ADF Code Corner

037. How-to build pagination into ADF POJO Data Control

twitter.com/adfcodecorner

Abstract:

The JavaBean Data Control in ADF (Oracle Application Development Framework) is special in that it can be used to bind the ADF model to business service or data sources that have no native Data Control representation in ADF. To use the JavaBean Data Control, developers build a POJO wrapper for the data source or business service they want to access and expose public methods on it that ADF metadata descriptions should be created for. A new feature of the ADF JavaBean Data Control that we refer to as pagination and that comes with Oracle JDeveloper 11g R1 PS1 is the ability to query data in chunks, which is of much better performance than querying all data at once. Of course, the feature requires the target data source or business service to supports pagination. In this article, I explain how to use the JavaBean Data Control and what needs to be done on the Java wrapper to enable pagination. For existing implementations of the JavaBean DataControl you learn what you need to do to change the behavior from "all you can eat" queries to pagination. The good news here is that there is no need required in the application, just the bean model.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
23-MAR-2010

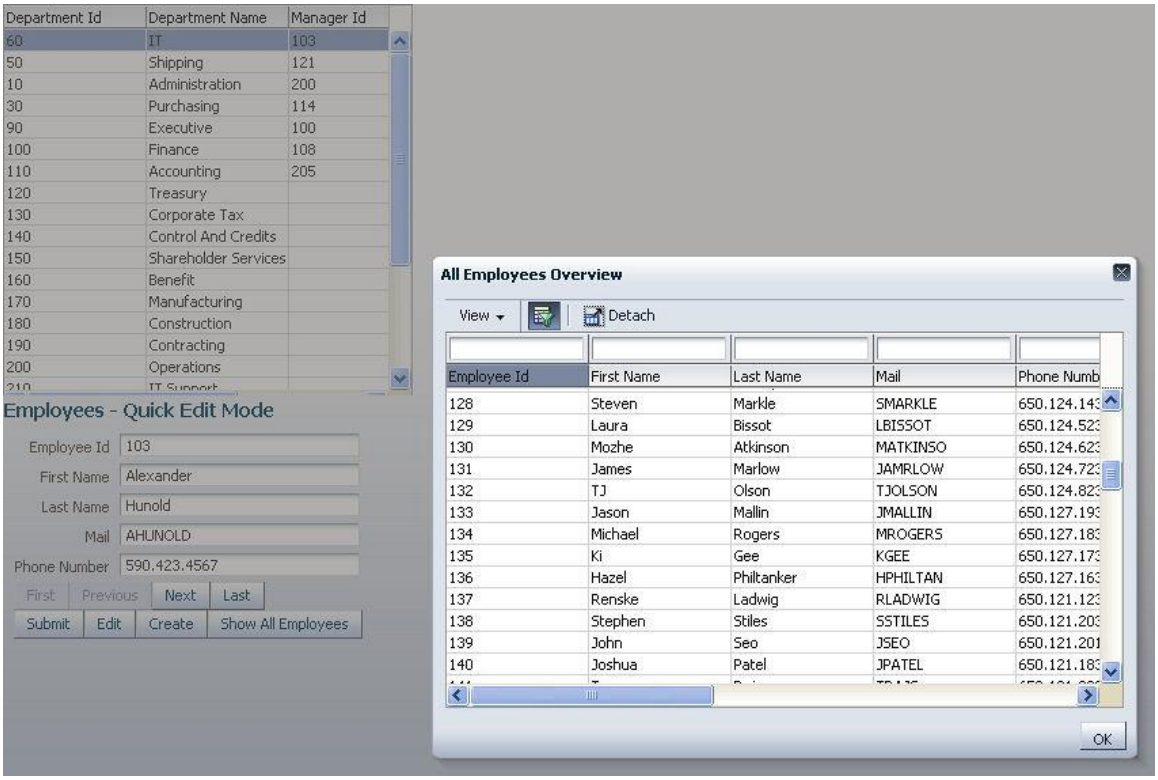
Oracle ADF Code Corner is a loose blog-style series of how-to documents that provide solutions to real world coding problems.

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

The JavaBean Data Control uses metadata that describes method and attributes exposed on a POJO to enable ADF access to the target business service or data source the POJO wraps. A JavaBean that exposes methods to access a WS proxy client for example can be used with the JavaBean Data Control to build a page like shown below . Same for a JavaBean layer that accesses LDAP or Spring, or a Coherence instance. The image below shows an ADF bound ADF Faces table and input form that accesses a JavaBean through the ADF binding layer and JavaBean Data Control.

