ADF Mobile Code Corner

m03. Creating dependent lists from independent collections

twitter.com/adfcodecorner

Abstract:

Dependent lists is a common functional requirement for web, desktop and also mobile applications. You can build dependent lists from dependent, nested, collections and from independent collections, in which the detail data is queried from a method call. With step-by-step instructions, this article explains the latter use case in which a selected parent data object is passed to a method to query the detail data set.

Note that fimilar practices however can be applied to nested collections as well.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
11-MAR-2013

Oracle ADF Mobile Code Corner is a blog-style series of how-to documents targeting at Oracle ADF Mobile that provide solutions to real world coding problems. ADF Mobile Code Corner is an extended offering to ADF Code Corner

Disclaimer: All samples are provided as is with no guarantee for future upgrades or error correction. No support can be given through Oracle customer support.

Please post questions or report problems related to the samples in this series on the OTN forum for Oracle JDeveloper: http://forums.oracle.com/forums/forum.jspa?forumID=83

Introduction

Step-by-step, this article shows how you build dependent lists in an ADF mobile application using the Oracle HR database schema accessed from a WS SOAP Service. Note that it doesn't matter if the data is queried from a SOAP service or REST service because for performance reason you want to cache queried data locally in SQLite database or POJO beans so that the list is built from a POJO data control structure (though using a WS DC works the same).

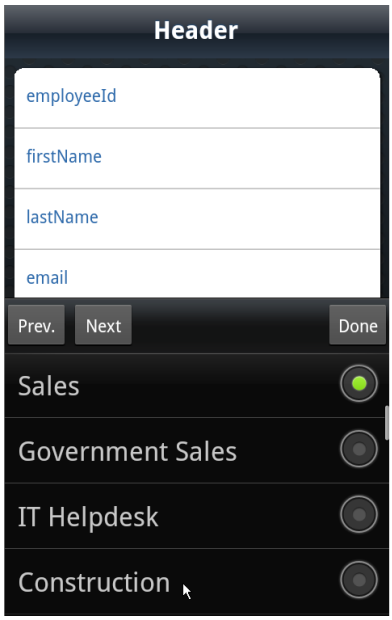
Note that the topic of how to query WS data from a sever to save them locally in a POJO data control structure is explained in an upcoming ADF Mobile Code Corner article *"m05. How-to cache WS queried data locally"*.

http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html#CodeCornerMobileSamples

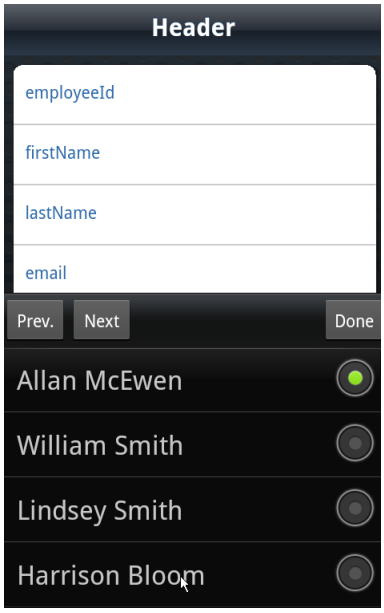
Dependent lists can be created from dependent master-detail collections or from independent collections. Bandwidth and UI real estate typically are limiting factors for mobile applications and so I think on-demand data queries are better for good application performance than downloading complex master-detail structures using SOAP services.

In this sample, the assumption is made that the detail data is queried in response to a selected master data row.

The screenshots below show the use case explained in this article. The dependent list is defined between the departments list and the manager Id list, which shows a list of all employees that are within the selected department.



E.g. selecting *Sales* as the department for a new employee to create on the mobile device …

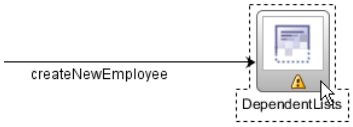


… synchronizes the list of values in the manager Id list to employees of the selected department.

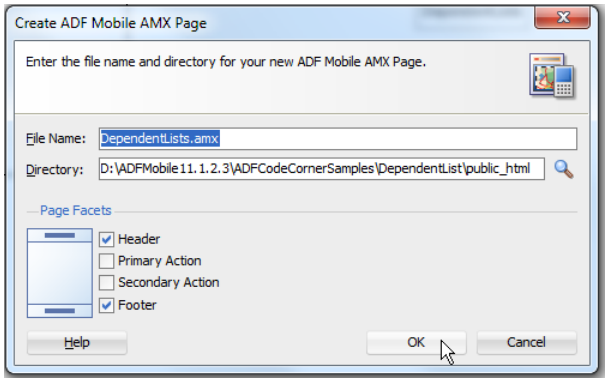
Creating Dependent Lists

This step-by-step instruction is following sample *m05. How-to cache WS queried data locally* that will be publish later on ADF Mobile Code Corner. However, the setup of the dependent list is generic and should easily translate to your custom use case and ADF Mobile model.

To create a new ADF mobile page you drag and drop a view activity from the JDeveloper component palette onto the bounded task flow diagram.



Double-click on the view activity to create the physical AMX page. In the sample I built the page with a *Header* and *Footer* facet. The footer facet holds the submit- and cancel buttons so they are always visible.



As shown in the image below, the new employee edit form is created from the return collection exposed on the *createEmployee* method. The method creates a new `ArrayList` and adds an empty employee object (instance of the Employee entity). The new employee object is accessible from the ADF iterator

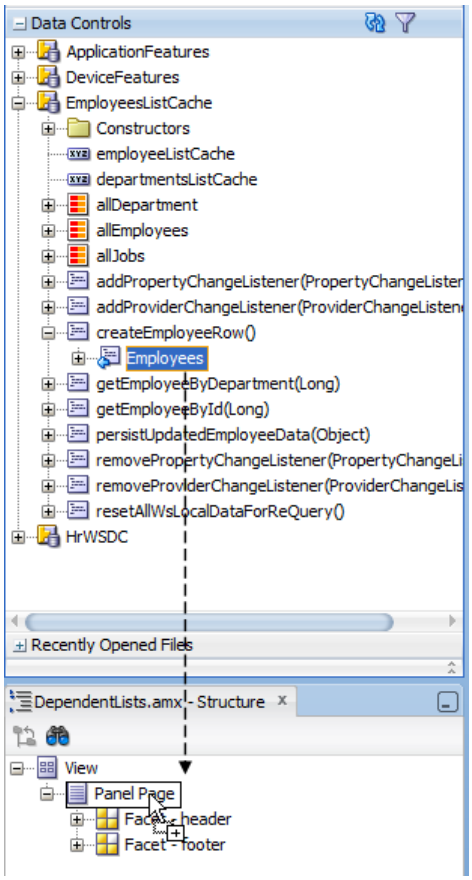
```
#{bindings.iterName.currentRow.dataProvider}
```

This object is referenced in a call to the *persistUpdatedEmployeeDataObject* method to commit changes.

