

# ADF Code Corner

Oracle JDeveloper OTN Harvest 03 / 2011



[twitter.com/adfcodecorner](http://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](http://twitter.com/fnimphiu)  
31-MAR-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>*

## March 2011 Issue – Table of Content

Best-practice for follow-up questions on OTN forums.....	3
How-to display JavaDocs for methods displayed in syntax help .....	3
"Internal Package Import" errors and how to switch them off.....	4
Building model driven dependent list with Oracle ADF BC.....	6
How to display a dependent list box disabled if no child data exist ...	12
Testing bounded task flow using page fragments.....	14
Oracle JDeveloper command line arguments.....	14
Task flow "new transaction" vs. "new db connection" .....	14
Configuring the ADF BC locking behavior in JDeveloper 11.1.1.4 ....	18
How-to filter table filter input to only allow numeric input .....	18
Best practices about creating and using backing beans .....	22
Extending the ADF Controller exception handler.....	22
How to create a model-driven multi column auto-suggest list .....	23
How-to delete a tree node using the context menu.....	26
How to open the LOV of af:inputListOfValues with a double click.....	30
Configuring projects for Java EE security annotations.....	32
Implementing Query pagination using EJB and ADF .....	33
How to equally stretch multiple table columns .....	34

## Best-practice for follow-up questions on OTN forums

I recently recognized users on the OTN forum to post a question to then, when answers are coming in, change subject to follow up questions that are not related to the previously asked question. The problem with changing subject in an OTN thread is that the follow up questions are stealth and not seen by many on the forum who don't read until the end of a thread but go with the subject mention in the header.

In addition, those who provided a correct answer to the original question, considering the question as answered and move on. So the obvious negative impact of stealth follow questions in a forum thread is that no one looks at it no matter how hard user bump it back to the top of the list. Therefore, if you have a follow up question on an original question that however changes subject, post it in a new thread. Its five minutes of your time to re-phrase the question to the new subject saving you days you spend waiting with no answer.

## How-to display JavaDocs for methods displayed in syntax help

When working within the Oracle JDeveloper Java code editor, syntax help is displayed when pausing your edits after adding a dot (".") or when pressing ctrl+blank key for the incomplete statement.



However, unless you are savvy with the component API you are working with, not all the methods may speak to you. To get an idea of what a specific method can do for you, you can enable quick Java docs to be displayed. To open the Java documentation for the selected method, click the little plus icon next to the *QuickDoc* label at the lower right corner of the method completion dialog as shown in the image below.



The Java doc window stays open and changes its content with you changing the selection on the method dialog. To close the Java documentation window, click the minus icon next to *QuickDoc* label or press the ctrl+d keyboard shortcut.

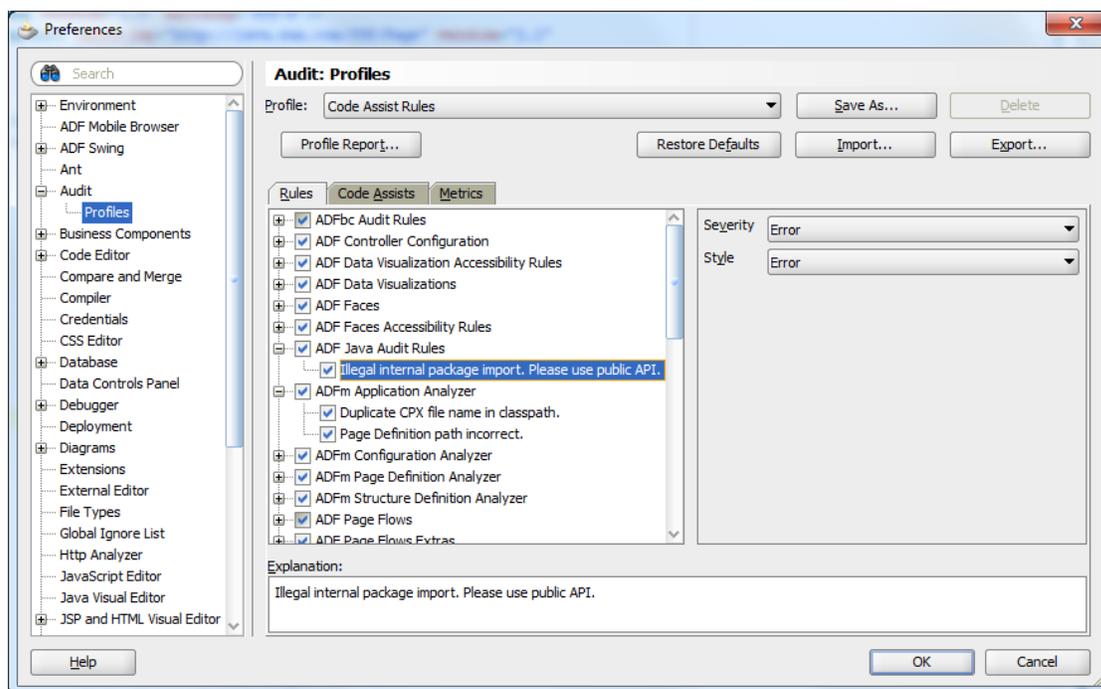


## "Internal Package Import" errors and how to switch them off

A new functionality in Oracle JDeveloper 11g (11.1.1.4) is an audit rule that flags an error when compiling Java files that use internal ADF framework classes. Internal ADF framework classes are public classes that reside in internal packages. For example, `FacesCtrlHierBinding` extends `JUCtrlHierBinding` and represents the component model used with the ADF Faces table, tree and tree table components. It is an internal implementation class that developers should not use in their application development. For this reason, Oracle packaged it in a package structure with the name **internal** in it:

```
oracle.adfinternal.view.faces.model.binding.FacesCtrlHierBinding.
```

Similar package structures exist for Oracle ADF Business Components, ADF Controller and other technologies in Oracle ADF. In previous versions of Oracle JDeveloper 11g, this audit rule did not exist, which means that without noticing, developers may have used those classes, which now, after upgrading ADF applications to Oracle JDeveloper 11.1.1.4, no longer compile, because the new audit rule prevents it from compiling. So there are reasons for you to want the audit rule to change. To change the audit rule settings for internal framework class uses, to either disable (less recommended) or smoothen it (more recommended) by setting the Severity to **Warning** instead of **Error**, you choose **Tools | Preferences | Audit | Profiles**. In here you expand the **ADF Java Audit Rules** node to change the settings for the internal package import or disable it.



Before disabling this audit rule or change it from **Error** to **Warning**, it is important that you understand why this audit is there. Like the Java and Java EE platforms, application development frameworks consist of public APIs and internal implementation classes. While in the normal Java case you protect internal implementations by flagging classes as private and protected, or using inner classes, you can't always do the same in frameworks because the classes may be referenced within the framework, for which they need to be public. Implementation details are subject to change, which means that there is not notification sent out ahead of time before a change happens. Changes may be required for example to add new features, fix

bugs or integrate new technologies. Look at the internal classes as "a framework developer's freedom to change" and you get an idea for what they are.

#### What should you do if you used internal classes in your existing application?

1. Set the audit rule to **Warning** so your project compiles.
2. Take a note about the list of issues found by the audit rule
3. Look at each use of internal class uses and see if you find public classes to use instead. For example, the `FacesCtrlHierBinding` class can be replaced by `JUCtrlHierBinding` for most of its functionality
4. If you can't find a public API, report this as a problem to customer support for Oracle to provide a public API for the functionality you need to access in Java.

**Important note:** The Oracle JDeveloper forum on OTN is not Oracle support

5. For the time being and to avoid using internal classes, use `ValueExpressions` or `MethodExpressions` in Java and access the internal functionality through their expression. For example, instead of calling `makeCurrent` on `FacesCtrlHierBinding`, you can resolve the EL string `#{bindings.treeBindingName.makeCurrent}` as a method expression in a managed bean method referenced from a `SelectionListener` property of a table:

```
public void onSelectTable(SelectionEvent selectEvent) {
    FacesContext fctx = FacesContext.getCurrentInstance();
    ELContext elctx = fctx.getELContext();
    ExpressionFactory exprFactory =
    fctx.getApplication().getExpressionFactory();
    MethodExpression me = exprFactory.createMethodExpression(
        elctx,
        "#{bindings.treeBindingName.makeCurrent}",
        Object.class,
        new Class[]{SelectionEvent.class});
    me.invoke(elctx, new Object[]{selectEvent});
}
```

#### What should you do if you need to use internal classes in your current application?

1. Post a question on the Oracle JDeveloper forum on OTN and ask for a public API alternative to what you think requires the use of internal framework classes
2. If you don't find a solution, use Expression language as explained above, starting from bullet #5

Using expression language as a substitution for internal Java API calls is considered a work around, though one that lasts for long. However, given expressions are resolved by the expression resolver before internal framework classes are accessed, it is your abstraction layer – or safety belt in this situation – that protects you from internal framework changes.

In summary:

- Keep the audit rule as it is

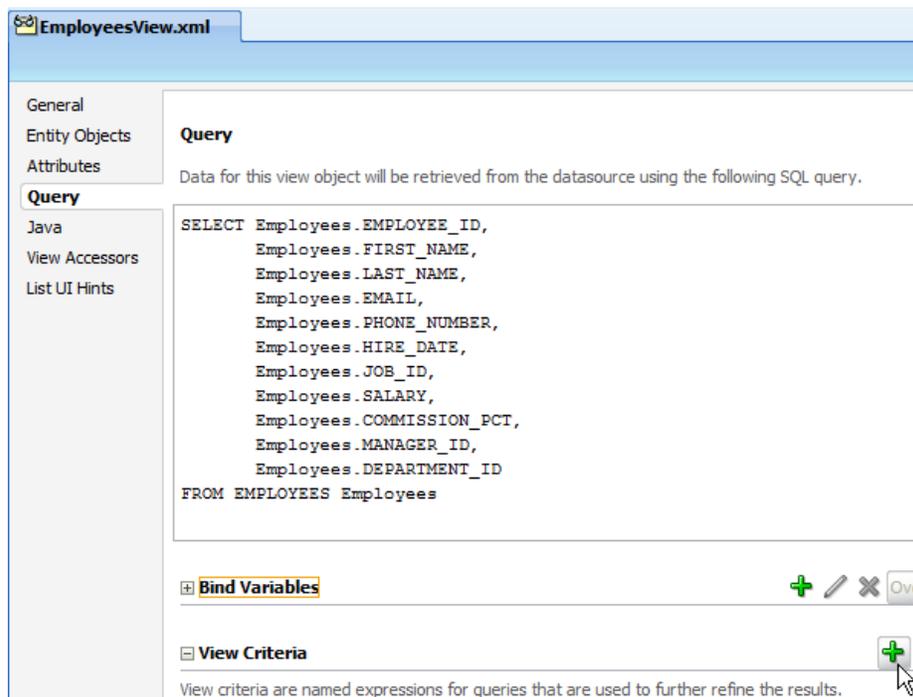
- Change it to **Warning** if you have to
- Avoid switching it off

## Building model driven dependent list with Oracle ADF BC

Creating dependent lists or list of values is a frequent developer requirement that is easy to implement using ADF Business Components. Instead of building the list of value dependency in the view layer, you define it on the View Object attribute level. Oracle JDeveloper automatically creates the dependent list components when the View Object is added as a form or table to the ADF Faces page.

The following example steps you through the creation of model driven dependent list boxes. The View Object in this sample represents a vacation request form with an attribute representing the *DepartmentId* and a dependent dependent *EmployeeId* attribute.

To create a dependent list component or list of value, you first need to edit the *EmployeesView* object to create a View Criteria that then is used to create the dependency between the selected *DepartmentId* in the vacation request form and the *EmployeeId*.

