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

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

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

Access to relational databases using Hibernate Annotations

Abelardo Martínez

Year 2024-2025

1. What is Hibernate annotations?

So far you have seen how Hibernate uses XML mapping file for the transformation of data from POJO to database tables and vice versa. Hibernate **annotations** are the newest way to **define mappings without the use of XML file**. You can use annotations in addition to or as a replacement of XML mapping metadata. Hibernate annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. All the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

1.1. Annotations vs XML

Advantages

- Annotations provide metadata configuration along with the Java code. That way **the code is easy to understand**.
- Annotations are **very easy to integrate with the code** even though they are not intrusive. That is, they are **completely transparent to the developer** who has to use the class model.
- Annotations are preconfigured with sensible default values, which **reduce the amount of coding required**, e.g. class name defaults to table name and field names defaults to column names.
- If you are going to make your application portable to other ORM applications, you must use annotations to represent the mapping information.

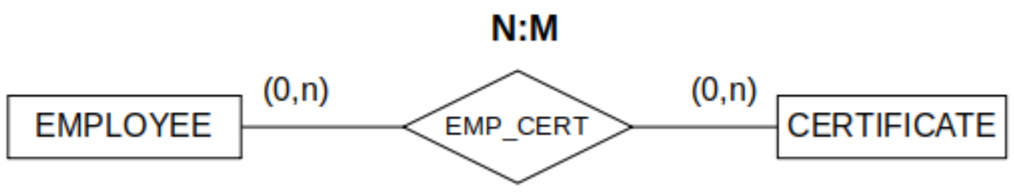
Drawbacks

- The use of **annotations requires the model sources to be available for mapping** (annotations are written in the code) and this is not always possible. In contrast, the XML version does not require them. In this sense, the XML system could be considered more independent.
- Annotations are **less powerful than XML configuration**. **XML** also **gives you the ability to change the configuration without building the project**. So use annotations only for table and column mappings, not for frequently changing stuff like database connection and other properties.
- If you want **greater flexibility**, then you should go with **XML**-based mappings.

2. Setting up the project and the database

2.1. The (MySQL) database

We will use the same database as seen in the ORM mapping with XML.



We create the database and the user:

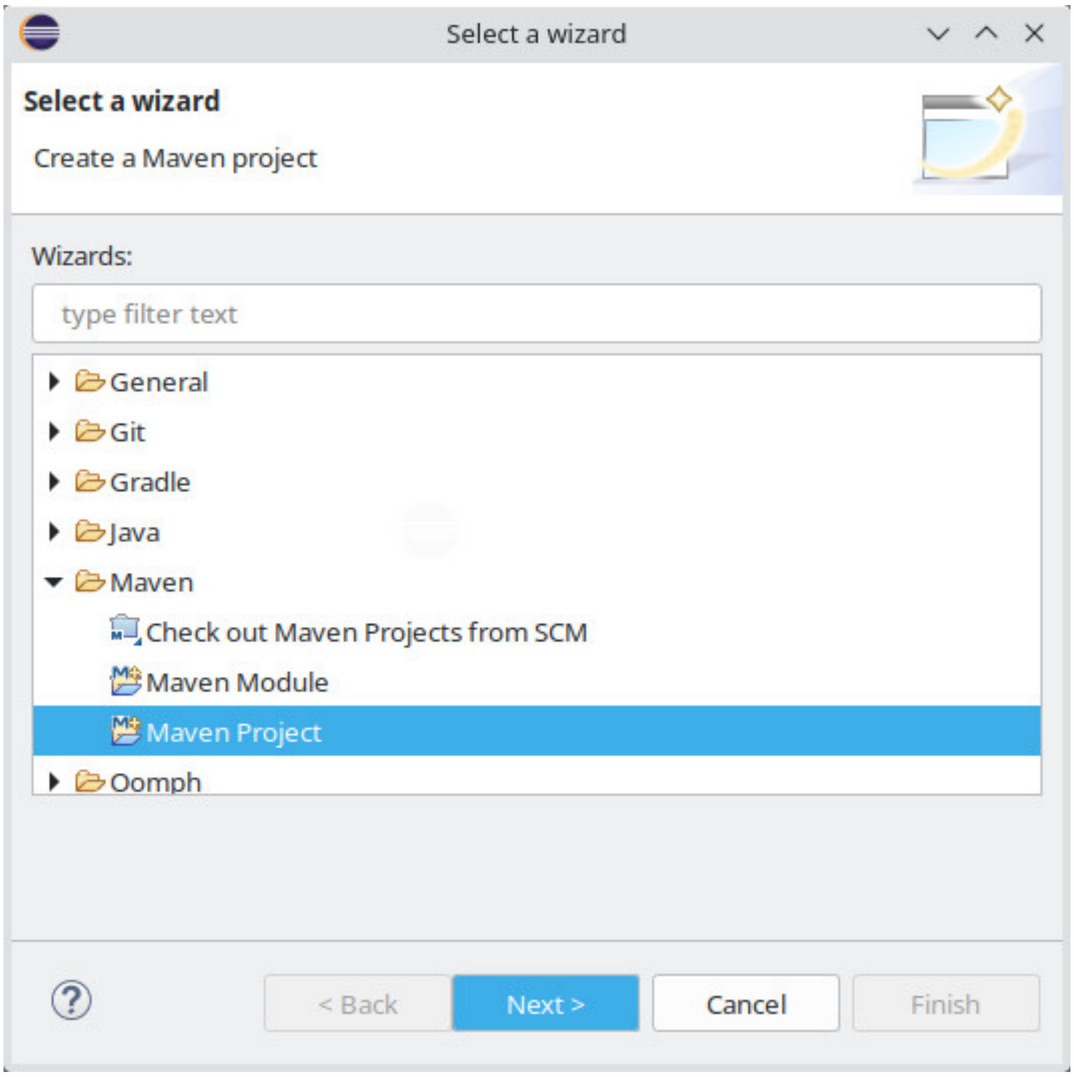
```
CREATE DATABASE IF NOT EXISTS ADAU3DBExample CHARACTER SET utf8mb4 COLLATE utf8mb4_es_0900_ai_ci;

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

USE ADAU3DBExample;
```

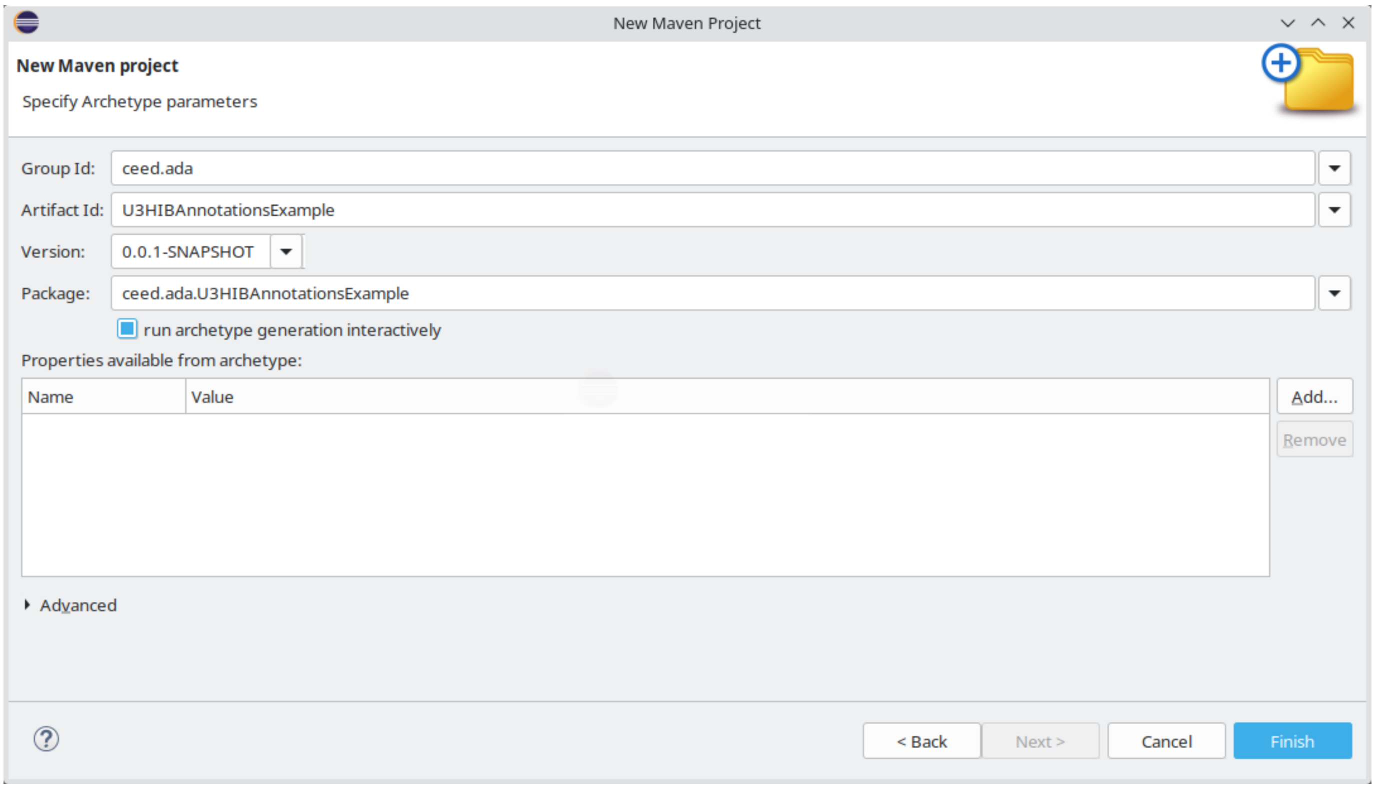
We create the structure:

```
CREATE TABLE Employee (
empID       INTEGER PRIMARY KEY AUTO_INCREMENT,
firstname   VARCHAR(20),
lastname    VARCHAR(20),
salary      FLOAT
);

CREATE TABLE Certificate (
certID      INTEGER PRIMARY KEY AUTO_INCREMENT,
certname    VARCHAR(30)
);

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

2.2. The (Java-Maven) project

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





Group Id: ceed.ada
Artifact Id: U3HIBAnnotationsExample
Version: 0.0.1-SNAPSHOT
Package: ceed.ada.U3HIBAnnotationsExample
run archetype generation interactively
Properties available from archetype:

Name	Value

2.3. 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

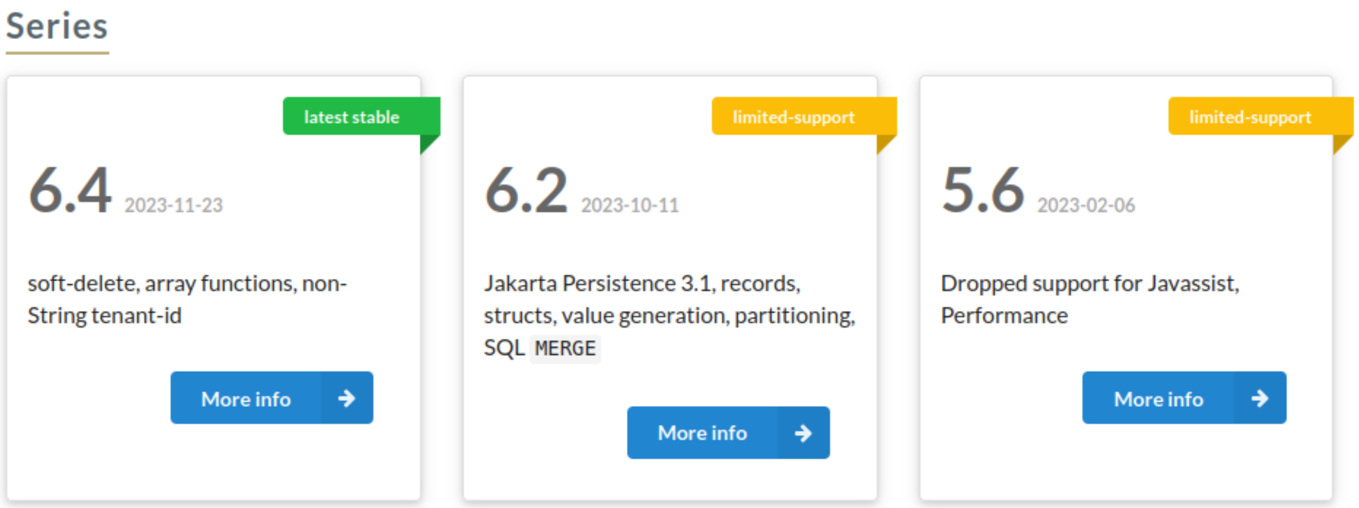
```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
        <!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
        <dependency>
       <groupId>com.mysql</groupId>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