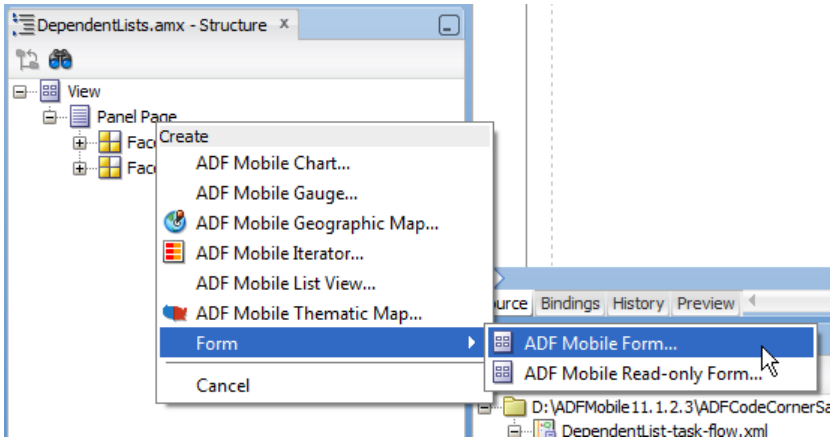
Note that with POJO DC, data persistence is not implicit (in opposite to ADF Business Components) and instead a persistence method needs to be called explicitly. This also is the case when using the REST or WS DC.

As shown in the image below, you create a new edit form by dragging a collection from the DataControl palette onto the AMX page. In this sample use case the page is for creating a new Employee record, and so the collection is dropped from the result set of the `createEmployeeRow` method that creates an new `ArrayList` with a new Employee entity in it.

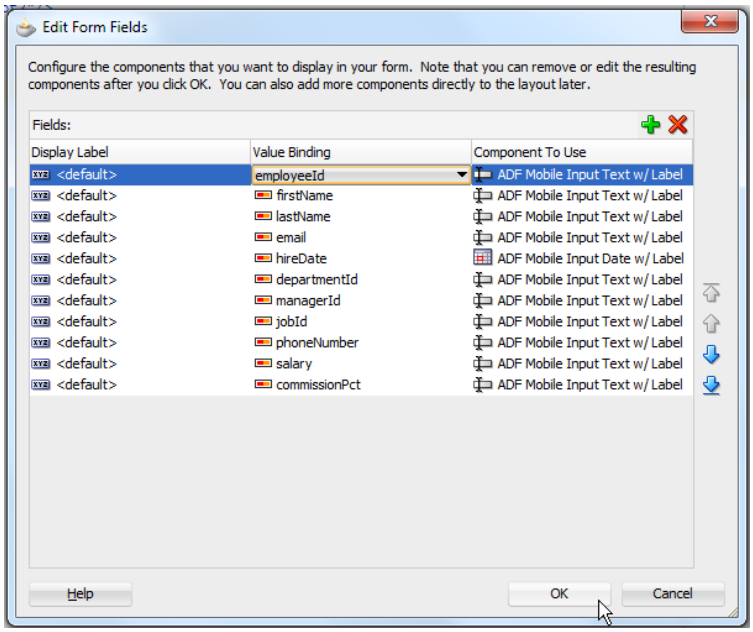
Note: Once sample *m05. How-to cache WS queried data locally* becomes available for download you can have a look at this method in the `EmployeesListCache.java` class. You will then also see that the ADF Mobile application contains local entity files that simulate entities on the server to update the data model.



When you drop the collection onto the AMX page (here I chose the drop target to be in the Structure Window as it gives me better control of where exactly components should be created into) a context menu opens that shows you all the mobile UI widgets that you can bind to a collection. To create a form for editing a new employee, you select *ADF Mobile form*.

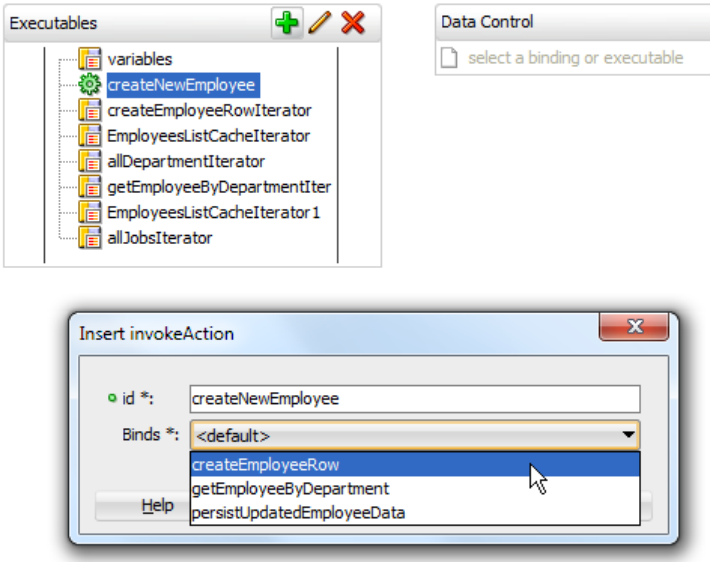


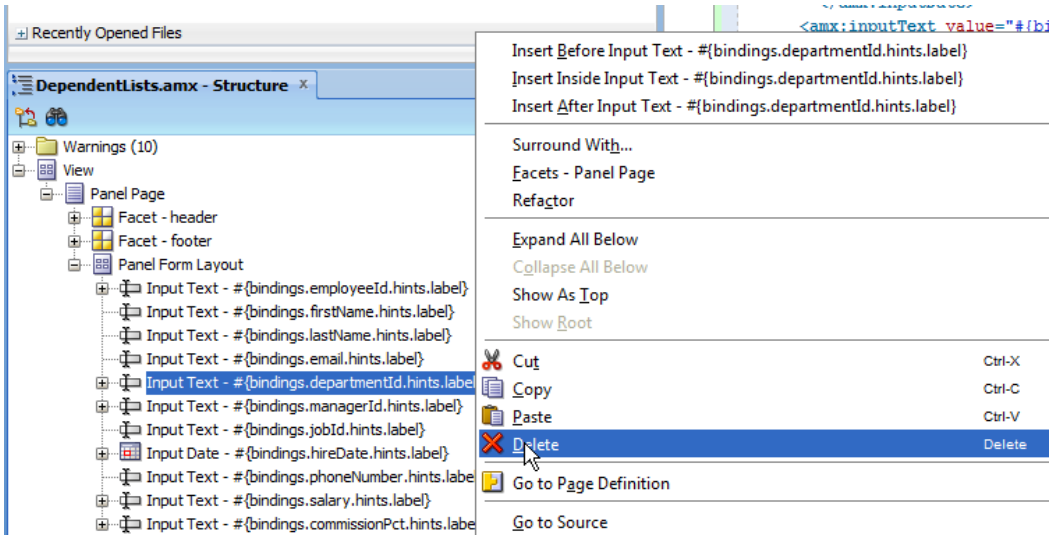
In the opened dialog, you can re-order the list of employees attributes, which by my experience is often necessary when working with POJO, WS and EJB business services.



After pressing OK, the form is created on the AMX page. It does not yet contain list of values because in version 1.0 there is no support for model driven list of values for the POJO data control. So next you will create the lists manually.