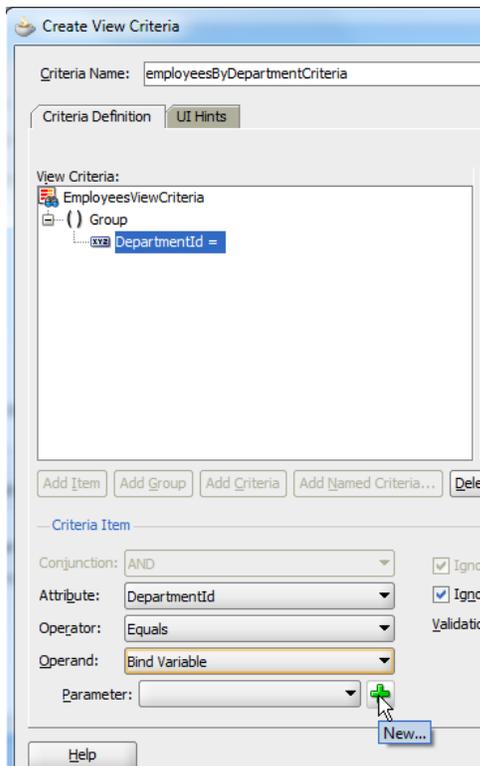


The screenshot shows the 'EmployeesView.xml' object editor. The 'Query' section is selected, displaying the following SQL query:

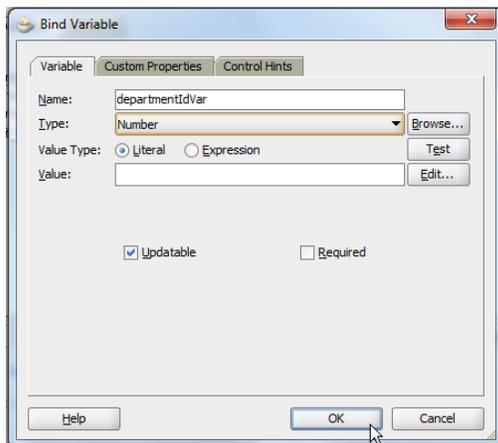
```
SELECT Employees.EMPLOYEE_ID,  
       Employees.FIRST_NAME,  
       Employees.LAST_NAME,  
       Employees.EMAIL,  
       Employees.PHONE_NUMBER,  
       Employees.HIRE_DATE,  
       Employees.JOB_ID,  
       Employees.SALARY,  
       Employees.COMMISSION_PCT,  
       Employees.MANAGER_ID,  
       Employees.DEPARTMENT_ID  
FROM EMPLOYEES Employees
```

Below the query, there are two sections: 'Bind Variables' and 'View Criteria'. A mouse cursor is pointing at a green plus icon next to the 'View Criteria' section, indicating the step to add a new view criteria.

Open the *EmployeeView* object editor and click the green plus icon next to the *View Criteria* section in the *Query* category. In the opened dialog, create a View Criteria that queries the *EmployeesView* object filtered by a *DepartmentId* value that is held in a bind variable.

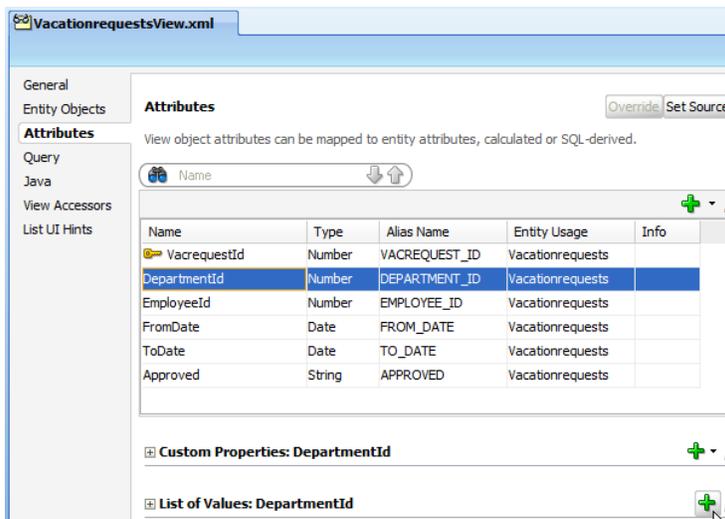


Click the **Add Item** button and choose the *DepartmentId* attribute. Choose the *Equals* operator and select **Bind Variable** as the *Operand*. Press the green plus icon to create the bind variable.

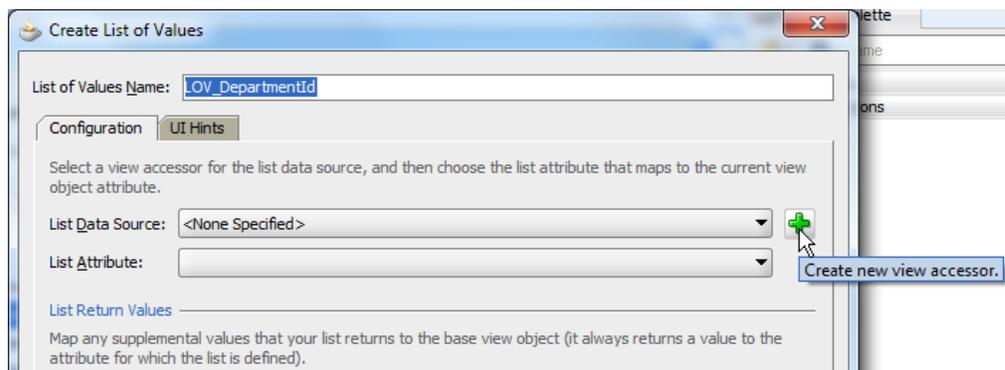


Define a Name, for example *departmentIdVar*, for the bind variable and set its Type to *Number*, which is the `oracle.jbo.domain.Number` type. Make sure the bind variable is updateable and OK the dialog. Ok the View Criteria too and open the *VacationRequestsView* object.

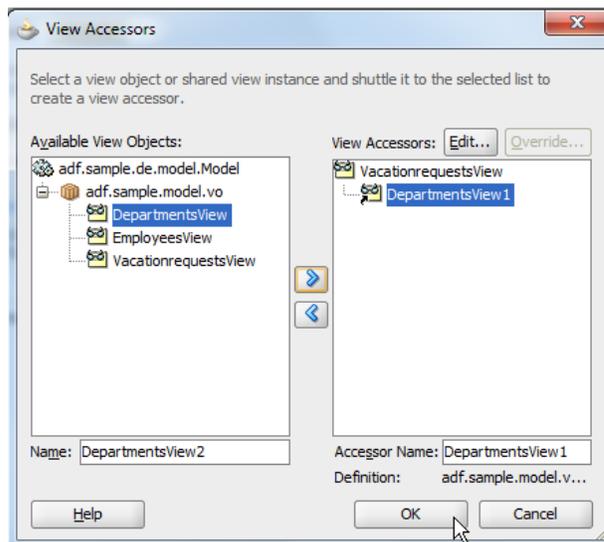
Select the *DepartmentId* attribute in the *Attribute* category of the View Object editor and press the green plus icon next to the *List of Values: DepartmentId* section.



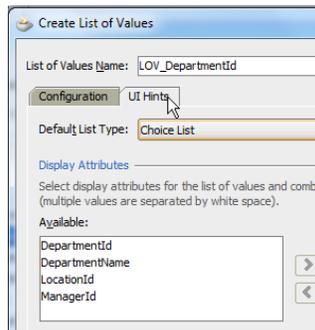
Click the green plus icon next to the *List Data Source* entry in the opened dialog to select the View Object source to provide the list data (*DepartmentsView*).



In return, Oracle JDeveloper creates a new accessor for the View Object. In *List Attribute*, select the list attribute matching the *DepartmentId* attribute in the vacation request form.

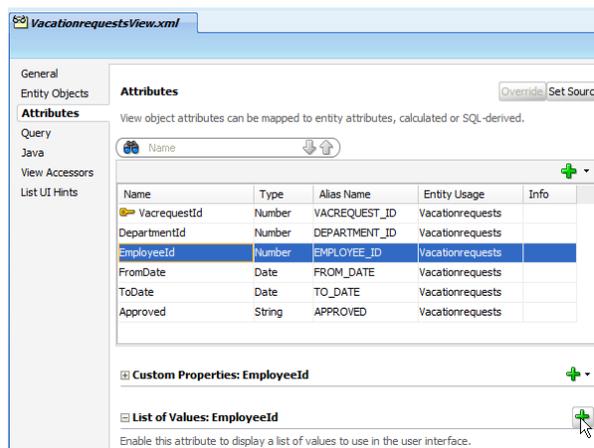


OK the dialog. In the *Create List of Values* dialog, select the *UI Hints* tab and choose **Choice List** as the component to build the list for this attribute at runtime.

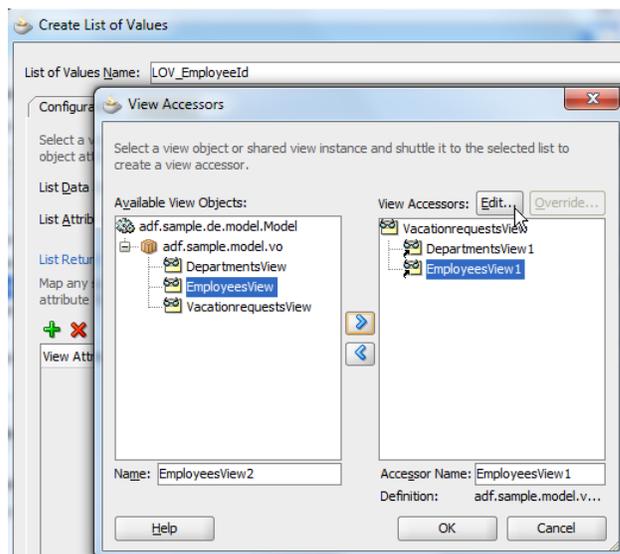


In the *Available* list, select the *DepartmentName* and move it to the list of selected display items. OK the dialog.

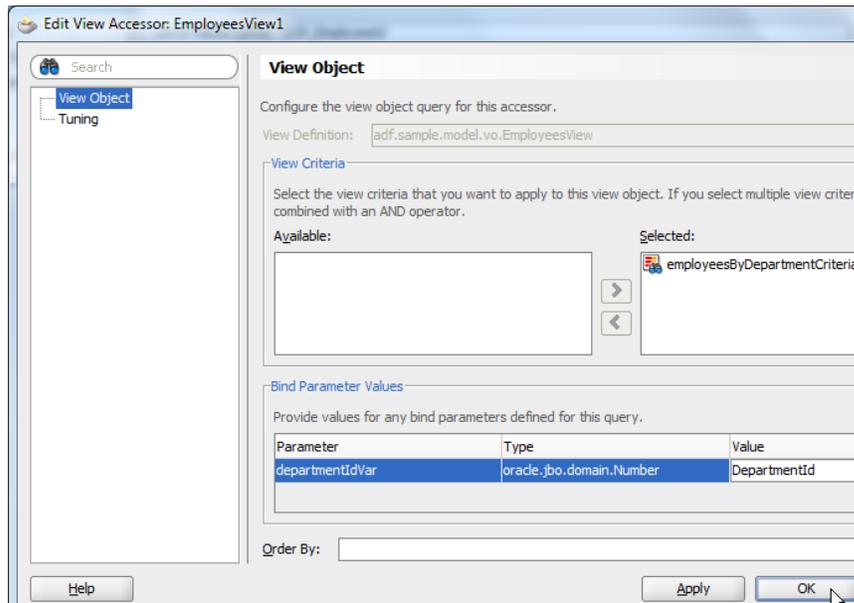
Repeat the list of values creation steps for the *EmployeesView* object.



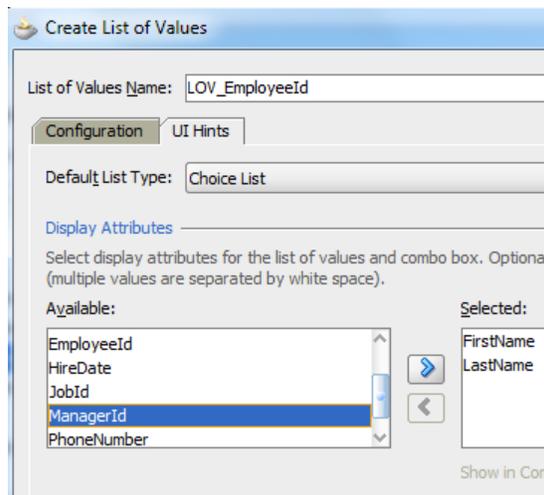
This time however, choose *EmployeesView* as the list object.



