Unit 4

Object-relational mapping.

Hibernate.

Content

1. What is Object-Relational Mapping?

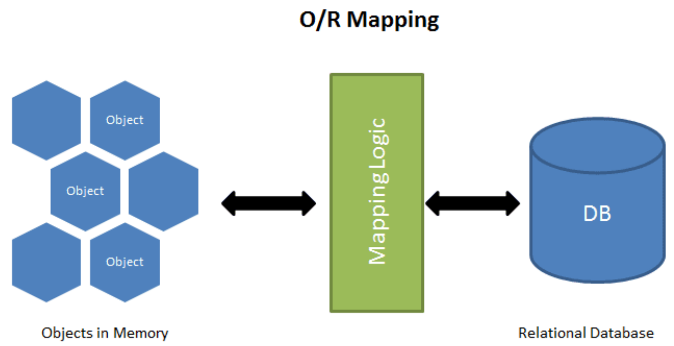
As far as we have seen in the previous unit, databases are widely used to ensure data persistence. Within DBMS, the most popular (and spreaded) are SQL-compliant DBMSs. To obtain data from a SQL DBMS we use the SQL query language, that is a powerful and easily understandable method to access data.

However, as an application grows bigger, the SQL queries turn more and more complex. Further, the queries not only become longer in size and complicated but also difficult to understand. To increase developer's problems, there exist multiple ways to write the same query which only makes it more difficult to understand. If we add to the equation the fact that we don't manage tables directly in our code but objects, the object-relational mismatch paradigm is served.

Relational objects are represented in a tabular format, while object models are represented in an interconnected graph of object format. While storing and retrieving an object model from a relational database, some mismatch occurs due to the following reasons:

- **Granularity** : Object model has more granularity than relational model.
- **Subtypes** : Subtypes (means inheritance) are not supported by all types of relational databases.
- **Identity** : Like object model, relational model does not expose identity while writing equality.
- **Associations** : Relational models cannot determine multiple relationships while looking into an object domain model.
- **Data navigation** : Data navigation between objects in an object network is different in both models.

Object-Relational Mapping is a technique that lets you query and manipulates data from a database using an object-oriented paradigm. ORM loves objects as much as developers, and is available for any programming language of your choosing. ORMs can be thought of as a translator converting our code from one form to another.



The Object-Relational Mapper generates objects (as in OOP) that virtually map the tables in your database. Then the programmer would use these objects to interact and play with the database! So the core idea is to try and shield the programmer from having to write optimized and complex SQL code.

If you're building a small project, installing an ORM library isn't required. Using SQL statements to drive your application should be sufficient. An ORM is quite beneficial for medium to large-scale projects that get data from hundreds of database tables. In such a situation, you need a framework that allows you to operate and maintain your application's data layer in a consistent and predictable way.

2. JPA (Java Persistence API)

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

To reduce the burden of writing codes for relational object management, a programmer follows the 'JPA Provider' framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.

A persistence entity is a Java Bean class whose state can be dumped into a table in a relational database. Instances of one of these entities are correspond to individual rows in the table. Entities typically have relationships with other entities, and these relationships are expressed through object/relational metadata. Such metadata can be specified directly in the class file using "Java annotations" (special tags preceded by @), or in a XML description file, which would be distributed with the application.

JPQL (short for "Java Persistence Query Language") is an object-oriented, platform-independent query language, defined as part of the JPA specification. Queries are made to entities stored in a relational DBMS. These queries resemble SQL queries in their syntax, but work with entity objects instead of directly with database tables.

There are different technologies that rely on JPA, for example:

- Enterprise JavaBeans (EJB): It is one of the programming interfaces (API) integrated in Java Enterprise Edition. Includes a number of objects that are useful in server-side programming, and that can make easier tasks such as transaction processing, concurrency management, invocation asynchronous methods, task scheduling, directory services, security, etc. The EJB 3.0

specification (in turn part of the Java EE platform 5) includes persistence support.

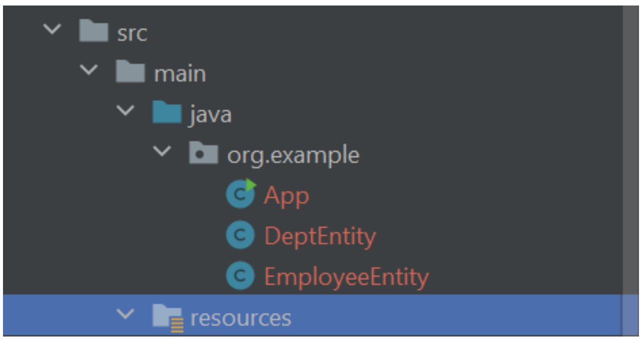
- Java Data Objects API: The Java Persistence API specifies the way to implement persistence for relational DBMS (although some vendors support other database models).
- Hibernate ORM (or simply Hibernate) is an open source framework for object-relational mapping in Java. Versions 3.2 and later implement the Java Persistence API. It will be the tool in which we will focus on this unit.

3. Hibernate set up with IntelliJ IDEA

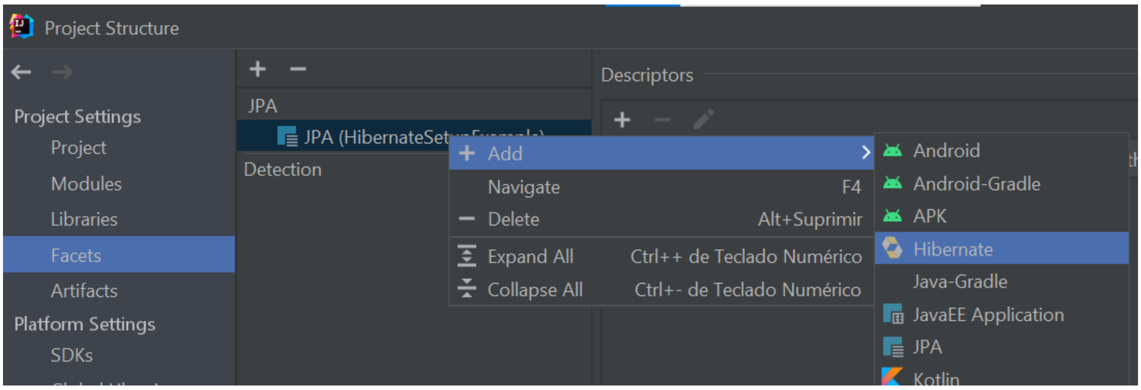
IntelliJ IDEA offers built-in support for Hibernate, but only when using the Ultimate edition. To start our first Hibernate project, we will create a Java project using Maven. Once created, we will add the needed dependencies both to use Hibernate and be able to access our DBMS (PostgreSQL). Project reloading will be necessary (*Maven/Reload project*).

```
<!-- https://mvnrepository.com/artifact/org.postgresql/postgresq
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.20</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.31.Final</version>
</dependency>
```