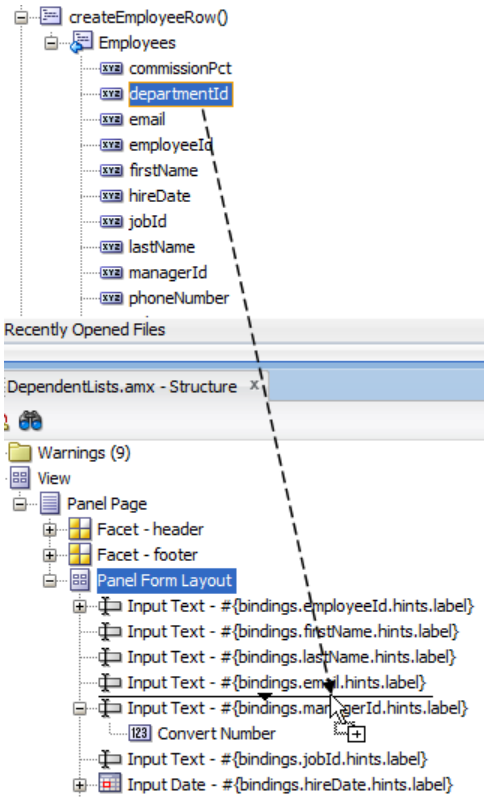
Hint: When you create a form from a method result then the method too is added to the ADF PageDef file (the binding file). In this sample case, a new employee record should be created when the create page is navigated to. To enforce this you use an **invokeAction** binding. As shown in the image below, click the green plus icon in the PageDef's *Executable* section and select the *invokeAction* operation. Provide a unique *Id* value and select the method binding to invoke.



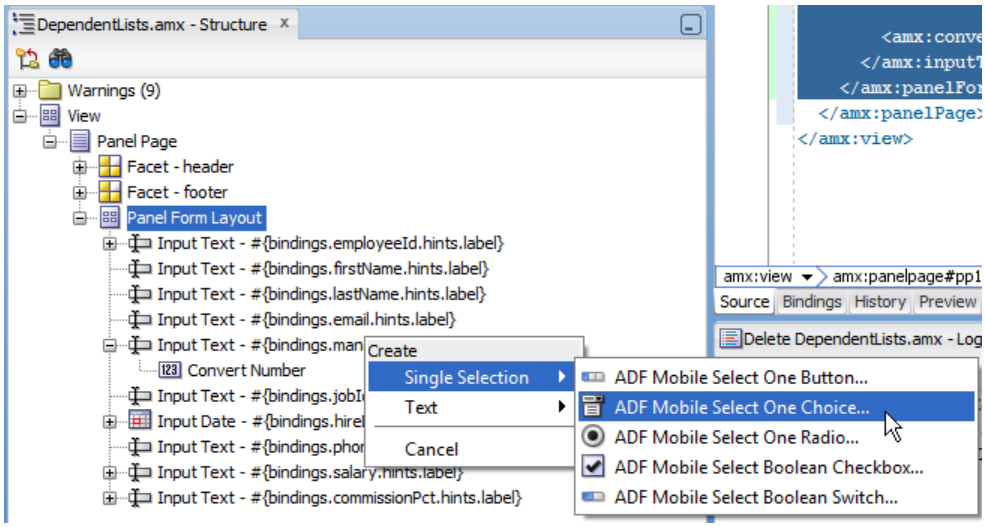


To create a list for the *departmentId* parent attribute, select the input text field in the Structure Window and choose **Delete** from the right mouse menu. This delete, unlike the delete of the same component in the page source code editor, also deletes the binding in the associated PageDef file. Note that it is important to delete the binding because the list requires a different type of binding (list binding) than the simple input field (attribute value binding).

To create the *departmentId* list, you drag and drop the *departmentId* attribute contained in the Employees collection node of the *createEmployeeRow* method.



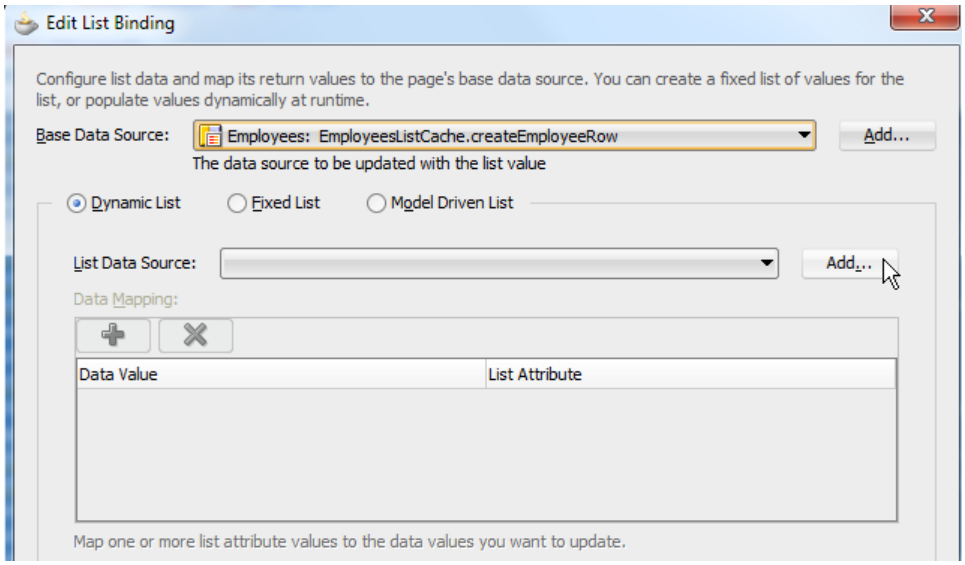
Drag and drop the departmentId attribute and choose *Single Selection* → *ADF Mobile Select One Choice* component from the context menu.



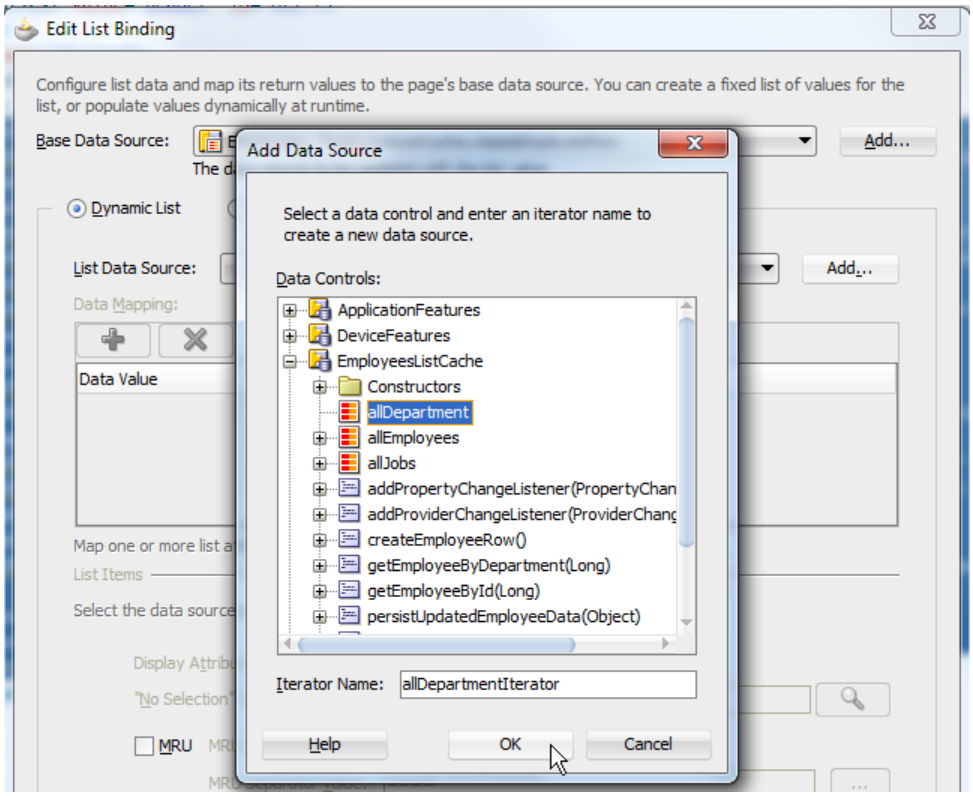
The opened *Edit List Binding* dialog allows you to build a new dynamic or static list to populate the list items. Dynamic lists is where the value options of a list are queried at runtime from a business service collection or method and also is the option used in this sample.