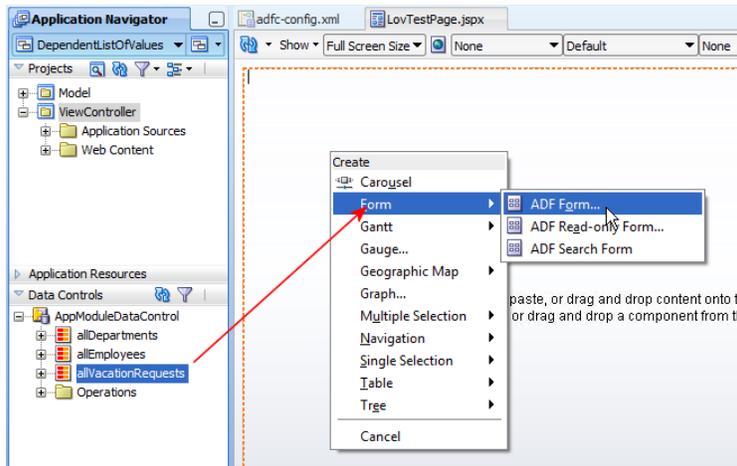
In the *View Accessors* dialog, click the **Edit** button to assign the View criteria created earlier.



Select the View Criteria and set its bind variable value to *DepartmentId*, the attribute in the vacation request view object that holds the selected parent value. Ok the dialog two times to return to the list of values creation dialog. Set *EmployeeId* as the matching list attribute and select the UI hints table and choose *FirstName* and *LastName* as the display values.

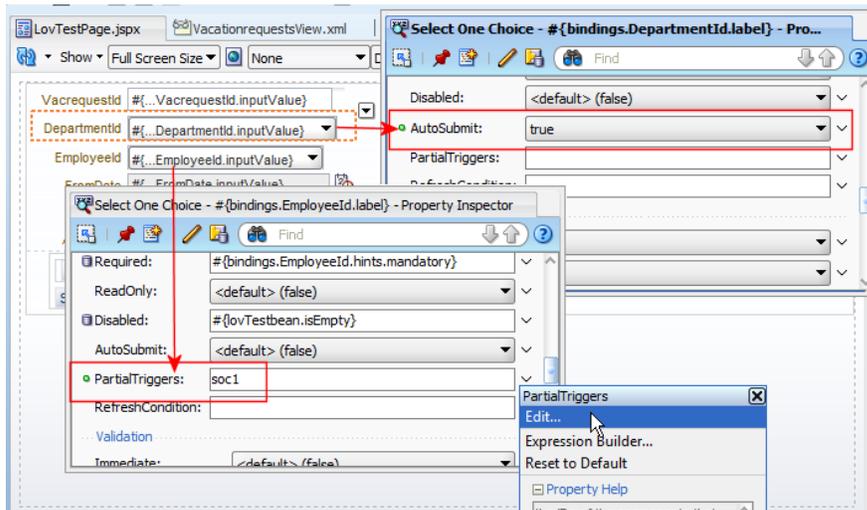


Ok the dialog and test the *VacationrequestsView* object in the ADF Business Components tester. For this, select the Application Module and choose the run option from the context menu. If the dependent lists work in the tester, create a new JSF page in the *ViewController* project and drag the *VacationrequestsView* collection from the DataControls panel and drop it as an ADF form onto the page.

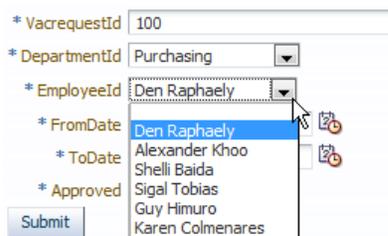


The form contains two instances of the `af:selectOneChoice` component, one for the `DepartmentId` attribute and one for the `EmployeeId` attribute. To make the two fields dependent, the `EmployeeId` list needs to be refreshed whenever the parent select list has the selected value changed.

For this, on the parent list, set the `autosubmit` property to "true" and have the `PartialTriggers` property of the dependent list box pointing to the parent list component Id.



This is all that it takes and you can now run the form and see the dependent list boxes in action.



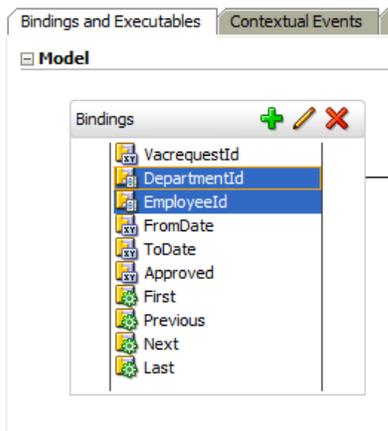
**Note:** The same dependency also works if the `af:inputListOfValues` component was chosen for providing the `DepartmentId` and `EmployeeId` attribute values.

## How to display a dependent list box disabled if no child data exist

A requirement on OTN was to disable the dependent list box of a model driven list of value configuration whenever the list is empty.

To disable the dependent list, the `af:selectOneChoice` component needs to be refreshed with every value change of the parent list, which however already is the case as the list boxes are already dependent.

When you create model driven list of values as choice lists in an ADF Faces page, two ADF list bindings are implicitly created in the `PageDef` file of the page that hosts the input form.



At runtime, a list binding is an instance of `FacesCtrlListBinding`, which exposes `getItems()` as a method to access a list of available child data (`java.util.List`). Using Expression Language, the list is accessible with

```
#{bindings.list_attribute_name.items}
```

To dynamically set the `disabled` property on the dependent `af:selectOneChoice` component, however, you need a managed bean that exposes the following two methods

```
//empty - but required - setter method
public void setIsEmpty(boolean isEmpty) {}

//the method that returns true/false when the list is empty or
//has values
public boolean isIsEmpty() {
```

```

FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
ExpressionFactory exprFactory =
    fctx.getApplication().getExpressionFactory();
ValueExpression vexpr =
    exprFactory.createValueExpression(elctx,
        "#{bindings.EmployeeId.items}",
        Object.class);
List employeesList = (List) vexpr.getValue(elctx);
return employeesList.isEmpty()? true : false;
}

```

If referenced from the dependent choice list, as shown below, the list is disabled whenever it contains no list data

**<!-- master list -->**

```

<af:selectOneChoice value="#{bindings.DepartmentId.inputValue}"
    label="#{bindings.DepartmentId.label}"
    required="#{bindings.DepartmentId.hints.mandatory}"
    shortDesc="#{bindings.DepartmentId.hints.tooltip}"
    id="soc1" autoSubmit="true">
    <f:selectItems value="#{bindings.DepartmentId.items}" id="si1"/>
</af:selectOneChoice>

```

**<!-- dependent list -->**

```

<af:selectOneChoice value="#{bindings.EmployeeId.inputValue}"
    label="#{bindings.EmployeeId.label}"
    required="#{bindings.EmployeeId.hints.mandatory}"
    shortDesc="#{bindings.EmployeeId.hints.tooltip}"
    id="soc2" disabled="#{lovTestbean.isEmpty}"
    partialTriggers="soc1">
    <f:selectItems value="#{bindings.EmployeeId.items}" id="si2"/>
</af:selectOneChoice>

```

\* VacrequestId 100

\* DepartmentId NOC

EmployeeId

\* FromDate 12/14/2010

\* ToDate 12/17/2010

\* Approved N

First Previous Next Last

Submit

## Testing bounded task flow using page fragments

Building reusable bounded task flows that are supposed to render in an ADF region require the use of page fragments to render the views. Page fragments are incomplete JSF pages and therefore bounded task flows that use them cannot be run from Oracle JDeveloper for testing.

The way to test bounded task flows that use page fragments for the views is to create a JSPX document and add the bounded task flow as a region to it. After this you can run and test the page.

The problem with this approach, however, is that the stand alone page, the JSX page for testing, is not supposed to be deployed with the application, which means you need to clean the project from it and its associated artifacts and changes (PageDef file created, entry in the DataBindings.cpx file).

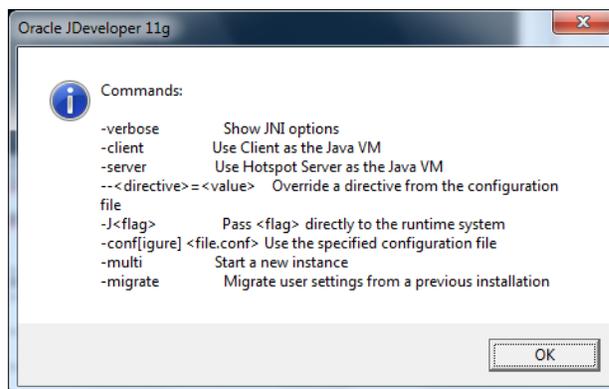
So a better testing option seems to be to deploy the ADF bounded task flow in an ADF library and have a separate project (in a separate workspace) to import the ADF library and adding its contained task flow to a test page for runtime testing.

This approach, though it appears a bit inconvenient has benefits:

1. The bounded task flow is tested in an environment that simulates how it would be later used
2. Artifacts created while testing don't need to be remembered and cleaned
3. You don't need to think about which libraries to remove when deploying the ADF library

## Oracle JDeveloper command line arguments

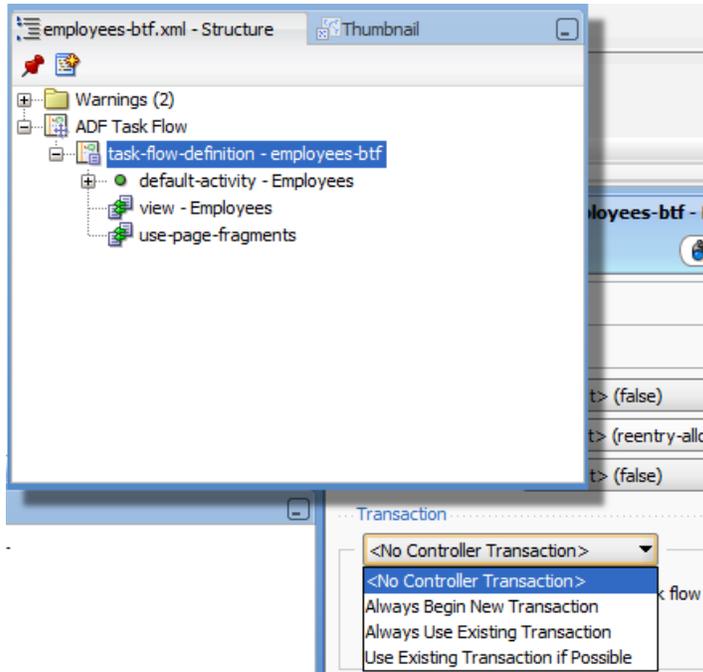
Oracle JDeveloper accepts command line arguments. To view the available list of command line arguments, start JDeveloper with the *-help* flag (<jdev\_home>\jdeveloper\jdeveloper -help). The following dialog, listing all supported command line arguments, is opened:



## Task flow "new transaction" vs. "new db connection"

Bounded task flow can represent a transaction and be used to declaratively manage transaction when using ADF Business Components as the business service. A transaction is a grouping of data model changes to be committed or rolled back at a certain point. A transaction is opened in ADF by the framework calling `beginTransaction` on the `ADF BindingContext`.

To configure the bounded task flow transaction behavior, you select the bounded task flow in the Oracle JDeveloper *Application Navigator* and open the Structure Window (ctrl+shift+S). In the Structure Window, expand the *ADF Task Flow* node and select the contained task flow.