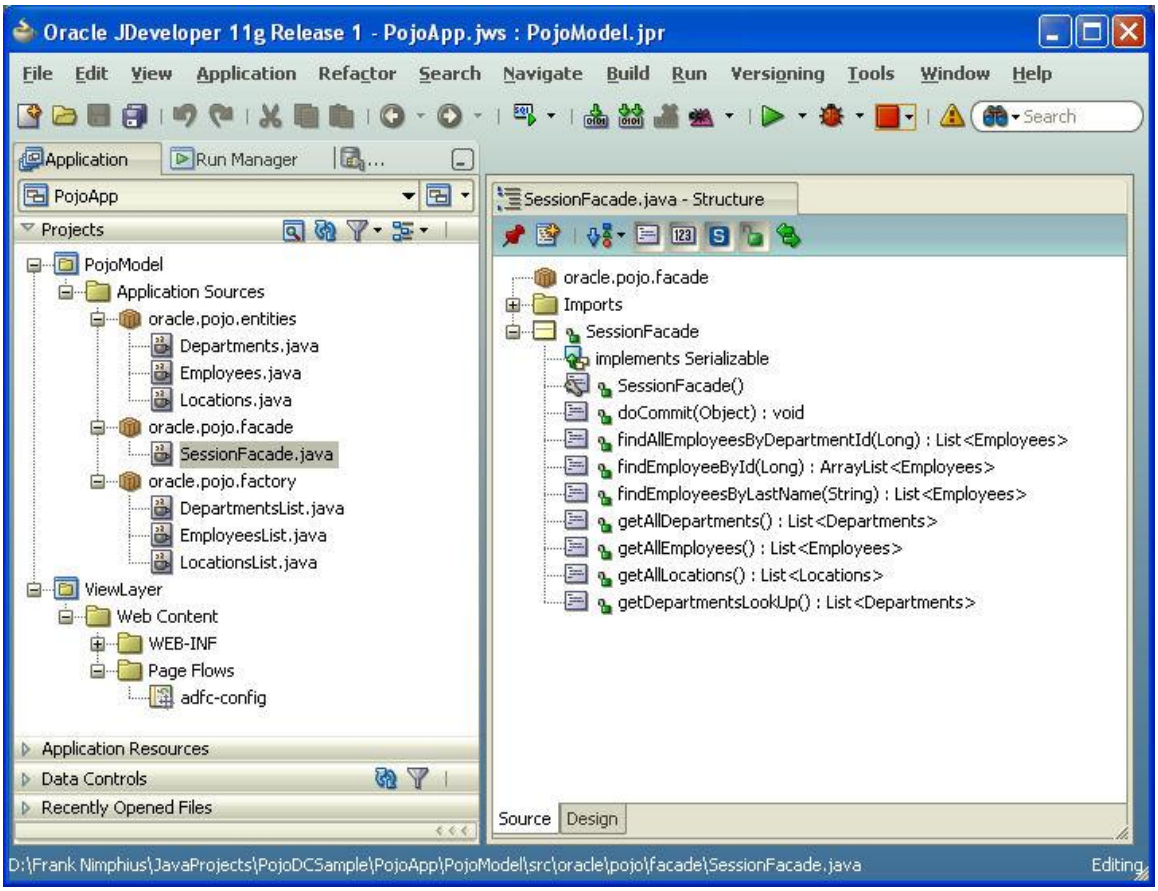


From the image above, its hard to tell what the underlying business service is, which is exactly what ADF is meant to do: providing a consistent API abstraction for application developers to focus on the business functionality to build and not the implementation technology of the underlying data source or business

service. If I told you that the image above uses ADF Business Components underneath, you would have believed me too, I have no doubt about it.

