ADF Code Corner

050. How-to create and synchronize edit forms for tree node entries

twitter.com/adfcodecorner

Abstract:

A tree displays hierarchical data structures and often is used as a mean of navigation within data. A common use case is to display an input form in response to the tree node or ree leaf selection by the user.

In this blog article I look at what it takes to synchronize an input form with a tree selection and how to switch between input forms created for different level in a tree. A generic selection listener introduced in an earlier blog posting is used to handle the user selection and display the form.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
29-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 image below shows the runtime view of the example discussed in this article. Upon selection of a tree node or leaf, an input form is dynamically displayed to allow users to edit data associated with the tree selection.

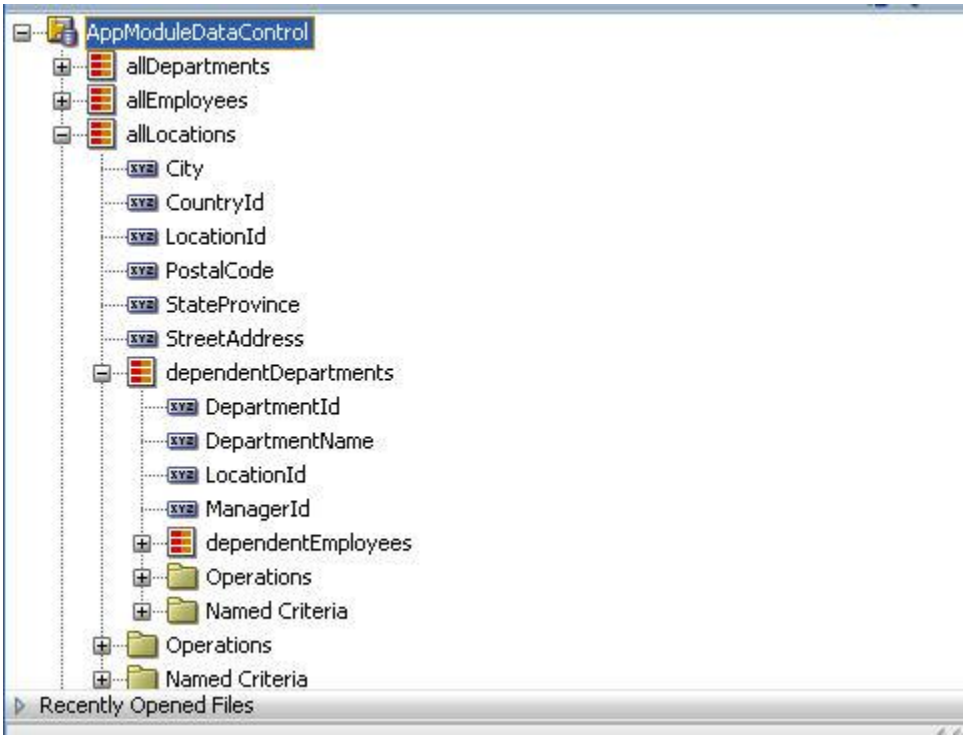
Though not shown in the example, information can also be created and deleted using this approach. In ADF, by default, the tree binding selection is handed by a method in an internal ADF binding class, which is referenced by EL from the SelectionListener property. The EL looks similar to shown next: #{bindings.<tree binding name>.makeCurrent.

Developers who need to add pre- or post processing information to the listener execution, as required in this example, can use the existing EL string in a Java EL MethodExpression, or write a custom listener completely in Java, which is approach taken in this article. The approach of using the default EL expression in a Java MethodExpression is explained in chapter 9 of the Fusion Developer Guide book Lynn Munsinger and I wrote for McGraw Hill. So if you are interested in this option, you can have a look there.

The Data Controls panel

The Data Controls panel is the starting point for the tree development. It exposes three collections that may come from ADF Business Components (used in this example) or any other DC implementation, including Web Services and POJO.

The *allLocations* collection is the one to start with in this example and contains collection the details, *dependentDepartments* and *dependentEmployees*. Note that the two other stand alone collections *allDepartments* and *alEmployees* are important for building the synchronized input forms.