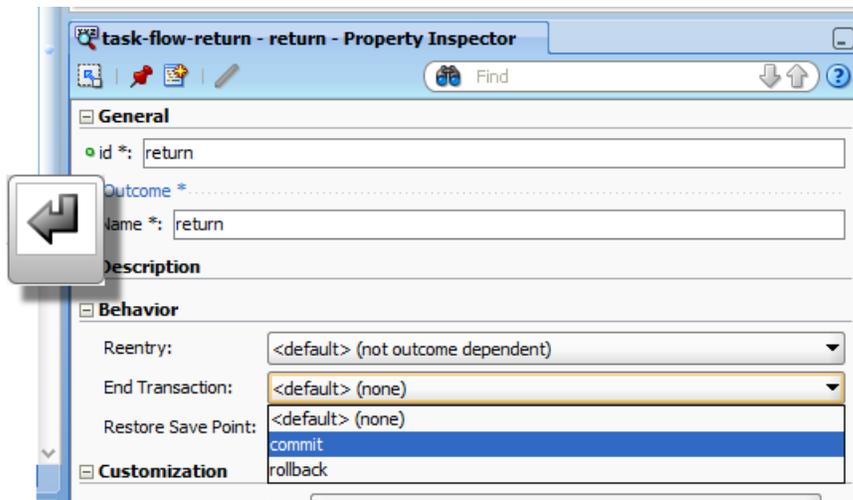


Open the *Property Inspector* (ctrl+shift+I) and navigate to the *Behavior* section and set the *Transaction* property to one of the following:

- No Controller Transaction (default) : The bounded task flow does not start a transaction when entered
- Always Begin New Transaction : When the task flow is entered, a new transaction is always started
- Always Use Existing Transaction : The bounded task flow expects a transaction to exist that it can reuse
- Use Existing Transaction if Possible : If a transaction exists, it is used, if not, a new one is created.

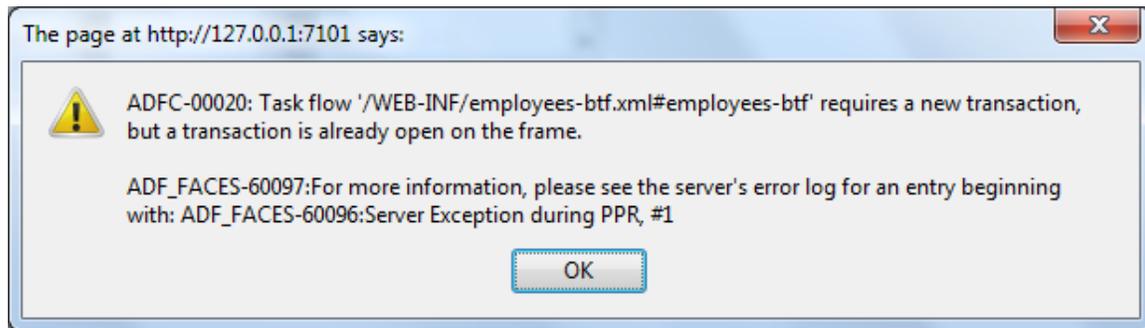
Whenever a bounded task flow is configured to start a new transaction it needs to either commit or rollback the transaction upon exiting the task flow. The configuration of how to exit a bounded task flow is configured on the return activity.

**Note:** The ADF Controller does not handle transactions. All it does is to pass the configured transaction behavior as a hint to the Data Control. It is up to the Data Control to implement these hints.



When a bounded task flow creates a new transaction, does it also mean it creates a new database connection? No.

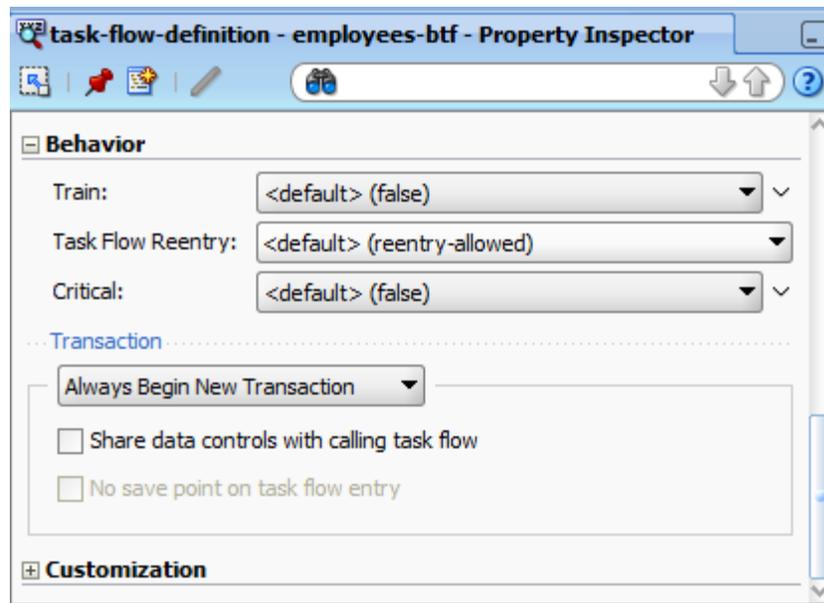
Bounded task flow that share the data control with the calling task flow share the database connection too, which also means they share the transaction if one exists. Using ADF Business Components a single transaction exists per database connection. Trying to open a second transaction in a bounded task flow will cause a task flow exception, ADFC-00020.



*oracle.adf.controller.activity.ActivityLogicException: ADFC-00020:*

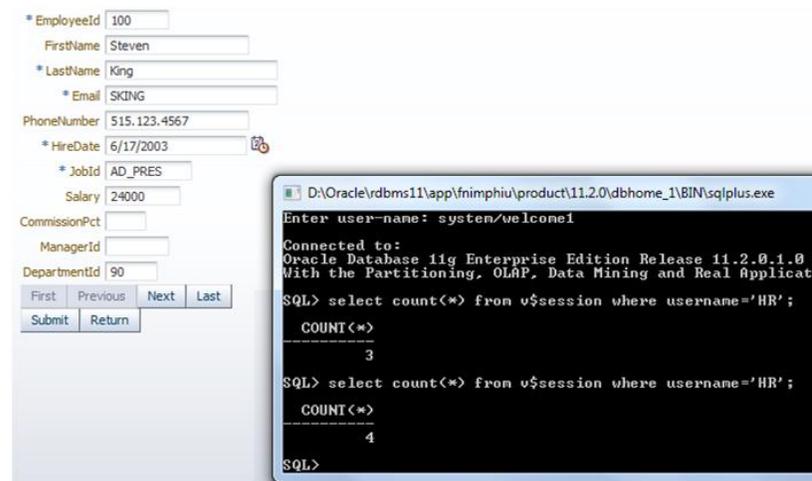
*Task flow '/WEB-INF/employees-btf.xml#employees-btf' requires a new transaction, but a transaction is already open on the frame.*

To open a second transaction, a second Data Control frame is needed, which you configure in the Property Inspector for the bounded task flow, **unchecking** the *Share data control with calling task flow* checkbox.



However, not sharing the data control means, internally, a new data control frame is opened, which is comparable to starting a new root Application Module in ADF Business Components, creating a new database connection. You can test this by opening SQL\*Plus and counting the connections for the application database connect.

```
select count (*) from v$sqlsession where username='<db user connect name>'
```

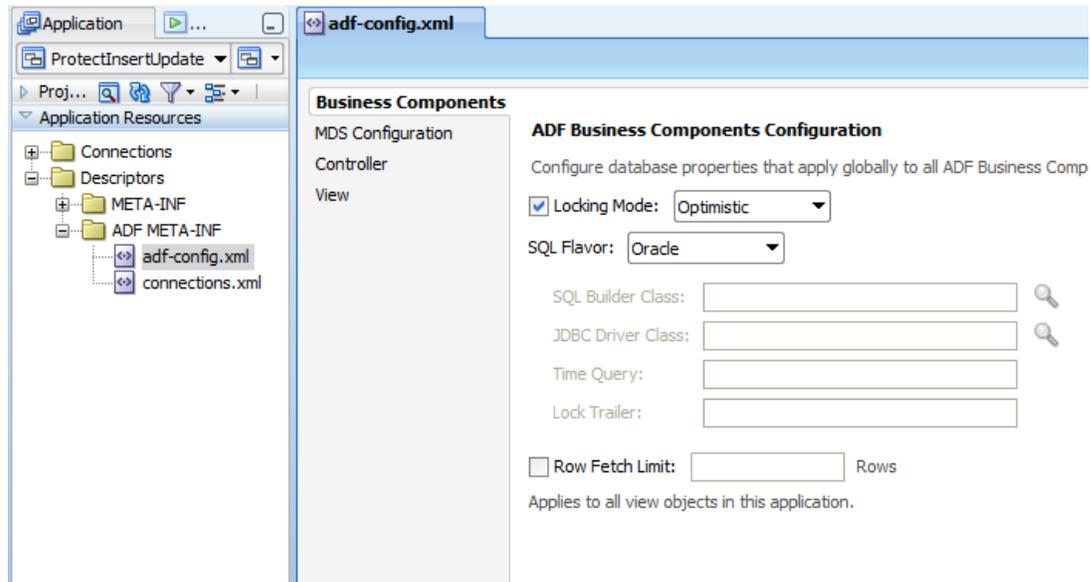


You issue the SQL command before and after navigating to a bounded task flow that is isolated from the calling task flow.

So the answer to the initial question is that a new database connection is created when the data control is **not** shared between a calling and the called task flow. Configuring the transaction on a bounded task flow to open a new transaction does not create a new database connection.

## Configuring the ADF BC locking behavior in JDeveloper 11.1.1.4

In Oracle JDeveloper releases prior to 11.1.1.4, the ADF Business Components locking behavior was defaulted to *pessimistic* though *optimistic* is what should be used for web applications. Also in JDeveloper 11.1.1.4, the configuration of this behavior has been simplified in that the behavior now is configured in the `adf-config.xml` file. To change the locking behavior, you expand the *Application Resources* accordion panel in the JDeveloper *Application Navigator* and expand the *Descriptors* | *ADF META-INF* node. Double click onto the `adf-config.xml` file entry to open the visual editor shown below.

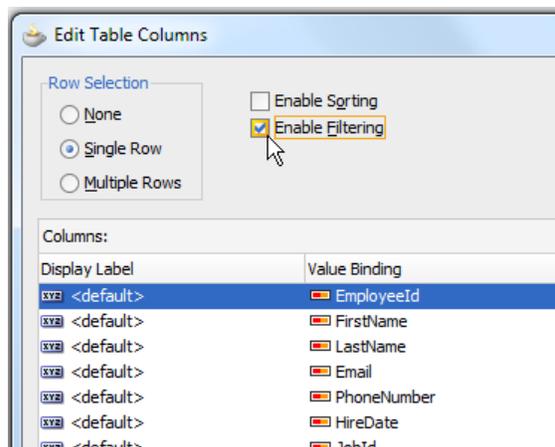


## How-to filter table filter input to only allow numeric input

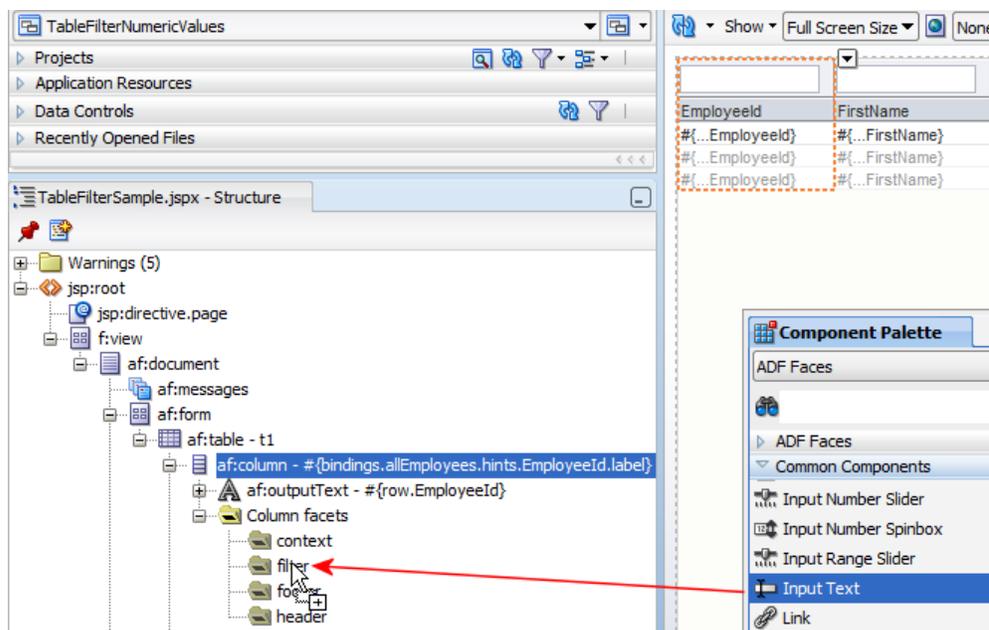
In a previous ADF Code Corner post, I explained how to change the table filter behavior by intercepting the query condition in a query filter. See sample #30 at <http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html>

In this OTN Harvest post I explain how to prevent users from providing invalid character entries as table filter criteria to avoid problems upon re-querying the table. In the example shown next, only numeric values are allowed for a table column filter.

