3
Entities

Entities are classes that need to be persisted, usually in a relational database. In this chapter we cover the following topics:

- EJB 3 entities
- Java persistence API
- Mapping an entity to a database table
- Metadata defaults
- Introduction to the entity manager
- Packaging and deploying entities
- Generating primary keys
- Overriding metadata defaults

Introduction

Entities are classes that need to be persisted; their state is stored outside the application, typically in a relational database. Unlike session beans, entities do not have business logic other than validation. As well as storing such entities, we want to query, update, and delete them.

The EJB 3 specification recognizes that many applications have the above persistence needs without requiring the services (security, transactions) of an application server EJB container. Consequently the persistence aspects of EJB 3 have been packaged as a separate specification—the Java Persistence API (JPA). JPA does not assume we have a container and can even be used in a Java SE (Standard Edition) application. As well as persistence, JPA deals with Object/Relational Mapping and Queries, these are covered in Chapters 4 and 5 respectively. Most of our examples assume that the persistence engine exists within an EJB 3 container such as GlassFish or JBoss. In Chapter 6 we shall show examples of persistence outside a container.

Successful standalone object-relational mapping products such as open source Hibernate and proprietary Oracle Toplink have implemented these persistence technologies for a number of years. Creators of Oracle Toplink and Hibernate have been influential in the development of the JPA specification. So, readers who are familiar with either Hibernate or Toplink will recognize similarities between the JPA and those products. Furthermore, under the covers, the GlassFish application server implements the JPA using Toplink and the JBoss application server uses Hibernate. These are pluggable defaults however, so it is possible to implement the JPA in GlassFish using Hibernate for example.

The JPA can be regarded as a higher level of abstraction sitting on top of JDBC. Under the covers the persistence engine converts JPA statements into lower level JDBC statements.

EJB 3 Entities

In JPA, any class or POJO (Plain Old Java Object) can be converted to an entity with very few modifications. The following listing shows an entity `Customer.java` with attributes `id`, which is unique for a Customer instance, and `firstName` and `lastName`.

```
package ejb30.entity;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer implements java.io.Serializable {
    private int id;
    private String firstName;
    private String lastName;
    public Customer() {}
    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getFirstname() { return firstName; }
    public void setFirstname(String firstName) {
        this.firstName = firstName;
    }
    public String getLastname() { return lastName; }
    public void setLastname(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                  firstName + ",last name=" + lastName + "]";
    }
}
```

The class follows the usual JavaBean rules. The instance variables are non-public and are accessed by clients through appropriately named getter and setter accessor methods. Only a couple of annotations have been added to distinguish this entity from a POJO. Annotations specify entity metadata. They are not an intrinsic part of an entity but describe how an entity is persisted or, as we shall see in Chapter 4, how an entity is related to other entities. The `@Entity` annotation indicates to the persistence engine that the annotated class, in this case `Customer`, is an entity. The annotation is placed immediately before the class definition and is an example of a **class level** annotation. We can also have **property-based** and **field-based** annotations, as we shall see.

The `@Id` annotation specifies the primary key of the entity. The `id` attribute is a primary key candidate. Note that we have placed the annotation immediately before the corresponding getter method, `getId()`. This is an example of a property-based annotation. A property-based annotation must be placed immediately before the corresponding getter method, and not the setter method. Where property-based annotations are used, the persistence engine uses the getter and setter methods to access and set the entity state.

An alternative to property-based annotations are field-based annotations. We show an example of these later in this chapter. Note that all annotations within an entity, other than class level annotations, must be all property-based or all field-based.

The final requirement for an entity is the presence of a no-arg constructor.

Our Customer entity also implements the `java.io.Serializable` interface. This is not essential, but good practice because the Customer entity has the potential of becoming a detached entity. Detached entities must implement the `Serializable` interface. We will discuss detached entities in Chapter 6.

At this point we remind the reader that, as throughout EJB 3, XML deployment descriptors are an alternative to entity metadata annotations.

