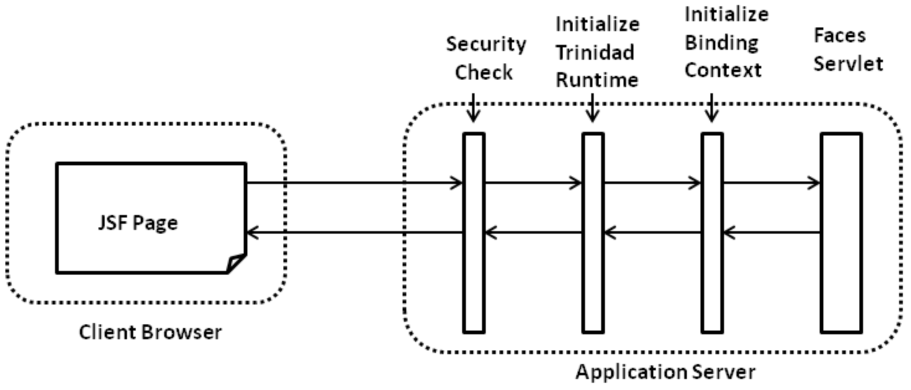
More on ADF Business Components and Fusion Page Runtime

This appendix discusses various useful features and techniques that we deferred while discussing specific topics in this book. Most of the topics that we will discuss here require a good understanding of the various layers of the ADF framework. Make sure you read the basics before you start reading this appendix. The following topics are discussed in this appendix:

- Lifecycle of an ADF Fusion web page with region
- Transaction management in Fusion web applications
- Building a dynamic model-driven UI with ADF
- Building composite view objects
- Building application modules with no database connection
- Looking up the UI component from the component tree at runtime

Lifecycle of an ADF Fusion web page with region

When a client requests for a page with region, at the server, ADF runtime intercepts and pre-processes the request before passing it to the page lifecycle handler. The pre-processing tasks include security check, initialization of Trinidad runtime, and setting up the binding context and ADF context. This is shown in the following diagram:



This appendix does not repeat the discussion on the Fusion page lifecycle phases that we have already discussed a while ago in *Chapter 7, Binding Business Services with the User Interface*. Refer back to the topic *What happens when you access a Fusion web page* in *Chapter 7, Binding Business Services with the User Interface* for a quick brush up on the Fusion page lifecycle phases.

After setting up the context for processing the request, the ADF framework starts the page lifecycle for the page. During the Before Restore View phase of the page, the framework will try to synchronize the controller state with the request, using the state token sent by the client. If this is a new request, a new root view port is created for the top-level page. In simple words, a view port maps to a page or page fragment in the current view. During view port initialization, the framework will build a data control frame for holding the data controls. During this phrase runtime also builds the binding containers used in the current page. The data control frame will then be added to the binding context object for future use. After setting up the basic infrastructure required for processing the request, the page lifecycle moves to the Restore View phase.

During the Restore View phase, the framework generates a component tree for the page. Note that the UI component tree, at this stage, contains only metadata for instantiating the UI components. The component instantiation happens only during the Render Response phase, which happens later in the page lifecycle. If this is a fresh request, the lifecycle moves to the Render Response phase. Note that, in this appendix, we are not discussing how the framework handles the post back requests from the client.

During the Render Response phase, the framework instantiates the UI components for each node in the component tree by traversing the tree hierarchy. The completed component tree is appended to `UIViewRoot`, which represents the root of the UI component tree.

Once the UI components are created, runtime walks through the component tree and performs the pre-rendering tasks. The pre-rendering event is used by components with lazy initialization abilities, such as region, to keep themselves ready for rendering if they are added to the component tree during the page cycle. While processing a region, the framework creates a new child view port and controller state for the region, and starts processing the associated task flow. The following is the algorithm used by the framework while initializing the task flow:

1. If the task flow is configured not to share data control, the framework creates a new data control frame for the task flow and adds to the parent data control frame (the data control frame for the parent view port).
2. If the task flow is configured to start a new transaction, the framework calls `beginTransaction()` on the control frame.
3. If the task flow is configured to use existing transaction, the framework asks the data control frame to create a save point and to associate it to the page flow stack.
4. If the task flow is configured to 'use existing transaction if possible', framework will start a new transaction on the data control, if there is no transaction opened on it. If a transaction is already opened on the data control, the framework will use the existing one.

Once the pre-render processing is over, each component will be asked to write out its value into the response object. During this action, the framework will evaluate the EL expressions specified for the component properties, whenever they are referred in the page lifecycle. If the EL expressions contain binding expression referring properties of the business components, evaluation of the EL will end up in instantiating corresponding model components. The framework performs the following tasks during the evaluation of the model-bound EL expressions:

- It instantiates the data control if it is missing from the current data control frame.