Building the input forms

In this example, I am using a **PanelSplitter** layout component to separate the tree from the input form. For this use case, it is most convenient to start with building the input forms before dragging the *allLocations* collection to become the tree. In the **Operations** panel of the **ADF Faces Component Palette**, you find the **Switcher** (af:switcher) component, which you drag and drop onto the right hand side of the PanelSplitter.

To quote the tag docs: *"The switcher component dynamically decides which facet component should be rendered. It has two properties. The switcher will render the facet matching "facetName"; however, if no such facet exists (or "facetName" is null), and "defaultFacet" has been set, then that facet will be used instead. (It's possible to achieve this same functionality by using a panelGroup and binding the "rendered" property of each child, but this component can be simpler. Ordinary children of the switcher component are not rendered at all.)*

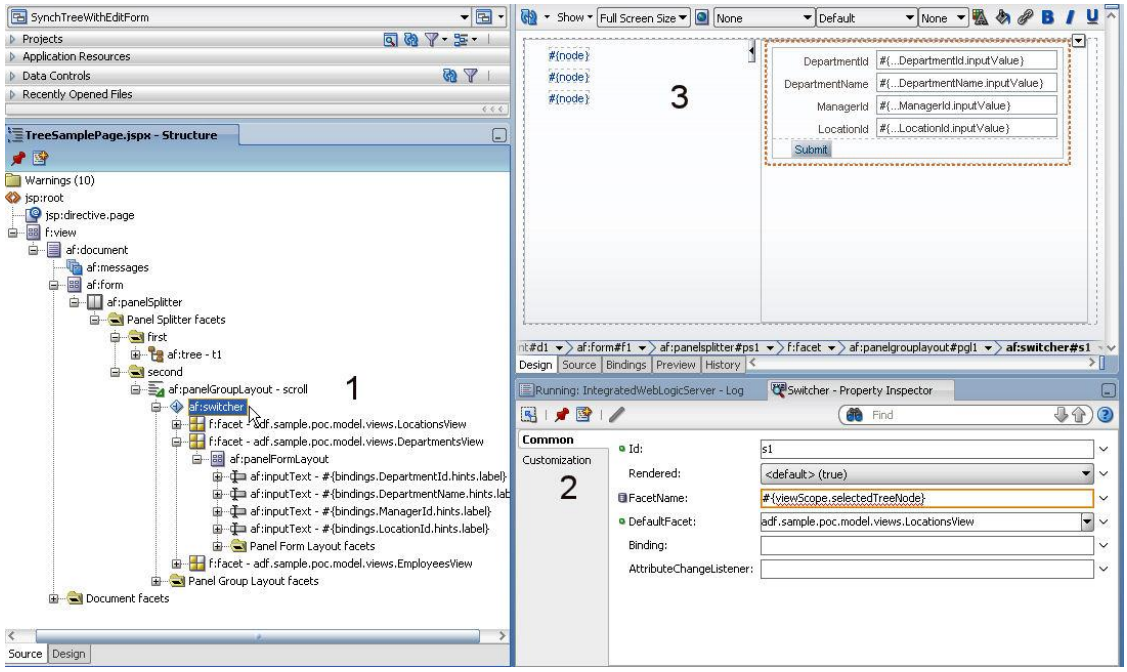
The switcher is a purely logical server-side component. It does not generate any content itself and has no client-side representation (no client component). Hence switching which facet of the switcher renders requires a server round-trip."

The **af:switcher** component does not support partial page refresh itself, I made sure the af:switcher tag is surrounded by **af:panelGroupLayout** component with the layout property set to "scroll". This way, refreshing the PanelGroupLayout implicitly refreshes the switcher.

The af:switcher component, shown in the image below (1), works like a switch statement in Java and allows you to change the UI based on an outer condition, which in this blog example is an attribute in the viewScope. Using the viewScope also means that ADF Task Flow (ADF Controller) must be configured for the project, which is the case when starting your development with the Fusion template in JDeveloper.

