribute.

The recommendation when using memory attributes is to go with the shortest scope so that whatever data is stored in memory gets cleared out at earliest opportunity. In the use case of passing an argument to

a managed bean method, **request scope** is good enough to use. For most of the requirements for passing data in ADF Faces, you don't need to go larger than request scope or view scope.



(!) Note that the **prefix** "requestScope" is only needed when writing directly into a memory scope. If you access a managed bean in the same scope then this prefix should not be used. Managed beans are only referenced with a scope prefix if they are defined in backingBeanScope, viewScope or pageFlowScope as these are custom ADFc scopes. For servlet scopes like request, session and application, using a prefix will cause NPE for when you access a managed bean that hasn't been instantiated before. So for all regular servlet scopes, don't use a prefix when accessing managed beans.

When clicking the command button, the department name that is read from the binding layer is copied into the *departmentNameTmp*memory attribute. To access this property in a managed bean, assuming you use ADF, the `ADFContext` object is used.

```
public String onCallManagedBean() {
  String department = "";
  ADFContext adfContext = ADFContext.getCurrent();
  Map requestScope = adfContext.getRequestScope();
  department = (String) requestScope.get("departmentNameTmp");
  System.out.println("The Department Name is "+department);
  return null;
}
```

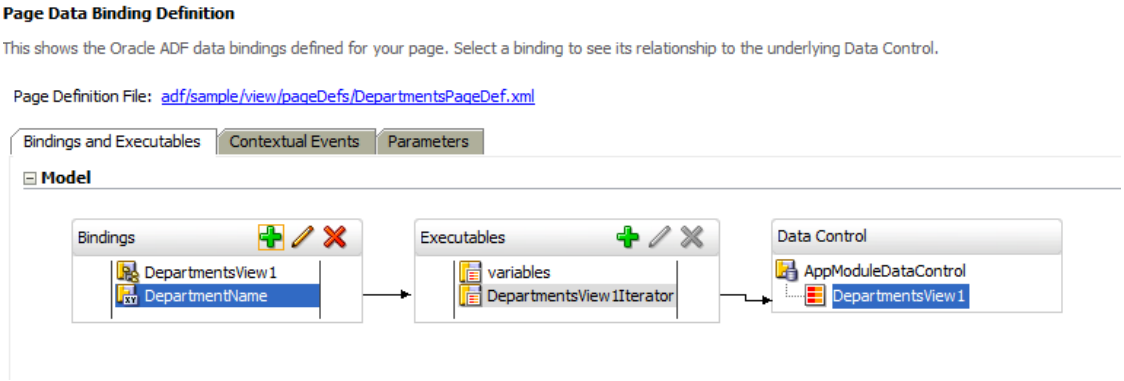
If you are not in an ADF environment, then the request scope Map is accessible from the `FacesContext | ExternalContext` in JavaServer Faces

Option 3: Access the ADF binding layer to obtain the required information

If the data used by the managed bean method comes from the ADF binding, as in this sample, then you can also directly access the binding layer without copying the values temporarily somewhere else. The two option explained earlier are good to use if the arguments to be passed to a managed bean method don't come from the binding layer but from somewhere else.

```
public String onCallManagedBean() {

  String department = "";
  BindingContext bctx = BindingContext.getCurrent();
  BindingContainer bindings = bctx.getCurrentBindingsEntry();
  AttributeBinding departmentNameBinding =
            (AttributeBinding) bindings.get("DepartmentName");
  department = (String) departmentNameBinding.getInputValue();
  System.out.println("The Department Name is "+department);
  return null;
}
```

The managed bean code accesses the ADF binding layer through the `BindingContext` class that is a runtime representation of the `DataBindings.cpx` file. It looks up the current binding container to access the value binding defined for the *DepartmentName*.



In summary: Expression Language in JavaServer Faces 1.2 does not support passing arguments to methods in a managed bean. Not a big deal though as you can pass the required information using memory attributes, managed bean properties of the ADF binding layer.

How-to switch the application locale at runtime

The application user language is set by the local browser language settings. Both, ADF and JavaServer Faces look up the local browser settings to find translated source strings.

To learn how to handle internationalization in ADF, read chapter 21 of the Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework. This chapter also contains information about how to change the language at runtime, overriding the locale definition read from the user browser setting.

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31973/af_global.htm#CIHIJJDG

How-to invoke the ADF select event from Java

This question truly is an evergreen! When you create an ADF table or tree, Oracle JDeveloper generates an expression similar to `#{bindings.treeBindingName.makeCurrent}` for the component **SelectionListener** property.

http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_table.html

To override one of the default listener settings, preserving its functionality, you can call the default ADF expression from Java, as shown below

```
public void handleTableSelection(SelectionEvent selectEvent){
  FacesContext fctx = FacesContext.getCurrentInstance();

  ELContext elctx = fctx.getELContext();
  ExpressionFactory expressionFactory =
                  fctx.getApplication().getExpressionFactory();
  MethodExpression methodExpression = null;
  methodExpression = expressionFactory.createMethodExpression (
            elctx,
            "#{bindings.treeBindingName.makeCurrent}",
            Object.class,
            new Class[]{SelectionEvent.class});
  methodExpression.invoke(elctx, new Object[]{selectEvent});
}
```

Similar code can be used for all sorts of component events, for example the table query listener, that are pre-configured in ADF.

