ADF Code Corner

051. How-to scroll ADF tables using an alphabetic index menu

twitter.com/adfcodecorner

Abstract:

In this example, an alphabet index anchor menu is used to scroll an ordered ADF bound ADF Faces table. The example provided for download is based on the HR Employees table. Selecting an entry from the index menu, for example the 'F' character, scrolls the table to the first employee record that has this character as the first character of the last name attribute value. The implementation dynamically builds the index list, ensuring that characters that are not available as leading characters in the last name attribute are rendered but disabled for selection.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
06-JAN-2010

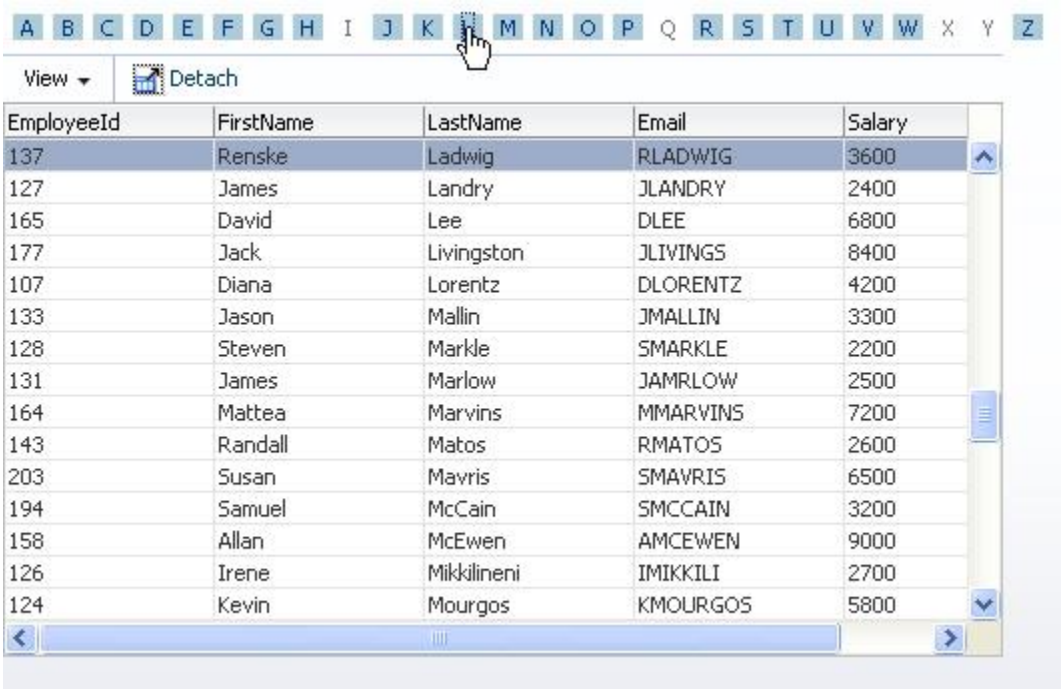
Oracle ADF Code Corner is a loose blog-style series of how-to documents that provide solutions to real world coding problems.

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 image below shows the employees table with an associated menu of index characters. Selecting a link in the menu scrolls the table to the first record that has its last name column value starting with the clicked character. The index menu itself is dynamically built using a View Object Impl class in ADF Business Components and allows you to easily customize this solution to your needs, like inlcuding non alphabetic characters and numbers. As shown in the image below, only those characters are enabled that have at least one occurrence in the row set of the View Object used to provide the table data. All characters that have a white background are read only and don't scroll the table.



