ADF Code Corner

84. Dynamically show or hide af:treeTable columns dependent on the disclosed node

twitter.com/adfcodecorner

Abstract:

The use case in this article is to reduce the number of columns in a treeTable to only show data that belongs to the disclosed treeTable node and its ancestor nodes. The assumption made in the sample is that the treeTable only discloses a single node at a time to simplify the detection of which columns should be shown and which should be hidden from the view. While theuse case covered is less common, the implementation of it may be interesting for many developers who need to dynamically work with the tree or treeTable component.

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

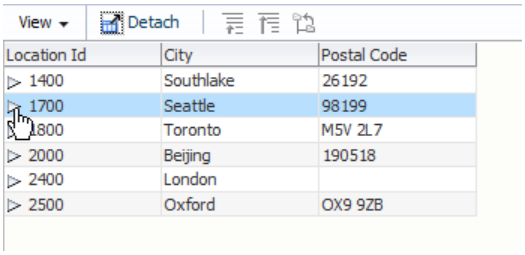
Oracle ADF Code Corner is a loose blog-style series of how-to documents that provide solutions to real world coding problems.

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.

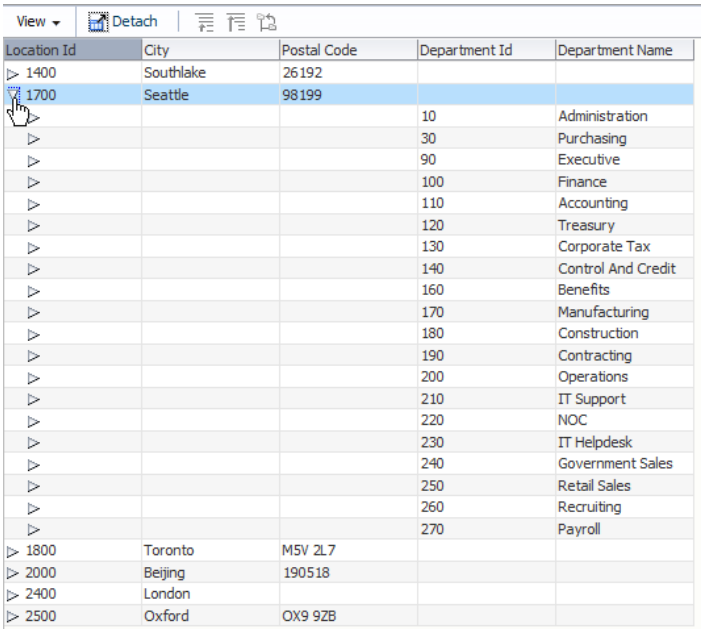
Please post questions or report problems related to the samples in this series on the OTN forum for Oracle JDeveloper: http://forums.oracle.com/forums/forum.jspa?forumID=83

Introduction

The image below shows the initially rendered treeTable component state, which is to only show information of the Locations view. The full hierarchical structure, which is read from the Oracle HR database schema, includes the Locations view, Departments view and Employees view. As the discloses a Location node to show its dependent detail information …

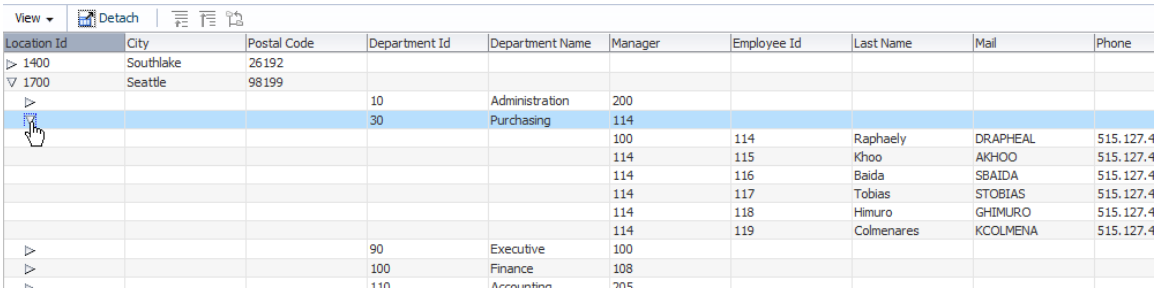


… the treeTable columns that present the detail information are added to the view in a partial refresh of the treeTable component.