- It performs a check out of the application module.
- It attaches the transaction object to the application module. Note that it is the transaction object that manages all database transactions for an application module. Runtime uses the following algorithm for attaching transactions to the application module:

    1. If the application module is nested under a root application module or if it is used in a task flow that has been configured to use an existing transaction, the framework will identify the existing `DBTransaction` object that has been created for the root application module or for the calling task flow, and attach it to the current application module.

        Under the cover, the framework uses the `jbo.shared.txn` parameter (named transaction) to share the transaction between the application modules. In other words, if an application module needs to share a transaction with another module, the framework assigns the same `jbo.shared.txn` value for both application modules at runtime. While attaching the transaction to the application module, runtime will look up the transaction object by using the `jbo.shared.txn` value set for the application module and if any transaction object is found for this key, it re-uses the same.

    2. If the application module is a regular one, and not part of the task flow that shares a transaction with caller, the framework will generate a new `DBTransaction` object and attach it to the application module.

- After initializing the data control, the framework adds it to the data control frame. The data control frame holds all the data control used in the current view port. Remember that a, view port maps to a page or a page fragment.
- Execute an appropriate view object instance, which is bound to the iterator.

At the end of the render response phase, the framework will output the DOM content to the client. Before finishing the request, the ADF binding filter will call `endRequest()` on each data control instance participating in the request. Data controls use this callback to clean up the resources and check in the application modules back to the pool.

Transaction management in Fusion web applications

A transaction for a business application may be thought of as a unit of work resulting in changes to the application state. Oracle ADF simplifies the transaction handling by abstracting the micro-level management of transactions from the developers. This section discusses the internal aspects of the transaction management in Fusion web applications. In this discussion, we will consider only ADF Business Component-based applications.

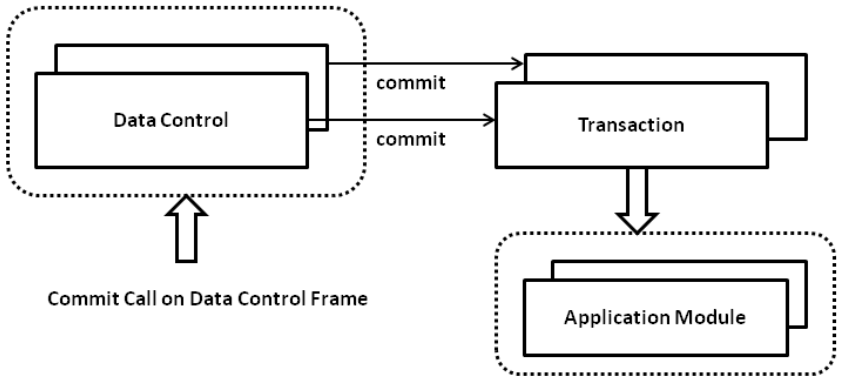
What happens when the task flow commits a transaction

Oracle ADF allows you to define transactional boundaries, using task flows. Each task flow can be configured to define the transactional unit of work. Refer back to the *Transaction management in a bounded task flow* section in *Chapter 10*, *Taking a Closer Look at the Bounded Task Flow* for a quick brush-up.

Let us see what happens when a task flow return activity tries to commit the currently opened transaction. The following is the algorithm used by the framework when the task flow commits the transaction:

1. When you action a task flow return activity, a check is carried over to see whether the task flow is configured for committing the current transaction or not. And if found true, runtime will identify the data control frame associated with the current view port and call the commit operation on it.
2. The data control frame delegates the "commit" call to the transaction handler instance for further processing. The transaction handler iterates over all data controls added to the data control frame and invokes `commitTransaction` on each root data control. It is the transaction handler that engages all data controls added to the data control frame in the transaction commit cycle.
3. Data control delegates the commit call to the transaction object that is attached to the application module. Note that if you have a child task flow participating in the transaction started by the caller or application modules nested under a root application module, they all share the same transaction object. The commit call on a transaction object will commit changes done by all application modules attached to it.

The following diagram illustrates how transaction objects that are attached to the application modules are getting engaged when a client calls commit on the data control frame:



Programmatically managing a transaction in a task flow

If the declarative solution provided by the task flow for managing the transaction is not flexible enough to meet your use case, you can handle the transaction programmatically by calling the `beginTransaction()`, `commit()`, and `rollback()` methods exposed by `oracle.adf.model.DataControlFrame`.

The data control frame acts as a bucket for holding data controls used in a binding container (page definition file). A data control frame may also hold child data control frames, if the page or page fragment has regions sharing the transaction with the parent. When you call `beginTransaction()`, `commit()`, or `rollback()` on a data control frame, all the data controls added to the data control frame will participate in the appropriate transaction cycle. In plain words, the data control frame provides a mechanism to manage the transaction seamlessly, freeing you from the pain of managing transactions separately for each data control present in the page definition. Note that you can use the `DataControlFrame` APIs for managing a transaction only in the context of a bounded task flow with an appropriate transaction setting (in the context of a controller transaction).