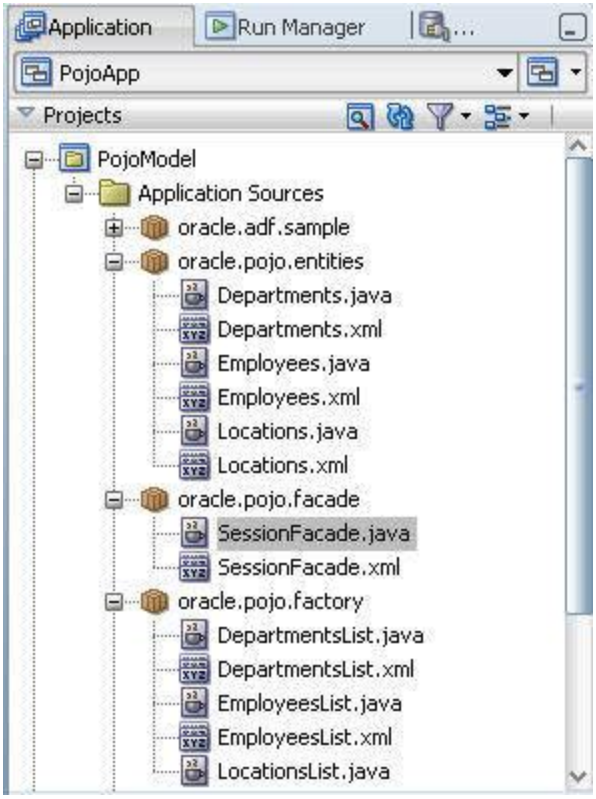
The POJO model

To use the JavaBean Data Control, developers build a POJO model for the data source or business service to access. In the example you can download at the end of this article, the JavaBean model exposes static data - copied from the Oracle HR database schema - provided by the LocationsList. DepartmentsList and EmployeesList object in public methods of the SessionFacade bean.

