

ADF Code Corner

83. How-to create bi-directional synchronization between a tree and an input form component

ORACLE
CODE CORNER



twitter.com/adfcodecorner

Abstract:

Oracle ADF makes it easy to synchronize input forms with the node or leaf selection in a tree components. Synchronizing a form however usually does not meet the needs and you usually want to make the other direction, which is the tree to synch with changes in a form, work as well. The changes however could include the creation of a new record, in which case a simple PPR of the tree is not enough to expand the tree to show the new record as the selected tree node. This article demonstrates how to build bi-directional synchronization between a tree and an input form.

Author:

Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
23-MAY-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>

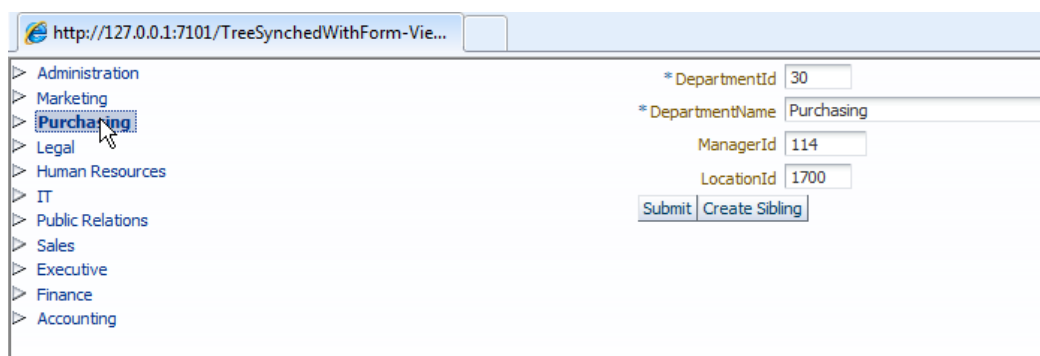
Introduction

Since most of this use case is easier to document by explaining the programming steps involved, lets directly start with the sample. The sample application is based on the HR schema and contains a single JSPX document for you to run.

The browser shows a tree that only lists departments having children. The query to only show departments with employees is achieved by a View Criteria that is applied to the Departments view instance in the Application Module model.

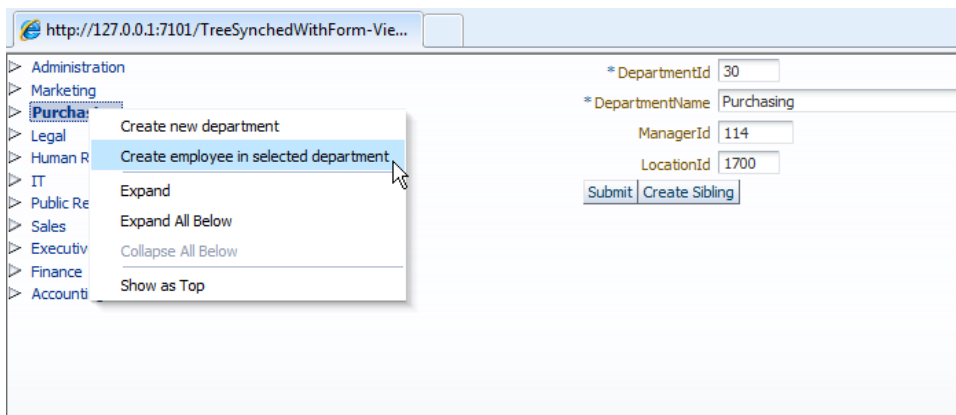


The edit form for the selected node is first time shown when you select a node.

