# ADF Framework Notes

# ADF Business Components – ADF – BC

Business Service

A business service is required to **query information from an underlying database** and **cache that data** while various operations are performed on it. It then has to **validate the data changes** and **complete the transaction by committing** the data back to the database.

* Defining application objects that map to database tables
* Managing data and business logic validation
* Creating application-specific views of the data
* Coordinating master/detail behavior of the business model based on foreign key relationships
* Providing default operations such as **commit**, **delete**, and **update** on the data model

The core features and functionality of the ADF Business Components framework are implemented in Java.

The developer generates and maintains metadata through property pages and declarative editors, and it is this metadata that drives the generic Java framework classes when the application runs.

When building ADF Business Components, JDeveloper queries the database and reads information about the tables on which the business service is to be based. For example, it discovers what columns are in the table, what data types, their precision, whether the field can be null, and whether it is a foreign key. This information is then encoded into XML files as part of the application project.

**Extending the Framework**

if you decide that the default framework feature for committing a transaction needs some application-specific code added to it, you can get JDeveloper to expose that framework class. This takes the form of JDeveloper creating a subclass of the framework class, into which you can add your own code.

**The Building Blocks of ADF Business Components**

ADF Business Components is itself based on three main building blocks, the, **entity object** the **view object**, and the **application module**.

The entity object maps directly to a database table and acts as an application cache for records from that table.

The view object defines an application-specific view of records queried into the underlying entity objects.

The application module, which is a collection of instances of view objects that defines the data model and transaction for a particular business task.

##### **Introduction to Entity Objects**

When you build an application based on a database table, your application needs somewhere to hold the records brought back from the database. It is the responsibility of the entity object to provide this functionality

As well as providing a data cache, the entity object performs the O/R mapping between the application and the database.

Validations

In entity object can be based on a database table, view, or synonym;

**Optional Java File** You can also optionally generate a Java class; by default, this would be called CustomersImpl.java for the Customers entity object, which exposes the methods the framework uses for things like setting attribute values and creating new records

Associations allow the framework to be aware that there is a relationship between, for example, Customers and Orders.

**Read-Only View Objects**

**Entity-Based View Object**

**View Object Based on Multiple Entity Objects**

 Orders entity object has an attribute SalesRepId. Given that your customers don’t know their sales representative by his employee number, this isn’t very meaningful in the context of the application. However, with a view object, you can define that some information comes from the Orders entity object and that the employee name comes from the Employees entity object as referenced by the SalesRepId.

A view object can also have attributes that are not based on entity attributes, but instead are based on expressions. These are called transient attributes and can be useful for features like calculated values. For example, a new view object attribute called TotalSalary, which isn’t based on an entity attribute, is the sum of the attributes Salary and Commission.

**Optional Java File**

View links, as the name suggests, link view objects together to implement behavior such as master/detail relationships.

##### **Introduction to Application Modules**

Having defined application-specific data views through view objects, the final step is to arrange instances of those view objects into a data model for a particular use case. This container of view object instances is called an application module. A typical application will contain one or more application modules, with the application module defining the transaction boundary for committing and rolling back changes made to the views of data contained within it.

Application modules can themselves contain other application modules, called nested application modules. This might be useful where a particular use case, as implemented by an application module, is also required within the transaction of a larger use case.

The application module can be thought of as a service façade or service interface to a consuming client, like a web page, that defines the public actions and data views for a particular application process or use case.

Entity Object

Therefore, as a broad generalization, you can think of each entity object as mimicking a database table inside your application. Or, to be more precise, each instance of the entity object represents a database row.

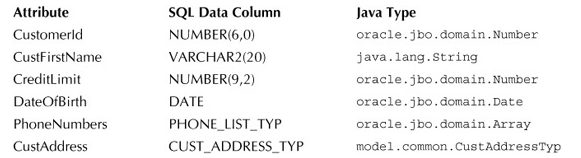
one-to-one mapping between a database table and an entity object

The entity object acts as a cache for data queried from the database and provides functionality to operate on this data

So, for example, setting attribute values, validating data, creating new records, and deleting records are all functions of the entity object.