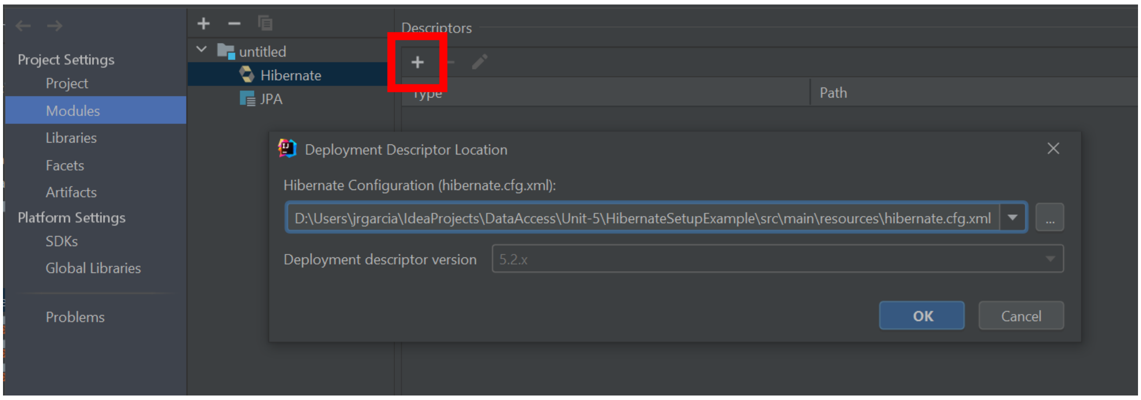
After adding the dependencies, we must create a resources folder. This folder must be inside the 'main' folder and must be marked as a resources folder (in case the folder is not created as a resources folder, it can be later marked as it). The xml files used by hibernate will be created here.



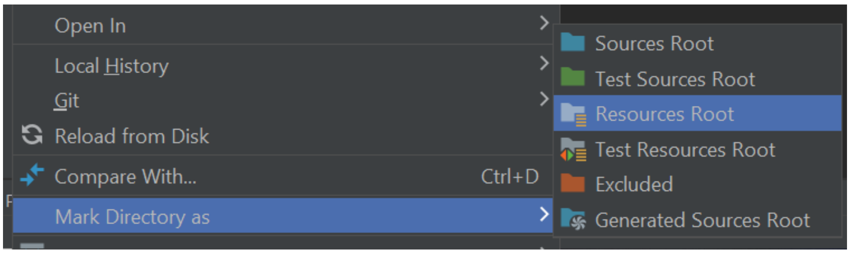
The next step will be to add the Hibernate facet to our project. To do that, we will open the *File/Project Structure* dialog and add the Hibernate facet to our project:



Once done, we will also add the corresponding descriptor, in order to create the Hibernate configuration file:



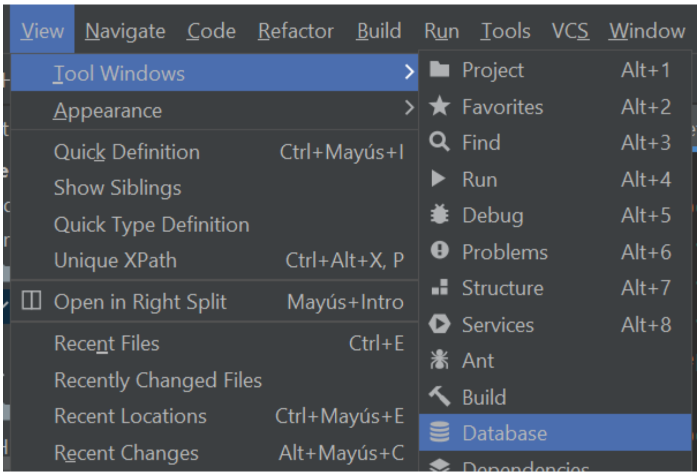
If everything has worked, we will see in our project window that a new file has appeared in the resources folder (`hibernate.cfg.xml`). If the file is generated out of the folder, then you must move it into the folder and marked the folder as a resources folder, to avoid later errors.



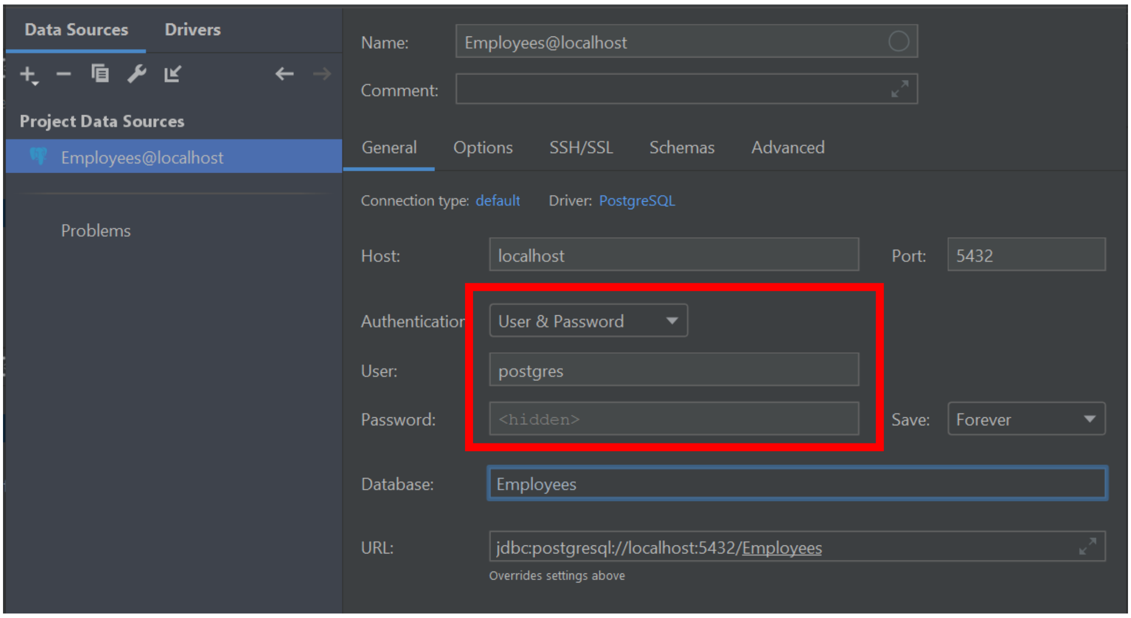
This file (`hibernate.cfg.xml`) is the file used by Hibernate both to keep the database connection and mapping information. Mapping can be made in two different -and exclusive- ways: via XML files or via annotations. We will explain both options, but we will end up using annotations.

4. Mapping entities

We have our project ready to connect to our database. To show how to do this, we will use the `Employee` database used in unit 3. The first step, as seen in unit 3, point 10, is to define the connection to the database, using the *Database tool window*.

