DAM. UNIT 3. ACCESS USING OBJECT-RELATIONAL MAPPING (ORM). ACCESS TO RELATIONAL DATABASES USING HIBERNATE CLASSIC

DAM. Acceso a Datos (ADA) (a distancia en inglés)

Unit 3. ACCESS USING OBJECT-RELATIONAL MAPPING (ORM)

Access to relational databases using Hibernate Classic

Abelardo Martínez

Year 2024-2025

1. Introduction

When we need to explain or capture a complex reality, we often resort to simplification by constructing a simpler conceptual model that behaves similarly to reality. The aim is to capture the essential aspects and, at the same time, to lighten the insignificant details in order to reduce complexity. The use of conceptual models during the implementation of software applications is extremely important for the success of any computerisation project.

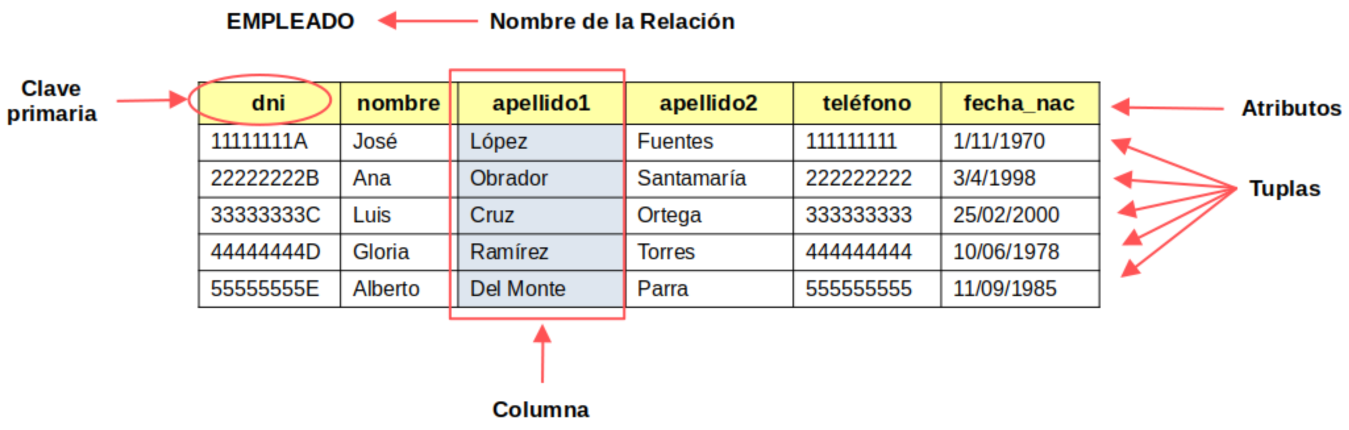
The problem is that conceptual models are mental representations resulting from a process of abstraction. There is no single way of capturing or representing them outside our brain. We often use schematic approximations that can come very close to mental representation, but even schematic representations are hardly representable in a computer's memory.

Relational systems translate the model into entity-relationship diagrams, which serve as the basis for defining the schema of tables and relationships needed to support the casuistry to be represented.

1.1. Relational model

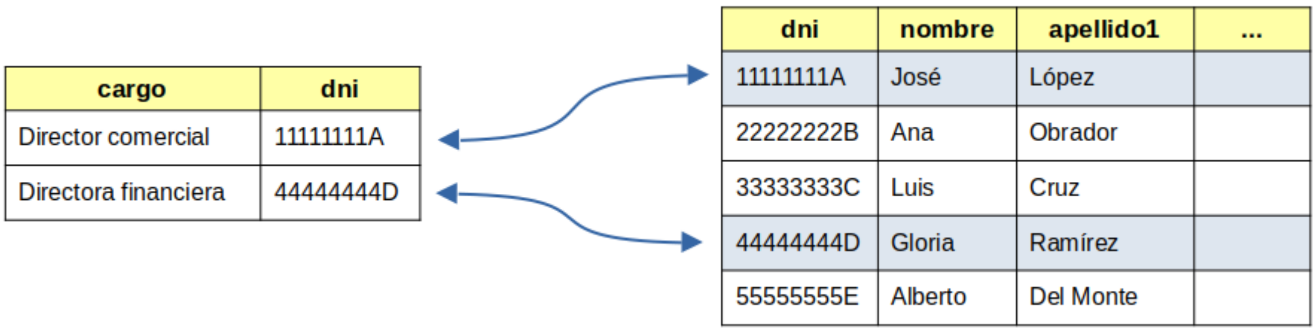
The relational model is a way of organising the data of an application by grouping them according to the relationship that can be established between them in accordance with the model. This grouping is materialised in the form of a **table**, where we distinguish the **columns** or set of data that represent the same concept of the data model and the **tuples or records** that represent entities differentiated from the application.

Tables keep the data corresponding to each record well related, according to the concept they represent (ID, name, address, etc.). The records are identified by means of the value of a column or a set of columns called **primary key**. The value of the primary key can never be repeated, so that there is a unique correspondence between records and keys (Figure 1.1).



dni	nombre	apellido1	apellido2	teléfono	fecha_nac
11111111A	José	López	Fuentes	111111111	1/11/1970
22222222B	Ana	Obrador	Santamaría	222222222	3/4/1998
33333333C	Luis	Cruz	Ortega	333333333	25/02/2000
44444444D	Gloria	Ramírez	Torres	444444444	10/06/1978
55555555E	Alberto	Del Monte	Parra	555555555	11/09/1985

In complex models, multiple tables will be necessary, the records of which can also be related from foreign keys (Figure 1.2). **Foreign keys** identify records from other tables (from their primary key) so that it is easy to identify all data related to a certain entity even though they belong to different tables.



cargo	dni
Director comercial	11111111A
Directora financiera	44444444D

dni	nombre	apellido1	...
11111111A	José	López	
22222222B	Ana	Obrador	
33333333C	Luis	Cruz	
44444444D	Gloria	Ramírez	
55555555E	Alberto	Del Monte	

The relational model also allows defining a set of rules and limitations on data values and actions to be performed, in order to ensure data consistency. Thus, it is possible to indicate what to do with the records of a table that are linked to the record of a second table at the time of deleting it. It also allows, for example, to define the range or set of possible values that a field in a table may take, or to ensure the non-repetition of certain values in different records of the same table, etc.

1.2. Object-oriented model

Object technology has recently made a major effort to regulate and standardise schematic representations of conceptual models. The result of these efforts has led to a schematic but very expressive language called UML (**U**nified **M**odeling **L**anguage). Languages such as UML are able to describe most conceptual models very accurately without the need for many adaptations, because they are semantically rich and expressive.

Objects can represent any element of the conceptual model: an entity, a characteristic, a process, an action, a relationship, etc. **Objects** are the result of an abstraction process focused on the important characteristics (**data**), but also on the behaviour or functionality that they will have when they materialise during the execution of the applications (**code**).

The importance of focusing the model on objects is therefore manifold.

a) In reference to the data, the objects act as hierarchical structures, so that the information is always perfectly contextualised within the objects. From the object-oriented paradigm it does not make sense to refer to an isolated variable. That is, the data will always be perfectly related to each other in a similar way to the relationship that is established in a table in the relational paradigm. The difference, however, lies in the fact that the relational perspective only maintains this relationship within the table, while the object-oriented perspective extends it to the entire application by incorporating it into the execution code itself.

b) In reference to behaviour or functionality, objects delimit the actions to be performed on their data and on the rest of the objects, defining the rules of the game of what is allowed during the execution of the applications. The set of data that make up an object is also called **state**, because it allows describing the evolution of any object during the execution of an application, from the moment of its creation until it is removed from memory.

In this paradigm, **the state of objects plays a fundamental role** because it is the common thread that runs through the execution of applications. The initial data, calculations and results will always take the form of object state during execution, which will evolve in a coordinated manner in favour of the application's objectives, as interactions between the objects themselves take place.

Interaction is performed by making calls to object methods (operations available according to types or class). In other words, data is not manipulated directly, but state changes or queries are requested from the owners of the information. In the OO (Object-Oriented) paradigm, the interaction between objects is completely governed by the type of relationships that we establish in the model. We will speak of one object being related to another when the former can access the methods of the latter. The relationships defined in the OO model will indicate which types of objects will be able to interact and how.

Unfortunately, the conceptual model is an eminently dynamic model that does not contemplate, a priori, the persistence of its objects. This, which at first glance may seem to be a defect of the paradigm, is in fact an opportunity to implement libraries and persistence frameworks that work in a way that is completely transparent to the model and its operation. **Persistence frameworks** will take care of staging objects by initialising them in the state in which they were stored at the time of instantiation, while the model logic contained in the objects' methods will evolve them from this initial state. Persistence frameworks shall periodically store the evolution of objects as changes occur, so that there is always a correspondence between objects in memory and their stored states.

1.3. The object-relational impedance mismatch

The main problem we find when trying to automate the process of persistence of the objects of an application in a DBMS under the relational paradigm, is that they are different conceptualisations and also focused on different aspects. While the **ER paradigm** is strongly centred on the data and the structure that must be given to them in order to store and retrieve them according to the conceptual model, the **OO paradigm** is centred on the objects, understood as groupings of data and also as processes of change.

It should be noted that **the relational model will always need a certain amount of extra information to maintain the relationships and consistency of the data**. This is not the model's own information, but has to be added to ensure proper functioning. Foreign keys are the clearest example. This is information added to some records to link them to others. **Linking between objects**, on the other hand, **is achieved structurally**. **No extra data is needed**, but the data structure itself defines linkage, visibility, access, etc.

The relational model, although it can also store some procedures called at the programmer's will (procedures) or associated with certain actions (triggers), uses an implementation and usage logic that is much less obvious than the simple coding of methods in the classes of the OO model.

These differences constitute what is known in the programming world as the object-relational impedance mismatch. This gap refers to the obvious difference between how an object-oriented application treats information and how a relational database works.

While relational databases treat information as relationships and sets (it is still a mathematical model), the object-oriented programming paradigm deals with instances of classes (objects) and the relationships between them. So, every time we want to read (or write) information from a relational database from our OO application we need a process of "adaptation" of it. This gap forces us, when we decide to work together with both paradigms, to code extra implementations that work as adapters.

In this unit we are going to work with Relational Databases from Java applications but, unlike previously, we are going to work directly on the object-relational impedance mismatch.

2. Object-relational mapping

Although relational database connection systems such as ODBC or JDBC allow a great deal of expressiveness, which makes them very powerful tools, they are low-level tools that unfortunately require a significant number of lines of code to meet the needs of each application. Developers have to continue to strive to design two models (relational and object-oriented) and have to continue to create translation tools between them, at a very significant cost.

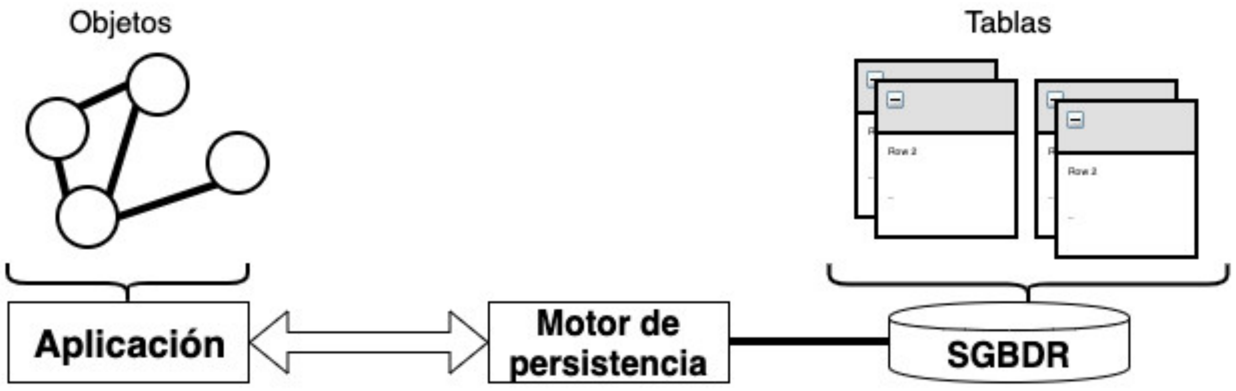
These difficulties inherent to the great difference between both paradigms, makes the programming of applications with access to relational databases can be arduous, complex and an important source of errors due to the casuistry that is not contemplated in the design of the applications.This is why object-relational mapping tools or ORM tools were born.

Object-**R**elational **M**apping (**ORM**) **tools** attempt to take advantage of the maturity and efficiency of relational databases by minimising the object-relational mismatch as much as possible. **These tools are responsible for "translating" the data representations in the relational model to a representation of classes and objects that help us in the development of our application**. They are libraries and programming frameworks that define a format for expressing multiple transformation situations between the two paradigms. This allows them to automate handover processes from one system to the other. In a way, we could say that they implement a virtual object-oriented database because they provide characteristics of the OO paradigm, but the substrate where the objects end up being stored is a relational DBMS.

2.1. ORM concept

As mentioned above, object-relational mapping (ORM) tools automate the necessary data exchange processes between OO and relational systems. **Object-relational mapping** is a **programming technique** for converting data between the type system used in an object-oriented programming language and that used in a relational database, using a persistence engine. Automation is achieved through a set of metadata describing which process to use and what correspondence there is between the primitive data of the two systems and the structures that support them.

In practice, this creates a virtual object-oriented database that acts as an interface between the relational database and the object-oriented application. This enables the use of object-oriented programming features.



2.2. ORM tools

The number of mapping tools that we can find today is really high. There are many free and commercial tools available and some programmers even choose to develop their own mapping tools. As a result, there are mapping tools for most object-oriented languages such as PHP, C++, C#, and, of course, Java. Basically they are tools in which we can distinguish three clearly differentiated conceptual aspects:

1. A system for expressing the mapping between classes and the database schema. ORM tools allow us to create a data access layer. A simple and valid way to do this is to create a class for each table in the database and map them one by one (similar to what we studied in unit 2).
2. An object-oriented query language to neutralise the O-R mismatch. These tools provide their own query language, totally independent of the database, which will allow us to migrate from one database to another without making any modifications to the code. It will only require minor changes in the configuration file.
3. A functional core that enables the synchronisation of the application's persistent objects with the database.

Advantages

Some of the advantages of these tools are:

- They reduce software development time.
- Abstraction of the database.
- Reusability.
- They allow the production of better code.
- They are independent of the DB, they work in any DBMS.
- Own language to carry out queries.
- They encourage the portability and scalability of software programs.

Drawbacks

- The major drawback is that the applications require somewhat more computational power because all database queries must first be transformed from the tool's own language to that of the DBMS.

2.2.1. Mapping techniques

Among the techniques that these tools use to create O-R maps, we distinguish between those that embed the definitions within the code of the classes and those that store the definitions in independent files. The former are usually techniques that are closely linked to the programming language, for example, in C++ macros are usually used and in Java annotations are used. Definition techniques based on code-independent files are usually based on XML, because it is a very expressive and easily extensible language.

Normally most tools support both mapping techniques and even allow them to coexist within the same application. Techniques that use the programming language itself to embed the mapping within the code have a lower learning curve for developers experienced in the host language, but on the other hand it will not be possible to take advantage of the definitions if you decide to switch programming languages.

Using XML-based techniques, however, it is possible to reuse definitions for different languages, provided that the tool used is available in the required programming language or if the format of the definitions follows a standard specification.

2.2.2. Query language

The query language will depend on the ORM tools used, although there is usually some similarity with SQL, as both are non-procedural query languages. However, the query language in an ORM is entirely object-oriented. That is, the query components are expressed using object-oriented syntax and the results returned return objects or collections of objects.

2.2.3. Synchronisation techniques

Synchronisation with the database is probably one of the most critical aspects of mapping tools. They are usually quite complex processes, where sophisticated programming techniques are involved to discover the changes that objects undergo, to create and initialise the new instances to be implemented within the application according to the stored data or to extract the information from the objects to revert it to the DBMS tables. This is probably the aspect that most differentiates the tools from each other.

The main techniques used are **pre-compilation**, **post-compilation** and **reflection**. The programming language supported by the tool will greatly influence the solutions adopted to solve the synchronisation. For example, C++ supports precompilation but not reflection, while Java supports postcompilation and reflection.

2.3. Java Persistence API (JPA)

Jakarta Persistence, also known as **JPA** (abbreviated from formerly name **J**ava **P**ersistence **A**PI) is a Jakarta EE application programming interface specification that describes the management of relational data in enterprise Java applications.

Main features:

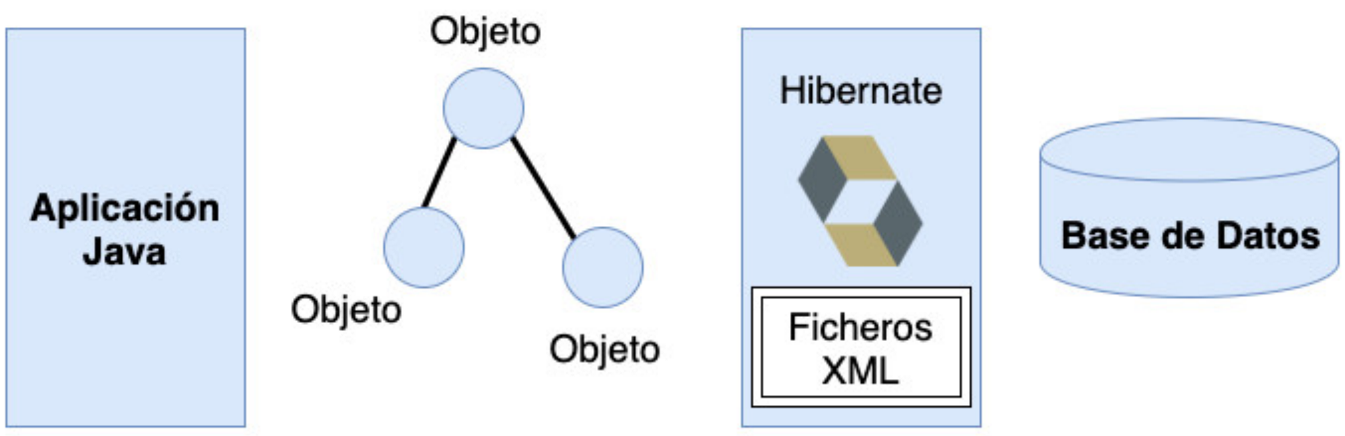
- **JPA supports the persistence of any type of object**. Persistent objects are only required to be serialisable and therefore implement the **Serializable** interface.
- **The mapping of classes can be configured using Java annotations or XML files**. Here we will look at both ways. In fact, JPA can use both configurations at the same time. In case of conflict, JPA gives priority to XML definitions because it is considered that annotations are typically used during the development phase and XML files during the maintenance and modification phases of applications. It should be noted that annotations require code to be compiled, whereas XML specifications do not.
- **JPA is based on the reflection technique**. This allows you to know the name and structure of classes, the names by which their elements are identified, etc. JPA uses this information as default metadata when generating databases just as if it were the metadata described in the mapping. JPA can act by default in the vast majority of cases, so that the configuration required to perform the mapping will be as minimal as possible. This is known as **configuration by exception**. That is, the mapping only needs to be implemented in case the default settings do not match the desired results. Obviously, this system can save developers a lot of work.

2.4. ORM with Hibernate

There are a wide variety of ORM tools on the market. Of all of them, we are going to deal specifically with **Hibernate** as it is developed with Java technology. Hibernate is free software under GNU LGPL license which has made it one of the most popular and widespread ORMs worldwide.

HIBERNATE

Hibernate is an object-relational mapping tool for the Java platform that facilitates the mapping of attributes between a relational database and the object model of an application, by means of declarative files (XML) that allow relationships to be established.



Hibernate is becoming the de facto standard for persistent storage when we want to make the business layer independent from the information storage layer. This persistence layer allows the Java programmer to abstract the particularities of a given database by providing classes that will encapsulate the data retrieved from table rows. Hibernate seeks to bridge the gap between the data models used to organise and manipulate data: the object model provided by the programming language and the relational model used in databases.

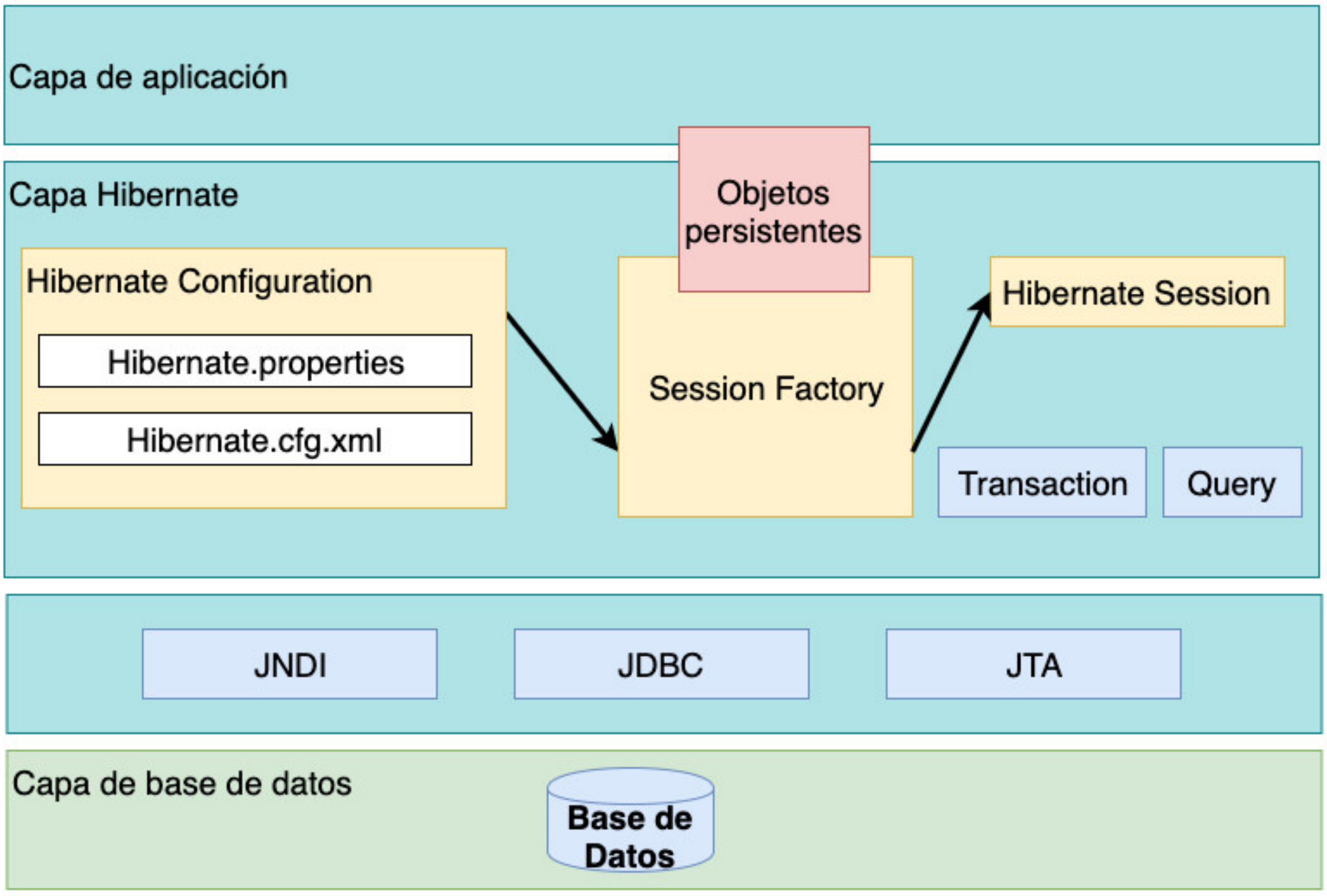
With Hibernate we will not usually use SQL to access the data, but the Hibernate engine itself, through the use of the factory (Factory design pattern) and other programming elements, will build these queries for us. Hibernate provides the programmer with a language called **HQL** (**H**ibernate **Q**uery **L**anguage) that allows access to data through OOP

(Object-oriented Programming).

