

# CHAPTER 4

## Building ADF Business Components



This chapter will take you through the concepts, building blocks, and steps to create a first-cut business service based on ADF Business Components. As a Fusion developer, you have the challenge of building a business service based on a number of tables in an Oracle database.

**NOTE**

*The examples used in this book are based on the OE (Order Entry) Sample schema available with the Oracle database.*

## The Goals for ADF Business Components

As outlined in Chapter 2, there are a number of goals for a business services framework like ADF Business Components. At the most fundamental level, 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. This use case can be broken down into the following high-level tasks:

- 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

ADF Business Components provides these core features within the framework in a generic way, allowing these generic services to be adapted for the application-specific case. However, before embarking on creating a business service based on ADF Business Components, let's first look behind the scenes at the various ADF Business Components building blocks.

## How ADF Business Components Works

The core features and functionality of the ADF Business Components framework are implemented in Java. For the most part, the Java classes responsible for the previously noted features, and more, are hidden from the Fusion developer. Instead, 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.

### Building a Default Business Service

JDeveloper offers a number of different ways to build ADF Business Components, including visually through modelers or by stepping through wizards. Regardless of the actual method you choose, JDeveloper does the same job for you behind the scenes. 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.

If you want to change any of this information, such as the order in which the data is retrieved or whether an attribute is updateable, you can do this by setting properties exposed in the ADF Business Components editors. It is these editors that give the developer a productive and declarative way of maintaining the metadata underpinning ADF Business Components.

Upon running the business services, either through an application or the built-in ADF Business Components tester, the framework reads the application's XML files to create application-specific instances of the more generic framework classes.

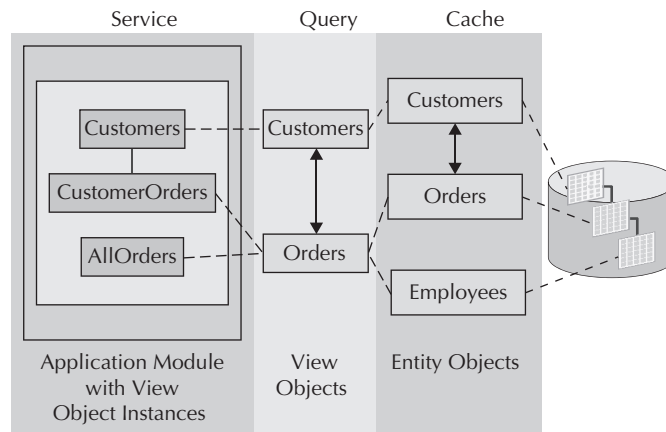
### Extending the Framework

Of course, one of the key features of ADF Business Components is that the framework classes can themselves be exposed to the developer so that application-specific code can be added to the more generic framework classes. For example, 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, as shown in Figure 4-1: the entity object, the view object, and the application module. You might be thinking, "What, more layers?" and you would be right; however, each layer of ADF Business Components has a well-defined role, and being separate and distinct makes it much more flexible and powerful.

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 final building block is a container called an application module, which is a collection of instances of view objects that defines the data model and transaction for a particular business task.



**FIGURE 4-1.** ADF Business Components consists of entity objects, view objects, and application modules.

## 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.

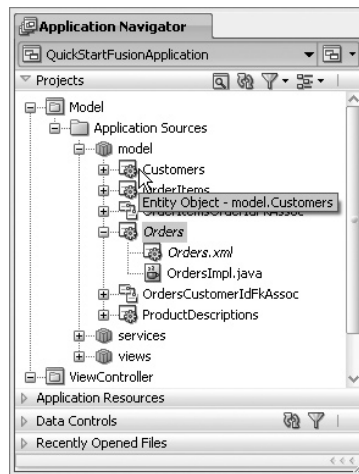
As well as providing a data cache, the entity object performs the O/R mapping between the application and the database. For example, a Customers entity object would map to the underlying Customers table in the database. Furthermore, because the entity object is the object in which application data is held and manipulated, it is also the place where business and data validation logic is implemented.