3. Setting up Hibernate

3.1. Hibernate dependencies

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

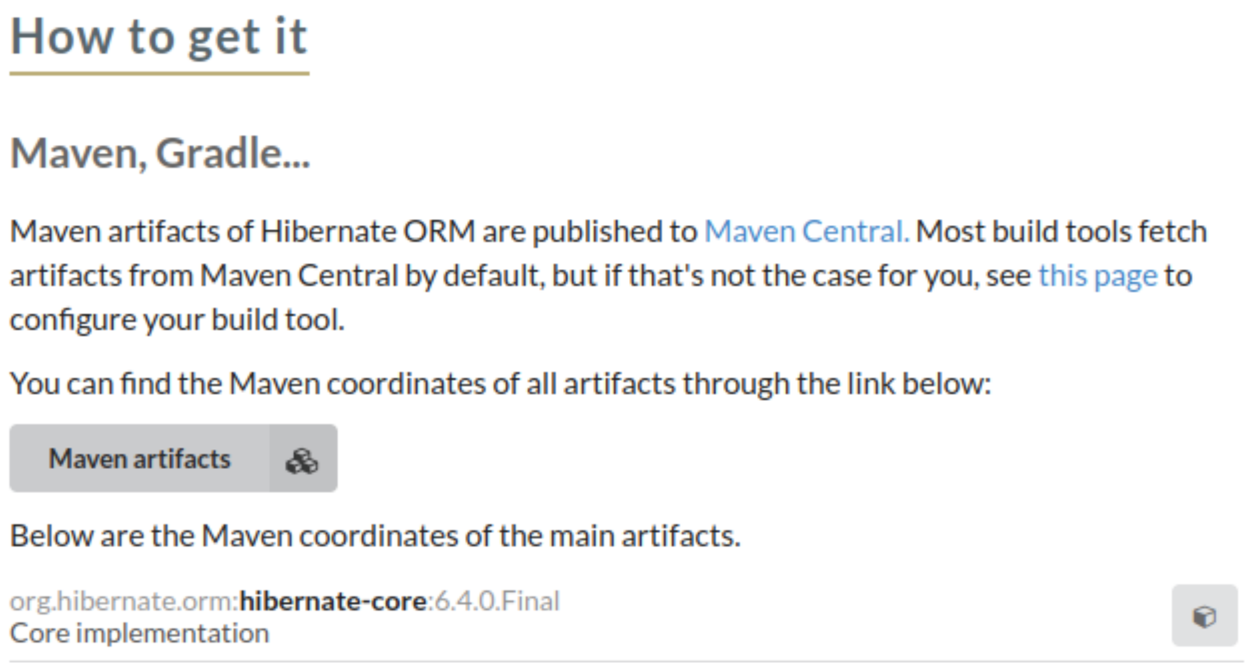
- Go to https://hibernate.org/orm/releases/
- Click on “More info” on the last stable version
  - Option 1:
    - Click on the link hibernate-core (x.x.x.Final)
  - Option 2:
    - Click on “Maven artifacts” button
    - Click on the link hibernate-core (x.x.x.Final)
    - Copy and paste the dependency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


Series

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

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

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



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/6.4.0.Final-->
<dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.4.0.Final</version>
</dependency>
```

3.2. How to include annotations?

We have 2 options for using annotations, although we will use the one in the POM file:

1) Install the annotation package

- First of all you would have to make sure that you are using JDK 5.0 (or higher) to take advantage of the native support for annotations.
- Second, you will need to install the Hibernate annotations distribution package, available from the sourceforge and copy hibernate-annotations.jar, lib/hibernate-comons-annotations.jar and lib/ejb3-persistence.jar from the Hibernate Annotations distribution to your CLASSPATH.

2) Using Maven through the POM file

Using Maven, you just need to add this new dependency to your pom.xml file:

```
<!--https://central.sonatype.com/artifact/org.hibernate/hibernate-annotations-->
    <dependency>
     <groupId>org.hibernate</groupId>
     <artifactId>hibernate-annotations</artifactId>
     <version>3.5.6-Final</version>
    </dependency>
```

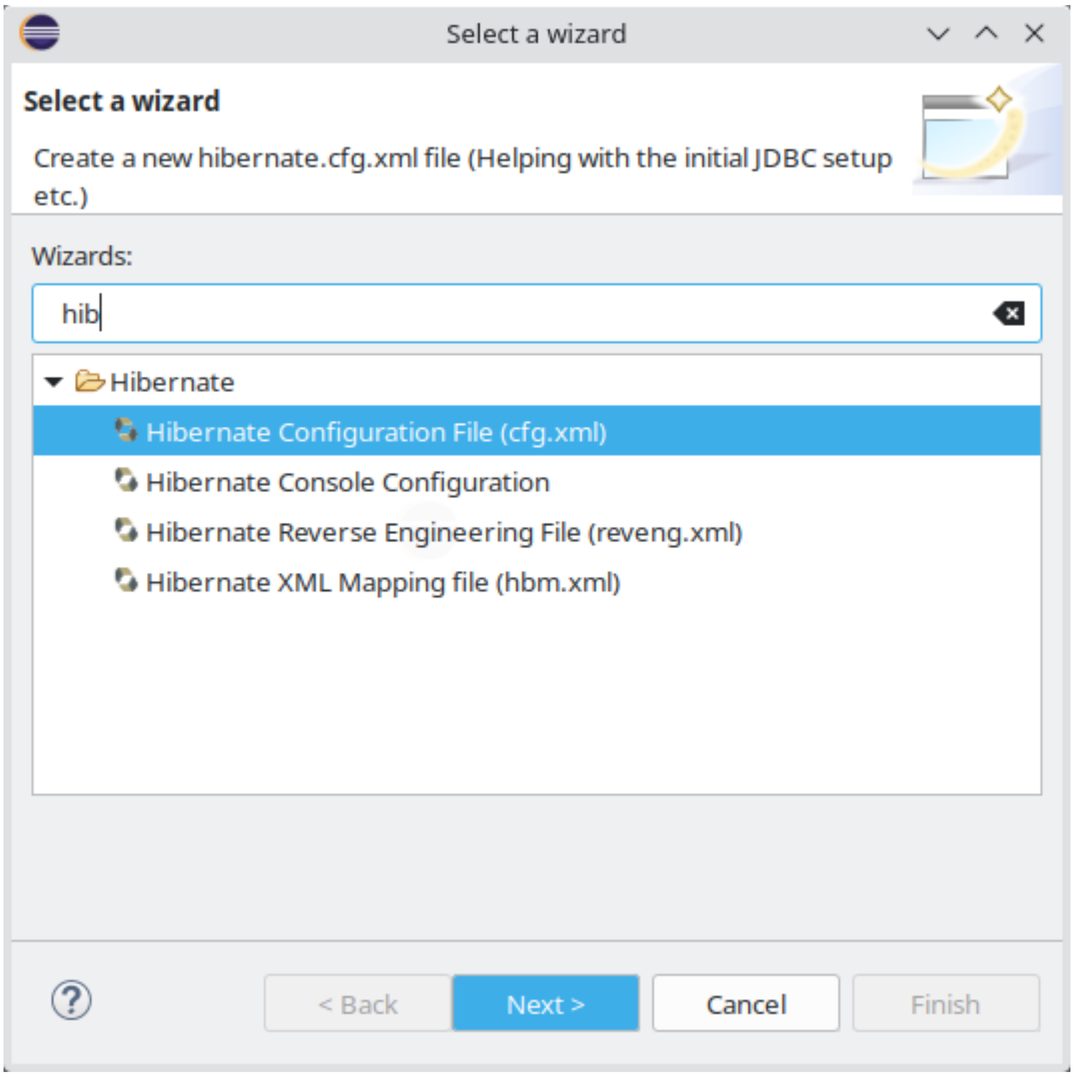
Maven repository: https://mvnrepository.com/artifact/org.hibernate/hibernate-annotations

3.3. XML configuration file

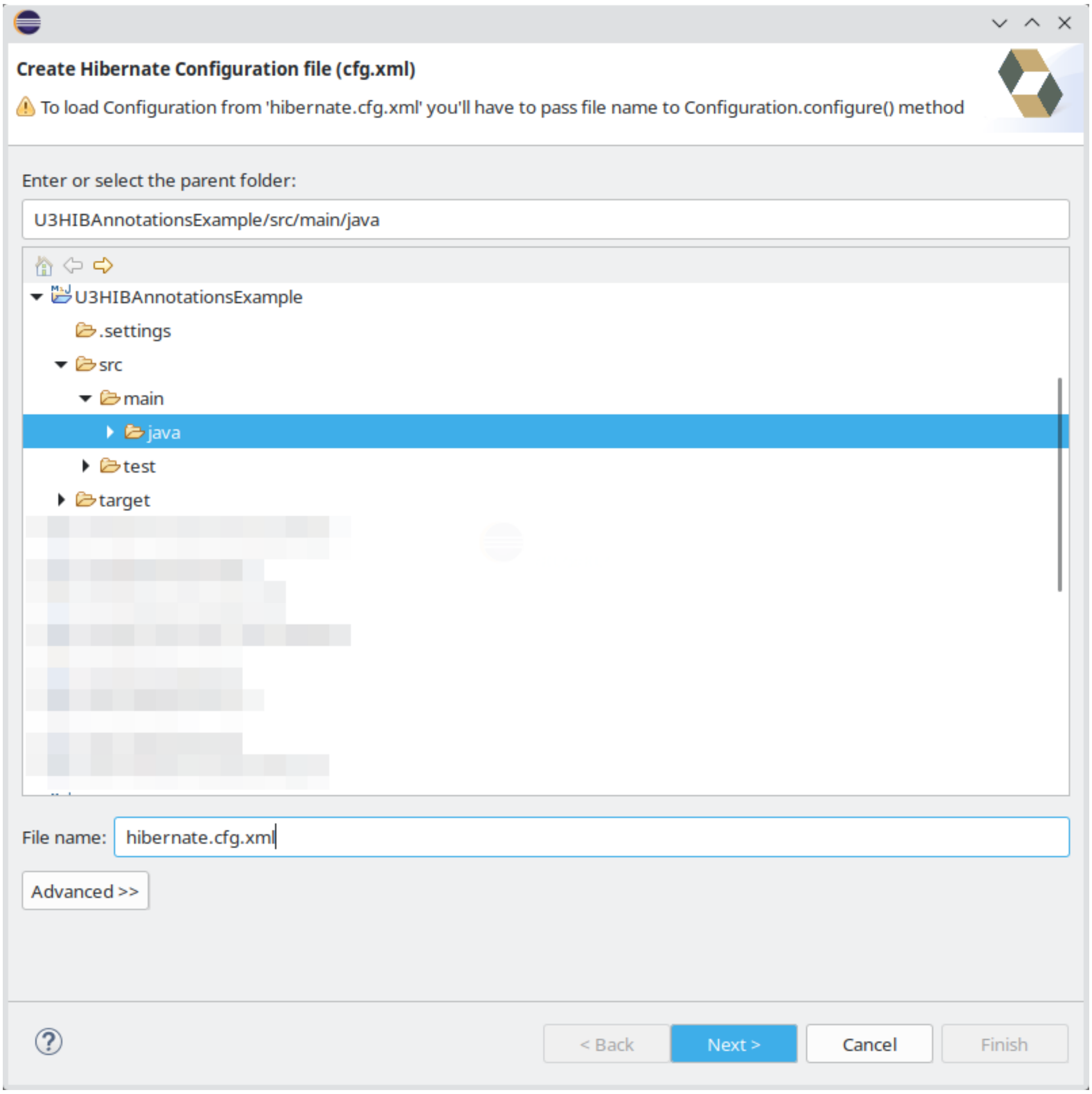
Generic config file (hibernate.cfg.xml)

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)
< Back
Next >
Cancel
Finish

- 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:

U3HIBAnnotationsExample/src/main/java

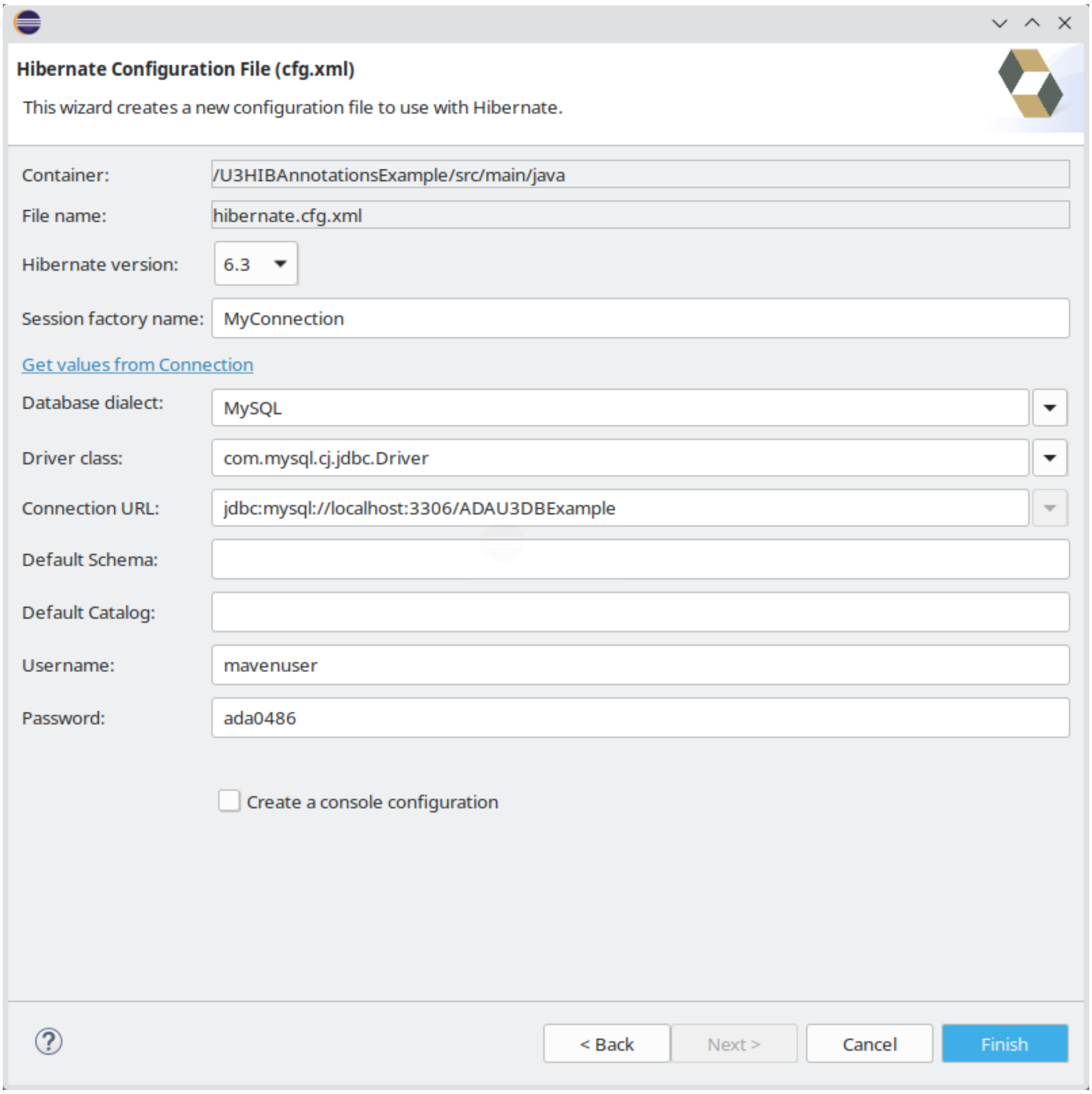
- U3HIBAnnotationsExample
  - .settings
  - src
    - main
      - java
    - test
  - target

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. In MySQL: jdbc:mysql://localhost:3306/DB name

- Username → User name
- Password → Password

- Finally click on the Finish button.



Hibernate Configuration File (cfg.xml)

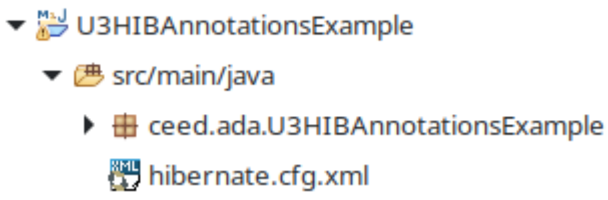
This wizard creates a new configuration file to use with Hibernate.

Container:	/U3HIBAnnotationsExample/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

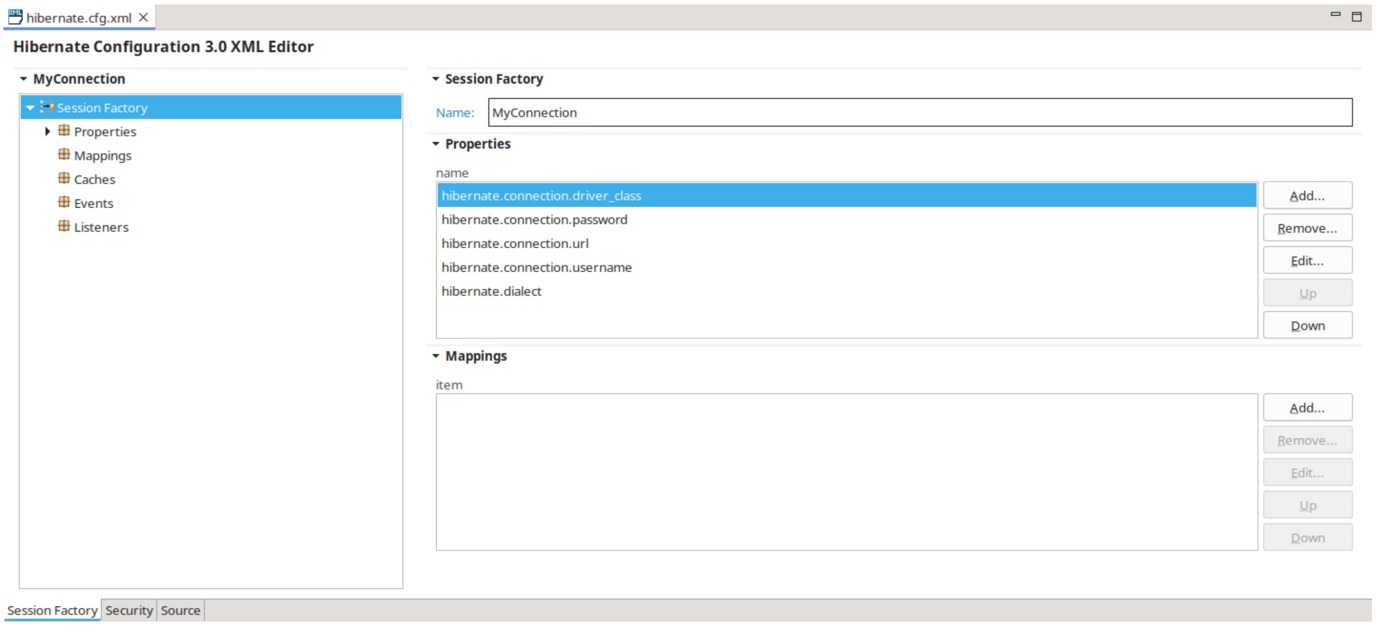