Location Id	City	Postal Code	Department Id	Department Name
▷ 1400	Southlake	26192		
▽ 1700	Seattle	98199		
▷			10	Administration
▷			30	Purchasing
▷			90	Executive
▷			100	Finance
▷			110	Accounting
▷			120	Treasury
▷			130	Corporate Tax
▷			140	Control And Credit
▷			160	Benefits
▷			170	Manufacturing
▷			180	Construction
▷			190	Contracting
▷			200	Operations
▷			210	IT Support
▷			220	NOC
▷			230	IT Helpdesk
▷			240	Government Sales
▷			250	Retail Sales
▷			260	Recruiting
▷			270	Payroll
▷ 1800	Toronto	M5V 2L7		
▷ 2000	Beijing	190518		
▷ 2400	London			
▷ 2500	Oxford	OX9 9ZB		

Disclosing a detail node in the dependent Departments further expands the treeTable component with additional columns for the Employees view.



Location Id	City	Postal Code	Department Id	Department Name	Manager	Employee Id	Last Name	Mail	Phone
▷ 1400	Southlake	26192							
▽ 1700	Seattle	98199							
▷			10	Administration	200				
			30	Purchasing	114				
					100	114	Raphaely	DRAPHEAL	515.127.4
					114	115	Khoo	AKHOO	515.127.4
					114	116	Baida	SBAIDA	515.127.4
					114	117	Tobias	STOBIAS	515.127.4
					114	118	Himuro	GHIMURO	515.127.4
					114	119	Colmenares	KCOLMENA	515.127.4
▷			90	Executive	100				
▷			100	Finance	108				
▷			110	Accounting	205				

This article explains how this solution works and also provides the sample as a download on ADF Code Corner.

Note: To get more code sample for ADF bound ADF Faces tree and treeTable components, visit ADF Code Corner at http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html and search for "tree".

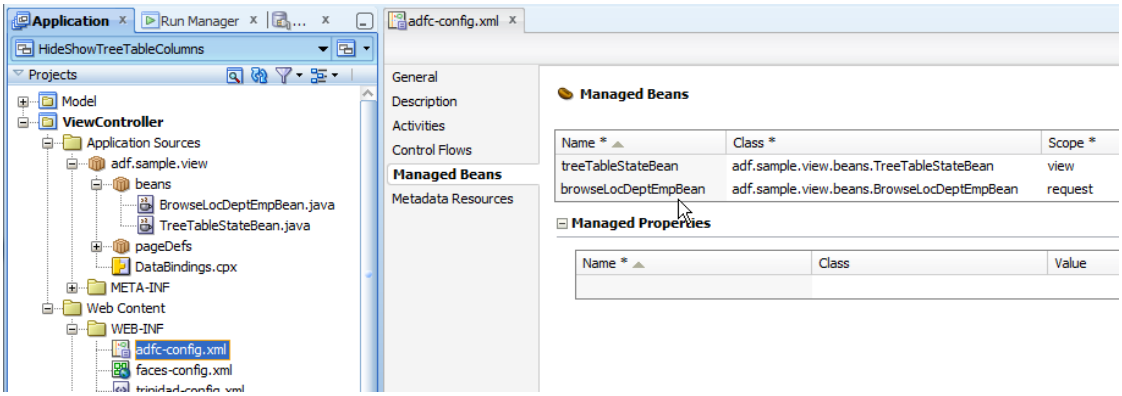
How-to implement this solution

The sample uses two managed beans

- The **treeTableStateBean** extends `java.util.HashMap` and is configured to exist in the ADF Controller view scope so it survives client-server requests. The treeTableStateBean keeps

