ADF Code Corner

38. How-to build an editable tree with the POJO Data Control

twitter.com/adfcodecorner

Abstract:

Editable tree nodes is a use case that frequently shows up in questions asked on internal and external Oracle JDeveloper forums. If you use ADF Business Components as a business service, then, because of its tight integration with ADF and its active data model, there isn't much for you to do other than making the tree nodes editable. However, if using a data control that accesses non-ADF BC business services, like POJO or Web Services, the call to persist the data changes performed through the ADF binding layer is an extra step to consider. This blog article discusses a POJO based ADF model and one of the strategies that exist to make editable trees to work in this environment. For most part of this article, using a POJO model is not different from using ADF Business Components.

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

Trees and tables both use the ADF tree binding to update the business service model. However, only for tables there is an option in the ADF Data Controls panel context menu to declaratively create an editable UI. However, only because it isn't in the context menus for drag and drop, doesn't mean it can't be done in ADF. In this blog article I explain how to create an editable tree component that also customizes the node UI to the exposed information. The example, that you can download at the end of this article uses a POJO bean that represents the Oracle HR database sample schema with hard coded strings. While the model can be updated, there is no mean of permanently persisting the information (because of the hard coded values in the bean) and therefore I simulated persistence by printing the saved entity ID to the JDeveloper message window. Doing it this way doesn't require you to perform any infrastructure configurations, but just to run the demo.

POJO and Web Services models

From an ADF perspective, POJO and Web Services are closely related and, in addition, Web Services could also be accessed from a Java proxy client that can be access from the POJO DataControl.

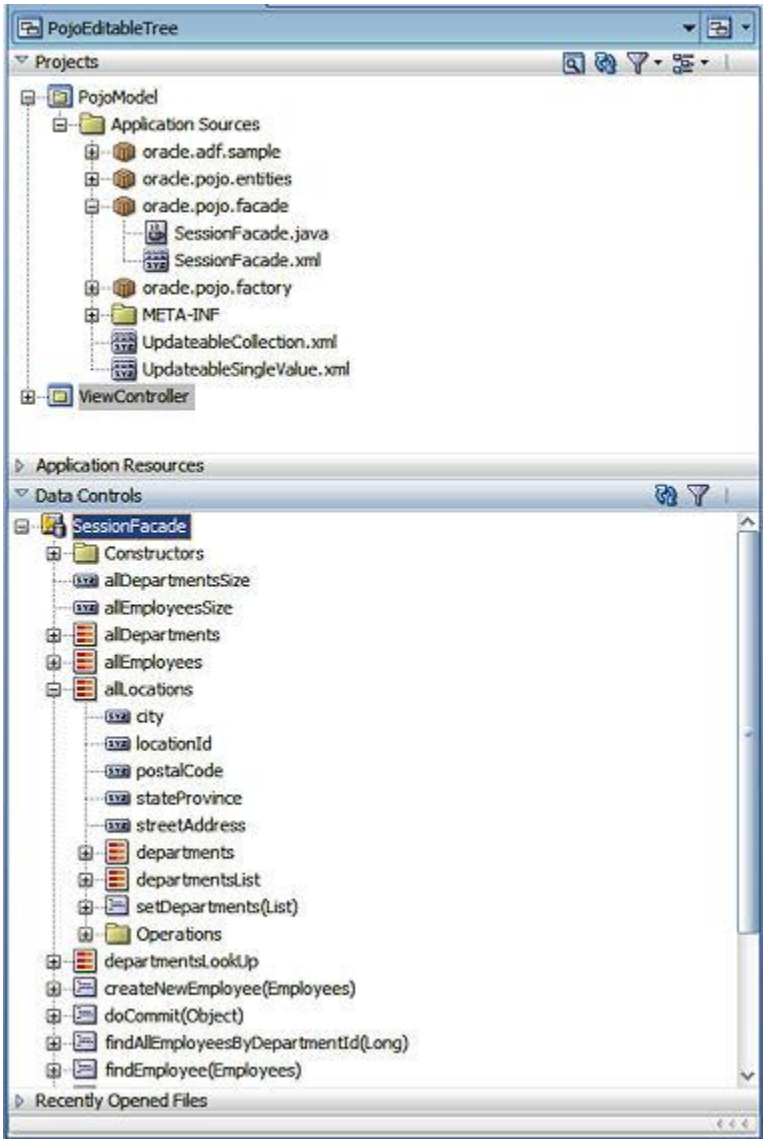
In fact both data controls, the POJO Data Control and the Web Service Data Control, internally use the same adapter Data Control framework as a basis. In general, POJOs makes a lot of sense to have and use Oracle ADF projects whenever the system or service you need to access does not expose method signatures needed by ADF, for example to implement pagination or batch updates. But also when using the Web Service Data Control instead of a Web Service proxy, if the service allows you to post data updates, it will expose a method to do so, which you can - of course - use with an editable tree.