The following example illustrates the APIs for programmatically managing a transaction, using the data control frame:

```
//In managed bean class

public void commit(){
  //Get the binding context
  BindingContext bindingContext = BindingContext.
    getCurrent();
  //Gets the name of current(root) DataControlFrame
  String currentFrame =
    bindingContext.getCurrentDataControlFrame();
  //Finds DataControlFrame instance
  DataControlFrame dcFrame =
    bindingContext.findDataControlFrame(currentFrame);
  try {
    // Commit the trensaction
    dcFrame.commit();
    //Open a new transaction allowing user to continue
    //editing data
    dcFrame.beginTransaction(null);
  } catch (Exception e) {
    //Report error through binding container
    ((DCBindingContainer)bindingContext.
    getCurrentBindingsEntry()).
    reportException(e);
  }
}
```

Programmatically managing a transaction in the business components

The preceding solution of calling the commit method on the data control frame is ideal to be used in the client tier in the context of the bounded task flows. What if you need to programmatically commit the transaction from the business service layer, which does not have any binding context?

To commit or roll back the transactions in the business service layer logic where there is no binding context, you can call `commit()` or `rollback()` on the `oracle.jbo.Transaction` object associated with the root application modules. The following example shows a method defined in an application module, which invokes commit on the `Transaction` object attached to the root application module:

```
//In application module implementation class
/**
 *This method calls commit on transaction object
 */
public void commit(){
  this.getRootApplicationModule().getTransaction().commit();
}
```

Sharing a transaction between application modules at runtime

An application module, nested under a root application module, shares the same transaction context with the root. We have discussed the design-time support for nesting the application modules in *Chapter 6*, *Introducing the Application Module*. This solution will fit well if you know that the application module needs to be nested during the development phase of the application. What if an application module needs to invoke the business methods from various application modules whose names are known only at runtime, and all the method calls require to happen in same transaction? You can use the following API in such scenarios to create the required application module at runtime:

```
DBTransaction::createApplicationModule(defName);
```

The following method defined in a root application module creates a nested application module on the fly. Both calling and called application modules share the same transaction context.

```
//In application module implementation class
/**
* Caller passes the AM definition name of the application
* module that requires to participate in the existing
* transaction. This method creates new AM if no instance is
* found for the supplied amName and invokes required service
* on it.
* @param amName
* @param defName
*/
public void nestAMIfRequiredAndInvokeMethod(String amName, String
defName) {
  //TxnAppModule is a generic interface implemented by all
  //transactional AMs used in this example
  TxnAppModule txnAM = null;
  boolean generatedLocally = false;
  try {

    //Check whether the TxnAppModuleImpl is already nested
    txnAM = (TxnAppModule)getDBTransaction().
      getRootApplicationModule().
      findApplicationModule(amName);
    //create a new nested instance of the TxnAppModuleImpl,
    // if not nested already

    if(txnAM == null) {
      txnAM = (TxnAppModule)this.
      getDBTransaction().
      createApplicationModule(defName);
      generatedLocally = true;
    }
    //Invoke business methods
    if (txnAM != null) {
      txnAM.updateEmployee();
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    //Remove locally created AM once use is over
    if (generatedLocally && txnAM != null) {
      txnAM.remove();
    }
  }
}
```

Building a dynamic model-driven UI with ADF

All the UI components that we have discussed so far in this book were static in nature. What if you want to build a UI that needs to change its display depending on the business conditions? ADF provides basic infrastructure support for implementing such use cases. At a high level, the solution is to programmatically build a dynamic data model (view object) and to bind them with dynamically built UI components. Note that in this solution, we are leveraging the model-driven developments support offered by ADF for building a dynamic UI. This involves the following tasks:

- Building the model by using a dynamic view object
- Building the binding definition for a dynamic view object
- Building a dynamic table UI component
- Using the method call activity to initialize the model for the dynamic UI

These tasks are discussed in detail in the following sections.

Building the model by using a dynamic view object

We are following a model-driven approach for building a dynamic UI. As the UI needs to pick up data from different datasources, we cannot use static view objects for building a dynamic UI. We will build view objects dynamically whose definition changes with application state. There are multiple approaches for building dynamic view objects. These approaches are discussed in the following sections. You must choose the right approach based on the use case.

Dynamic view objects with entity usage

You will use an entity-based dynamic view object, if the user can edit data displayed in the dynamic UI. We have discussed APIs for programmatically building the view object backed up by entity objects in *Chapter 5*, *Advanced Concepts on Entity Objects and View Objects*. If you need a quick brush up on APIs, refer back to the topic *Building Business Components Dynamically* in *Chapter 5*. Here is a quick overview of the methods that we have already discussed in Chapter 5.