The **af:switcher** component has three custom facets that you create with a right mouse click onto the **af:switcher** node in the JDeveloper Structure Window. This opens a context dialog for you to create a new facet. The name of the facet in this example is the absolute name - package and class name - of the collection to display. In this example I use ADF Business Components with the View Objects located in the *adf.sample.poc.model.view* package. For each View Object, I created one facet. If your collection is based on POJO entities, then the name of the facet is the package and class name of the POJO entities represented in the tree.

If you have a look at the Property Inspector for the **af:switcher** (2), you see two property settings: *FacetName* and *DefaultFacet*. The FacetName property determines the current facet to display and looks up an attribute in the viewScope. This attribute will be filled in the custom tree selection listener of this article with the information - package name and class name - of the selected node. The *DefaultFacet* property determines the node to display if the attribute is missing or the value in it does not resolve to a facet defined for the **af:switcher** component. The visual editor (3) shows the input form dragged into the DepartmentsView facet. To build the input forms you drag the *"allLocations"* collection entry into the *"LocationsView"* facet, the *"allDepartments"* collection into the *"DepartmentsView"* and the *"allEmployees"* collection into the *"EmployeesView"* facet.



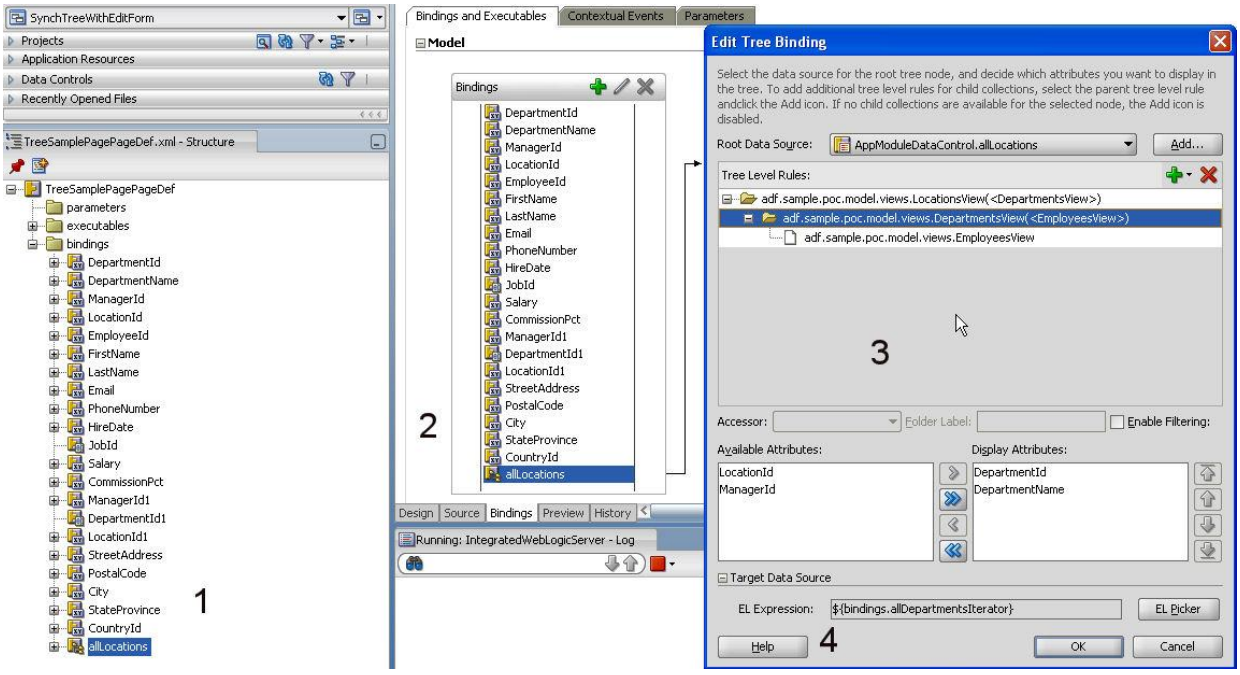
Note that it is **important** that the input form is build with collections that are **not** dependent on the the *allLocations* collection (except for the *allLocations* form itself).