Building the editable ADF Faces tree component

Building trees in ADF starts with what Oracle ADF is really good at: drag and drop. Having created a data control definition from the entry Java Bean (session facade, in geek terms), the Data Controls panel shows a list of methods, colllections and attributes exposed by the POJO model.

To build this, you select the POJO bean - Session Facade in the sample - and choose "Create Data Control" from the context menu. Personally I think that "Create Data Control" is a misleading term

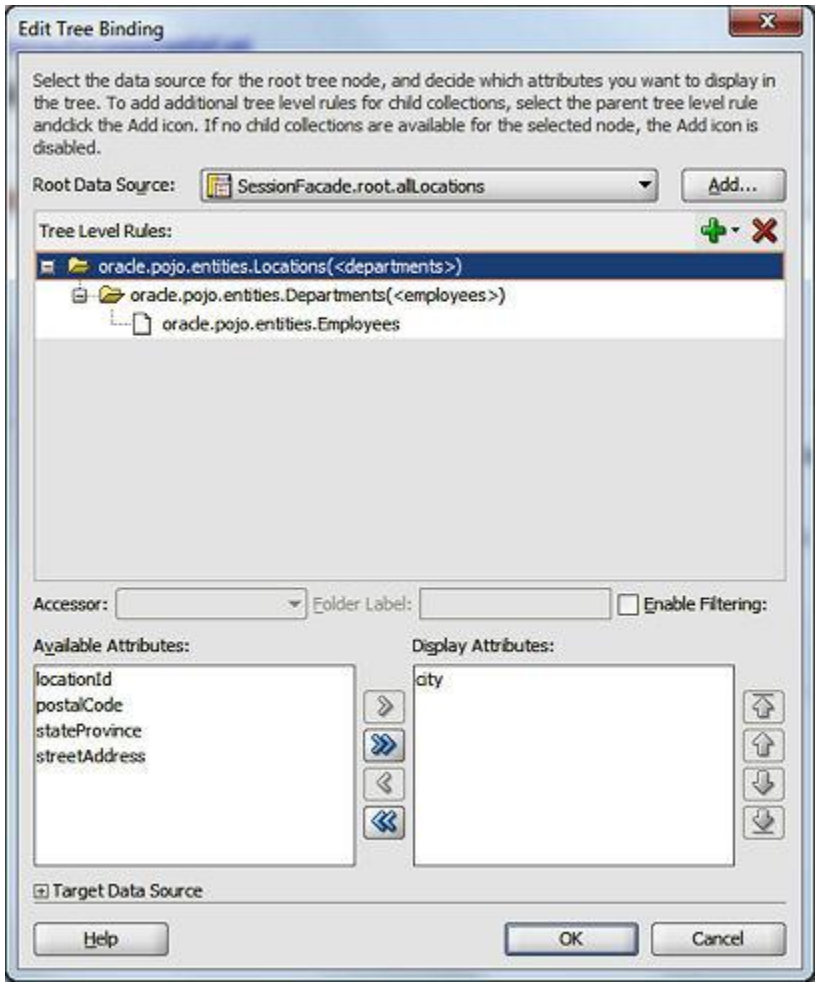
because it doesn't really describe what is happening. Using the menu option you basically don't create a data control but configure it. The JavaBean Data Control exists within the ADF framework and all that is needed to make it work for a custom POJO model is to describe the POJO entities and facade methods in metadata so their functionality and attributes show in the Data Controls panel. I am not going into POJO Data Control building detail here, but as you can see, all JavaBeans have metadata descriptions generated. You can use these metadata files to also specify UI hints, validation rules and default values. Just double click on them to see what they have in store for you to use.