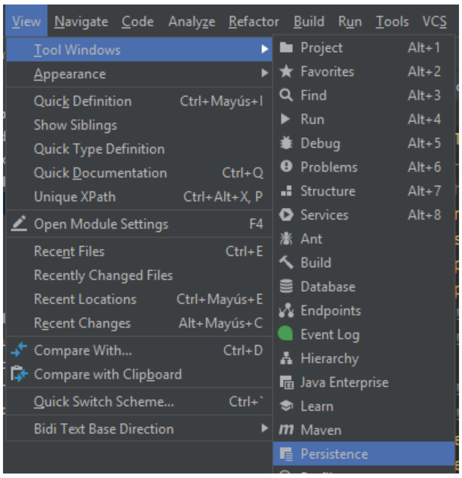


In this window we will create a new connection (+), choosing as data source PostgreSQL, and defining the connection parameters (database name, user and password).

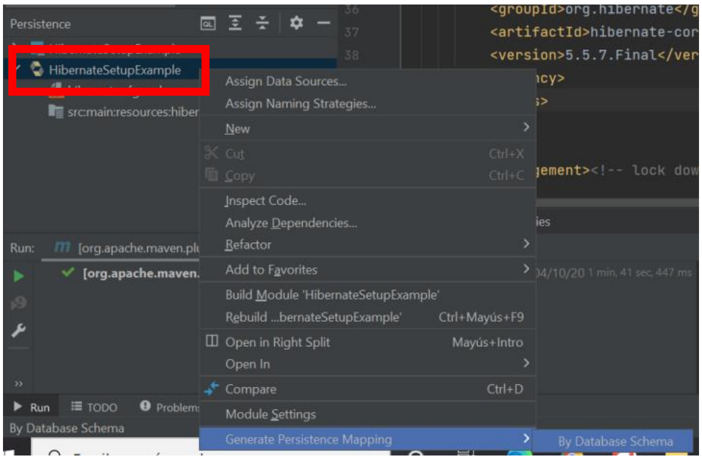


Once the connection has been created, the *Database tool window* can be closed. We can re-open it later if we need to deal with the database, using a console, for example.

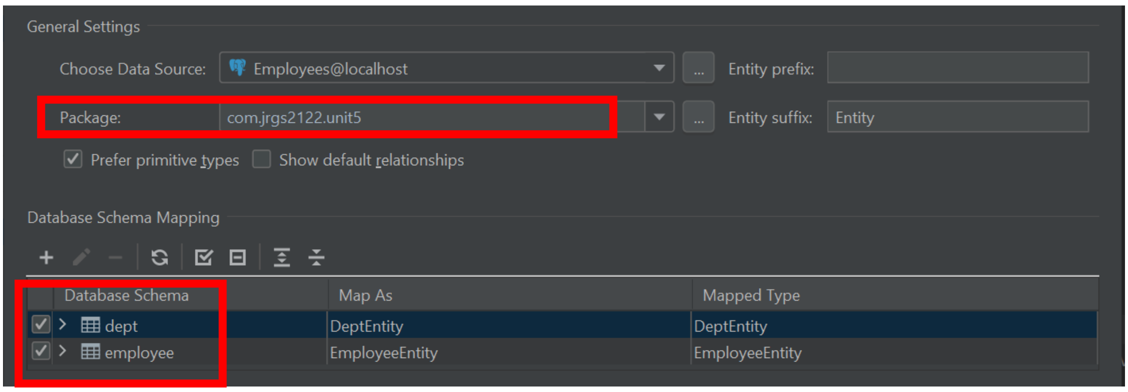
The next step will be to create an XML file to inform Hibernate how to deal with the data stored in the database. We can create this file manually or use the IntelliJ IDEA wizard to do that. This wizard can be found in the *Persistence Tool Window*, that should be made visible.



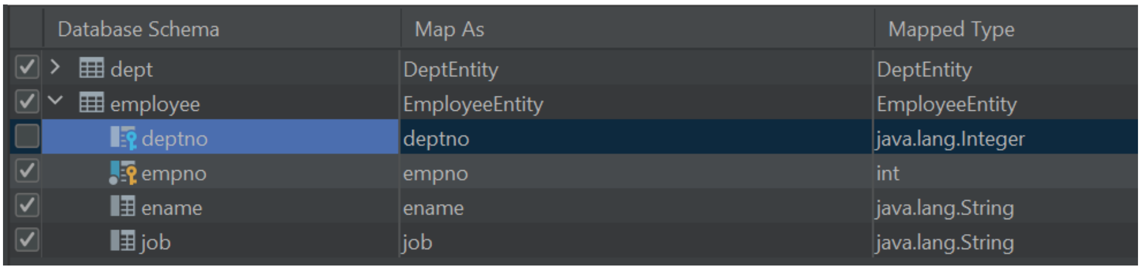
A new tool window will appear in the left side of our window. To define the connection, we will choose the *Generate persistence mapping* option.



In the windows that will appear we will select the connection previously created and pick the needed options. Don't forget to mark every table that must be mapped, and the name of the package.

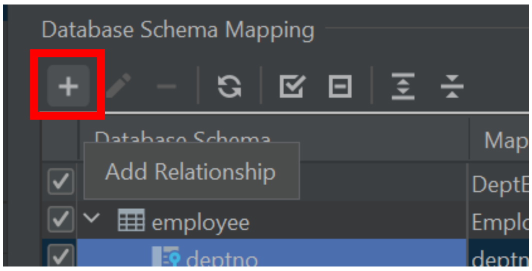


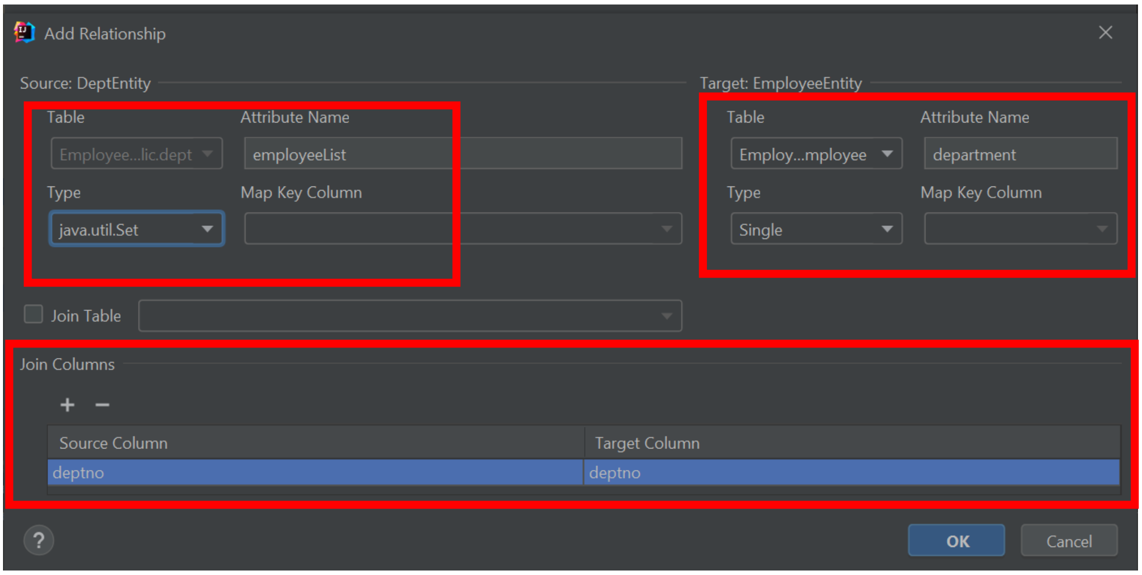
Please note that some attributes (usually foreign keys) are not selected by default and should be explicitly marked.