3. Hibernate architecture

Hibernate is based on a philosophy of mapping normal Java objects, better known as "**POJO**s" (**P**lain **O**ld **J**ava **O**bjects). To store and retrieve these objects from the database, the developer must maintain a conversation with the Hibernate engine by means of a special session (Session class) that could be compared to the concept of connection in JDBC. In the same way that happens in JDBC connections, we have to create and close the session when it is no longer necessary.

The following figure shows the architecture of a standard Hibernate application:



3.1. Session

We can understand a **session** as a cache or a collection of objects loaded (to or from the database) related to a single unit of work.

The **Session class** (**org.hibernate.Session**) offers methods to interact with the DB in a similar way to a JDBC connection, except that it is simpler (for example, to save an object we simply invoke the session.persist(myObject) method, without the need to specify a SQL statement):

Method	Description
persist(Object object)	Saves an object into the DB
createQuery(String query)	Creates a query to read information from the DB
beginTransaction()	Starts a transaction in the DB
close()	Closes the session
Etc.	

A session instance consumes very little memory and is very cheap to create and destroy (in computational terms). This is important because, due to the idiosyncrasy of Hibernate, an infinite number of sessions are created and destroyed during an execution.

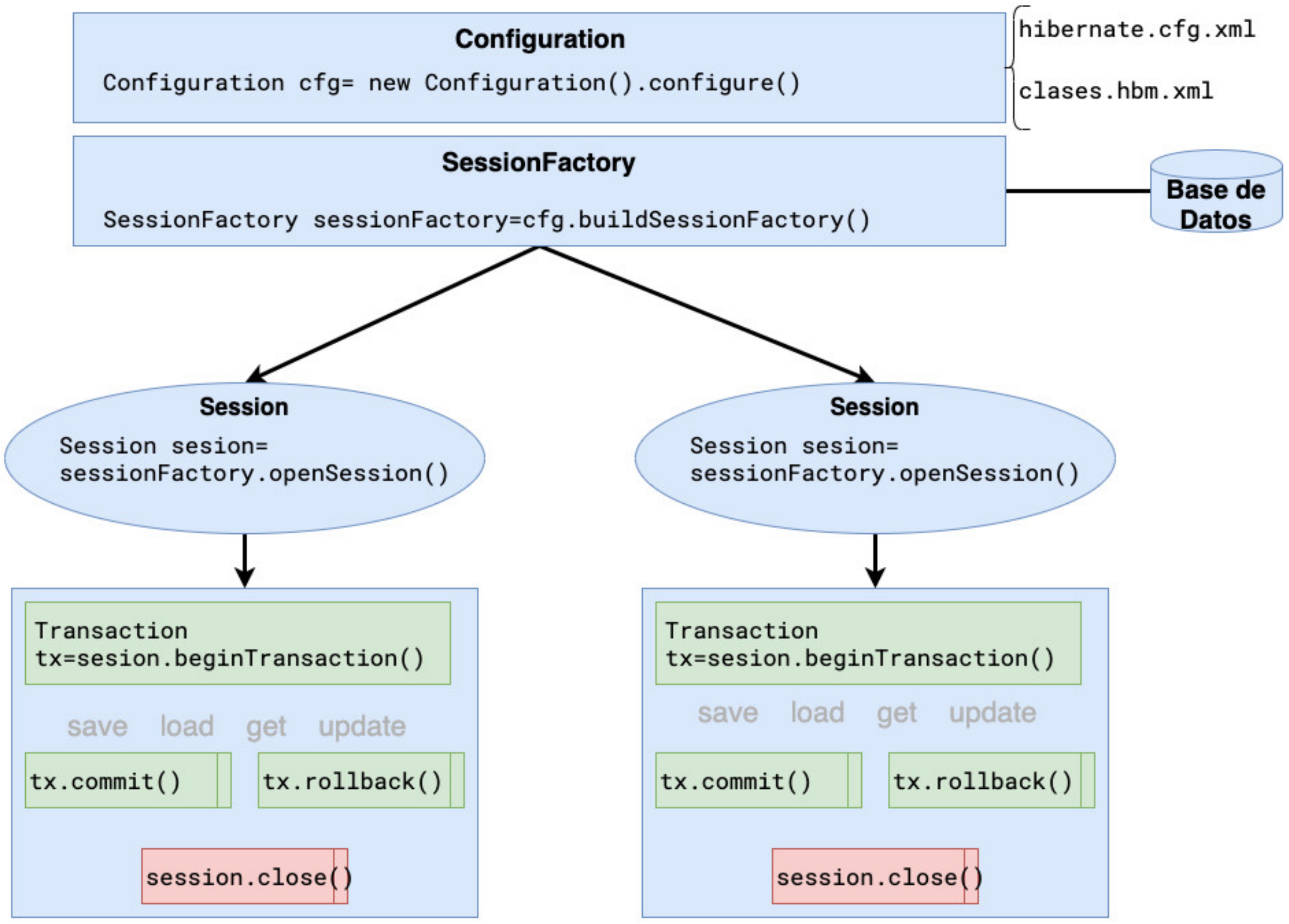
3.2. Interfaces

The hibernate interfaces are as follows:

Interface	Description
Configuration (org.hibernate.cfg.Configuration)	Used to configure Hibernate. The application uses an instance of Configuration to specify the location of documents that specify the mapping of Hibernate-specific objects and properties, and then the SessionFactory is created.
Query (org.hibernate.Query)	Allows queries to be performed on the database and controls how those queries are executed. Queries are written in HQL or in the native SQL dialect of the DBMS being used. A Query instance is used to bind the query parameters, limit the number of results returned and to execute the query.
Transaction (org.hibernate.Transaction)	Allows us to ensure that any error that occurs between the start and end of a transaction causes the transaction to fail (Rollback).

3.3. APIs

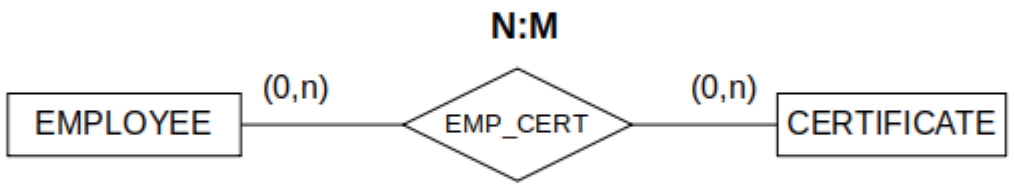
Hibernate makes use of Java APIs such as JDBC, JTA (Java Transaction API) and JNDI (Java Naming Directory Interface). In the following figure we can see the structure of a Hibernate application together with the interfaces we will use.


Configuration
Configuration cfg= new Configuration().configure()
hibernate.cfg.xml
clases.hbm.xml
SessionFactory
SessionFactory sessionFactory=cfg.buildSessionFactory()
Base de Datos
Session
Session sesion= sessionFactory.openSession()
Session
Session sesion= sessionFactory.openSession()
Transaction tx=sesion.beginTransaction()
save load get update
tx.commit()
tx.rollback()
session.close()
Transaction tx=sesion.beginTransaction()
save load get update
tx.commit()
tx.rollback()
session.close()

4. Setting up the project and the database

4.1. The (MySQL) database

For now, just create a simple database with these tables to represent a N:M relationship between EMPLOYEE and CERTIFICATE.



We create the database and the user:

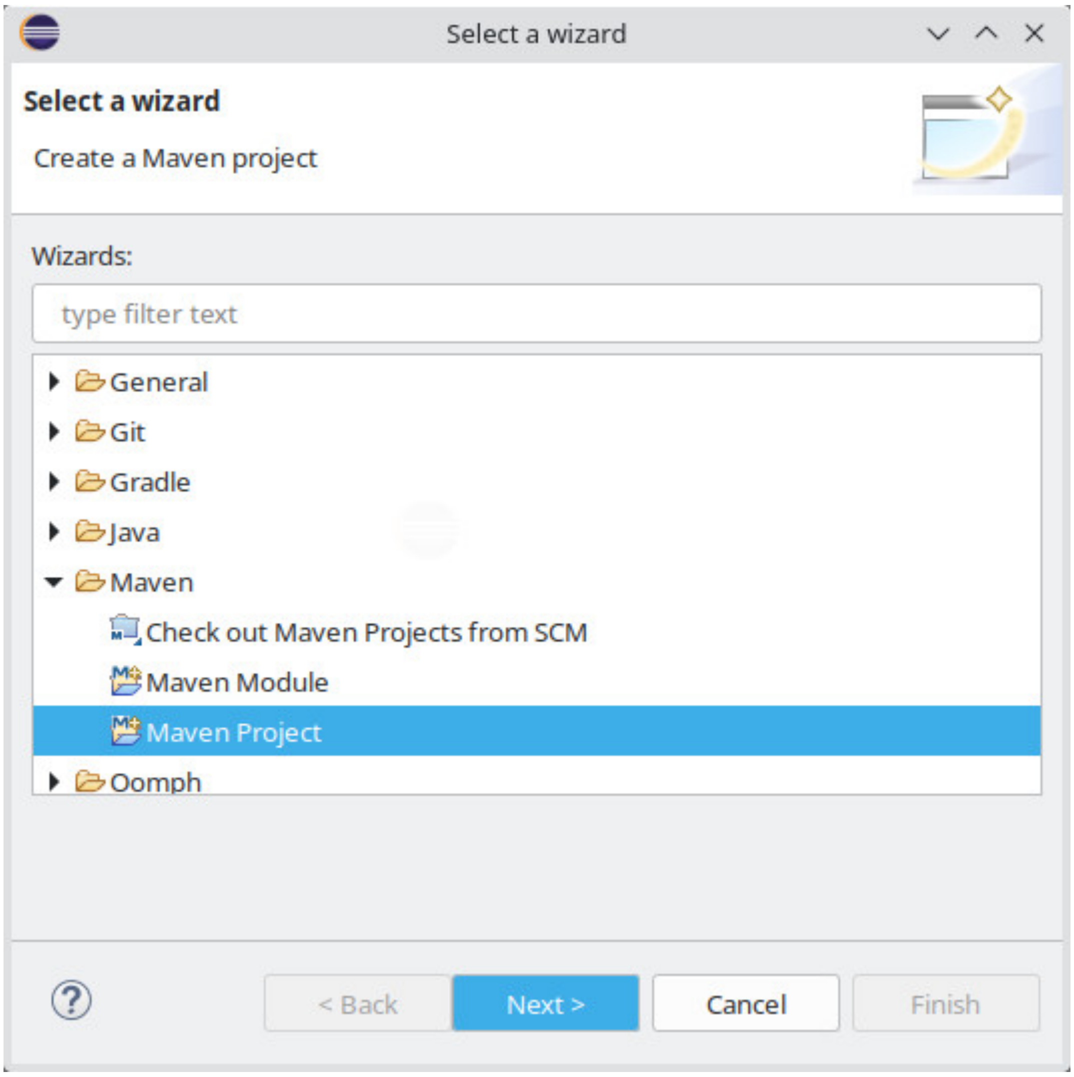
```
CREATE DATABASE ADAU3DBExample CHARACTER SET utf8 COLLATE utf8_spanish_ci;

CREATE USER mavenuser@localhost IDENTIFIED BY 'ada0486';
GRANT ALL PRIVILEGES ON ADAU3DBExample.* to mavenuser@localhost;
```

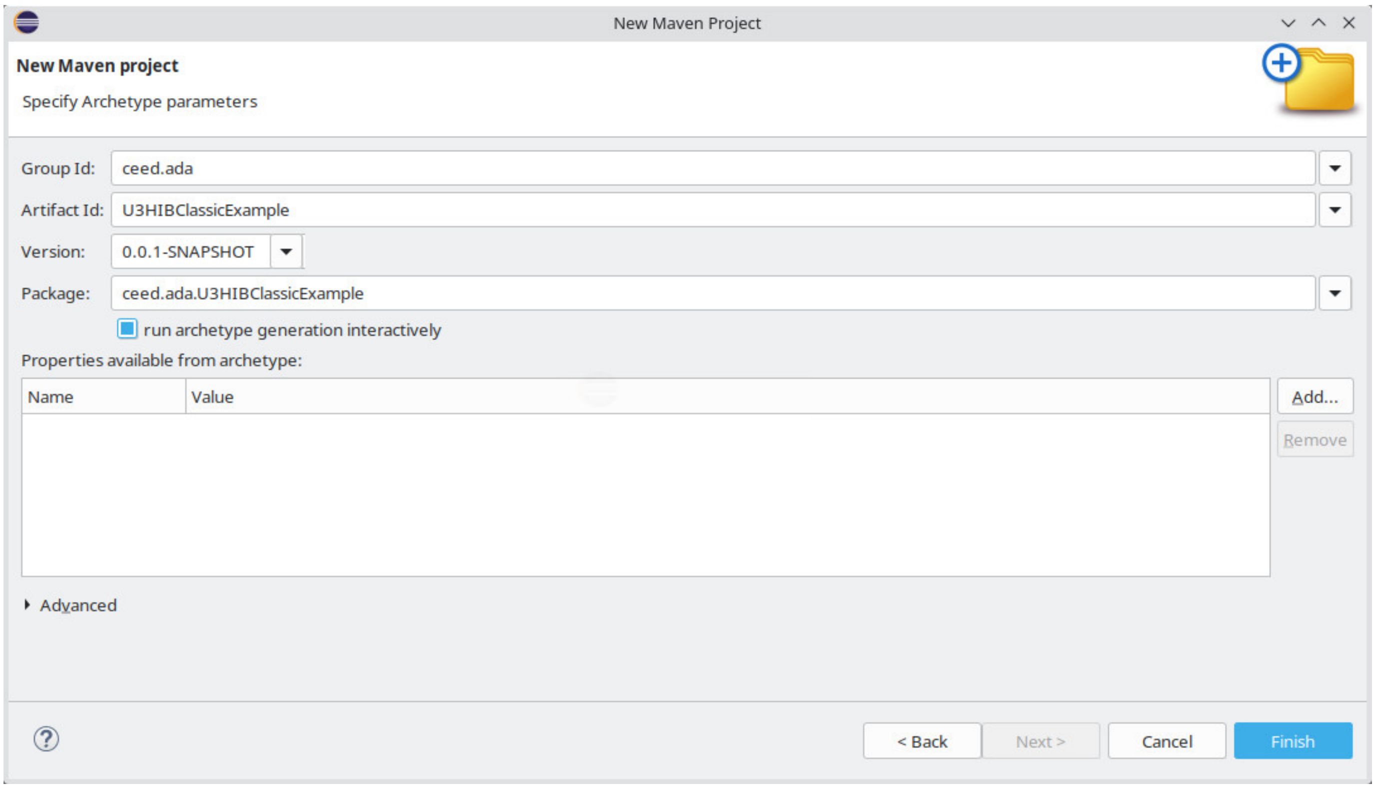
We create the structure:

```
USE ADAU3DBExample;

CREATE TABLE Employee (
   empID        INTEGER NOT NULL AUTO_INCREMENT,
   firstname    VARCHAR(20),
   lastname     VARCHAR(20),
   salary       DOUBLE,
   CONSTRAINT emp_id_pk PRIMARY KEY (empID)
);

CREATE TABLE Certificate (
   certID      INTEGER NOT NULL AUTO_INCREMENT,
   certname    VARCHAR(30),
   CONSTRAINT cer_id_pk PRIMARY KEY (certID)
);

CREATE TABLE EmpCert (
   employeeID     INTEGER,
   certificateID  INTEGER,
   CONSTRAINT empcer_pk PRIMARY KEY (employeeID, certificateID),
   CONSTRAINT emp_id_fk FOREIGN KEY (employeeID) REFERENCES Employee(empID),
   CONSTRAINT cer_id_fk FOREIGN KEY (certificateID) REFERENCES Certificate(cer
);
```

4.2. The (Java-Maven) project

In ECLIPSE, create an empty Java Maven Project with these parameters as we saw at UNIT 2:





New Maven Project

New Maven project

Specify Archetype parameters

Group Id: ceed.ada

Artifact Id: U3HIBClassicExample

Version: 0.0.1-SNAPSHOT

Package: ceed.ada.U3HIBClassicExample

run archetype generation interactively

Properties available from archetype:

Name	Value

4.3. The (Database) connection

Add the dependencies to the Maven Project (pom.xml) for MySQL. Check how to get your version here: https://phoenixnap.com/kb/how-to-check-mysql-version

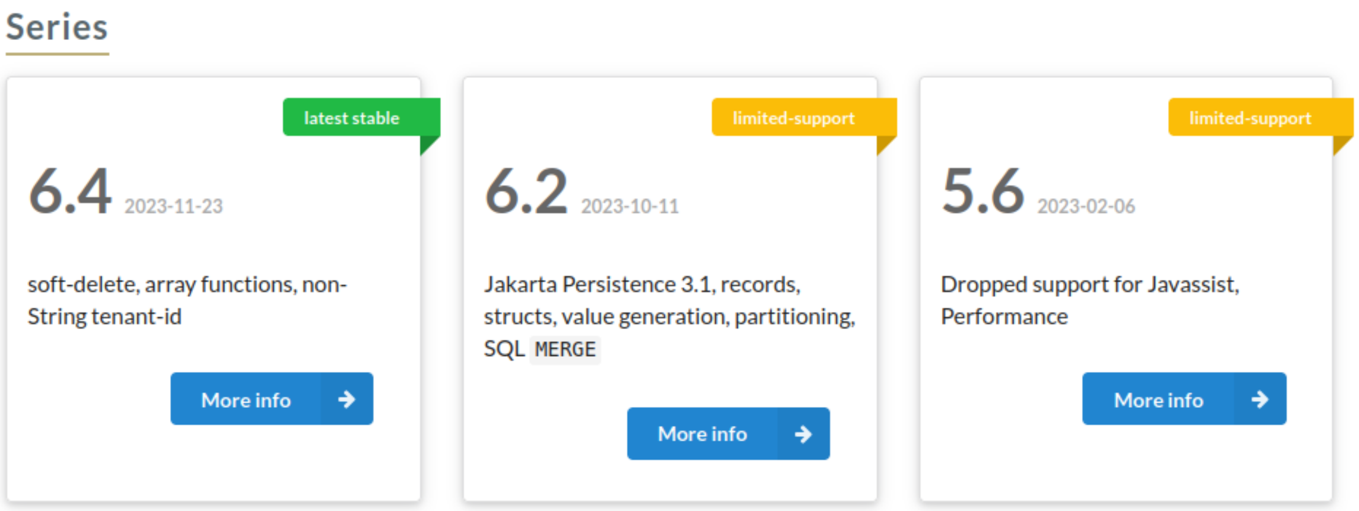
```xml
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
        <!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
        <dependency>
       <groupId>com.mysql</groupId>
       <artifactId>mysql-connector-j</artifactId>
       <version>8.0.33</version>
        </dependency>
  </dependencies>
```

5. Setting up Hibernate

5.1. Hibernate dependencies

Add the dependencies to the Maven Project (pom.xml) for Hibernate. Follow these simple steps to download Hibernate:

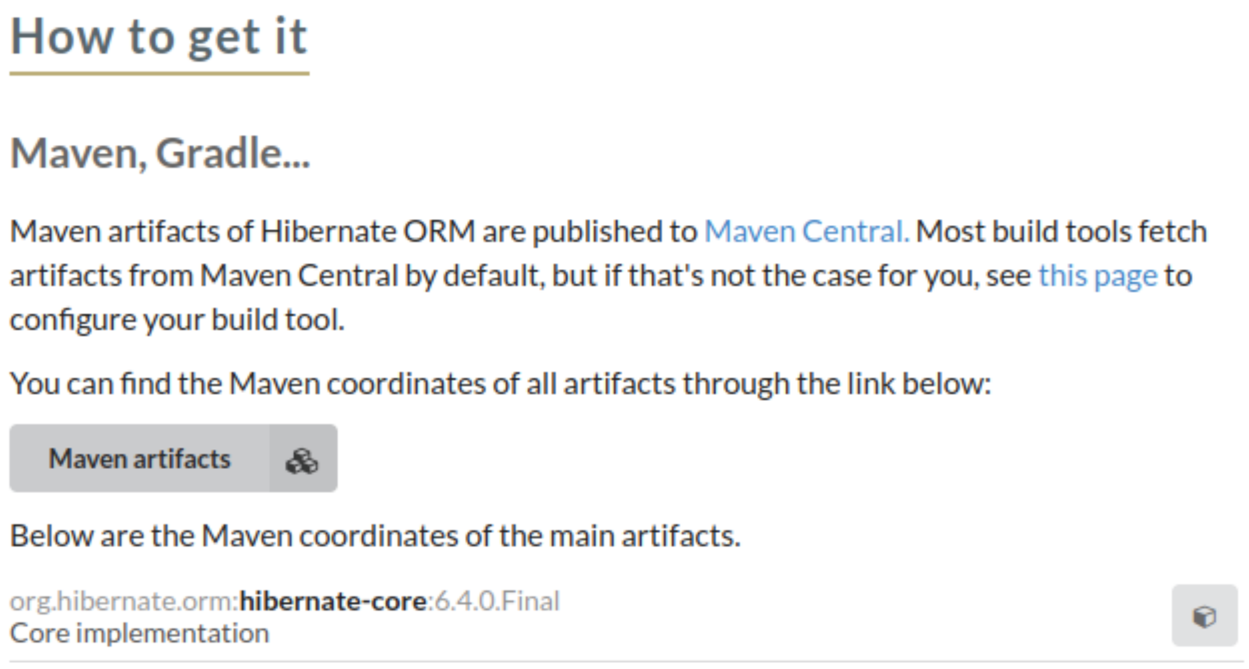
- Go to https://hibernate.org/orm/releases/
- Click on “More info” on the last stable version
  - Option 1:
    - Click on the link hibernate-core (x.x.x.Final)
  - Option 2:
    - Click on “Maven artifacts” button
    - Click on the link hibernate-core (x.x.x.Final)
    - Copy and paste the dependency to your POM


Series

latest stable
6.4 2023-11-23
soft-delete, array functions, non-String tenant-id

limited-support
6.2 2023-10-11
Jakarta Persistence 3.1, records, structs, value generation, partitioning, SQL `MERGE`

limited-support
5.6 2023-02-06
Dropped support for Javassist, Performance



How to get it

Maven, Gradle...