**Synchronize with Database**

This feature does not support synchronizing all possible database changes, such as dropped database columns or data type changes. For these cases you have to manually delete the entity object attribute or change the attribute data type.



Some of the types, like VARCHAR2, map directly to standard Java classes. Others, like NUMBER, map to framework classes called domains, while more specific types, like CUST\_ADDRESS\_TYP, map to domains that are created within your project.

Domains are simply specialized Java classes that wrap the basic Java types and provide additional checks on the data assigned to that type. Oracle ADF provides a number of these classes within the oracle.jbo.domain package. For custom-defined database types like CUST\_ADDRESS\_TYP, JDeveloper automatically creates a domain for that type within the project, and if you look in the Application Navigator you will see domains created for these complex types in the Model project. You can double-click the domain to open the domain editor.

ou might decide to represent an e-mail address by setting up a domain that contains your own logic on how that domain validates any attribute of that type

**Attribute properties**

**Persistent** This property indicates that the attribute is mapped to a column in the database. You can also have attributes that are not based on database columns, which are called transient attributes. Transient attributes can be useful for representing derived or calculated data.

**Updatable** You can define when an entity object attribute can be updated: Always, While New, or Never. For example, you might want to define that an order date is only updatable when creating a new record, and cannot be changed once that record has been created.

**Internationalizing Your Control Hints**

for German the file would be ModelBundle\_de.properties, since “de” is the language identifier for German. You can then copy and paste the strings from the ModelBundle.properties file into the German version of the file and translate accordingly.

###### **View Accessors**

View accessors provide a convenient way of accessing the data associated with a view object from within an entity object. Why is this useful? Well, suppose you have an entity object validation rule that validates that a salary must be within a specific range for the person’s job. However, the salary ranges aren’t part of the entity object; they are part of a separate view object. The view accessor allows the source entity object to directly access the view object and read the salary range for that specified job. So you can think of the view accessor as a **parameterized connection to a view object.**

###### **Accessors**

Associations also give you the option of exposing accessors in the optional Java class that backs an entity object. These accessors allow you to programmatically “walk” between entity objects. For example, while writing code to perform a complex validation on the credit limit for a customer, the accessor allows you to programmatically access the instance of the Employees entity object for that customer.

Depending on the cardinality of the association, the accessors could use a framework class representing a single row of data or a collection of rows.

###### **Composition Association**

When creating an association between two entity objects, you can choose between two styles of relationship:

**Reference** The association references a destination entity object

**Contains** The destination entity can be thought of as a logical subpart of the source

##### **The Framework Classes**

###### **EntityDefImpl**

The functionality you’ve built in the entity objects so far is purely defined in XML. When you run your application, this XML information is used to instantiate a class oracle.jbo.server.EntityDefImpl, which is the definition of your entity object. There will be one instance of this class per unique entity object definition, and the methods available on this class generally relate to modifying the definition of the entity object.

###### **EntityCache**

As you would expect, an entity object has to cache rows of data coming back from the database. An instance of oracle.jbo.server.EntityCache is responsible for providing this cache. You can think of EntityCache as a drawer of a filing cabinet where each entity object definition has its own drawer. Again, at this stage you will probably never need to override or directly call methods on this class.

###### **EntityImpl**

The EntityImpl class is the entity object framework class you will come into contact with most often. Each instantiation of oracle.jbo.server.EntityImpl equates to a single row of data, and, continuing the analogy of a filing cabinet drawer, it is like a single sheet of paper held in the filing cabinet drawer. When you create a new row of data, the framework instantiates a new instance of EntityImpl. Typical methods available on this class include creating a single row, performing a DML operation on a row, and setting attribute values of a row.