4.1. Creating one-to-many relationships

If you want a reverse relationship, it must be explicitly added. For example, suppose that we want to have in every department the list of employees that worked in it, we must generate that relationship explicitly:





In this case, we can unmark the *dptno* foreign key, as it's already mapped by the recently created relationship (and, in addition, is mapped as a class, and not simply returning the code of the department).

When pressing 'ok' you will be prompted to pick a directory to create the mapping files, usually the source folder. If everything has worked, you will see two new files in your project, two java classes (known as POJOs, Plain Old Java Objects) with some annotations indicating how to map the table into the entity. A POJO should meet some requirements:

- A constructor without parameters is needed.
- Getters and Setters for every property stored in the database are also required.

4.2. Adjusting the hibernate.cfg file

The configuration file generated by Intellij IDEA may look like this:

```
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:postgresql://localhost:5432/Employees</property>
    <property name="connection.driver_class">org.postgresql.Driver</property>
      <mapping resource="DeptEntity.hbm.xml"/>
      <mapping resource="EmployeeEntity.hbm.xml"/>
      <!-- <property name="connection.username"/> -->
    <!-- <property name="connection.password"/> -->

    <!-- DB schema will be updated if needed -->
    <!-- <property name="hibernate.hbm2ddl.auto">update</property> -->
  </session-factory>
</hibernate-configuration>
```

Here we can see that the connection parameters (username/password) has not been created, so we will need to do this manually.:

```
  <session-factory>
    <property name="connection.url">jdbc:postgresql://192.168.224.125:5432/Emplo
    <property name="connection.driver_class">org.postgresql.Driver</property>
    <property name="connection.username">postgres</property>
    <property name="connection.password">postgres</property>
    <mapping class="com.jrgs2425.aad.entities.Dept"/>
    <mapping class="com.jrgs2425.aad.entities.Employee"/>
```

Let's make a simple application to test if everything is working fine.

4.3. Using entities to manage data

4.3.1. Selecting data

Once the automated process has successfully ended, we can make use of the newly created entities to access the DBMS. The first step will be to create and open a session:

```
SessionFactory sessionFactory =
          new Configuration().configure().buildSessionFactory();
   Session session = sessionFactory.openSession();
   if (session != null) {
       System.out.println("Session successfully opened!");
   } else {
       System.out.println("Error opening session!");
   }
```

The *SessionFactory* class should be instantiated just once in a project. It establishes a 'hard link' with the DBMS. Once we have a *sessionFactory* object, we can create sessions (*Session*) using this object, one session for every job we would like to send to the DBMS.

Once the session is created and open, we can send queries to the database using the generated entities to retrieve the data. Hibernate will transform our query in a SQL sentence and send it to the DBMS. With the data received it will create objects (one per row received) with the retrieved information.

```
Query<EmployeeEntity> myQuery =
         session.createQuery("from com.jrgs2122.Unit5.EmployeeEntity");
List<EmployeeEntity> employees = myQuery.list();

for ( EmployeeEntity employee : employees ) {
     System.out.printf("Number  :  %d   Name:  %s",  employee.getEmpno(),
employee.getEname());
}
```

If we run our project, we should get an output similar to this:

```
oct 22, 2021 5:30:28 P. M. org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.PostgreSQLDialect
oct 22, 2021 5:30:29 P. M. org.hibernate.engine.transaction.jta.platform.internal.JtaP
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.j
Session successfully opened!
Number : 7369  Name: SMITH
Number : 7499  Name: ALLEN
Number : 7521  Name: WARD
```

To suppress the information messages from Hibernate (we will usually do this when the application is running fine) we will use this piece of code, prior to open the session:

```
@SuppressWarnings("unused")
java.util.logging.Logger logger =
       java.util.logging.Logger.getLogger("org.hibernate");
logger.setLevel(Level.SEVERE);
```

4.3.2. Updating data

The process to update a record in the database will be quite similar to the previous one. Firstly, we will retrieve the record that needs updating into an entity, we will update the information in the entity and, finally, we will store it into the database. Note that we have created an specific function to open an Hibernate session.

```
public static SessionFactory sessionFactory = null;

public Session openSession() {
     if ( sessionFactory == null )
         sessionFactory =
              new Configuration().configure().buildSessionFactory();
     Session session = sessionFactory.openSession();
     if (session == null) {
       throw new Exception("Error opening session!");
     }
     return session;
}

public static void updateEmployee( int employeeNumber ) {
     try ( Session session = openSession() ) {
         EmployeeEntity employee =
                 (EmployeeEntity)session.get( EmployeeEntity.class,
                                                 employeeNumber );
         if ( employee != null ) {
             employee.setDeptno(30);
             session.update(employee); // alternatively, session.merge
             }
         else
             System.out.println("Employee not found");
     }
     catch( Exception e ) {
          System.out.println( e.getMessage() );
     }
}
```

4.3.3. Inserting data

To insert data we will do a pretty similar process to the previous one but, instead of retrieving a record from the database, we will create a new object with the information to store. We can also add transactional capabilities to this process:

```
public static void insertDepartment() {
     Scanner scanner = new Scanner(System.in);
     System.out.print("Department name?: ");
     String dname = scanner.nextline();
     System.out.print("Department location?: ");
     String dloc = scanner.nextline();
     Transaction transaction;
     try ( Session session = openSession() ) {
         transaction = session.beginTransaction();
         DeptEntity department = new DeptEntity();
         department.setDeptname( dname );
         department.setLoc( dloc );
         session.save( department ); // alternatively, session.persist
         transaction.commit();  // End of transaction
     }
     catch( Exception e ) {
          transaction.rollback();
          System.out.println( e.getMessage() );
     }
}
```

Note that, if an error -exception- occurs, the transaction is rolled back.