Step 1—building a dynamic entity object

The following code snippet illustrates the APIs for building a dynamic entity object. This example builds an entity object for the `DEPARTMENTS` table.

```
//In application module implementation class
/**
 * Build entity definition for DEPARTMENTS table
 * @return
 */
private EntityDefImpl buildDeptEntitySessionDef() {
  EntityDefImpl entDef =
    new EntityDefImpl
      (oracle.jbo.server.EntityDefImpl.DEF_SCOPE_SESSION,
      "DynamicDeptEntityDef");

  entDef.setFullName(entDef.getBasePackage() + ".dynamic." +
  entDef.getName());
  entDef.setName(entDef.getName());
  entDef.setAliasName(entDef.getName());
  //Set the database table name
  entDef.setSource("DEPARTMENTS");
  entDef.setSourceType(EntityDefImpl.DBOBJ_TYPE_TABLE);
  //Add the attributes
  entDef.addAttribute("DepartmentId", "DEPARTMENT_ID",
    Integer.class, true, false, true);
  entDef.addAttribute("DepartmentName", "DEPARTMENT_NAME",
    String.class, false, false, true);
  entDef.addAttribute("ManagerId", "MANAGER_ID",
    Integer.class, false, false, true);
  //Resolves and validates
  //entity definition before this definition object can be
  //used.
  entDef.resolveDefObject();

  //Write to XML stream
  entDef.writeXMLContents();
  //Save to hard disk
  entDef.saveXMLContents();

  return entDef;
}
```

Step 2—building a dynamic view object with the entity object usage

The following code snippet illustrates the APIs for building a dynamic view object with the entity object usage from Step 1:

```
/**
 * Build view deintion for dept EntityDefImpl
 * @param entityDef
 * @return
 */
private ViewDefImpl buildDeptViewSessionDef(EntityDefImpl entityDef) {
  ViewDefImpl viewDef = new ViewDefImpl
    (oracle.jbo.server.ViewDefImpl.DEF_SCOPE_SESSION,
    "DynamicDeptViewDef");
  viewDef.setFullName(viewDef.getBasePackage() +
    ".dynamic." + viewDef.getName());
  System.out.println("ViewDef :" + viewDef.getFileName());
  viewDef.setUseGlueCode(false);
  viewDef.setIterMode(RowIterator.ITER_MODE_LAST_PAGE_FULL);
  viewDef.setBindingStyle(
    SQLBuilder.BINDING_STYLE_ORACLE_NAME);
  viewDef.setSelectClauseFlags(
  ViewDefImpl.CLAUSE_GENERATE_RT);
  viewDef.setFromClauseFlags(ViewDefImpl.CLAUSE_GENERATE_RT);

  viewDef.addEntityUsage("DynamicDeptUsage",
    entityDef.getFullName(), false, false);

  viewDef.addAllEntityAttributes("DynamicDeptUsage");
  /**
  * It resolves attribute definitions
  * with its entity bases.
  */
  viewDef.resolveDefObject();

  viewDef.writeXMLContents();
  viewDef.saveXMLContents();

  return viewDef;
}
```

Step 3—creating a view object instance and adding it to the application module

The following is the application module method that controls the creation of an entity object and a view object, which we have seen in the preceding steps. This method delegates the call to the appropriate routines for creating the view object definition at runtime and generates a view object instance out of it. This will be exposed to be used by the client. When you build a UI for a dynamic view object, typically you drop this method as the method call activity in the task flow preceding the page (view) displaying the dynamic UI.

```
//In application module implementation class

public class HRServiceAppModuleImpl extends ApplicationModuleImpl
implements HRServiceAppModule {

  //Dynamic view object instance name used in this e.g
  private static final String DYNAMIC_DETP_VO_INSTANCE =
  "DynamicDeptVO";
/**
 * This method generates dynmaic entity definition and view
 * object definition for
 * DEPARTTMENTS table and add it to AM instance
 */
public void buildDynamicDeptViewCompAndAddtoAM() {
  //Check if view definition exist for
  //DYNAMIC_DETP_VO_INSTANCE
  ViewObject internalDynamicVO =
  findViewObject(DYNAMIC_DETP_VO_INSTANCE);
  if (internalDynamicVO != null) {
    return;
  }
  //Build entity definition
  EntityDefImpl deptEntDef = buildDeptEntitySessionDef();
  //Build view object definition
  ViewDefImpl viewDef = buildDeptViewSessionDef(deptEntDef);
  //Add view object to application module
  addViewToPdefApplicationModule(viewDef);

}

/**
 * Adds the view definition to application module
 * @param viewDef
 */
private void addViewToPdefApplicationModule(ViewDefImpl viewDef) {
    oracle.jbo.server.PDefApplicationModule pDefAM =
    oracle.jbo.server.PDefApplicationModule.findDefObject
    (getDefFullName());

    if (pDefAM == null) {
      pDefAM = new oracle.jbo.server.PDefApplicationModule();
      pDefAM.setFullName(getDefFullName());
    }

    pDefAM.setEditable(true);
    pDefAM.createViewObject(
    DYNAMIC_DETP_VO_INSTANCE, viewDef.getFullName());
    //Apply the changes to AM's PDef object
    pDefAM.applyPersonalization(this);
    //Write the changes to XML
    pDefAM.writeXMLContents();
    pDefAM.saveXMLContents();
  }
  //Other methods go here...
}
```