**Accessors** Either by using the Java editor or the Structure window, you should see pairs of set and get methods for each attribute in the entity object. For example:

public String getCustLastName() {  
    return (String)getAttributeInternal(CUSTLASTNAME);  
}  
public void setCustLastName(String value) {  
    setAttributeInternal(CUSTLASTNAME, value);  
}

public String getCustLastName() {  
        String lastName = (String)getAttributeInternal(CUSTLASTNAME);  
        if (lastName != null) {  
            return lastName.toUpperCase();  
        } else {  
            return lastName;  
        }  
    }

**create() Method** In the Select Java Options dialog, you chose to expose the following method:

protected void create(AttributeList attributeList) {  
    super.create(attributeList);  
}

This is the method the framework will use when a new customer row is created, allowing you to augment the default behavior, such as setting default values.

**remove() Method** You also chose to expose the following method:

public void remove() {  
    super.remove();  
}

This allows you to add your own code to the framework method called when a customer row is removed. For example, you can add validation logic to prevent the removal of a record if a certain condition is met.

protected void doDML(int operation, TransactionEvent e) {  
    super.doDML(operation, e);   
}

This method allows you to write code at the point at which cached rows are inserted, updated, or deleted from the database. This might be an ideal point for adding extra logic before data changes are passed to the database. The operation parameter allows the method to distinguish between DML\_UPDATE,DML\_INSERT, and DML\_DELETE.

**Reading a Sequence in EntityImpl**

protected Number nextVal(String sequenceName) {  
    SequenceImpl s = new SequenceImpl(sequenceName, getDBTransaction());  
    return s.getSequenceNumber();   
}

protected void create(AttributeList attributeList) {  
    super.create(attributeList);  
    setCustomerId(nextVal("Customer\_Seq"));  
}

Usimng Groovy

“adf.object.nextVal(“Customer\_seq”);”

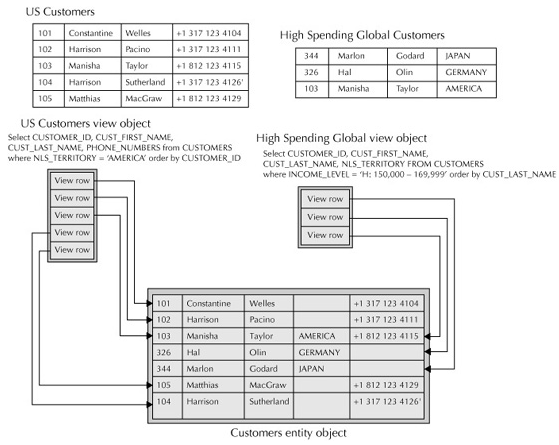
View Object

For each view object, the framework automatically creates a runtime instance of a view object cache. This view object cache doesn’t hold any rows of data as such, but is instead a collection of pointers to rows in the underlying entity object. It is this view object cache that is bound to a UI item, like a table, so that you see the queried records and attributes as defined by your view object.

 Sometimes the view object caches contain pointers to the same entity object row. This means that if a row is updated through one view, then the change is automatically seen in the other view.

As well as defining an application-specific view of data, view objects are linked together by view links to define relationships between those views

Thus, the combination of view objects, linked together by view links, becomes the building blocks of your application data model