Note: The collection names in my example are different from their ViewObject names. This change in the naming is instance specific and defined for the View Object instances when building the data model in the ADF Business Component Application Module.

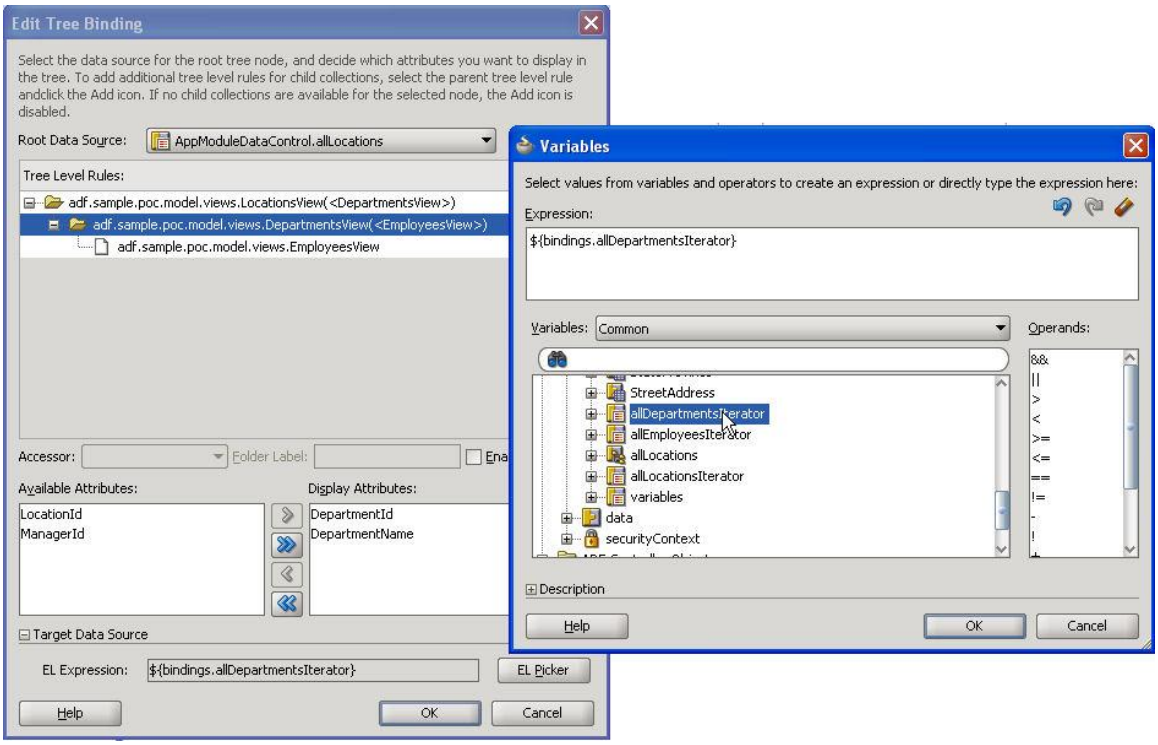
Building the tree

Now its time to build the tree. For this, drop the *"allLocations"* (1) collection (The LocationsView) onto the left hand side of the tree, selecting the tree option from the ADF context menu.For each of the tree node levels, press the green plus icon to create a node rule (3). For the **DepartmentsView** and **EmployeesView** node there is an **extra** step (4) required. The "Target Data Source" option synchronizes the iterator of the input form with the selected tree node. For this, it uses EL to reference the iterators. Because the top level tree node - Locations - is automatically synchronized when selecting a location in the tree, the "Target Data Source" only needs to be configured for the Departments and Employees View. The value of the Target Data Source is a EL as shown below: ${bindings.<iterator name>}. The "$" in the EL syntax indicates that the EL string is immediately resolved when the tree is rendered.

Note: Trees use internal accessors to display dependent node information. Because of this internal access, by default, no iterator is created in ADF for child tree nodes. This is also why I said earlier that starting with creating the input forms is more convenient for this use case because it created the iterators to reference from the Target Data Source feature.