The preceding code snippet uses `oracle.jbo.server.PDefApplicationModule` to store the changes made at runtime, so that it will be available across sessions. The `PDefApplicationModule` implementation internally uses MDS to store the changes. To make this example work, you may need to configure MDS for the application. The following code snippet from `adf-config.xml` illustrates a sample configuration, which can be used for testing the dynamic UI example (discussed in this section):

```
<adf-config ...
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config version="11.1.1.000"
    xmlns="http://xmlns.oracle.com/mds/config">
      <persistence-config>
        <metadata-namespaces>
          <namespace path="/sessiondef"
            metadata-store-usage="mdsRepos"/>
          <namespace path="/persdef"
            metadata-store-usage="mdsRepos"/>
          <namespace path="/xliffBundles"
            metadata-store-usage="mdsRepos"/>
        </metadata-namespaces>
        <metadata-store-usages>
          <metadata-store-usage id="mdsRepos"
```

```
              deploy-target="true" default-cust-store="true"/>
          </metadata-store-usages>
        </persistence-config>
        <cust-config>
          <match path="/">
            <customization-class name=
            "oracle.adf.share.config.SiteCC"/>
          </match>
        </cust-config>
      </mds-config>
    </adf-mds-config>
  </adf-config>
```

Dynamic read-only view object

In the previous section, we were talking about the APIs for building entity-based view objects. This need not be the case always. You will build a SQL-based read-only view object on the fly if the dynamic UI that you are building needs to simply display the data without any edit capabilities. The following example illustrates the APIs to be used for building view object definitions on the fly:

```
//In application module implementation class

//Dynamic view object instance name used in this e.g
private static final String DYNAMIC_DETP_VO_INSTANCE =
"DynamicDeptVO";
/**
 * This method defined in application module. It
 * generates dynamic view object definition
 * at runtime
 */
public void createSQLBasedDepartmentViewObject() {
  //Remove view object if already exists
  ViewObject vo = findViewObject(DYNAMIC_DETP_VO_INSTANCE);
  if (vo != null)
    vo.remove();

  // Create a new "com.packtpub.adfguide.DepartmentView"
  //view definition
  ViewDefImpl deptViewDef = new
    ViewDefImpl("com.packtpub.adfguide.DepartmentView");
  // Define the names and types of the view attributes
  deptViewDef.addViewAttribute("DepartmentId",
    "DEPARTMENT_ID", Integer.class);
  deptViewDef.addViewAttribute("DepartmentName",
    "DEPARTMENT_NAME", String.class);
  deptViewDef.addViewAttribute("LocationId", "LOCATION_ID",
    Integer.class);
  // Define the SQL query that this view object will perform
  deptViewDef.setQuery("SELECT DEPARTMENT_ID," +
    "DEPARTMENT_NAME, LOCATION_ID FROM DEPARTMENTS");
  deptViewDef.setFullSql(true);
  deptViewDef.setBindingStyle(
    SQLBuilder.BINDING_STYLE_ORACLE_NAME);
  deptViewDef.resolveDefObject();

  // Create an instance of the new view definition named
  //"DynamicDeptVO "
  vo = createViewObject(DYNAMIC_DETP_VO_INSTANCE,
  deptViewDef);
}
```

Dynamic view object from the query statement

This is another approach for building a dynamic read-only view object. If you want to quickly build read-only view objects at runtime by using the SQL query statement, this is for you. In this case, the framework generates the attribute definitions based on the database table definition. This approach uses `createViewObjectFromQueryStmt()` defined on `oracle.jbo.server.ApplicationModuleImpl` for creating the view object instances from the query statement at runtime. Note that with this approach there will be a separate round trip to the database for reading table metadata in order to build the view object definition at runtime. An example is here:

```
  //In application module implementation class

  //Dynamic view object instance name used in this e.g
  private static final String DYNAMIC_DETP_VO_INSTANCE =
  "DynamicDeptVO";

  /**
   * This method is defined in application module. It creates
   * view object from query statement
   */
  public void createDynamicVOFromQuery() {
    //Remove view object if already exists
    ViewObject vo = findViewObject(DYNAMIC_DETP_VO_INSTANCE);
    if (vo != null)
    vo.remove();
    String query = "SELECT Departments.DEPARTMENT_ID, " +
    " Departments.DEPARTMENT_NAME, " +
    " Departments.MANAGER_ID, " +
    " Departments.LOCATION_ID " +
    "FROM DEPARTMENTS Departments";
    //Creates DynamicDeptVO instance from the query
    vo = createViewObjectFromQueryStmt(
    DYNAMIC_DETP_VO_INSTANCE,
    query);
  }
```

Building a binding definition for the dynamic view object

In the last section, we learned how to build a dynamic data model to be used in the UI. The next step in building a dynamic UI is to define the data bindings for the data model. You cannot build the data bindings by dropping the view object instance on to the page, because the view object instance that we use in the page does not exist at design time.