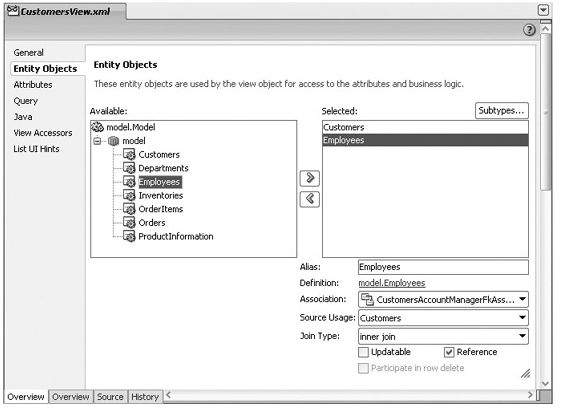


Read-only VO

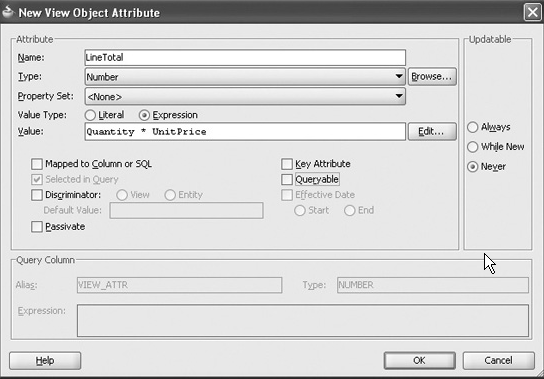
the current thinking is that for database data that never changes, there is actually very little to be gained from, and in fact some drawbacks to, using a view object with no underlying entity object. If you do require a view object that represents read-only data, create the entity object and the view object, and set the view object to be read-only.

Static List

The strings you just entered as static values are stored in a translatable resource bundle, so you can easily translate them.



Calculated Attribute



**Accessors**

**VO Classes**

**ViewDefImpl**

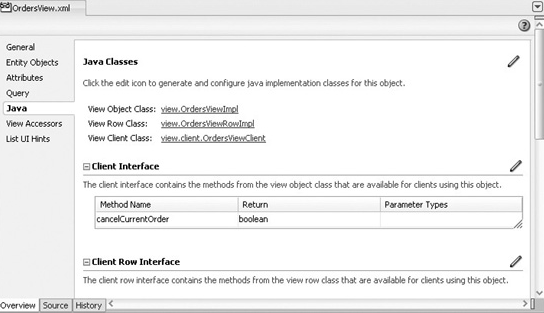
**ViewObjectImpl**

This class is responsible for managing the view object query and the rows queried. Typical methods available on this class include setting bind variables, changing a where clause, executing a query, and navigating through the view object rows.

**ViewRowimpl**

Each instance of ViewObjectImpl manages a set of queried rows, each row being a pointer to the underlying entity object instance. Each of these view object rows is an instance of oracle.jbo.server.ViewRowImpl. ViewRowImpl provides methods for getting and setting view object attribute values.

public boolean cancelCurrentOrder() {   
  
    OrdersViewRowImpl cr = (OrdersViewRowImpl)getCurrentRow();  
  
    //Check to see if the order is currently "live" (orderStatus=1)  
    if (cr.getOrderStatus().equals(new Number(1))) {  
        //Set the order to cancelled by customer (status 3)  
        cr.setOrderStatus(new Number(3));  
  
        //Now assign to a new customer care rep.  
        cr.setSalesRepId(new Number(101));  
        return true;  
  
    }  
    return false;  
}



### ****THE APPLICATION MODULE****

An application module represents a specific business use case, within which are instances of view objects required for that use case. That could be a use case like “manage customer order,” “maintain employee details,” or “inventory restock.” Each of these use cases would typically be represented by a separate application module.

Remember, an application module contains instances of view objects and view links. This means that there may be many instances of the same view object. For example, there may be an instance of OrdersView representing all orders, and an instance representing orders for a specific customer. Each instance has its own current record indicator and state information.

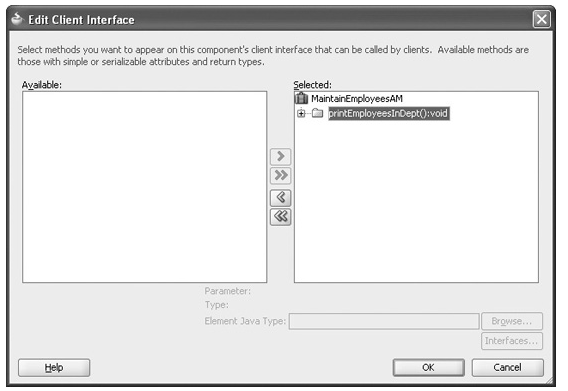
###### **ApplicationModuleDefImpl**

Just like for the other ADF Business Components artifacts such as view objects and entity objects, the definition of the artifact is held in XML. At runtime a single Java class is instantiated for each application module definition to hold information about that application module definition. This class isoracle.jbo.server.ApplicationModuleDefImpl.