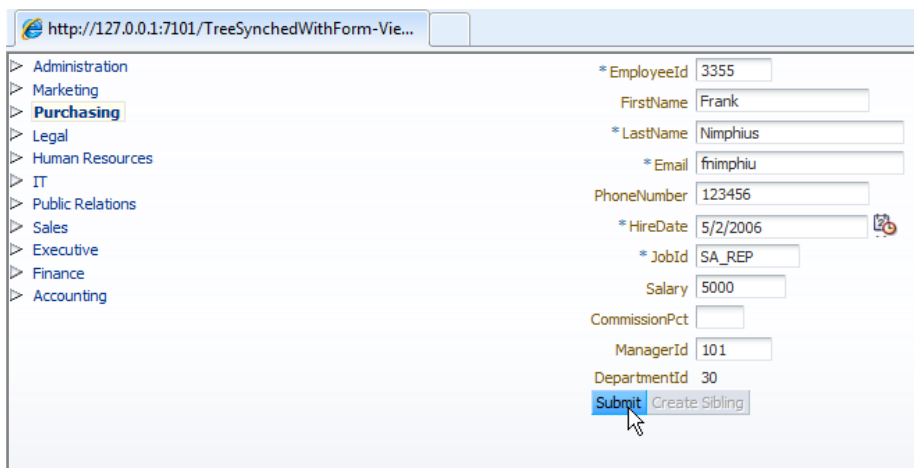


The Sample provides two options for you to create new departments and employees:

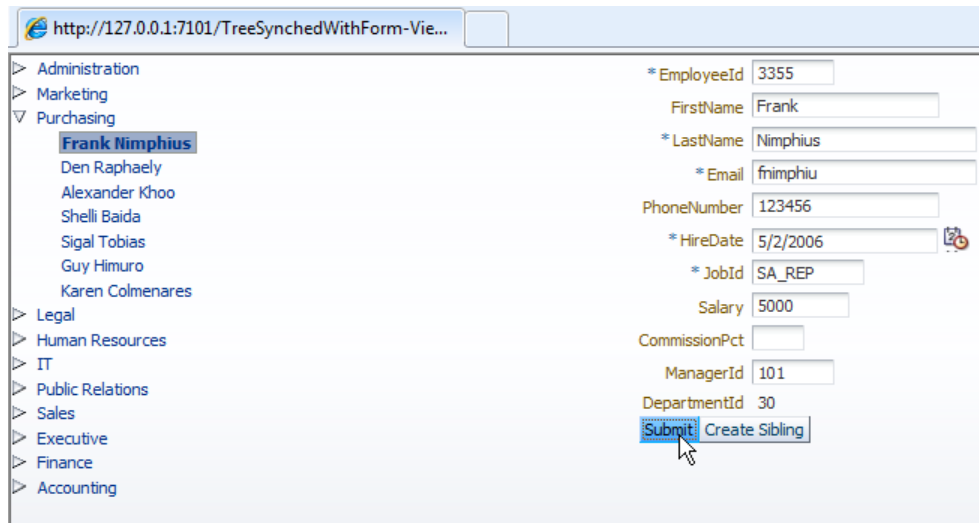
- The context menu on the tree allows you to create a new department or a new employee for the selected department. Unless you re-query the the departments view, the applied filter that only departments with employees should be shown does not hide newly created departments. When you create a new tree entry, after pressing the **Submit** button, the tree is synchronized such that the node represented by the edit form is selected, and, if needed, its ancestor node disclosed.
- The **Create Sibling** command buttons allow you to create new sibling nodes. Note that when you create a new employee as a sibling, the department id of the selected parent node is automatically copied to the new record. The copy action is handled in a managed bean



In the sample, create a new employee using the context menu option.



Note how the **Create Sibling** button is disabled for as long as you edit a newly created row. The button is enabled only for submitted data rows. To see how this disabling is implemented, have a look at the button's disabled property and the EL therein.

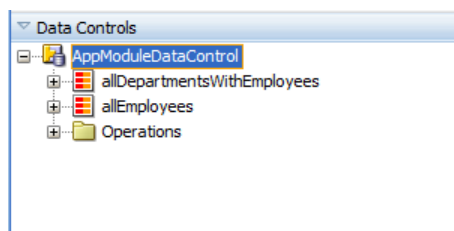


As shown above, on submit, the tree view is synchronized with the newly created data row. If the row is a child record, then the parent node is disclosed. The sample uses a two level hierarchy in the tree. If your tree has a deeper hierarchy, then the solution introduced in this article will still work but require adaption.