For each of the columns in the underlying table, the entity object will typically contain an attribute that maps to that column and reflects its characteristics, like data type, precision, and whether it allows null values. Of course, an entity object doesn't require an attribute to be mapped to every column in the underlying database table. If your application never needs to manipulate, access, or display a particular column from the database, then you can remove the corresponding attribute from the entity object.

An entity object can be based on a database table, view, or synonym; however, in this book the primary use case will be an entity object based on a database table, and so you can think of an entity object as being like a local copy of the table inside the application.

### Behind the Scenes of an Entity Object

The entity object is displayed in the Application Navigator as a single ADF Business Components artifact. Double-click the entity object to edit it in the appropriate editor. You can also expand the node in the Application Navigator to view the entity object implementation files.

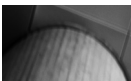


The core implementation of an entity object is through a single XML file; for example, Customers.xml in the case of an entity object called Customers. This alone is enough for the framework to provide rich business service functionality based on the Customers table. If you look at the source of the XML file that implements the entity object, you should be able to recognize the information read from the database that is used to implement specific entity object behavior. For example, the CustomerId attribute maps to the CUSTOMER\_ID database column, which is a primary key and should be not null.

```

Customers.xml
Find
DBObjectName="CUSTOMERS"
AliasName="Customers"
BindingStyle="OracleName"
UseGlueCode="false"
xmlns:validation="http://xmlns.oracle.com/adfm/validation">
<DesignTime>
  <Attr Name="_codeGenFlag2" Value="Access"/>
  <AttrArray Name="_publishEvents"/>
</DesignTime>
<Attribute
  Name="CustomerId" ← Attribute name
  IsNotNull="true" ← Is not null
  Precision="6"
  Scale="0"
  ColumnName="CUSTOMER_ID" ← Database column name
  SQLType="NUMERIC"
  Type="oracle.jbo.domain.Number"
  ColumnType="NUMBER"
  TableName="CUSTOMERS"
  PrimaryKey="true">
  <DesignTime>
    <Attr Name="_DisplaySize" Value="22"/>
  </DesignTime>
</Attribute>
<Attribute
  Name="CustFirstName"
  IsNotNull="true"
  Precision="20"
  ColumnName="CUST_FIRST_NAME"

```



#### NOTE

As a developer, you will probably never have to look at or directly edit the XML source; however, it is useful to know that this is how ADF Business Components implements your business service functionality.

**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. This is covered in more detail in Chapter 5 but is mentioned here to show that you can go beyond the declarative features of the framework to extend the default functionality, as you require.

### Associations

You may have noticed that Figure 4-1 shows arrowed lines between the first two entity objects. These lines represent associations that define a relationship between entity objects, usually based on the foreign key constraints defined in the underlying tables. Associations allow the framework to be aware that there is a relationship between, for example, Customers and Orders.

### Introduction to View Objects

The main role of the view object is to give an application-specific view of records queried into the underlying entity objects.

For example, a Customers entity object maps to the Customers table, and this table contains all customer records. However, if your application is applicable only for U.S. customers, you will want to define that the business service should only retrieve the records of U.S. customers. Furthermore, the application does not make use of some data and so will not expose date of birth and marital status.

The view object is responsible for providing this “shaping” of data for the application by defining an SQL statement that selects and orders only the necessary data into the underlying entity objects. A view object can be based on none, one, or many entity objects.

### Read-Only View Objects

A view object that is not based on an entity object is called a read-only view object. A read-only view object can be a static list of values defined at design time or a read-only list selected directly from a database.

Read-only view objects might be used for a view of data that is never likely to change, such as a list of all countries selected from a database table, or a static list of salutations (Mr., Mrs., Miss, and so forth).

### Entity-Based View Object

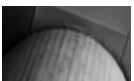
The most typical example of a view object is one that is based on a single entity object. So, for a Customers entity object, you would typically have a CustomersView view object that selects all, or some, of the columns into the underlying entity object.

You can also define multiple view objects based on the same entity object. For example, for stock control, you want a view of all products and their suppliers. But elsewhere in your application, in order to push the sale of certain products, you might require a view of all products that are reaching their end-of-life date. In both cases you have the same underlying source of product data in the entity object, but different application-specific views. Because both views are based on the same entity object, if a product is withdrawn by a supplier, then it will not appear in the view of end-of-life products since both view objects are pointing at the same source of data cached in the entity object.