Maven artifacts of Hibernate ORM are published to Maven Central. Most build tools fetch artifacts from Maven Central by default, but if that's not the case for you, see this page to configure your build tool.

You can find the Maven coordinates of all artifacts through the link below:

Below are the Maven coordinates of the main artifacts.

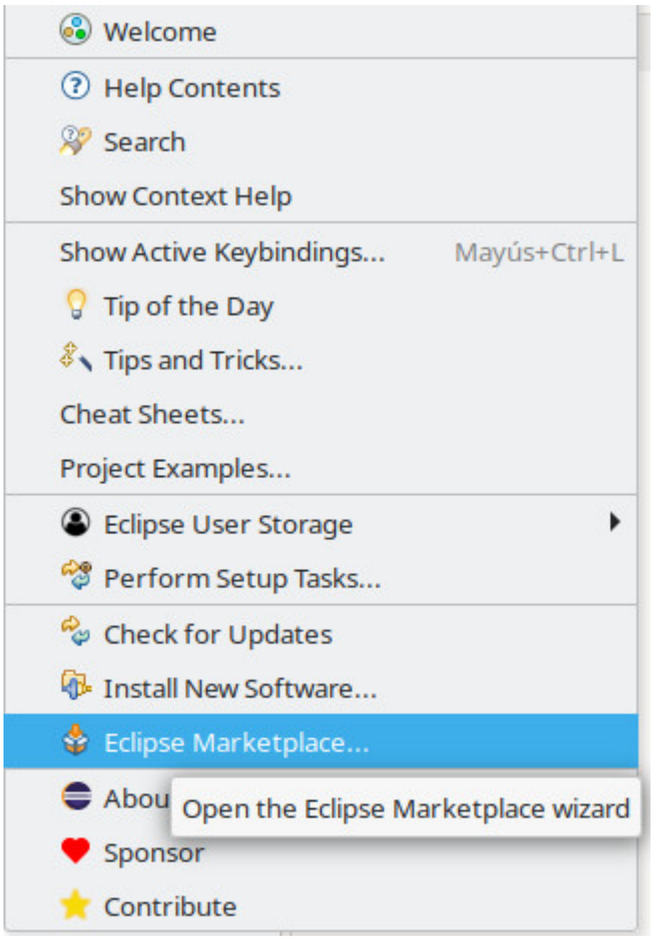
org.hibernate.orm:**hibernate-core**:6.4.0.Final
Core implementation

For example:

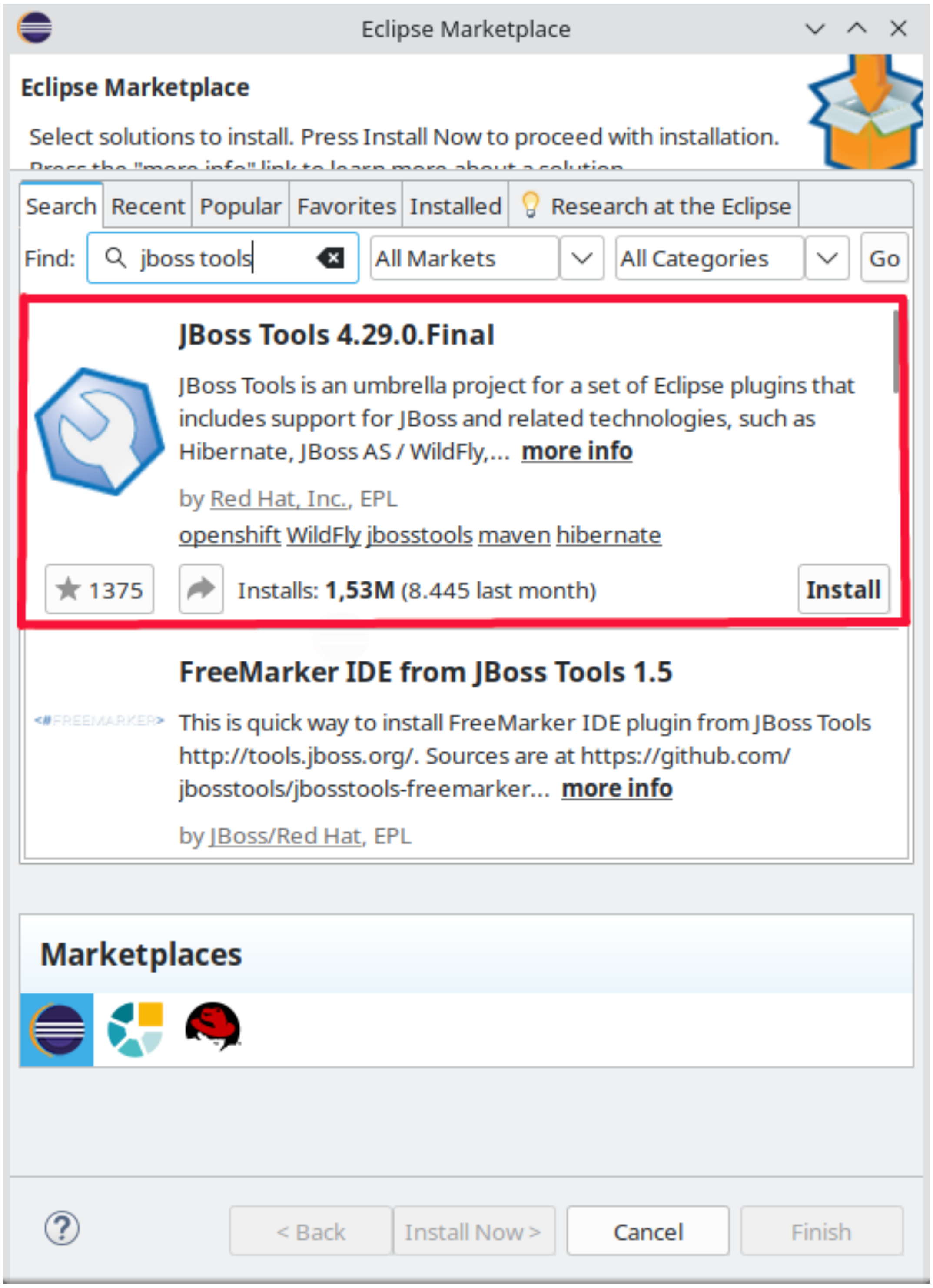
```
<!-- https://central.sonatype.com/artifact/org.hibernate.orm/hibernate-core/6.
<dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.4.0.Final</version>
</dependency>
```

5.2. Installing the plugin

- Open Eclipse and go to the horizontal menu option **Help→ Eclipse Marketplace**.



- Go to the **Find** field and type **jboss tools**:



Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.

Find: jboss tools

JBoss Tools 4.29.0.Final

JBoss Tools is an umbrella project for a set of Eclipse plugins that includes support for JBoss and related technologies, such as Hibernate, JBoss AS / WildFly,... **more info**

by Red Hat, Inc., EPL

openshift WildFly jbosstools maven hibernate

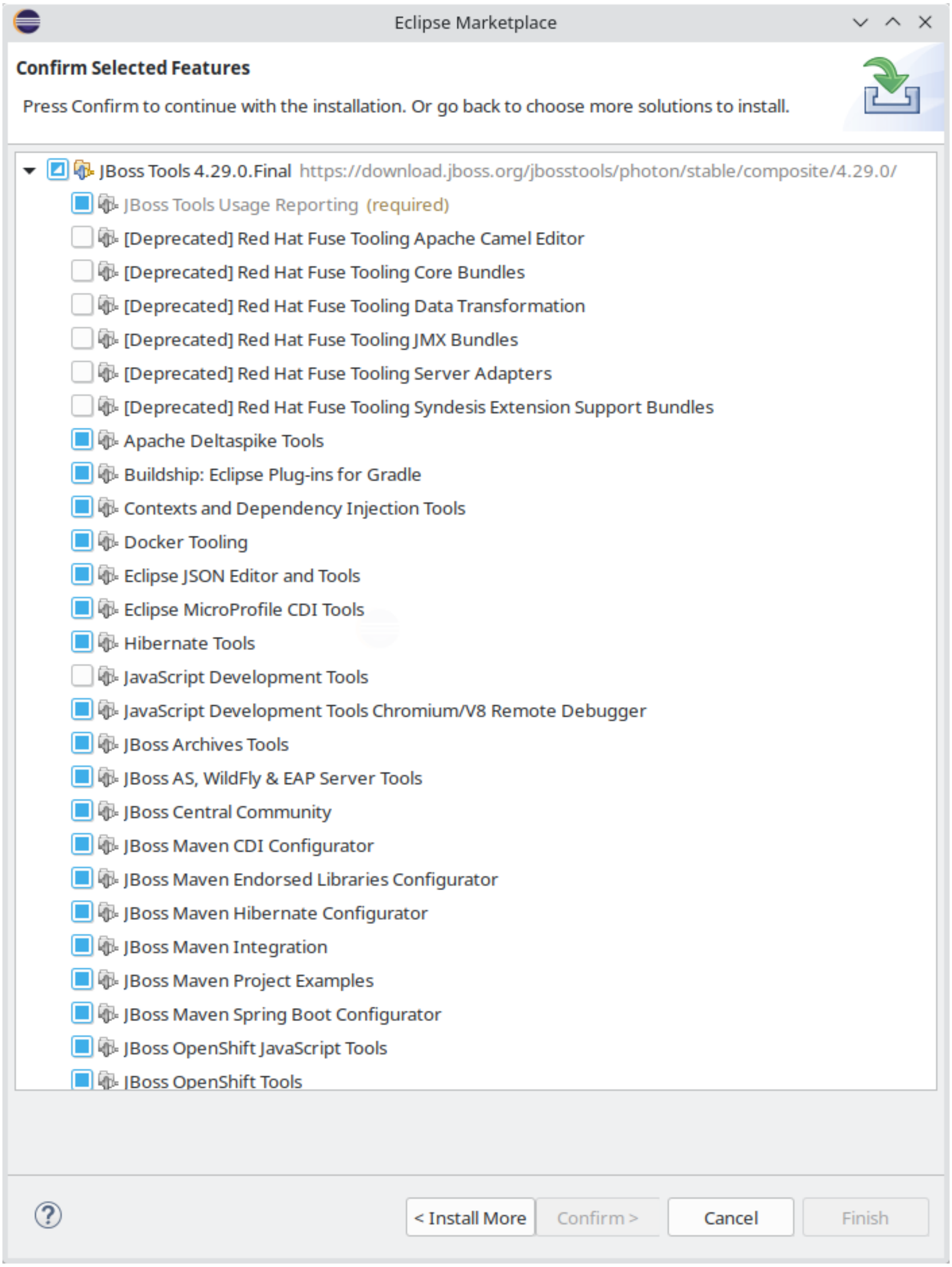
1375 Installs: **1,53M** (8.445 last month)

FreeMarker IDE from JBoss Tools 1.5

This is quick way to install FreeMarker IDE plugin from JBoss Tools http://tools.jboss.org/. Sources are at https://github.com/jbosstools/jbosstools-freemarker... **more info**

by JBoss/Red Hat, EPL

- Click on Install. Confirm all options and validate data sources as trusted:

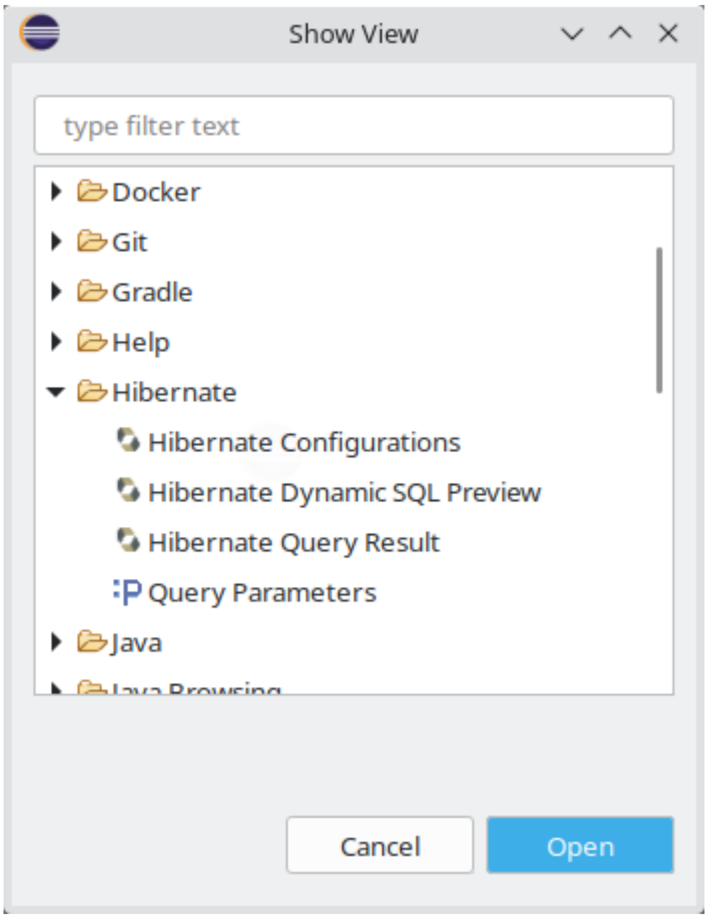

Eclipse Marketplace

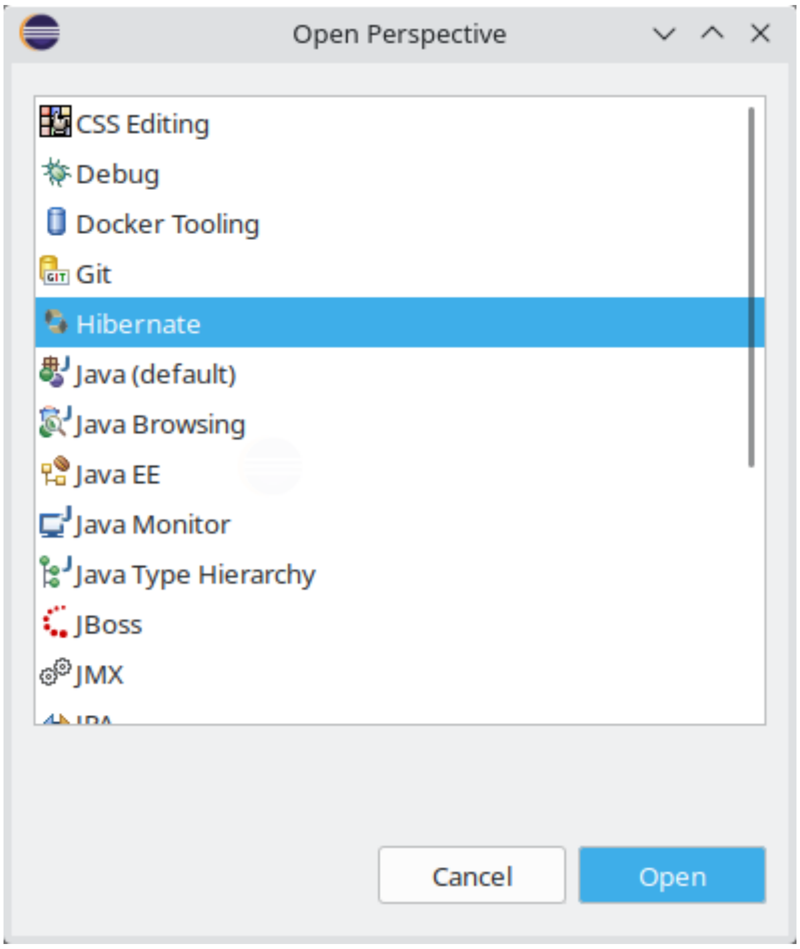
Confirm Selected Features

Press Confirm to continue with the installation. Or go back to choose more solutions to install.

- [✓] JBoss Tools 4.29.0.Final https://download.jboss.org/jbosstools/photon/stable/composite/4.29.0/
  - [✓] JBoss Tools Usage Reporting (required)
  - [ ] [Deprecated] Red Hat Fuse Tooling Apache Camel Editor
  - [ ] [Deprecated] Red Hat Fuse Tooling Core Bundles
  - [ ] [Deprecated] Red Hat Fuse Tooling Data Transformation
  - [ ] [Deprecated] Red Hat Fuse Tooling JMX Bundles
  - [ ] [Deprecated] Red Hat Fuse Tooling Server Adapters
  - [ ] [Deprecated] Red Hat Fuse Tooling Syndesis Extension Support Bundles
  - [✓] Apache Deltaspike Tools
  - [✓] Buildship: Eclipse Plug-ins for Gradle
  - [✓] Contexts and Dependency Injection Tools
  - [✓] Docker Tooling
  - [✓] Eclipse JSON Editor and Tools
  - [✓] Eclipse MicroProfile CDI Tools
  - [✓] Hibernate Tools
  - [ ] JavaScript Development Tools
  - [✓] JavaScript Development Tools Chromium/V8 Remote Debugger
  - [✓] JBoss Archives Tools
  - [✓] JBoss AS, WildFly & EAP Server Tools
  - [✓] JBoss Central Community
  - [✓] JBoss Maven CDI Configurator
  - [✓] JBoss Maven Endorsed Libraries Configurator
  - [✓] JBoss Maven Hibernate Configurator
  - [✓] JBoss Maven Integration
  - [✓] JBoss Maven Project Examples
  - [✓] JBoss Maven Spring Boot Configurator
  - [✓] JBoss OpenShift JavaScript Tools
  - [✓] JBoss OpenShift Tools

- If during the installation we are asked for confirmation to continue, we accept and continue.
- Once the installation process is finished, we will be asked to restart Eclipse.
- To check that it has been installed correctly, click on the menu **Window** → **Show View** → **Other**, the Hibernate options should appear. Or in **Window** → **Perspective** → **Open Perspective** → **Other**.



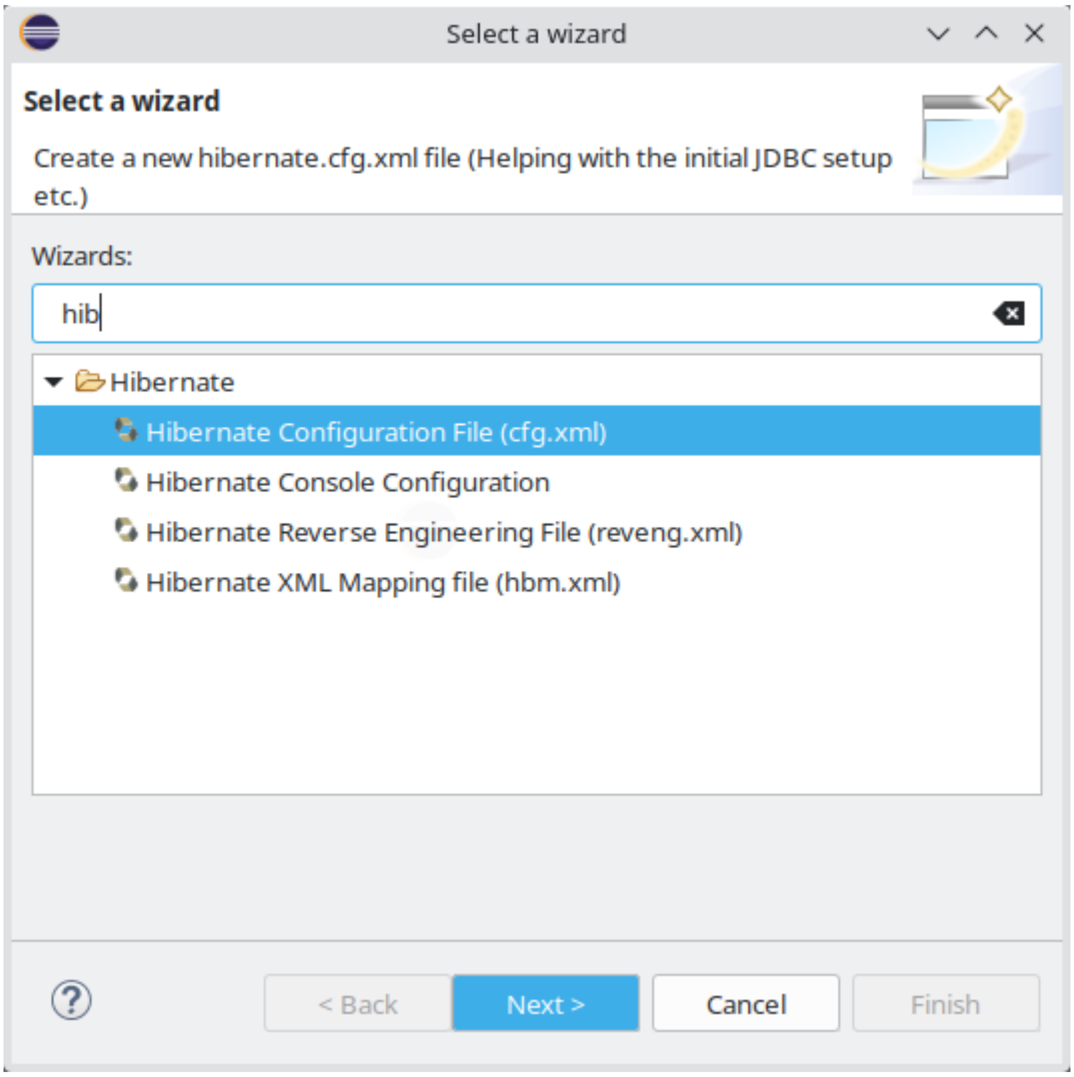

Open Perspective
CSS Editing
Debug
Docker Tooling
Git
Hibernate
Java (default)
Java Browsing
Java EE
Java Monitor
Java Type Hierarchy
JBoss
JMX
Cancel
Open

5.3. XML configuration files

Generic config file (hibernate.cfg.xml)

Once we have the MySQL library in our project we can create the Hibernate configuration file, called hibernate.cfg.xml.