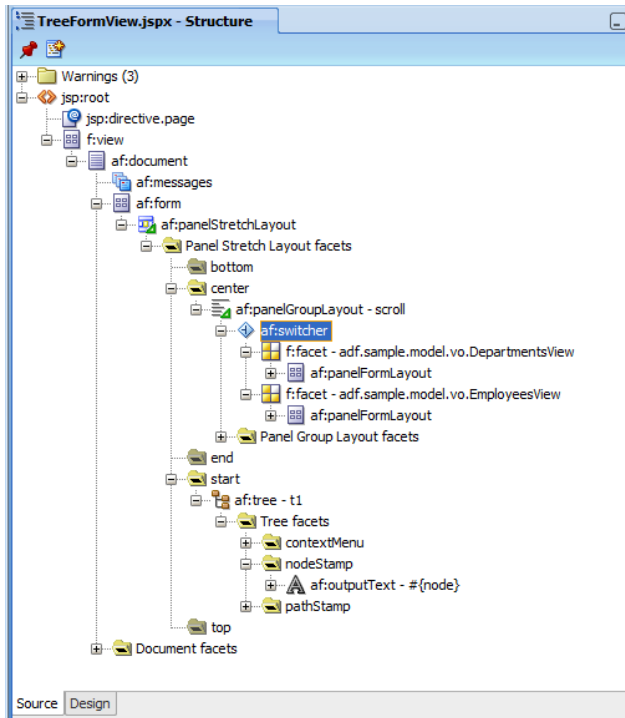
How-to

The data model exposed on the Application Module consists of two View Object instances

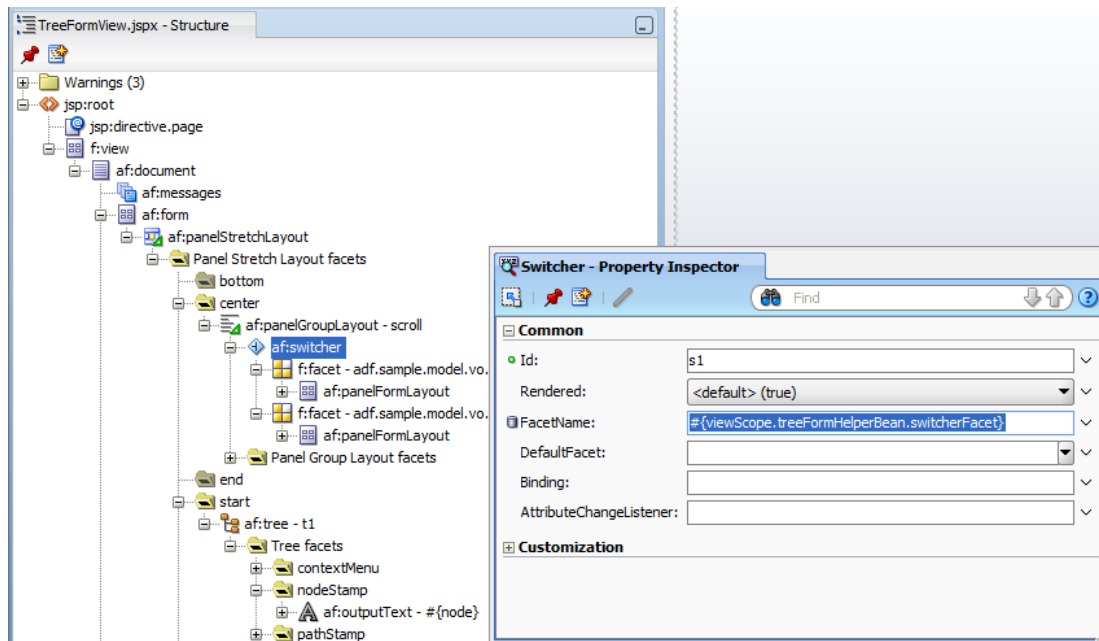
- **allDepartmentsWithEmployees** view is an instance of DepartmentsView with a View criteria applied that only shows departments having employees (the reason for this is that trees with child data just demo better). The DepartmentsView has a link to the EmployeesView for master-detail correlation.
- **allEmployees** is a list a view that queries all employees. For each node in a tree, except for the top level nodes, you will need to create a View Object instance that exposes all possible data. The allEmployees view is referenced from the tree child node so that selecting a node synchronizes the allEmployees iterator.