To add a binding definition for the dynamic view object, open the page definition file in the overview editor and then add the `<iterator>` binding definition as well as the `<tree>` binding definition. You can even manually enter the binding definitions by switching to the **Source** tab of the page definition file's editor window. You must be extra careful while doing so.

The following example illustrates the `<iterator>` definition for `DynamicDeptVO`. The `DynamicDeptVO` view object refers the view object instance added to the application module at runtime. In this example, the tree binding definition contains a dummy node, which is required as the framework expects at least one child node entry for the tree binding.

```
<executables>
  <variableIterator id="variables"/>
  <iterator Binds="DynamicDeptVO"
  DataControl="HRServiceAppModuleDataControl"
  id="DynamicDeptsIterator"/>
</executables>
<bindings>
  <tree IterBinding="DynamicDeptsIterator" id="DynamicDepts">
    <nodeDefinition Name="Dummy"></nodeDefinition>
    <!-- Dummy node is just a place holder node def -->
  </tree>
```

```
</bindings>
```

`HRServiceAppModuleDataControl` referred in the preceding iterator definition is defined in the `DataBindings.cpx` file as follows:

```
<Application ...> ...
  <dataControlUsages>
    <BC4JDataControl id="HRServiceAppModuleDataControl"
    Package="com.packtpub.adfguide.model"
    FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
    SupportsTransactions="true"
    SupportsFindMode="true" SupportsRangesize="true"
    SupportsResetState="true"
    SupportsSortCollection="true"
    Configuration="HRServiceAppModuleLocal" syncMode="Immediate"
    xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
  </dataControlUsages>
</Application>
```

Building a dynamic table UI component

The basic infrastructure for building a model-driven dynamic UI is in place by now. There are two solutions for rendering the UI for a dynamic model:

- ADF Faces offers dynamic UI components for rendering the data collection returned by the dynamic view object
- Another option is to use the `af:iterator` or `af:forEach` tag to iterate over a collection of objects and render the appropriate UI components dynamically

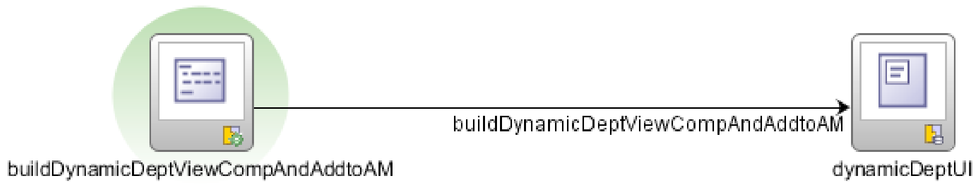
We are not discussing the dynamic UI component in this appendix. Refer to the Oracle Technology Network website to learn more about dynamic ADF Faces components (available online at `http://www.oracle.com/technetwork/developer-tools/adf/overview/index.html`).

The following is an example illustrating the usage of `<af:forEach>` for rendering the table UI at runtime. This uses the dynamic data binding definition that we created in the preceding section. In the following ADF Faces code snippet, `af:forEach` evaluates the EL specified for value attribute to identify the attribute definitions associated with the dynamic view object that we created in the previous step. Then, for each attribute present in the dynamic view object, `af:forEach` generates the `af:column` (along with `af:inputText`) element and adds it to the `af:table` component. The attribute values for dynamically added components are specified through EL.

```
<af:table rows="#{bindings.DynamicDepts.rangeSize}"
  fetchSize="#{bindings.DynamicDepts.rangeSize}"
  emptyText="#{bindings.DynamicDepts.viewable ?
  'No data to display.' : 'Access Denied.'}" var="row"
  rowBandingInterval="0"
  value="#{bindings.DynamicDepts.collectionModel}"
  selectedRowKeys=
  "#{bindings.DynamicDepts.collectionModel.selectedRow}"
  selectionListener=
  "#{bindings.DynamicDepts.collectionModel.makeCurrent}"
  rowSelection="single" id="t1"
  editingMode="clickToEdit">
  <af:forEach
    items="#{bindings.DynamicDeptsIterator.attributeDefs}"
    var="def">
    <af:column headerText="#{def.name}" sortable="true"
    sortProperty="#{def.name}" id="c1">
      <af:inputText
      value="#{row.bindings[def.name].inputValue}"
      maximumLength=
      "#{row.bindings[def.name].hints[def.name].precision}"
      id="fld1"/>
    </af:column>
  </af:forEach>
</af:table>
```