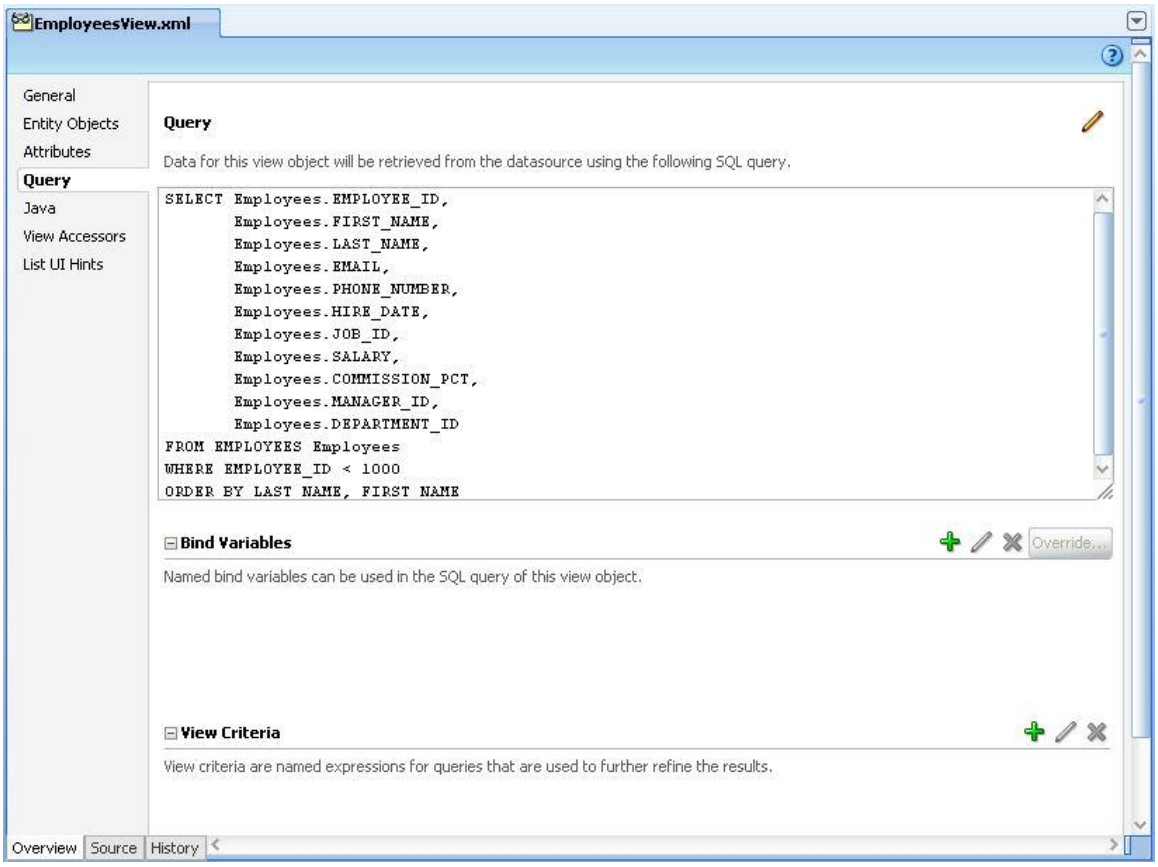
EmployeeId	FirstName	LastName	Email	Salary
137	Renske	Ladwig	RLADWIG	3600
127	James	Landry	JLANDRY	2400
165	David	Lee	DLEE	6800
177	Jack	Livingston	JLIVINGS	8400
107	Diana	Lorentz	DLORENTZ	4200
133	Jason	Mallin	JMALLIN	3300
128	Steven	Markle	SMARKLE	2200
131	James	Marlow	JAMRLOW	2500
164	Mattea	Marvins	MMARVINS	7200
143	Randall	Matos	RMATOS	2600
203	Susan	Mavris	SMAVRIS	6500
194	Samuel	McCain	SMCCAIN	3200
158	Allan	McEwen	AMCEWEN	9000
126	Irene	Mikkilineni	IMIKKILI	2700
124	Kevin	Mourgos	KMOURGOS	5800

The implementation of this use case uses the View Object Impl class to define a public method that is exposed to ADF as a client method. The exposed method queries the View Object and

creates a list of alphabet characters with information about the row key of their first occurrence and whether or not there is an occurrence at all.

ADF Business Components

The View Object query in the example is ordered by the last name column and limited to employees with an EmployeeId below 1000 (just in case your Employees table contains random testdata as mine has). The index search is for the last name column of the Employees table. However, studying the source code used, it is easy to determine how to change the search to another column.