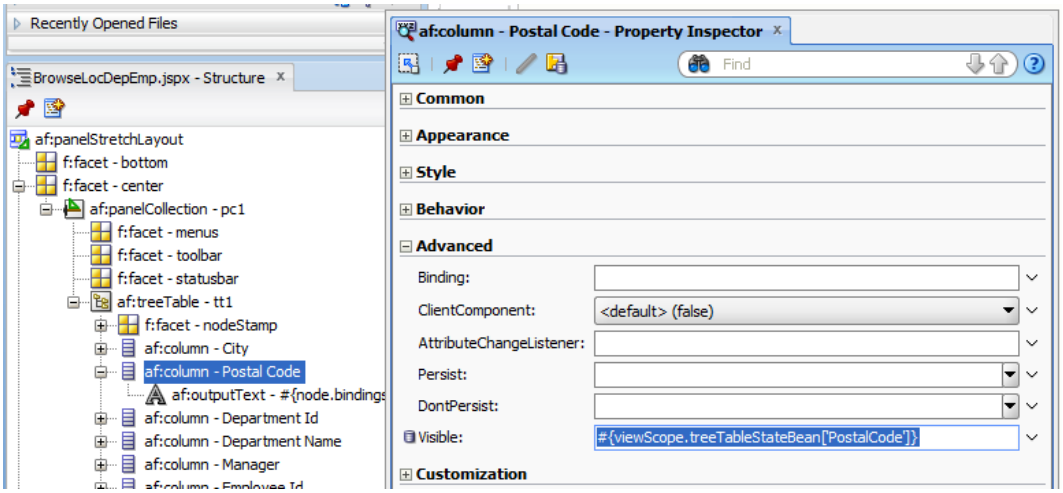
track of the visibility of the treeTable columns for the duration that the page containing the treeTable is shown.

- The **browseLocDeptEmpBean** accesses the **treeTableStateBean** from a custom disclosure listener to change the visible state of the columns according to the needs of the disclosed tree node.



The managed beans are configured in the adfc-config.xml file but need to be configured in another XML configuration file if used within a bounded task flow.

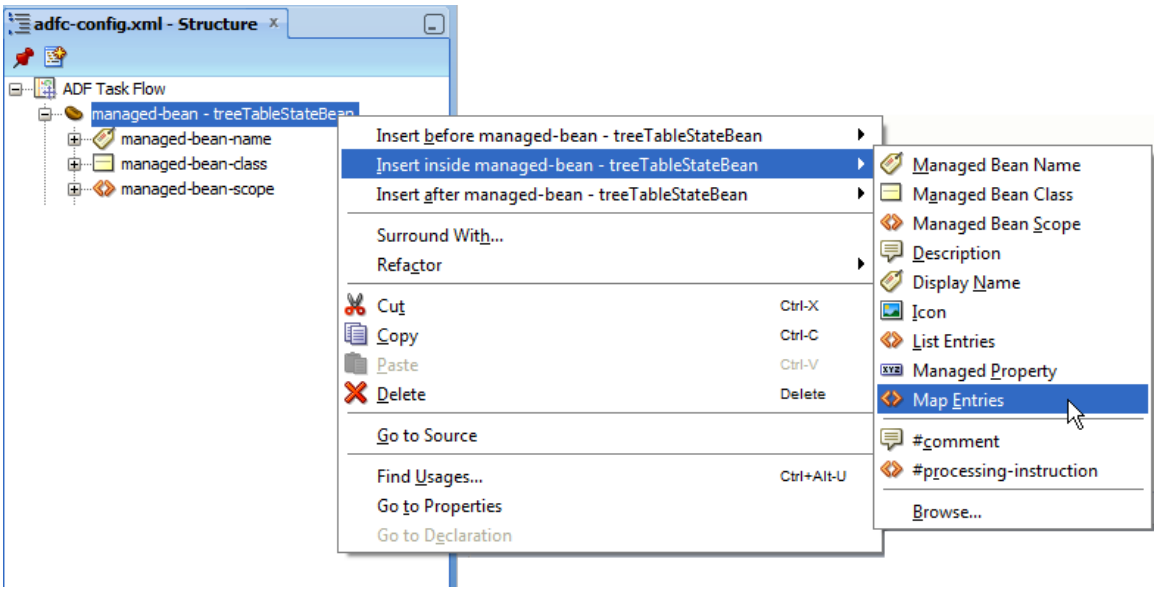
The treeTable component is changed from its original layout (which is to have a single node defined in the nodeStamp facet) created by ADF when dragging the Locations view collection as a treeTable to a page. `af:column` and `af:outputText` components are added as children of the `af:treeTable` component to display the node attributes in their own column.