Shown in the image above, one of the collections in the POJO model of the sample you can download at the end of this article is "allLocations", which is created from a method "getAllLocations" in my session facade.

Hint: If there is something to know about building POJO models for ADF then surely this is not to use "findAllLocations" as a method name to query collection data, but "getAllLocations" so the collection properly displays in the Data Controls panel. The "allLocations". collection has details objects, which are the departments and employees collection.

The hierarchy in the object model allows us to build a tree from it. So dragging the "allLocations" from the Data Controls panel onto the page and choosing "tree" as the component to build, produces the following dialog:



Pressing the green plus icon is how you create tree rules for the top level and the child nodes, which basically means to select the attributes to displayed for each tree node at runtime. Implicitly the dialog creates an ADF tree binding in the page's associated PageDef.xml file. An af:tree component is added to the page and updated with a reference to the ADF tree binding.

```
<af:tree value="#{bindings.allLocations.treeModel}" var="node"
```

The "node" variable is a temporary value holder that is filled with the content of each node to render while the tree builds or when it is getting refreshed. This variable is emptry when the tree finished

rendering. By default, the following markup is generated for the tree to render its nodes in read-only mode

```
<f:facet name="nodeStamp">
    <af:outputText value="#{node}" id="ot1"/>
</f:facet>
```

To change the tree from read-only nodes to editable nodes, there is more needed than just changing af:outputText to af:inputText, especially if a node is supposed to display more than a single attribute, like when showing a firstName and lastName pair. Also, you may not want or need the whole tree to be editable and therefore may want to render node levels differently. This is where the **af:switcher** component in ADF Faces comes in handy.

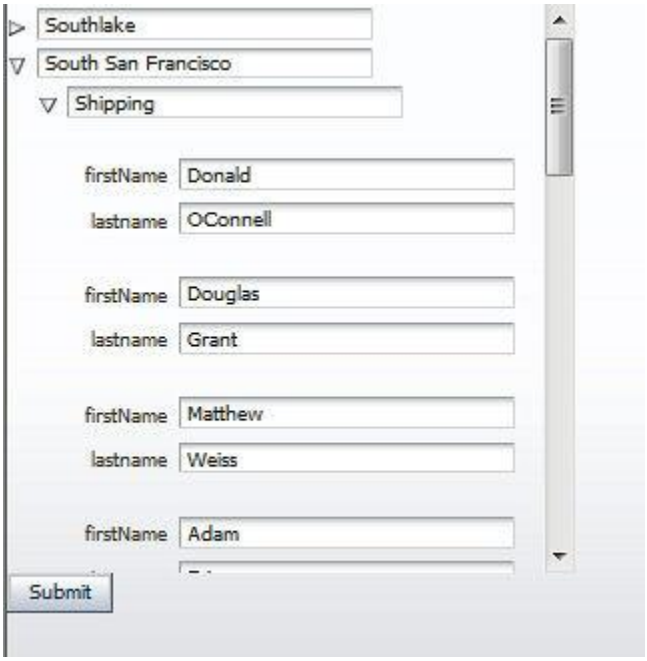
```
<af:tree value="#{bindings.allLocations.treeModel}" var="node"
     rowSelection="single" id="t1"
     selectionListener="#{bindings.allLocations.treeModel.makeCurrent}"
     binding="#{InputTreeBean.tree1}">
     <f:facet name="nodeStamp">
         <af:switcher
             facetName="#{node.hierTypeBinding.structureDefName}"
             defaultFacet="oracle.pojo.entities.Locations">
            <f:facet name="oracle.pojo.entities.Locations">
               <af:inputText value="#{node.city}" id="it1"
                             label="#{bindings.city.hints.label}"
                             autoSubmit="false"/>
            </f:facet>
            <f:facet name="oracle.pojo.entities.Departments">
               <af:inputText value="#{node.departmentName}"
                         id="it2"
                         label="#{bindings.departmentName.hints.label}"
                         autoSubmit="false"/>
            </f:facet>
            <f:facet name="oracle.pojo.entities.Employees">
              <af:panelFormLayout id="plm1">
                <af:inputText value="#{node.firstName}" id="it3"
                             simple="false"
                             label="#{bindings.firstName.hints.label}"
                             autoSubmit="false"/>
                <af:inputText value="#{node.lastname}" id="it4"
                             simple="false"
                             label="#{bindings.lastname.hints.label}"
                             autoSubmit="false"/>
              </af:panelFormLayout>
            </f:facet>
         </af:switcher>
     </f:facet>
</af:tree>
```