A helper class we use in the implementation is IndexCharacterObject. The IndexCharacterObject class contains information about the availability of a alphabet character in the current View Object row set, the row key of its first occurrence, as well as the character itself.

```
public class IndexCharacterObject {
 String character;
 Boolean found;
 Key rowIndex;

 public IndexCharacterObject() {
   super();
 }

 public void setCharacter(String character) {
```

```
    this.character = character;
  }

  public String getCharacter() {
    return character;
  }

  public void setFound(Boolean found) {
    this.found = found;
  }

  public Boolean getFound() {
    return found;
  }

  public void setRowIndex(Key rowIndex) {
    this.rowIndex = rowIndex;
  }

  public Key getRowIndex() {
    return rowIndex;
  }
}
```

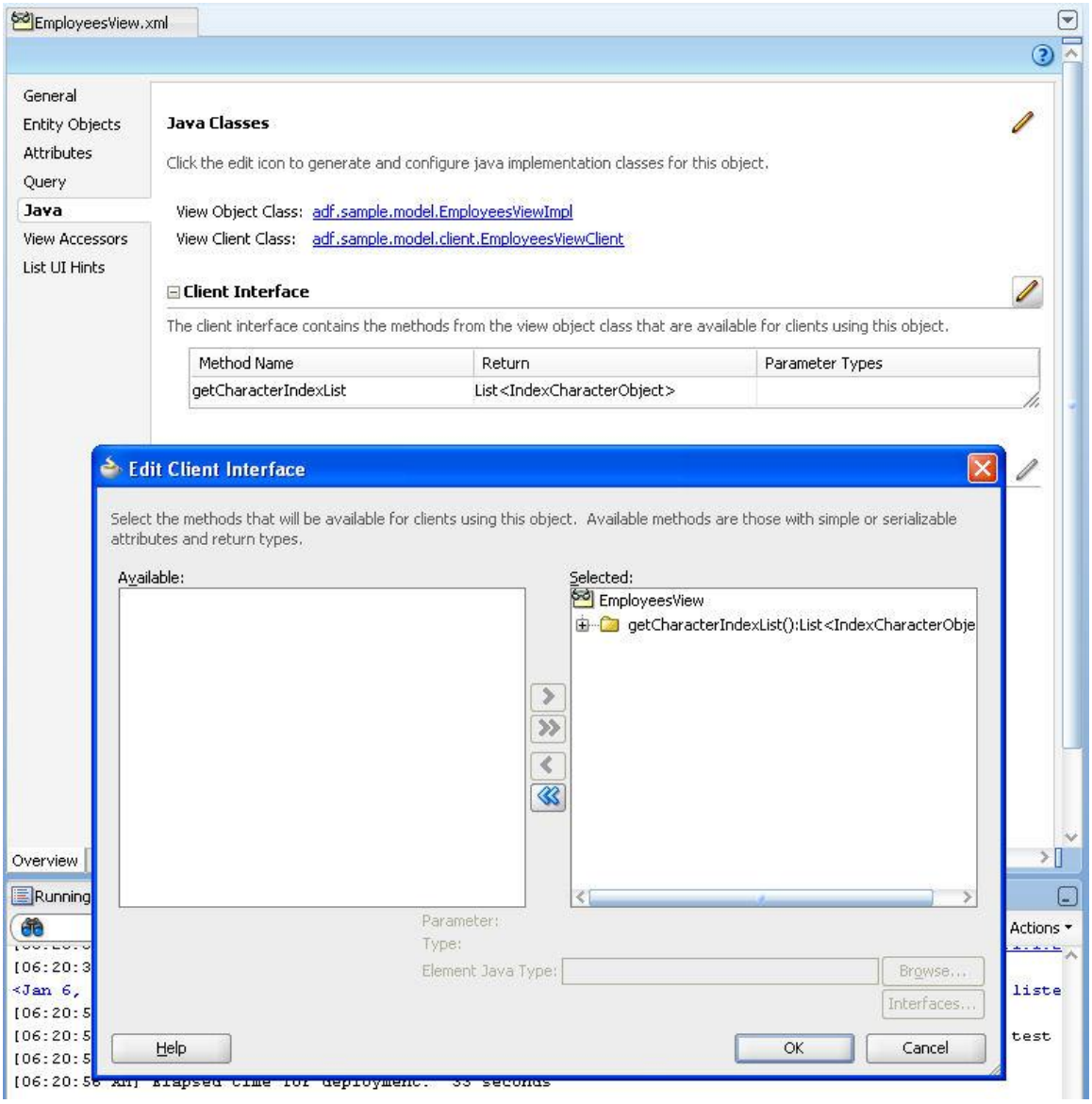
The source code below shows the EmployeesViewImpl class that we created declaratively for the View Object. The method we added is getCharacterIndexList, which returns a list of the 26 characters in the alphabet. In the beginning of this method, the alphabet is defined. This also is the place for you to customize the index list, for example to extend it to use non Latin characters or numbers.