4.3.4. Deleting data

Finally, to delete a record, we firstly retrieve the record from the database (as we did in 4.3.2) and then delete the record.

```
public static void deleteEmployee( int employeeNumber ) {
     try ( Session session = openSession() ) {
         session.beginTransaction();
         EmployeeEntity employee = session.get( EmployeeEntity.class,
              employeeNumber );
          if ( employee != null ) {
             session.delete(employee);
             session.getTransaction().commit();  // End of transaction
             System.out.println("The employee has been deleted.");
          }
          else {
             System.out.println("Employee not found.");
             }
     }
```

```
    catch( Exception e ) {
        System.out.println( e.getMessage() );
    }
}
```

4.4. Activities

1. Taking as the starting point the code in point 4.3, create a CRUD application for both employees and departments. Deleting operations should be confirmed before processing, showing the data to be deleted. CRUD stands for Create, Read, Update and Delete. When displaying the information of a department, the list of employees belonging to the department should be also shown.

```
Name: ACCOUNTING
    Number : 7839   Name: KING
    Number : 7934   Name: MILLER
    Number : 7782   Name: CLARK
```

5. JPA annotations

Hibernate Annotations are based on the JPA 2 specification and support all its features. JPA annotations are defined in the javax.persistence (Hibernate 5 and prior) or jakarta.persistence package (Hibernate 6). Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification.

If we take a look at the generated entities they should look like this:

```
@Entity
@Table(name = "employee", schema = "public", catalog = "Employees")
public class EmployeeEntity {
    private int empno;
    private String ename;
    private String job;
    private DeptEntity department;

    @Id
    @Column(name = "empno", nullable = false)
    public int getEmpno() { return empno; }

    public void setEmpno(int empno) { this.empno = empno; }

    @Basic
    @Column(name = "ename", nullable = true, length = 10)
```

```
@Id
@Column(name = "empno", nullable = false)
public int getEmpno() { return empno; }

public void setEmpno(int empno) { this.empno = empno; }

@Basic
@Column(name = "ename", nullable = true, length = 10)
public String getEname() { return ename; }
```

If we take a closer look to our entities:

```
@Entity
@Table(name = "employee", schema = "public", catalog = "Employees")
public class EmployeeEntity {
```

Here we can see that the *EmployeeEntity* is an entity linked to the table *Employee*, which resides in the *public* schema of the *Employees* database.

```
@Id
@Column(name = "empno", nullable = false)
public int getEmpno() { return empno; }

public void setEmpno(int empno) { this.empno = empno; }
```

Our first property –field- in the *EmployeeEntity* entity is *empno*, linked to the column with the same name in the table. It's the key field (*@Id*) and it's obviously not nullable.

```
@Basic
@Column(name = "ename", nullable = true, length = 10)
public String getEname() { return ename; }
```

The second property is a basic column of the table, with a length of ten characters and it admits null values. We can change this values to match the ones specified in the table.

```
@ManyToOne
@JoinColumn(name = "deptno", referencedColumnName = "deptno")
public DeptEntity getDepartment() { return department; }

public void setDepartment(DeptEntity department) { this.department = department; }
```

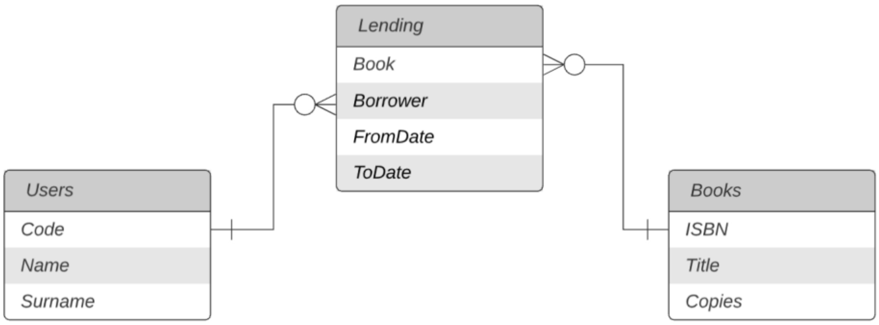
Here we can see that, instead of the foreign key field *deptno*, we have the full *Department* entity as a property. If we check the Department entity, we will find the inverse relationship in this entity as a list of employees:

```
@OneToMany(mappedBy = "department")
public Set<EmployeeEntity> getEmployeeList() { return employeeList; }

public void setEmployeeList(Set<EmployeeEntity> employeeList) { this.employeeList = employeeList; }
```

This annotation maps the *getEmployeeList* property as a one-to-many relationship, connected with the *EmployeeEntity* entity through the *department* property. If we check this property, we will see that is marked with a *@ManyToOne* annotation, actually.

5.1. Many-to-many relationships

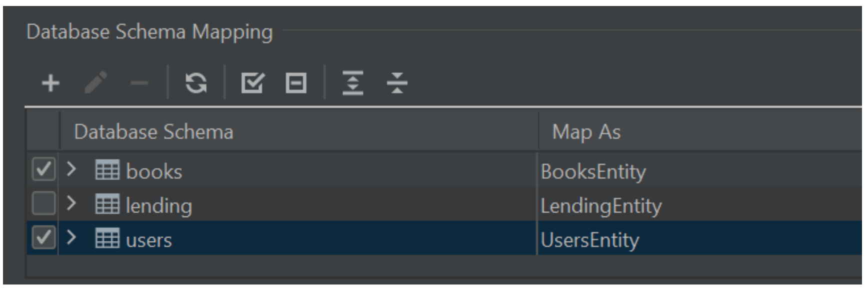
Many-to-many relationships can also be modelled in Hibernate, depending on whether the relationship has its own attributes or not. To show the two approaches available, we will use a very simple database to manage a public library, where books can be lent by registered users. An entity-relationship for this database will be similar to the following one:



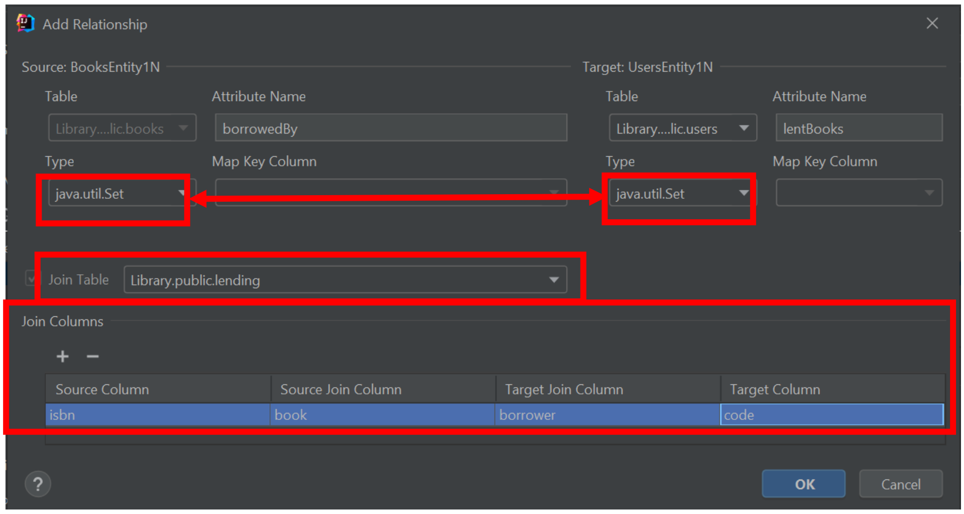
A script to create this database can be downloaded from the moodle platform.