Comparison with EJB 2.x Entity Beans

An EJB 3 entity is a POJO and not a component, so it is referred to as an entity and not an entity bean. In EJB 2.x the corresponding construct is an entity bean component with the same artifacts as session beans, namely an XML deployment descriptor file, a remote or local interface, a home or localhome interface, and the bean class itself. The remote or local interface contains getter and setter method definitions. The home or local interface contains definitions for the `create()` and `findByPrimaryKey()` methods and optionally other finder method definitions. As with session beans, the entity bean class contains callback methods such as `ejbCreate()`, `ejbLoad()`, `ejbStore()`, `ejbRemove()`, `ejbActivate()`, `ejbPassivate()`, and `setEntityContext()`.

The EJB 3 entity, being a POJO, can run outside a container. Its clients are always local to the JVM. The EJB 2.x entity bean is a distributed object that needs a container to run, but can have clients from outside its JVM. Consequently EJB 3 entities are more reusable and easier to test than EJB 2.x entity beans.

In EJB 2.x we need to decide whether the persistence aspects of an entity bean are handled by the container (Container Managed Persistence or CMP) or by the application (Bean Managed Persistence or BMP).

In the case of CMP, the entity bean is defined as an abstract class with abstract getter and setter method definitions. At deployment the container creates a concrete implementation of this abstract entity bean class.

In the case of BMP, the entity bean is defined as a class. The getter and setter methods need to be coded. In addition the `ejbCreate()`, `ejbLoad()`, `ejbStore()`, `ejbFindByPrimaryKey()`, and any other finder methods need to be coded using JDBC.

Mapping an Entity to a Database Table

We can map entities onto just about any relational database. GlassFish includes an embedded Derby relational database. If we want GlassFish to access another relational database, Oracle say, then we need to use the GlassFish admin console to set up an Oracle data source. We also need to refer to this Oracle data source in the `persistence.xml` file. We will describe the `persistence.xml` file later in this chapter. These steps are not required if we use the GlassFish default Derby data source. All the examples in this book will use the Derby database.

EJB 3 makes heavy use of defaulting for describing entity metadata. In this section we describe a few of these defaults.

First, by default, the persistence engine maps the entity name to a relational table name. So in our example the table name is `CUSTOMER`. If we want to map the Customer entity to another table we will need to use the `@Table` annotation as we shall see later in this chapter.

By default, property or fields names are mapped to a column name. So `ID`, `FIRSTNAME`, and `LASTNAME` are the column names corresponding to the `id`, `firstname`, and `lastname` entity attributes. If we want to change this default behavior we will need to use the `@Column` annotation as we shall see later in this chapter.

JDBC rules are used for mapping Java primitives to relational datatypes. So a `String` will be mapped to `VARCHAR` for a Derby database and `VARCHAR2` for an Oracle database. An `int` will be mapped to `INTEGER` for a Derby database and `NUMBER` for an Oracle database.

The size of a column mapped from a `String` defaults to 255, for example `VARCHAR(255)` for Derby or `VARCHAR2(255)` for Oracle. If we want to change this column size then we need to use the `length` element of the `@Column` annotation as we shall see later in this chapter.

To summarize, if we are using the GlassFish container with the embedded Derby database, the Customer entity will map onto the following table:

CUSTOMER
ID INTEGER PRIMARY KEY
FIRSTNAME VARCHAR(255)
LASTNAME VARCHAR(255)

Most persistence engines, including the GlassFish default persistence engine, Toplink, have a schema generation option, although this is not required by the JPA specification. In the case of GlassFish, if a flag is set when the application is deployed to the container, then the container will create the mapped table in the database. Otherwise the table is assumed to exist in the database.

