JPA Cheatsheet

Rodrigo García Carmona (v1.2.0)

This guide has been written with SQLite and EclipseLink in mind.

Project setup

To be able to use JPA we must add the JPA provider (in our case, EclipseLink) to our project. This is done by importing the following two jar files:

- *eclipselink.jar*
- *javax.persistence.jar*

We must also add a *META-INF* folder in our *src* (source code) folder and put a *persistence.xml* file inside it. The *persistence.xml* should be similar to this one:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

<persistence-unit name="provider-name" transaction-type="RESOURCE_LOCAL">
 <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

 <!-- Entity Classes -->
 <class>a.java.package.EntityClass1</class>
 <class>a.java.package.EntityClass2</class>
 <class>a.java.package.EntityClass3</class>

 <properties>
  <!-- Connection properties -->
  <property name="javax.persistence.jdbc.driver"
   value="org.sqlite.JDBC" />
     <property name="javax.persistence.jdbc.url"
   value="jdbc:sqlite:database-url" />
  <!-- Fill if we need user and password -->
  <property name="javax.persistence.jdbc.user" value="" />
  <property name="javax.persistence.jdbc.password" value="" />

     <!-- JPA creates the database schema if it doesn't exist -->
  <property name="eclipselink.ddl-generation" value="create-tables" />
 </properties>

</persistence-unit>
</persistence>
```

In this file, we should change:

- The *provider-name*. We're going to refer to this name in the Entity Manager creation.
- The *Entity Classes* that we're going to model.
- The *database-url* that points to our database's location.

Finally, we must also annotate (see later in this same document) all the entity classes we're going to manage with JPA.

An important note: Before executing any JPA code in a project that also uses JDBC (like our examples) **be completely sure** that you've already created the database's tables. This is needed so we're certain that the database schema is the one we want, and not any other one that JPA could decide works better.

Working with JPA

To actually work with JPA, the first thing to do is create an Entity Manager, which fulfills the same role as a connection in JDBC:

```
EntityManager em = Persistence.createEntityManagerFactory("provider-name").
                   createEntityManager();
```

Like with JDBC, if we work with SQLite, we must enable support for foreign key constraints. In order to do that, these lines of code must be run just after the creation of the Entity Manager:

```
em.getTransaction().begin();
em.createNativeQuery("PRAGMA foreign_keys=ON").executeUpdate();
em.getTransaction().commit();
```

Note that *provider-name* is the one that we specified in the *persistence.xml* file before.

After working with the database, we must remember to close the Entity Manager:

```
em.close();
```

Transactions are used with operations that modify the database state: *create*, *update* and *delete*.

To start a transaction we write:

```
em.getTransaction().begin();
```

Too finally commit a transaction we write:

```
em.getTransaction().commit();
```

And if we want to undo all changes (rollback) since the last commit we write:

```
em.getTransaction().rollback();
```

Most of the time, the database and objects states will match each other, but while working in a multi-user or remote environment, we might want to be sure that the objects reflect the state of the database, or vice-versa. To do that we use the following two methods:

To force the database to reflect the object model state:

```
em.flush();
```

To force the object model to reflect the database state:

```
em.refresh();
```

CRUD Operations

We can execute the four CRUD operations using JPA.

Create:

```
em.persist(object);
```

Read

```
Query q = em.createNativeQuery("SELECT Query", ClassName.class);
```

And then either this:

```
List<ClassName> list = q.getResultList();
```

Or (only when we know we'll get just **one** result):

```
ClassName object = (ClassName) q.getSingleResult();
```

Update

```
object.setMethod(setValue);
```

It is important to remember that we must make changes that affect an association in both sides:

```
employee.setDepartment(department);
department.addEmployee(employee);
```

Delete