For the table selection listener use case, you can also write a generic solution that does not contain the expression string in it. An example for a generic table listener is documented as sample #23 on ADF Code Corner:

http://www.oracle.com/technetwork/developer-tools/adf/learnmore/23-generic-table-selection-listener-169162.pdf

A generic listener has no dependency to the binding layer, making it suitable for reusable in a shared library. Though writing a generic listener is more work (and not always possible) investigating into it may produce an advantage because of increased reuse.

Another example of the table select use case, which uses the ADFUtils library for simplified EL access is documented here:

http://www.adftips.com/2010/11/adf-ui-selectionlistener-example-for.html

ADF Security authentication providers

ADF Security delegates authentication to the Java EE container, which in Oracle Fusion Middleware is Oracle WebLogic Server. At design time however, you use `jazn-data.xml` in ADF Security to create users and user groups for testing. At runtime in Oracle JDeveloper, the users and groups are deployed to the integrated WebLogic Server and configured in the integrated LDAP provider.

In a production environment however, you may want to configure authentication to be checked against a 3rd party LDAP server, the database or SAML. WebLogic Server authentication can be configured to use one or many of the authentication providers listed below

- Oracle Internet Directory
- Oracle Virtual Directory
- iPlanet
- Active Directory
- Open LDAP
- Novell
- generic LDAP
- RDBM SQL
- WebLogic Server integrated LDAP
- Windows NT
- SAML

In addition, if you don't find the identity provider of your choice in this list, you can write a custom JAAS Login Module and configure it as an authentication provider in WebLogic Server.

So in summary, while at design time, ADF Security authenticates against the integrated LDAP server in WebLogic Server using users and groups defined in the application `jazn-data.xml` file, at runtime you can use a variety of LDAP servers, RDBMS and other identity management systems to hold enterprise users and groups.

To learn more about authentication providers in WebLogic Server and how to configure them, see the "Configuring Authentication Providers" chapter, which is part of the Oracle FMW documentation library on OTN

http://download.oracle.com/docs/cd/E17904_01/web.1111/e13707/atn.htm

ADF tree binding vs. table binding

In Oracle JDeveloper 11g, the ADF Faces tree, tree table and table components are bound to the ADF tree binding in the pageDef file. Looking at the choice of ADF bindings there also exists a specific table binding, leading to the question of what this is good for and why it is not used for binding tables. Since Oracle JDeveloper 11g, the tree, tree table and table components are bound to the ADF tree binding, or, at runtime, to the `FacesCtrlHierBinding` class. In Oracle JDeveloper releases before 11g, the table used a specific table binding, which still exists for backward compatibility reasons. The bottom line is that the table binding in Oracle JDeveloper 11g is a legacy binding that you should not use outside of migrated ADF 10.1.x style applications.

Using af:resource tag in page fragments

The `af:resource` tag is an ADF Faces tag that allows developers to add or reference JavaScript and CSS resources from a page. Given that ADF Faces uses the skinning framework for applying CSS, the use of the `af:resource` tag to reference or add JavaScript is more common.

The tag documentation mistakenly states that the `af:resource` tag should be a direct child of the `af:document` tag, which excluded its use in page fragments, which appeared limiting for developers that use JavaScript in ADF regions. So far the recommendation was to use the `trh:script` tag instead, which then added another tag library dependency to the project.

So the official stance thus is that you can use `af:resource` in page fragments as well. The JavaScript sources then load when the page fragment loads.

Using multiple Data Controls in ADF applications?

Especially when working with ADF Business Components you may only a single Data Control being used when working within an application. Because the ADF Business Components Data Control is used a lot in written collateral, guided how-to and samples, as well as video recordings, it neglects the fact that you can of course work with multiple Data Controls in an ADF application.

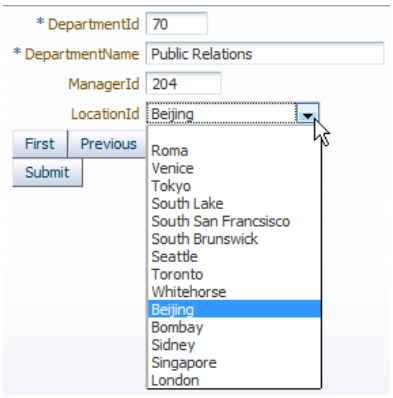
There is no restriction other than developer wisdom in the number of Data Controls to use within a project. Data Controls can also be of different types and, if needed be used to pass data from one business service to another based on user interaction. If you need to look up and access a Data Control from Java, you can do so by calling `BindingContext.getCurrent()` to obtain a handle to the runtime object that represents the `DataBindings.cpx` design time file. On the binding context you then call `findDataControl(String)` to access the data control, where the "String" argument is the ID the data control has in the `DataBinings.cpx` configuration.

When creating new bindings declaratively in the PageDef file of a page or a view, you can select from a list of available Data Controls defined in the `DataBinings.cpx` file.

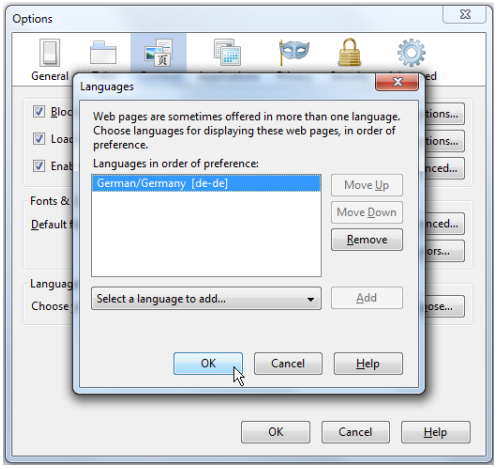
Note however, that not all Data Controls need to be defined in the `DataBinings.cpx` file, as they can also be imported as part of an ADF Library.