- To do this, right click on the project and click **New** → **Other** → **Hibernate** → **Hibernate Configuration File (cfg.xml)**.



Select a wizard

Select a wizard

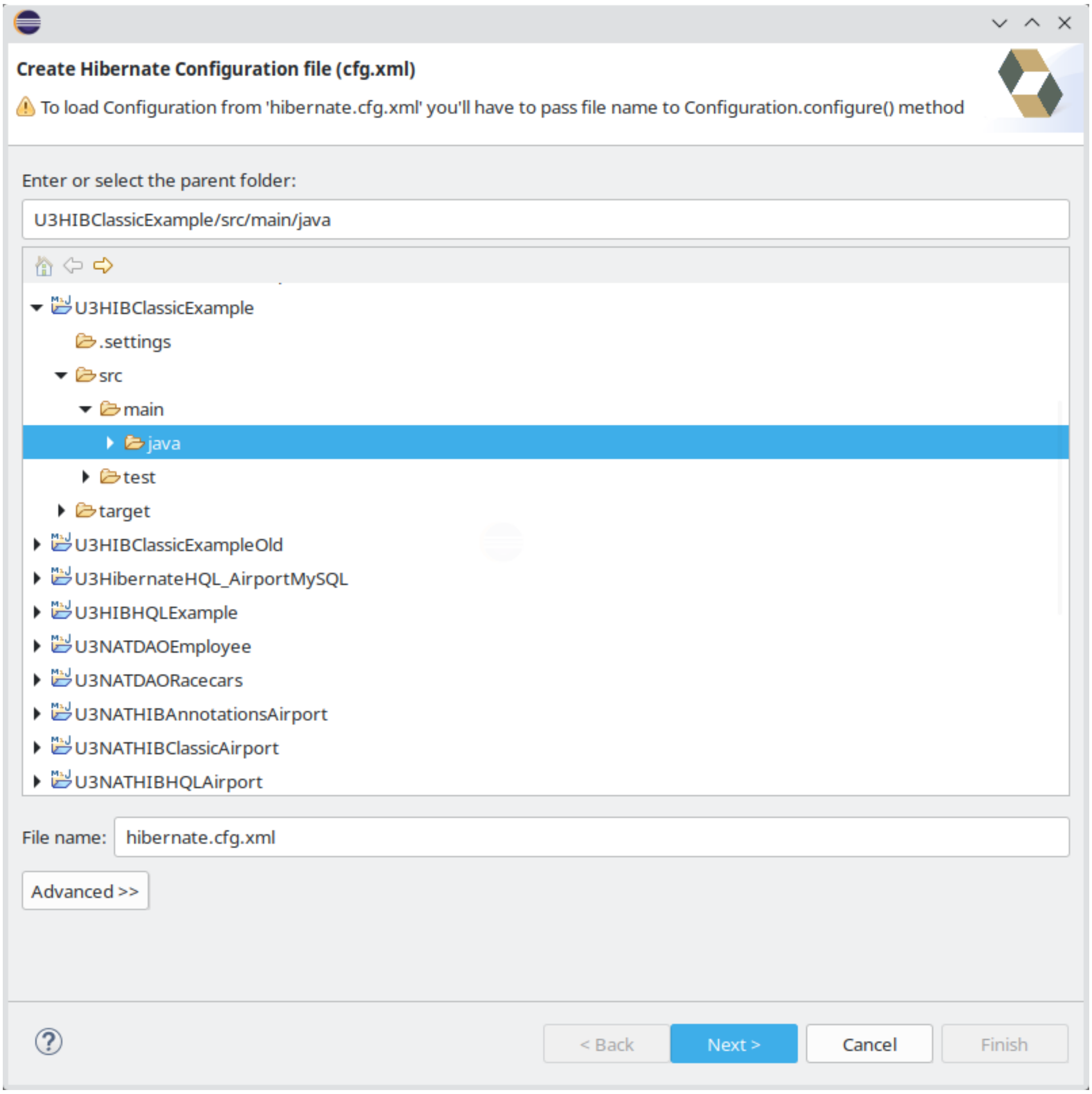
Create a new hibernate.cfg.xml file (Helping with the initial JDBC setup etc.)

Wizards:

hib

- Hibernate
  - Hibernate Configuration File (cfg.xml)
  - Hibernate Console Configuration
  - Hibernate Reverse Engineering File (reveng.xml)
  - Hibernate XML Mapping file (hbm.xml)

- Click on the Next button and indicate where to create the file. In our example we are going to create it inside the **src/resources** folder.



Create Hibernate Configuration file (cfg.xml)

To load Configuration from 'hibernate.cfg.xml' you'll have to pass file name to Configuration.configure() method

Enter or select the parent folder:

U3HIBClassicExample/src/main/java

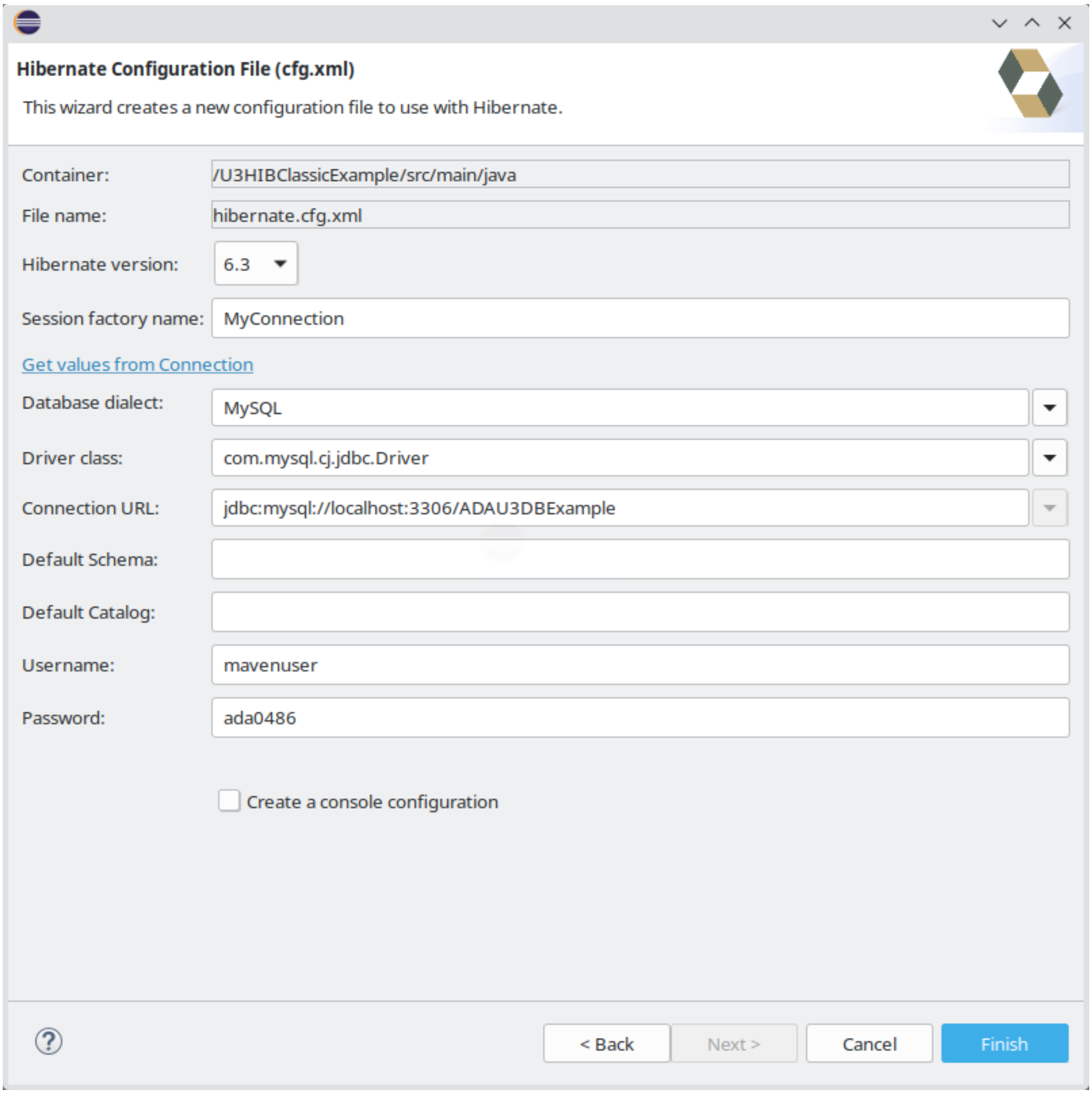
- U3HIBClassicExample
  - .settings
  - src
    - main
      - java
    - test
  - target
- U3HIBClassicExampleOld
- U3HibernateHQL_AirportMySQL
- U3HIBHQLExample
- U3NATDAOEmployee
- U3NATDAORacecars
- U3NATHIBAnnotationsAirport
- U3NATHIBClassicAirport
- U3NATHIBHQLAirport

File name: hibernate.cfg.xml

- The next step is to configure the **connection to the database**:
  - Hibernate Version → 6.3
  - Session Factory Name → Name of our database connection.
  - Database Dialect → Select how JDBC will connect to the DB, select MySQL.
  - Driver Class → com.mysql.cj.jdbc.Driver
  - Connection URL → URL of connection to the DB.
  - Username → User name

  - Password → Password

- Finally click on the Finish button.

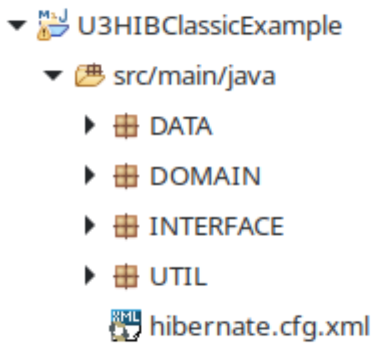


Hibernate Configuration File (cfg.xml)

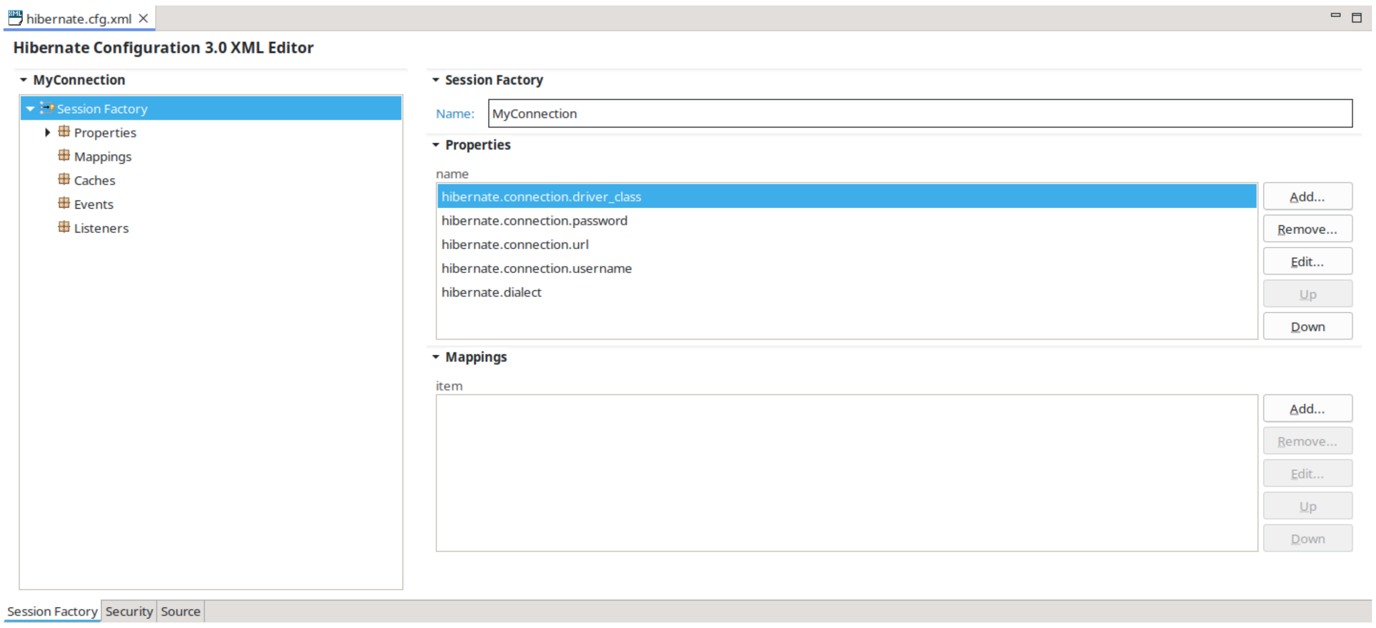
This wizard creates a new configuration file to use with Hibernate.

Container:	/U3HIBClassicExample/src/main/java
File name:	hibernate.cfg.xml
Hibernate version:	6.3
Session factory name:	MyConnection
Database dialect:	MySQL
Driver class:	com.mysql.cj.jdbc.Driver
Connection URL:	jdbc:mysql://localhost:3306/ADAU3DBExample
Default Schema:	
Default Catalog:	
Username:	mavenuser
Password:	ada0486

- A new file has been created.



- At this point you should be able to see the Hibernate configuration editor.


hibernate.cfg.xml

Hibernate Configuration 3.0 XML Editor

MyConnection
- Session Factory
  - Properties
  - Mappings
  - Caches
  - Events
  - Listeners

Session Factory

Name: MyConnection

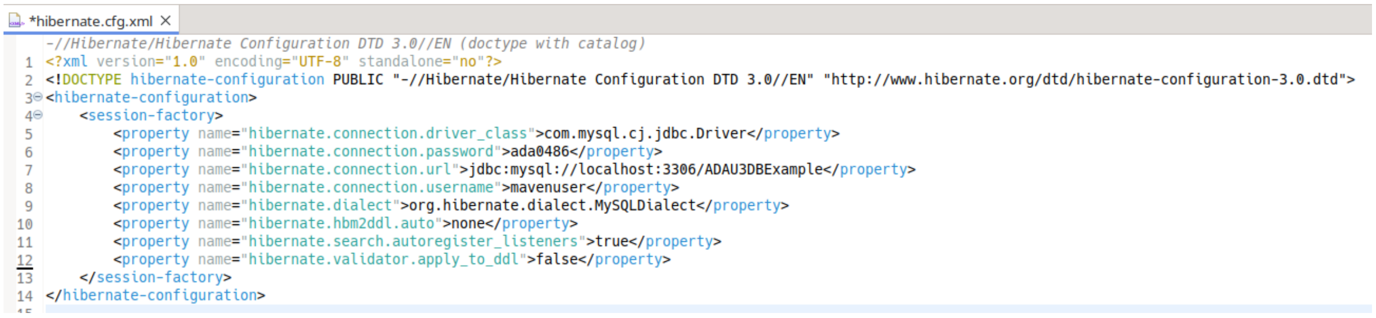
Properties

name
hibernate.connection.driver_class
hibernate.connection.password
hibernate.connection.url
hibernate.connection.username
hibernate.dialect

Mappings

item

- And if you view it with a generic editor, you will see the XML code:

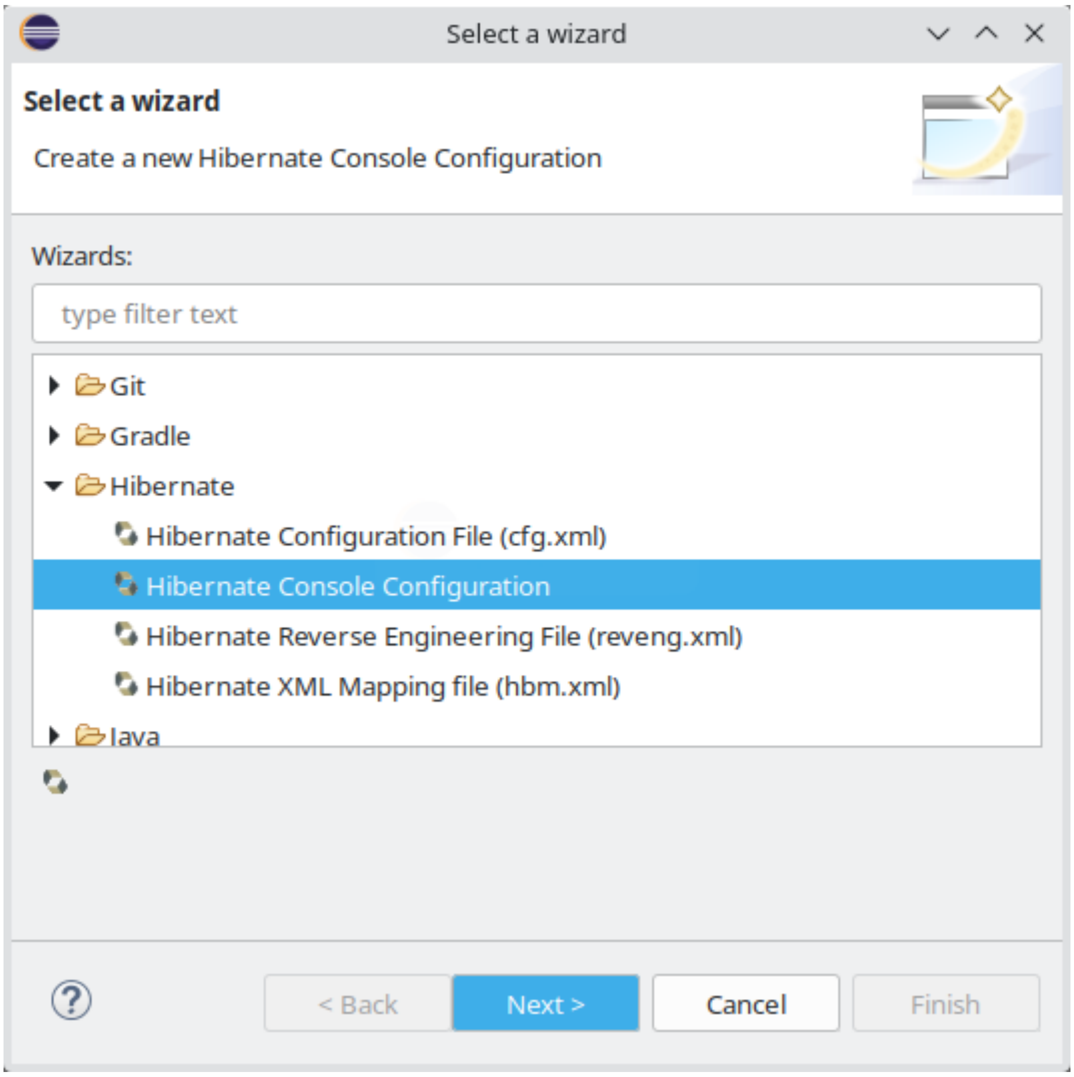

*hibernate.cfg.xml

```
-//Hibernate/Hibernate Configuration DTD 3.0//EN (doctype with catalog)
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.password">ada0486</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/ADAU3DBExample</property>
        <property name="hibernate.connection.username">mavenuser</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.hbm2ddl.auto">none</property>
        <property name="hibernate.search.autoregister_listeners">true</property>
        <property name="hibernate.validator.apply_to_ddl">false</property>
    </session-factory>
</hibernate-configuration>
```

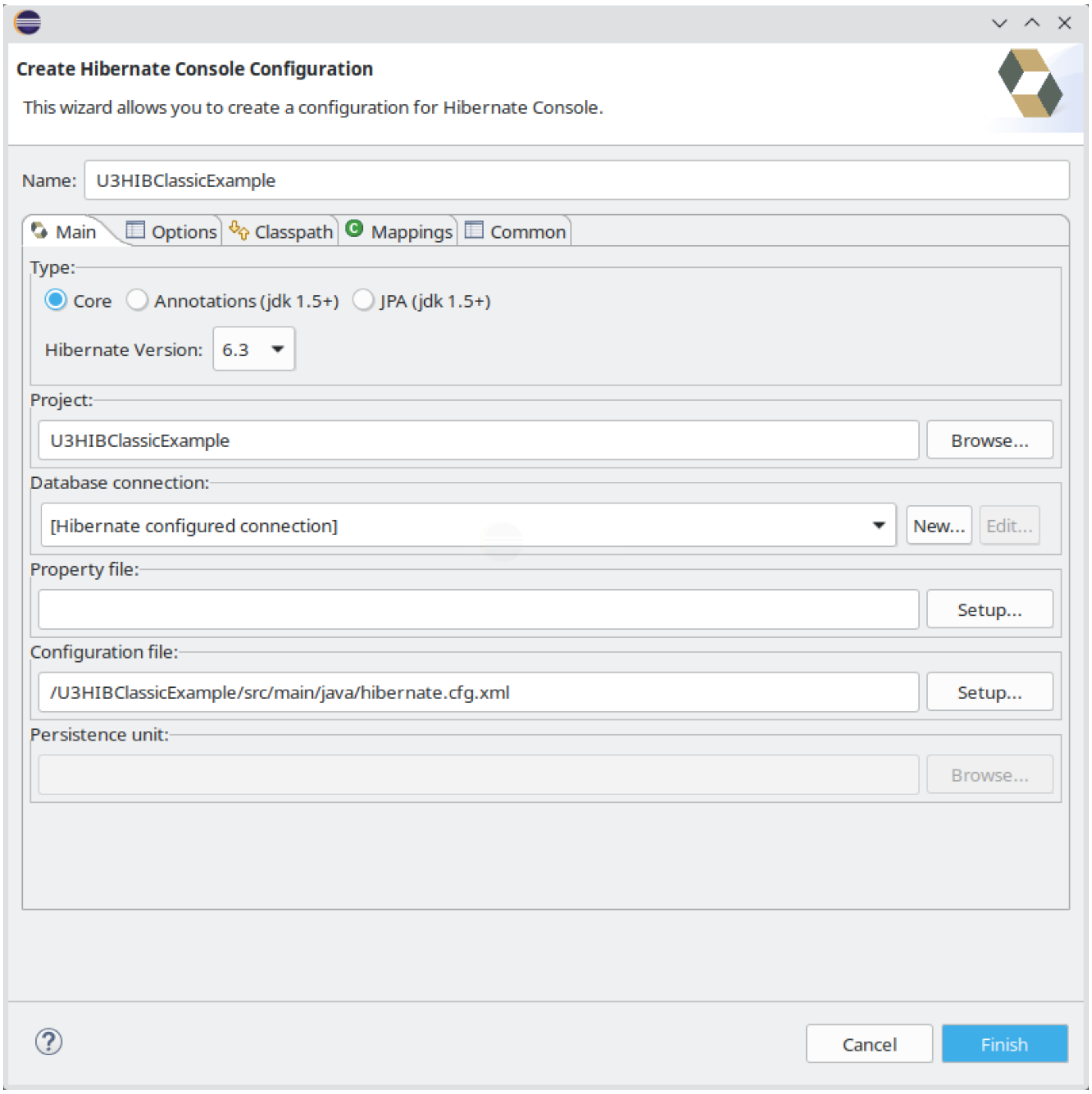
XML file Hibernate Configuration

Once the configuration file hibernate.cfg.xml has been created, we have to create the **XML file Hibernate Configuration**.