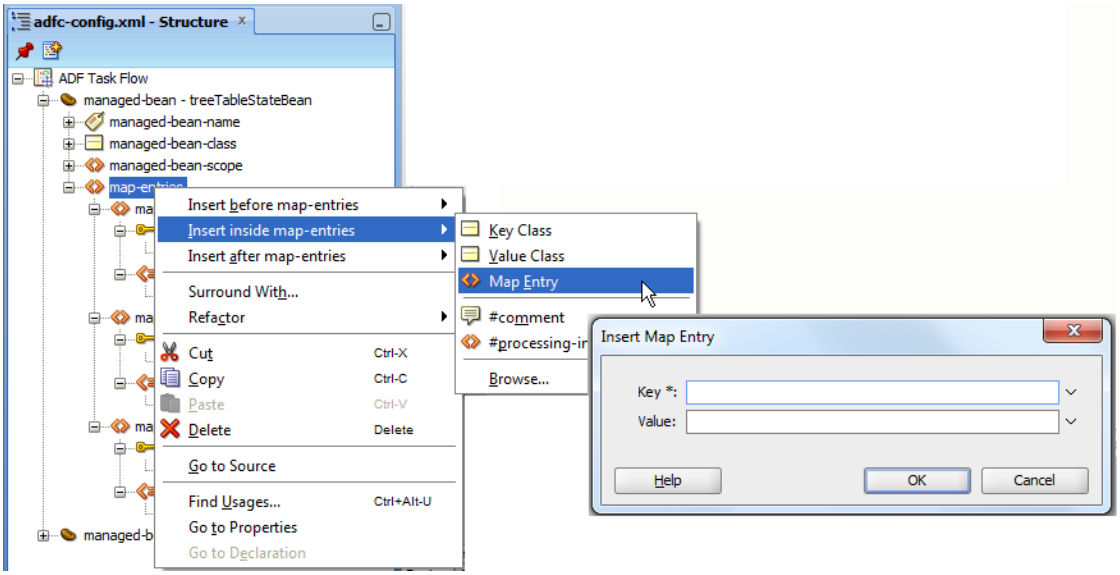
As shown in the image above, each `af:column` references the **treeTableStateBean** in its *Visible* property. As mentioned, the treeTableStateBean extends `HashMap` and because of this exposes a *getter* method and a *put* method that you can access from EL using the ['key name'} argument. In the sample, each column passes the argument of the attribute name it represents (for example "PostalCode", which is an attribute of the Locations view in the ADF Business Component model of the sample.

To define default settings for the initial treeTable views, which in the case of this sample shows all attributes that belong to the Locations view, you use a **Map Entries** configuration on the managed bean.

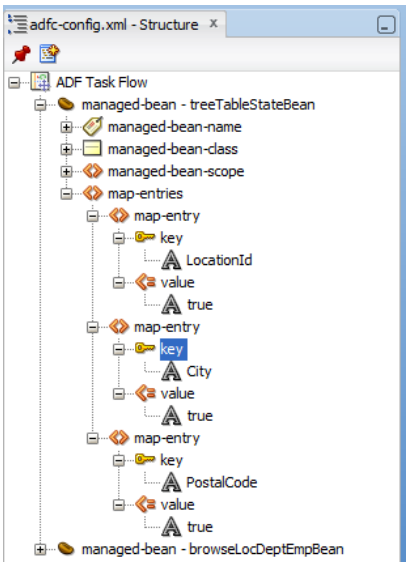
Managed beans that extend `HashMap` use map entries to define key / value pairs of type String. The Strutcure Window in Oracle JDeveloper helps you to configure this.