In the example above, the tree nodes are distinguished by the node entity package and class name. Knowing the entity that renders a specific node level requires some knowledge about the POJO model and its entity hierarchy, but it is an easy to use approach. The af:switcher uses facets you create to render markup based on a ceratin condition, like a specific node level. Each facet has a name, which is the

condition that need to be met by the value referenced by he af:switcher "facetName" property. The facetName property in the example above references #{node.hierTypeBinding.structureDefName}, which returns the absolute name of the entity that is rendered by a node.

Hint: If you use ADF Business Components, then this value is the absolute name (package and object name) of the View Object that renders the node (e.g. adf.sample.model.EmployeesView). The View Object name is the name of the View Object definition, not the name of the instance exposed on teh Application Module data model.

Each facet in the code above is named after an entity rendered in the tree nodes. The "node" variable represents a row in the collection and thus can be extended to address a specific attribute of the row object. For example, the Employees entity has a property "firstName", which is accessible using #{node.firstName} when the node renders. Trying to use this EL on any other (non employees) node, whill return an empty value. At runtime, the tree created by this metadata looks as shown below.



Hint: When you download the workspace with the demo project, you will notice that the PageDef file contains attribute bindings for each of the input fields. These attribute bindings and their associated iterators are not used for the tree update. I added them to define the labels for the nodes so I could define them in the data control metadata for each attribute and method. This way labels and tooltips could be defined consistently in a single location - the Data Control - and translated. Labels that are defined on a Data Control are read from a properties file, which you can use to create internationalized version of your Data Control configuration. This is why the attribute bindings are added to the PageDef file. To add them, you select the "bindings" node in the PageDef file and use the right mouse button to choose Generic Bindings | Attribute values from the context menu

Pressing the Submit button does update the ADF tree binding in the PageDef file with the changed node data. In the POJO case, the update is passed to the POJO model (in the demo, it updates the LocationsList, EmployeesList and DepartmentsList). As mentioned earlier, the update is to the model, not

yet persisted to the database. Only if you use ADF Business Components, the next step - to explicitly persist the changed entity individually - is not required. For all other Data Controls, you need to explicitly persist the changed entity objects. Especially when using Web Services, the service so far would not know of a value update by the tree binding. Depending on the model you use, a specific method is provided that allows you to persist objects. In the example provided with this article, the method is **doCommit(Object o)** on the session facade, which just prints the persisted entity ID (e.g. department Id, employee Id or localtions Id).

But how do you determine which entity needs to be persisted? Well this is a good question and one with no easy answer. ADF does not provide information about the changed entity objects or the object state from the iterators or tree binding. A common practice thus is to just take the entities that are cached in the POJO model that you built the Data Control from and pass them to the persistence layer in the hope that - like when using EJB - this layer knows how to tell that entities have been changed and the database needs to be updated (if at all using a database). In this case you may not even need to pass data from the binding layer to the POJO model, but just invoke a method to flush the cached model entities to the persisting layer. Another option - used in this sample - is to iterate the tree in a managed bean to get all entities exposed in the tree and pass them to the peristence method.