The expensive operation in this method, which also is why we recommend to not refresh the index with each request, is to browse over all rows in the result set to retrieve the row keys of the unique characters. Once this is done in a HashMap, the map is compared with the alphabet defined at the top. This allows us to determine which characters should be rendered as links in the menu and which characters should be rendered read only.

```
public class EmployeesViewImpl extends ViewObjectImpl implements
EmployeesView {
  /**
  * This is the default constructor (do not remove).
  */
  public EmployeesViewImpl() {
  }

  public List<IndexCharacterObject> getCharacterIndexList() {
    String[] alphabet =
    { "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
      "N","O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z" };

    //list that contains 26 entries
    List<IndexCharacterObject> list =
                                new ArrayList<IndexCharacterObject>();

    //a list of all initial characters that are found in the row set
    //last_name attribute
```

```
    HashMap map = new HashMap();
    RowSet rs = this.getRowSet();

    this.executeQuery();

    Row row = rs.first();

    //query all rows and memorize the initial characters of
    //all employees last name entries
    if (row != null) {
      //Optionally, change "LastName" to another column attribute name
      //to change the search attribute
      String lastname = (String)row.getAttribute("LastName");
      String character = lastname.toUpperCase().substring(0,1);
      updateMap(map, row, character);
      while (rs.hasNext()) {
        row = rs.next();
        character = ((String)
        row.getAttribute("LastName")).toUpperCase().substring(0,1);
        updateMap(map, row, character);
     }
   }

  for (int i = 0; i < alphabet.length; ++i) {
    if (map.containsKey(alphabet[i].toUpperCase())) {
      IndexCharacterObject ico = new IndexCharacterObject();
      ico.setCharacter(alphabet[i].toUpperCase());
      ico.setFound(true);
      //get rowIndex from map
      ico.setRowIndex(((Key)map.get((alphabet[i].toUpperCase()))));
      list.add(i, ico);
    } else {
     IndexCharacterObject ico = new IndexCharacterObject();
     ico.setCharacter(alphabet[i].toUpperCase());
     ico.setFound(false);
     //get rowIndex from map
     ico.setRowIndex(null);
     list.add(i, ico);
    }
    rs.first();
  }
  return list;
}

private void updateMap(HashMap map, Row row, String character) {
  if (!map.containsKey(character)) {
    //remember character and first occurrence in rowSet
    map.put(character, row.getKey());
   }
  }
}
```

Using the Java option of the EmployeesView editor, the public method is exposed as a client method to ADF. This is an important step as otherwise the method is not available in the Data Control palette. You may ask why we expose the method on the View Object level and not the Application Module level. Well, both would be possible. However, exposing a method where it is defines a cleaner contract between the business service developer and the web application developer. It is less error prone as well.