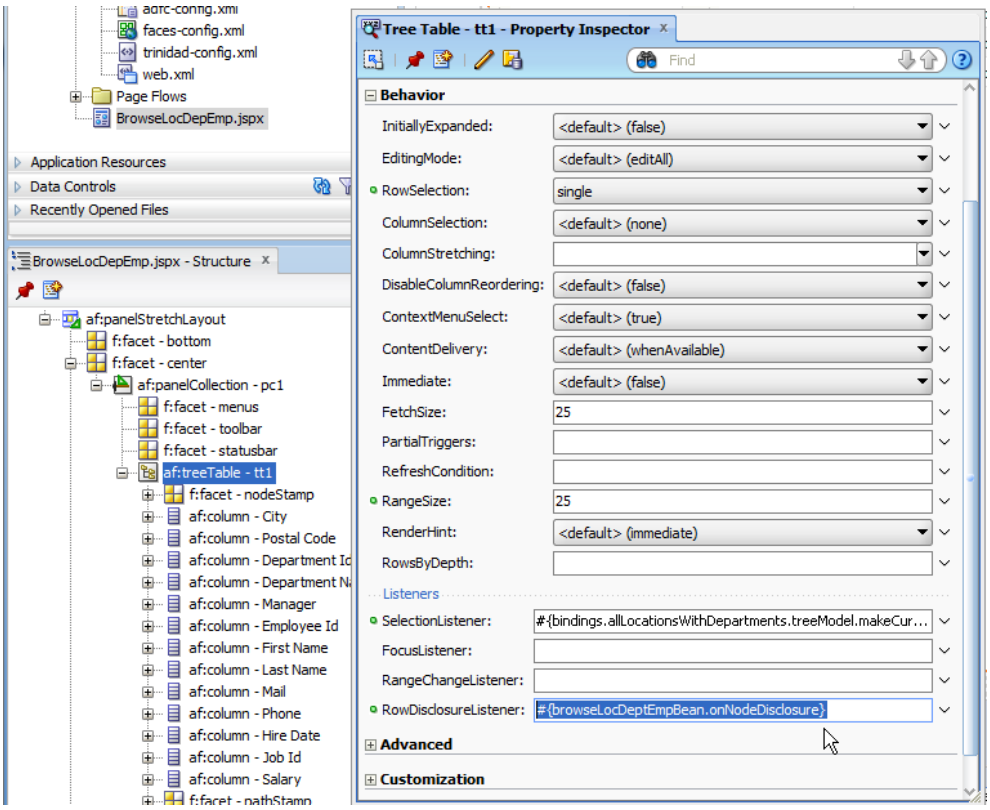
The *Key* entry is the name of an attribute in the underlying model (LocationId, or City). The value then is set to **true** for attributes that should display initially and false for those that should not show. To reduce the work developers have with setting these defaults, all keys that are not configured are assumed to have a visibility of **false**, so that only those attributes need to be configured in the Map Entries that should have a setting of **true**. For this, the managed bean overrides the **get** method of the `HashMap` it extends.



The sample **Map Entries** (shown in the image below) has only three attributes defined to show all attributes of the Location view.



The next step is to create a **RowDisclosureListener** for the treeTable in its own managed bean. The managed bean handling the disclosed listener does not need to be in view scope as there is no need for it to keep track of any state because this is what the **treeTableStateBean** is for.



With only two configurations, the whole use case is implemented. The row disclosure listener notifies a method in the **browseLocDeptEmpBean** about the disclosed node. The method then determines the attributes that belong to the disclosed node level and adds an entry in the **treeTableStateBean** to change

the visible state from false to true. Upon a partial refresh of the treeTable (which is through a PPR on the parent container) the columns re-read their visible state from the `HashMap` and display their content.

Note: Because the treeTable requires the partial refresh to be performed on the parent component (or the parent's parent component) to re-render accurately, you should also be able to use the *Rendered* property, which then has a smaller footprint in the downloaded page as no hidden markup would be generated. However, for this sample we tried and tested the *Visible* property only.

TreeTableStateBean Managed Bean Code

There are two managed beans used in this sample. The simple bean is to hold the treeTable display state for the columns. It extends `java.utilHashMap` and overrides the get method to translate string values into Boolean values and to set the default visible state to **false**.

```
import java.util.HashMap;

public class TreeTableStateBean extends HashMap{
    private static final long serialVersionUID = 0L;

    public TreeTableStateBean() {
        super();
    }
  /**
   * The HashMap returns TRUE / FALSE for the keys passed in. The bean
   * is used as a managed bean and referenced from the TreeTable column
   * display property. It allows to dynamically show/hide a tree table
   * columns based on the node selection.
   * @param key
   * @return true / false
   */
   @Override
   public Boolean get(Object key) {
     if(super.get(key)==null){
     //set the default visible state to false for all unregistered
     //attributes
     return Boolean.FALSE;
   }
   else{
     try {
      String colStateString = (String) super.get(key);
      Boolean colState = Boolean.parseBoolean(colStateString);
      return colState;
     } catch (Exception e) {
       //cannot parse value String, thus return false
       return Boolean.FALSE;
   }
  }
```

```
  }
}
```

BrowseLocDeptEmpBean Managed Bean Code

This managed bean handles the disclosure state switching and contains a lot of interesting ADF coding. It accesses the `TreeTableStateBean` to read and write the treeTable column visible states.