### View Object Based on Multiple Entity Objects

With a view object based on many entity objects, you join together information from different database tables into an application-specific shape. This is particularly useful when you want to bring back information defined by a foreign key lookup. For example, the 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.

In all three cases previously described, a view object allows you to define an application-specific view of your data.

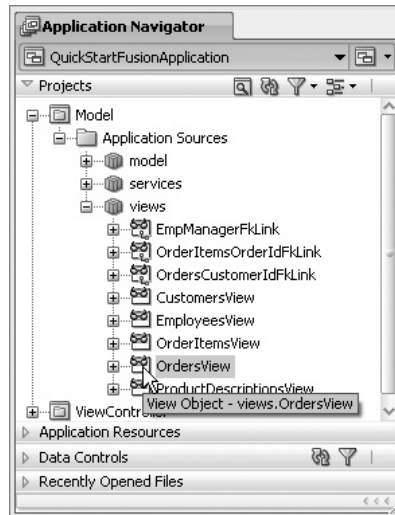


#### NOTE

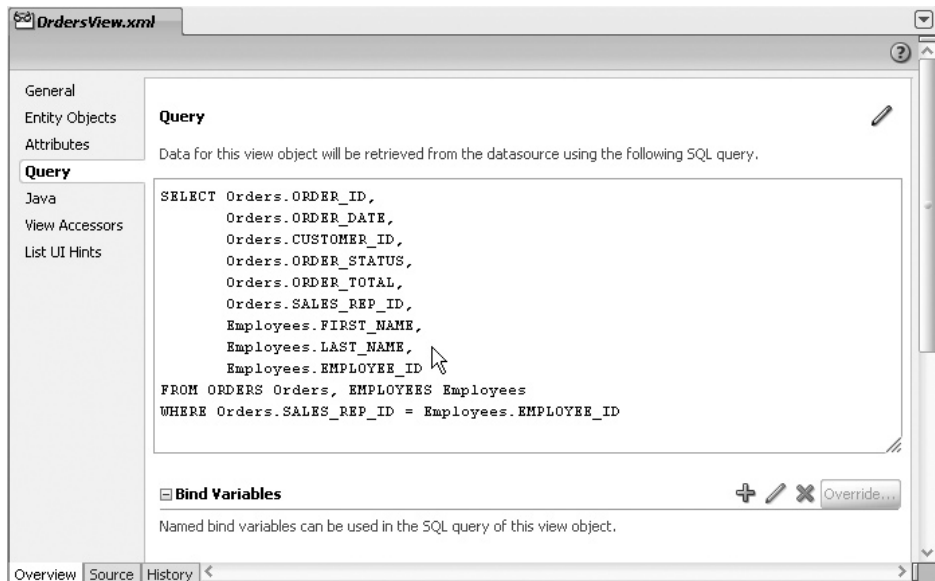
*A view object can also have attributes that are not based on entity attributes, but instead are based on expressions. These are called transient attributes and can be useful for features like calculated values. For example, a new view object attribute called TotalSalary, which isn't based on an entity attribute, is the sum of the attributes Salary and Commission. This is covered in more detail in Chapter 6.*

## Behind the Scenes of a View Object

Just like the entity object, the view object is displayed in the Application Navigator as a single ADF Business Components artifact, symbolized by the reading spectacles icon. Double-clicking the view object opens it in its associated editor.



The view object is essentially a `SELECT` statement defined in an XML file, with the view object editor providing a simple and intuitive way of editing information about the view object to build up this `SELECT` statement. As well as defining the attributes that make up the view object, you can also define information such as a `WHERE` and `ORDER BY` clause and even edit the `SELECT` statement directly if you feel comfortable writing SQL.



**Optional Java File** As with an entity object, you can extend the functionality of the view object by exposing and augmenting the underlying framework classes. You might choose to do this if you wish to implement functionality such as programmatically manipulating the `where` clause or creating a custom method that performs some application function on a view of data. This is also covered in more detail in Chapter 6.