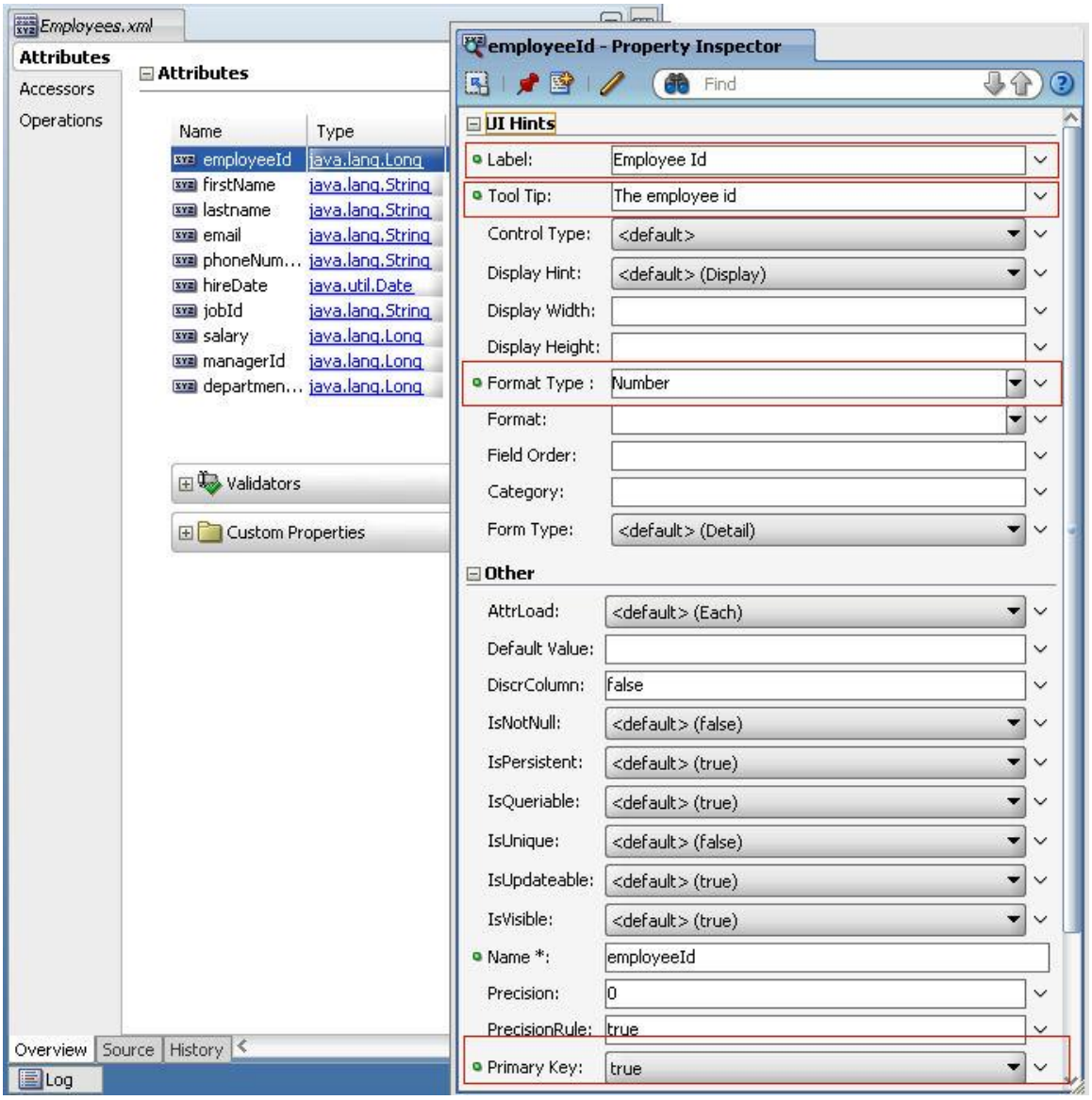


Methods shown in the Structure window that start with "get", like getAllDepartments, getAllEmployees and getAllLocations, show as collections "allLocations","allDepartments", and "allEmployees" in the Data Control. To create a JavaBean Data Control, select the bean wrapper, "SessionFacade" in the example, and choose "Create Data Control" from the context menu.

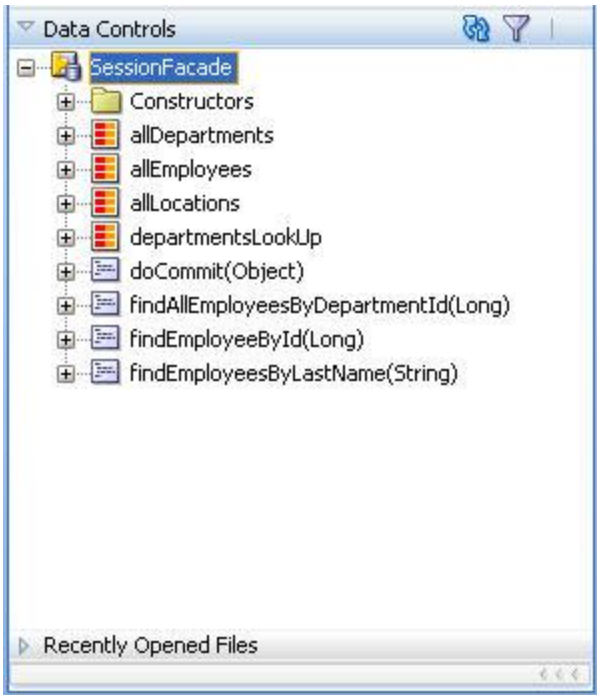
The image below shows the outcome of the Data Control metadata creation. Note that the menu option "Create Data Control" not quite describes what it does. The JavaBean Data Control exists in ADF and all that this menu option does is to create the metadata that describes the JavaBean exposed public methods and attributes.



As shown in the image above, metadata is created for all Java classes in the model. The metadata for the entities, Locations, Departments and Employees, can be further customized using the JDeveloper Property Inspector or the edit dialog that opens when double clicking on it.