Note: Dynamic lists also exist in the Oracle ADF version for building web applications. So this should be a familiar concept for all ADF developers.

In the *Edit List Binding* dialog, you press the **Add** button next to the *List Data Source* label to select a collection that holds the list data.

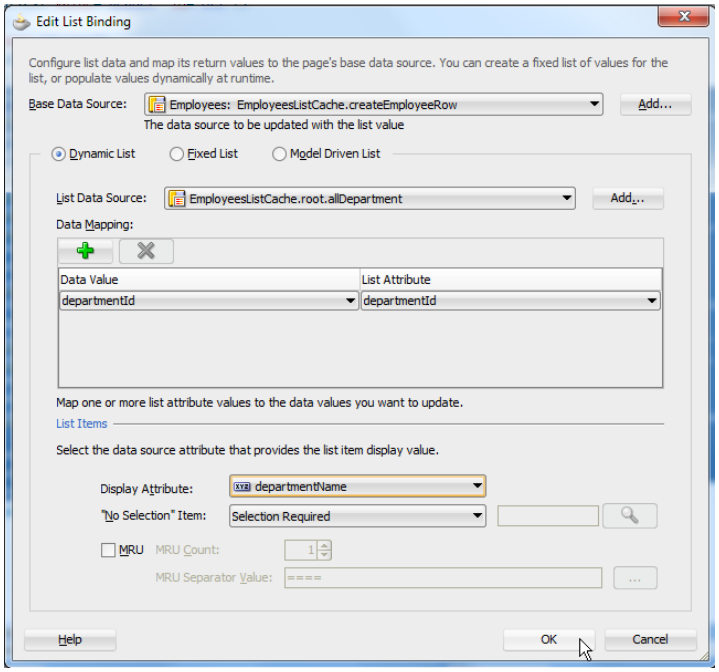


In the sample case, the list data comes from the *allDepartments* collection. Note the difference in the icons between methods that require input arguments and methods that don't.



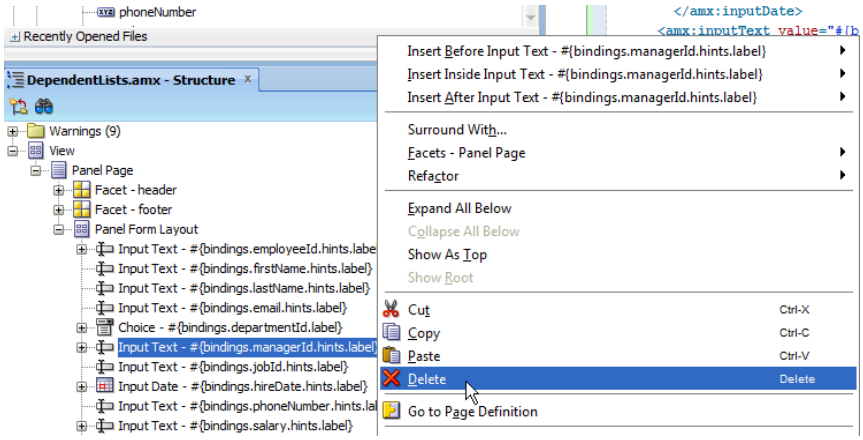
Clicking **OK** creates a iterator in the AMX page PageDef file.

As shown in the image below, once the iterator is created, you map the list attribute to the corresponding source collection attribute to update with the selection. The *Display Attribute* list box allows you to choose one or more attributes of the list collection to display in the select component.

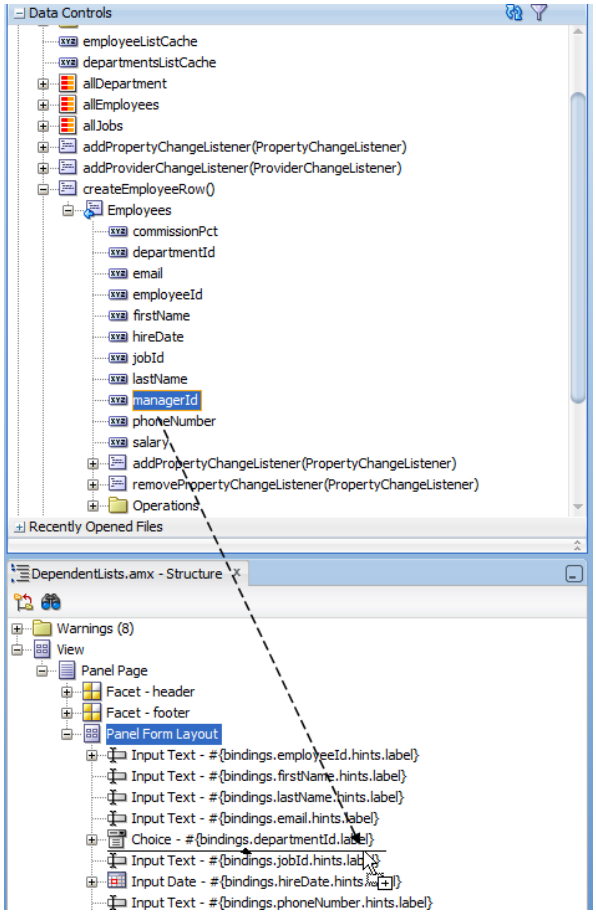


Pressing OK the creates the parent list.

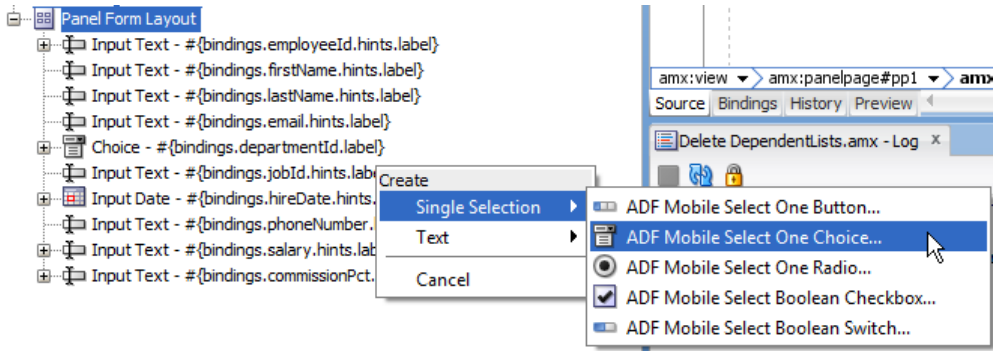
To build the dependent lists, you do the same as for the parent list box and start with deleting the input field for the attribute to replace with a select list.