###### **ApplicationModuleImpl**

Each instantiation of oracle.jbo.server.ApplicationModuleImpl represents an instance of an application module. Typical methods available in this class include getting information about the database connection and accessing and managing the view object instances.

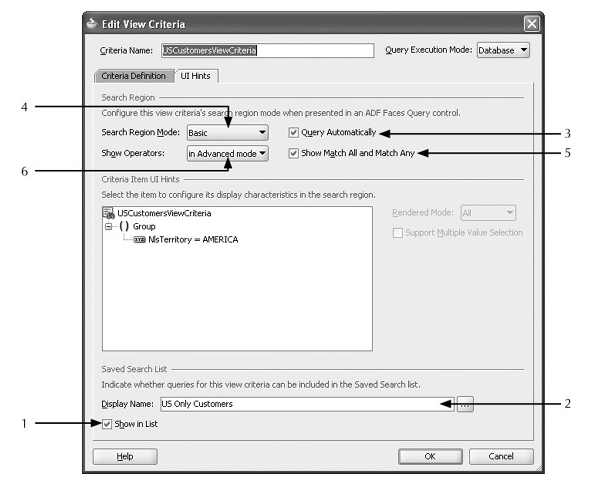
public void printEmployeesInDept() {  
    RowSetIterator rsi = getEmployeesView1().createRowSetIterator(null);  
    while (rsi.hasNext()) {  
        Row r = rsi.next();  
        System.out.println(r.getAttribute("LastName"));  
    }  
    rsi.closeRowSetIterator();  
}



In doing this, you are telling JDeveloper that you want this method to be available through the application module interface.

You may notice in the Application Navigator that JDeveloper has created a class to support this behavior.

#### ****View Criteria****



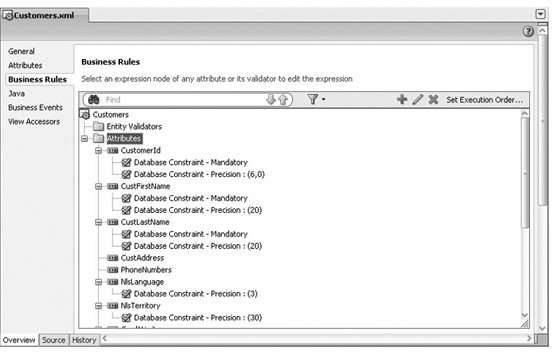
If the user applies a view criteria from the search panel, then there is no way of unsetting the use of that view criteria without selecting another view criteria. To avoid this you may want to define a view criteria that shows all records defined by the view object. This allows you to “undo” any applied view criteria