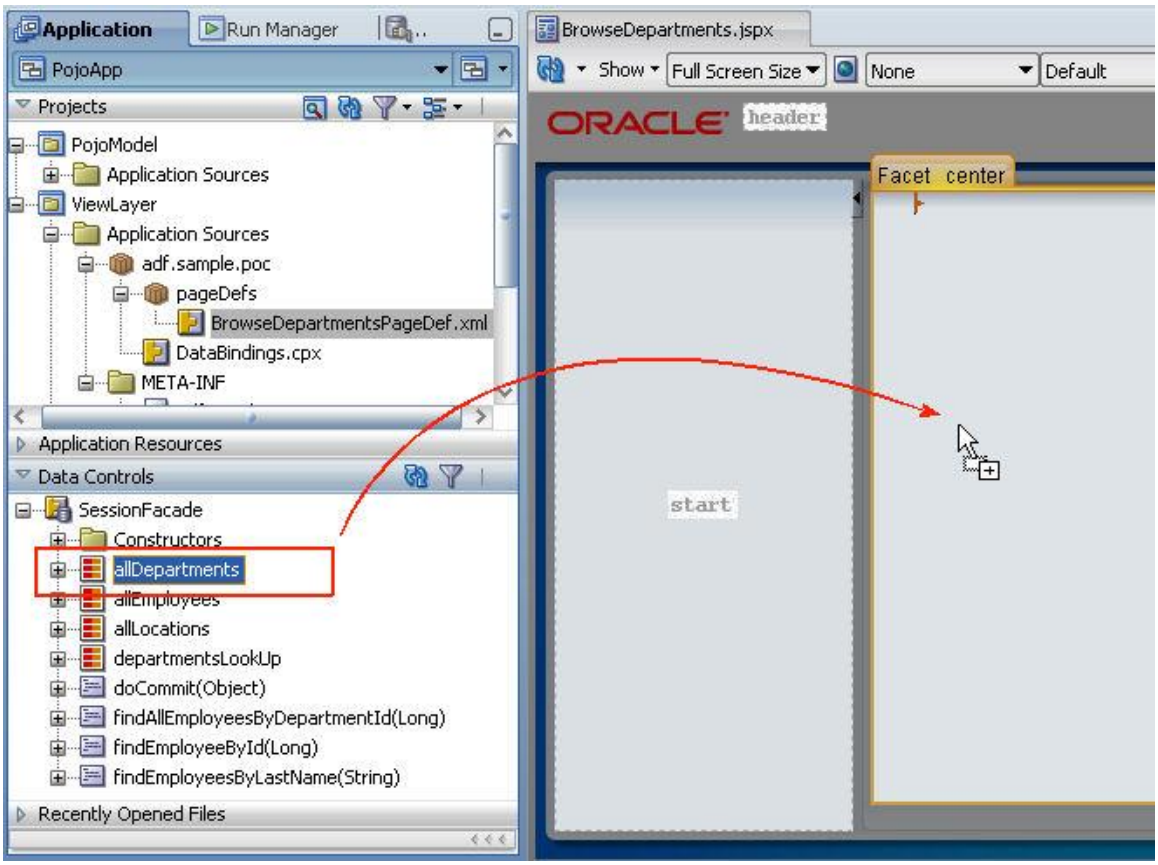
In the image above, the Property Inspector shows the metadata attributes you can set for the EmployeeId attribute in the Employees POJO. Note the ability to define proper prompts (label), tool tips, format types and primary key settings. All strings are stored and referenced in a properties file so they can be translated. The Employees.xml editor that lists all the entity attributes has a pencil icon in its upper right corner, which in the image above is hidden by the Property Inspector. Pressing the icon opens another editor to define whether or not a attribute is update able or query able, which are settings that are picked up by the ADF binding layer and enforced at runtime in the UI developed with it. in addition you can define validation, like required field validation, length, range and pattern validation, which is enforced during the JSF lifecycle when submitting a form. The Data Controls panel shows the SessionFacade as the DataControl and its exposed contents as collectios and methods as shown below



The Data Control displays collections as it does for any other Data Control, including ADF Business Components.

Building an ADF bounded ADF Faces page using the JavaBean Data Control

To create an ADF bound ADF Faces table, select for example the "allDepartments" collection and drag it from the Data Controls panel to the JSF page. In the opened Context menu, choose Table | Read only table. Then, in the table dialog, keep all columns and select the sort and selection option. If you want the table to be filterable, select the filter option so JDeveloper generates the search binding for in memory filtering of the data. Note that the table is create from the getAllDepartments method on the underlying POJO.



Implementing Pagination for the JavaBean Data Control

To implement pagination, changes are required to the JavaBean wrapper and the DataControls.dcx file, which is the registry file that maps the JavaBean Data Control Factory class of the ADF framework to the POJO implementation class. For existing JavaBean applications, there is no need to change the existing ADF binding.