Get values from Connection

☐ Create a console configuration

- A new file has been created.



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



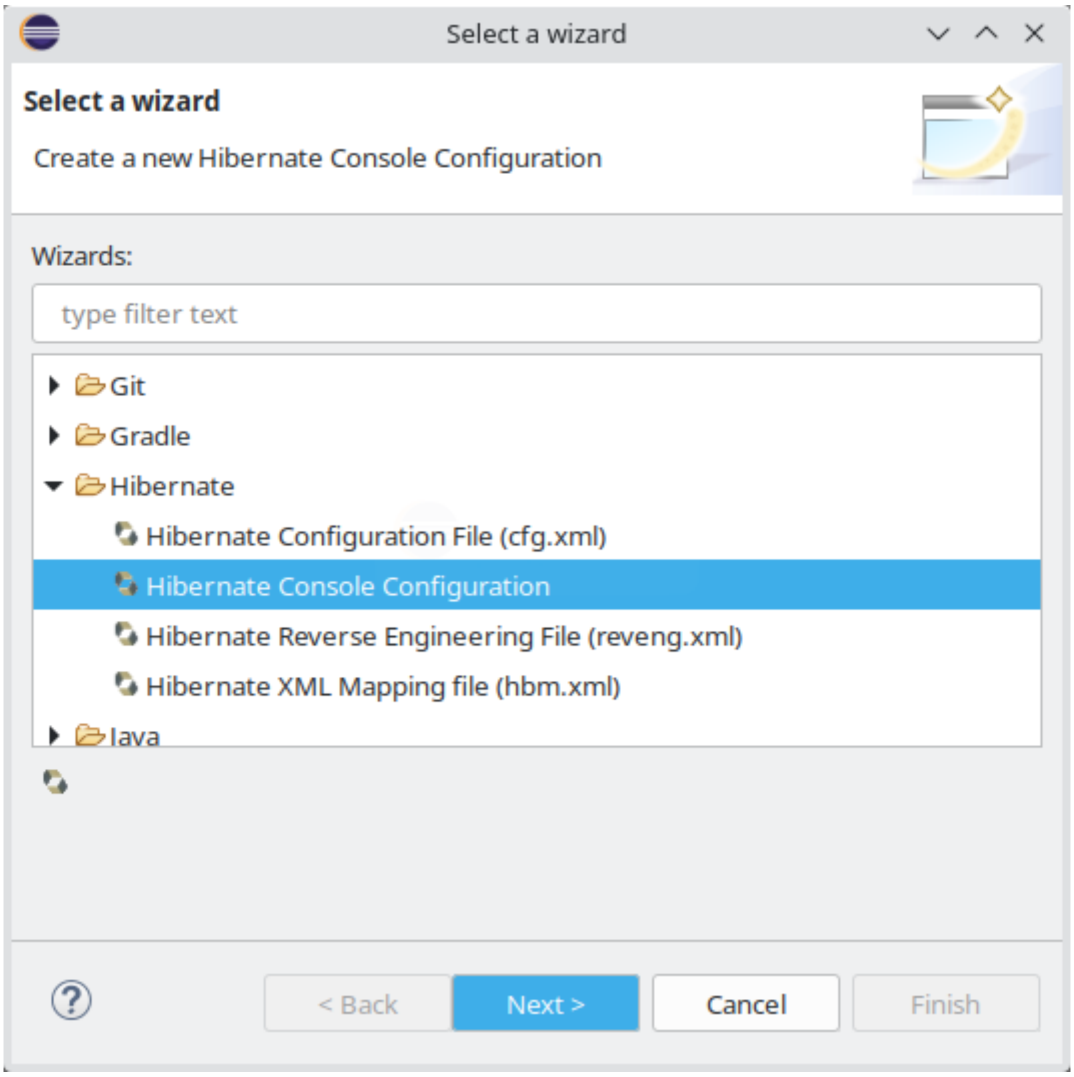
- And if you view it with a generic editor, you will see the XML code:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!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 name="MyConnection">
        <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>
    </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

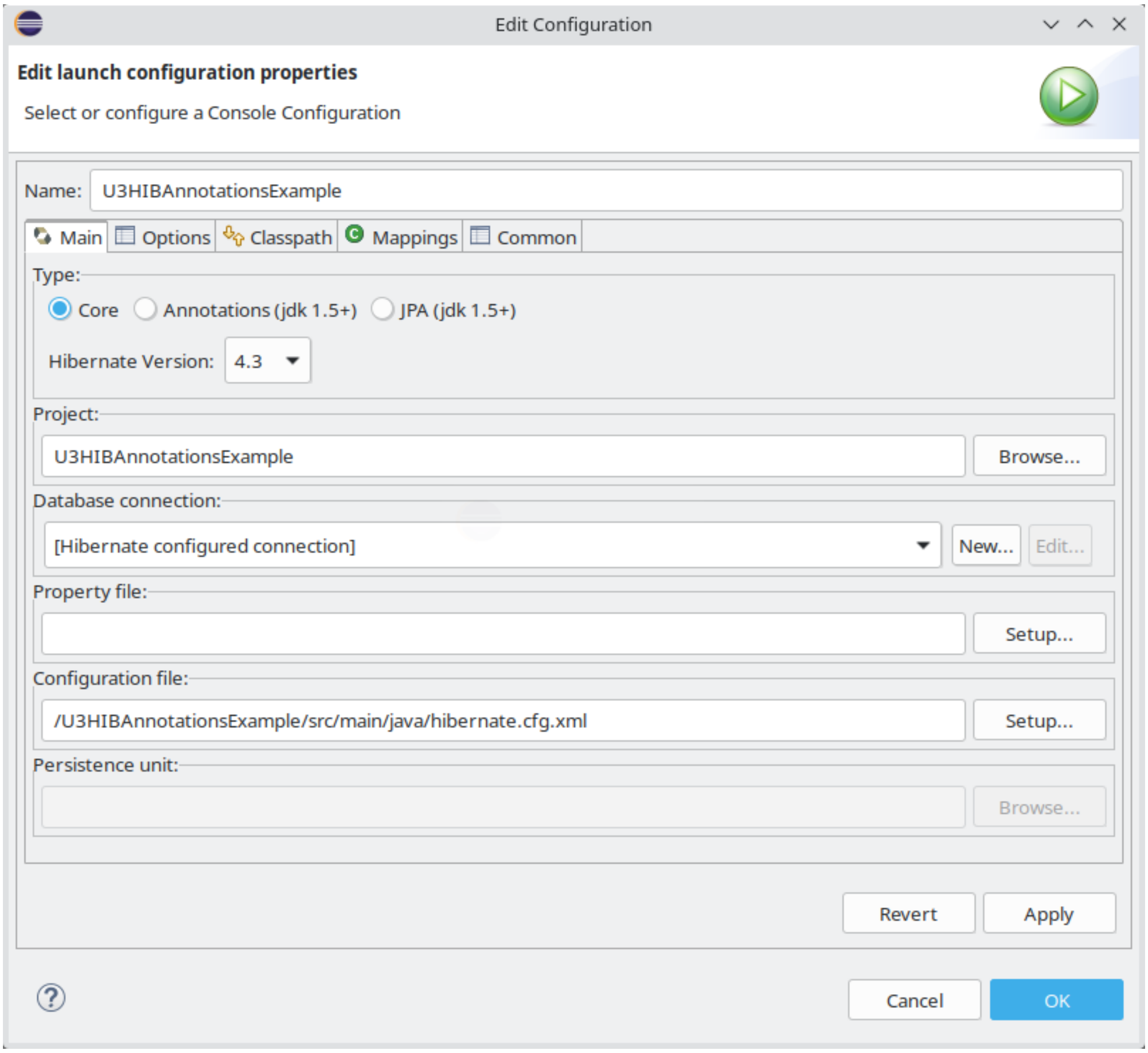
Wizards:

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

- 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.

> Note that the **Hibernate version must be 4.3** for the annotations to be generated (generation tool bug).



Name: U3HIBAnnotationsExample

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

Hibernate Version: 4.3

Project: U3HIBAnnotationsExample

Database connection: [Hibernate configured connection]

Property file:

Configuration file: /U3HIBAnnotationsExample/src/main/java/hibernate.cfg.xml

Persistence unit:

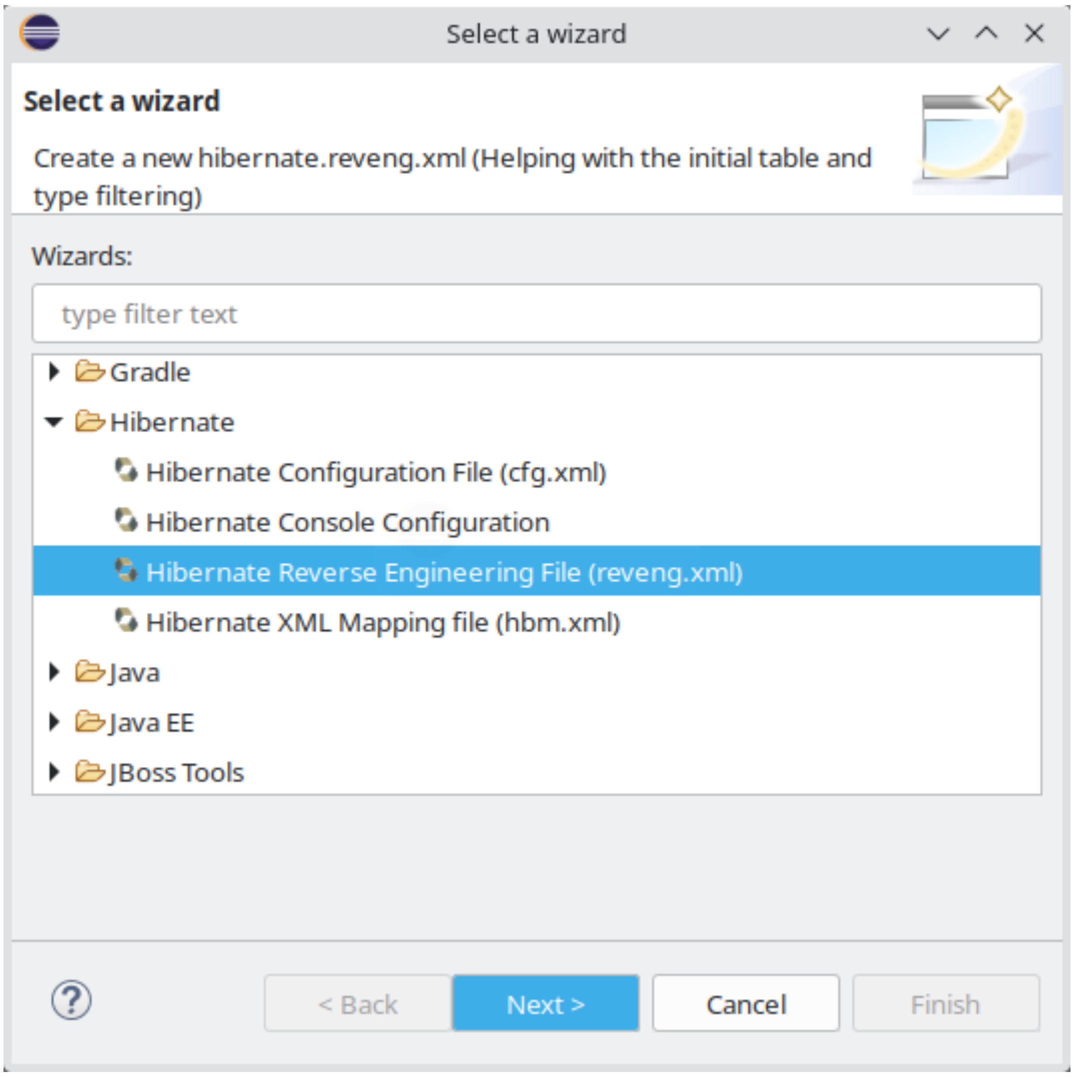
- Click Finish to finish the creation of the Console Configuration.

XML Hibernate Reverse Engineering