Creating localized static list of values

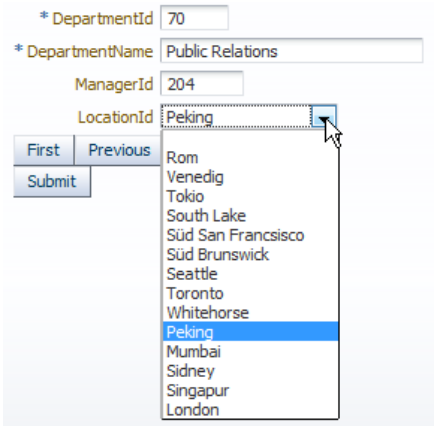
Using ADF Business Components and model drivel list-of-values, it is easy to create localized static list of values. In the example below, the **LocationId** attribute is populated by a list of values that by default shows English labels.



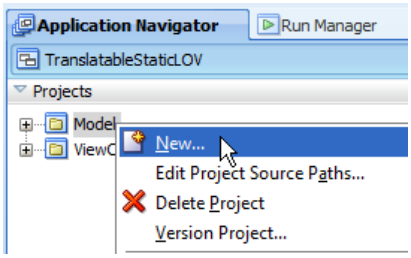
Changing the browser locale from English to German and re-running the page …



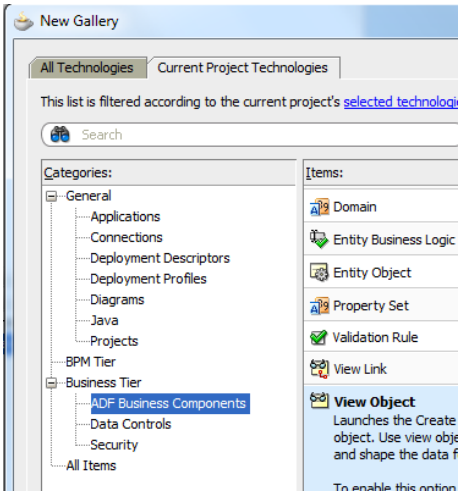
Then shows the same list of values with German labels, which you can tell easily by the Chinese capital city name being "Beijing" in English, while in German it is "Peking".



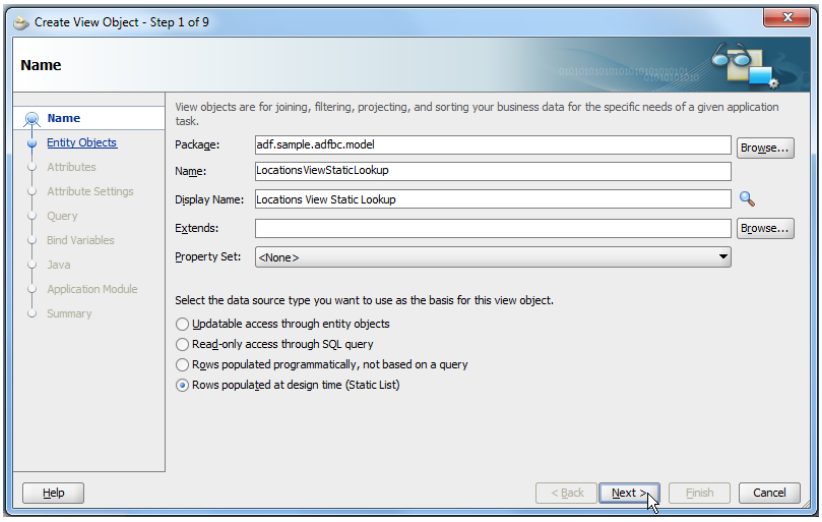
The list-of-values in this sample is created as a static View Object. To create the view object, choose **New** from the context menu while pointing to the ADF BC model project.



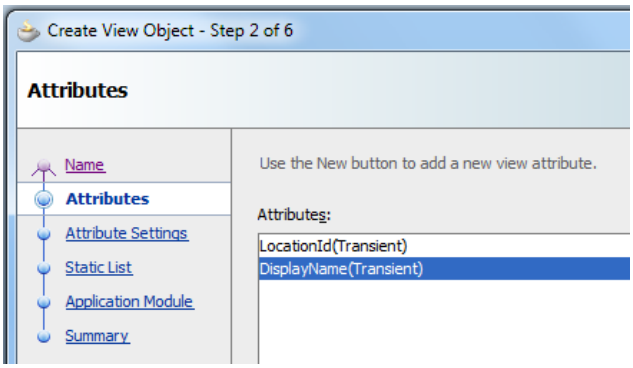
Select the ADF Business Components node and choose **View Object**. Press **Ok**.