- Right click on our project and select **New** → **Other** → **Hibernate Console Configuration**.


Select a wizard
Select a wizard
Create a new Hibernate Console Configuration
Wizards:
type filter text
Git
Gradle
Hibernate
Hibernate Configuration File (cfg.xml)
Hibernate Console Configuration
Hibernate Reverse Engineering File (reveng.xml)
Hibernate XML Mapping file (hbm.xml)
Java
< Back
Next >
Cancel
Finish

- A new window appears, where we have to indicate a Name for our configuration. We make sure that in the Project field our project appears and in the Configuration File field the previously created configuration file appears.



Create Hibernate Console Configuration

This wizard allows you to create a configuration for Hibernate Console.

Name: U3HIBClassicExample

Type: [✓ Core] Annotations (jdk 1.5+) JPA (jdk 1.5+)

Hibernate Version: 6.3

Project: U3HIBClassicExample

Database connection: [Hibernate configured connection]

Property file:

Configuration file: /U3HIBClassicExample/src/main/java/hibernate.cfg.xml

Persistence unit:

- Click Finish to finish the creation of the Console Configuration.

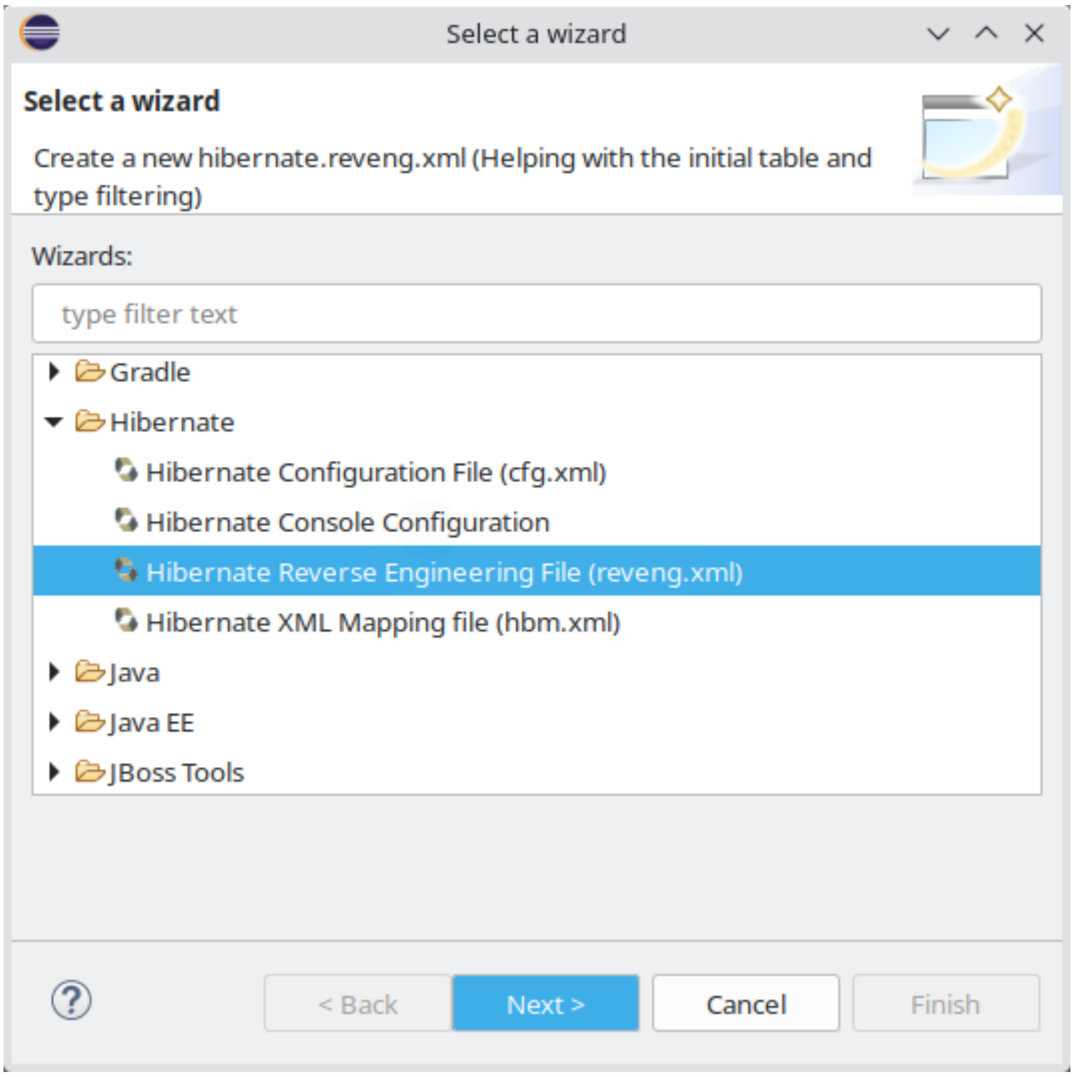
XML Hibernate Reverse Engineering

Finally we are going to create the **XML file Hibernate Reverse Engineering (reveng.xml)** which is in charge of creating the classes of our MySQL tables.

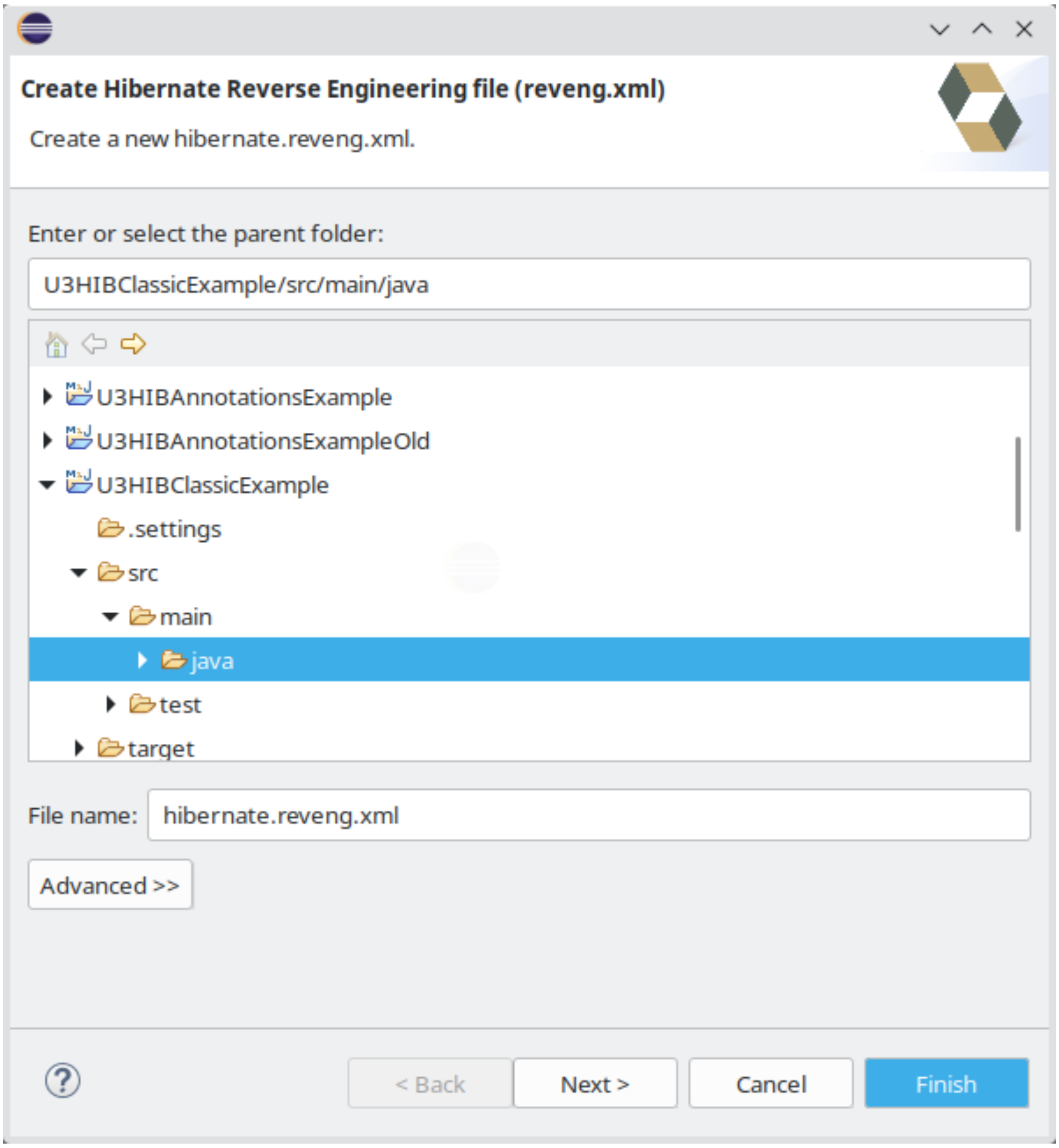
> Make sure you have created the former database and have MySQL open.

- Right click on the project and **New** → **Other** → **Hibernate** → **Hibernate Reverse**

Engineering File (reveng.xml).


Select a wizard
Select a wizard
Create a new hibernate.reveng.xml (Helping with the initial table and type filtering)
Wizards:
type filter text
Gradle
Hibernate
Hibernate Configuration File (cfg.xml)
Hibernate Console Configuration
Hibernate Reverse Engineering File (reveng.xml)
Hibernate XML Mapping file (hbm.xml)
Java
Java EE
JBoss Tools
< Back
Next >
Cancel
Finish

- Click the Next button and indicate that it will be saved in the same folder where we have the configuration file.

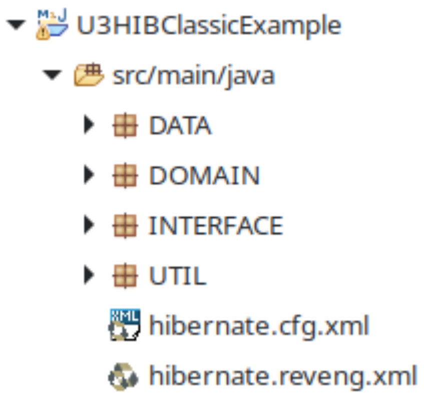

Create Hibernate Reverse Engineering file (reveng.xml)

Create a new hibernate.reveng.xml.

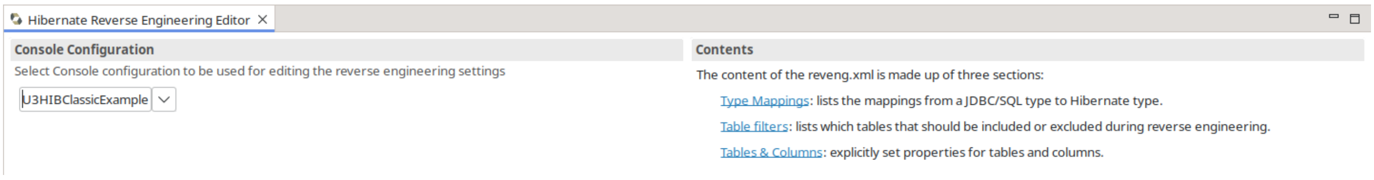
Enter or select the parent folder: U3HIBClassicExample/src/main/java

File name: hibernate.reveng.xml

- A new file has been created.



- A window is displayed in which we must indicate the tables we want to map.



Hibernate Reverse Engineering Editor

Console Configuration

Select Console configuration to be used for editing the reverse engineering settings

U3HIBClassicExample

Contents

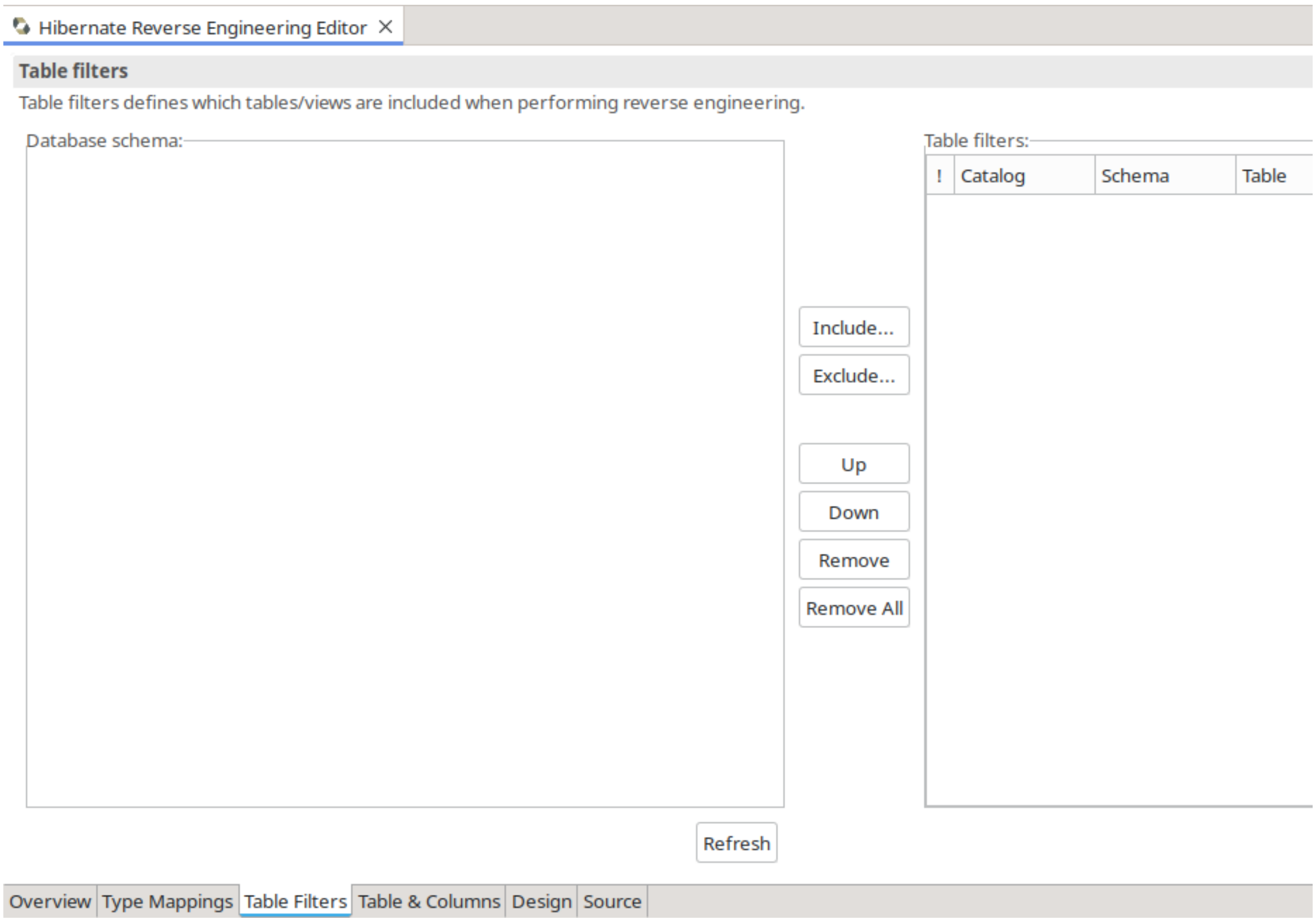
The content of the reveng.xml is made up of three sections:

Type Mappings: lists the mappings from a JDBC/SQL type to Hibernate type.

Table filters: lists which tables that should be included or excluded during reverse engineering.

Tables & Columns: explicitly set properties for tables and columns.

- First select the Console Configuration from the combo box at the top and click on the tab "**Table Filters**".



Hibernate Reverse Engineering Editor

Table filters

Table filters defines which tables/views are included when performing reverse engineering.

- Click on **Refresh** button to display the tables of our database. Select the tables and click on the "**Include**" button.

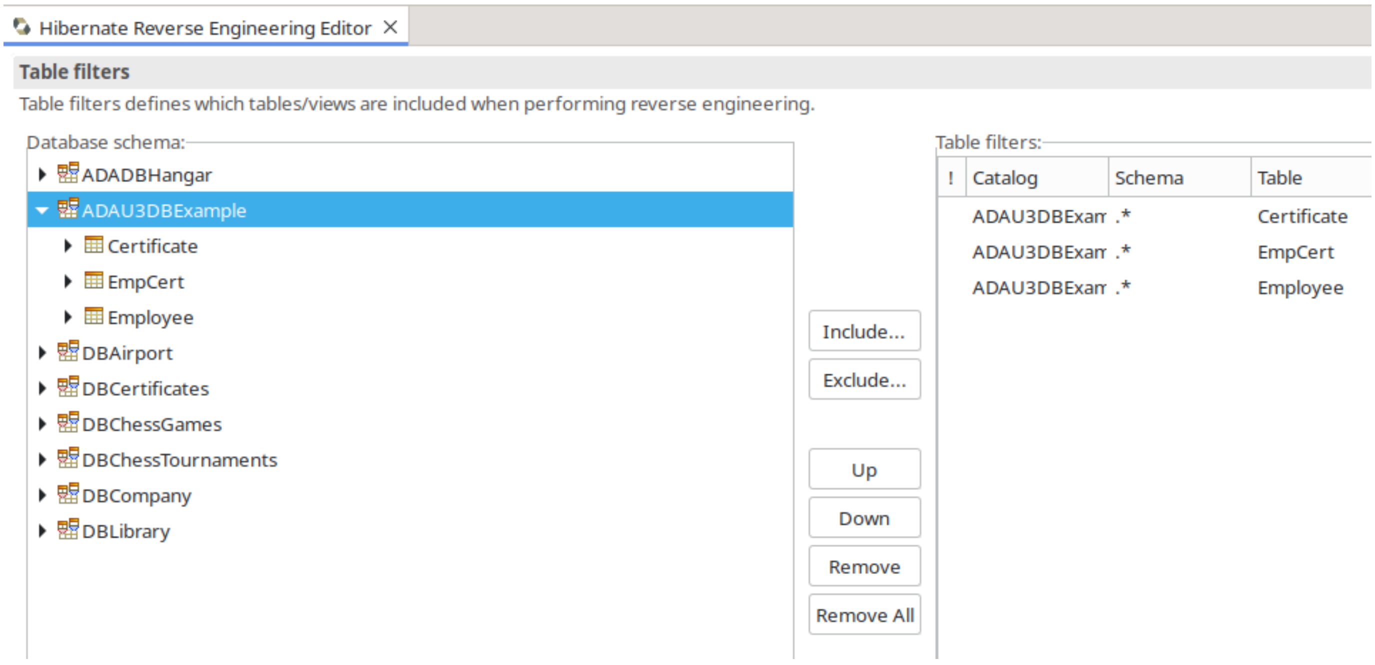
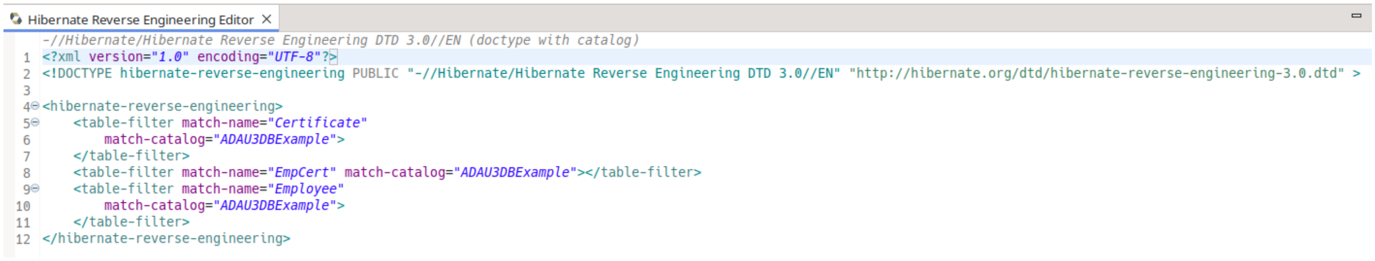


Table filters defines which tables/views are included when performing reverse engineering.