Like before, you drag the list attribute (managerId in this sample) from the Data Control palette to the AMX page (again using the Structure Window).



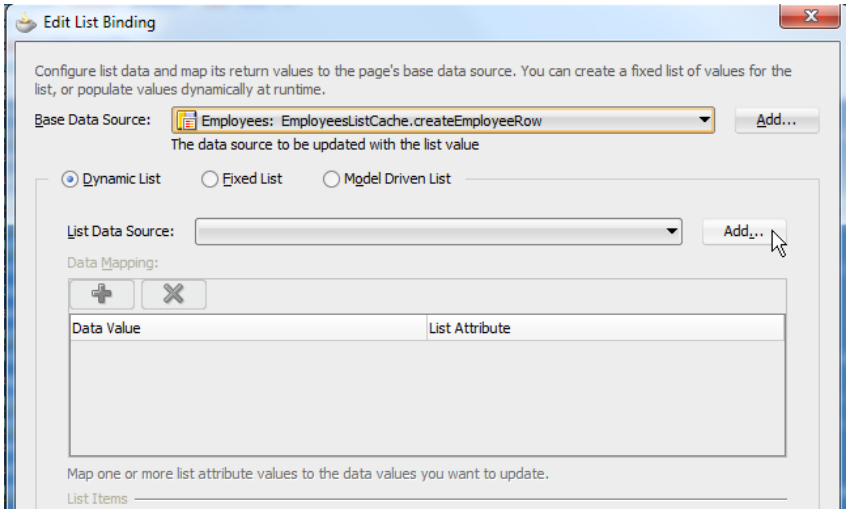
Like before, you choose the *Single Selection* → *ADF Mobile Select One Choice* component option in the context menu.



In the Edit List Binding dialog, you now select the dependent list (child list) collection. In this sample, the dependent list shows all employees of a department, pretending that only employees of a department can be a manager for employees therein.

As mentioned earlier, dependent lists can be created through nested collections (e.g. the allDepartments collection containing a collection of employees) in which case the master-detail synchronization happens simply by making a selected parent row the current row in the binding layer, or through a method call in which you pass the dependency as an argument.

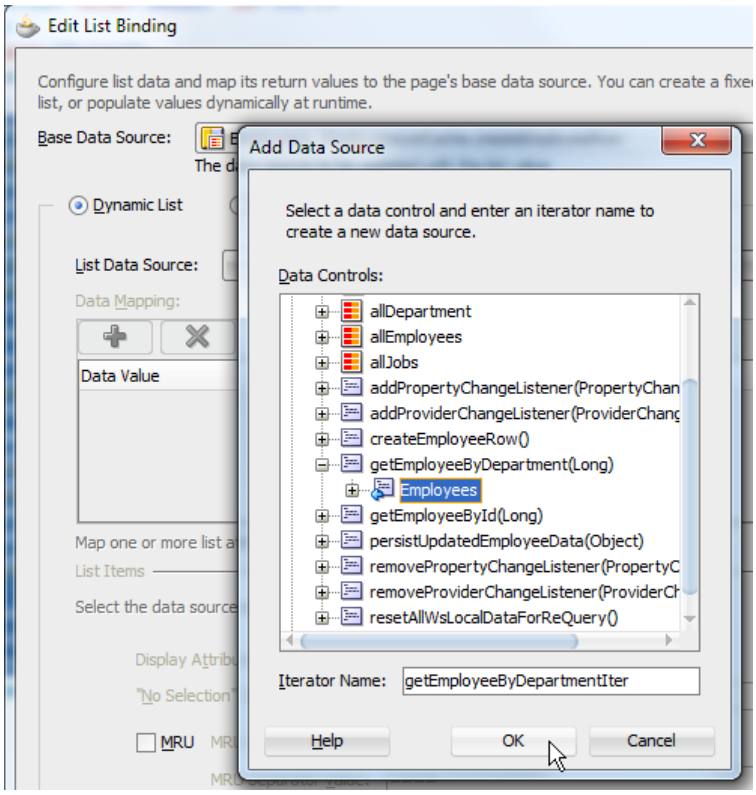
The latter approach is used in this sample.



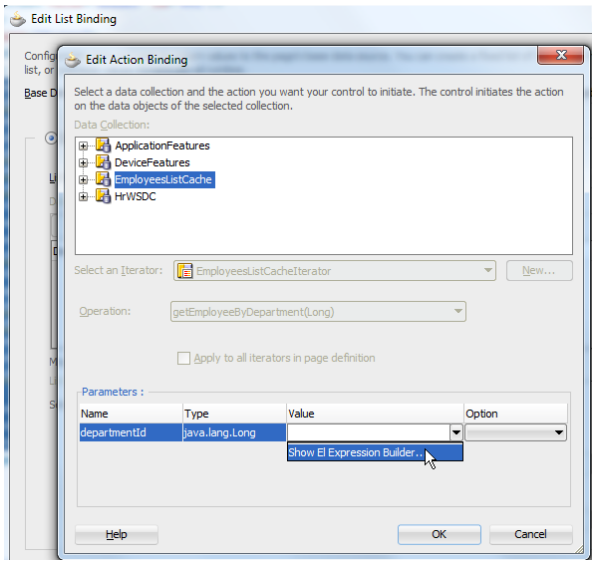
The dependent list is created from the *Employees* collection exposed on the *getEmployeeByDepartmentId* method that takes the *departmentId* as an input parameter.

Note: Sample *m05. How-to cache WS queried data locally* is about caching server queried data locally on the mobile device. This caching is handled by a POJO sample in *m05*. The *getEmployeeByDepartmentId* method searches the cashed data for all employees that meet the input argument criteria.

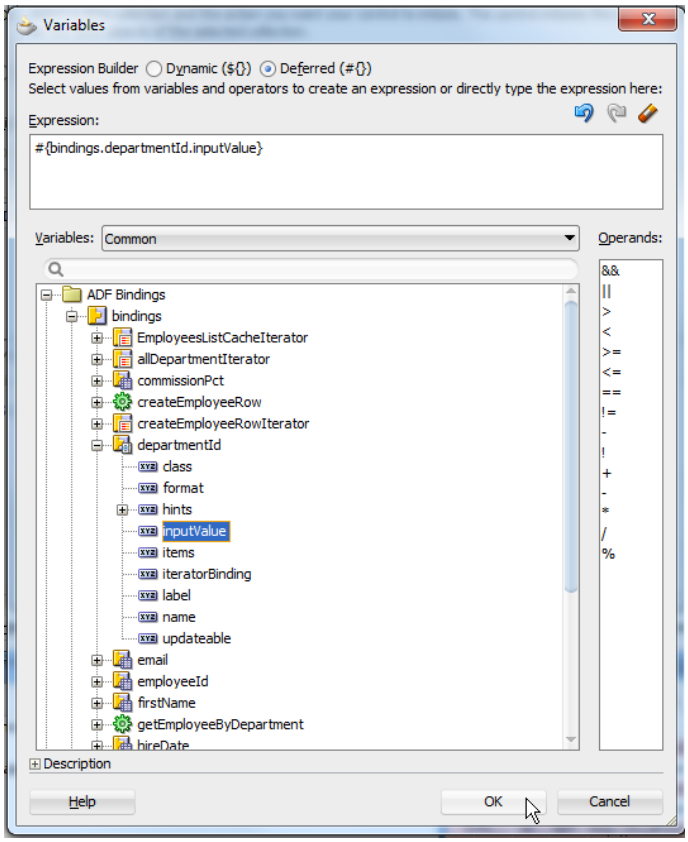
Note: Using locally cached data you can provide functionality to WS or RS service queried data that is not part of the WS or RS service API definition.