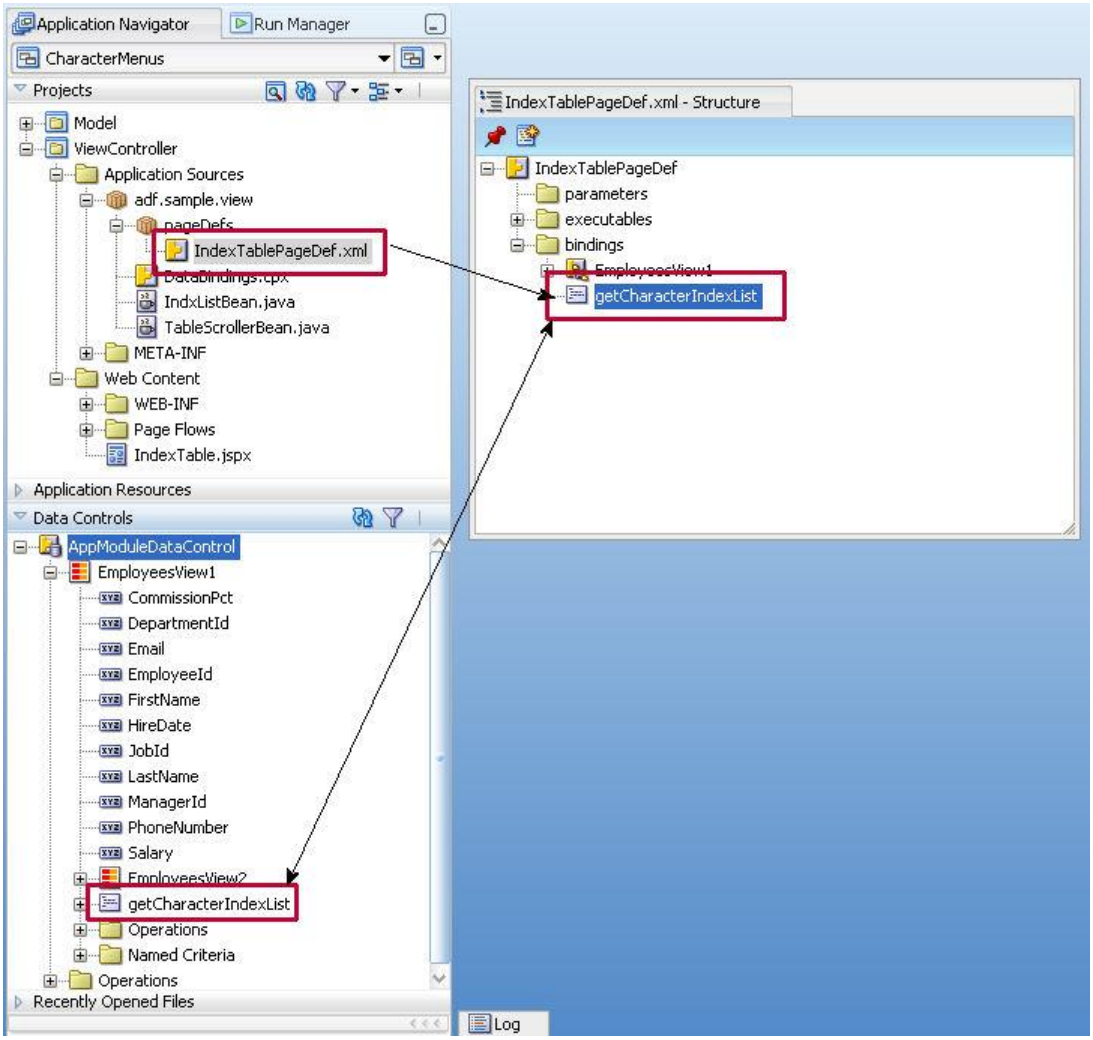


This is all that we need to do on the ADF Business Service layer. You can run the ADF Business Component tester and test the method if you like.

Preparing the ADF Binding Layer

The getCharacterIndexList method of the View Object Impl class is accessed from a manged bean that prepares the list for display on the ADF Faces page. We want to access the method through the ADF binding layer to leverage the consistent API that the ADF layer provides to web application developers. As you can see in the image below, the method is exposed in the Data Controls palette as an entry in the EmployeesView1 node (actually this method is listed as an entry of all View Object instances of the EmployeesView). To create a method binding, select the PageDef file of the ADF Faces page or page fragment you want to create the index on and choose Bindings | Generic Binding | Method . In the

opened dialog, select the EmployeesView1 instance and choose the getCharacterIndexList method. Note that you could use a dependent instance of the EmployeesView as well. However, in this case the index will be created for a result set filtered by the selected View Object parent. In our example, we want to create the index for all employees, thus we use the EmployeesView1 instance.



In the image above, the PageDef file contains a tree binding "EmployeesView1", which is referenced by the ADF Faces table to read the employees data. This binding is created implicitly when dragging and dropping the EmployeesView1 View Object as a read only table onto the page.

View Layer Implementation

The view layer implementation is with ADF Faces RC and includes the dynamic anchor index as well as a managed bean method to perform the table scrolling

Building the Index Menu

The index menu is built using a managed bean in view scope that accesses the ADF binding layer to read the index entries. The view scope is chosen to avoid renewing the index for each request, which would be pretty expensive in a rich Internet application usecase. To refresh the index, developers could set the list

variable that holds the index to null and then PPR the menu, in which case the list is renewed by querying the ADF Business Components model.

The index menu is dynamically built so that we can delegate the hard work of determining the indexes that should be enabled and those that should be read only to the ADF Business Component model. In addition, its easier to have the business service returning the row keys for each of the first time occurences of the characters in the table than to get this information after the fact onthe view Layer (which also would lead to bad performance).