To create a table that allows data filtering, drag a View Object – or a data collection of a Web Service or JPA business service – from the DataControls panel and drop it as a table. Choose the *Enable Filtering* option in the *Edit Table Columns* dialog so the table renders with the column filter boxes displayed.

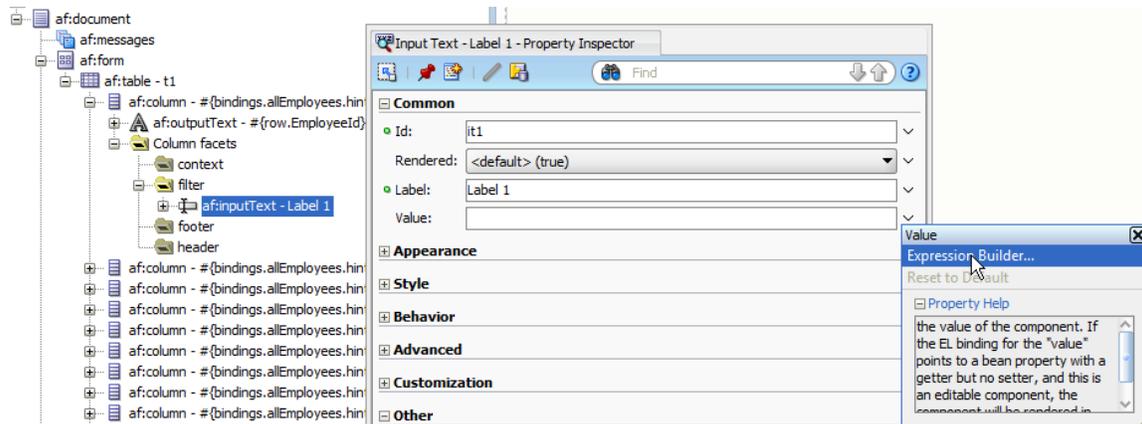


The table filter fields are created using implicit `af:inputText` components that need to be customized for you to apply a custom filter input component, or to change the input behavior. To change the input filter, so only a defined set of input keys is allowed, you need to change the default filter field with your own `af:inputText` field to which you apply an `af:clientListener` tag that filters user keyboard entries.

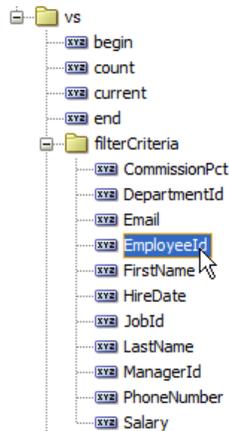


For this, in the Oracle JDeveloper visual editor, select the column which filter you want to change and expand the column node in the Oracle JDeveloper Structure Window. Part of the column definition is the *Column facet* node. Expand the facets so you see the *filter* facet entry. The filter facet is grayed out as there is no custom facet defined. In a next step, open the Component Palette (`ctrl+shift+P`) and drag an *Input Text* component onto the facet. This demarks the first part in the filter customization.

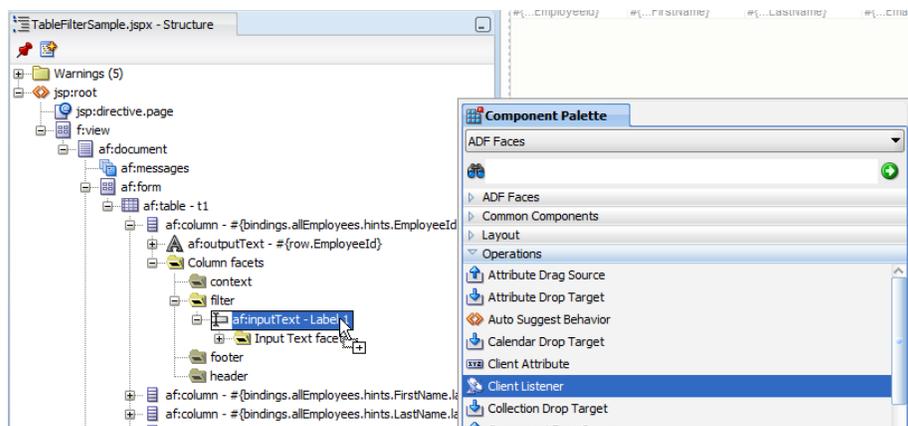
To make the custom filter component work, you need to map the `af:inputText` component *value* property to the ADF filter criteria that is exposed in the Expression Builder.



Open the *Expression Builder* for the filter input component *value* property by clicking the arrow icon to its right. In the *Expression Builder* expand the **JSP Objects | vs | filterCriteria** node to select the attribute name represented by the table column. The **vs** entry is the name of a variable that is defined on the table and that grants you access to the table attributes.



Now that the filter works as before – though using a custom filter input component – you can add the *af:clientListener* tag to your custom filter input component – *af:inputText* – call out to JavaScript when users type in the column filter field

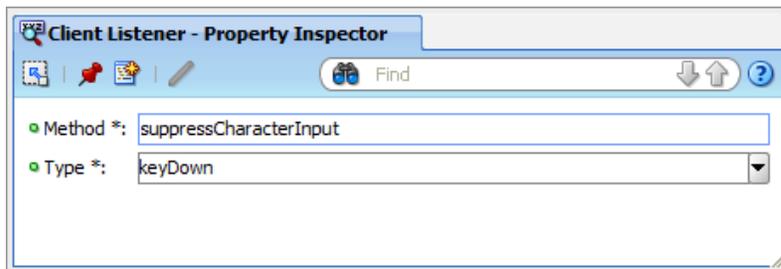


Point the client filter *method* property to a JavaScript function that you reference or add through using the *af:resource* tag and set the *type* property value to *keyDown*.

```
<af:document id="d1">
  <af:resource type="javascript" source="/js/filterHandler.js"/>
  ...
```

The filter definition looks as shown below

```
<af:inputText label="Label 1" id="it1"
  value="#{vs.filterCriteria.Employe
  <af:clientListener method="suppressCharacterInput"
    type="keyDown"/>
</af:inputText>
```



The JavaScript code that you can use to either filter character inputs or numeric inputs is shown below. Just store this code in an external JavaScript (.js) file and reference it from the `af:resource` tag.

```
//Allow numbers, cursor control keys and delete keys
function suppressCharacterInput(evt) {
  var _keyCode = evt.getKeyCode();
  var _filterField = evt.getCurrentTarget();
  var _oldValue = _filterField.getValue();

  if (!((_keyCode < 57) || (_keyCode > 96 && _keyCode < 105))) {
    _filterField.setValue(_oldValue);
    evt.cancel();
  }
}

//Allow characters, cursor control keys and delete keys
function suppressNumericInput(evt) {
  var _keyCode = evt.getKeyCode();
  var _filterField = evt.getCurrentTarget();
  var _oldValue = _filterField.getValue();

  //check for numbers
  if ((_keyCode < 57 && _keyCode > 47) ||
    (_keyCode > 96 && _keyCode < 105)){
    _filterField.setValue(_oldValue);
    evt.cancel();
  }
}
```

**But what if browsers don't allow JavaScript ?** Don't worry about this. If browsers would not support JavaScript then ADF Faces as a whole would not work and you had a different problem.

## Best practices about creating and using backing beans

Backing beans are special uses of managed beans and have a 1:1 relation to a page or page fragment. By default, Oracle JDeveloper doesn't create backing beans for pages you create. Automatic backing bean creation is a setting you can configure in the **Design | Page Properties | Component Binding** menu option that shows when you opened the JSF visual editor in Oracle JDeveloper. Best practices however is to not create backing beans for the pages you create, which also is the default behavior.

Creating backing beans provides easy access to the component instance for programmatic manipulation of the component state and data, but also represents unnecessary overhead as there is no option to tell the IDE when not to create component bindings or to remove component bindings that are longer needed. Especially complex pages thus quickly end up with lots of Java entries created in the managed bean, which is hard to maintain and also hard to keep track of.

Best practices for using backing bean is not to use the auto-generate feature in Oracle JDeveloper but to create component binding references on an as needed basis. To create a component binding to a managed bean, which then turns into a backing bean for this page, select the component binding property in the Property Inspector and open the context menu by pressing the arrow icon. Choose **Edit** from the context menu to create a component binding reference.

## Extending the ADF Controller exception handler

The Oracle ADF controller provides a declarative option for developers to define a view activity, method activity or router activity to handle exceptions in bounded or unbounded task flows. Exception handling however is for exceptions only and not handling all types of Throwable. Furthermore, exceptions that occur during the JSF RENDER RESPONSE phase are not looked at either as it is considered too late in the cycle.

For developers to try themselves to handle unhandled exceptions in ADF Controller, it is possible to extend the default exception handling, while still leveraging the declarative configuration. To add your own exception handler:

- Create a Java class that extends ExceptionHandler
- Create a textfile with the name "oracle.adf.view.rich.context.Exceptionhandler" (without the quotes) and store it in .adf\META-INF\services (you need to create the "services" folder)
- In the file, add the absolute name of your custom exception handler class (package name and class name without the ".class" extension)

For any exception you don't handle in your custom exception handler, just re-throw it for the default handler to give it a try

```
import oracle.adf.view.rich.context.ExceptionHandler;

public class MyCustomExceptionHandler extends ExceptionHandler {
    public MyCustomExceptionHandler() {
```

```

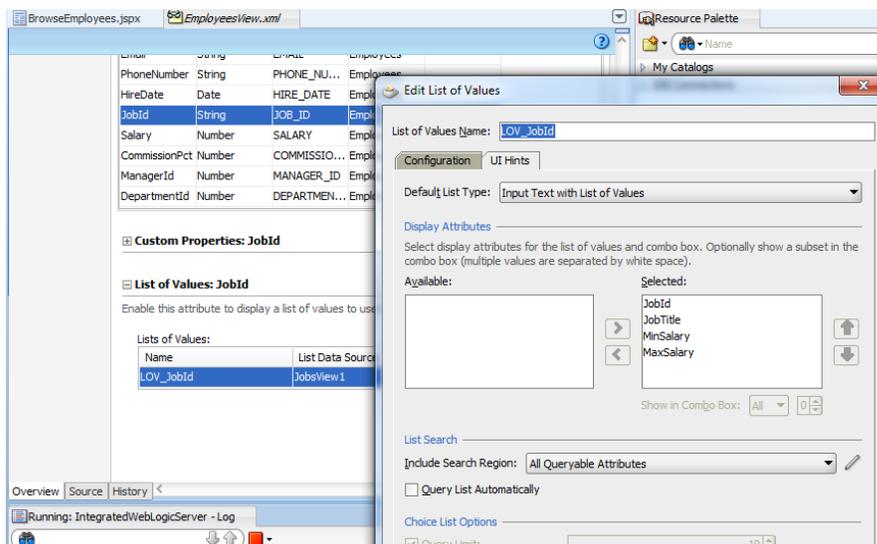
        super();
    }
    public void handleException(FacesContext facesContext,
                               Throwable throwable, PhaseId phaseId)
        throws Throwable
    {
        String error_message;
        error_message = throwable.getMessage();
        //check error message and handle it if you can
        if( ... ){
            //handle exception
            ...
        }
        else{
            //delegate to the default ADFc exception handler
            throw throwable;}
    }
}