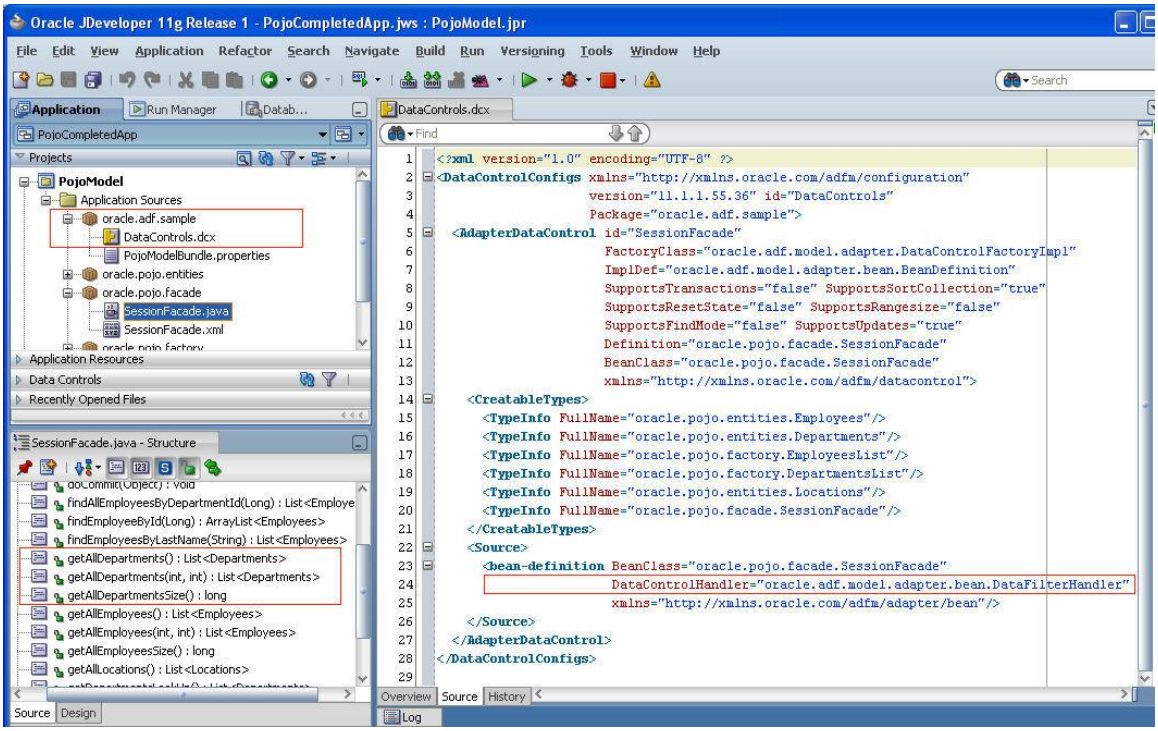
The changes in the JavaBean wrapper class, SessionFacade in the example, are two additional methods that are provided for each collection that is supposed to provide pagination support.

In the example below the getAllDepartments collection is prepared for pagination by adding the getAllDepartments(int,int) and getAllDepartmentsSize() methods. So if your application has a collection queryAlllMyItems, which is based on a method getQueryAllMyItems, then you need to create the following additional methods: queryAlllMyItems(int,int), queryAlllMyItemsSize().

The queryAlllMyItems(int,int) method performs the pagination and is automatically used by ADF when the pagination feature is set up. In my example, the method accesses the data collection held in a List to extract the requested data range identified by the start index and the range size passed in by ADF. In a custom implementation of yours, this method would look up the business service or data source wrappe by the JavaBean wrapper to query data. This, as I mentioned before, however requires some kind of support for incremental data fetching on the other end. The queried data ends up in the ADF iterator managed by the ADF binding container so - e.g. in a table - you can scroll within the data, without a need to re-fetch already queried data.

```
/*
 * QUERY Departments and handle pagination
 */
 public List<Departments> getAllDepartments(){
    return departments.getDepartmentsList();
 }

 //method called by the ADF binding layer to perform paging. ADF
 //passes in the current start range and the range size to query
 //next. The range size is determined by the range size property on
 //the allDepartments iterator in the pageDef.xml file (binding
 //container)
 public List<Departments> getAllDepartments(int index,
                                            int range){
   ArrayList<Departments> rangeList = new ArrayList<Departments>();

  for (int i = index; i < departments.getDepartmentsList().size()
                      && i < index + range;  i++) {
      rangeList.add(departments.getDepartmentsList().get(i));
  }
     return rangeList;
 }

 //method required by the ADF binding layer to perform paging
 public long getAllDepartmentsSize(){
     return departments.getDepartmentsList().size();
 }
```

The same two methods, getAllEmployees(int,int) and getAllEmployeesSize(), are created for the getAllEmployees method in the example you can download at the end of this article.