Introducing the EntityManager

We will use a remote stateless session bean for business operations on the Customer entity. We will call our session bean interface `BankService`. In later chapters, we will add more entities and business operations that are typically used by a banking application. At this stage our banking application deals with one entity—Customer, and business methods `addCustomer()` and `findCustomer()`, which respectively add a customer to the database and retrieve a customer entity given the customers identifier. The interface listing follows:

```
package ejb30.session;
import javax.ejb.Remote;
import ejb30.entity.Customer;
@Remote
public interface BankService {
    void addCustomer(int custId, String firstName,
                     String lastName);
    Customer findCustomer(int custId);
}
```

The code below shows the session bean implementation, `BankServiceBean`:

```
package ejb30.session;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import ejb30.entity.Customer;
import javax.persistence.PersistenceContext;
@Stateless
public class BankServiceBean implements BankService {
    @PersistenceContext(unitName="BankService")
    private EntityManager em;
    public Customer findCustomer(int custId) {
        return ((Customer)
                          em.find(Customer.class, custId));
    }
    public void addCustomer(int custId, String firstName,
                            String lastName) {
        Customer cust = new Customer();
        cust.setId(custId);
        cust.setFirstname(firstName);
        cust.setLastname(lastName);
        em.persist(cust);
    }
}
```

The `EntityManager` is a service provided by the persistence engine which provides methods for persisting, finding, querying, removing, and updating entities. The set of managed entity instances is known as a **persistence context**. It is important to stress that a persistence context is associated with a managed set of entity *instances* or *objects*. If we have, say, 10 clients accessing the bean, we will have 10 persistence contexts. Associated with an `EntityManager` is a **persistence unit** which specifies configuration information. The statement:

```
@PersistenceContext(unitName="BankService")
private EntityManager em;
```

defines an `EntityManager` instance `em` with an associated persistence context and a persistence unit named `BankService`. The actual persistence unit configuration is specified in a `persistence.xml` file which we shall see shortly.

Actually, behind the scenes, the `@PersistenceContext` annotation causes the container to:

- Use the `EntityManagerFactory` to create an instance of the `EntityManager`
- Bind the `EntityManager` instance to the `BankService` persistence unit
- Inject the `EntityManager` instance into the `em` field.

Now let's take a look at the `addCustomer()` method. First, the statement

```
Customer cust = new Customer();
```

creates, as expected, an instance of the Customer entity. However, at this stage the entity instance is not managed. The instance is not yet associated with a persistence context. Managed entity instances are also referred to as **attached** instances and unmanaged instances as **detached** instances.

The next few statements invoke the Customer setter methods in the usual way. The statement:

```
em.persist(cust);
```

invokes the `EntityManager.persist()` method. At this stage the entity instance is not necessarily written immediately to the database. Rather, the entity instance becomes managed or attached and associated with a persistence context. The `EntityManager` manages the synchronization of a persistence context with the database. There may be more setter methods in our method after the `persist()` statement. The `EntityManager` is unlikely to update, or flush, the database after each setter as database write operations are expensive in terms of performance. More likely the entity state would be kept in a cache and flushed to the database at the end of the current transaction. The latest point at which the `EntityManager` will flush to the database is when the transaction commits, although the `EntityManager` may chose to flush sooner.

By default, a transaction starts when a method is invoked, and is committed on leaving a method. We will discuss transactions further in Chapter 7.

Now let's look at the `findCustomer()` method. The statement:

```
return ((Customer) em.find(Customer.class, custId));
```

invokes the `EntityManager.find()` method. This method retrieves an entity by its primary key. The parameters of `find()` are the entity class and a primary key, which is an `Object` type, which uniquely identifies the entity. In our example we use the variable `custId` as the primary key (the **autoboxing** feature of Java SE 5 converts the supplied `int` type to an `Object` type).

The `EntityManager` service also provides a query language, **JPQL**, which we will cover in Chapter 5. Furthermore the `EntityManager` provides entity management methods, such as merging detached entities. At this stage the reader might think that entities should always be in an attached (managed) state, however we shall see examples in Chapter 6 where entities do exist in a detached state.