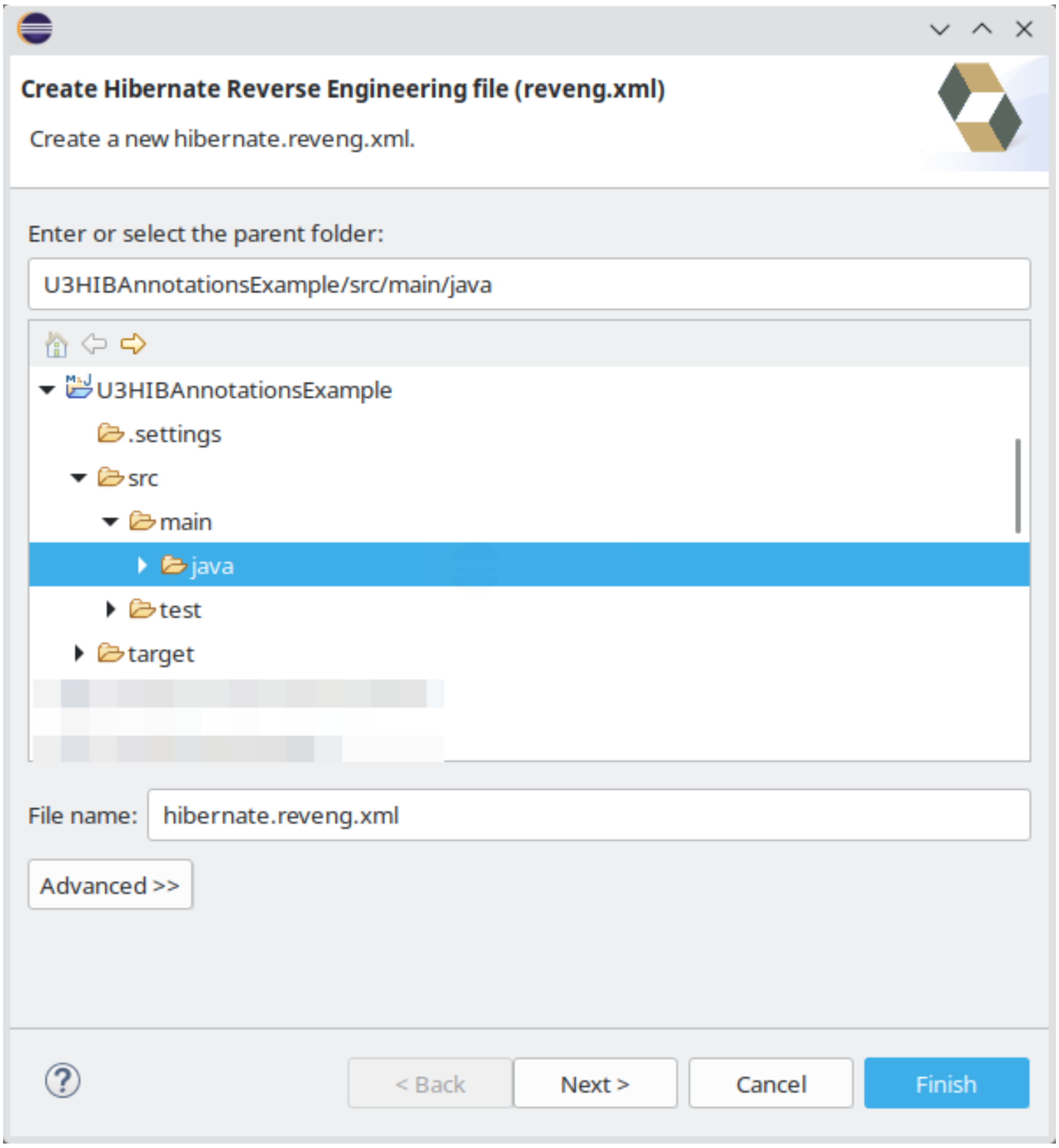
Finally we are going to create the **XML file Hibernate Reverse Engineering (reveng.xml)** which is in charge of creating the classes of our MySQL tables.

> 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)

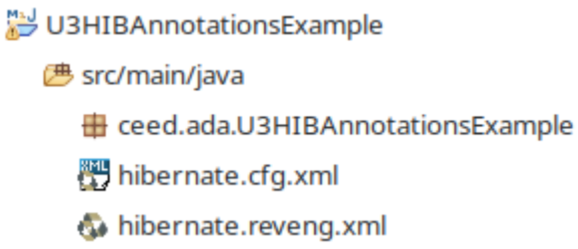
Create a new hibernate.reveng.xml.

Enter or select the parent folder:

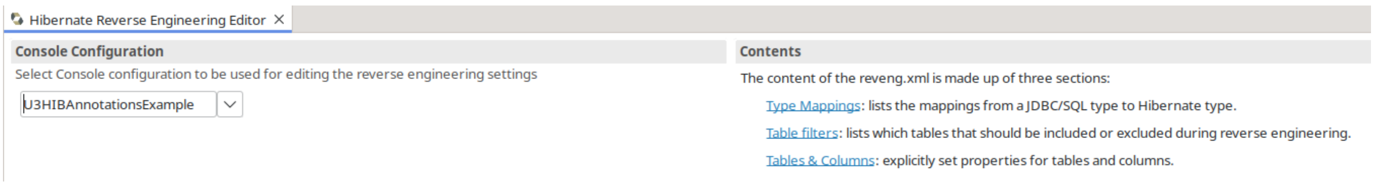
U3HIBAnnotationsExample/src/main/java

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.


Console Configuration
Select Console configuration to be used for editing the reverse engineering settings
U3HIBAnnotationsExample

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**".

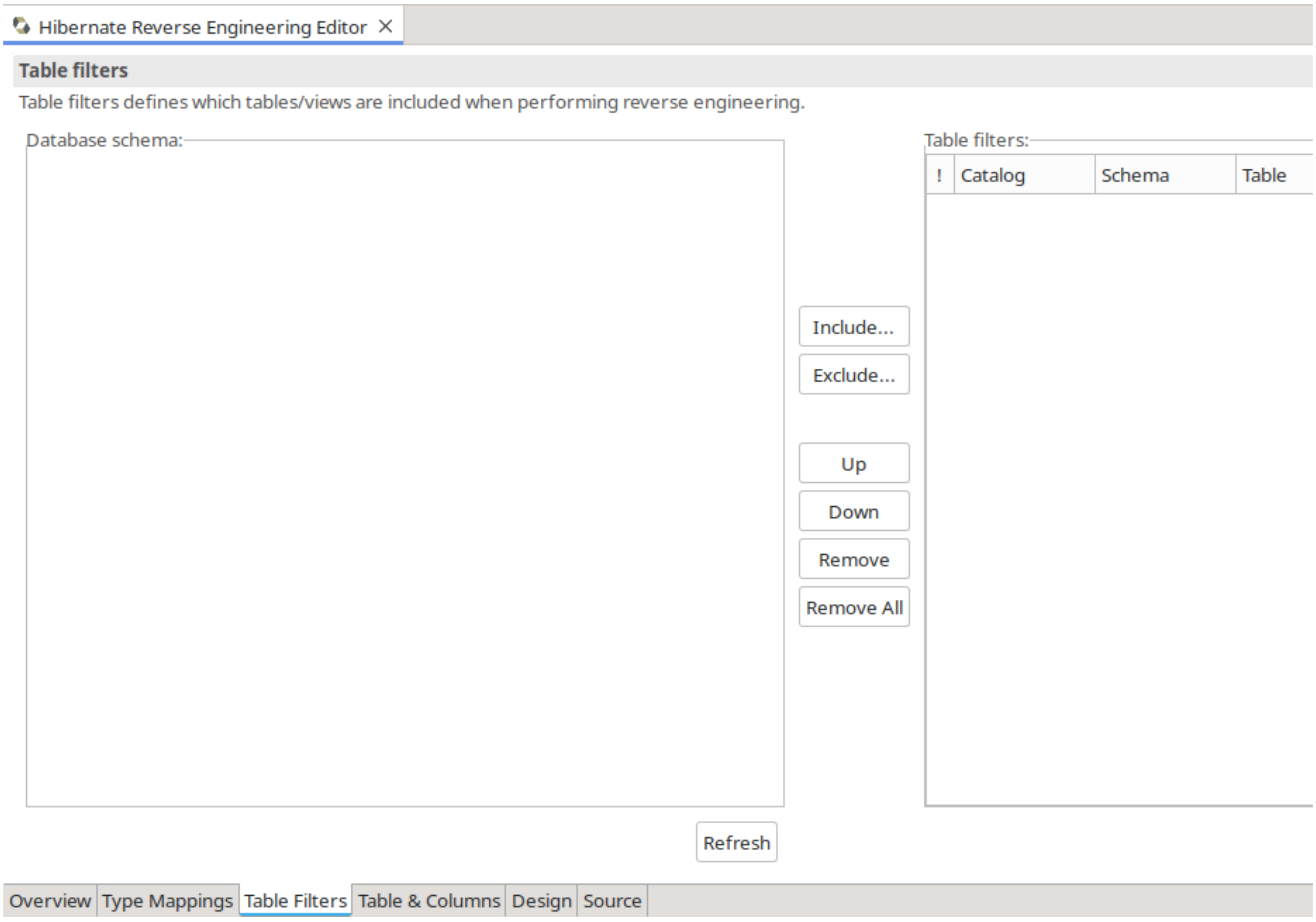
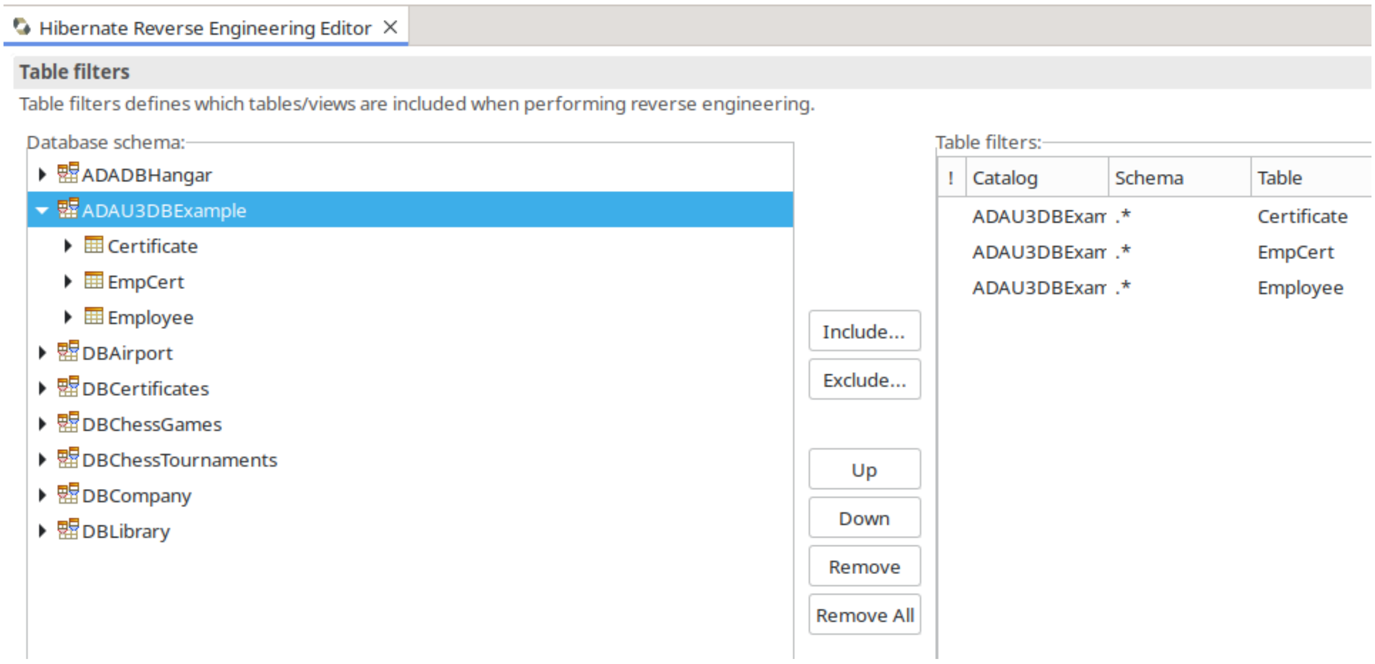


Table filters
Table filters defines which tables/views are included when performing reverse engineering.

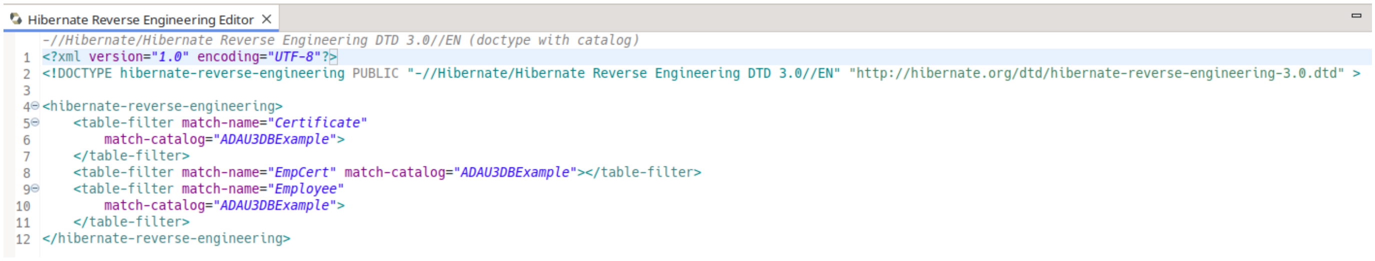
Hibernate Reverse Engineering Editor
Database schema:
Table filters:
! Catalog Schema Table
Include...
Exclude...
Up
Down
Remove
Remove All
Refresh
Overview Type Mappings Table Filters Table & Columns Design Source

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



!	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>
```