!	Catalog	Schema	Table
	ADAU3DBExam	.*	Certificate
	ADAU3DBExam	.*	EmpCert
	ADAU3DBExam	.*	Employee

- Click on tab "**Source**". If everything went well we should be able to see the Hibernate Reverse Engineering editor.



```
-//Hibernate/Hibernate Reverse Engineering DTD 3.0//EN (doctype with catalog)
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse Engineering DTD 3.0//EN" "http://hibernate.org/dtd/hibernate-reverse-engineering-3.0.dtd" >

<hibernate-reverse-engineering>
    <table-filter match-name="Certificate"
        match-catalog="ADAU3DBExample">
    </table-filter>
    <table-filter match-name="EmpCert" match-catalog="ADAU3DBExample"></table-filter>
    <table-filter match-name="Employee"
        match-catalog="ADAU3DBExample">
    </table-filter>
</hibernate-reverse-engineering>
```

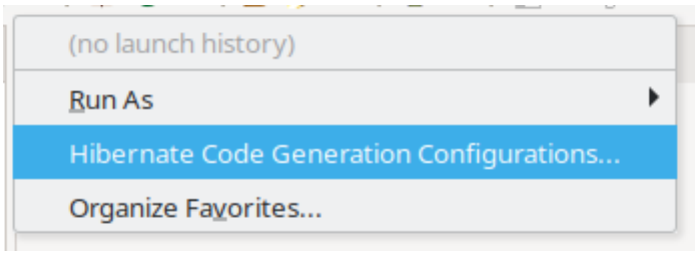
- Save the changes (**Ctrl+S** or **File** → **Save**).

Once this last step is completed we should have finished configuring Hibernate for our project.

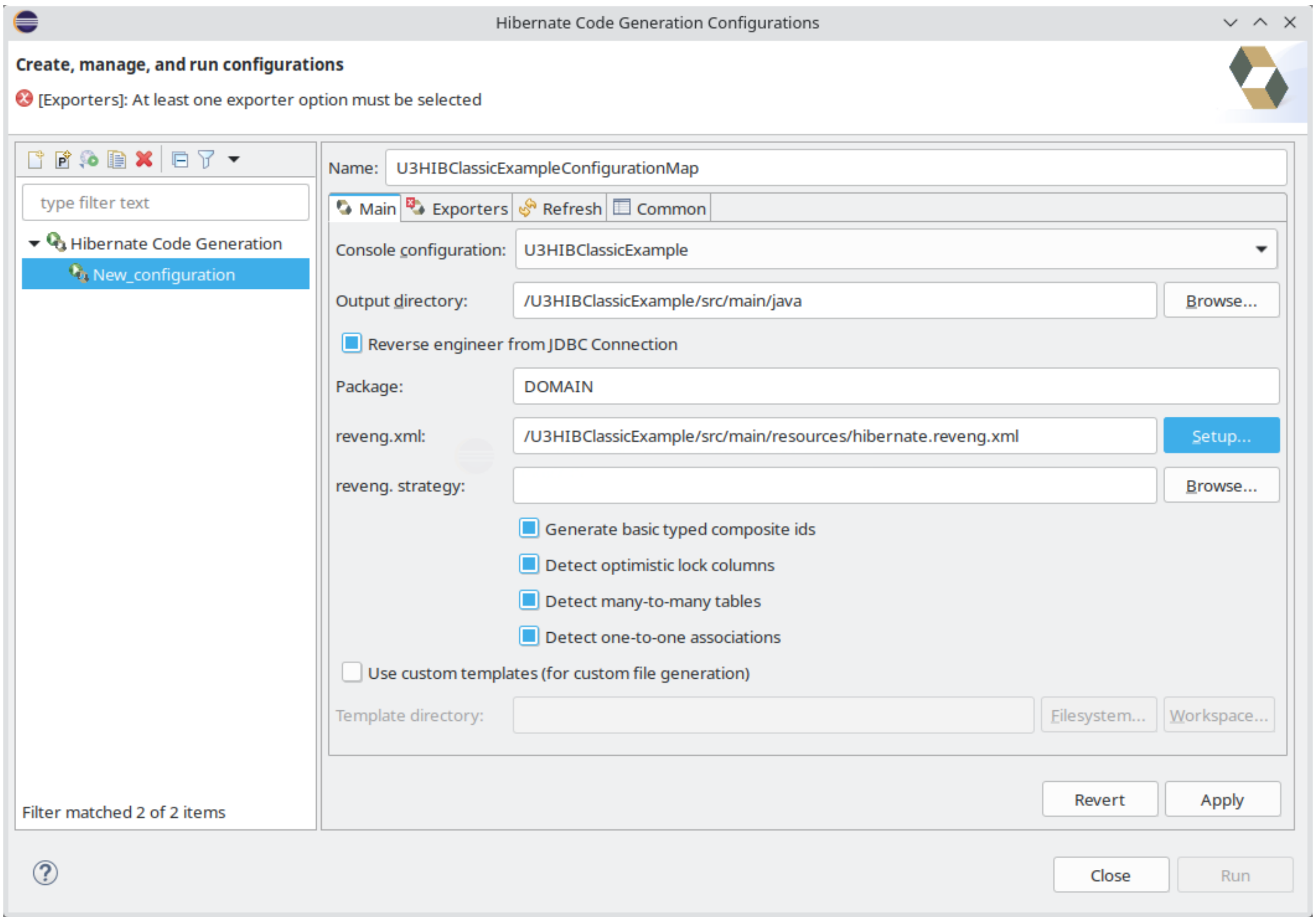
5.5. Generate database classes (POJO)

The next step is to generate the classes of our database.

- To do this, click on the arrow to the right of the "**Run As**" button and click on "**Hibernate Code Generation Configurations**". Or menu **Run** → **Hibernate Code Generation** → **Hibernate Code Generation Configurations**.



- From the window that appears we select (double click) the option "Hibernate Code Generation" and configure:
  - Name → **U3HIBClassicExampleConfigurationMap**
  - Console Configuration → Select the one we had previously created.
  - Output Directory → **src/main/java** directory.
  - Package → The package where our classes will be created, for example "DOMAIN".
  - reveng.xml → Select the reveng.xml file created earlier.


Hibernate Code Generation Configurations

Create, manage, and run configurations

[Exporters]: At least one exporter option must be selected

Name: U3HIBClassicExampleConfigurationMap

Console configuration: U3HIBClassicExample

Output directory: /U3HIBClassicExample/src/main/java

[✓] Reverse engineer from JDBC Connection

Package: DOMAIN

reveng.xml: /U3HIBClassicExample/src/main/resources/hibernate.reveng.xml

reveng. strategy:

[✓] Generate basic typed composite ids

[✓] Detect optimistic lock columns

[✓] Detect many-to-many tables

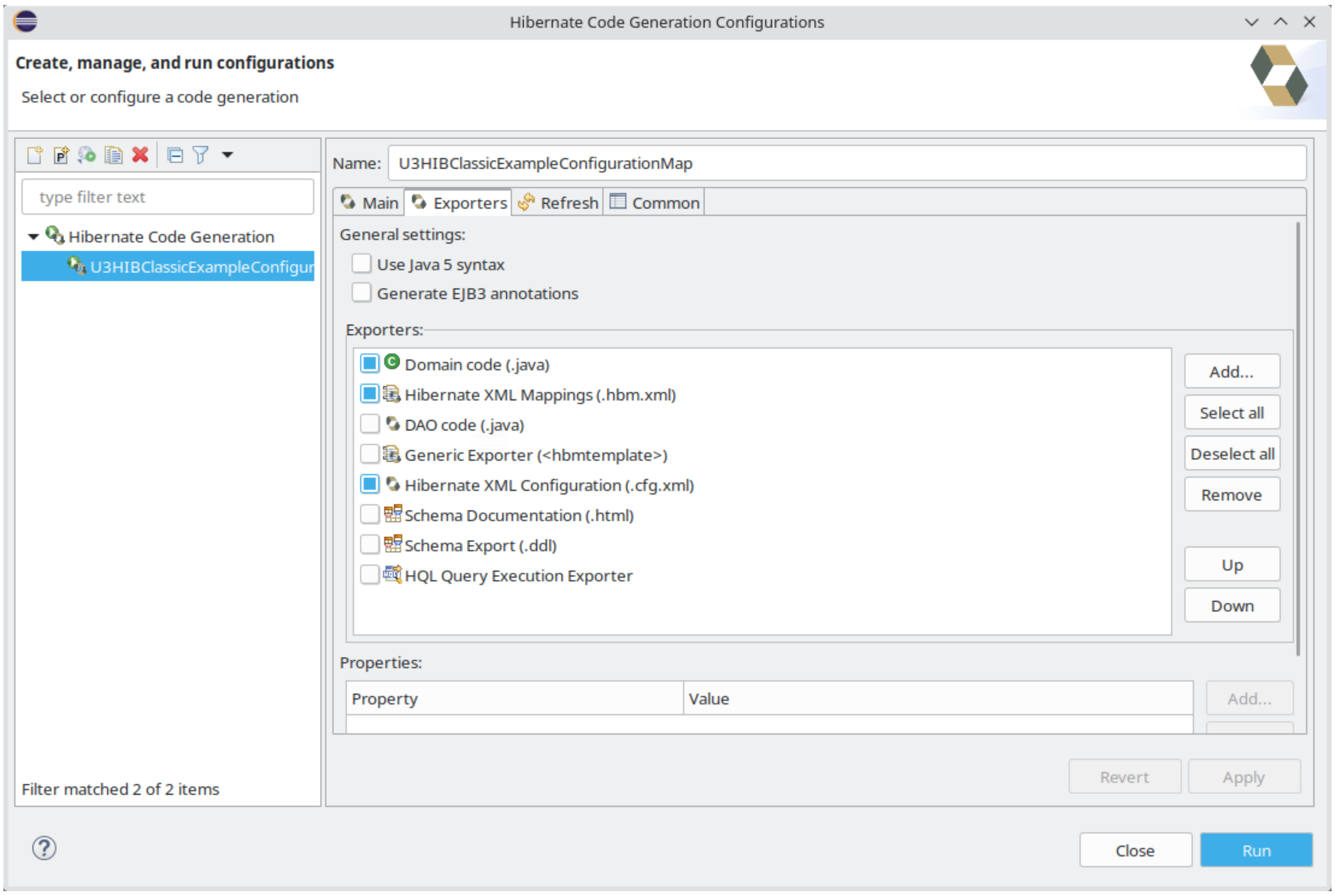
[✓] Detect one-to-one associations

[ ] Use custom templates (for custom file generation)

Template directory:

Filter matched 2 of 2 items

- From the "Exporters" tab, indicate the files you want to generate, tick the boxes:
  - Domain Code
  - Hibernate XML Mappings
  - Hibernate XML Configuration


Hibernate Code Generation Configurations
Create, manage, and run configurations
Select or configure a code generation
type filter text
Hibernate Code Generation
U3HIBClassicExampleConfigur
Filter matched 2 of 2 items
Main
Exporters
Refresh
Common
Add...
Select all
Deselect all
Remove
Up
Down
Add...
Revert
Apply
Close
Run

Name: U3HIBClassicExampleConfigurationMap

General settings:

- [ ] Use Java 5 syntax
- [ ] Generate EJB3 annotations

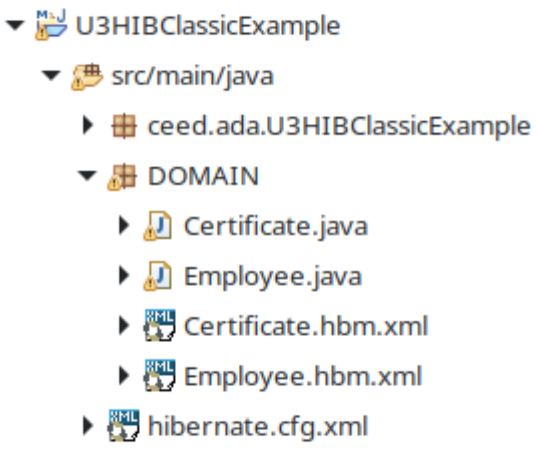
Exporters:

- [x] Domain code (.java)
- [x] Hibernate XML Mappings (.hbm.xml)
- [ ] DAO code (.java)
- [ ] Generic Exporter (<hbmtemplate>)
- [x] Hibernate XML Configuration (.cfg.xml)
- [ ] Schema Documentation (.html)
- [ ] Schema Export (.ddl)
- [ ] HQL Query Execution Exporter

Properties:

Property	Value

- Once selected, click on the Apply button and finally on "Run".
- If everything went well, after a few seconds you should be able to see the DOMAIN package with the model classes generated by Hibernate inside.



```
U3HIBClassicExample
  src/main/java
    ceed.ada.U3HIBClassicExample
    DOMAIN
      Certificate.java
      Employee.java
      Certificate.hbm.xml
      Employee.hbm.xml
    hibernate.cfg.xml
```

For each class, a series of attributes are generated and included. These attributes represent:

- The columns of the table they map and their foreign key relationships
- The constructors and the corresponding getter and setter methods

Class Certificate

In this class we can see:

```
private int certId;
private String certname;
private Set employees = new HashSet(0);
```

The attributes **certId** and **certname** correspond to the columns of the table. However, the attribute **employees** defines the foreign key relationship between employees and certificates. This attribute will allow us to store the employees of a specific certificate.

Class Employee

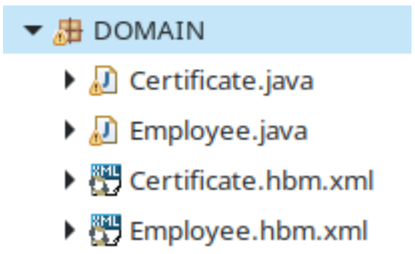
In this class we can see:

```
private int empId;
private String firstname;
private String lastname;
private Double salary;
private Set certificates = new HashSet(0);
```

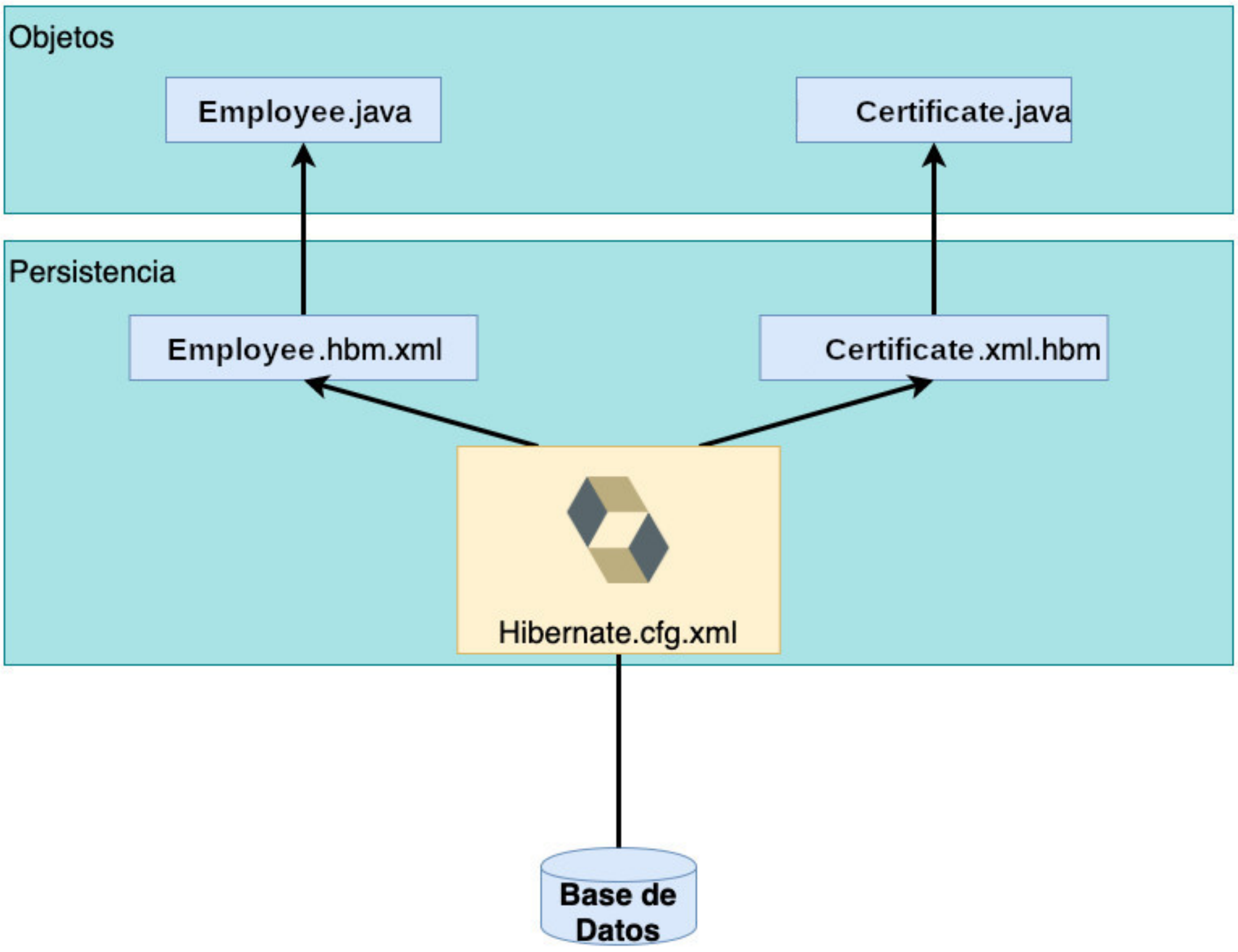
The attributes **empId**, **firstname**, **lastname** and **salary** correspond to the columns of the table. However, the attribute **certificates** defines the foreign key relationship between certificates and employees. This attribute will allow us to store the certificates of a specific employee.

6. Hibernate. Structure of the mapping files

Hibernate uses mapping files to relate the DB tables to the Java classes. These files are in XML format and have the extension .hbm.xml. In our example, the files **Certificate.hbm.xml** and **Employee.hbm.xml** have been generated. These files are stored in the same directory as the classes Employees.java and Certificates.java and represent employee and certificate objects respectively.



The following figure shows the correspondence between the mapping files and the generated classes:



The structure of a typical cfg.xml file is as follows:

- **hibernate-mapping**. All mapping files begin and end with this tag.
- **class**. Encloses the class with its attributes indicating the class name, the table the mapping comes from and the DB.

Certificate.hbm.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN
<hibernate-mapping>
    <class catalog="ADAU3DBExample" name="DOMAIN.Certificate" optimistic-lock=
        <id name="certId" type="int">
            <column name="certID"/>
            <generator class="assigned"/>
        </id>
        <property name="certname" type="string">
            <column length="30" name="certname"/>
        </property>
        <set fetch="select" inverse="false" lazy="true" name="employees" table
            <key>
                <column name="certificateID" not-null="true"/>
            </key>
            <many-to-many entity-name="DOMAIN.Employee">
                <column name="employeeID" not-null="true"/>
            </many-to-many>
        </set>
    </class>
</hibernate-mapping>
```

Employee.hbm.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN
```

```
<hibernate-mapping>
    <class catalog="ADAU3DBExample" name="DOMAIN.Employee" optimistic-lock="nc
        <id name="empId" type="int">
            <column name="empID"/>
            <generator class="assigned"/>
        </id>
        <property name="firstname" type="string">
            <column length="20" name="firstname"/>
        </property>
        <property name="lastname" type="string">
            <column length="20" name="lastname"/>
        </property>
        <property name="salary" type="java.lang.Double">
            <column name="salary" precision="22" scale="0"/>
        </property>
        <set fetch="select" inverse="false" lazy="true" name="certificates" ta
            <key>
                <column name="employeeID" not-null="true"/>
            </key>
            <many-to-many entity-name="DOMAIN.Certificate">
                <column name="certificateID" not-null="true"/>
            </many-to-many>
        </set>
    </class>
</hibernate-mapping>
```

hibernate.cfg.xml

In the main Hibernate configuration file we can see that 2 lines corresponding to the 2 new classes generated have been added.

```
<mapping resource="DOMAIN/Employee.hbm.xml"/>
<mapping resource="DOMAIN/Certificate.hbm.xml"/>
```

6.1. Tag "id"

Refers to the primary key of the table.

- The **name** attribute refers to the name of the java class attribute.
- The **column** element indicates the name of the column in the database table.
- The **generator** element indicates the nature of the key field:
  - *Assigned* → Value must be given by the user.
  - *Increment* → Identifier auto-generated by the database.
  - *Native* → Native generation. If the database supports an identity generator, then it acts as identity otherwise it will check for the database support of other generators.

In our case we have an **auto-incremental primary key** in the Employees and Certificates tables, so we must modify the "**generator class**" parameter of the column in both files and set the value to "**native**".

Certificate.hbm.xml

```
<id name="certId" type="int">
      <column name="certID"/>
      <generator class="native"/>
</id>
```

Employee.hbm.xml

```
<id name="empId" type="int">
      <column name="empID"/>
      <generator class="native"/>
</id>
```

6.2. Tag "property"

Relates each non-key field in the table to an attribute of the Java class. Indicates, like "id":

- Attribute name
- Name of the field in the table
- Type of the field in the table
- Length of the field (Optional)

```
<property name="certname" type="string">
       <column length="30" name="certname"/>
</property>
```

The types we declare and use in the mapping files are not Java or SQL types, they are **Hibernate mapping types**, converters that can translate Java data types to SQL and vice versa. Although Hibernate generally tries to identify the optimal mapping type on its own, we may occasionally need to make manual modifications to these files in order to improve their accuracy and performance.

6.3. Tag "set"

The set element is used to map collections. Several attributes are defined within set:

- **name** → Indicates the name of the generated attribute.
- **table** → Table from which the information will be taken to generate the collection.
- **lazy**→ Indicates the way in which the list will be loaded from the database.
  - True → The list starts loading when the parent element has finished loading.
  - False → The list and the parent element are loaded simultaneously.
- The **key** element defines the name of the column defined as foreign key for loading the list.
- The **many-to-many** element defines the relationship, in this case many-to-many, indicating to which persistent class the objects in the list belong.

```
<set fetch="select" inverse="false" lazy="true" name="employees" table="EmpCer
        <key>
                <column name="certificateID" not-null="true"/>
        </key>
        <many-to-many entity-name="DOMAIN.Employee">
                <column name="employeeID" not-null="true"/>
        </many-to-many>
</set>
```

Relationship mapping

Of the employee and certificate mapping files, the **many-to-many** element is noteworthy:

- It refers to a bidirectional many-to-many association (one certificate can be associated to many employees and vice versa).
- It is used to define a many-to-many relationship between two Java classes.
  - The name attribute defines the name of the Java class attribute.
  - class-name → Defines the class from which to create the object.