## View Links

Referring back to Figure 4-1, notice that an arrowed line also appears between the view objects. This represents a view link. View links, as the name suggests, link view objects together to implement behavior such as master/detail relationships. For example, suppose you have a view of customers and a view of orders, but are really only interested in seeing the orders for a selected customer. When the orders view object is linked to the Customers view object with a view link, the framework automatically restricts the view of orders based on the customer.



### TIP

*While the view object isn't necessarily driven by the user interface, it can be a useful way of conceptualizing which view objects you need and what information they need to encapsulate.*

## 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.

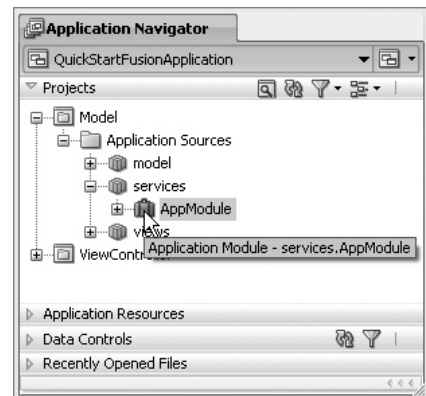
Application modules can themselves contain other application modules, called nested application modules. This might be useful where a particular use case, as implemented by an application module, is also required within the transaction of a larger use case.

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.

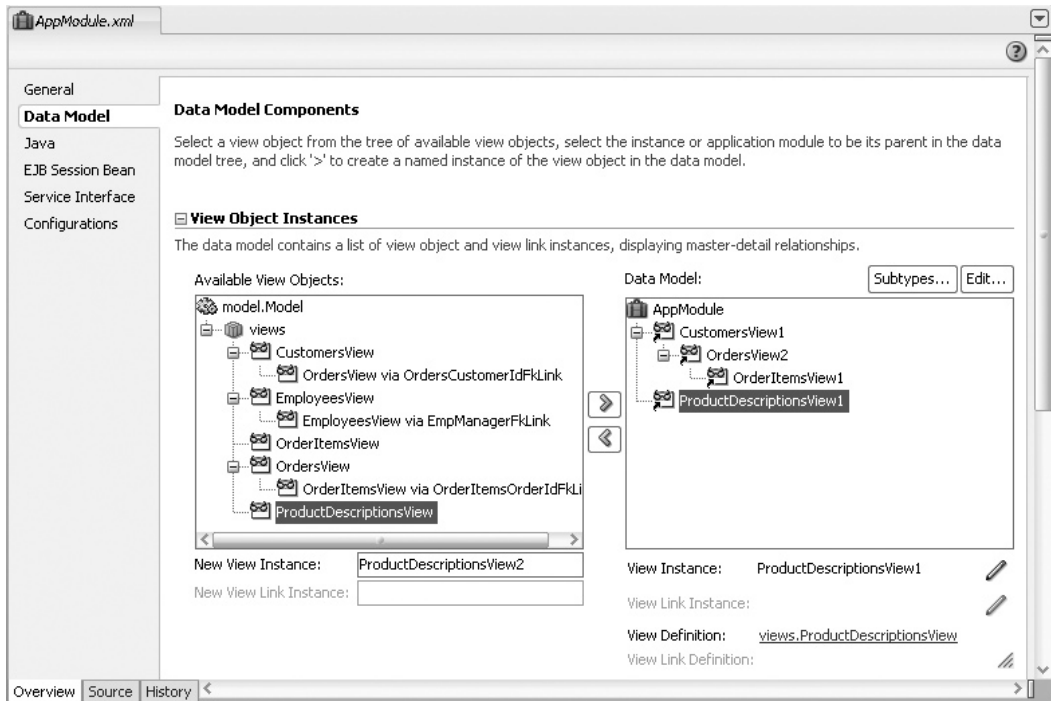
### Behind the Scenes of an Application Module

The application module is displayed in the Application Navigator as a single ADF Business Components artifact. Double-click it to open the application module editor, where you can see the view objects within the project and can shuttle them over to define your application module.

As with the previously described ADF Business Components artifacts, the definition of an application module is through XML; however, the application



module editor provides a simple and declarative way of defining the view object instances and methods that make up an application module.



### Optional Java File

The framework class that implements the application module functionality already provides methods for committing and rolling back the transaction. However, if you wish to augment that functionality, you can generate the application module implementation class and add your own code to this class.