```

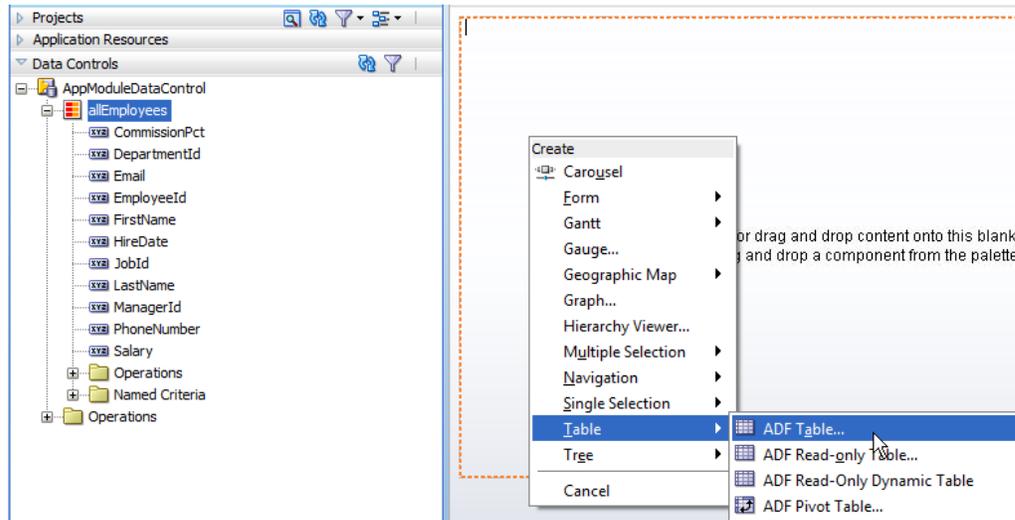
Note however, that it is recommended to first try and handle exceptions with the ADF Controller default exception handling mechanism. In the past, I've seen attempts on OTN to handle regular application use cases with custom exception handlers for where there was no need to override the exception handler. So don't go for this solution to quickly and always think of alternative solutions. Sometimes a try-catch-final block does it better than sophisticated web exception handling.

## How to create a model-driven multi column auto-suggest list

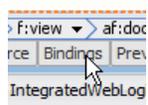
ADF Faces provides an auto suggest behavior tag – `af:autoSuggestBehavior` – that you use to suggest selectable values based on user input into a text field or combo box. Using the ADF list of values binding, you can build the suggest behavior declarative and model driven. For this, you select the View Object attribute for which want to provide list of values support (that later renders as suggest items).



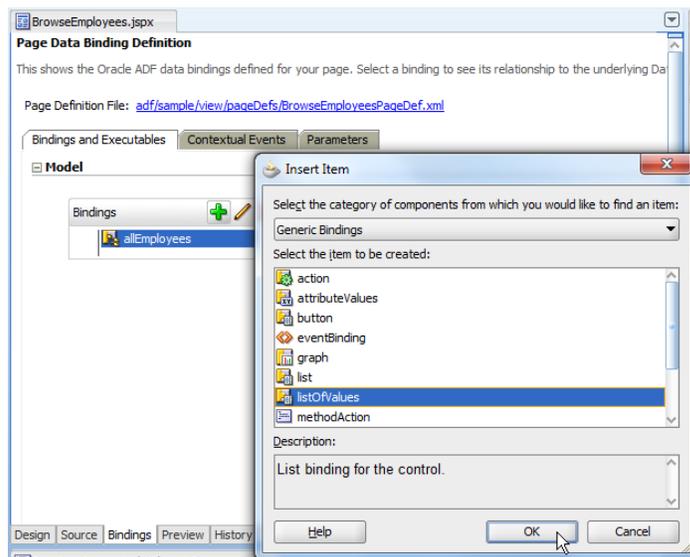
Press the green plus icon in the *List of Values* header section to create a new list of values. Choose a list data source and map the list attribute to the base attribute you want to copy the selected value to. This you do in the *Configuration* tab. In the *UI Hint* tab, you specify the UI control that is used to render the attribute when it is dragged from the Data Controls panel to the JSF view (for example: *Input Text With List of Values*). Ok the dialog so the list of value definition is created.



When dragging the View Object that has the list of value defined on its attribute(s) from the Data Controls panel and dropping it as a form or table, the attribute is rendered with the UI component specified in the UI Hints.

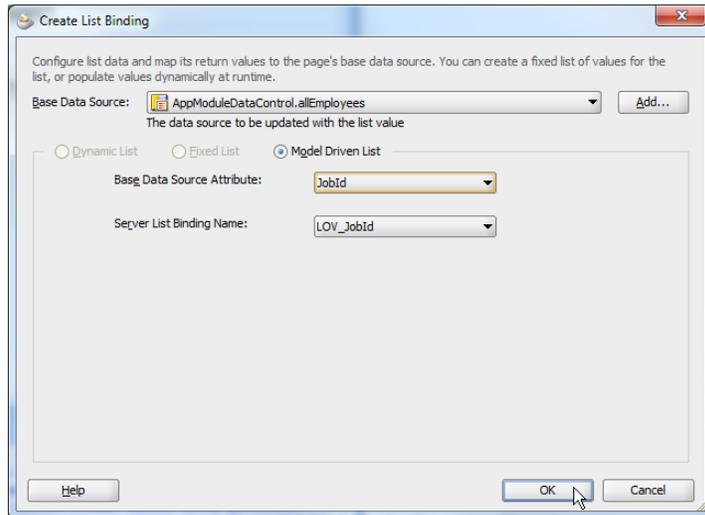


Press the *Bindings* tab at the bottom of the ADF Faces view to add a list of values binding, as shown in the image below.

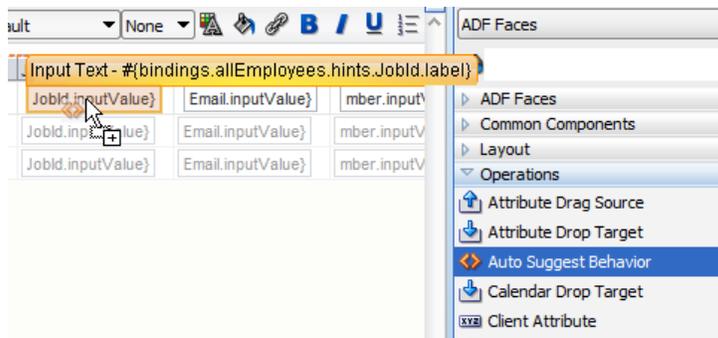


Press the green plus icon in the *Bindings and Executables* tab of the binding editor and choose the *listOfValues* entry to create a new list of value binding.

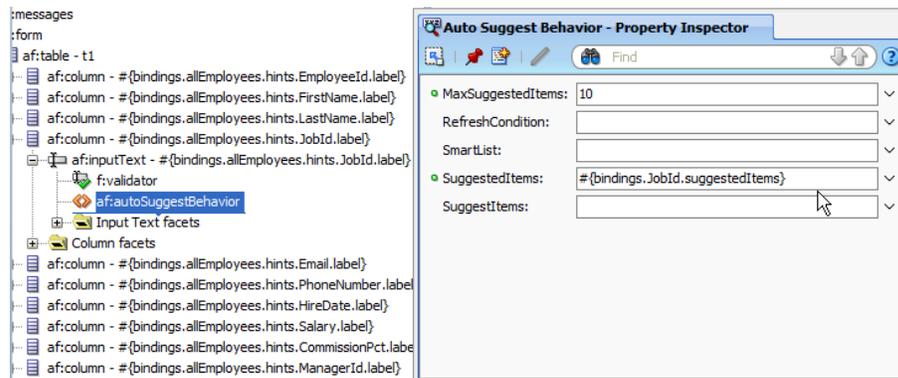
Choose the View Object with the list of value definition in one of its attributes as the *Base Data Source* and select the attribute itself as the *Base Data Source Attribute*. The *Server List Binding Name* field is populated automatically and you can OK the dialog.



Back in the visual page editor, expand the *Operations* accordion in the Component Palette and drag the *Auto Suggest Behavior* tag onto the column input component that renders the model driven-list-of-value, as shown in the image below.



Edit the `af:autoSuggestBehavior` component to reference the list of value binding you created before from the *SuggestedItems* property.



For example, if the attribute that has the model driven list of values defined is "JobId", as used in the example, then the expression is

```
#{bindings.JobId.suggestedItems}
```

**Note:** When you configure a model driven list of values for a View Object attribute, then, when designing ADF Faces views, the attribute is represented by the UI component you specified as the *Default Type* in the *UI Hints* tab of the *Edit List of Values* dialog. If you only want to use the model driven LOV for the suggest item behavior, you can change the component in the page source editor: For example instead of an *Input Item with List of Values*, you just use a plain *Input Text* component, changing the page source to `af:inputText`.

At runtime, a list of suggest choices is automatically shown based on the user typed input. The benefit of using model driven list of values to populate the suggest list is that you display additional information as shown in the image below.

EmployeeId	FirstName	LastName	JobId	Email	PhoneNum
100	Steven	King	A	SKING	515.123.
101	Neena	Kochhar	AD_PRES President 20080 40000		23.
102	Lex	De Haan	AD_VP Administration Vice President 15000 30000		23.
103	Alexander	Hunold	AD_ASST Administration Assistant 3000 6000		23.
104	Bruce	Ernst	AC_MGR Accounting Manager 8200 16000		23.
105	David	Austin	AC_ACCOUNT Public Accountant 4200 9000		23.
			IT_PROG	DAUSTIN	590.423.

## How-to delete a tree node using the context menu

Hierarchical trees in Oracle ADF make use of View Accessors, which means that only the top level node needs to be exposed as a View Object instance on the ADF Business Components Data Model. This also means that only the top level node has a representation in the PageDef file as a tree binding and iterator binding reference. Detail nodes are accessed through tree rule definitions that use the accessor mentioned above (or nested collections in the case of POJO or EJB business services).

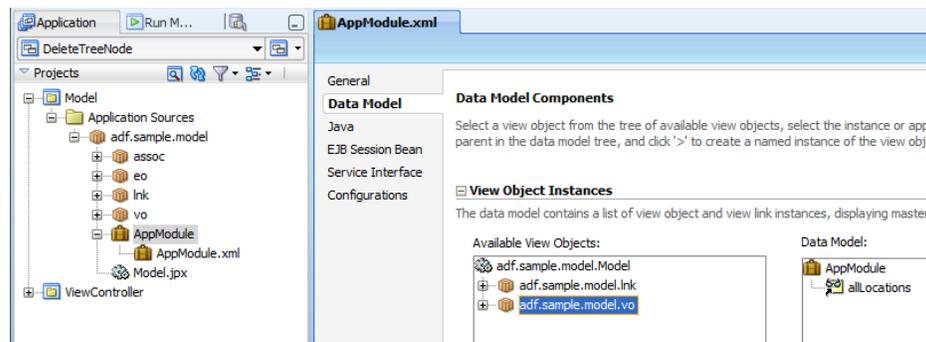
The tree component is configured for single node selection, which however can be declaratively changed for users to press the *ctrl* key and selecting multiple nodes.

In the following, I explain how to create a context menu on the tree for users to delete the selected tree nodes. For this, the context menu item will access a managed bean, which then determines the selected node(s), the internal ADF node bindings and the rows they represent.