5.1.1. Joining table without attributes

When we model a M:M relationship in a DBMS, a new table containing the primary keys (as foreign keys) of the two tables implied in the relationship must be added. If this table has no more attributes, then we can model this relationship with Hibernate in a pretty much similar way we did in point 4.1. Inthis case, there is no need to include the joining table in the final solution:



We will have to bear in mind that, in this case, every attribute will have a M multiplicity and, thus, must be modelled with a Java collection (or derived). We will also have to provide the joining table and the join columns.

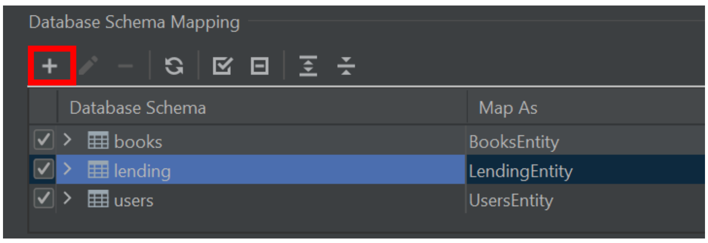


In this case, we will obtain just two entities (users and books), each of them with a set containing the books lent and the borrowers of a specific book. The specific attributes of the lending table would not be accessible. The code obtained (*BooksEntity*) will look like this:

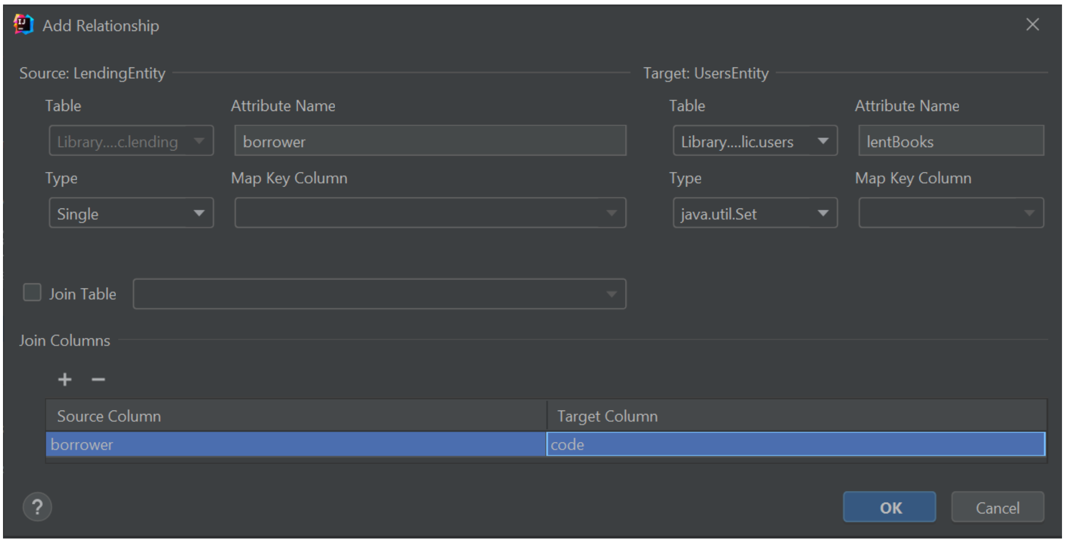
```
@ManyToMany
@JoinTable(name = "lending", catalog = "Library", schema = "public",
    joinColumns = @JoinColumn(name = "book", referencedColumnName =
    "isbn", nullable = false), inverseJoinColumns = @JoinColumn(name =
    "borrower", referencedColumnName = "code", nullable = false))
public Set<UsersEntity> getBorrowedBy() {
    return borrowedBy;
}
```

5.1.2. Joining table with attributes, specific primary key

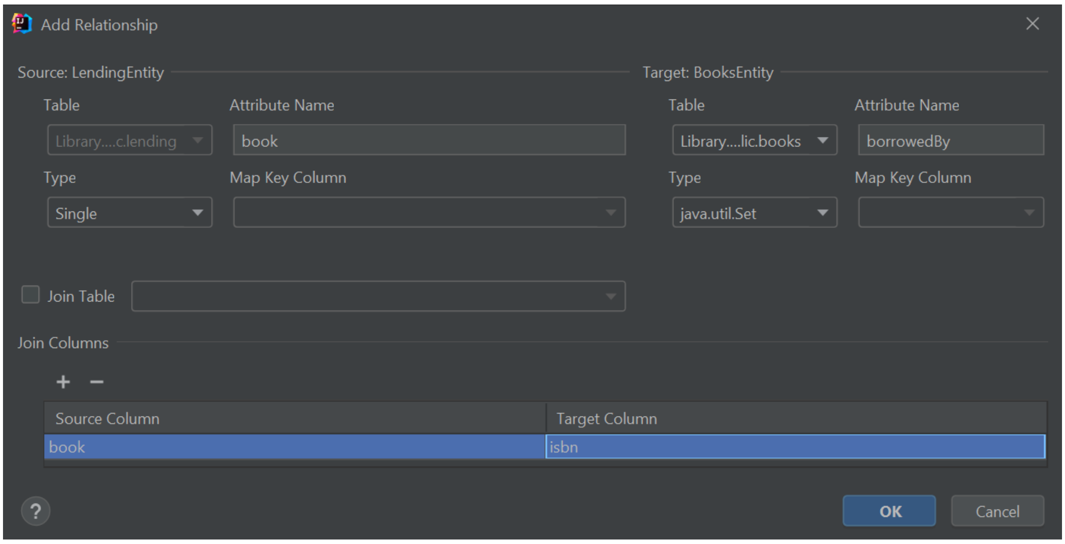
If our joining table has specific attributes (FromDate and ToDate), the previous approach is not valid, as we lost this information in the modelling process. So, in this case, we need to add the joining table to the solution and model two one-to-many relationships instead of a many-to-many relationship. In case the joining table has han specific primary key (is not composed by the fields populated from the tables related) the process is quite easy:



Now we will have to add to one-to-many relationships, one in the books table and one in the users table, both of them pointing to the lending table. Note that in this case we won't mark the joining table check.



Moreover, the multiplicity turns into 1:M, as we saw in point 4.1.