You can also define application module–level methods that you want to expose to the consuming client. This is covered in more detail in Chapter 7.

## First-Cut Business Service Development

Now that you are armed with an understanding of the building blocks of ADF Business Components, the next step is to put that knowledge into action and build ADF Business Components based on database tables. There are a number of different ways to build ADF Business Components; however, JDeveloper provides an incredibly powerful wizard that creates entity objects, associations, view objects, view links, and an application module, based on a number of database tables, all in one go. This is a great way of getting a first cut of your business services.

### Application Schema for This Book

The goal of this book is to give you a quick start on how to build a Fusion application, and as such, many of the features of JDeveloper and Oracle ADF are explained in the context of building a Fusion application. However, the book doesn't aim to hold your hand through every keypress or demonstrate every feature within that sample application. Instead, the aim is to explain the concepts, how they can be built in JDeveloper, and then how they can be used in a typical Fusion application. You can then explore, build, and experiment at your convenience—after all, hands-on is the best way to learn.

Where an application example is used, it is based on the Oracle OE (Order Entry) schema that is available with the Oracle database. This schema includes a number of tables for implementing an order entry system based around customers, orders, order items, and products.

The scenario is that the application is being built to manage customer orders. The application should allow the creation, viewing, and editing of customers and their order information. Each customer can have zero or more orders, and each order is made up of one or more order items. The application has a number of pages for viewing and displaying data that is presented in a number of different ways, including data entry forms, tables, and graphs. The application also includes business rules to validating data, and features such as search facilities and lists of values.

A completed application that demonstrates the features discussed in this book is available to download from <https://qsfusionsample.samplecode.oracle.com/>.

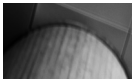
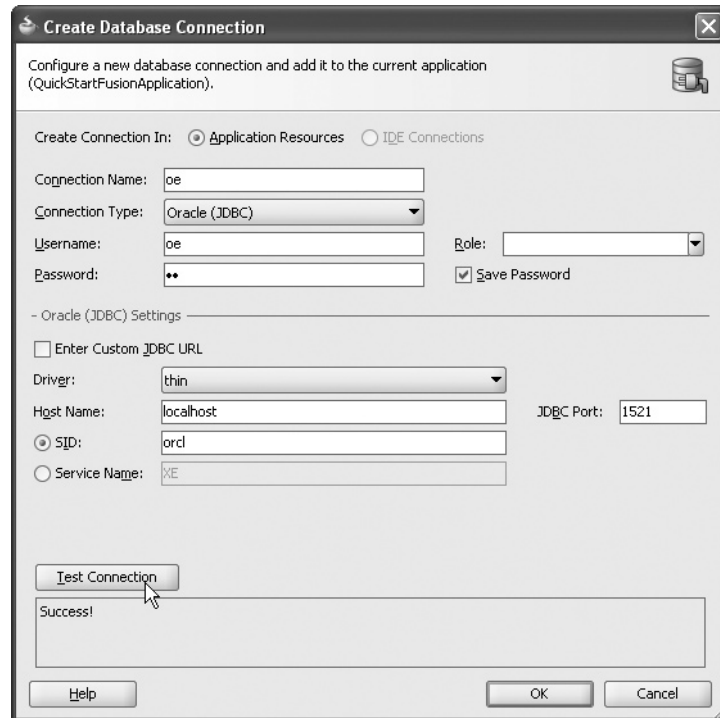
## Create Business Components from Tables

Creating new ADF Business Components from database tables is reasonably quick and intuitive. Assuming you've already created an empty application based on the Fusion Web Application template, as described in Chapter 3, select the Model project and then select **File | New**. In the New Gallery dialog, select **ADF Business Components** and then **Business Components from Tables**. This launches the Create Business Components from Tables wizard, which steps you through creating entity objects, updatable view objects, read-only view objects, and an application module.