Using the method call activity to initialize the model for the dynamic UI

Defining an appropriate navigation case for the dynamic UI is equally important as building dynamic business components and a UI for displaying them. When you use dynamic business components for building a UI at runtime, you must make sure that the underlying business components are ready, before displaying them on the UI. A practical solution for such a use case is to make use of the method call activity in a task flow to hold the initialization logic for the UI. The following navigation case illustrates this idea:


buildDynamicDeptViewCompAndAddtoAM
buildDynamicDeptViewCompAndAddtoAM
dynamicDeptUI

In this example, the method activity `buildDynamicDeptViewCompAndAddtoAM` builds the dynamic view object component backed up by the entity object and adds it to the application module to be used by the client. This method call is marked as a default activity for the task flow. The business components created through the method call activity are used by the succeeding view activity in the task flow, which is marked as `dynamicDeptUI` in the diagram.

At runtime, when you execute this task flow, `buildDynamicDeptViewCompAndAddtoAM` creates a view object dynamically and navigates to the `dynamicDeptUI`.

`dynamicDeptUI` displays the data collection returned by the dynamically created view object component.

A working example demonstrating the APIs for building dynamic business components and a dynamic UI can be found in the example code available with this book. To access the example code for the dynamic UI, open the `ADFDevGuideAppendixSamples` workspace in JDeveloper and look for `dynamicUIMain.jsf` in the `ViewController` project.

Building composite view objects

The composite view objects help you to combine hierarchical results from two or more master detail view objects linked through a view link into a single composite view with flattened query retrieving the same result set. This feature is used by various analytical and data integration tools such as **Oracle Business Intelligence** (**OBIEE**) and **Oracle Data Integration** (**ODI**).

For example, consider the `Department` and `Employee` hierarchies built using the `Department` and `Employee` view objects, linked through a view link. You can build a composite view object at runtime by combining these two view objects as shown in the following code snippet:

```
//In application module implementation class

public void createCompositeDeptEmpVO() {

  // createCompositeViewDef is available in
  //ApplicationModuleImpl
  //Step 1 -Build Composite VO
    ViewDefImpl compVODef = (ViewDefImpl)
    createCompositeViewDef
```

```
("DeptEmpDetailCompVO", "model.DeptEmpDetail");
//Step 2- Add existing DepartmentVO to the composite VO
compVODef.addViewUsage("Dept", "model.DepartmentVO");
//Step 3- Add existing EmployeeVO to the composite VO
//with view link
compVODef.addViewUsage("Emp", "model.EmployeeVO",
"model.DeptToEmpViewLink", "EmployeeVO", "Dept");

//Step 4 -Include all attributes from Dept
compVODef.addAllRowAttributes("Dept");

//Step 5 -Add specifc attributes from Emp to avoid name
//collision as some might have already added in Step 4
compVODef.addRowAttribute("EmployeeId", "Emp",
"EmployeeId");
compVODef.addRowAttribute("FirstName", "Emp",
"FirstName");
compVODef.addRowAttribute("LastName", "Emp", "LastName");

//Give a different alaiase name to avoid name conflict
compVODef.addRowAttribute("EmpDepId", "Emp",
"DepartmentId");
compVODef.addRowAttribute("HireDate", "Emp", "HireDate");

// Step 6- Define a view criteria on composite VO
ViewCriteria vc = compVODef.createViewCriteria();
ViewCriteriaRow vcr = vc.createViewCriteriaRow();
ViewCriteriaItem vcItem =
vcr.ensureCriteriaItem("HireDate");
vcItem.setOperator(">=");
vcItem.setValue("2008-02-13");
vc.add(vcr);
compVODef.putViewCriteria("DeptEmpDetailVC", vc);

// Step 6 - Set Order By clause, if needed
compVODef.setOrderByClause("DepartmentName, FirstName");

// Step 7 - Resolve the definition and save it
compVODef.resolveDefObject();
//Write to XML
compVODef.writeXMLContents();
//Save it to hard disk
compVODef.saveXMLContents();
```

```
        //Create the composite VO instance
        ViewObjectImpl compDeptEmpVO = (ViewObjectImpl)
        createViewObject("DeptEmpDetail",
        compVODef.getFullName());

        //Step 8 - Apply VC and execute
        compDeptEmpVO.getViewCriteriaManager().
        setApplyViewCriteriaName("DeptEmpDetailVC");
        //Specify access mode
        compDeptEmpVO.setForwardOnly(true);

        //Test the  composite view object
        compDeptEmpVO.executeQuery();
        while (compDeptEmpVO.hasNext()) {
        Row r = compDeptEmpVO.next();
        // Process current row as per usecase
        }
    }
```

In the preceding example, the resulting composite (or flattened) view object `DeptEmpDetail` includes the data from both the `Department` and `Employee` view objects connected through an inner join. The joining criteria used in the flattened query for the composite view object is derived by using the view link definitions for the view objects. The client can include all or a subset of attributes from the participant view objects based on the use case requirement. It is the calling client's responsibility to provide attribute alias names to avoid name collisions by providing proper aliases names when attributes are added, using `addAllRowAttributes` on `oracle.jbo.server.ViewDefImpl`.