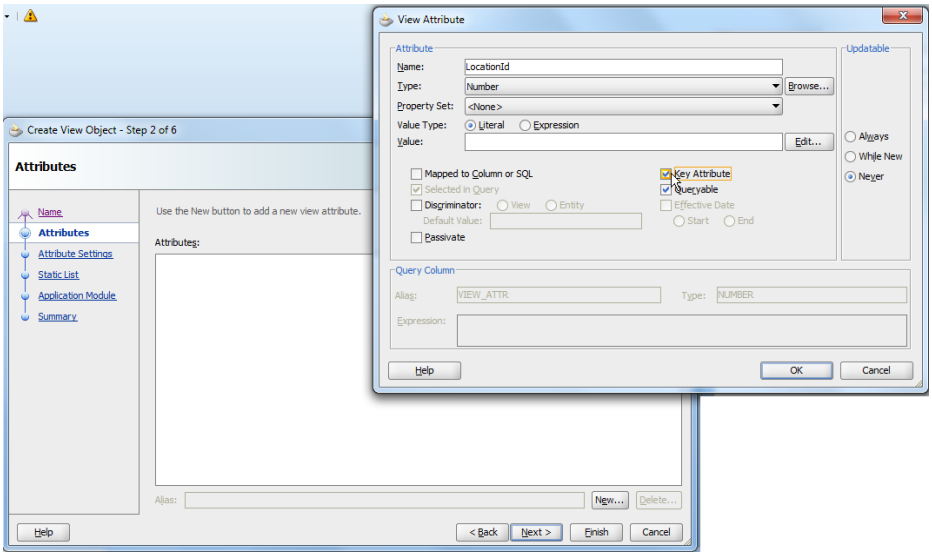
In the **Create View Object** dialog, define a name for the view object and select the **Rows populated at design time (Static List)** option.



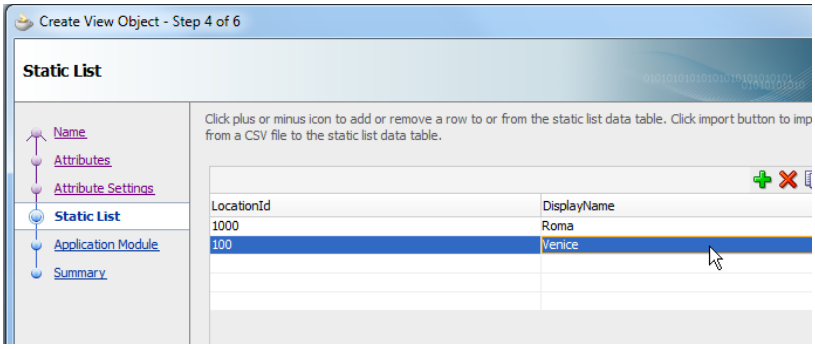
Define at least two attributes, one for the list value and one for the display label.



For each attribute, define the attribute data type and, **important (!)**, ensure one of the attributes has the **Key Attribute** select box checked.



In the next dialog, provide the static data values for the view object attributes.

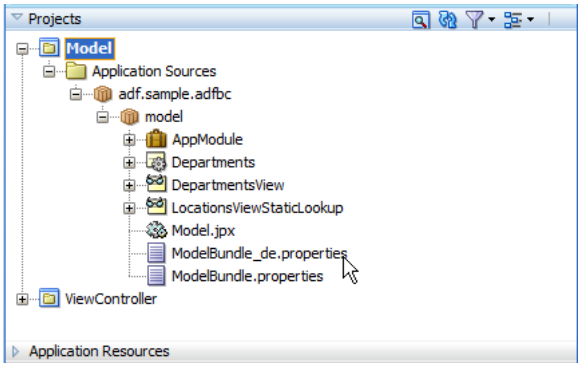


When you are done with the data edits, finish the dialog for Oracle JDeveloper to produce the view object. Note that the view object does not need to be part of an Application Module if you only want to use it as a data source for a list of values.

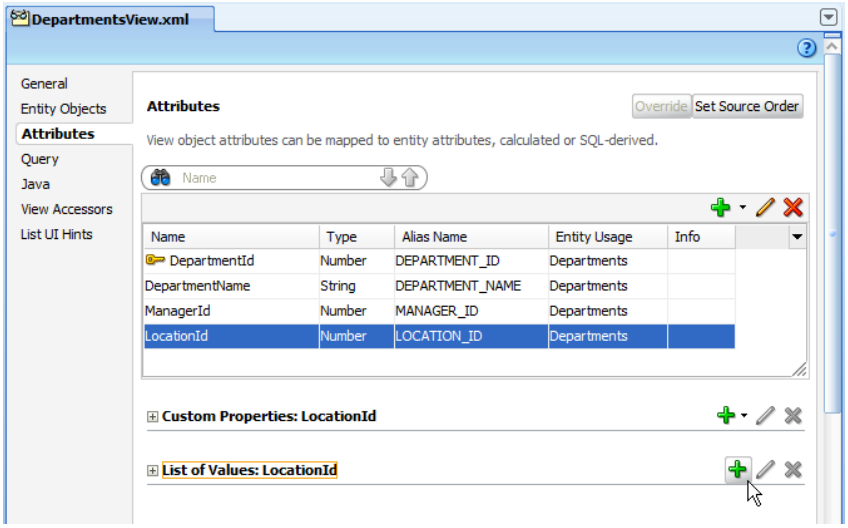
As part of the view object creation, a ".properties" file is created that contains the static values you defined for the view object attributes. To create a translation, copy the **ModelBundle.properties** file

content and create a new file in the same directory that you name **ModelBundle_<lang>.properties**, for example: **ModelBundle_de.properties.**