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.ValueExpression;
import javax.faces.component.UIComponent;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;
import oracle.adf.view.rich.component.rich.data.RichTreeTable;
import oracle.adf.view.rich.context.AdfFacesContext;

import oracle.jbo.uicli.binding.JUCtrlHierBinding;
import oracle.jbo.uicli.binding.JUCtrlHierNodeBinding;
import oracle.jbo.uicli.binding.JUCtrlHierTypeBinding;

import org.apache.myfaces.trinidad.event.RowDisclosureEvent;
import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.RowKeySet;


public class BrowseLocDeptEmpBean {

 //HashMap that holds the column names of each tree level in a string
 //array. The Map key is the StructureDefName of the node, which is the
 //class name and package name of the object representing the node

 private HashMap nodeAttributes = null;

 public BrowseLocDeptEmpBean() { }

 //listen to the tree table disclosure event
 public void onNodeDisclosure(RowDisclosureEvent rowDisclosureEvent) {

   String nodeDefName = null;
   JUCtrlHierNodeBinding disclosedNode = null;
   //the typBinding is the ADF hierarchical tree binding node
   //structure. We use this information to determine, which node's
   //columns must be shown and which node's columns must be hidden
   JUCtrlHierTypeBinding typeBinding[] = null;

   //the handling of a node close event is different the disclosure of
   //a node. If the event's addedSet is empty then we know a node is
```

```
  //closed. If the addedSet has content, then a node has been discloed
  boolean isCloseEvent = false;

  //get access to the RichTreeTable instance. To keep this code
  //generic, we use the event object as a starting point to the tree
  //table component and the ADF hierarchical tree binding

  RichTreeTable treeTable =
                (RichTreeTable)rowDisclosureEvent.getSource();
  CollectionModel model = (CollectionModel)treeTable.getValue();
  JUCtrlHierBinding treeBinding =
                (JUCtrlHierBinding)model.getWrappedData();


 //query a list of tree nodes and node attributes. Do this only once
 //per managed bean instantiation (viewScope) as it is assumed that
 //the tree structure does not change at runtime.

if(nodeAttributes == null){
   nodeAttributes = new HashMap();
   typeBinding = treeBinding.getTypeBindings();
   //get the attributes for each node level. Note that this also
   //find node attribute that are not displayed in the tree table.
   //However, the way this sample implements the show/hide function
   //does not require to check for a node's attribute to be visible
   //in the tree as there is no risk for a NPE
   for (int i = 0; i < typeBinding.length; i++) {
     String[] attributeNames = typeBinding[i].getAttrNames();
     String nodeObjectTypeName = typeBinding[i].getStructureDefName();
            nodeAttributes.put(nodeObjectTypeName, attributeNames);
    }
  }

  //get the disclosed node
  RowKeySet rowKeySet = rowDisclosureEvent.getAddedSet();
  //did disclosure event open a new node ?
  if(rowKeySet.iterator().hasNext()){
     isCloseEvent = false;
  }
  else{
     isCloseEvent = true;
     //get the previously disclosed set
     rowKeySet = rowDisclosureEvent.getRemovedSet();
  }


     Iterator iterator = rowKeySet.iterator();
```

```
if (iterator.hasNext()) {
   List rowKey = (List)iterator.next();
   disclosedNode = treeBinding.findNodeByKeyPath(rowKey);
   //determine the node the user selected. We determine this by
   //the Object that represents the node, which is the full
   //package and class name of the View Object
  JUCtrlHierTypeBinding nodeType =
                       disclosedNode.getHierTypeBinding();
  nodeDefName =  nodeType.getStructureDefName();
}