The form is created using an `af:panelStretchLayout`. The **start** facet of the component is the area to add the tree into. The **center** area holds the edit form. To switch between the department form and the employee form, an `af:switcher` component is used.



The `af:switcher` component is bound to a managed bean in **viewScope**. The reason for the bean to be in viewScope is that it needs to hold the state of the switcher component so that it survives requests for creating and saving new data. The next longer scope after request is viewScope.



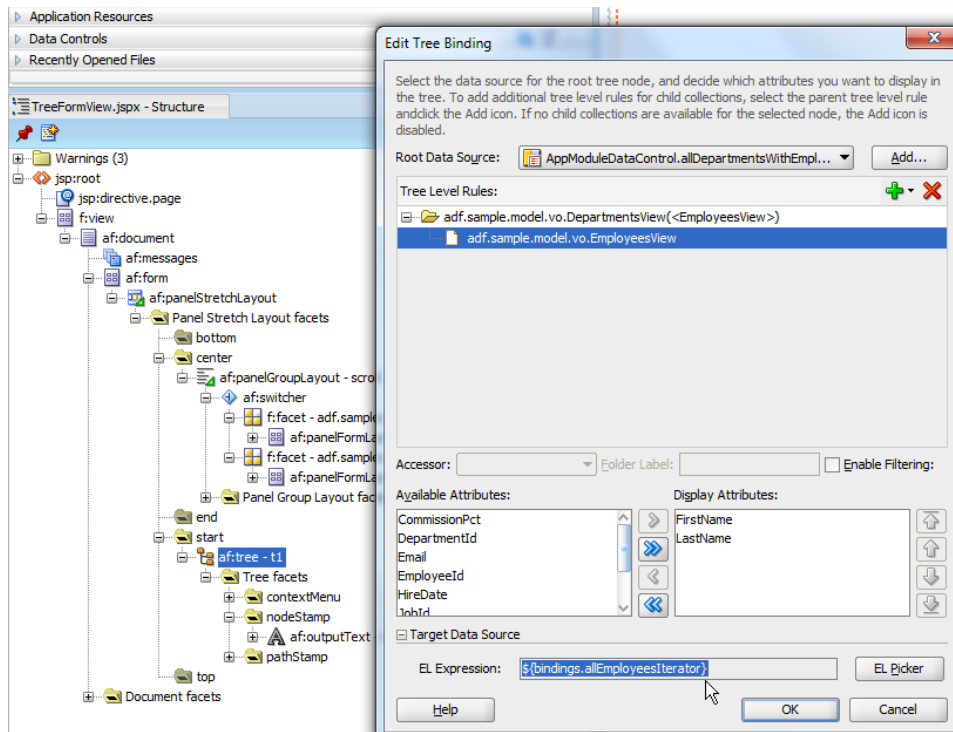
The **DefaultFacet** property is left empty so that initially, when no node is selected, the form is hidden. Note that the names of the facets in the `af:switcher` component are set to the name and package of the View Object that represents a node in the tree. While this doesn't look friendly, it is very descriptive and easy to understand for developers reviewing this code. If you are not certain of what the name for the facet should be, have a look at the pageDef file.



The tree **nodeDefinition** elements have a **DefName** attribute that helps you with the name of the object. This also works with other DataControls, like POJO.

When the managed bean returns a name that matches one of the names defined for the facets in the `af:switcher` component then the content of the facet is displayed in the UI.