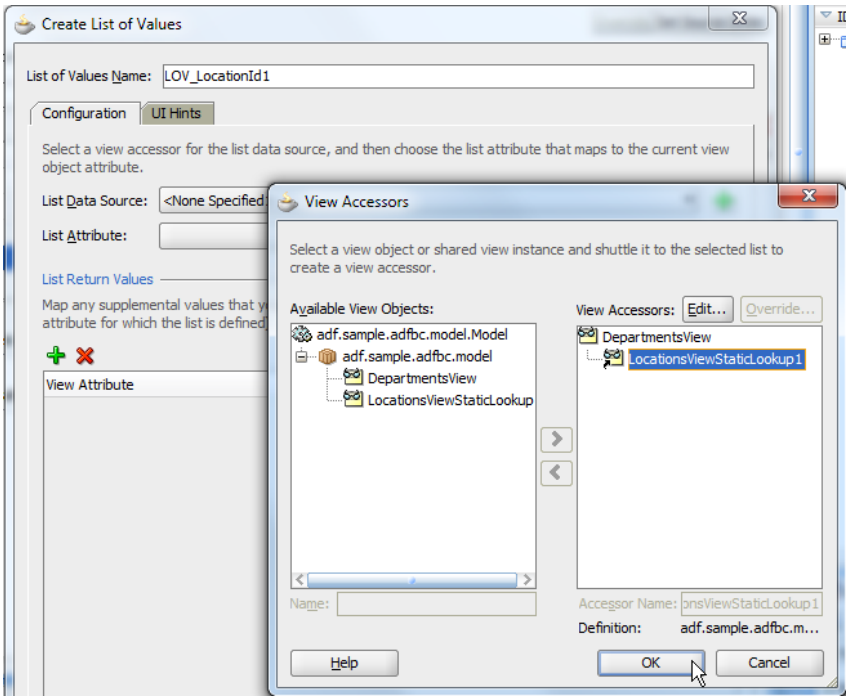
Note: you can always add more static data. In this case however, you only copy and paste the new values and labels to the localization files.



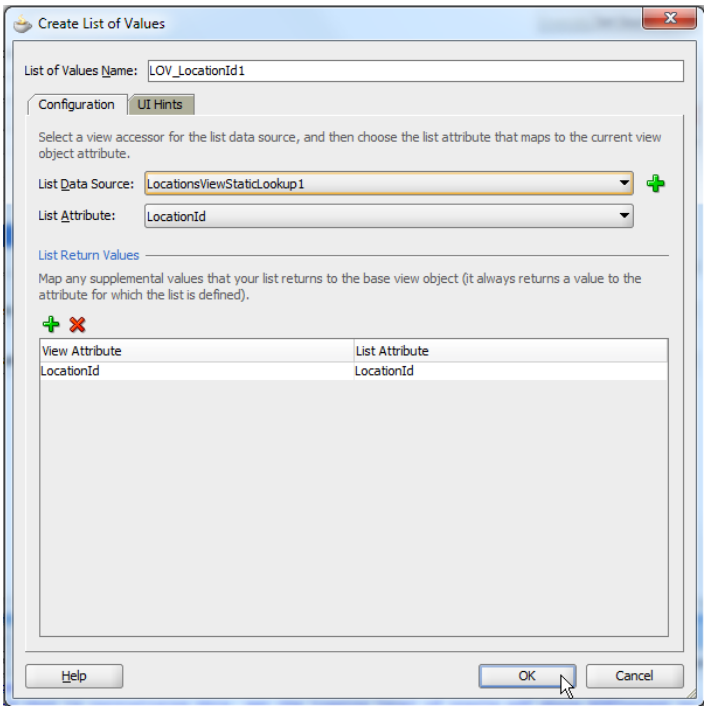
Open the View Object editor for the view object that should have the localized list of values assigned. Select the **Attributes** category, choose an attribute and press green plus icon next to the **List of Values** header.



Next to the **List Data Source** select list in the **Create List Value** dialog, press the **New** button to bring up the **View Accessors** dialog. Select the static View Object instance and click **Ok** to close the dialog.

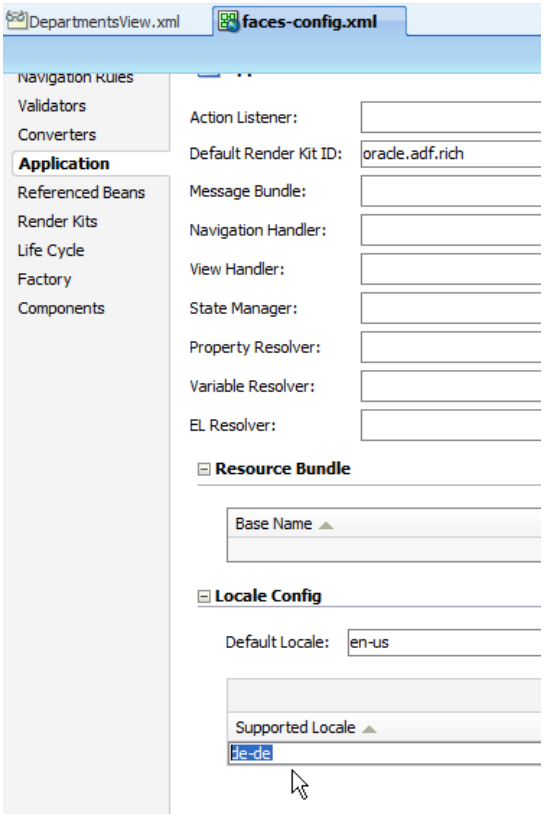


Back in the **Create List of Values** dialog, you select the **List Attribute**, which is the attribute that should be used to update the target view object. Click on the **UI hints tab** to select the other attribute of your static View Object to show as the list label. You don't need to change anything else because the default list of values component setting is the select one choice.