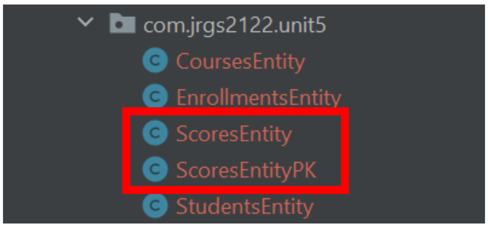


As a result, we will obtain a new table (*lending*) with the two relationships:

```
@ManyToOne
@JoinColumn(name = "borrower", referencedColumnName = "code", nullable = false)
public UsersEntity1N getBorrower() { return borrower; }

public void setBorrower(UsersEntity1N borrower) { this.borrower = borrower; }

@ManyToOne
@JoinColumn(name = "book", referencedColumnName = "isbn", nullable = false)
public BooksEntity1N getBook() { return book; }

public void setBook(BooksEntity1N book) { this.book = book; }
```

5.1.3. Joining table with attributes, populated primary key

Let's suppose now that our joining table takes its primary key from the key fields populated from the related tables. We can see this in the Enrollment-Subjects relationship from our database in Unit 3 Final Activity. The joining table (*scores*) has as a primary key the two populated fields (*enrollment_id, subjects_id*). In this case, when doing the above process, we will obtain not just one table but two: the joining table (*ScoresEntity*) and, linked to this table, we will have another one containing just the primary key (*ScoresEntityPK*):



Let's test our brand new POJOs including the following code in our *main* class:

```
Query<ScoresEntity> myQuery = session.createQuery("from ScoresEntity");
List<ScoresEntity> scores = myQuery.list();

for (ScoresEntity score : scores) {
     System.out.printf("Enrollment : %d  Score: %d\n",
                    score.getEnrollement(), score.getScore() );
}
```

If we run our application and open a new Hibernate session, we will be very likely to get this error:

```
: Repeated column in mapping for entity: com.jrgs2122.unit5.ScoresEntity column: enrollment_id
enConnection(VTIModel.java:21)
tStudentsList(VTIModel.java:36)
```

Hibernate is reporting that we have mapped twice the same table column (*enrollment_id*), once as a primary key, and another as the many-to-one field in a relationship. The workaround is also suggested in the error message: just declare the second mapping as non insertable and non updatable. We'll have to do that with every field in the primary key (*enrollment_id, subject_id*).

```
@ManyToOne
@JoinColumn(name = "subject_id", referencedColumnName = "code", nullable = fals , insertable = false, updatable = false)
public SubjectsEntity getSubjectEnt() { return subjectEnt; }

public void setSubjectEnt(SubjectsEntity subjectEnt) { this.subjectEnt = subjectEnt; }

@ManyToOne
@JoinColumn(name = "enrollment_id", referencedColumnName = "code", nullable = false, insertable = false, updatable = fals
public EnrollmentsEntity getEnrollmentEnt() {
    return enrollmentEnt;
```

As a final consideration, keep in mind that, if you use different suffixes during the automatic process generation, it is possible to generate entities to cover more than one approach (one many-to-many relationship, two one-to many). This can be interesting in those cases where the specific information contained in the joining table is not relevant.

5.2. Activity

1. Write a console application that helps the librarian to rent/return a book from the library database. No more than three books can be held for the same user at the same time. The rental period for a book will be a week. If a book is returned after that period, the system must inform the librarian.

6. Hibernate Query Language (HQL)

HQL or Hibernate Query Language is the object-oriented query language of Hibernate Framework. HQL is very similar to SQL except that it use objects instead of table names, emphasizing then the main aim of an ORM. The second important advantage of using HQL is that we make our code independent of the DBMS and its native SQL.

HQL is case-insensitive, except for java class and variable names. So `SeLeCT` is the same as `sELEct` is the same as `SELECT`, but `com.jrgs.model.Employee` is not same as `com.jrgs.model.EMPLOYEE`.

You can find the full HQL specification in the following link:

https://docs.jboss.org/hibernate/orm/6.4/querylanguage/html_single/Hibernate_Query_Language.html

6.1. HQL Queries

To create a query using HQL we use the `session` method `createQuery`.

```
Query<Employee> myQuery =
        session.createQuery("from com.jrgs2122.Unit5.Employee");
List<Employee> employees = myQuery.list();
```

As we can see, in this query we use the clause `from` from SQL but using as the goal an entity instead of a table. We can add a `where` clause to this query:

```
Query<Employee> myQuery =
        session.createQuery("from com.jrgs2122.Unit5.Employee" +
```

```
                                   "where deptno = 10");
```

Or, even better, use a parameter to build our query:

```
Query<Employee> myQuery =
         session.createQuery("from com.jrgs2122.Unit5.Employee" +
                             "where deptno = :deptno");
myQuery.setParameter("deptno", 10);
```

Of course, it's also possible to use sorting and aggregation functions:

```
Query<Employee> myQuery =
         session.createQuery("from com.jrgs2122.Unit5.Employee" +
                             "order by empno");
```

If you have mapped your relationships with entities, you can also take advantage of this when writing querys that use more than a table (joins). The use of alias is also supported:

```
Query<Employee> myQuery =
       session.createQuery("from com.jrgs2122.Unit5.Employee e" +
                           " join fetch e.department d where " +
                           " d.name like  'SALES'");
```

6.2. Native Queries

Native queries are a tool that allows the user to express database-specific queries, in case you want to use features such as query hints or the CONNECT keyword in Oracle, for example. Hibernate allows you to specify handwritten SQL sentences, including stored procedures/functions calls, for all create, update, delete, and load operations.

Your application will create a native SQL query from the session with the *createNativeQuery()* method on the Session interface:

```
public  Query  createNativeQuery(String  sqlString,  Class  resultClass)
throws HibernateException
```

Suppose we want to call the stored function we create in unit 3, point 7.4.2, to obtain the list of employees belonging to a specific department, using a native query. We can do it like that:

```
public List<Employees> getEmployeesByDepartment(String dptName) {
   // Suppose the session object has been previously created
   String sqlString = "SELECT employee_list_by_department(:dptname)";
   return session.createNativeQuery(sqlString, Employee.class)
            .setParameter("dptname", dptName)
            .getResultList();
}
```

It is also possible to call functions not returning a SETOF but scalar values using a different method signature:

```
public    Query    createNativeQuery(String    sqlString)    throws
HibernateException
```

Usando el método `getSingleResult()` se puede obtener un `Object` que posteriormente se puede transformar al tipo de dato adecuado.

6.3. Named Queries

When we are developing a database application, one of the major disadvantages is the fact of having the database calls (SQL/HQL) scattered across the code. To avoid this, in Hibernate we can use named queries, grouping the queries with the corresponding entity that is suppose to be returned.

Let's take a look at the example queries in point 6.1. Everyone is returning a set of employees. So we can rewrite this queries as named queries:

```
@Entity
@Table(name = "employee")
@NamedQuery(
      name = "Employee.findEmployeesByDeptno",
      query = "from com.jrgs2122.Unit5.Employee e" +
              "where e.deptno = :deptno"
)
@NamedNativeQuery(
      name = "Employee.findEmployeesByDeptName",
      query = "SELECT employee_list_by_department(:dptname)",
      resultClass = Employee.class
)
public class Employee {
    @Id
    @Column(name = "empno", nullable = false)
    private Integer id;
```

To invoke a named query, we do it through the session object:

```
public List<Employee> findEmployeesByDeptName(String dptName) {
    return session.createNamedQuery("Employee.findEmployeeByDeptName",
                                    Employee.class)
                        .setParameter("dptname", dptName)
                        .getResultList();
}
```

7. Customizing our entities

8.1. @Transient annotation

Our entities, in addition to map a database table, are classes itself. Thus, we can add them any field needed by our application. We only have to annotate all this fields with `@Transient`, pointing out to Hibernate that the field annotated shouldn't be mapped to the database.