##### **Model Driven List of Values**

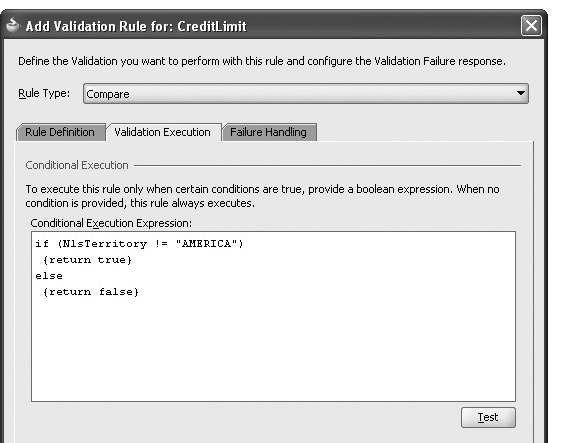
#### ****Dependent List of Values****

Using View Accessors and Bind Variables

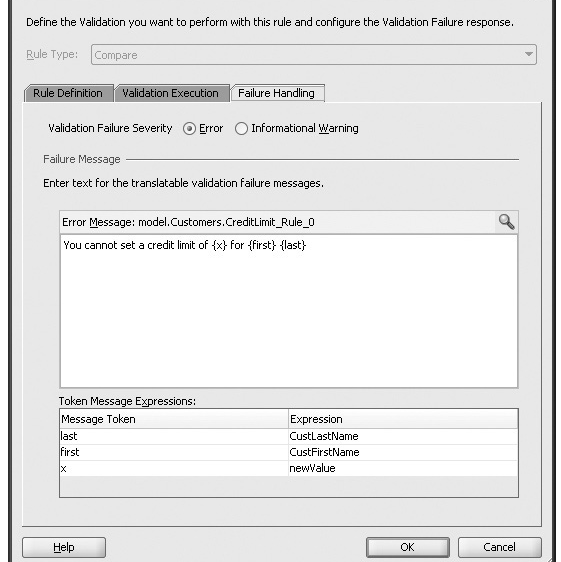
### ****BUSINESS SERVICE VALIDATION****



DB validation reverse-engineered



Groovy



Validation Message

Groovy is a Java-like scripting notation that runs inside a JVM and has access to all the Java APIs, any custom classes and methods you write, as well as a few APIs of its own. Also, because it is interpreted at runtime, rather than being compiled, it can be customized on an application-to-application basis.

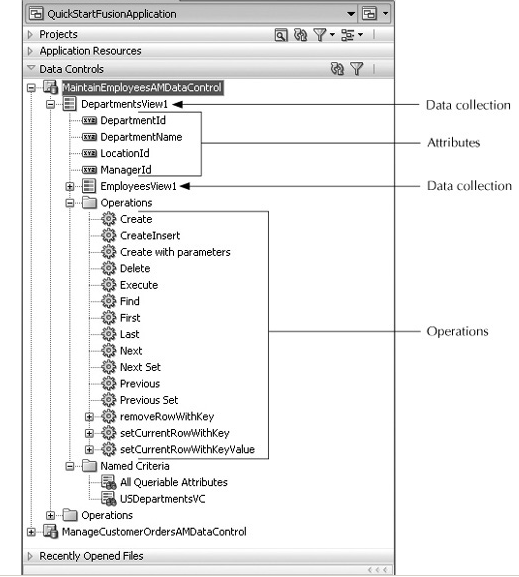
public boolean validateCustomers() {  
    String job = (String)getEmployees().getAttribute("JobId");  
    if (getIncomeLevel().equalsIgnoreCase("L: 300,000 and above") &&  
        !job.equals("AD\_PRES")) {  
        return false;  
    }  
    return true;  
}

<https://www.safaribooksonline.com/library/view/quick-start-guide/9780071744287/ch09.html>

## ADF Model

ADF Model provides a number of significant benefits to the developer. The first is that it implements the physical binding of a UI component to a business service. ADF Model enables you to do this by providing a framework feature, called a **binding container**, that acts as the glue between a page and the business service.

The second benefit is that ADF Model provides a layer of abstraction over the underlying implementation.



There are essentially two halves to the binding problem:

the business service has to be exposed in a consistent way,   
and a UI component has to know how to bind to that abstraction.

a UI component such as a table must be bound to many rows and columns of data, whereas a text field is bound to only a single piece of data

ADF Model provides the two halves of the binding solution: data controls and bindings.