The following listing is an example of how a client might invoke the `BankService` methods to create and then find an entity.

```
package ejb30.client;
import javax.naming.Context;
import javax.naming.InitialContext;
import ejb30.session.*;
import ejb30.entity.Customer;
import javax.ejb.EJB;
public class BankClient {
    @EJB
    private static BankService bank;
    public static void main(String[] args) {
        try {
            int custId = 0;
            String firstName = null;
            String lastName = null;

            try {
                custId = Integer.parseInt(args[0]);
                firstName = args[1];
                lastName = args[2];
            } catch (Exception e) {
                System.err.println(
                 "Invalid arguments entered, try again");
                System.exit(0);
            }
            // add customer to database
            bank.addCustomer(custId, firstName, lastName);
            Customer cust = bank.findCustomer(custId);
            System.out.println(cust);
        } catch (Throwable ex) {
            ex.printStackTrace();
        }
    }
}
```

The code is fairly straightforward. The variables `custId`, `firstName`, and `lastName` are initialized to parameters supplied to the main method when it is invoked. The client uses the `BankService.addCustomer()` method to create and add a Customer entity to the database. The `BankService.findCustomer()` method is then invoked to retrieve the entity just created.

We now return to the persistence unit which we referred to in the `@PersistenceContext` annotation that we used in the `BankService` session bean. The actual configuration is specified in an XML file, `persistence.xml`, which is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
version="1.0">
<persistence-unit name="BankService"/>
</persistence>
```

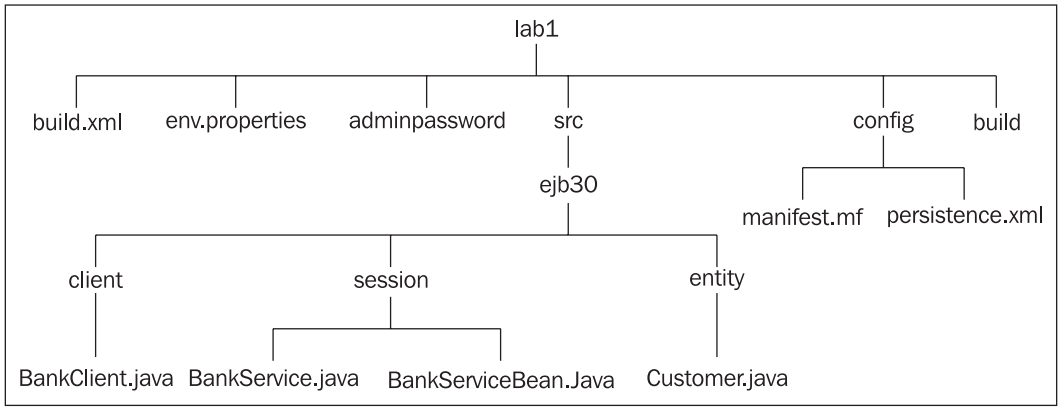
Again we make heavy use of defaults here. A `persistence.xml` file is required whenever an application makes use of `EntityManager` services, even if we rely entirely on defaults. In our case we have just used the `name` element to define the `BankService` persistence unit. Note the name must match the `unitName` used earlier in the `@PersistenceContext` annotation. There are a number of persistence unit elements, in particular regarding transactions and data sources. We will return to these elements later in the book.

Packaging and Deploying Entities

As we have introduced a number of new source code artifacts that were not present in the previous chapter on session beans, the packaging process will be a little different. First we look at the program directory structure for our sample application.

The Program Directory Structure

Below is the program directory structure for the `BankClient` application which invokes the `BankServiceBean` in order to add the `Customer` entity to the database.



There are a couple of differences in the directory structure from the one described in the previous chapter. We have added an `entity` subdirectory; this contains the entity source code, `Customer.java`. We have also added a `config` subdirectory containing the `manifest.mf` and `persistence.xml` files.

Building the Application

Because we are persisting entities from an EJB container we need to place the `persistence.xml` file in the `META-INF` directory within the EJB module, `BankService.jar`. The `package-ejb` target in the Ant build file does this:

```
<target name="package-ejb" depends="compile">
   <jar jarfile="${build.dir}/BankService.jar">
       <fileset dir="${build.dir}">
           <include name="ejb30/session/**" />
         <include name="ejb30/entity/**" />
       </fileset>
       <metainf dir="${config.dir}">
             <include name="persistence.xml" />
        </metainf>
   </jar>
</target>
```

We can use the `jar -tvf` command to examine the contents of the JAR file:

```
C:\EJB3Chapter03\glassfish\lab1\build>jar  -tvf  BankService.jar
   ...META-INF/
   ...META-INF/MANIFEST.MF
   ...ejb30/
   ...ejb30/entity/
   ...ejb30/session/
   ...ejb30/entity/Customer.class
   ...ejb30/session/BankService.class
   ...ejb30/session/BankServiceBean.class
   ...META-INF/persistence.xml
```

This is only one of a number of ways to package the persistence unit. Entities can be persisted not only from an EJB container but also from a web container or from a Java SE application running outside the container. We shall see an example of Java SE persistence in Chapter 6. Consequently the persistence unit (entity classes together with the `persistence.xml` file) can be placed in `WEB-INF/classes` directory of a WAR file or in a Java SE application client jar file.

Alternatively we can package a persistence unit into a separate jar file. This jar file can then be added to the `WEB-INF/lib` directory of a WAR file, the root of an EAR file or in the library directory of an EAR file. Adding a persistence unit to a Java EE module limits its scope to that module. If we place the persistence unit in an EAR it is visible to all modules within the application.

In our example we want the database tables to be created at deploy time, and dropped when we undeploy an application from the container. We modify the `deploy` target in our Ant build file, adding a `createtables=true` clause:

```
<target name="deploy">
  <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
    <arg line="deploy --user admin --passwordfile
        adminpassword --createtables=true
        ${build.dir}/BankService.ear"/>
  </exec>
</target>
```

Similarly we add a `droptables=true` clause for the `undeploy` target.

```
<target name="undeploy">
  <exec executable="${glassfish.home}/bin/asadmin"
        failonerror="true"
        vmlauncher="false">
    <arg line="undeploy --user admin  --passwordfile
        adminpassword  --droptables=true  BankService"/>
  </exec>
</target>
```

Note that before we can deploy `BankService` within GlassFish, we need to start up the embedded Derby database. We do this from the command-line with the `asadmin` utility:

```
C:\> asadmin start-database
```

Field-Based Annotations

Recall we used a property-based annotation for the primary key `@Id` in the Customer entity. This results in the persistence engine using getter and setter methods to access and set the entity state. In this section we will modify the Customer entity to use field-based, rather than property-based annotations. The following listing demonstrates this:

```
@Entity
public class Customer implements java.io.Serializable {
    @Id
    private int id;
    private String firstName;
    @Basic
    private String lastName;
    public Customer() {};
    public Customer(int id, String firstName,
                    String lastName){
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getFirstname() { return firstName; }
    public void setFirstname(String firstName) {
        this.firstName = firstName;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                 firstName + ",last name=" + lastName + "]";
    }
}
```

Note that we have placed the primary key `@Id` annotation immediately before the field declaration. If we use a field-based annotation then all other annotations, other than class level annotations such as `@Entity`, must also be field-based annotations. In the case of field-based annotations the persistence engine will use reflection to access and set the entity state. Getter and setter methods may be present in the entity but they are ignored by the persistence engine. To demonstrate this point we have removed the `getLastName()` and `setLastName()` methods from the Customer entity.

We have seen that attributes other than the primary key are mapped to a database by default. We call these mappings **basic mappings**. A basic mapping is used where the attribute is a Java primitive type or any Serializable type such as a `String`. We can explicitly flag these mappings using the `@Basic` annotation. We have done this for the `lastName` field.

Generating Primary Keys

Up to now we have relied on the application to set an entity's primary key by supplying a primary key identifier as a parameter. However, we may want to relieve the application of this responsibility and use automatically generated keys. JPA provides the **Table**, **Sequence**, **Auto**, and **Identity** strategies for generating primary keys.

Table Strategy

With this strategy the persistence engine uses a relational database table from which the keys are generated. This strategy has the advantage of portability; we can use it with any relational database. Here is an example of how we would specify a table generation strategy for the `id` attribute of the Customer entity:

```
@TableGenerator(name="CUST_SEQ",
                table="SEQUENCE_TABLE",
                pkColumnName="SEQUENCE_NAME",
                valueColumnName="SEQUENCE_COUNT")
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
                generator="CUST_SEQ")
public int getId() { return id; }
```

First we use the `@TableGenerator` annotation to specify the table used for key generation. This annotation can be placed on the primary key attribute, as we have done here, or on the entity class.

The `name` element in our example `CUST_SEQ`, identifies the generator. The `table` element is the name of the table that stores the generated values. In our case we have chosen `SEQUENCE_TABLE` as the table name. There is a default table element, but this is dependent on the persistence engine being used. In the case of GlassFish, this table name is `SEQUENCE`. The `pkColumnName` element is the name of the primary key column in the sequence table. We have chosen `SEQUENCE_NAME` as the column name. Again the default is persistence engine dependent. In the case of GlassFish this value is `SEQ_NAME`. The `valueColumnName` element is the name of the column that stores the last key value generated. The default is persistence engine dependent. In the case of GlassFish this value is `SEQ_COUNT`.

One element of `@TableGenerator` for which we assumed the default is `pkColumnValue`. This is the `String` value that is entered in the `pkColumnName` column. The default is persistence engine dependent. In the case of GlassFish, this value is set to the name element of `@TableGenerator`. In our example this is `CUST_SEQ`. So `SEQUENCE_TABLE` will initially have the following row present:

```
SQL> SELECT * FROM SEQUENCE_TABLE;
    SEQUENCE_NAME   SEQUENCE_COUNT
    -------------   --------------
    CUST_SEQ                     0
```

If we want to store several sequences in the same sequence table, for example a separate sequence for each entity, then we need to manually specify the `pkColumnValue` element. The following statement sets the `pkColumnValue` to `CUSTOMER_SEQ`:

```
    @TableGenerator(name="CUST_SEQ",
                    table="SEQUENCE_TABLE",
                    pkColumnName="SEQUENCE_NAME",
                    valueColumnName="SEQUENCE_COUNT",
                    pkColumnValue="CUSTOMER_SEQ")
```

We should mention two other `@TableGenerator` elements: `initialValue` and `allocationSize`. `initialValue` is the initial value assigned to the primary key sequence. The default value is 0. The sequence is incremented by a value of 1. The `allocationSize` is the cache size into which the persistence engine reads from the sequence table. The default value is 50.

The `@GeneratedValue` annotation specifies the key generation strategy with the `strategy` element. In our case we have chosen the `TABLE` strategy. The default strategy is `AUTO` which we describe later in this chapter. The `generator` element provides the name of the primary key generator. In our case this is `CUST_SEQ` and must match the name element of the `@TableGenerator` annotation. The `@GeneratedValue` annotation is a field-based or property-based annotation, so must be present immediately before the primary key field or getter method.

Sequence Strategy

Some databases, such as Oracle, have a built-in mechanism called sequences for generating keys. To invoke such a sequence we need to use the `@SequenceGenerator` annotation. For example:

```
    @SequenceGenerator(name="CUST_SEQ",
                       sequenceName="CUSTOMER_SEQUENCE")
```

As with the `@TableGenerator` annotation, the `name` element identifies the generator. The `sequenceName` element identifies the database sequence object. `initialValue` is the initial value assigned to the primary key sequence. The default differs from the `@TableGenerator` equivalent, and is equal to 1. The `allocationSize` is the cache size into which the persistence engine reads from the sequence. The default value is 50.

The `@SequenceGenerator` annotation can be placed on the primary key attribute or on the entity class.

As with Table generated sequences, we use the `@GeneratedValue` annotation to specify the generation strategy. For example,

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="CUST_SEQ")
```

This time we have specified the `SEQUENCE` strategy. The name of the primary key generator is `CUST_SEQ`, and this must match the name element of the `@SequenceGenerator` annotation. Remember the `@GeneratedValue` annotation is a field-based or property-based annotation, so it must be present immediately before the primary key field or getter method.