- The fetch attribute allows you to choose between retrieval by outer-join or retrieval by sequential select. By default select.
- The column-name element refers to the name of the column within our database table.

We also have the **many-to-one** element, which allows one-way many-to-one associations to be referenced.

7. Persistent classes

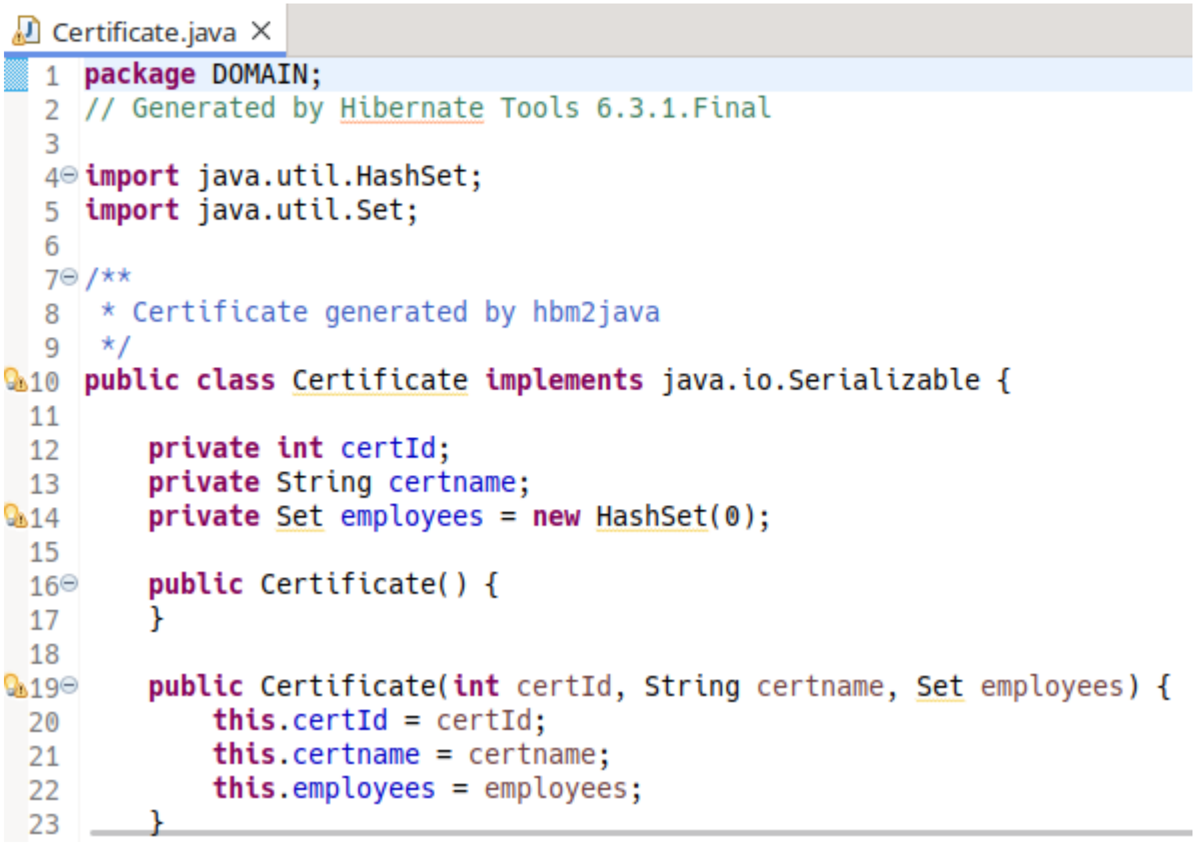
We have seen that the <hibernate-mapping></hibernate-mapping> elements in the xml mapping files include a class element that refers to a Java class.

In our project we have generated the classes **Employee.java** and **Certificate.java**, which are called **persistent classes**. These classes implement the entities of the problem and must implement the *Serializable* interface. Its main characteristics are:

- They are equivalent to a table in the DB, and a record or row is a persistent object of that class.
- They have only the attributes of the class and the corresponding getter and setter methods.
- They use standard JavaBean naming conventions for methods and properties.
- These rules are also called the POJO programming model.

7.1. Review persistent classes

In the automatically generated classes, we can see that yellow triangle warnings are displayed in the corresponding code. Although the installed Jboss Tools plugin saves us a lot of work, we must review and polish small details afterwards to have a perfectly working code.



Certificate.java

```java
package DOMAIN;
// Generated by Hibernate Tools 6.3.1.Final

import java.util.HashSet;
import java.util.Set;

/**
 * Certificate generated by hbm2java
 */
public class Certificate implements java.io.Serializable {

    private int certId;
    private String certname;
    private Set employees = new HashSet(0);

    public Certificate() {
    }

    public Certificate(int certId, String certname, Set employees) {
        this.certId = certId;
        this.certname = certname;
        this.employees = employees;
    }
```

Now we must do the following things:

- Add the version code. Just click on the yellow triangle and add default serial version ID.
- Specify the type of HashSet and add it.
- Modify the corresponding methods, taking into account that the primary key of both tables (Employee and Certificate) is auto-incremental. Therefore, we do not need to specify the employee or certificate code when instantiating the object.
- Add the necessary comments to the code to make it easier to understand.

The revised and corrected classes are shown below.

The automatic generation of the Hibernate mapping does not follow the Hungarian notation. It takes into account the name of the various tables and columns but without adding the typical prefixes. Modifying everything would be as expensive as mapping manually, so we will leave it as it is.

Since the Hungarian notation is optional, in the rest of the classes and code of our program we can apply it if we choose to do so.

7.1.1. Class Certificate

The structure of the Certificate.java class would look like this:

```
package DOMAIN;
// Generated by Hibernate Tools 6.3.1.Final

import java.util.HashSet;
import java.util.Set;

/**
 * ======================================================================
 * Object Certificate generated by hbm2java
 * Modified and adapted
 * @author Abelardo Martínez
 * ======================================================================
 */

public class Certificate implements java.io.Serializable {

    /*
     * ------------------------------------------
     * ATTRIBUTES
     * ------------------------------------------
     */
    private static final long serialVersionUID = 1L;
    private int certId;
    private String certname;
    private Set<Employee> employees = new HashSet<Employee>(0);

    /*
     * ------------------------------------------
     * METHODS
     * ------------------------------------------
     */
    /*
     * Empty constructor
     */
    public Certificate() {
    }

    /*
     * Constructor without ID. All fields, except primary key
     */
    public Certificate(String certname, Set<Employee> employees) {
        this.certname = certname;
        this.employees = employees;
    }

    /*
     * ------------------------------------------
     * GETTERS & SETTERS
     * ------------------------------------------
     */
    public int getCertId() {
        return this.certId;
    }

    public void setCertId(int certId) {
        this.certId = certId;
    }

    public String getCertname() {
        return this.certname;
    }

    public void setCertname(String certname) {
        this.certname = certname;
    }
    public Set<Employee> getEmployees() {
        return this.employees;
    }

    public void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }

}
```

7.1.2. Class Employee

The structure of the Employee.java class would look like this:

```java
package DOMAIN;
// Generated by Hibernate Tools 6.3.1.Final

import java.util.HashSet;
import java.util.Set;

/**
 * ======================================================================
 * Object Employee generated by hbm2java
 * Modified and adapted
 * @author Abelardo Martínez
 * ======================================================================
 */

public class Employee implements java.io.Serializable {

    /*
     * ------------------------------------------
     * ATTRIBUTES
     * ------------------------------------------
     */
    private static final long serialVersionUID = 1L;
    private int empId;
    private String firstname;
    private String lastname;
    private Double salary;
    private Set<Certificate> certificates = new HashSet<Certificate>(0);

    /*
```

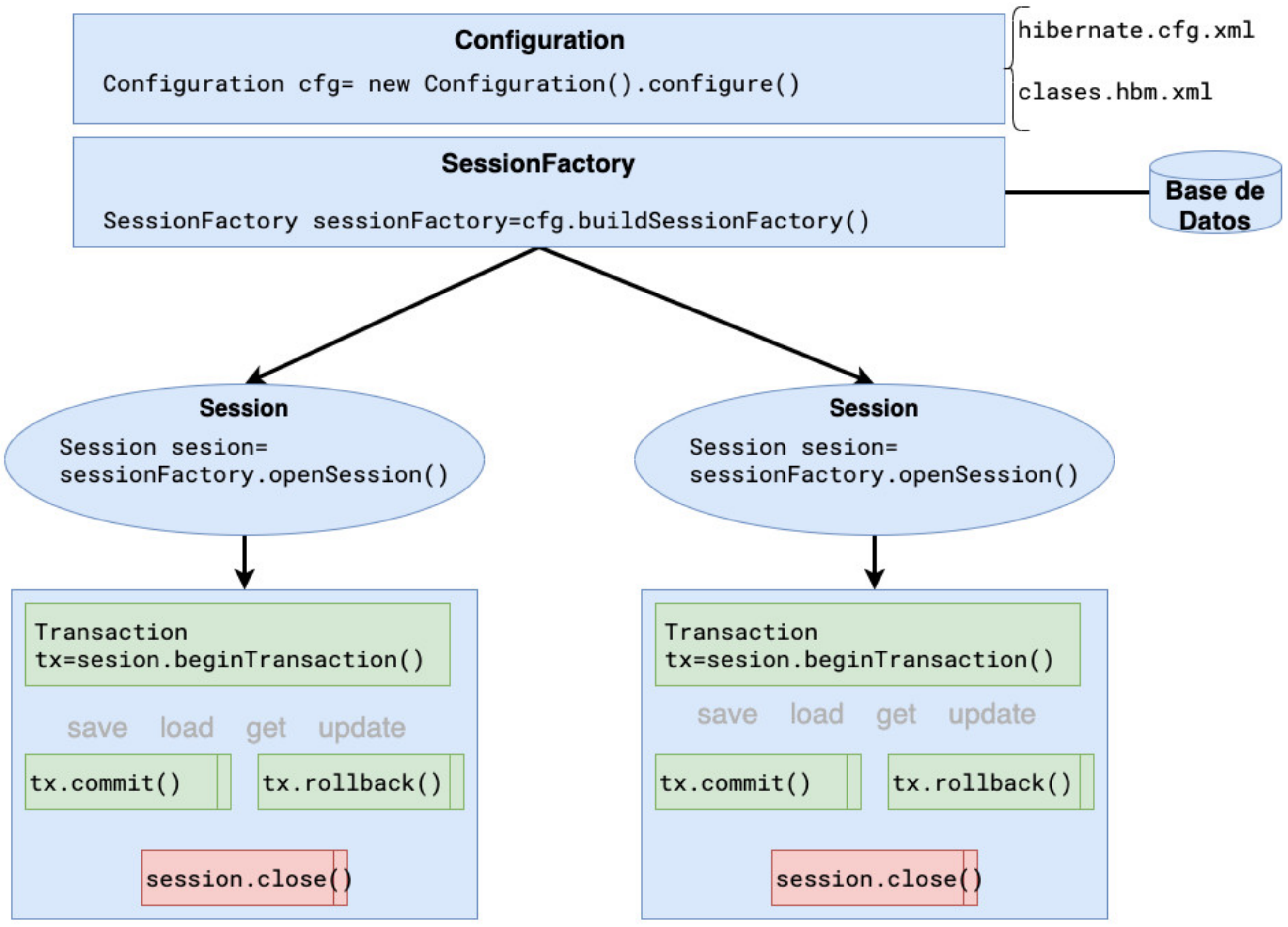
```
     * ------------------------------------------
     * METHODS
     * ------------------------------------------
     */
    /*
     * Empty constructor
     */
    public Employee() {
    }

    /*
     * Constructor without ID. All fields, except primary key
     */
    public Employee(String firstname, String lastname, Double salary, Set<Cert
        this.firstname = firstname;
        this.lastname = lastname;
        this.salary = salary;
        this.certificates = certificates;
    }

    /*
     * ------------------------------------------
     * GETTERS & SETTERS
     * ------------------------------------------
     */
    public int getEmpId() {
        return this.empId;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }

    public String getFirstname() {
        return this.firstname;
    }
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastname() {
        return this.lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    public Double getSalary() {
        return this.salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    public Set<Certificate> getCertificates() {
        return this.certificates;
    }

    public void setCertificates(Set<Certificate> certificates) {
        this.certificates = certificates;
    }

}
```

8. Hibernate. Sessions and objects

Once the whole environment has been configured and tested, it is time to create our first Java class that makes use of Hibernate. The first thing to do is to create a singleton class to obtain the **SessionFactory**. **Singleton** is a design pattern that implements mechanisms to ensure that we can only build a single object of a given class.

Our singleton object will be an auxiliary class that accesses the SessionFactory and obtains a Session object, there is only one SessionFactory for the whole application. In the class we will define a static object of type SessionFactory that we will build based on the Hibernate configuration. The only public method is getSessionFactory, so we guarantee that no different instances of this object are created.

With this class we can get the current sessionFactory from anywhere in our application. We'll be working with an object called FACTORY and several SESSIONS like this:



In order to be able to use Hibernate's persistence mechanisms, the Hibernate environment must be initialised and a **Session** object must be obtained using the **SessionFactory** class. The following code fragment illustrates the process:

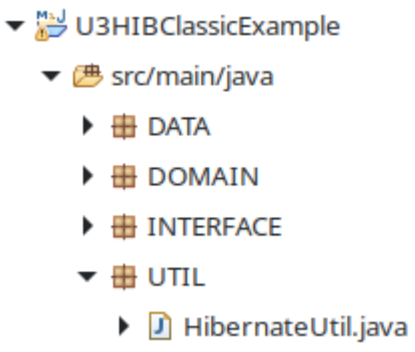
```
// CONFIGURATION
Configuration = cfg = new Configuration().configure();

// FACTORY
SessionFactory sessionFactory = cfg.buildSessionFactory();

// SESSIONS
Session session = sessionFactory.openSession();
session.beginTransaction();
Employee emp = new Employee();
emp.setEmpName("Pepe");
    …
session.persist(emp);
session.getTransaction().commit();
session.close();
```

where:

- The call to **Configuration().configure()** loads the configuration file **hibernate.cfg.xml** and initialises the Hibernate environment.
- The **SessionFactory** is normally only created once and is used to get all the **sessions** related to a given context.

8.1. Class HibernateUtil

In our example we create the HibernateUtil.java class using the singleton pattern for this purpose:


```
U3HIBClassicExample
    src/main/java
        DATA
        DOMAIN
        INTERFACE
        UTIL
            HibernateUtil.java
```

```
package UTIL;

import java.util.logging.Level;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

/**
 * ======================================================================
 * Class to manage Hibernate session
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ======================================================================
 */

public class HibernateUtil {

    /**
     * ----------------------------------------
     * GLOBAL CONSTANTS AND VARIABLES
     * ----------------------------------------
     */
    //Persistent session
    public static final SessionFactory SFACTORY = buildSessionFactory();

    /*
     * ----------------------
     * SESSION MANAGEMENT
     * ----------------------
     */
    /*
     * Create new hibernate session
     */
    private static SessionFactory buildSessionFactory() {
        java.util.logging.Logger.getLogger("org.hibernate").setLevel(Level.OFF
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        } catch (Throwable sfe) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("SessionFactory creation failed." + sfe);
            throw new ExceptionInInitializerError(sfe);
        }
    }

    /*
     * Close hibernate session
     */
    public static void shutdownSessionFactory() {
        // Close caches and connection pools
        getSessionFactory().close();
    }

    /*
     * Get method to obtain the session
     */
    public static SessionFactory getSessionFactory() {
        return SFACTORY;
    }

}
```

8.2. Transactions

A Hibernate **Session** object represents a single unit of work for a given data store that is opened by a **SessionFactory** instance. When creating a session, a transaction associated with it must be created. A transaction is defined as an atomic set of operations that are performed on a given data set. The transaction will be terminated when we make an accept invocation (**commit** method) or a revoke invocation (**rollback** method).

When all operations associated with the transaction have been completed, the session must be closed.

- The **beginTransaction()** method marks the start of the transaction.
- The **commit** method commits the changes to the database.
- The **rollback** method undoes the changes made.

For example:

```
//get a session
Session hibSession = HibernateUtil.SFACTORY.openSession();
//create a transaction
Transaction txDB = hibSession.beginTransaction();
//persistence code
..........................
//validate the transaction
txDB.commit();
//close the session
hibSession.close();
```

8.3. Objects management in the "EntityManager"

For a successful JPA-managed application, the **EntityManager** must be given a central role in the processing of entities. In fact, we can consider it as the local representative or interlocutor of the DBMS to our application. In order to manage the synchronisation between the entities of the object-oriented model and the DBMS, the EntityManager needs to always keep in memory a reference to the instances that the application generates during the execution of the application.

The entity manager can obtain the references to be managed in several ways. If it is a stored entity, each time the EntityManager obtains one or more entities using the find method or from the execution of a query, at the time of instantiating each entity, it will also record a reference to it in order to control the synchronisation with the database from that moment onwards.

a) Method persist

If it is a new entity, not yet stored in the database, it will be possible to pass the reference to the EntityManager by invoking the **persist** method. This method, in addition to passing the reference to the manager, will insert all the necessary records to the DBMS tables to store the entity data.

b) Method merge

If it is an entity instantiated in the application, already stored in the DBMS, but not yet referenced in the EntityManager, the **merge** method must be used, which will save the reference in the EntityManager in a similar way to persist, but instead of trying to register the entity in the DBMS, if necessary, it will opt for an update of the DBMS records corresponding to the referenced entity.

The merge method is actually a very flexible method. The main objective of this method is to synchronise the instance passed by parameter with the DBMS, so that depending on whether the entity is already registered or not in the DBMS, it will choose to insert or only update the entity, respectively. As long as the entity manager remains active, it is not necessary to invoke the merge method every time the state of an entity changes, since the references of the manager also maintain the same changes for each instance.

c) Method flush

It is possible to **force a real synchronisation with the DBMS** by invoking the **flush** method. The execution of this method implies synchronisation of all the entities recorded by the manager that are pending updating. Using the flush method will minimise the network traffic between the application and the DBMS, since we will concentrate the synchronisation dialogue at the points where we invoke the method. Using this strategy it will be necessary to always invoke the method before closing in case an update is pending. The use of flush, instead of merge, is also called passive persistence, since the programmer does not have to worry about having to perform each update one by one.

d) Method remove

Using the entity manager, we can also persistently delete those instances that no longer need to be part of those stored in the DBMS, by invoking the **remove** method of the manager. To do this, the manager must have a reference of the entity that we want to delete and, therefore, it will be necessary to obtain it by one of the ways indicated above.

8.4. States of a Hibernate object

Hibernate defines the following three object states:

State	Description
Transient	An object is transient if it has been instantiated but is not associated with any session. It has no persistent representation in the database. Transient objects will be destroyed by the rubbish collector. We can make a transient object persistent by associating it with a Hibernate session.
Persistent	A persistent object shall be associated with a Hibernate session. It may have been saved or loaded to/from the DB. Hibernate will detect any changes to a persistent object and synchronise its state with the database when the unit of work is completed.
Detached	An object is in this state when we log out. A detached instance is an object that has been made persistent but whose session has been closed. A detached instance can be associated with a new instance and become a persistent object again.

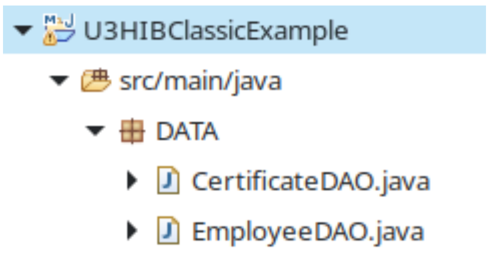
For example:

```java
//define a transient object
Employee objEmp = new Employee();
objEmp.setFirstname("Pepe");
objEmp.setLastname("Pérez");
objEmp.setSalary(1100);
//persist() makes the object persistent
hibSession.persist(objEmp);
```

9. Hibernate. CRUD operations

We are now going to implement the DAO classes following the POJO methodology. These classes will allow us to perform operations on the database: creation, reading, update and deletion (**CRUD**, **C**reate, **R**ead, **U**pdate and **D**elete).

We create a new **DATA package** with all the necessary DAO classes, each with the necessary methods to interact with the database via Hibernate:



EmployeeDAO

```
package DATA;

import java.util.List;
import java.util.Set;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.exception.*;
import org.hibernate.TransientPropertyValueException;
import org.hibernate.ObjectNotFoundException;

import DOMAIN.*;
import UTIL.*;

/**
 * ================================================================================
 * Data layer. Program to manage CRUD operations to the DB. Table Employee
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ============================================================================
 */

public class EmployeeDAO {

    /**
     * ----------------------------------------
     * GLOBAL CONSTANTS AND VARIABLES
     * ----------------------------------------
     */
    //Object name constant
    static final String OBJEMPNAME = "Employee";
    static final String OBJCERTNAME = "Certificate";

    /**
     * ----------------------------------------
     * METHODS. CRUD OPERATIONS
     * ----------------------------------------
     */
```

CertificateDAO

```
package DATA;

import java.util.List;
import java.util.Set;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.exception.*;
import org.hibernate.TransientPropertyValueException;
import org.hibernate.ObjectNotFoundException;
import DOMAIN.*;
import UTIL.*;

/**
 * ================================================================================
 * Data layer. Program to manage CRUD operations to the DB. Table Certificate
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ================================================================================
 */

public class CertificateDAO {

    /**
     * ----------------------------------------
     * GLOBAL CONSTANTS AND VARIABLES
     * ----------------------------------------
     */
    //Object name constant
    static final String OBJCERTNAME = "Certificate";
    static final String OBJEMPNAME = "Employee";

    /**
     * ----------------------------------------
     * METHODS. CRUD OPERATIONS
     * ----------------------------------------
     */
```

9.1. Loading objects

To load objects from the database we will use the following methods of the Session class:

Method	Description
<T> T load(Class<T> Clase, Serializable id) (DEPRECATED)	Returns the persistent instance of the specified class with the given identifier. The instance must exist, otherwise the method throws an exception, that is, throws the ObjectNotFoundException if the row does not exist. f we are not sure that the record is actually in the database, it is more convenient to use the get() method, which returns null if the object does not exist.
Object load (String nombreClas, Serializable id) (DEPRECATED)	Similar to the previous one, in this case we indicate the name of the class in the form of a String.
<T> get(Class <T> Clase, Serializable id)	Returns the persistent instance of the specified class with the given identifier. The instance must exist, otherwise null is returned.
Object get(String nombreClase, Serializable id)	Similar to the previous one, in this case we indicate the name of the class in the form of String.