### Connecting to the Database

If you haven't yet created a connection to your database, the Initialize Business Components Project dialog prompts you to create a connection to the database. Click the green plus sign, then enter a name for the connection in the *Connection Name* field. Enter values for the *Username*,

*Password, Host Name, and SID, and then click Test Connection to confirm the connection to the database.*



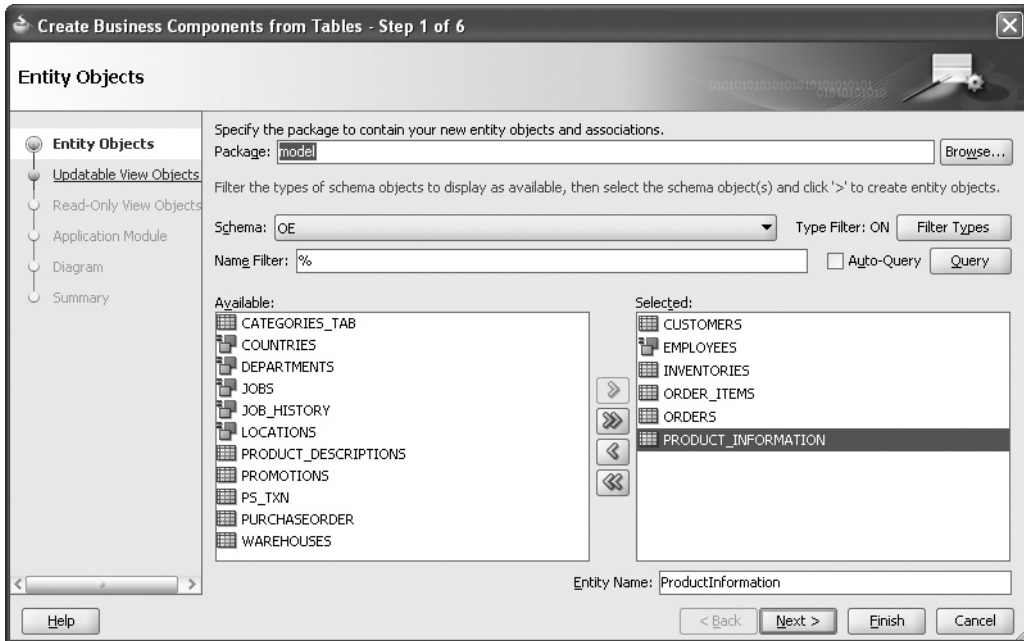
#### TIP

*You can alternatively create a database connection by selecting **File | New** and, in the New Gallery dialog, selecting **Connections** and then **Database Connection**. When you create a database connection, you can choose to create the connection as part of the application or associate the connection with the IDE. An IDE database connection adds the connection to the Resource Palette window and provides that connection for use within the IDE. You can then right-click a database connection in the Resource Palette to browse the database or add that connection to the current project.*

## Selecting Tables for ADF Business Components

Now that you have established a connection to the database, the Create Business Components from Tables wizard prompts you to select the tables for which you want to build entity objects.

Click Query to view the tables in the database. You may also want to click Filter Types and choose to view synonyms as well as tables.



Shuttle over the Customers, Employees, Inventories, Order\_Items, Orders, and Product\_Information tables. By default, the entities will be named as per the tables but using CamelCase, but you can change the entity object name on this page. You can also choose a package name to help manage your entity objects. Click Next.

For updateable view objects, shuttle over all the entity objects from the list and enter a different package name for the view objects. In this step you are creating a view object for every entity object. The next step is for read-only view objects, which we are not creating in this case, so you will move to the next step.

Click Next to move to Step 4 of 6, where you can select the *Application Module* check box and enter a package name. This will create an application module based on all of the view objects. You can now click Finish or, optionally, click Next and select *Business Components Diagram* to generate a diagram of the ADF Business Components you have just created.