//the logic in the following is as follows:
//1.  We need to show all the columns of the children that belong
//    to the disclosed node. If we disclose Locations, then we
//    need to show the columns for Departments.
// 2. So what we do is that we check for a match of the disclosed
//    node's StructureDef name with a StructureNodeDefName
//    found in the ADF HierarchicalTree. If we find a match then
//    we know that the immediate child columns must be shown and
//    all the other child nodes below must be hidden
// 3. We use "matchingNodeFound" and "directChildNodeHandled" to
//    determine if the disclosed node was found and if the
//    immediate child was handled (displayed its columns)
// 4. The code updates the managed bean that holds teh visible
//    state of the tree table columns and then refresh the parent
//    of the tree table
boolean matchingNodeFound = false;
boolean directChildNodeHandled = false;
  for (int i = 0; i < typeBinding.length; i++) {
    if(matchingNodeFound == true
       && directChildNodeHandled == false){
      String childNodeToShowHide =
                    typeBinding[i].getStructureDefName();
      //get node column attributes from binding layer
      String[] columnAttributes =
              (String[]) nodeAttributes.get(childNodeToShowHide);
      for (int colIndex = 0; colIndex < columnAttributes.length;
           colIndex++) {
        //set a new visible state for the column
        this.getTreeTableStateObject().
         put(columnAttributes[colIndex],
             isCloseEvent == true ? "false" : "true");
```

```
            }
            directChildNodeHandled = true;
          }
           else if(matchingNodeFound == true &&
                   directChildNodeHandled == true){
           String childNodeToShowHide =
                       typeBinding[i].getStructureDefName();
           //get node column attributes from binding layer
           String[] columnAttributes =
               (String[]) nodeAttributes.get(childNodeToShowHide);
           for (int colIndex = 0; colIndex < columnAttributes.length;
                colIndex++) {
              this.getTreeTableStateObject().
                  put(columnAttributes[colIndex], "false");
           }
          }

          //ensure to first search for a matching tree node definition
          if (nodeDefName.equalsIgnoreCase
                              (typeBinding[i].getStructureDefName())){
            //ensure close and disclose is applied to the child items
            //of the disclosed node
            matchingNodeFound = true;
           }

           //make sure only the selected row key set is disclosed and
           //the rest is closed. This use case will become quite
           //complex if we allow multiple disclosed nodes. So for this
           //level, we have a parent node to find

           if (isCloseEvent == false &&
               disclosedNode.getParent() != null &&
               !disclosedNode.getParent().getKeyPath().isEmpty()){
                rowKeySet.add(disclosedNode.getParent().getKeyPath());
            }

            if (isCloseEvent == false) {
                treeTable.setDisclosedRowKeys(rowKeySet);
            }
       }
        partiallyRefreshUIComponent("pc1");
}


/*
 ********************************************************
 * PRIVATE METHODS
```

```
     * ********************************************************/
      private TreeTableStateBean getTreeTableStateObject(){
        TreeTableStateBean stateBean = null;
        FacesContext fctx = FacesContext.getCurrentInstance();
        ELContext elctx = fctx.getELContext();
        ExpressionFactory elFactory =
               fctx.getApplication().getExpressionFactory();
        ValueExpression ve = elFactory.createValueExpression(
                              elctx,
                              "#{viewScope.treeTableStateBean}",
                              Object.class);
        stateBean = (TreeTableStateBean) ve.getValue(elctx);
        return stateBean;
      }

       //refreshes component by passed on component id. Be aware of
       //naming containers! Components in a naming container must have
       //a prefix "<namingContainerId>:"
       private void partiallyRefreshUIComponent(String uid) {
           FacesContext fctx = FacesContext.getCurrentInstance();
           UIViewRoot viewRoot = fctx.getViewRoot();
           UIComponent component = viewRoot.findComponent(uid);
           partiallyRefreshUIComponent(component);
       }


       private void partiallyRefreshUIComponent(UIComponent component)
       {
          AdfFacesContext adfFacesContext =
                            AdfFacesContext.getCurrentInstance();
          if (component != null) {
           adfFacesContext.addPartialTarget(component);
          }
       }
}
```

Conclusion & Sample Download

The sample application for this article can be downloaded as sample #84 from the ADF Code Corner website

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

You need to configure it to access the HR schema in a local Oracle XE, standard or enterprise database installation. Run the JSPX to start testing.

Note that the LocationsView uses a Viewcriteria that only displays Locations having Departments. It is easier to demonstrate a solution if there is data available for a hierarchy than without.

Note: The use case implemented in the sample comes with a "little penalty", which is the partial refresh request required to show / hide columns that are dependent on the disclosed node state.

RELATED DOCOMENTATION

☒	
☒	
☒	