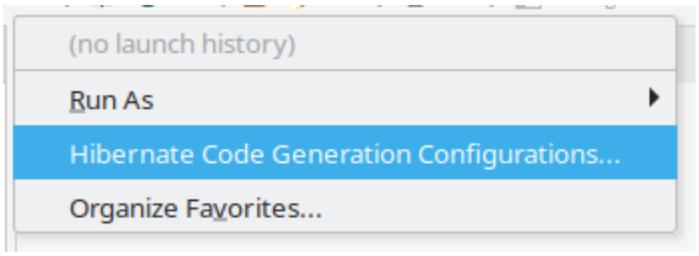
- Save the changes (**Ctrl+S** or **File** → **Save**).

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

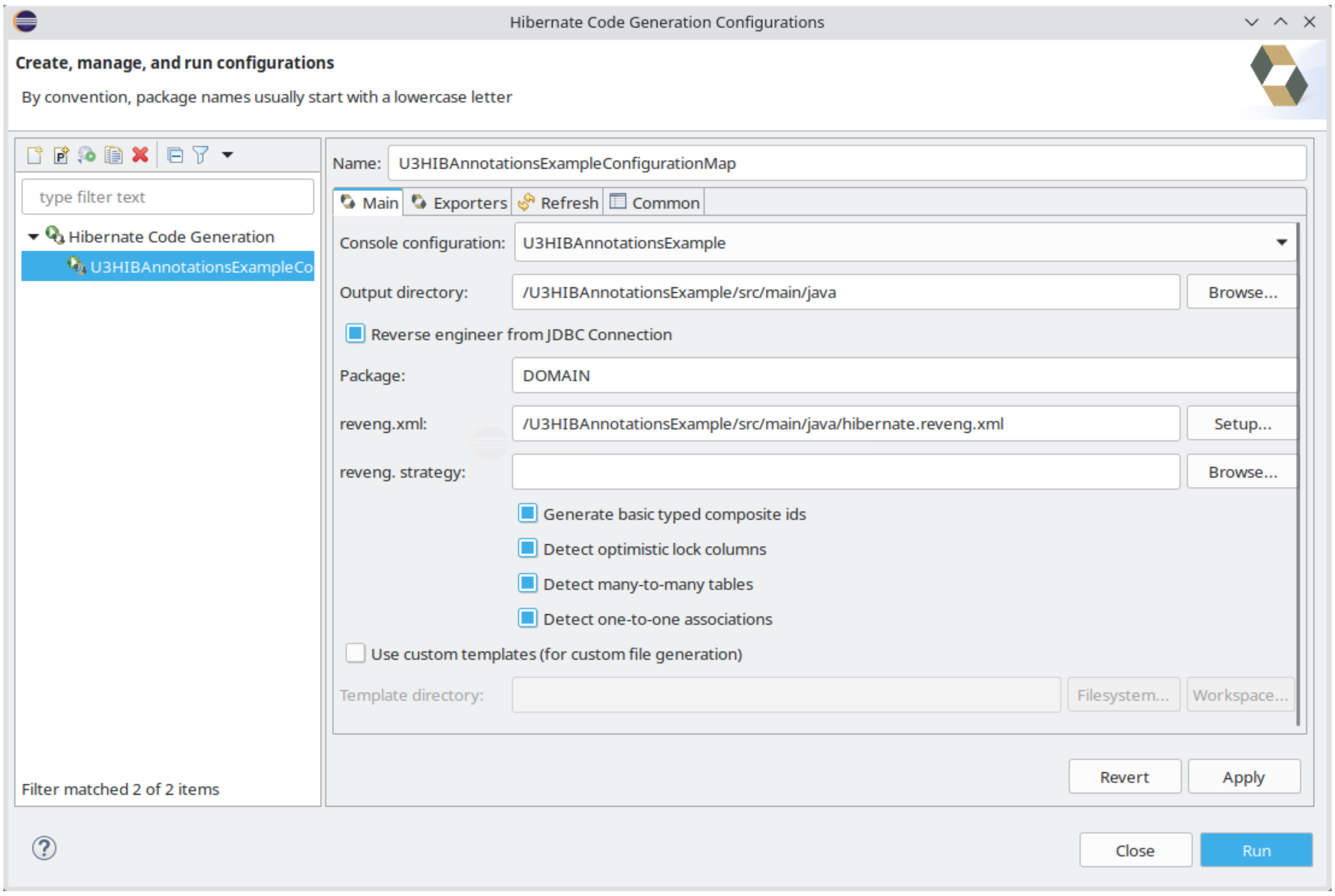
3.4. Generate database classes (POJO)

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 → **U3HIBAnnotationsExampleConfigurationMap**
  - 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
type filter text
Hibernate Code Generation
U3HIBAnnotationsExampleCo
Filter matched 2 of 2 items
Main
Exporters
Refresh
Common
Browse...
Setup...
Browse...
Filesystem...
Workspace...
Revert
Apply
Close
Run

Create, manage, and run configurations

By convention, package names usually start with a lowercase letter

Name: U3HIBAnnotationsExampleConfigurationMap

Console configuration: U3HIBAnnotationsExample

Output directory: /U3HIBAnnotationsExample/src/main/java

[✓] Reverse engineer from JDBC Connection

Package: DOMAIN

reveng.xml: /U3HIBAnnotationsExample/src/main/java/hibernate.reveng.xml

reveng. strategy:

[✓] Generate basic typed composite ids

[✓] Detect optimistic lock columns

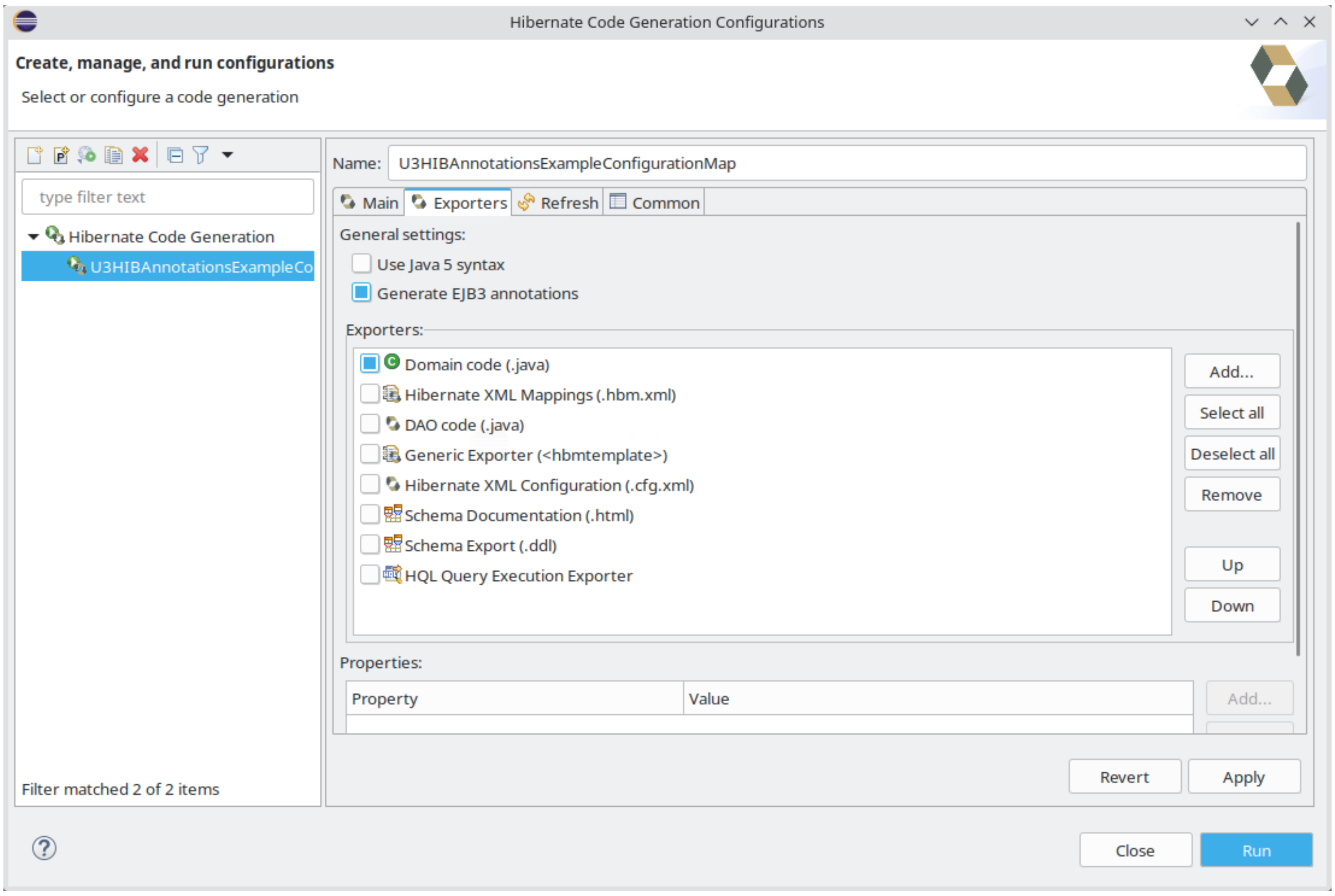
[✓] Detect many-to-many tables

[✓] Detect one-to-one associations

[ ] Use custom templates (for custom file generation)

Template directory:

- From the "Exporters" tab, indicate the files you want to generate, tick the boxes:
  - **Generate EJB3 annotations**
  - Domain Code


Hibernate Code Generation Configurations

Create, manage, and run configurations

Select or configure a code generation

type filter text

- Hibernate Code Generation
  - U3HIBAnnotationsExampleCo

Filter matched 2 of 2 items

Name: U3HIBAnnotationsExampleConfigurationMap

Main | Exporters | Refresh | Common

General settings:

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

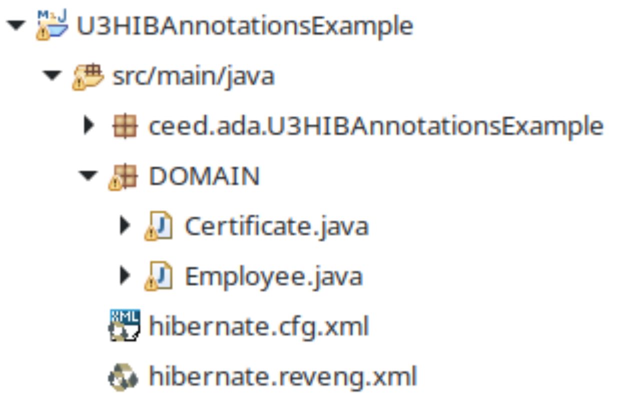
Exporters:

- [x] Domain code (.java)
- [ ] Hibernate XML Mappings (.hbm.xml)
- [ ] DAO code (.java)
- [ ] Generic Exporter (<hbmtemplate>)
- [ ] 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.


```
U3HIBAnnotationsExample
  src/main/java
    ceed.ada.U3HIBAnnotationsExample
    DOMAIN
      Certificate.java
      Employee.java
    hibernate.cfg.xml
    hibernate.reveng.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

3.5. Modifying the configuration file

We must **change the hibernate.cfg.xml** file to set this new mapping type. To do so, we add the corresponding class mapping lines. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!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 name="MyConnection">
        <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>
        <mapping class="DOMAIN.Employee" />
        <mapping class="DOMAIN.Certificate" />
    </session-factory>
</hibernate-configuration>
```

4. Hibernate. Annotations

Hibernate Annotations are based on the JPA 2 specification and supports all the features. All the JPA annotations are defined in the jakarta.persistence package. Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification. The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

> **Hibernate 6** moves from Java Persistence as defined by the Java EE specs to Jakarta Persistence as defined by the Jakarta EE spec. The most immediate impact of this change is that applications would need to be updated to **use the Jakarta Persistence classes (jakarta.persistence.*)** instead of the Java Persistence ones (javax.persistence.*).
> https://docs.jboss.org/hibernate/orm/6.0/migration-guide/migration-guide.html#_jakarta_persistence

There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc. The following are the most common annotations used in POJO classes specifically for hibernate:

Annotation	Description
@Entity	Used for declaring any POJO class as an entity for a database. This annotation can be applied on Class, Interface of Enums.
@Table	It specifies the table in the database with which this entity is mapped. Used to change table details, some of the attributes are: - name. Specifies the table name - schema - catalogue - enforce unique constraints
@Id	Specifies the primary key of the entity. Used for declaring a primary key inside our POJO class.
@GeneratedValue	Specifies the generation strategies for the values of primary keys. Hibernate automatically generate the values with reference to the internal sequence and we don't need to set the values manually.
@Column	It is used to specify column mappings. It means if in case we don't need the name of the column that we declare in POJO but we need to refer to that entity you can change the name for the database table. Some attributes are: - Name. Used for specifying the table's column name - length. The size of the column mostly used in strings - unique. The column is marked for containing only unique values - nullable. The column values should not be null. It's marked as NOT
@Transient	Tells the hibernate, not to add this particular column. Every non static and non-transient property of an entity is considered persistent, unless you annotate it as @Transient.
@Temporal	This annotation is used to format the date for storing in the database.
@Lob	Used to tell hibernate that it's a large object and is not a simple object.
@OrderBy	This annotation will tell hibernate to sort the data (OrderBy) as we do in SQL.

4.1. Marking entities

Entity is a class annotation that allows us to mark the entities we need to control. Any class containing this annotation can be made persistent with the **EntityManager**. It has already been mentioned that all entities must have an identifier. In order to be able to mark a class attribute as an identifying value, we have the Id annotation.

In case you do not want to store any of the attributes, you have to mark them as **Transient**. Attributes that are not so marked shall be considered persistent and, if nothing else is indicated, shall be considered to be stored in a column with the same name as the attribute.

4.2. Automatic generation of the primary key

It is very likely that in the processing of some entities, we will want to be unconcerned about the value of the primary key and decide to generate it automatically. If the application we are running, for example, were to generate order documents identified with a number that follows a sequential order, it would be quite tedious to having to remember the last number entered. It is easier to leave it to the system to automatically generate the corresponding value. The automatic generation of the primary key can also be a way to make the E-R model independent of the application objects. The OO model does not require the existence of primary keys and automatic generation could allow to completely ignore the requirement imposed by relational systems.

Most DBMSs have tools to generate primary keys automatically. The problem is that not all DBMSs share the same tools. While some DBMSs use sequences, others use a specific column type that will auto-increment the value each time there is an insertion. There are also DBMSs that do not have any tool at all. To accommodate all DBMSs, JPA has four strategies to achieve automatic key generation. Two of them use one of the mentioned DBMS tools. The other two are specific to JPA.

To select one of the strategies it will be sufficient to indicate which strategy we want to use using the notation **GeneretedValue** before the identifier attribute. This notation supports a parameter called **strategy**. The 4 possible values are defined as constants of the GenerationType class. The constants are:

- auto
- table
- sequence
- identity

4.2.1. Auto

The **GenerationType.AUTO** strategy is useful to work with during the development phase. In reality, each JPA implementation may handle this strategy differently. It is usually based on a RAM register that is incremented at each insertion and updated every time we start using JPA. It is useful during the development phase because it is a very fast strategy that does not require any manipulation of the DBMS, but it is not advisable to use it in the exploitation phase because there is no real guarantee that the generated key is indeed unique.

4.2.2. Table

The **GenerationType.TABLE** strategy is based on a normal DBMS table capable of storing one or more counters that will help to generate a unique value each time it is needed. This is a very flexible strategy that can be adapted to any DBMS. The table is created automatically along with the rest of the tables and contains two columns. The first is a string that will take the name of the class where the primary key value is to be generated and is used as the identifier of the counter. The second column is of integer type and stores the next value with which the new primary key will be generated.

4.2.3. Sequence

The **GenerationType.SEQUENCE** strategy is based on the use of DBMS proprietary sequences. This means that it can only be used in those DBMSs that support sequences. In most database systems, sequences have a single counter per sequence associated with them and are identified by a name.

Using annotations, we will indicate this using the SequenceGenerator annotation. This can be parameterised by specifying a name that will identify the sequence you want to use for the annotated class. **SequenceGenerator** will have to identify the sequence name and the name of the primary key generator used.

4.2.4. Identity

The **GenerationType.IDENTITY** strategy is also a DBMS-dependent strategy. It is specifically intended for DBMSs that have an auto-incremental type such as MySQL, Access or other DBMSs. It does not require any extra configuration elements. It is enough to indicate it and that the DBMS supports it.

4.3. Tables

The **Table** element allows you to indicate the name of the table where the entities will be stored, so that it is possible to work with a table that has a different name from the name of the entity. The catalogue and/or schema of the DBMS to which the table belongs can also be specified. Both are optional and only need to be specified in case the schema or catalogue of the table does not match the one configured by default in the PersistenceUnit.

The Table element **can also be used to generate single-value constraints** during table creation. A variable number of constraints may be specified, each of which may involve a variable number of columns. The Table attribute to be specified is **uniqueConstraints**, which shall contain a collection of values from another annotation called UniqueConstraint. This supports a parameter with the collection of column names that will be involved in the single-value constraint.

4.4. Columns

In the same way that the name of the tables is specified, the name of the columns can also be specified, as well as the characteristics with which they are to be generated. The annotation used is **Column**, indicated as a prefix to any attribute of a class.

Column is a very parameterisable annotation, since it allows to express many characteristics referring to the column. The following attributes commonly being overridden:

- **name**: permits the name of the column to be explicitly specified—by default, this would be the name of the property.
- **length**: permits the size of the column used to map a value (particularly a String value) to be explicitly defined. The column size defaults to 255, which might otherwise result in truncated String data, for example.
- **nullable**: permits the column to be marked NOT NULL when the schema is generated. The default is that fields should be permitted to be null; however, it is common to override this when a field is, or ought to be, mandatory.
- **unique**: permits the column to be marked as containing only unique values. This defaults to false, but commonly would be set for a value that might not be a primary key but would still cause problems if duplicated (such as username).

4.5. Delayed data upload

JPA allows marking certain attributes of entities with the **LAZY** mark, so that when retrieving stored objects, the data so marked will not be retrieved immediately, but only when there is an attempt to access the attribute. This feature is known as "lazy", delayed or deferred loading.