```
em.remove(object);
```

JPA Entity Classes

All entity classes must have the following characteristics:

- Implement *Serializable*.
- A parameter-less constructor.
- Getters and setters for all attributes.
- *equals* and *hashcode* methods that use just the primary key attribute.
- Bidirectional associations.

And it's recommended that they have the following characteristics:

- A *toString* method.
- Add and remove methods for all *List* attributes.

Attributes should be of one of the following types:

- *Integer* (INTEGER in SQL)
- *Double* (REAL in SQL)
- *String* (TEXT in SQL)
- *java.sql.Date* (DATE in SQL)
- *byte[]* (BLOB in SQL)
- Another entity class or a list of elements from another entity class (representing a relationship with foreign keys in SQL)

The table represented by the entity class must have:

- A PRIMARY KEY with the AUTOINCREMENT constraint (in SQLite).

We must annotate every entity class in the following way:

```
@Entity
@Table(name = "table_name")
public class EntityClass implements Serializable {

 @Id
 @GeneratedValue(generator="generator_name")
 @TableGenerator(name="generator-name", table="sqlite_sequence",
     pkColumnName="name", valueColumnName="seq",
     pkColumnValue="tabl_name")
 private Integer id;
 ...
}
```

- *table_name* must be the relational table name.
- *generator_name* can be any name that's different to all other generator names.

Note that the TableGenerator structure shown here is specific to SQLite.

We annotate blobs this way:

```
@Basic(fetch=FetchType.LAZY)
@Lob
private byte[] photo;
```

The LAZY fetch type means that this attribute is only going to be retrieved from the database when it's really accessed; that is, when the appropriate get method is called. The opposite of LAZY is EAGER.

All other attributes are automatically mapped to columns with the same name in the corresponding table. If we want to link an attribute with a column of a different name, we must use the Column annotation:

```
@Column(name="surname)
private String familyName;
```

All relationships between entities must also be annotated. We can choose to make these EAGER or LAZY; the former if we know that we're going to need the linked entity right away, and the latter if not. **Be careful of making all associations EAGER.** If you do that you might end getting the whole database with each read operation.

One to One Relationships

Here's an example on how to implement a one to one relationship using JPA.

Tables

employees

id	name	surname	salary	address_id
1	Bob	Way	50000	6
2	Sarah	Smith	60000	7

addresses

id	street	city	country
6	Mayor 5	Madrid	Spain
7	Fernán 17	Burgos	Spain

Classes

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
 @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @OneToOne(fetch=FetchType.LAZY)
 @JoinColumn(name="address_id")
 private Address address;
 ...
}

@Entity
@Table(name="addresses")
public class Address implements Serializable {
 @Id
 // Other Id annotations as needed
 private Integer id;
 ...
```

```
 @OneToOne(fetch=FetchType.LAZY, mappedBy="address")
 private Employee owner;
 ...
}
```

Many to One Relationships

Here's an example on how to implement a many to one relationship using JPA.

Tables

employees

id	name	surname	salary
1	Bob	Way	50000
2	Sarah	Smith	60000

phones

id	number	type	owner_id
1	917555555	home	1
2	669696969	work	1
3	666666666	work	2

Classes

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
  @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @OneToMany(mappedBy="owner")
 private List<Phone> phones;
 ...
}

@Entity
@Table(name="addresses")
public class Phone implements Serializable {
 @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @ManyToOne(fetch=FetchType.LAZY)
 @JoinColumn(name="owner_id")
 private Employee owner;
 ...
}
```

Many to Many Relationships

Here's an example on how to implement a many to many relationship using JPA.

Tables

employees

id	firstname	lastname
1	Bob	Way
2	Sarah	Smith

projects

id	name
1	GIS
2	SIG

proj_emp

emp_id	proj_id
1	1
1	2
2	1

Classes

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
   @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @ManyToMany
 @JoinTable(name="proj_emp",
     joinColumns={@JoinColumn(name="emp_id", referencedColumnName="id")},
     inverseJoinColumns={@JoinColumn(name="proj_id", referencedColumnName="id")})
 private List<Project> projects;
 ...
}

@Entity
@Table(name="projects")
public class Project implements Serializable {
 @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @ManyToMany(mappedBy="projects")
 private List<Employee> employees;
 ...
}
```

Many to Many Relationships with an Association Class

When a many to many relationship has an association class, we need to represent this extra information as another class. In this case, the many to many relationship is simulated as two many to one relationships between the association class and the two related classes.