8.2. Nested relationships

As we have seen in points 4 and 5, the one-to-many and the many-to-many relationships can be represented in our entities as collections. But, how does Hibernate deal with this collections?

By default, Hibernate does not retrieve inmediately the collection(s) associated with an entity, because of performance considerations. This behaviour is called the *lazy fetching*, that means that collections will be retrieved only when needed. Despite that this is quite good in terms of performance, it implies that the session used to retrieve the entity can not be closed if we want to lately retrieve the nested collection.

This drawback can be bypassed by indicating that the collection should be retrieved at the same time that the entity is. This is the `eager` fetch mode. We should use it with special care, as it should have a big impact on performance.

```
@OneToMany(mappedBy = "book", fetch = FetchType.EAGER)
public List<LendingEntity> getBorrowedBy() { return borrowedBy; }
```

As we can see in the previous example, the list of `LendingEntity` will be retrieved inmediately. We can do that safely if we know that this list won't keep many elements.

8.3. @Where annotation

Another option to improve the performance is to use the `@Where` annotation. Annotated over an entity, will restrict the entities retrieved to those who matches the specified clause. Annotated over a relationship, will restrict the entities in the collection to those who match the clause. Let' see an example:

```
@Entity
@Table(name = "lending", schema = "public", catalog = "Library")
@Where(clause = "returningdate is null")  // This annotation allows to filter entities
public class LendingEntity {              // that match the given clause
```

In the example above, lending entities will only be retrieved if the returning date is *null*, that is, the user has the book at home. We can also remove the annotation from the entity and place it over the relationship:

```
@OneToMany(mappedBy = "borrower")
@Where(clause = "returningdate is null")
public List<LendingEntity> getLentBooks() { return lentBooks; }
```

In this case, we will be able to get historical reports of lendings, while the user entity only will have the active lendings.

8.4. @Filter and @FilterDef

More powerful than using the *@Where* annotation is the *@FilterDef* and *@Filter* annotations. Filters allow us to create parametrized restrictions to access our entities. Let's see an example:

```
@Entity
@Table(name = "customer", schema = "public", catalog = "dvd2324")
@FilterDef(name = "customerStoreFilter", parameters = @ParamDef(name = "storeNumber", type = "integer"))
@Filter(name = "customerStoreFilter", condition = "store_id = :storeNumber")
```

In the example above we have an entity *customer.* This customer can belong to any of a set of stores. We can use a filter to decide which store we would like to access. First we use *@FilterDef* to define the filter name and the parameters, and then we define the filter by specifying the condition. This filter must be applied to the session in which we want it to work:

```
session.enableFilter( customerStoreFilter )
                        .setParameter( "storeNumber", 2 );
```

With the code above we are indicating that we will use the store with id = 2 in this session.

8.5. @DynamicInsert and @DynamicUpdate

By default, Hibernate send to the database every field in the object, even when the value field is null. If the field in the database has a *DEFAULT* constraint, it's better not to send the *null* values to establish the field value via the constraint.

We can achieve that annotating the entity with the `@DynamicInsert` annotation. We can also use the `@DynamicUpdate` annotation to avoid sending values that haven't changed since they were retrieved (improving, thus, the performance).

8. JPA Annotations table

Annotation	Modifier	Description
`@Entity`		Marks a class as a Hibernate Entity (Mapped class)
`@Table`	`name`	Maps this class with a database table specified by `name` modifier. If `name` is not supplied it maps the class with a table having same name as the class
`@Id`		Marks this class field as a primary key column
`@GeneratedValue`	`strategy`	Instructs database to generate a value for this field automatically. A usual value for `strategy` is `GenerationType.IDENTITY`
	`generator`	Uses a sequence object of the database. The name of the generator must be provided.
`@Column`	`name`	Maps this field with table column specified by `name` and uses the field name if `name` modifier is absent
`@ColumnDefault`	`value`	Allows to specify a default value for the column.
`@CreationTimestamp`		Specifies current date as the default value for a Date/Time field.
`@OneToMany`	`mappedBy`	Maps this field as the owning side of a one-to-many relationship. The modifier `mappedBy` holds the field which specifies the inverse side of the relationship
	`fetch`	Defines when Hibernate gets the related entities from the database. It's a very important modifier in terms of efficiency considerations. The possible values are `FetchType.EAGER` (fetch it so you'll have it when you need it) and `FetchType.LAZY` (fetch it when you need it, used by default).

@ManyToOne		Mark this field as the 'many' side of a one-to-many relationship. Must be used together with the @OneToMany annotation
@JoinColumn	name	Maps a join column specified by the *name* identifier in the one-to-many relationship
	referencedColumnName	Identifies the owning side of the column which is necessary to identify a unique owning object
@ManyToMany	cascade	Marks this field as the owning side of the many-to-many relationship and *cascade* modifier specifies which operations should cascade to the inverse side of relationship
	mappedBy	This modifier holds the field which specifies the inverse side of the relationship
@JoinTable	name	For holding this many-to-many relationship, maps this field with an intermediary database join table specified by *name* modifier
	joinColumns	Identifies the owning side of columns which are necessary to identify a unique owning object
	inverseJoinColumns	Identifies the inverse (target) side of columns which are necessary to identify a unique target object
@JoinColumn	name	Maps a join column specified by the *name* identifier to the relationship table specified by *@JoinTable*
@LazyCollection		Instructs hibernate about the fetching mode. To set the EAGER mode, we will use *LazyCollectionOption.FALSE*
@Where	clause	Allows to add a filter when retrieving the POJOs, can be used either on a relationship or directly on a POJO. *Clause* is a regular SQL Where clause.

For a complete list of JPA annotations, you can check this link:

https://www.javaguides.net/2018/11/all-jpa-annotations-mapping-annotations.html

Also, to check the specific Hibernate annotations, please visit this one:

https://dzone.com/articles/all-hibernate-annotations-mapping-annotations

9. Bibliography

Object–relational impedance mismatch - Wikipedia

Object–relational mapping - Wikipedia

Java Persistence/OneToMany - Wikibooks, open books for an open world

Java Persistence/ManyToMany - Wikibooks, open books for an open world

Jakarta Persistence - Wikipedia

Hibernate ORM 5.4.32.Final User Guide (jboss.org)

HQL full specification

JavaFX official documentation

JavaFX documentation (with examples and support for Intellij IDEA)