The image below shows the EL editor that opens when clicking the "EL Picker" button.



Preparing the af:panelGroupLayout for partial refresh

I mentioned earlier, that the **af:panelGroupLayout** surrounds the **af:switcher** tag so a partial refresh can change the displayed input form according to the selected tree item. For this you create a managed bean in request scope, which then you reference from the af:panelGroupLayout "Binding" property. Note that the component "Binding" property has nothing in common with ADF. It can be used to build an EL reference to a setter and getter method in a managed bean, given developers a chance to pass a handle to the UI component into the managed bean. To create the managed bean, you i) create a POJO class (JavaBean) and ii) register it in the overview tab of the adfc-config.xml file (or the taskflow meta data configuration that the page is part of)

Building the Custom Selection Listener

- The custom Selection Listener is a public method in the same managed bean that hold the component binding to the **af:panelGroupLayout** component. Below is the complete code of the managed bean, including the af:panelGroupLayout component binding reference. Since the listener code is generic, **its only the part highlighted in bold that is specific to this use case**.

```
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.ValueExpression;
import javax.faces.context.FacesContext;
```

```
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.adf.model.binding.DCIteratorBindingDef;
import oracle.adf.view.rich.component.rich.data.RichTree;
import oracle.adf.view.rich.component.rich.layout.RichPanelGroupLayout;
import oracle.adf.view.rich.context.AdfFacesContext;
import oracle.jbo.Key;
import oracle.jbo.uicli.binding.JUCtrlHierBinding;
import oracle.jbo.uicli.binding.JUCtrlHierNodeBinding;
import oracle.jbo.uicli.binding.JUCtrlHierTypeBinding;
import oracle.jbo.uicli.binding.JUIteratorBinding;
import org.apache.myfaces.trinidad.event.SelectionEvent;
import org.apache.myfaces.trinidad.model.CollectionModel;
import org.apache.myfaces.trinidad.model.RowKeySet;


public class TreeHelperBean {
    private RichPanelGroupLayout pgSwitcher;

    public TreeHelperBean() {
    }

/**
* Custom managed bean method that takes a SelectEvent input argument
* to generically set the current row corresponding to the selected row
* in the tree. Note that this method is a way to replace the
* "makeCurrent" EL expression (#{bindings.<tree binding>.
* treeModel.makeCurrent}that Oracle JDeveloper adds to the tree
* component SelectionListener property when dragging a collection from
* the Data Controls panel. Using this custom selection listener allows
* developers to add pre- and post processing instructions. For
* example, you may want to enforce PPR on a specific item after a new
* tree node has been selected. This methods performs the following
* steps
*
* i.   get access to the tree component
* ii.  get access to the ADF tree binding
* iii. set the current row on the ADF binding
* iv.  get the information about target iterators to synchronize
* v.   synchronize target iterator
*
* @param selectionEvent object passed in by ADF Faces when configuring
* this method to become the selection listener
*
```

```
* @author Frank Nimphius
*/
  public void onTreeSelect(SelectionEvent selectionEvent) {
   //get the tree information from the event object
   RichTree tree1 = (RichTree) selectionEvent.getSource();
   //in a single selection case ( a setting om the tree component )
   //the added set only has a single entry. If there are more then