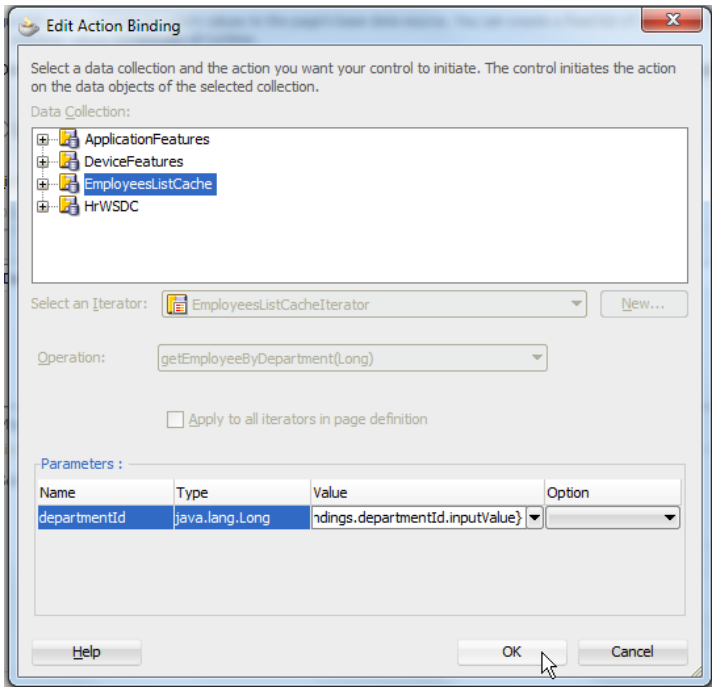
Because the selected method requires an argument to be passed with the request, the *Edit Action Binding* dialog opens for you to define the argument value object.



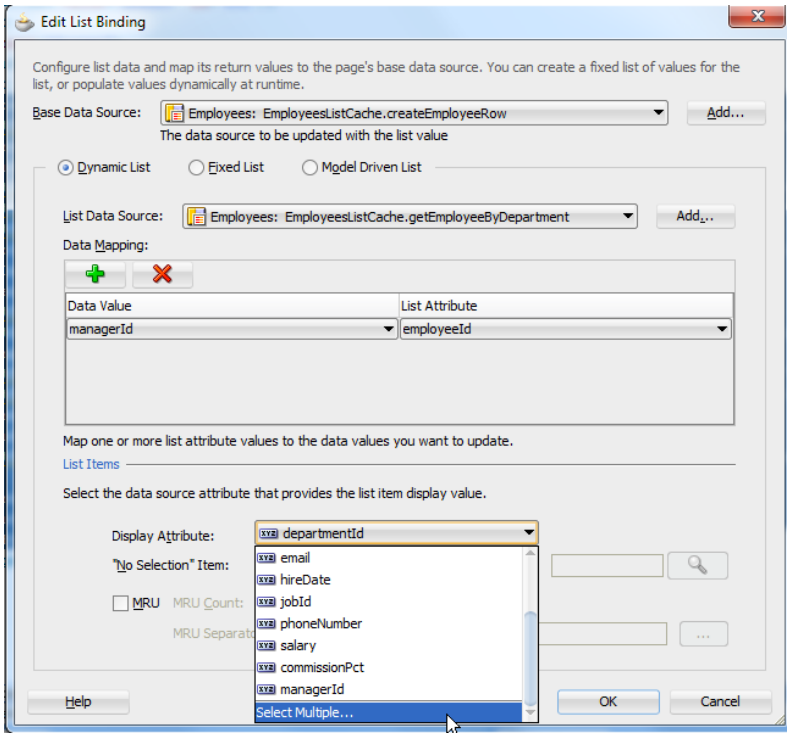
Because the dependent value is the selected *departmentId* value, you use the *EL Expression Builder* dialog to select the *bindings* → *departmentId* → *inputValue* entry. This reads the current ADF binding value of the departmentId attribute into the method call.



The EL resulting out of the above action is shown in the image below.

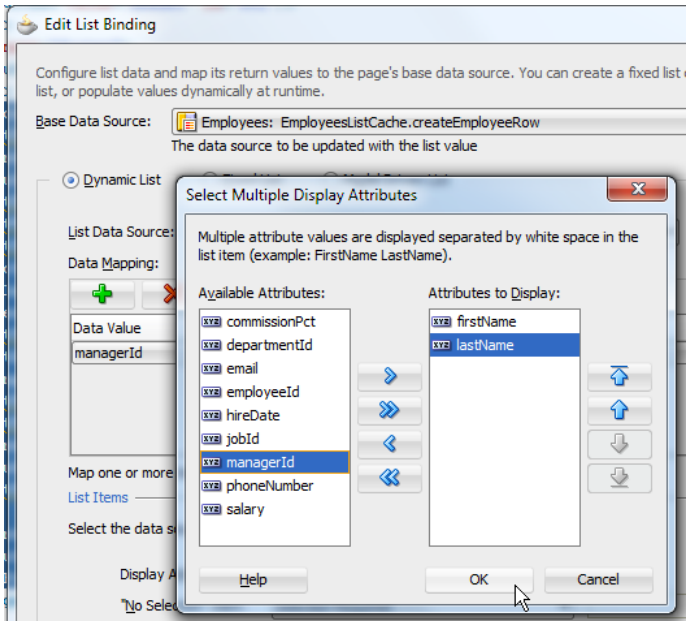


Like before, you use the *Edit List Binding* dialog to map the select list attribute to the attribute to be updated by the selection. Similar, you choose a display value.

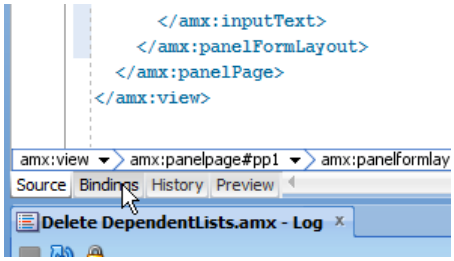


In the sample, as shown in the image below, I defined the display label as First Name and Last Name

Note: Though the lists on mobile devices show with enough space for longer list labels, you want to make sure the list shows well on mobile phone devices and tablet devices.



With the two dependent lists items (departmentId and managerId) created, you need to switch to the AMX page's PageDef file, which you can from the page source editor. As shown in the image below, you simply click onto the **Bindings** tab at the bottom of the editor.