As shown in the image below, the menu is created using an af:forEach component that references the list entry in the IndxListBean managed bean. The af:forEach component defines a variable called "list" which iterates over each of the 26 list entries, referencing the IndexCharacterObject. Just a reminder: the IndexCharacterObject contains information about the character, the row key of the first occurrence and whether or not it should be rendered as an active link.

```
<af:forEach  var="list" items="#{viewScope.IndxListBean.list}">
```

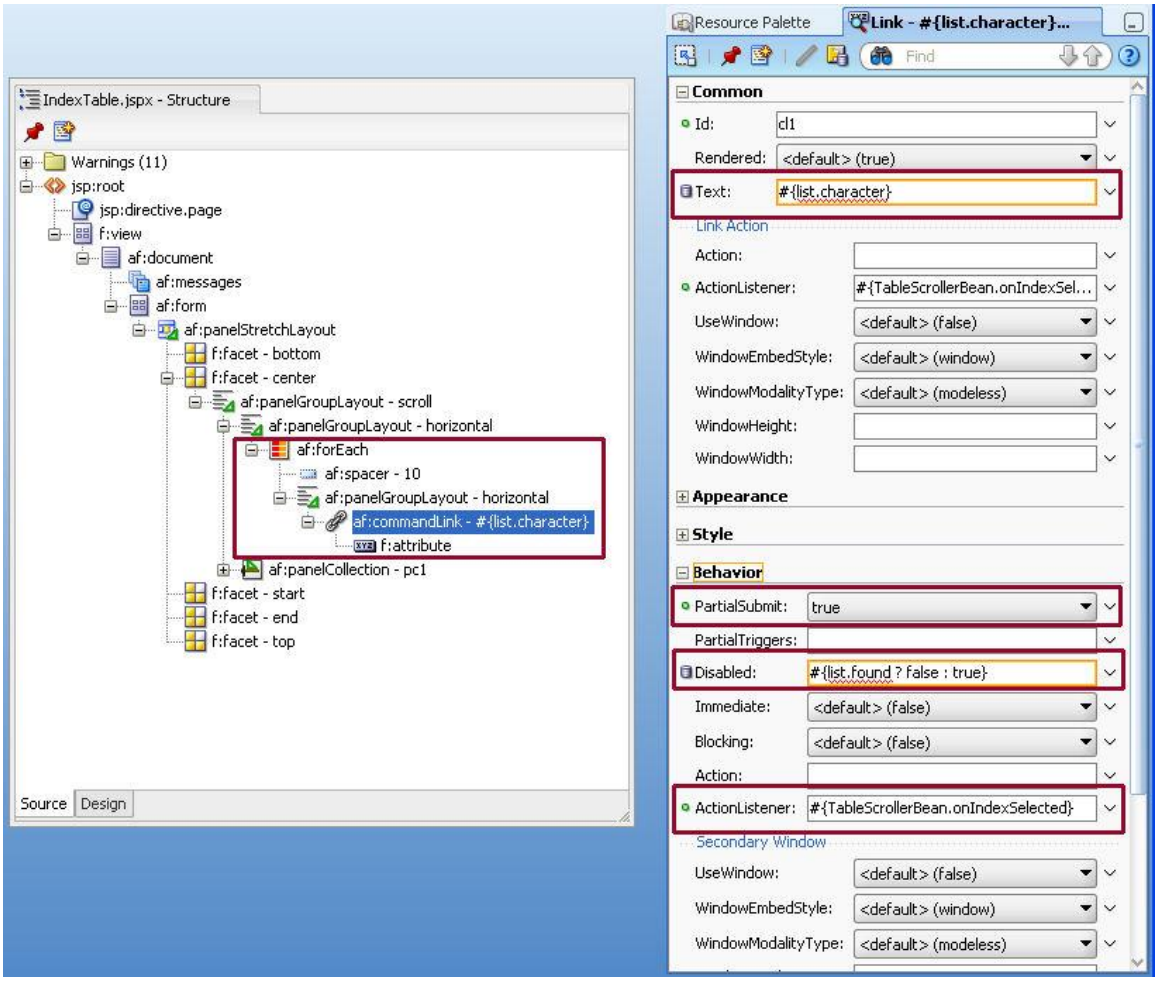
The af:forEach component contains an af:panelGroupLayout component which builds the container for the command links used by the menu. The af:panelGroupLayout uses its inlineStyle property to render the background color and size of the links.

```
<af:panelGroupLayout id="pgl2" layout="horizontal"
    halign="center"
    inlineStyle="#{list.found ? 'background-color:#aeccd8;' :
                                'background-color:white;'} text-align:center; width:15px; ">
```

As you can see, the inlineStyle property references the "found" method of the IndexCharacterObject object referenced by the "list" variable. This way we set the blueish color for the active links and the white background for the inactive links.