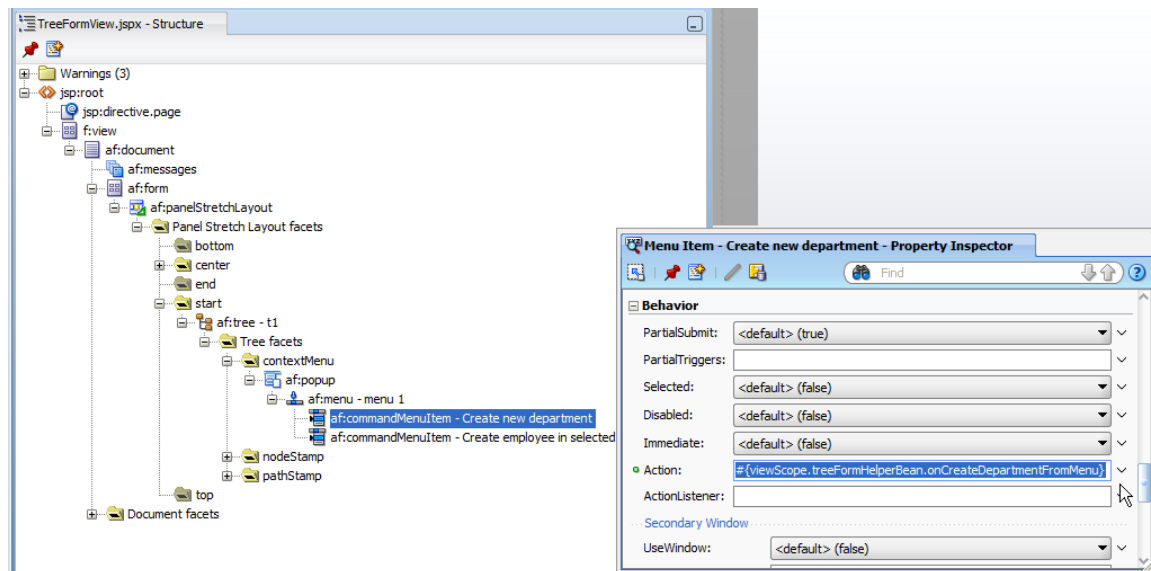
The tree configuration is such that the child node uses the **Target Data Source** option to reference the independent child iterator (`allEmployees`). Note that you have it easier to build the tree synchronization if you first create the forms in the `af:switcher` component facets. This way all the iterators you need in the tree configuration are created.



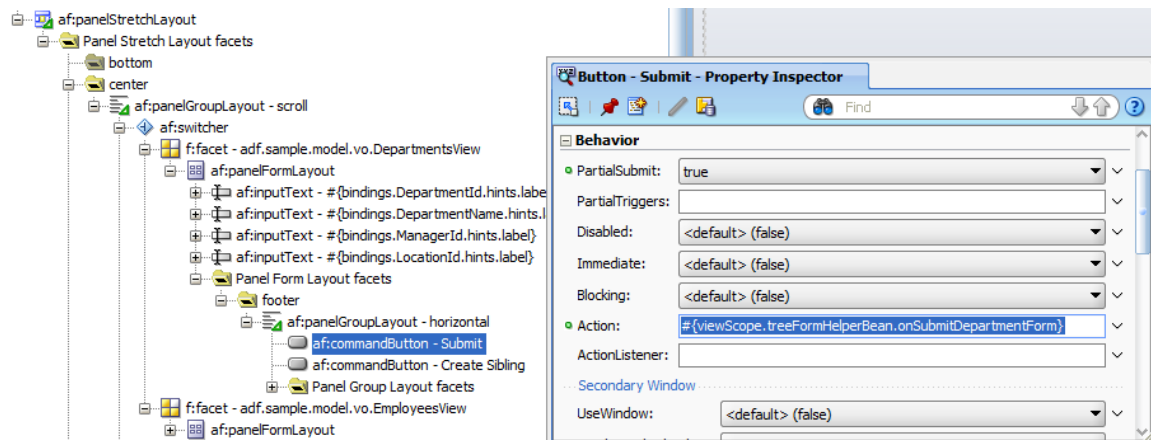
The **Target Data Source** setting ensures that when users click on a detail node, the referenced iterator is synchronized with the selection. If in response to a tree selection you partially refresh the layout container that contains the `af:switcher` component, then the edit form shows synchronized with the tree selection. However, to display the form, you need to make sure the managed bean returns the name of the `af:switcher` facet that contains the form fields for the selected node.