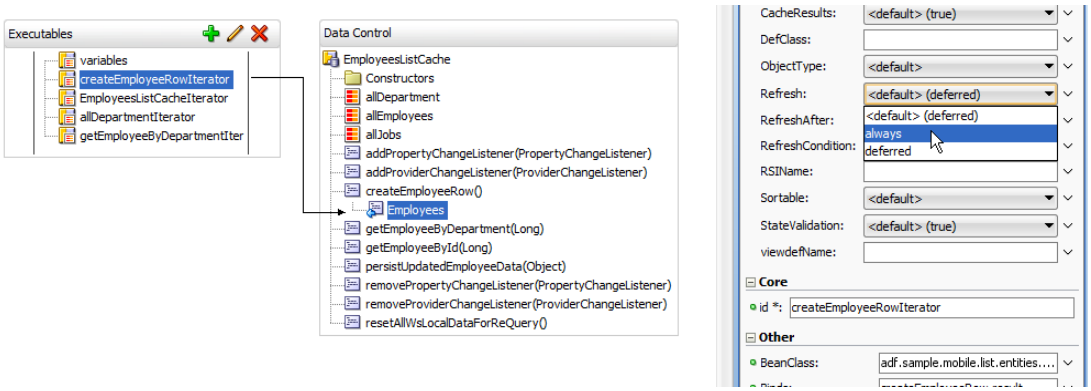
In the Binding editor, select the base iterator (which is the editor that you update with the selected list data. Iterators are always shown in the *Executable* sections of the binding editor.

******* IMPORTANT *******

Open the Property Inspector (ctrl+shift+I) and set the **Refresh** property to **always**. This is an important step for this use case which is why I highlighted this. Setting the *Refresh* property value to **always** ensures that the attributes of this iterator are refreshed twice and not only before the UI renders.

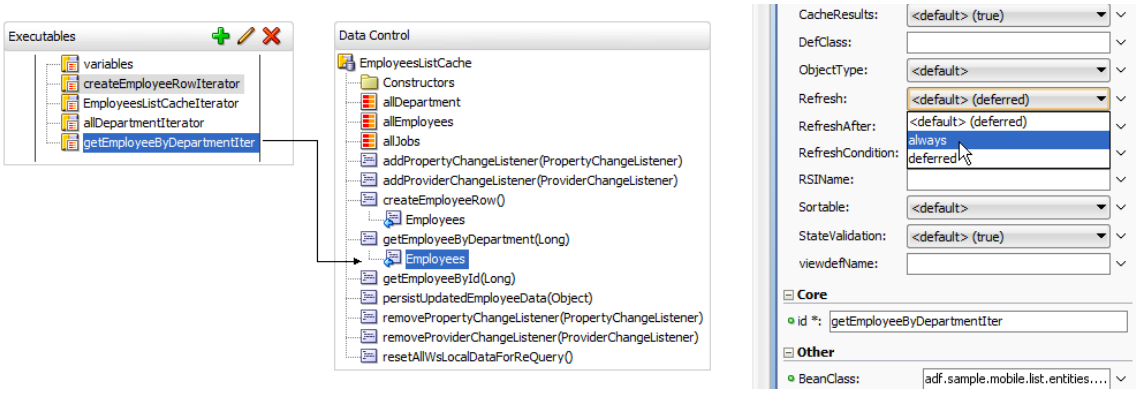
****

Because the *managerId* list requires the selected *departmentId* attribute value to be available, we need the departmentId value selection to be saved in the ADF model before rendering the UI.



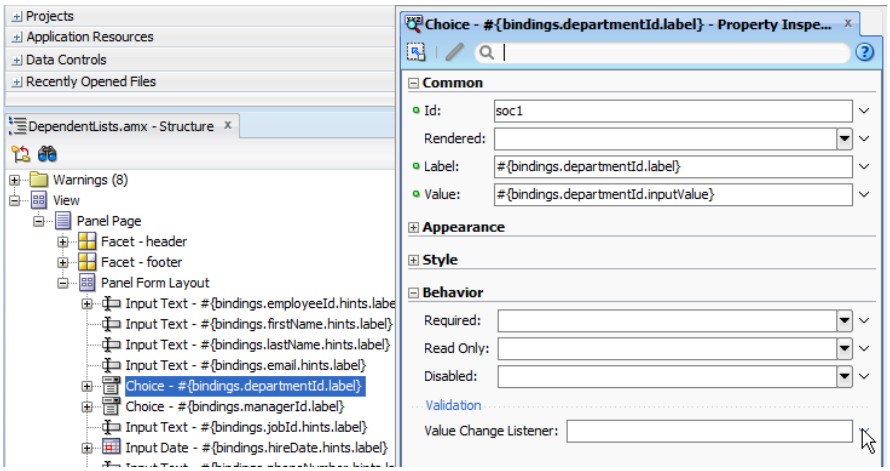
Do the same setting for the *getEmployeeByDepartmentIter* entry. This way the two iterators are refreshed twice, which gives you a good chance that the dependent data lists are synchronized before the UI renders.

Note that because data is cached and queried locally, there is no cost with this double refresh behavior because not server side data is re-queried.

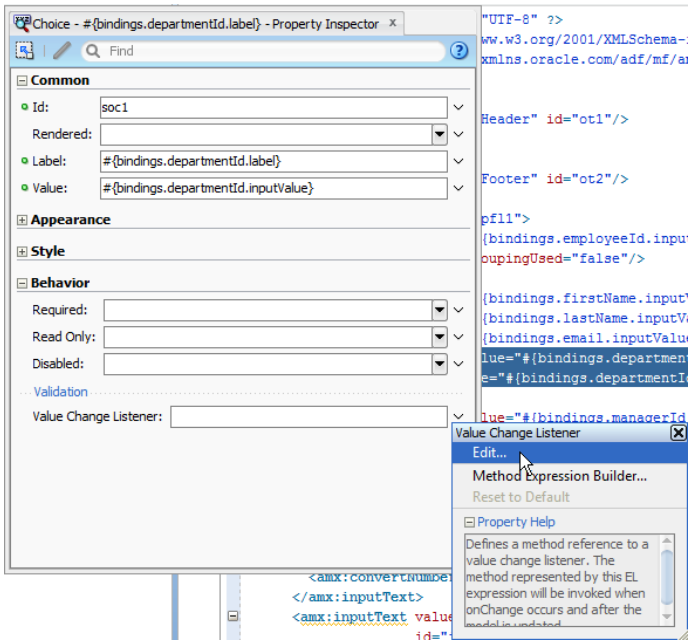


Last but not least, you need to tell the ADF binding about the value change in the departmentId, which is what you use the component's *ValueChangeListener* for.

Note: This step may not be required if the dependent list data is queried from a embedded (nested) collection. Note however that using nested collections may unnecessarily increase the downloaded data size.

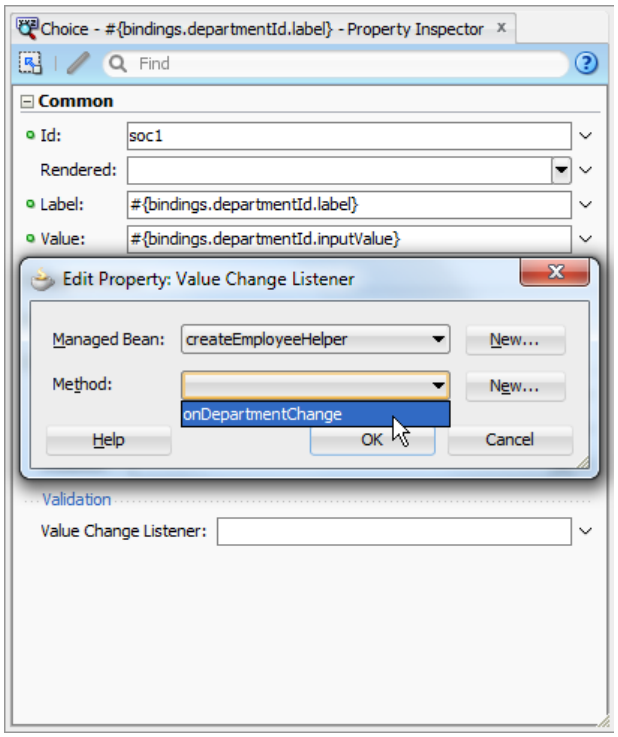


Navigate to the input field *Value Change Listener* property and press the *arrow* icon at the end to create the managed bean (or select a managed bean) and the method therein.