##### **Data Controls**

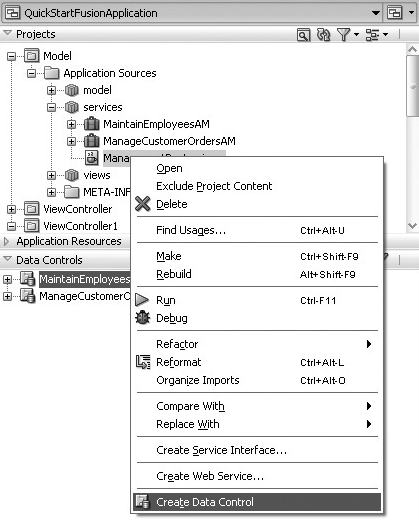
Data controls provide the abstraction of the business service. Like many artifacts in Oracle ADF, data controls are simply XML definitions of the underlying business services. The XML describes the business service, including what attributes it has and any operations it exposes.

Because the definition of a business service developed using ADF Business Components, specifically the application module, is already XML, ADF Model knows how to work with that definition. Thus, ADF Business Components is already “data control ready.” All the information defined in the application module, such as the view object instances, their attributes, and their operations, define the façade for the business service in a way that ADF Model already understands.

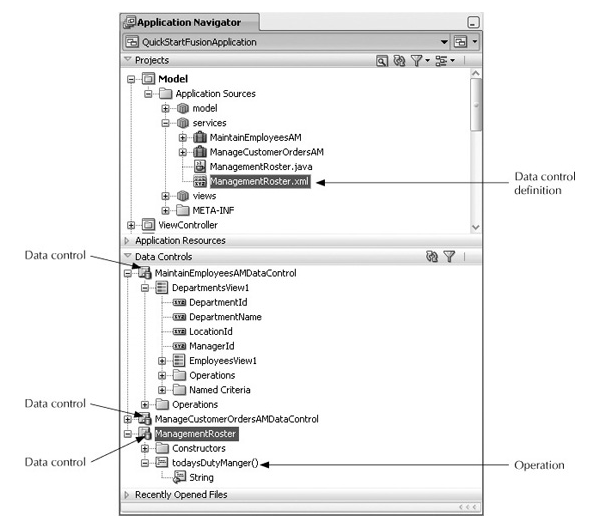
###### **Creating a Data Control**

Suppose you create some part of your business service and don’t use ADF Business Components; how do you create a data control for that? As you might expect, JDeveloper takes care of this for you.

Create a Java Class in Model and expose it as Data Control



This create an XML file which is data control definition



You may need to click the Refresh button on the Data Controls panel to see the new data control.

each of the application modules created in the business service is represented as a data control.

The file DataControls.dcx is also created in your Model project. This file acts as an index for all non–ADF Business Components data controls.

##### **Binding**

So with data controls you have one half of the solution, providing a common abstraction of the business services. The second part of the puzzle is the binding, which is responsible for connecting the UI component to the data control.

The first thing to recognize is that there are different flavors of binding, because there are different types of UI components; a button requires a different kind of binding than a table component requires, since one is binding to an action and the other to a collection of rows of data.

**Types of Binding**

**Attribute binding** Used to bind to a single attribute value such as DepartmentId, typically using a component like an input text or output text field.

**Tree binding** Used to bind collections of data, such as Departments, using components like a tree control or a table.

**Action binding** Used for binding the default actions of an ADF Business Components service such as Create, Delete, Next, and Previous. This would typically be bound to a push button or a link.

**Method binding** Used to bind to any business service function implemented as a Java method such as an application module method or a Java class method. This would typically be bound to a push button or a link.

**List binding** Used to bind data lists, such as a choice list or combo box.

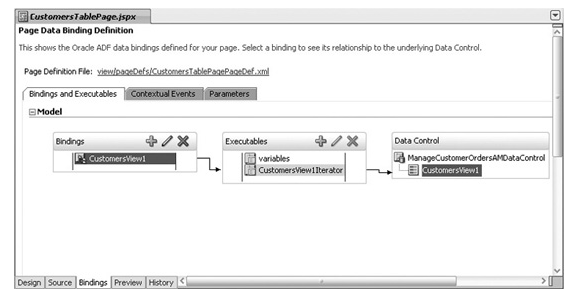
**ListofValues binding** Used to bind data to a list of values component. This component typically involves a pop-up dialog with search capabilities.

explicitly create a binding

you might be viewing a table of data and wish to have an attribute of the currently selected record displayed as the title of a tabbed panel. The table requires a tree binding since it is a collection of data; however, the title for the tabbed panel requires only a single value. In this case, it would make sense to create an attribute binding to the same data control as the table is using.