When a client builds composite view objects, passing multiple view objects programmatically, the following points are applicable:

- Only top-level and entity-based view objects can be used for building the composite view objects
- View objects used for building the composite view objects should be based on either normal SQL mode or declarative SQL mode
- View objects used for building the composite view objects can have the multiple, associated, or reference only entity usages
- The composite view object inherits the attribute-level hints from the underlying entity objects

Building application modules with no database connection

The requirement for building an application module with no database connection arises when your application module contains only programmatic entity objects and view objects, which do not use a database as their data store. To build an application module without any database connection, you may need to customize the core framework classes used by the application module runtime. The high-level steps are as follows:

1. Build an application module with a dummy database connection. This is because JDeveloper will not allow you to build an application module without using any database connection. You can delete it once you configure the application module to use the following custom components. Provide custom implementations for these components used by the application module:
    - **Pool class name**: You can sub-class the `oracle.jbo.common.ampool.ApplicationPoolImpl` class to add custom logic, and supply it to the application module. You can turn off the passivation support through this custom implementation.
    - **Connection strategy class**: You can sub-class `oracle.jbo.common.ampool.DefaultConnectionStrategy` to add custom logic, and supply it to the application module. You can use this custom implementation to set the configuration properties `jbo.dofailover` and `RequiresConnection` to `false` for the newly created application module instances.
    - **Session class**: You can sub-class `oracle.jbo.server.SessionImpl` to add the custom logic, and supply it to the application module. This custom implementation can be used for supplying custom `TransactionHandlerFactory`. This factory class is used for providing a custom `TransactionHandler`.

2. Configure the application module to use the preceding implementations by overriding the corresponding properties in the `bc4j.xcfg` file. Here is an example:

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/
configuration">
  <AppModuleConfigBag ApplicationName=
    "com.packtpub.nondb.NonDBAppModuleService">
    <AppModuleConfig DeployPlatform="LOCAL"
      jbo.project=
      "com.packtpub.adfguide.model.nondb.NonDBModel"
      name="NonDBAppModuleService"
```

```
            SessionClass=
            "com.packtpub.nondb.CustomSessionImpl"
            ApplicationName=
            "com.packtpub.nondb.NonDBAppModuleService">
            <AM-Pooling
            jbo.ampool.connectionstrategyclass=
            "com.packtpub.nondb.CustomConnectionStrategy"
            PoolClassName=
            "com.packtpub.nondb.CustomApplicationPoolImpl"/>
            <Database jbo.TypeMapEntries="OracleApps"
            jbo.locking.mode="optimistic"/>
            <Security AppModuleJndiName=
            "com.packtpub.nondb.NonDBAppModuleService"/>
            <Custom JDBCDataSource="java:comp/env/jdbc/HRDS"/>
          </AppModuleConfig>
        </AppModuleConfigBag>
      </BC4JConfig>
```

A working sample for an application module without any database connection can be found in the example code available with this book. To access this sample code, open the `ADFDevGuideCh6HRModel` work space in JDeveloper and look for the `NonDBModel` project.

Looking up the UI components from the component tree

The following code snippet illustrates how you can use a visitor pattern for getting hold of a specific UI component from the component tree at runtime. This is useful if you need to look up specific UI components in the component tree from the managed bean code. The following code snippet in a managed bean looks up for the `af:query` component with the client ID `qryId1` in the component tree. To learn more, refer to the API documentation available online at `http://javaserverfaces.java.net/nonav/docs/2.0/javadocs/javax/faces/component/UIComponentBase.html`.

```
//In managed bean class

private transient RichQuery deptQuery;

//This method gets the component reference for af:query with
//client id qryId1
public RichQuery getDeptQuery() {
  if (deptQuery == null){
    //'qryId1' - client Id for af:query component present
    // in JSF
    findRichQueryInView("qryId1");
  }
  return deptQuery;
}

/**
 *Finds the Query Component using the visitor pattern
 * @param id
 */
private void findRichQueryInView(String id) {

  FacesContext fctx = FacesContext.getCurrentInstance();
  UIViewRoot root = fctx.getViewRoot();
  root.invokeOnComponent(fctx, id, new ContextCallback() {
    public void invokeContextCallback(FacesContext
    facesContext, UIComponent uiComponent) {
      deptQuery = (RichQuery)uiComponent;
    }
  });
}
```

Summary

In this appendix, we discussed the lifecycle for a page with regions (task flow), transaction management in Fusion web application, dynamic business components, application modules with no database connection, and the dynamic UI creation techniques.