We are not talking rocket science here and having chosen the facet names to match with the tree node `defNames` will greatly simplify this code.

The command menu items on the context menu reference the same managed bean that holds the switcher component facet selection information. The information doesn't need to go into the same managed bean. If, for example, you wanted to also build JSF component bindings in the managed bean – for example from the tree **binding** property – then the bean must be in request or backing bean scope. However, in this sample, whenever a handle to the tree or `panelGroupLayout` component is needed, it is looked up on the JSF `UIViewRoot`.



The command buttons in the forms also reference the managed bean so we have a single place that holds all the code.



Managed Bean Code

I understand that you would expect everything in ADF to work declaratively. However, some use cases demand some Java to be written. The solution to the use case introduced in this article also requires some Java. The commented Java code of the managed bean in this sample is shown in the following.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.MethodExpression;
import javax.faces.component.UIComponent;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;

import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.adf.view.rich.component.rich.data.RichTree;
import oracle.adf.view.rich.context.AdfFacesContext;

import oracle.binding.BindingContainer;
import oracle.binding.OperationBinding;

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

import org.apache.myfaces.trinidad.event.SelectionEvent;
import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.RowKeySet;
import org.apache.myfaces.trinidad.model.RowKeySetImpl;

/**
 * TreeFormHelperBean is a managed bean configured in view scope to
 * survive a request. It is in viewScope because it keeps track of the
 * af:switcher state. Because no component binding references this
 * class, it is okay to have it in viewScope. To avoid component
 * bindings, the tree component is either looked up from the selection
 * event object or the UIViewRoot. Same for the panelGroup that
 * surrounds the af:switcher component
 */
```

```
public class TreeFormHelperBean {
    //keeps track of current selected facet
    String switcherFacet = "";
    public TreeFormHelperBean() {
        super();
    }

    public void setSwitcherFacet(String switcherFacet) {
        this.switcherFacet = switcherFacet;
    }

    public String getSwitcherFacet() {
        return switcherFacet;
    }

    /*
     * Method that decorates the default ADF tree selection listener.
     * It is responsible for setting the state of the switcher property
     * so correct input form (Department or Employee) is shown
     */
    public void treeSelectionHandler(SelectionEvent selectionEvent) {
        //preserve default selection functionality by calling EL below
        //#{bindings.allDepartmentsWithEmployees.treeModel.makeCurrent}
        //Do so using generic code that does not fail if binding names
        //in the PageDef file change
        RichTree tree = (RichTree)selectionEvent.getSource();
        CollectionModel model = (CollectionModel)tree.getValue();
        JUCtrlHierBinding treeBinding =
            (JUCtrlHierBinding)model.getWrappedData();

        //the EL code below is generic because the tree binding name is
        //read from the tree binding itself. Using this EL approach
        //ensures that the Target Data Source feature of the tree - which
        //is that the tree nodes synch up with an independent iterator in
        //the page - continues working
        String methodExpression =
            "#{bindings." + treeBinding.getName() + ".treeModel.makeCurrent}";
        //resolve method expression in private function
        this.handleTreeSelection(selectionEvent, methodExpression);
        //access selected rowKey. Because the tree is configured for
        //single node selection, we take the first entry in the RowKeySet
        //if there is one
        RowKeySet rks = selectionEvent.getAddedSet();
        Iterator iterator = rks.iterator();
        if (iterator.hasNext()) {
            List rowKey = (List)iterator.next();
        }
    }
}
```

```
//get access to the ADF tree binding
JUCtrlHierNodeBinding node =
    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 = node.getHierTypeBinding();
String defName = nodeType.getStructureDefName();
//update the managed bean state property with the DefName
//of the selected node
switcherFacet = defName;
//refresh the pabek group containing the switcher
//component. The argument passed to the method is the id of
//the panel group. If the group is contained in a naming
//container, then the container ID needs to be added as a
//prefix to the component Id, separated by a colon (:)
partiallyRefreshUIComponent("pg11");