Open the **faces-config.xml** file in the **ViewController** project and select the **Application** category. Here, edit the default locale, as well as any additional **Supported Locale**. In this sample, German (de-de)

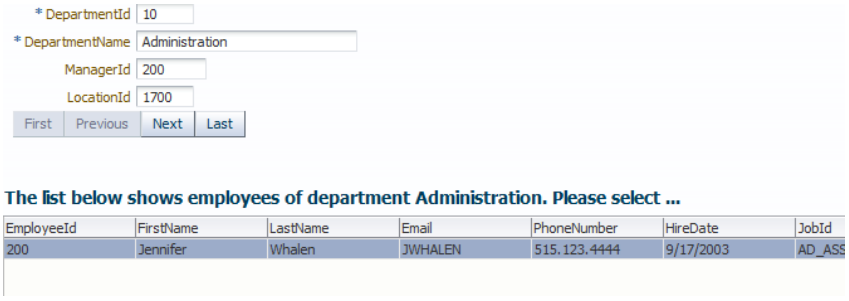
is added as a supported locale, which means that all users see the English version except for those that have their browsers configured with the German locale.



Using parameterized translation strings in ADF Faces

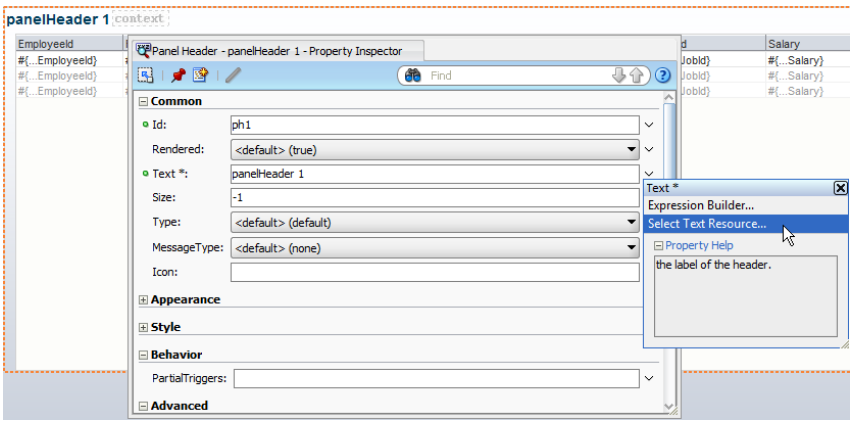
To define translatable labels and messages in ADF Faces that don't reference the ADF binding layer to obtain the translated string values from the model, you use a resource bundle in the view layer project.

For example, to create a translatable title string for a **Panel Header** component with parameterized values, as shown below, …

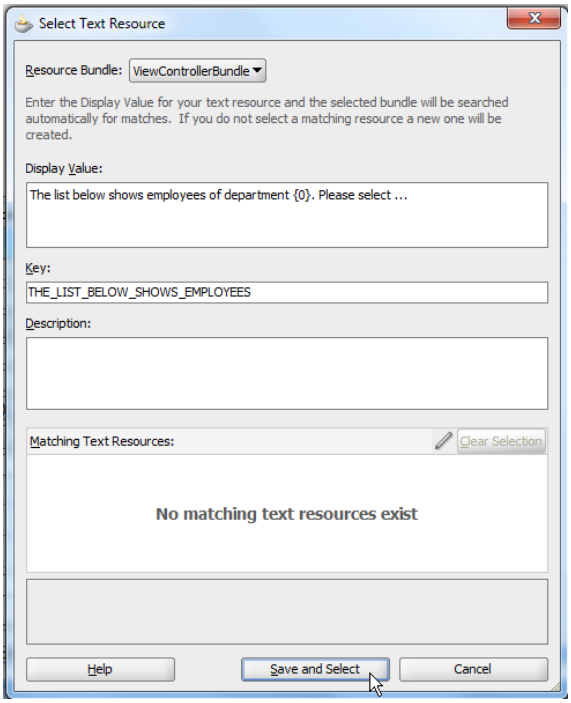


… you do as follows:

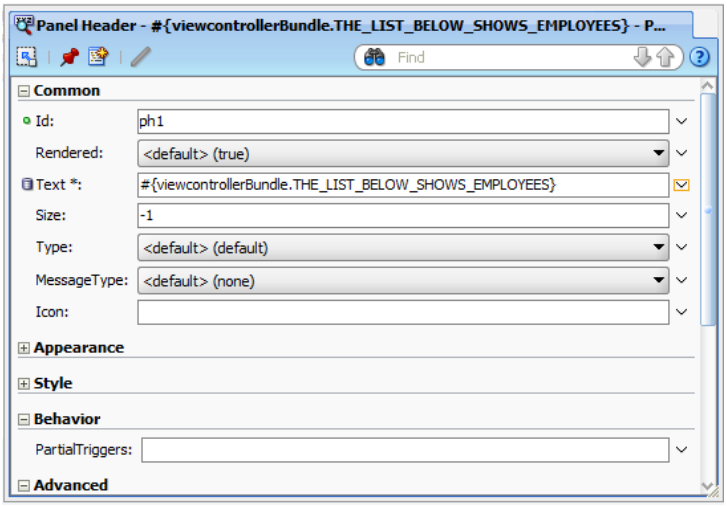
Select the component's message property, like *Text* in the Panel Collection sample. From the context menu, select the **Select Text Resource** option.