Tables

employees

id	firstname	lastname
1	Bob	Way
2	Sarah	Smith

projects

id	name
1	GIS
2	SIG

proj_emp

emp_id	proj_id	is_lead
1	1	true
1	2	false
2	1	false

Classes

```
@Entity
@Table(name="employees")
public class Employee implements Serializable {
   @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @OneToMany(mappedBy="employee")
 private List<ProjectAssociation> projects;
 ...
}

@Entity
@Table(name="projects")
public class Project implements Serializable {
   @Id
 // Other Id annotations as needed
 private Integer id;
 ...
 @OneToMany(mappedBy="project")
 private List<ProjectAssociation> employees;
 ...
}

@Entity
@Table(name="proj_emp")
@IdClass(ProjectAssociationId.class)
public class ProjectAssociation implements Serializable {

 @Id
 private Integer emp_id;
 @Id
 private Integer proj_id;
 @Column(name="is_lead")
 private boolean isTeamLead;
```

```
 @ManyToOne
 @PrimaryKeyJoinColumn(name="emp_id", referencedColumnName="id")
 private Employee employee;
 @ManyToOne
 @PrimaryKeyJoinColumn(name="proj_id", referencedColumnName="id")
 private Project project;
 ...
}

public class ProjectAssociationId implements Serializable {

 private Integer emp_id;
 private Integer proj_id;

 // equals() and hashcode() using both ids
 public int hashCode() {
     return (int)(employeeId + projectId);
 }

 public boolean equals(Object object) {
     if (object instanceof ProjectAssociationId) {
      ProjectAssociationId otherId = (ProjectAssociationId) object;
      return (otherId.employeeId == this.employeeId) &&
    (otherId.projectId == this.projectId);
      }
     return false;
 }

}
```

Note that, in the example, we needed to create a special class *ProjectAssociationID*, whose sole purpose is to compute the primary key of the association class. This is needed because such primary key is made by combining two foreign keys.

Here's an example on how to create an association class' object for this many to many relationship:

```
public void addEmployee(Employee employee, boolean teamLead) {
    ProjectAssociation association = new ProjectAssociation();
    association.setEmployee(employee);
    association.setProject(this);
    association.setEmployeeId(employee.getId());
    association.setProjectId(this.getId());
    association.setIsTeamLead(teamLead);

    this.employees.add(association);
    // Also add the association object to the employee.
    employee.getProjects().add(association);
}
```

The previous code is designed to be part of the *Project* class.

Id generation with JDBC, JPA and SQLite

To be able to work with JDBC and JPA at the same time while using SQLite (as happens in our examples), we must force JPA to use the SQLite primary key generation strategy.

When SQLite needs to assign a primary key to a row, it looks in a special table called *sqlite_sequence*, which is automatically created in each SQLite database. This table has two columns: *name* that contains the other tables' names, and *seq*, which stores the next primary key to be assigned. Here's an example of this table:

sqlite_sequence

name	seq
employees	7

departments	4

In this example, the next new item in the employees table will have an id of 7, and the next department will have a primary key of 4.

We need to reflect this in the entity classes, so we annotate the id attribute in the following way:

```
@Id
@GeneratedValue(generator = "employees")
@TableGenerator(name = "employees", table = "sqlite_sequence",
                pkColumnName = "name", valueColumnName = "seq",
                pkColumnValue = "employees")
private Integer id;
```

What table to use for the id is stored in the *table* parameter, which in our case is always *sqlite_Sequence*. In the same way, *pkColumnName* and *valueColumnName* must always be *name* and *seq*, respectively, since they point to the two columns of the aforementioned table.

The *generator* and *name* properties must have the same name (*employees* in the example), and that name should be unique for each entity.

Finally, *pkColumnValue* have to contain the name of the entity's table as it appears on the database. In this example, *employees*. Notice that is the same name that we chose for the previous two properties, for consistency's sake.