Identity Strategy

Some databases, such as Microsoft SQL Server, use an identity column for generating keys. To use this we specify the `IDENTITY` strategy in the `@GeneratedValue` annotation:

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

Note that there is no `generator` element that we have for the Table and Sequence strategies.

Auto Strategy

The final strategy is the `AUTO` strategy. With this strategy the persistence engine selects the strategy. In the case of GlassFish the `TABLE` strategy is selected. We can specify an `AUTO` strategy either explicitly:

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

or implicitly:

```
@GeneratedValue
```

as the default strategy is `AUTO`.

In the case of GlassFish the default sequence table has the name `SEQUENCE`, with columns `SEQ_NAME` and `SEQ_COUNT`:

SEQUENCE
SEQ_NAME VARCHAR(50) PRIMARY KEY
SEQ_COUNT DECIMAL

Overriding Metadata Defaults

In this section we return to the Customer entity and override some of the default mapping options.

First we want to use the `CLIENT` table with columns `CUSTOMER_ID`, `FIRST_NAME` and `LAST_NAME` to store Customer entities. The primary key is `CUSTOMER_ID`. The table definition is:

CLIENT
CUSTOMER_ID NUMBER(38) PRIMARY KEY
FIRST_NAME VARCHAR(30) NOT NULL
LAST_NAME VARCHAR(30) NOT NULL

The modified listing for `Customer.java` is shown below:

```
@Entity
@Table(name = "CLIENT")
public class Customer implements java.io.Serializable {
    private int id;
    private String firstName;
    private String lastName;
    public Customer() {};
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "CUSTOMER_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    @Column(name = "FIRST_NAME")
    public String getFirstname() { return firstName; }
    public void setFirstname(String firstName) {
        this.firstName = firstName;
    }
```

```
    @Column(name = "LAST_NAME")
    public String getLastname() { return lastName; }
    public void setLastname(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return "[Customer Id =" + id + ",first name=" +
                  firstName + ",last name=" + lastName + "]";
    }
}
```

Note we have used the `@Table` annotation to specify the table name as `CLIENT`. The default table name is the entity name, `CUSTOMER` in our example. The `catalog` element specifies the database catalog where the table is located. Many database systems, Derby included, do not support the concept of a catalog. So in our example we can leave this element out. The `schema` element specifies the database schema where the table is located. The default is persistence provider specific, in the case of the GlassFish `APP` is the default schema for the embedded Derby database. Again, in our example, we rely on the default.

We have explicitly specified an `AUTO` primary key generation strategy:

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
```

Recall that GlassFish selects the `TABLE` strategy in this case.

Recall that by default column names are set to attribute names. To override this we use the `@Column` annotation. For example,

```
@Column(name = "CUSTOMER_ID")
public int getId() { return id; }
```

will map the `id` attribute to the `CUSTOMER_ID` column. In a similar fashion we map the `firstName` and `lastName` attributes to the `FIRST_NAME` and `LAST_NAME` columns respectively.

Summary

Entities are classes that need to be persisted, usually in a relational database. The persistence aspects of EJB 3 have been packaged as a separate specification, the Java Persistence API (JPA), so that applications that do not need EJB container services can still persist their entities to a relational database. Persistence services are handled by a persistence engine. In this chapter we make use of the Toplink persistence engine that comes bundled with the GlassFish application server.

Any Java class, or POJO, can be converted to an entity using metadata annotations. We described by means of an example the default rules for mapping an entity to a relational database table.

We introduced the `EntityManager` service, which provides methods for persisting, finding, querying, removing and updating entities. We saw examples of the `EntityManager.persist()` and `EntityManager.find()` methods. We introduced the concept of a persistence context, which is the set of managed entity instances.

We looked at Ant scripts for packaging and deploying an application which uses entities.

We examined strategies for generating primary keys. Finally we looked at examples of overriding default rules for mapping entities to relational tables.