In the **Select Text Resource** dialog, start typing the message that you want to show as the translated component string. It automatically derives unique key from the entry, which then is used later to identify the string for re-use on other pages.



If your message needs to include dynamic data, add positional parameters in the form of {n}, like {0} in the sample string shown above.



The expression references a page variable **viewControllerBundle**, which is added to the page the first time you create or reference a resource bundle string.

```
<c:set var="viewcontrollerBundle"
  value="#{adfBundle['adf.sample.adfbc.view.ViewControllerBundle']}"/>
```

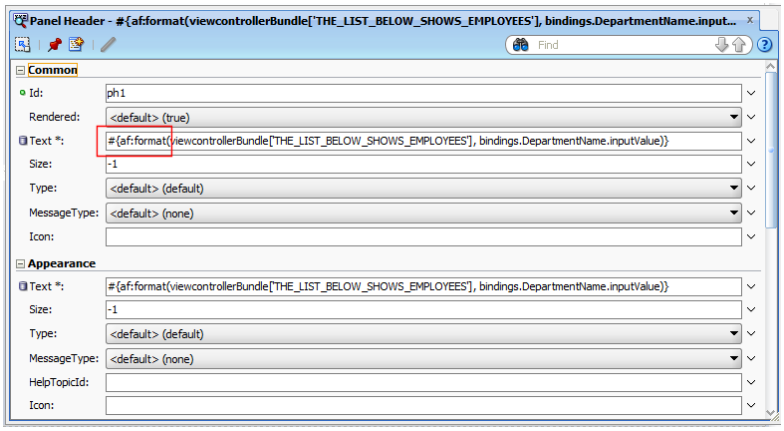
The EL string referencing this bundle looks similar to:

```
#{viewcontrollerBundle.THE_LIST_BELOW_SHOWS_EMPLOYEES}
```

If the translated message only contains static data, then following the declarative approach would be all you need to do.

In the example however, the Panel Header component title should also display the department name, which changes when a user navigates the list of departments. So this information needs to be passed in dynamically, which is what the {0} parameter is for.

To pass positional parameter values to the translation text defined in a message bundle, regular JavaServer Faces components use nested child `f:param` tags. ADF Faces however doesn't support the `f:param` tag as a component child and instead uses special EL expressions, `af:format`, `af:format2`.