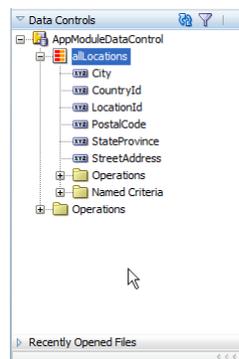


As mentioned, the ADF Business Components *Data Model* only needs to expose the top level node data sources, which in this example is an instance of the Locations View Object. For the tree to work, you need to have associations defined between entities, which usually is done for you by Oracle JDeveloper if the database tables have foreign keys defined

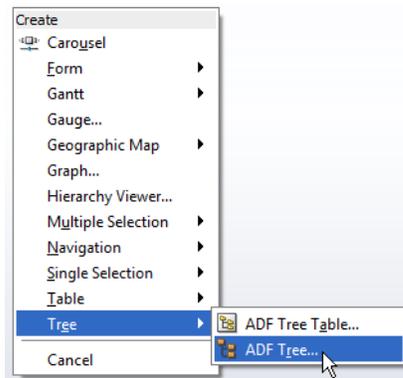
**Note:** As a general hint of best practices and to simplify your life: Make sure your database schema is well defined and designed before starting your development project. Don't treat the database as something organic that grows and changes with the requirements as you proceed in your project. Business service refactoring in response to database changes is possible, but should be treated as an exception, not the rule. Good database design is a necessity – even for application developers – and nothing evil.



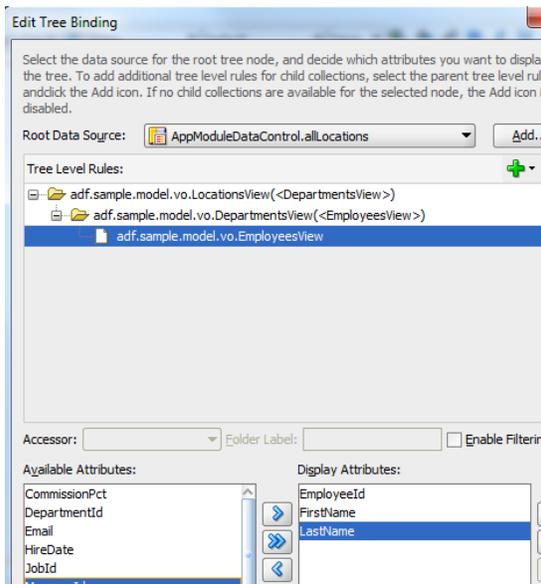
To create the tree component, expand the Data Controls panel and drag the View Object collection to the view.



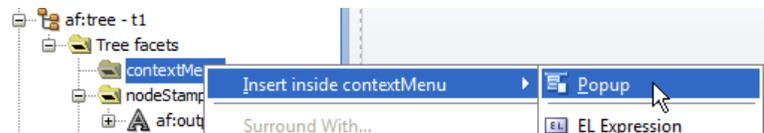
From the context menu, select the tree component entry and continue with defining the tree rules that make up the hierarchical structure.



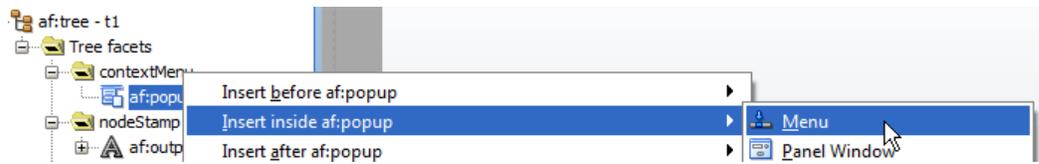
As you see, when pressing the *green plus icon* in the *Edit Tree Binding* dialog, the data structure, Locations - Departments – Employees in my sample, shows without you having created a View Object instance for each of the nodes in the ADF Business Components *Data Model*.



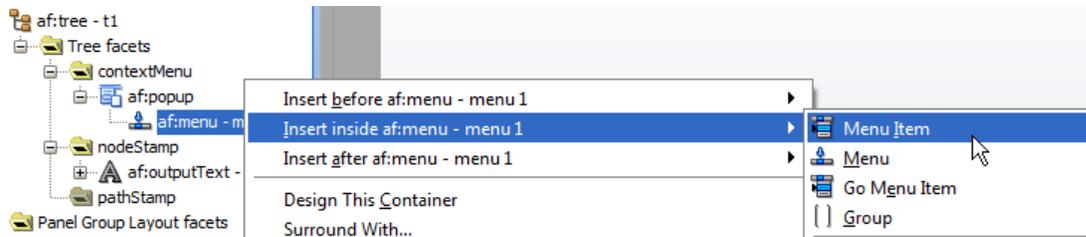
After you configured the tree structure in the *Edit Tree Binding* dialog, you press **OK** and the tree is created. Select the tree in the page editor and open the Structure Window (ctrl+shift+S). In the Structure window, expand the tree node to access the *contextMenu* facet. Use the right mouse button to insert a *Popup* into the facet.



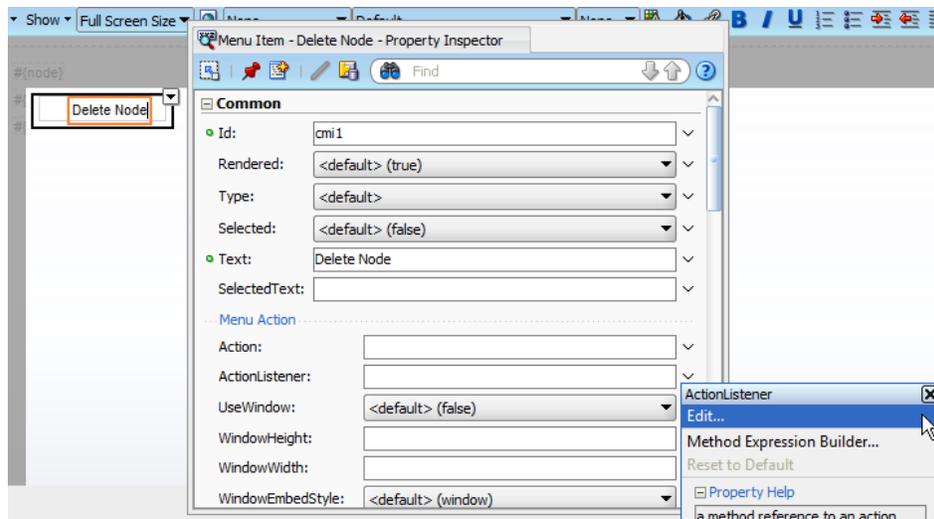
Repeat the same steps to insert a *Menu* and a *Menu Item* into the *Popup* you created.



The Menu item text should be changed to something meaningful like "Delete". Note that the custom menu item later is added to the context menu together with the default context menu options like *expand* and *expand all*.



To define the action that is executed when the menu item is clicked on, you select the *Action Listener* property in the Property Inspector and click the *arrow icon* followed by the *Edit* menu option. Create or select a managed bean and define a method name for the action handler.



Next, select the tree component and browse to its *binding* property in the Property Inspector. Again, use the **arrow icon** | **Edit** option to create a component binding in the same managed bean that has the action listener defined. The tree handle is used in the action listener code, which is shown below:

```
public void onTreeNodeDelete(ActionEvent actionEvent) {
    //access the tree from the JSF component reference created
    //using the af:tree "binding" property. The "binding" property
    //creates a pair of set/get methods to access the RichTree instance
    RichTree tree = this.getTreeHandler();
    //get the list of selected row keys
    RowKeySet rks = tree.getSelectedRowKeys();
    //access the iterator to loop over selected nodes
    Iterator rksIterator = rks.iterator();
}
```

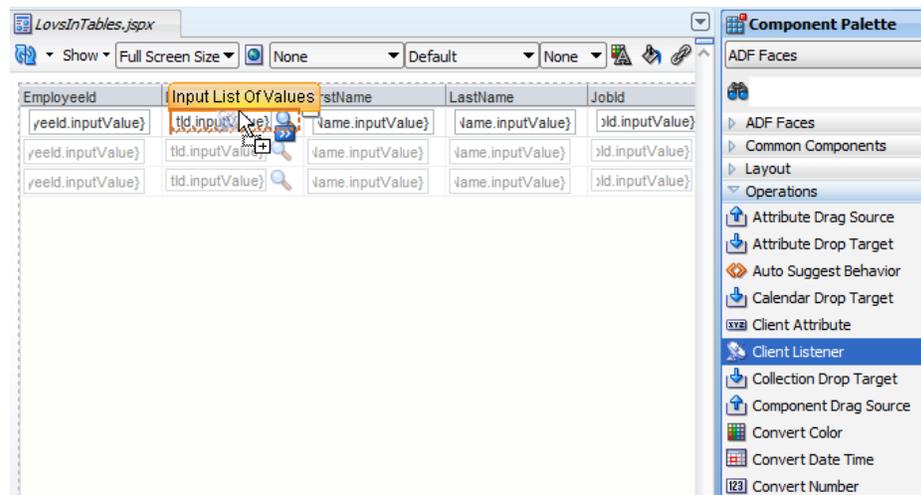
```
//The CollectionModel represents the tree model and is
//accessed from the tree "value" property
CollectionModel model = (CollectionModel) tree.getValue();
//The CollectionModel is a wrapper for the ADF tree binding
//class, which is JUCtrlHierBinding
JUCtrlHierBinding treeBinding =
    (JUCtrlHierBinding) model.getWrappedData();

//loop over the selected nodes and delete the rows they
//represent
while(rksIterator.hasNext()){
    List nodeKey = (List) rksIterator.next();
    //find the ADF node binding using the node key
    JUCtrlHierNodeBinding node =
        treeBinding.findNodeByKeyPath(nodeKey);
    //delete the row.
    Row rw = node.getRow();
    rw.remove();
}
//only refresh the tree if tree nodes have been selected
if(rks.size() > 0){
    AdfFacesContext adfFacesContext =
        AdfFacesContext.getCurrentInstance();
    adfFacesContext.addPartialTarget(tree);
}
}
```

**Note:** To enable multi node selection for a tree, select the tree and change the row selection setting from "single" to "multiple".

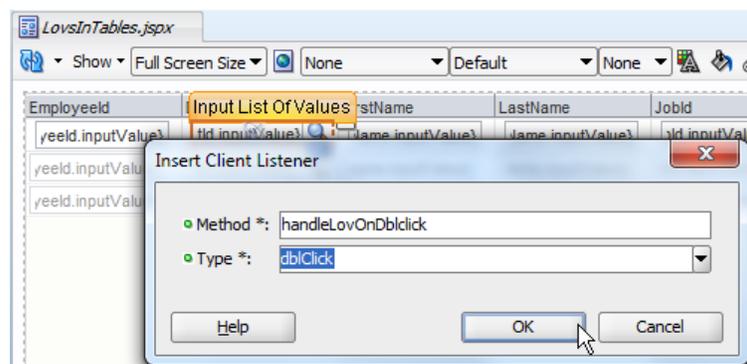
## How to open the LOV of af:inputListOfValues with a double click

To open the LOV popup of an `af:inputListOfValues` component in ADF Faces, you either click the magnifier icon to the right of the input field or tab onto the icon and press the *Enter key*. If you want to open the same dialog in response to a user double click into the LOV input field, JavaScript is a friend.



For this solution, I assume you created an editable table or input form that is based on a View Object that contains at least one attribute that has a model driven list of values defined. The *Default List Type* is should be set to *Input Text with List of Values* so that when the form or table gets created, the attribute is rendered by the `af:inputListOfValues` component.

To implement the use case, drag a *Client Listener* component from the **Operations** accordion in the Component Palette and drop it onto the `af:inputListOfValues` component in the page. In the opened *Insert Client Listener* dialog, define the **Method** as `handleLovOnDbclick` and choose `dblClick` in the select list for the **Type** attribute.