    }
}

/**
 * Method is invoked by the "Create Sibling" button on the
 * Department form. It calls the Create operation binding
 * configured in the PageDef file for the View Object that is used
 * to build the tree root and the Department form
 * @return null
 */
public String onCreateDepartment() {
    BindingContainer bindings = getBindings();
    OperationBinding operationBinding =
        bindings.getOperationBinding("Create");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        //TODO add error handling
        return null;
    }
    return null;
}

/**
 * Method called from the Create Sibling button on the employee
 * form. It calls the Create1 binding defined in the PageDef file.
```

```
* The Create1 binding is created by dragging the Create operation
* binding from the View Object that is used to build the employee
* form. Note that the View Object is not dependent on the
* allDepartments View Object
* @return null
*/
public String onCreateEmployee() {
    BindingContainer bindings = getBindings();
    //Create1 is the operation binding created from the "Create"
    //operation of the allEmployees View Object that is synchronized
    //with the selected tree node using the Target Data Source
    //configuration in the tree binding
    OperationBinding operationBinding =
        bindings.getOperationBinding("Create1");
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
        //TODO add error handling
        return null;
    }
    //set department Id to the value of the tree parent node. This is
    //needed so the new employee row becomes a child of the selected
    //department. This is not automatically, even if the association
    //on the EO level was a composition, because the allEmployees
    //View object instance the input form is bound to is not
    //dependent on the allDepartmentsWithEmployees View Object
    //instance
    //
    //Read current parent ID from allDepartmentsWithEmployeesIterator
    //which is possible because the parent node in this sample is the
    //tree root level.
    DCIteratorBinding departmentIter =
        (DCIteratorBinding)bindings.get
        ("allDepartmentsWithEmployeesIterator");
    Row currentDepartmentRow = departmentIter.getCurrentRow();
    Object deptId =
        currentDepartmentRow.getAttribute("DepartmentId");
    DCIteratorBinding employeesIter =
        (DCIteratorBinding)bindings.get("allEmployeesIterator");
    Row currentEmployeeRow = employeesIter.getCurrentRow();
    currentEmployeeRow.setAttribute("DepartmentId", deptId);
    return null;
}
```

```
/**
 * Method is invoked from the context menu to create a new
 * department
 * @return null
 */
public String onCreateDepartmentFromMenu() {
    switcherFacet = "adf.sample.model.vo.DepartmentsView";
    onCreateDepartment();
    partiallyRefreshUIComponent("pgl1");
    return null;
}

/**
 * Method is called from the context menu to create a new employee
 * for the selected department
 * @return null
 */
public String onCreateEmployeeForSelectedDepartmentFromMenu() {
    //you get the switcherFacet value from the tree node definition
    //in the pageDef, as explained in the article.
    switcherFacet = "adf.sample.model.vo.EmployeesView";
    onCreateEmployee();
    partiallyRefreshUIComponent("pgl1");
    return null;
}

/**
 * whenever the department form is submitted, this method ensures
 * the edited record is shown as selected in the tree and the tree
 * is refreshed
 * @return null
 */
public String onSubmitDepartmentForm() {
    RowKeySet selectedRowKeys = new RowKeySetImpl();
    BindingContainer bindings = getBindings();
    DCIteratorBinding departmentIter =
        (DCIteratorBinding)bindings.get
            ("allDepartmentsWithEmployeesIterator");
    Row currentDepartmentRow = departmentIter.getCurrentRow();
    ArrayList list = new ArrayList();
    list.add(currentDepartmentRow.getKey());
    selectedRowKeys.add(list);

    //dynamically look up the tree component to set the selected
    //rowKeys and PPR it. The bean cannot have a JSF component
    //binding to the tree because it is in a broader memory scope
}
```

```
//than request scope
RichTree tree = findTreeComponentOnPage("t1");
if (tree != null) {
    tree.setSelectedRowKeys(selectedRowKeys);
    partiallyRefreshUIComponent(tree);
}
return null;
}