It is a technique widely used during the retrieval of entities with a large volume of data. Late retrieval applies to attributes such as images, collections, long text strings or any other type that involves mobilising a large number of bytes.

4.6. Relationship Mapping

Unidirectional associations are commonly used in object-oriented programming to establish relationships between entities. However, it’s important to note that in a unidirectional association, only one entity holds a reference to the other. To define a unidirectional association in Java, we can use annotations such as:

Annotation	Description
@OneToMany	In a one-to-many relationship, an entity has a reference to one or many instances of another entity.
@ManyToOne	In a many-to-one relationship, many instances of an entity are associated with one instance of another entity.
@OneToOne	In a one-to-one relationship, an instance of an entity is associated with only one instance of another entity.
@ManyToMany	In a many-to-many relationship, many instances of an entity are associated with many instances of another entity.
@PrimaryKeyJoinColumn	This annotation is used to associate entities sharing the same primary key.
@JoinColumn	It is used for one-to-one or many-to-one associations when foreign key is held by one of the entities.
@JoinTable	@JoinTable and mappedBy should be used for entities linked through an association table.
@MapsId	Two entities with shared key can be persisted using @MapsId annotation.

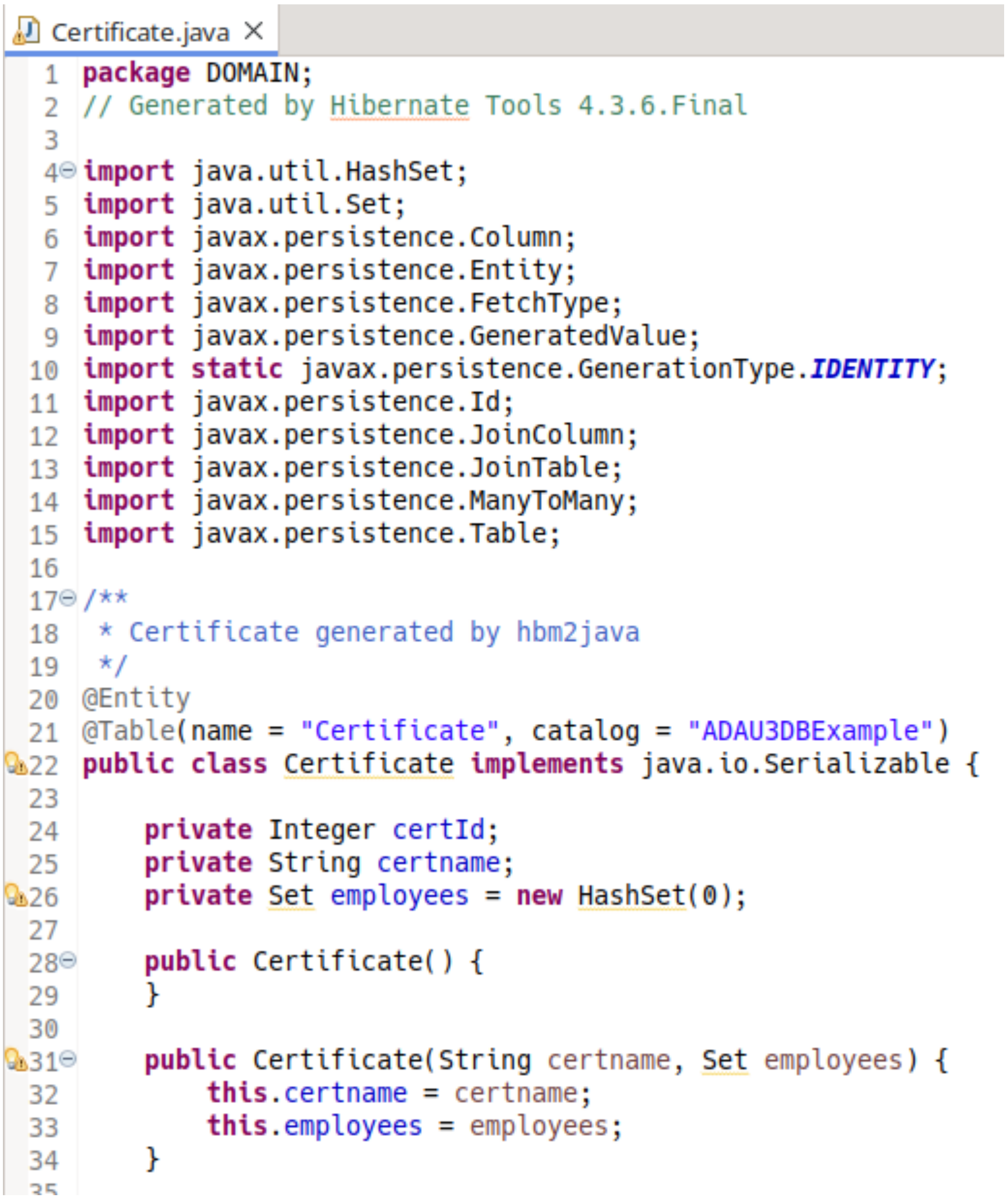
5. Persistent classes

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.

5.1. 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

```
package DOMAIN;
// Generated by Hibernate Tools 4.3.6.Final

import java.util.HashSet;
import java.util.Set;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import static javax.persistence.GenerationType.IDENTITY;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

/**
 * Certificate generated by hbm2java
 */
@Entity
@Table(name = "Certificate", catalog = "ADAU3DBExample")
public class Certificate implements java.io.Serializable {

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

    public Certificate() {
    }

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

Now we must do the following things:

- **Change** all references from **javax.persistence to jakarta.persistence**.

- Add the version code. Just click on the yellow triangle and add default serial version ID.
- Specify the type of HashSet and add it.
- 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.

5.1.1. Class Certificate

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

```java
package DOMAIN;
// Generated by Hibernate Tools 4.3.6.Final

import java.util.HashSet;
import java.util.Set;
import jakarta.persistence.*;
import static jakarta.persistence.GenerationType.IDENTITY;

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

@Entity
@Table(name = "Certificate", catalog = "ADAU3DBExample")
public class Certificate implements java.io.Serializable {

    /*
     * ------------------------------------------
     * ANNOTATIONS. ATTRIBUTES
     * ------------------------------------------
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "certID", unique = true, nullable = false)
    private Integer certId;

    @Column(name = "certname", length = 30)
    private String certname;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "EmpCert", catalog = "ADAU3DBExample", joinColumns = {
            @JoinColumn(name = "certificateID", nullable = false, updatable = false) },
inverseJoinColumns = {
                    @JoinColumn(name = "employeeID", nullable = false, updatable = false) })
    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 Integer getCertId() {
        return this.certId;
    }

    public void setCertId(Integer 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;
    }

}
```

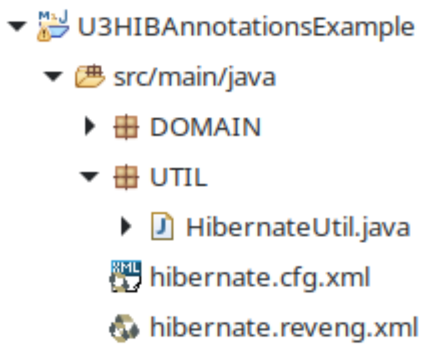
5.1.2. Class Employee

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

```java
package DOMAIN;
// Generated by Hibernate Tools 4.3.6.Final

import java.util.HashSet;
import java.util.Set;
import jakarta.persistence.*;
import static jakarta.persistence.GenerationType.IDENTITY;

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

@Entity
@Table(name = "Employee", catalog = "ADAU3DBExample")
public class Employee implements java.io.Serializable {

    /*
     * ------------------------------------------
     * ANNOTATIONS. ATTRIBUTES
     * ------------------------------------------
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "empID", unique = true, nullable = false)
    private Integer empId;

    @Column(name = "firstname", length = 20)
    private String firstname;

    @Column(name = "lastname", length = 20)
    private String lastname;
    @Column(name = "salary", precision = 22, scale = 0)
    private Double salary;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "EmpCert", catalog = "ADAU3DBExample", joinColumns = {
            @JoinColumn(name = "employeeID", nullable = false, updatable = false) },
inverseJoinColumns = {
                    @JoinColumn(name = "certificateID", nullable = false, updatable = false) })
    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<Certificate>
certificates) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.salary = salary;
        this.certificates = certificates;
    }

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

    public void setEmpId(Integer 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;
    }

}
```

6. Sessions and objects. Class HibernateUtil

In our example we create the HibernateUtil.java class using the singleton pattern for this purpose, as we did with XML mapping:

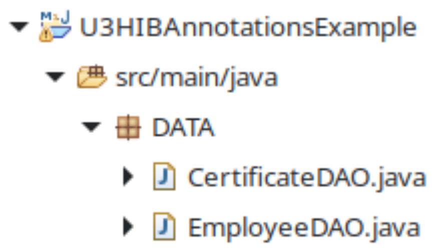


```java
package UTIL;

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;
    }

}
```

7. 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).

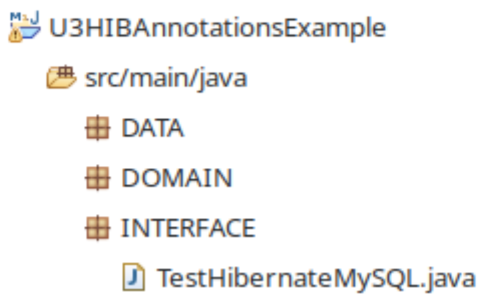
In our case **we only need to copy the DATA package from the previous project with XML mapping**. We use the same code, since we have only changed the XML mapping to annotation mapping.



```
U3HIBAnnotationsExample
  src/main/java
    DATA
      CertificateDAO.java
      EmployeeDAO.java
```

8. Hibernate. Interface layer

Finally we will create the class structure of the application. In our case **we only need to copy the INTERFACE package from the previous project with XML mapping**. We use the same code, since we have only changed the XML mapping to annotation mapping.



```
U3HIBAnnotationsExample
    src/main/java
        DATA
        DOMAIN
        INTERFACE
            TestHibernateMySQL.java
```

9. 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/
- Hibernate Annotations. https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/
- Hibernate Annotations Reference Guide. https://docs.jboss.org/ejb3/app-server/HibernateAnnotations/reference/en/html_single/index.html
- Tutorialspoint. Hibernate - Annotations. https://www.tutorialspoint.com/hibernate/hibernate_annotations.htm
- Digital Ocean. JPA Annotations - Hibernate Annotations. https://www.digitalocean.com/community/tutorials/jpa-hibernate-annotations
- JBoss Tools. Eclipse Plugins for JBoss Technology. https://tools.jboss.org/
- Java Hibernate Reverse Engineering Tutorial with Eclipse and MySQL. https://www.codejava.net/frameworks/hibernate/java-hibernate-reverse-engineering-tutorial-with-eclipse-and-mysql
- 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/