Add the following code snippet to the page source directly below the `af:document` tag.

```
<af:document id="d1">
  <af:resource type="javascript">
    function handleLovOnDbclick(evt) {
      var lovComp = evt.getSource();
      if (lovComp instanceof AdfRichInputListOfValues &&
        lovComp.getReadOnly() == false) {
        AdfLaunchPopupEvent.queue(lovComp, true);
      }
    }
  </af:resource>
</af:document>
```

```

    }
</af:resource>

```

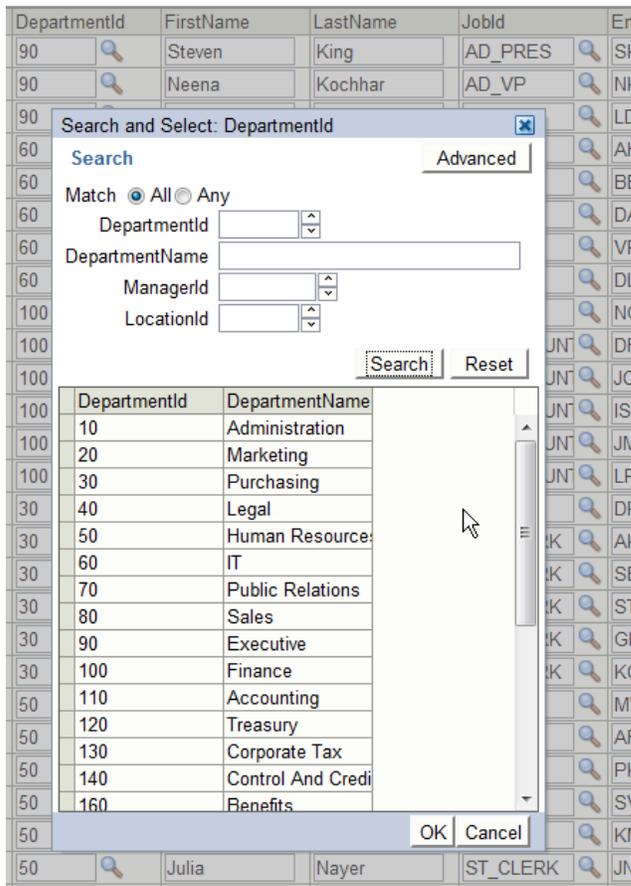
The JavaScript function is called whenever the user clicks into the LOV field. It gets the source component reference from the event object that is passed into the function and verifies the LOV component is not read only. It then queues the launch event for the LOV popup to open. The page source for the LOV component is shown below:

```

<af:inputListOfValues id="departmentIdId" ... >
  <f:validator binding="..."/>
  ...
  <af:clientListener method="handleLovOnDblick" type="dblClick"/>
</af:inputListOfValues>

```

At runtime, the popup opens in response to a mouse double click as shown in the image below:



## Configuring projects for Java EE security annotations

Java EE security annotations are used in Enterprise Java Beans and JPA to protect user access to methods exposed in entities and the session façade. Creating a new EJB project in Oracle JDeveloper, or using the *Java EE Web Application* template to build a web application using EJB and ADF Faces, does not add the classes of the `javax.annotation.security` package to the project class path. To solve this issue,

and to make security annotations like `@DenyAll` available in the code editor, you need to add the **WebLogic 10.3. Remote Client** library as follows:

- Open the Model project properties by double clicking onto the project node or using the context menu
- Select **Libraries and Classpath**
- Click the **Add Library** button
- Search for and add the **WebLogic 10.3. Remote Client** entry

## Implementing Query pagination using EJB and ADF

With pagination, data is queried on demand instead of all –at-once and ad-hoc. It's a desirable feature especially when working with large data sets to query, e.g. through scrolling in a table. ADF Code Corner (<http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html>) sample #37 explains how to enable query pagination for ADF models that are based on a POJO data model using the JavaBean Data Control. But how do you enable pagination for Enterprise JavaBean models that use the EJB Data Control? The good news is that you don't need to do anything if you started developing your EJB model and ADF applications with Oracle JDeveloper 11g release 11.1.1.3 (PS2) or later.

If you generated the EJB session façade with one of these Oracle JDeveloper, then the following method is automatically added:

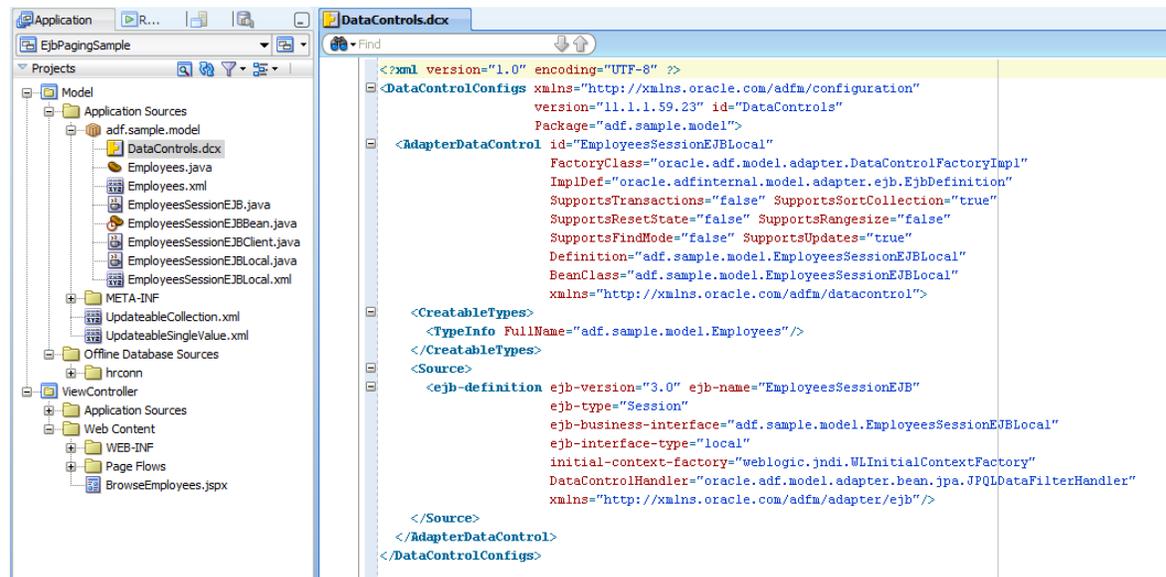
```
public Object queryByRange(String jpqlStmt, int firstResult,
                           int maxResults) {
    Query query = em.createQuery(jpqlStmt);
    if (firstResult > 0) {
        query = query.setFirstResult(firstResult);
    }
    if (maxResults > 0) {
        query = query.setMaxResults(maxResults);
    }
    return new ArrayList<Employees>(); query.getResultList();
}
```

**Note** This method is also defined in the EJB local and remote interface definition.

When generating the Oracle ADF Data Control configuration for the EJB session façade by right mouse clicking on the class in the JDeveloper Application Navigator and choosing **Create Data Control** from the context menu, the **DataControls.dcx** file gets created. The **DataControls.dcx** file describes the EJB session façade and its interfaces for the generic Data Control class to use. One information it contains is the definition of the **DataControlHandler** property:

```
DataControlHandler="oracle.adf.model.adapter.bean.jpql.JPQLDataFilterHandler"
```

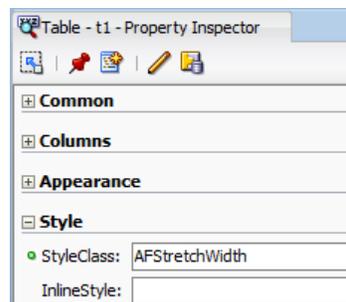
The **DataControlHandler** definition, if set to *JPQLDataFilterHandler*, ensures that the EJB access uses pagination of the ELB interfaces and the session façade contain the **queryByRange** method.



For existing, pre Oracle JDeveloper 11.1.13 or non-Oracle JDeveloper created EJB session facade, you can add pagination support by adding the *JPQLDataFilterHandler* configuration in the DataControls.dcx file and the **queryByRange** method entries in the EJB interfaces and session facade after upgrading to a recent version of Oracle JDeveloper.

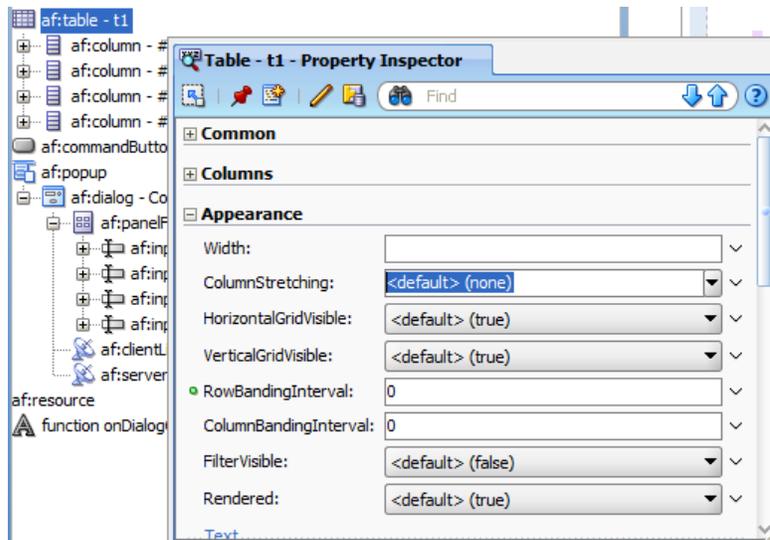
## How to equally stretch multiple table columns

The default table stretch behavior is such that no column changes its width in response to changes of the available real estate. In the example below, the table is not contained in a layout container that stretches its child components. The table shown below has the **StyleClass** property set to **AFStretchWidth** to force it to take the maximum width.



**Note:** If the table is enclosed by an **af:panelCollection** component, it automatically stretched to the size of the parent container.

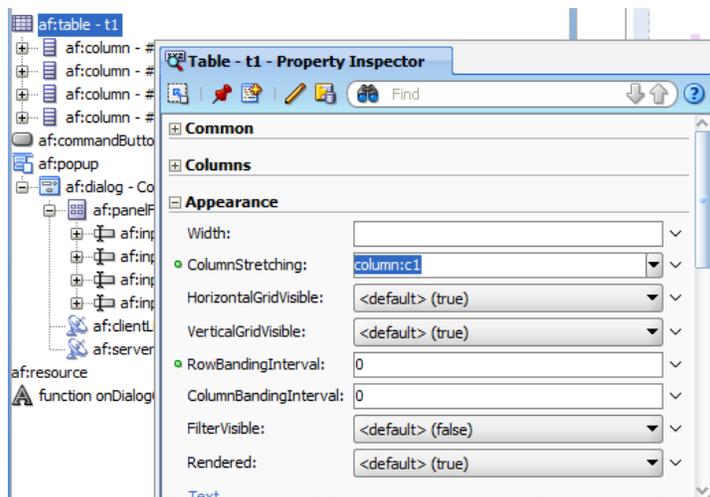
The images below show the default configuration of the table, as well as the behavior that shows when the available space changes, for example in response to users resizing the browser window.



As you can see, the table does not take all the available space. The table scrollbar shows to the right, indicating the possible width the table can take. Resizing the browser will change the blank area between the table columns and the scroll bar but doesn't change the size of the table columns.

DepartmentId	DepartmentName	ManagerId	LocationId
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Legal	203	2400
50	Human Resources	203	2400
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700

Changing the table configuration to maximize a specific column now fills the available blank space with the columns content.



The resize behavior is that a change of the available width first shrinks the column that is configured to fill the available space.

DepartmentId	DepartmentName	ManagerId	LocationId
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Legal	203	2400
50	Human Resources	203	2400
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700

Changing the **ColumnStretching** property to **multiple** and configuring the **af:column width** property (not the inline style width !) to a value of 33% now resizes all columns equally when the available maximal width changes.

The screenshot shows the Property Inspector for an `af:table` component. The `ColumnStretching` property is set to `multiple`. The `Width` property of an `af:column` is set to `33%`. Red arrows indicate the configuration changes mentioned in the text.

Resizing the browser window now treats the columns equal, as shown below.

DepartmentId	DepartmentName	ManagerId	LocationId
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Legal	203	2400
50	Human Resources	203	2400
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700

Before setting all tables and columns to multiple column stretching and 33%, make sure you have a look at the **af:table** and **af:column** tag documentations to learn about the performance impact that the dynamic column resizing comes with.

[http://download.oracle.com/docs/cd/E17904\\_01/apirefs.1111/e12419/tagdoc/af\\_table.html](http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_table.html)

[http://download.oracle.com/docs/cd/E17904\\_01/apirefs.1111/e12419/tagdoc/af\\_column.html](http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_column.html)

---

**RELATED DOCUMENTATION**

---

<input type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	