The af:panelGroupLayout component contains an af:commandLink that also references the "list" variable in its "Text" property and the "Disabled" property. In addition, the "PartialSubmit" is set to true so that the page does not refresh when the link is pressed.

The command link reference a second managed bean, TableScrollerBean, from its ActionListener property. This bean is called when the user clicks on a link.



The IndxLstBean shown next is referenced from the af:forEach component and returns a list of IndexCharacterObjets. The getList method refreshes the index only if the list variable is null or if it has a zero length. This way we avoid double invocation within the JavaServer Faces request life cycle.

```
/*
 * BEAN is in viewscope, thus ensuring that the index query is executed
 * only once. Changing the bean scope to request refreshes the list for
 * each request but is expensive and should be avoided
 */
public class IndxListBean {

 List<IndexCharacterObject> list = null;

 public IndxListBean() {
  super();
 }

 public void setList(List<IndexCharacterObject> list) {
  this.list = list;
 }

 public List<IndexCharacterObject> getList() {
   if (list == null || list.size() == 0) {
```

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

      OperationBinding createIndexList =
          (OperationBinding)bindings.get("getCharacterIndexList");
      list = (List<IndexCharacterObject>)createIndexList.execute();

      if (createIndexList.getErrors().size() != 0) {
        //create empty list in case of error
        list = new ArrayList<IndexCharacterObject>();
      }
    }
    return list;
  }
}
```

Scrolling the Table

The TableScrollerBean is referenced from the ActionListener property of the command link component. It reads the rowKey of the table row to scroll to from the custom attribute "jboRowKey", which we created for the command link.

```
<af:commandLink text="#{list.character}" id="cl1"
    partialSubmit="true"
    disabled="#{list.found ? false : true}"
    actionListener="#{TableScrollerBean.onIndexSelected}">
    <f:attribute name="jboRowKey" value="#{list.rowIndex}"/>
</af:commandLink>
```

The jboRowKey custom attribute contains the value of the rowIndex property of the IndexCharacterObject object referenced by the "list" variable.

```
public class TableScrollerBean {
 private RichTable table1;

 public TableScrollerBean() {
   super();
 }

 public void onIndexSelected(ActionEvent actionEvent) {

   RichCommandLink linkPressed =
                  (RichCommandLink) actionEvent.getSource();
   Key jboRowKey = (Key)linkPressed.getAttributes().get("jboRowKey");

   BindingContext bctx = BindingContext.getCurrent();
   BindingContainer bindings = bctx.getCurrentBindingsEntry();
   DCIteratorBinding employeesIterator =
           (DCIteratorBinding) bindings.get("EmployeesView1Iterator");
   employeesIterator.setCurrentRowWithKey(
                                    jboRowKey.toStringFormat(true));
   ArrayList tableRowKey = new ArrayList();
   tableRowKey.add(jboRowKey);
   RowKeySetImpl rks = new RowKeySetImpl();
   rks.add(tableRowKey);
   table1.setSelectedRowKeys(rks);
```

```
    //partially refresh the table to display the new selection
    AdfFacesContext.getCurrentInstance().addPartialTarget(table1);
  }

  public void setTable1(RichTable table1) {
    this.table1 = table1;
  }

  public RichTable getTable1() {
    return table1;
  }
}
```

Note that the table must have its "DisplayRow" property set to "selected" so that the displayed row is the selected row after refreshing the table. The table has a JSF component binding defined that points to the same managed bean so that a partial refresh can be called within the Java code.

```
<af:table value="#{bindings.EmployeesView1.collectionModel}"
  var="row" rows="#{bindings.EmployeesView1.rangeSize}"
  ...
  selectedRowKeys="#{bindings.EmployeesView1.collectionModel.selectedRow}"
  selectionListener="#{bindings.EmployeesView1.collectionModel.makeCurrent}"
  rowSelection="single" id="t1"
  displayRow="selected"
  binding="#{TableScrollerBean.table1}">
```

Download Oracle JDeveloper 11g R1 PS1 Sample workspace

The sample workspace introduced in this blog entry can be downloaded **ADF Code Corner**

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

To run this sample, change the database configuration to point to a database of yours that has the HR schema installed and enabled.

RELATED DOCOMENTATION

☒	
☒	
☒	