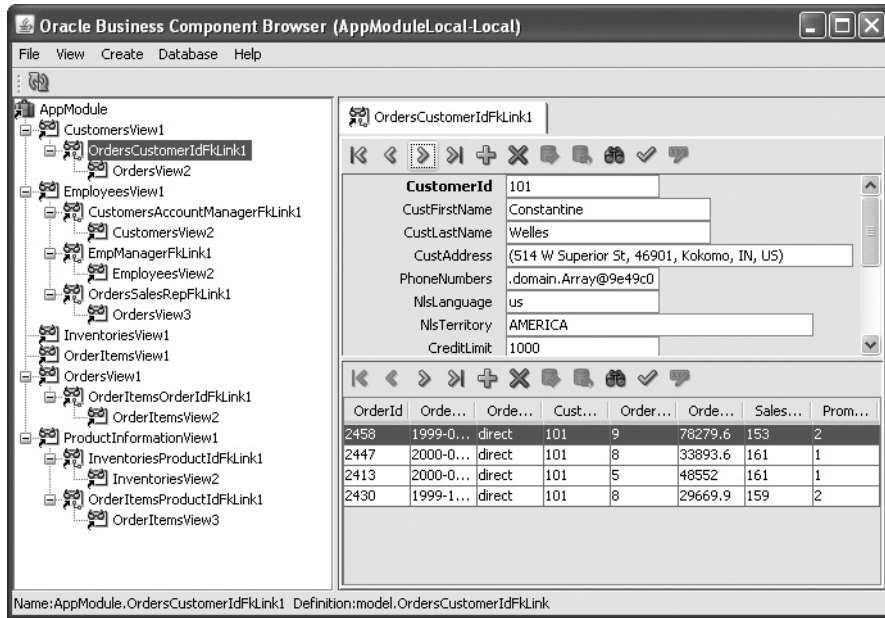
JDeveloper will now generate the entity objects, associations, view links, view objects, and an application module for implementing a business service based on the chosen database tables.

## Testing ADF Business Components

Congratulations! You've just created your first business service using ADF Business Components. Now you will see how you can actually test that business service to see what it does.

JDeveloper provides an indispensable feature called the ADF Business Component browser: it's like a default UI on top of your business service that allows you to test your business service without having to build a UI. Right-click the application module in the Application Navigator and select **Run**. The ADF Business Component browser connects to the database, shows the

application module you just created, and shows the view object instances and view links within that application module.



### TIP

Once you get further into Fusion application development, you will find the ADF Business Component browser an invaluable aid in debugging your application. Even if you have already built a UI on top of your business service, it is worth checking any business service changes in this tool to help isolate the testing.

## Testing Your Business Service

The first thing you will probably notice when looking at the browser for the business service you just created is that your application module has a lot of view object instances in it. The Create Business Components from Tables wizard obviously doesn't know what precise business use case you are trying to build, so it pretty much includes every combination of view object and how it can relate to other view objects. Later in the book, you will see how you can refine the application module to only contain the view object instances that you need.

For the moment, notice that some of the view object instances are top-level nodes while others are subnodes. This is a master/detail relationship, where the top node is the master and the subnode is the detail. This master/detail relationship can be nested down any number of levels.

**Browsing Data for a View Object** To browse data for a particular view object instance, double-click the instance name. This opens the instance as a form and retrieves records. If you double-click the link between a master and detail, the data is displayed as a master form with the detail as a table.

You can now use the browser toolbar and menu to navigate through records. Here you can test that the framework is correctly connecting to your database, retrieving, caching, and navigating records.

**Testing Operations on View Objects** You learned earlier that ADF Business Components provides default operations on business services. Functions such as deleting a record, adding a record or committing or rolling back the translation are available to test from the browser toolbar. There is also a feature to search for specific records.

**Testing Default Validation** When you created the business service from database tables, information from the underlying tables, including data types and constraints, was read and implemented by the framework. You can use the browser to test this behavior. For example, try to create a new customer record without specifying a value for CustomerId, or use a duplicate value. Note that the framework traps the exception and display an appropriate error. Alternatively, test what happens if you try to input a string into a numeric field like CreditLimit.

## Summary

In this chapter, you have built your first business service based on ADF Business Components and have learned that

- An entity object is a cache for holding records from an underlying database table.
- A view object allows you to select application-specific data into entity objects.
- You can also have read-only view objects that are based on static lists of data.
- View objects can be used to reference information from different entity objects.
- The application module is a transactional container for view object instances.
- The Create Business Components from Tables wizard allows you to quickly build a business service based on tables.
- The ADF Business Component browser allows you to test your business service without the need to develop a UI.

You should now feel more comfortable with the building blocks of ADF Business Components and how you can quickly create and test a business service. The next step is to fine-tune your entity objects and understand more about their features and properties.