9.1.1. Read

EmployeeDAO

The following example obtains, in addition to the data of the employee passed as a parameter, the name of all certificates held by that employee. To do so, we use the method getCertificates() of the Employee class and an iterator to go through the list of certificates.

```
/*
 * ------------------
 * SELECT
 * ------------------
 */
/* READ an employee and all its certificates */
public void listEmployee(int iEmpID) {

    Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
    Transaction txDB = null; //database transaction

    try {
        txDB = hibSession.beginTransaction(); //starts transaction
        Employee objEmployee = (Employee) hibSession.get(Employee.class, i
        if (objEmployee != null) {
            System.out.print(OBJEMPNAME + " first name: " + objEmployee.ge
            System.out.print(OBJEMPNAME + " last name: " + objEmployee.get
            System.out.print(OBJEMPNAME + " salary: " + objEmployee.getSal
            Set<Certificate> setCertificates = objEmployee.getCertificates
            for (Iterator<Certificate> itCertificate = setCertificates.ite
                Certificate objCertificate = (Certificate) itCertificate.n
                System.out.println(OBJCERTNAME + " name: " + objCertificat
            }
        } else {
            System.out.print("The " + OBJEMPNAME + iEmpID + "does not exis
        }
```

```
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

CertificateDAO

The following example obtains, in addition to the data of the certificate passed as a parameter, the name, surname and salary of all the employees who hold that certificate. To do so, we use the method getEmployees() of the Certificate class and an iterator to go through the list of employees.

```
    /*
     * -----------------
     * SELECT
     * -----------------
     */
    /* READ a certificate and all its employees */
    public void listCertificate(int iCertID) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            Certificate objCertificate = (Certificate) hibSession.get(Certific
            if (objCertificate != null) {
                System.out.print(OBJCERTNAME + " name: " + objCertificate.get(
                Set<Employee> setEmployees = objCertificate.getEmployees();
                for (Iterator<Employee> itEmployee = setEmployees.iterator();
                    Employee objEmployee = (Employee) itEmployee.next();
                    System.out.println(OBJEMPNAME + " name: " + objEmployee.ge
                                    OBJEMPNAME + " last name: " + objEmployee.
                                    OBJEMPNAME + " salary: " + objEmployee.get
                }
            } else {
                System.out.print("The " + OBJCERTNAME + iCertID + "does not ex
            }
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

9.2. Storage, modification and deletion of objects

For storage, modification and deletion we use the following Session methods:

Method	Description
Serializable persist (Object obj)	Stores the object passed as an argument in the database. Makes the transient instance of the object persistent. (save method is deprecated: https://stackoverflow.com/questions/71211904/alternative-to-using-deprecated-save-method-in-hibernate)
void merge (Object obj)	Updates the object passed as an argument in the DB. The object to be modified must be loaded with the load() or get() method. (update method is deprecated: https://stackoverflow.com/questions/71211904/alternative-to-using-deprecated-save-method-in-hibernate)
void remove (Object obj)	Removes the object passed as an argument from the database. The object to be deleted must be loaded with the load() or get() method. (delete method is deprecated: https://stackoverflow.com/questions/71211904/alternative-to-using-deprecated-save-method-in-hibernate)

Both merge and remove are VOID methods; therefore, in order to inform the user of the success or failure of an operation, we can use the structure shown below and catch the exception org.hibernate.ObjectNotFoundException:

```
try{}
catch(){}
```

9.2.1. Create

When inserting rows in a table we may violate:

- The **primary key constraint**. → The **ConstraintViolationException** will be thrown.
- The **foreign key constraint**. → The **TransientPropertyValueException** will be thrown.

It is necessary to control both exceptions to ensure the correct functioning of the application.

EmployeeDAO

```
/*
 * ------------------
 * INSERT
 * ------------------
 */
/* CREATE an employee in the database */
public Employee addEmployee(String stFirstName, String stLastName, double

    Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
    Transaction txDB = null; //database transaction
    Employee objEmployee = new Employee(stFirstName, stLastName, dSalary,

    try {
        txDB = hibSession.beginTransaction(); //starts transaction
        hibSession.persist(objEmployee);
        txDB.commit(); //ends transaction
        System.out.println("***** " + OBJEMPNAME + " added.\n");
    } catch (ConstraintViolationException cve) {
        System.out.println("Duplicate "+ OBJEMPNAME +".\n");
    } catch (TransientPropertyValueException tve) {
        System.out.println("The " + OBJEMPNAME + " entered does not exist.
    } catch (HibernateException hibe) {
        if (txDB != null)
```

```
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
        return objEmployee;
    }
```

CertificateDAO

```
    /*
     * ------------------
     * INSERT
     * ------------------
     */
    /* CREATE a certificate in the database */
    public Certificate addCertificate(String stCertName) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction
        Certificate objCertificate = new Certificate(stCertName, null);

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            hibSession.persist(objCertificate); //object to state persistent
            txDB.commit(); //ends transaction
            System.out.println("***** " + OBJCERTNAME + " added.\n");
        } catch (ConstraintViolationException cve) {
            System.out.println("Duplicate "+ OBJCERTNAME +".\n");
        } catch (TransientPropertyValueException tve) {
            System.out.println("The " + OBJCERTNAME + " entered does not exist
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
        return objCertificate;
    }
```

9.2.2. Update

EmployeeDAO

The following code allows us to modify an employee in the database. If there is a foreign key, we must take into account that a ConstraintViolationException may occur if we set a value that does not exist.

```
/*
     * ------------------
     * UPDATE
     * ------------------
     */
    /* Method to UPDATE salary for an employee */
    public void updateEmployee(int iEmpID, double dSalary) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            Employee objEmployee = (Employee) hibSession.get(Employee.class, i
            objEmployee.setSalary(dSalary);
            hibSession.merge(objEmployee);
            txDB.commit(); //ends transaction
            System.out.println("***** " + OBJEMPNAME + " updated.\n");
        } catch (ObjectNotFoundException onfe) {
            System.out.print("The " + OBJEMPNAME + iEmpID + "does not exist.\r
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

CertificateDAO

The following code allows us to modify a certificate in the database. If there is a foreign key, we must take into account that a ConstraintViolationException may occur if we set a value that does not exist.

```
/*
     * -----------------
     * UPDATE
     * -----------------
     */
    /* UPDATE name for a certificate */
    public void updateCertificate(int iCertID, String stCertName) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            Certificate objCertificate = (Certificate) hibSession.get(Certific
            objCertificate.setCertname(stCertName);
            hibSession.merge(objCertificate);
            txDB.commit(); //ends transaction
            System.out.println("***** " + OBJCERTNAME + " updated.\n");
        } catch (ObjectNotFoundException onfe) {
            System.out.print("The " + OBJCERTNAME + iCertID + "does not exist.
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

9.3.3. Delete

EmployeeDAO

The following code allows us to delete an employee from the database.

```
/*
     * -----------------
     * DELETE
     * -----------------
     */
    /* Method to DELETE an employee from the records */
    public void deleteEmployee(int iEmpID) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            Employee objEmployee = (Employee) hibSession.get(Employee.class, i
            hibSession.remove(objEmployee);
            txDB.commit(); //ends transaction
            System.out.println("***** " + OBJEMPNAME + " deleted.\n");
        } catch (ObjectNotFoundException onfe) {
            System.out.print("The " + OBJEMPNAME + iEmpID + "does not exist.\r
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

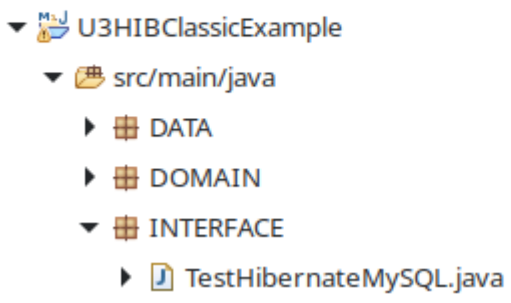
CertificateDAO

The following code allows us to delete a certificate from the database.

```
/*
     * -----------------
     * DELETE
     * -----------------
     */
    /* DELETE a certificate from the records */
    public void deleteCertificate(int iCertID) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            Certificate objCertificate = (Certificate) hibSession.get(Certific
            hibSession.remove(objCertificate);
            txDB.commit(); //ends transaction
            System.out.println("***** " + OBJCERTNAME + " deleted.\n");
        } catch (ObjectNotFoundException onfe) {
            System.out.print("The " + OBJCERTNAME + iCertID + "does not exist.
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

10. Hibernate. Interface layer

Finally we will create the class structure of the application. For our example we will create a Main.java class (main programme).



In this example we perform the following operations:

- Insert 2 certificates.
- Create the set of certificates.
- Insert 2 employees and assign all certificates to each of them.
- List the 2 employees together with their certificate.
- Update the salary of the first employee.
- Delete the second employee.
- List the 2 certificates together with their employees.

```
package INTERFACE;

import java.util.HashSet;

import DATA.*;
import DOMAIN.*;
import UTIL.*;

/**
 * ======================================================================
 * Interface layer. Program to manage Employees and Certificates
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ======================================================================
 */

/*
 * For further information see this tutorial:
 * https://www.tutorialspoint.com/hibernate/hibernate_examples.htm
 */

public class TestHibernateMySQL {

    /*
     * -----------------
     * MAIN PROGRAMME
     * -----------------
     */
    public static void main(String[] stArgs) {

        //Create new objects DAO for CRUD operations
        EmployeeDAO objEmployeeDAO = new EmployeeDAO();
        CertificateDAO objCertificateDAO = new CertificateDAO();

        // Add certificates in the database
        Certificate objCert1 = objCertificateDAO.addCertificate("MBA");
        Certificate objCert2 = objCertificateDAO.addCertificate("PMP");

        //Set of certificates
        HashSet<Certificate> hsetCertificates = new HashSet<Certificate>();
        hsetCertificates.add(objCert1);
        hsetCertificates.add(objCert2);

        // Add employees in the database
        Employee objEmp1 = objEmployeeDAO.addEmployee("Alfred", "Vincent", 40
        Employee objEmp2 = objEmployeeDAO.addEmployee("John", "Gordon", 3000,

        // Update employee's salary field
        objEmployeeDAO.updateEmployee(objEmp1.getEmpId(), 5000);
        // List down all the employees and their certificates
        objEmployeeDAO.listEmployee(objEmp1.getEmpId());
        objEmployeeDAO.listEmployee(objEmp2.getEmpId());
        // Delete an employee from the database
        objEmployeeDAO.deleteEmployee(objEmp2.getEmpId());
        // List down all the certificates and their employees
        objCertificateDAO.listCertificate(objCert1.getCertId());
        objCertificateDAO.listCertificate(objCert2.getCertId());

        //Close global hibernate session factory
        HibernateUtil.shutdownSessionFactory();
    }

}
```

11. Hibernate. HQL

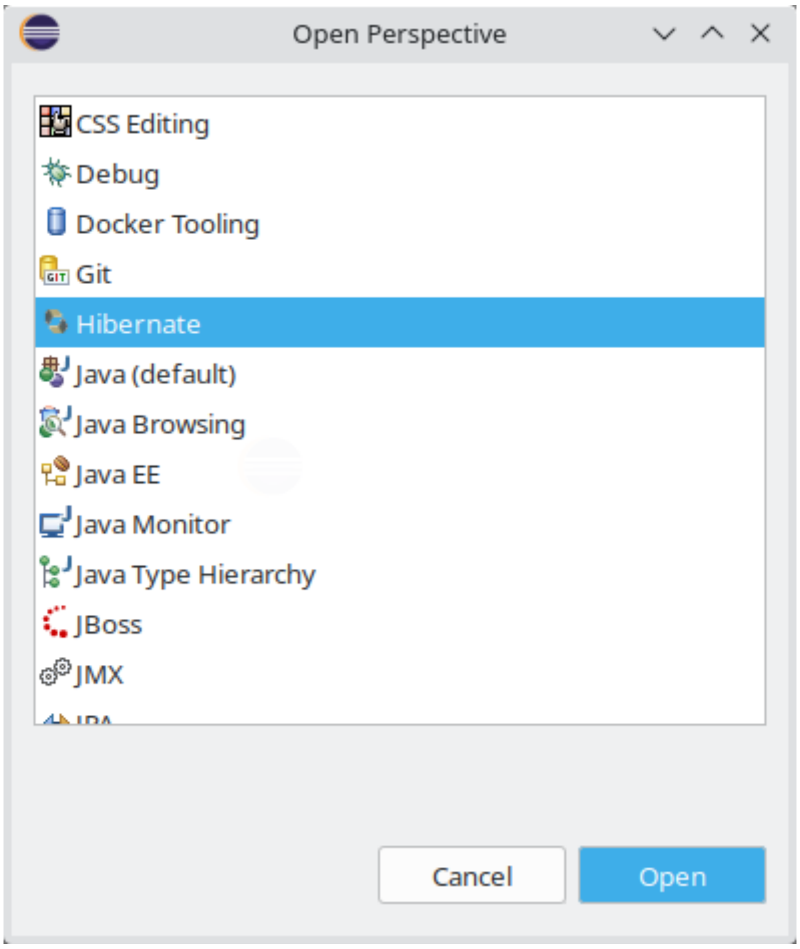
HQL (**H**ibernate **Q**uery **L**anguage) is the object-oriented query language of Hibernate Framework. HQL is very similar to SQL except that we use Objects instead of table names, that makes it more close to object oriented programming.

In this section we will see a simple example of how to make a query in HQL. We will look at this language in more detail later on.

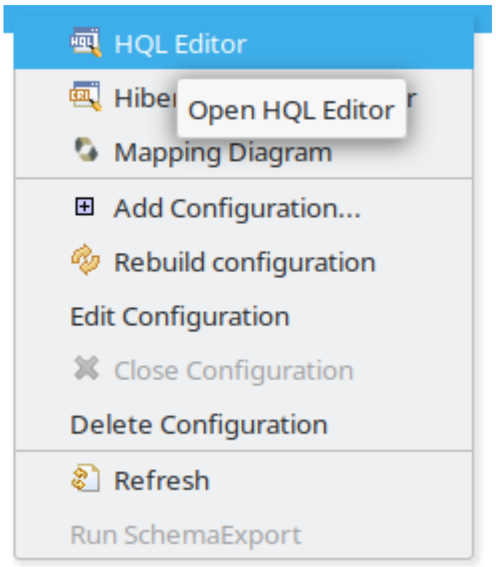
11.1. First select at HQL

Next we are going to perform an HQL query to check that the database connection is working correctly.

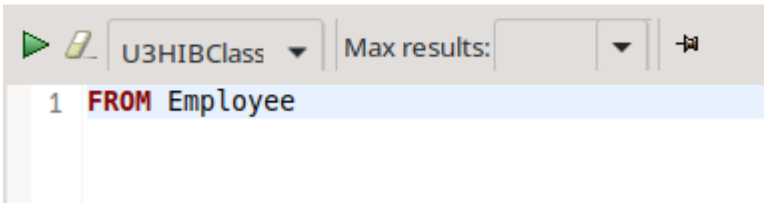
- We open the Hibernate Perspective from **Window** → **Perspective** → **Open Perspective** → **Other** → **Hibernate**.



- Click on the button Open.
- From the Hibernate Configurations tab, right-click on the configuration named U3HIBClassicExample and select **HQL Editor**.



- A tab with the same name as the Hibernate configuration opens. From here we can write HQL statements to query the database.
  - We write the query "FROM Employee".
  - Click on the "Play" button to execute it.



```
FROM Employee
```

- If everything went well we should be able to see a list of employee objects in response to the query.

11.2. Listing objects

EmployeeDAO

The following method makes a query that returns all the rows of the table Employee and their certificates.

```
    /* READ all the employees */
    public void listEmployees() {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            //old createQuery method is deprecated, but still working in the l
            //https://www.roseindia.net/hibernate/hibernate5/hibernate-5-query
            List<Employee> listEmployees = hibSession.createQuery("FROM " + OB
            if (listEmployees.isEmpty())
                System.out.println("******** No items found");
            else
                System.out.println("\n***** Start listing ...\n");

            for (Iterator<Employee> itEmployee = listEmployees.iterator(); itE
                Employee objEmployee = (Employee) itEmployee.next();
                System.out.print(OBJEMPNAME + " First Name: " + objEmployee.ge
                System.out.print("Last Name: " + objEmployee.getLastname() + '
                System.out.println("Salary: " + objEmployee.getSalary());
                Set<Certificate> setCertificates = objEmployee.getCertificates
                for (Iterator<Certificate> itCertificate = setCertificates.ite
                    Certificate objCertificate = (Certificate) itCertificate.r
                    System.out.println(OBJCERTNAME + " name: " + objCertificat
                }
            }
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

CertificateDAO

The following method makes a query that returns all the rows of the table Certificate.

```
/* READ all the certificates */
    public void listCertificates() {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            //old createQuery method is deprecated
            //https://www.roseindia.net/hibernate/hibernate5/hibernate-5-query
            List<Certificate> listCertificates = hibSession.createQuery("FROM
            if (listCertificates.isEmpty())
                System.out.println("******** No items found");
            else
                System.out.println("***** Start listing ...\n");

            for (Iterator<Certificate> itCertificate = listCertificates.iterat
                Certificate objCertificate = (Certificate) itCertificate.next(
                System.out.print(OBJCERTNAME + " name: " + objCertificate.get(
            }
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

11.3. Deleting objects

EmployeeDAO

```
/* Method to DELETE all records */
    public void deleteAllItems() {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            //old createQuery method is deprecated, but still working in the l
            //https://www.roseindia.net/hibernate/hibernate5/hibernate-5-query
            List<Employee> listEmployees = hibSession.createQuery("FROM " + OE
            if (!listEmployees.isEmpty())
                for (Iterator<Employee> itEmployee = listEmployees.iterator();
                    Employee objEmployee = (Employee) itEmployee.next();
                    hibSession.remove(objEmployee);
                }
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

CertificateDAO

```
/* Method to DELETE all records */
    public void deleteAllItems() {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibe
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            //old createQuery method is deprecated, but still working in the l
            //https://www.roseindia.net/hibernate/hibernate5/hibernate-5-query
            List<Certificate> listCertificate = hibSession.createQuery("FROM '
            if (!listCertificate.isEmpty())
                for (Iterator<Certificate> itCertificate = listCertificate.ite
                    Certificate objCertificate = (Certificate) itCertificate.r
                    hibSession.remove(objCertificate);
                }
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

11.4. Interface layer

Now we modify the main class to use the new methods with HQL language:

```java
package INTERFACE;

import java.util.HashSet;

import DATA.*;
import DOMAIN.*;
import UTIL.*;

/**
 * ======================================================================
 * Interface layer. Program to manage Employees and Certificates
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ======================================================================
 */

/*
 * For further information see this tutorial:
 * https://www.tutorialspoint.com/hibernate/hibernate_examples.htm
 */

public class TestHibernateMySQL {

    /*
     * -----------------
     * MAIN PROGRAMME
     * -----------------
     */
    public static void main(String[] stArgs) {
```

```
//Create new objects DAO for CRUD operations
EmployeeDAO objEmployeeDAO = new EmployeeDAO();
CertificateDAO objCertificateDAO = new CertificateDAO();

//TRUNCATE TABLES. Delete all records from the tables
objEmployeeDAO.deleteAllItems();
objCertificateDAO.deleteAllItems();

// Add certificates in the database
Certificate objCert1 = objCertificateDAO.addCertificate("MBA");
Certificate objCert2 = objCertificateDAO.addCertificate("PMP");

//Set of certificates
HashSet<Certificate> hsetCertificates = new HashSet<Certificate>();
hsetCertificates.add(objCert1);
hsetCertificates.add(objCert2);

// Add employees in the database
Employee objEmp1 = objEmployeeDAO.addEmployee("Alfred", "Vincent", 400
Employee objEmp2 = objEmployeeDAO.addEmployee("John", "Gordon", 3000,

// Update employee's salary field
objEmployeeDAO.updateEmployee(objEmp1.getEmpId(), 5000);
// List down all the employees and their certificates
//objEmployeeDAO.listEmployee(objEmp1.getEmpId());
//objEmployeeDAO.listEmployee(objEmp2.getEmpId());
objEmployeeDAO.listEmployees();
// Delete an employee from the database
objEmployeeDAO.deleteEmployee(objEmp2.getEmpId());
// List down all the certificates and their employees
//objCertificateDAO.listCertificate(objCert1.getCertId());
//objCertificateDAO.listCertificate(objCert2.getCertId());
// List down all the certificates
objCertificateDAO.listCertificates();
// List down all the employees
objEmployeeDAO.listEmployees();
```

```
            //Close global hibernate session factory
            HibernateUtil.shutdownSessionFactory();
        }

    }
```

12. Bibliography

Sources

- How to Create a Maven Project in Eclipse. https://www.simplilearn.com/tutorials/maven-tutorial/maven-project-in-eclipse
- Josep Cañellas Bornas, Isidre Guixà Miranda. Accés a dades. Desenvolupament d'aplicacions multiplataforma. Creative Commons. Departament d'Ensenyament, Institut Obert de Catalunya. Dipòsit legal: B. 29430-2013. https://ioc.xtec.cat/educacio/recursos
- Alberto Oliva Molina. Acceso a datos. UD 3. Herramientas de mapeo objeto relacional (ORM). IES Tubalcaín. Tarazona (Zaragoza, España).
- Hibernate ORM Documentation - 6.5. https://hibernate.org/orm/documentation/6.5/
- JBoss Tools. Eclipse Plugins for JBoss Technology. https://tools.jboss.org/
- How to Write Doc Comments for the Javadoc Tool. https://www.oracle.com/es/technical-resources/articles/java/javadoc-tool.html
- A Guide to Hibernate Query Language. https://docs.jboss.org/hibernate/orm/6.3/querylanguage/html_single/Hibernate_Query_Language.html
- Hibernate – Query Language. https://www.geeksforgeeks.org/hibernate-query-language/