/**
 * whenever the employee form is submitted, this method ensures the
 * edited record is shown as selected in the tree and the tree is
 * refreshed
 * @return
 */
public String onSubmitEmployeeForm() {
    RowKeySet selectedRowKeys = new RowKeySetImpl();
    RowKeySet disclosedRowKeys = new RowKeySetImpl();
    BindingContainer bindings = getBindings();
    DCIteratorBinding employeesIter =
        (DCIteratorBinding)bindings.get("allEmployeesIterator");
    Row currentEmployeeRow = employeesIter.getCurrentRow();
    DCIteratorBinding departmentIter =
        (DCIteratorBinding)bindings.get
            ("allDepartmentsWithEmployeesIterator");
    Row currentDepartmentRow = departmentIter.getCurrentRow();
    //determine selected row key and disclosed row key (parent node).
    //In a two level tree as in this sample, it is easy to do this by
    //getting this information from the iterators. In a complex tree
    //you need to find a generic example similar to what Lynn
    //Munsinger and I describe in chapter 9 of the Fusion Developer
    //Guide book published by McGraw Hill in 2010
    ArrayList list = new ArrayList();
    list.add(currentDepartmentRow.getKey());
    list.add(currentEmployeeRow.getKey());
    selectedRowKeys.add(list);
    ArrayList disclosedList = new ArrayList();
    disclosedList.add(currentDepartmentRow.getKey());
    disclosedRowKeys.add(disclosedList);

    //get the tree component by ID. Make sure that the ID of
    //surrounding naming containers is included as a prefix
    RichTree tree = findTreeComponentOnPage("t1");
```

```
    if (tree != null) {
        tree.setSelectedRowKeys(selectedRowKeys);
        //ensure the department node gets disclosed, which we do by
        //adding the node to the list of disclosed rowKeys
        tree.setDisclosedRowKeys(disclosedRowKeys);
        partiallyRefreshUIComponent(tree);
    }
    return null;
}

/* *****
 * PRIVATE METHODS
 * *****/
private BindingContainer getBindings() {
    return BindingContext.getCurrent().getCurrentBindingsEntry();
}

//helper method to synch the current tree node selection with the
//ADF binding
private void handleTreeSelection(SelectionEvent selectionEvent,
                                String el) {
    FacesContext fctx = FacesContext.getCurrentInstance();
    ELContext elctx = fctx.getELContext();
    ExpressionFactory exprFactory =
        fctx.getApplication().getExpressionFactory();
    MethodExpression me =
        exprFactory.createMethodExpression(
            elctx,
            el,
            Object.class,
            new Class[] { SelectionEvent.class });
    me.invoke(elctx, new Object[] { selectionEvent });
}

//quick and dirty approach to find a component in the view root.
//There are better options for this:
//see sample 058 on the ADF Code Corner website
// ... but this here does for what we need in this sample
private RichTree findTreeComponentOnPage(String treeId) {
    FacesContext fctx = FacesContext.getCurrentInstance();
    UIViewRoot viewRoot = fctx.getViewRoot();
    UIComponent component = viewRoot.findComponent(treeId);
    if (component != null && component instanceof RichTree) {
        return (RichTree)component;
    }
}
```

```

    return null;
}

//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);
    }
}
}

```

Download

You can download the sample for this article as sample 083 from the ADF Code Corner website

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

Configure the database connection of the sample to access the HR schema in a local database of yours and run the JSPX page.

Note: You also find related articles on the ADF Code Corner website

RELATED DOCUMENTATION

<input type="checkbox"/>	af:switcher component tag http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_switcher.html
<input type="checkbox"/>	af:panelStretchLayout component tag http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_panelStretchLayout.html
<input type="checkbox"/>	The ADFCode Corner website contains related articles http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html
<input type="checkbox"/>	Oracle Fusion Developer Guide (McGraw Hill), Frank Nimphius, Lynn Munsinger http://www.mhprofessional.com/product.php?cat=112&isbn=0071622543