The sample provided with this article uses a modified version of the latter approach. In a managed bean, it traverses the ADF Faces tree to get to the entities that need to be persisted. Because I created the POJO model myself (demos are cheaters), the entities have a boolean property "entityStateChanged", which is set to true whenever the ADF framework or any other accessor calls a setter method on one of its properties. This also may be an option for you to use if you have the chance to wrap the business service model entities in your POJO layer the data control is created from.

If, for example, the department name is updated, then for this entity, implicitly, the change flag is set to true. The "doCommit" method that is called to persist the change, sets the flag back. This way I implemented a simple mechanism to distinguish changed objects from others. The managed bean code "onCommitChange" that is associated with the actionListener property of the "submit" button is shown in the managed bean code below.

Note The code below is not needed if you use ADF Business Components as the business service. In this case, all you need to do is to invoke the "commit" operation to some point in time

```
public class InputTreeBean {
  private RichTree tree1;

  public InputTreeBean() {
  }

  public void onCommitChange(ActionEvent actionEvent) {
    //just in case, avoid double invocation
   if (actionEvent.getPhaseId() == PhaseId.INVOKE_APPLICATION) {
       RichTree tree = this.getTree1();
       CollectionModel model = (CollectionModel)tree.getValue();
       JUCtrlHierBinding adfTreeBinding =
                     (JUCtrlHierBinding)model.getWrappedData();
       JUCtrlHierNodeBinding root =
                     adfTreeBinding.getRootNodeBinding();
       recursiveAdfTreeTraversal(root);
```

```
        }
    }

    //traverse the tree and check the tree node entities for updates
    private void recursiveAdfTreeTraversal(JUCtrlHierNodeBinding node){
      if (node.getRow() != null){
       //use the generic DCDataRow object instead of oracle.jbo.Row
       DCDataRow row = (DCDataRow) node.getRow();
       updatePojoModel(row.getDataProvider(),
                    node.getDCIteratorBinding().getBindingContainer());
    }
     //hasChildren() always returns true, so a null check is used
     //instead to determine if a node has children
     if(node.getChildren() != null){
        for(JUCtrlHierNodeBinding childNode :
           (ArrayList<JUCtrlHierNodeBinding>) node.getChildren()){
          recursiveAdfTreeTraversal(childNode);
        }
      }
      return;
   }

 private void updatePojoModel(Object entity,
                              DCBindingContainer bindings){
    boolean update = false;
    //check the internal flag defined on the entities to see if they
    //are changed by ADF
    update = entity instanceof Departments ?
           ((Departments)entity).isEntityStateChanged() : update
    update = entity instanceof Employees ?
           ((Employees)entity).isEntityStateChanged() : update;
    update = entity instanceof Locations ?
          ((Locations)entity).isEntityStateChanged() : update;
   //if an entity has been changed, get the doCommit from the method
   //binding and persist the change
   if (update == true) {
      OperationBinding pojoUpdateBinding =
              bindings.getOperationBinding("doCommit");
      pojoUpdateBinding.getParamsMap().put("o", entity);
      pojoUpdateBinding.execute();
      }
   }

   public void setTree1(RichTree tree1) {
       this.tree1 = tree1;
   }

   public RichTree getTree1() {
        return tree1;
   }
}
```

Hint: The private method "recursiveAdfTreeTraversal" is recursively called for all child nodes found for a given node. With a bit of code changes, this method can be used e.g. to search a tree for specific search attribute values.

Hint (inportant): The allLocations iterator RangeSize by default is set to 10. Make sure this value is set to -1 as otherwise not all tree nodes are updated by ADF.

```
<accessorIterator MasterBinding="SessionFacadeIterator" Binds="allLocations"
                  RangeSize="-1" DataControl="SessionFacade"
                  BeanClass="oracle.pojo.entities.Locations"
                  id="allLocationsIterator"/>
```

Download the Sample

The Oracle JDeveloper 11.1.1.3 workspace can be **downloaded from ADF Code Corner**:

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

RELATED DOCOMENTATION

☒	
☒	
☒	