   //using this method may not be desirable.
   //Implicitly we turn the multi select in a single select later,
   //ignoring all set entries than the first
   RowKeySet rks2 = selectionEvent.getAddedSet();
   Iterator rksIterator = rks2.iterator();

   //support single row selection case
   if (rksIterator.hasNext()){
   //get the tree node key, which is a List of path entries describing
   //the location of the node in the tree including its parents nodes
   List key = (List)rksIterator.next();
   //get the ADF tree  binding to work with
   JUCtrlHierBinding treeBinding = null;
   //The Trinidad CollectionModel is used to provide data to trees and
   //tables. In theADF binding case, it contains the tree binding as
   // wrapped data
   treeBinding = (JUCtrlHierBinding)
              ((CollectionModel)tree1.getValue()).getWrappedData();
   //find the node identified by the node path from the ADF binding
   //layer. Note that  we don't need to know about the name of the
   // tree binding in the PageDef file because all information is
   //provided
   JUCtrlHierNodeBinding nodeBinding =
                              treeBinding.findNodeByKeyPath(key);
   //the current row is set on the iterator binding. Because all
   //bindings have an internal reference to their iterator usage, the
   //iterator can be queried from the ADF binding object
   DCIteratorBinding _treeIteratorBinding = null;
   _treeIteratorBinding = treeBinding.getDCIteratorBinding();
    Key rowKey = nodeBinding.getRowKey();
   JUIteratorBinding iterator = nodeBinding.getIteratorBinding();
   iterator.setCurrentRowWithKey(rowKey.toStringFormat(true));
   //get selected node type information
   JUCtrlHierTypeBinding typeBinding =
                         nodeBinding.getHierTypeBinding();

   // The tree node rule may have a target iterator defined. Target
   // iterators are
```

```
    // configured using the Target Data Source entry in the tree node
    // edit dialog and allow developers to declaratively synchronize an
    // independent iterator binding with the node selection in the
    // tree.
    String targetIteratorSpelString =
                                typeBinding.getTargetIterator();

    // chances are that the target iterator option is not configured.
    // We avoid NPE by checking this condition

    if (targetIteratorSpelString != null &&
        !targetIteratorSpelString.isEmpty()) {

      //resolve SPEL string for target iterator
      DCIteratorBinding targetIterator =
          resolveTargetIterWithSpel(targetIteratorSpelString);
      //synchronize the row in the traget iterator
      targetIterator.setCurrentRowWithKey(rowKey.toStringFormat(true));
    }

  /* CUSTOM POST PROCESSING  */

    //get the information about the object represented by the clicked
    //node. This is the absolute package and VO name in the ADF BC
    //case. Using POJO, this is the package and class name of the
    //entity object

    String nodeStuctureDefname = typeBinding.getStructureDefName();

   //store it in view scope for the input field switcher to pick up in
   //this example
     Map viewScope =
          AdfFacesContext.getCurrentInstance().getViewScope();
     viewScope.put("selectedTreeNode", nodeStuctureDefname);

     //PPR - refreshes the af:panelGroupLayout that switches the input
     //form
    AdfFacesContext adffacesctx = AdfFacesContext.getCurrentInstance();
    adffacesctx.addPartialTarget(this.getPgSwitcher());
   }
  }

 /**
  * Helper method to resolve EL expression into DCIteratorBinding
  * instance
```