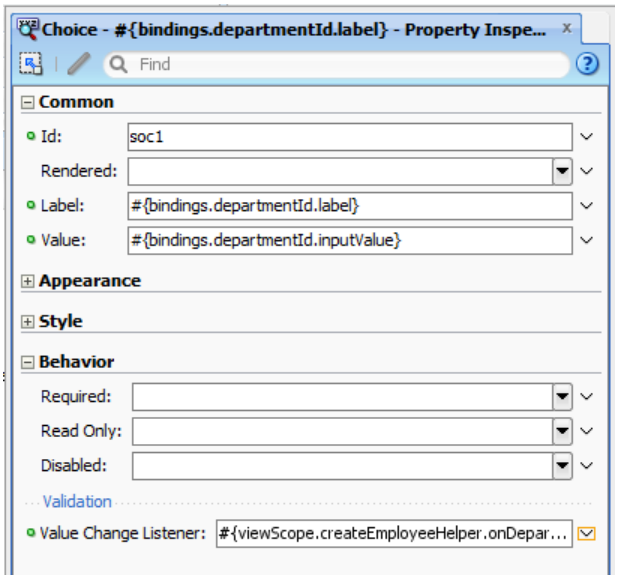


As shown in the image above, you use the *Edit* option in the opened *Value Change Listener* edit dialog.

Create a managed bean (viewScope) and the listener method (*onDepartmentChange* method in this sample).



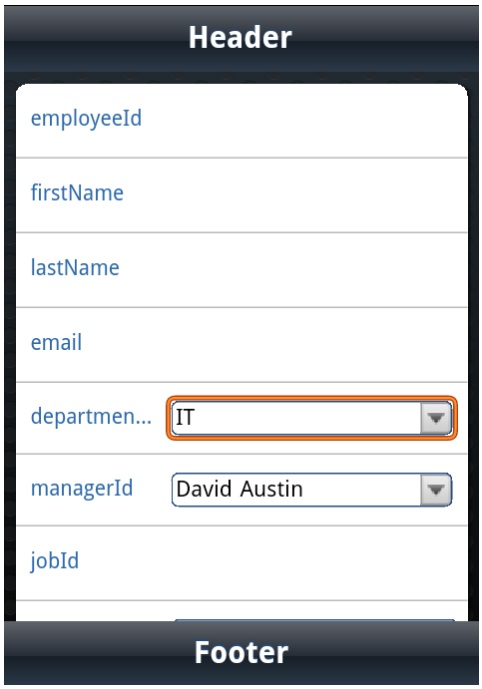
The configuration will look as shown in the image below. Notice the memory scope prefix – viewScope – in the EL.

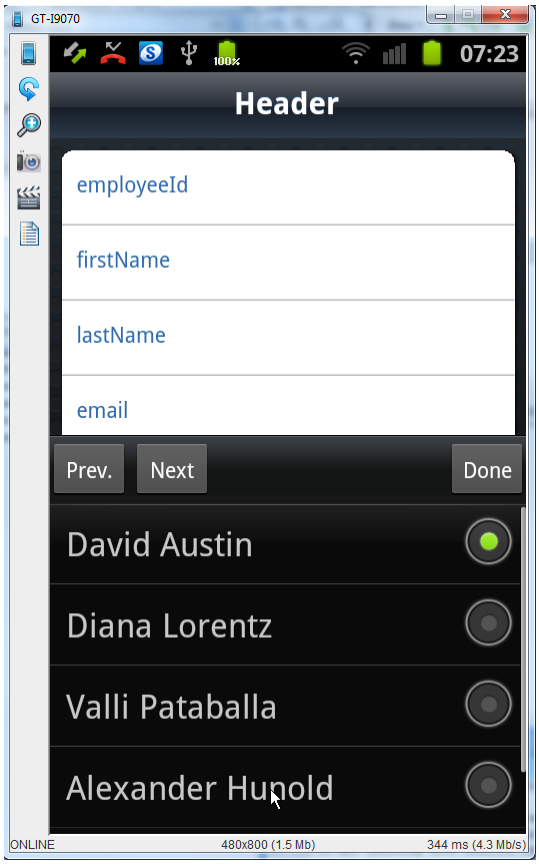


The managed bean source code is provided at the end of this article. Basically what it does is:

- Get the selected department Id value
- Access the method to query the dependent employee collection
- Pass the selected departmentId value as an argument to the method and execute it
- Refresh the list iterator

The two images below show the runtime UI for the dependent lists.





Managed Bean: Value Change Listener

As mentioned before, the managed bean does the following:

- Get the selected department Id value
- Access the method to query the dependent employee collection
- Pass the selected departmentId value as an argument to the method and execute it

The work is done within the value change event handler method, which is listed below.

```
import oracle.adfmf.amx.event.ValueChangeEvent;
import oracle.adfmf.bindings.OperationBinding;
import oracle.adfmf.bindings.dbf.AmxIteratorBinding;
import oracle.adfmf.framework.api.AdfmfJavaUtilities;
```

[…]

```
public void onDepartmentChange(ValueChangeEvent valueChangeEvent) {
 //get reference to method binding that queries data for the dependent
```

```
 //list
  ValueExpression ve =
                (ValueExpression)AdfmfJavaUtilities.getValueExpression
                ("#{bindings.getEmployeeByDepartment}",Object.class);

  OperationBinding method = (OperationBinding)ve.getValue
                            (AdfmfJavaUtilities.getAdfELContext());

  //pass new departmentId argument and execute method
  method.getParamsMap().put("departmentId",
                            valueChangeEvent.getNewValue());
  method.execute();


  //refresh iterator
  ValueExpression veIter =
              (ValueExpression)AdfmfJavaUtilities.getValueExpression
              ("#{bindings.getEmployeeByDepartmentIter}",Object.class);

  AmxIteratorBinding iteratorBinding =
                       (AmxIteratorBinding)veIter.getValue
                       (AdfmfJavaUtilities.getAdfELContext());
  iteratorBinding.getIterator().refresh();
}
```

RELATED DOCOMENTATION

☒	ADF Mobile Developer Guide http://docs.oracle.com/cd/E35521_01/doc.111230/e24475/toc.htm
☒	ADF Mobile Home Page http://www.oracle.com/technetwork/developer-tools/adf-mobile/overview/index.html
☒	ADF Mobile Framework API http://docs.oracle.com/cd/E35521_01/apirefs.111230/e27204/index.html?oracle/adfmf/framework/api/AdfmfJavaUtilities.html
☒	ADF Mobile Blog https://blogs.oracle.com/mobile/