To configure ADF to use the getAllDepartments(int,int) , you need to edit the DataControls.dcx file as shown below

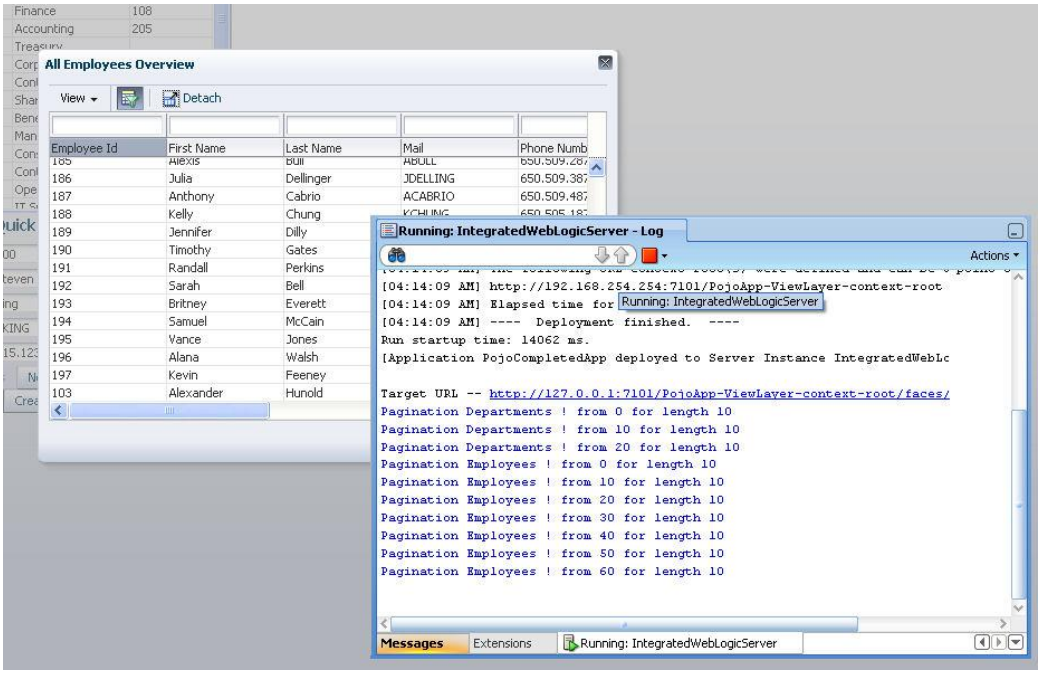
```
 <Source>
      <bean-definition BeanClass="oracle.pojo.facade.SessionFacade"

DataControlHandler="oracle.adf.model.adapter.bean.DataFilterHandler"

xmlns="http://xmlns.oracle.com/adfm/adapter/bean"/>
    </Source>
```

The oracle.adf.model.adapter.bean.DataFilterHandler needs to be configured in the DataControls.dcx file for pagination to work. Note that the DataControls.dcx file is recreated - and thus reset - whenever you re-generate the JavaBean Data Control metadata file. So keep this in mind when re-run generating the POJO DataControl using the "Create Data Control" menu option. If you need to customize the pagination behavior, you can extend the DataFilterHandler class.

Note: Don't forget to set the RangeSize property on the PageDef file to the desired pagination fetch size.

POJO Example for Download and Testing

The JDeveloper 11g R1 PS1 example workspace used in this article is **provided here for download** so you can have a look and play with pagination using the JavaBean Data Control. I added print statement in both, the getAllDepartments(int,int) and the getAllEmployees(int,int) methods so you can see pagination working. Run the "BrowseDepartments.jspx" file to start the application and watch the JDeveloper log window. First you'll see two print messages when the Departments table renders. Scrolling in the table produces more print statements. Press the "Show All Employees" button to launch a dialog with a larger set of data in a table. Again watch the log window when you scroll the table.



Note: Unlike this example, which has all the data to show in a table hard coded in the bean model, the <method>(int,int) method in a real life implementation should access a method or functionality on the target system that supports paging. Only if the target system has a mean to support pagination you really avoid querying all records on the initial page load. For example, to use paging with Web Services accessed from a WS proxy client, the WS must expose a method that allows you to pass in the start and length of a query.

Note: The sample you can download from ADF Code Corner

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

provides more functionality than the pagination. It also allows you to explore CRUD operations using the JavaBean Data Control, as well as table filtering in the Employees table. Have a play with it and explore the great advantage POJO developer gain from using ADF Data Controls and bindings. Note that most of the partial page refreshes use the ChangeEventProperty set to "ppr" on the binding container (pageDef.xml).

RELATED DOCOMENTATION

⊠	
⊠	
⊠	