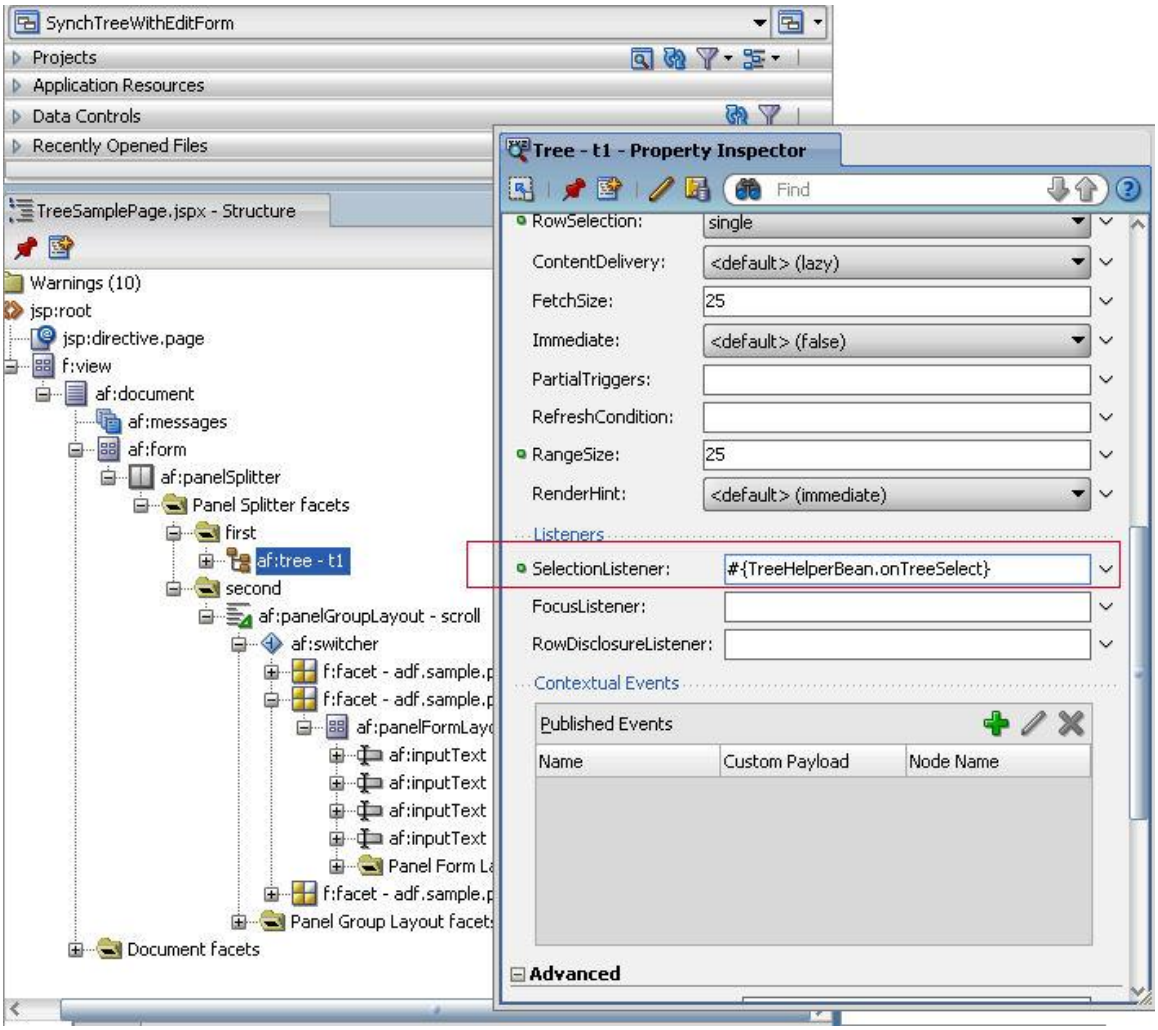
```
 * @param spelExpr the SPEL expression starting with ${...}
 * @return DCIteratorBinding instance
 */
  private DCIteratorBinding resolveTargetIterWithSpel(
                                        String spelExpr){
    FacesContext fctx = FacesContext.getCurrentInstance();
    ELContext elctx = fctx.getELContext();
    ExpressionFactory elFactory =
              fctx.getApplication().getExpressionFactory();

    ValueExpression valueExpr =
        elFactory.createValueExpression(elctx, spelExpr,Object.class);
    DCIteratorBinding dciter =
                  (DCIteratorBinding) valueExpr.getValue(elctx);
    return dciter;
  }

  public void setPgSwitcher(RichPanelGroupLayout pgSwitcher) {
    this.pgSwitcher = pgSwitcher;
  }

 public RichPanelGroupLayout getPgSwitcher() {
   return pgSwitcher;
 }
}
```

Note: Using the default selection listener reference created by ADF (#{bindings.<tree name>.makeCurrent()} in a MethodExpression definitively shortens the amount of code to write.

However, he advantage of the pure Java solution in this article is that it is save in that refactoring on the binding layer, like re-naming of the tree binding, does not break functionality.

Configuring the custom Selection Listener

The last thing to do is to reference the managed bean selection listener method from the SelectionListener property of the tree component as shown below



Download Example

You can download a JDeveloper 11g R1 PS1 workspace with the example discussed in this article from ADF Code Corner;

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

Make sure you configure the ADF Business Components data connection to the HR schema of your database. Then run the JSPX page in the ViewLayer project to see it in action.

RELATED DOCOMENTATION

☒	
☒	
☒	