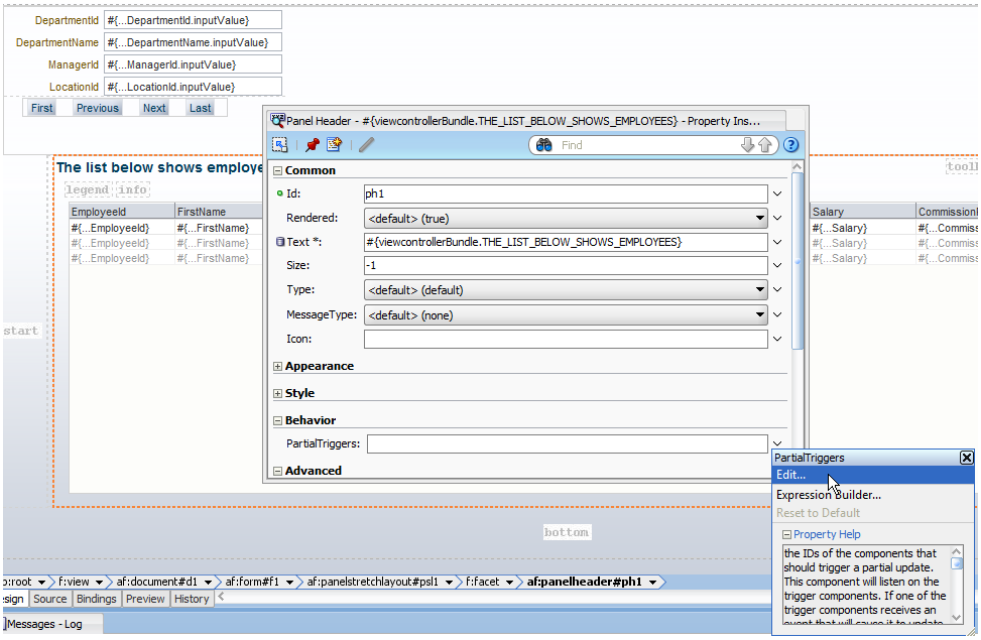


http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/toc.htm

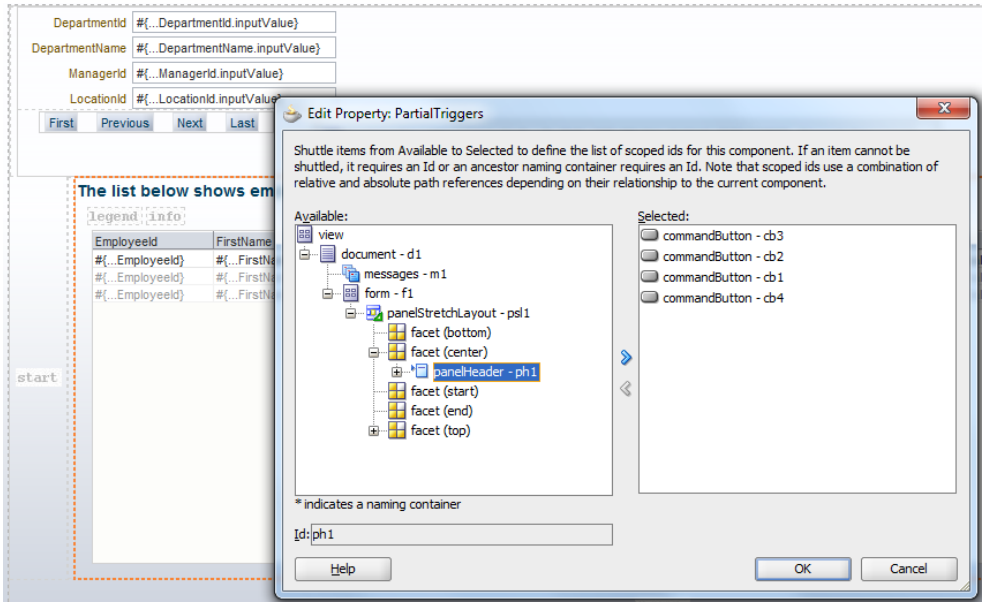
The numeric suffix e.g. in `af:format2`, `af:format3` indicate the number of parameters to be passed into the message. The tag usage pattern is that the first argument indicates the message key.

Other arguments are added separated by commas and provide the values for positional or named message parameters.

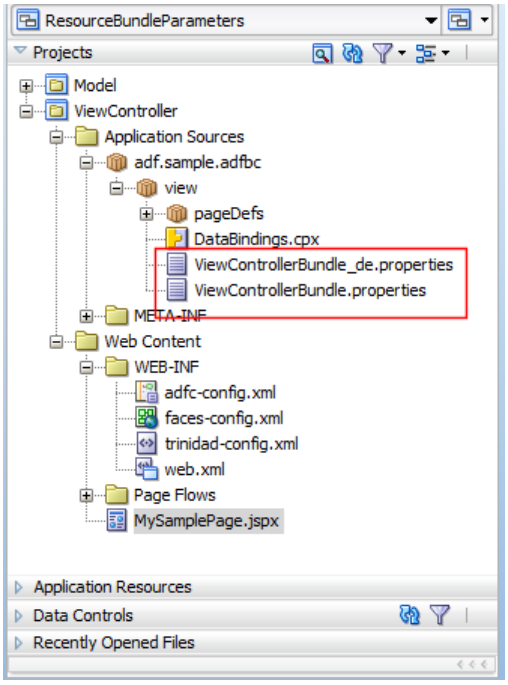
In the example, to make the Panel Header component refresh when users navigate the department list, the **PartialTriggers** property of it needs to be configured to point to the navigation buttons.



After finding and selecting the command buttons and pressing OK, the **PartialTriggers** property is setup for the sample to work.



What's left is to create translated copies of the resource bundle file that Oracle JDeveloper created. The bundle file is in a **Properties** format by default, but can be changed to Java or XLIFF formats. The image below shows a German version of the original bundle.



The `faces-config.xml` JavaServer Faces configuration to support multiple application languages is explained in the previous section "Creating localized static list of values" and, therefore, is missing in this section.

Read more about internationalization of ADF Faces applications in chapter 21 " Internationalizing and Localizing Pages" of the Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31973/af_global.htm#CHDGCAFI

Note: If internationalization is you concern, I recommend chapter 18 of the "Oracle Fusion Developer Guide - Building Rich Internet Applications with Oracle ADF Business Components and Oracle ADF Faces" book published by McGraw Hill http://www.mhprofessional.com/product.php?cat=112&isbn=0071622543

Also note: When writing this OTN Harvest entry, I recognized two product issues that I filed bugs for:

- A documentation bug to better document the `af:format` tags
- An IDE rendering issue in the visual editor when the `af:format` tag is added

Integrating ADF and Servlets

Back to his technical roots, Duncan Mills found time again for blogging. In his first post he documents a sample of how to share the ADF context with a Servlet.

http://blogs.oracle.com/groundside/2011/04/integrating_adf_and_servlets.html

The idea is to make ADF application resources accessible to a servlet, which though not being a page, can have access to the Data Control and the Oracle ADF binding if you know how.

"This code demonstrates how a Servlet within an application can share the same data control context (frame) as the underlying UI pages and Task Flows within that application. This approach is useful when you are creating integrated applications where servlets are leveraged to add functionality to the application such as AJAX calls or email generation."

As Duncan mentions in his blog, the question also came up a while ago on the OTN forum, though in a different context. For those who are only interested in how the ADF integration works for Servlets, here's a brief summary

- You manually edit the `DataBindings.cpx` file and map the Servlet path to the same pageDef as the required ADF content
- You then edit the `web.xml` file to register the Servlet with the ADF Binding filter
- The "calling" page (or taskflow) will temporarily store the current Data Control Frame name on the session
- The Servlet retrieves the data control frame name and use that to access the binding that is required in the correct context

To download Duncan's sample, follow the link below and authenticate with your OTN account.

https://www.samplecode.oracle.com/tracker/tracking/linkid/prpl1004/remcurreport/true/template/ViewIssue.vm/id/S734/nbrresults/13

RELATED DOCOMENTATION

☒	
☒	
☒	