The first thing you need to do is to create a new page to display a table of customers. Create a new JSF page and drop on a Panel Tabbed component and inside the Panel Tabbed component drop a Panel Collection. Inside the Panel Collection, drag and drop the customers data collection from the Data Controls panel as a read-only table, as shown in [Figure 11-3](https://www.safaribooksonline.com/library/view/quick-start-guide/9780071744287/ch11.html#ch11fig3). You can choose which attributes you want to display.

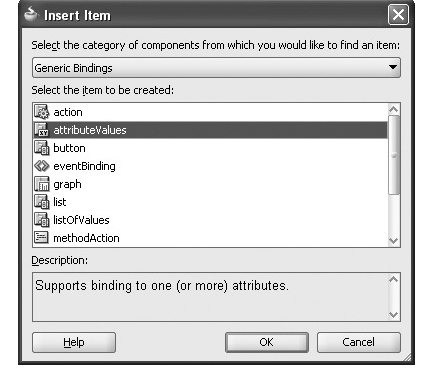
You will also notice that there is a middle column, Executables. For the moment you can assume one executable per collection. This executable is an iterator, or pointer to the current data object.

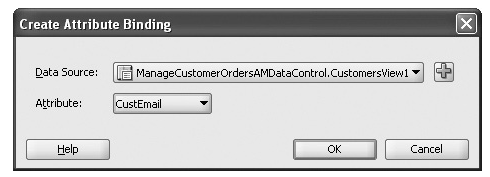


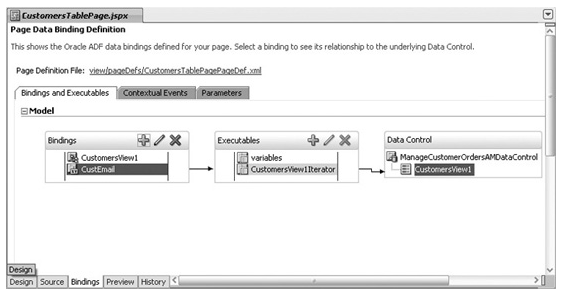
You should see two new files created in the ViewController project when you drag and drop from the Data Controls panel. There is a file <PageName>PageDef.xml for each JSF page. This defines the bindings for each page and is the file you are viewing when you select the Bindings tab as shown in[*Figure 11-4*](https://www.safaribooksonline.com/library/view/quick-start-guide/9780071744287/ch11.html#ch11fig4).

There is also a file **DataBindings.cpx**, which holds information about which JSF page is mapped to which PageDef.

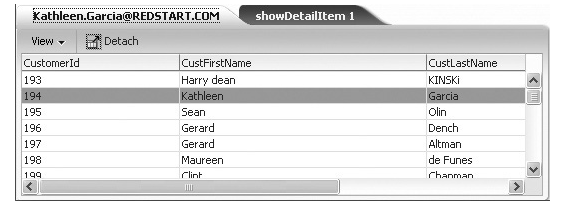
To create a new binding, click the green plus sign in Bindings. In the Insert Item dialog, select “Generic Bindings, select “attributeValues,” and click OK.







Because the tree binding and the attribute value binding are referencing the same data control through the same executable, the CustEmail binding will correctly reference the currently selected data object. The executable is maintaining the pointer to the current row.



In order for the text in the tabbed panel to be refreshed, an action has to happen on the page, such as switching tabs. However, if you want to force the refresh automatically, select the CustEmail binding and set itsChangeEventPolicy property to “ppr.” This is a feature called partial page refresh, which is covered later in the book. This will force the component that references the binding to automatically refresh when the binding has been updated.