Hibernate tutorial (annotations)

Rui Couto António Nestor Ribeiro

April 3, 2018
March 31, 2020 (updated)

Contents

1 Introduction

In the previous tutorial the Hibernate framework was already introduced. In that tutorial a model driven approach was used and the automatic mappings were created. That allowed us to explore hibernate as well as to understand how the mapping works.

In this session the manual creation of persistence will be addressed, and all the code will be created from scratch using an IDE. In the previous tutorial the Hibernate XML approach was addressed. However, Hibernate supports also direct annotation of the classes, which will be presented in this document.

The tutorial starts with the configuration process and then it uses the example of a games' library system to illustrate the process. The usage of an IDE is optional, however the configuration is demonstrated resorting to NetBeans, which makes the process, in our opinion, considerably easier. To use it with other IDEs one can follow the tutorial but some adjustments must be made.

2 Add the required libraries

NOTICE: this section must be updated. Netbeans 11 no longer provides by default the Hibernate package, so it must be manually installed. The following list of libraries and installation screens should be used just for reference while installing the package in Netbeans or in another IDE.

2.1 Hibernate

Add the Hibernate JPA `4.3.X` libraries to your project. The following libraries are required:

- `Hibernate 4.3.X (JPA 2.1) - antlr-2.7.7`
- `Hibernate 4.3.X (JPA 2.1) - c3p0-0.9.2.1`
- `Hibernate 4.3.X (JPA 2.1) - hibernate c3p0-4.3.1`
- `Hibernate 4.3.X (JPA 2.1) - mchange-commons-java-0.2.3.4`
- `Hibernate 4.3.X (JPA 2.1) - dom4j-1.6.1`
- `Hibernate 4.3.X (JPA 2.1) - ehcache-core-2.4.3`
- `Hibernate 4.3.X (JPA 2.1) - hibernate-ehcache-4.3.1`
- `Hibernate 4.3.X (JPA 2.1) - hibernate-core-4.3.1`

- `Hibernate 4.3.X (JPA 2.1) - jboss-logging-3.1.3`
- `Hibernate 4.3.X (JPA 2.1) - hibernate-commons-annotations-4.0.4`
- `Hibernate 4.3.X (JPA 2.1) - hibernate-entitymanager-4.3.1`
- `Hibernate 4.3.X (JPA 2.1) - javassist-3.18.1`
- `Hibernate 4.3.X (JPA 2.1) - jboss-transaction-api_1.2_spec-1.0.0`
- `Hibernate 4.3.X (JPA 2.1) - slf4j-api-1.6.1`
- `Hibernate 4.3.X (JPA 2.1) - slf4j-simple-1.6.1`
- `Hibernate 4.3.X (JPA 2.1) - hibernate-jpa-2.1-api-0.1`

In NetBeans one can follow through the next steps, in order to include the libraries above depicted:

Figure 1: Open the project `Properties`.

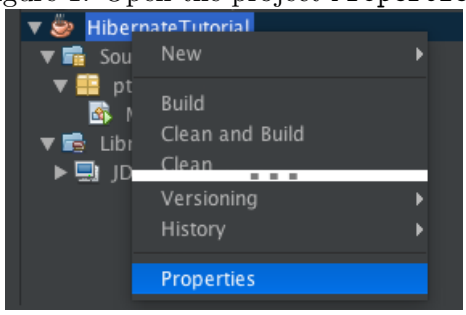


Figure 2: Select `Libraries` in the left panel.

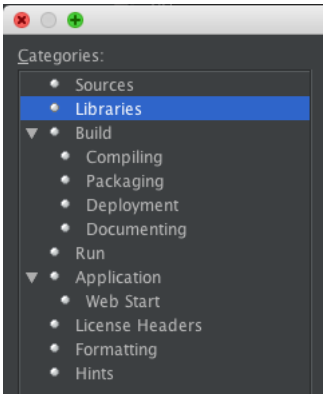


Figure 3: Select `Add Library...` on the right options.

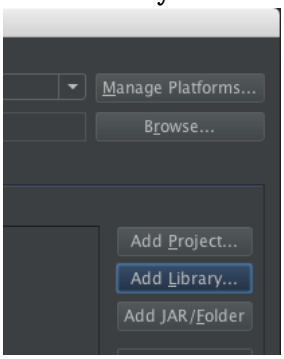
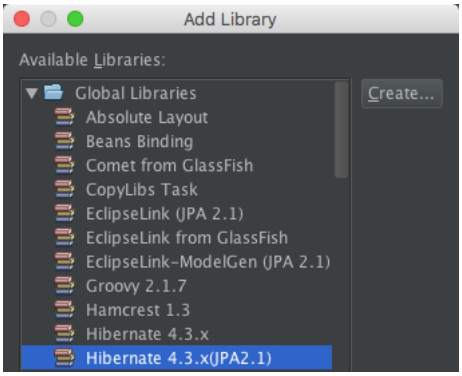


Figure 4: Choose `Hibernate 4.3.x(JPA2.1)`.



2.2 Database

Add the corresponding database libraries to your project. For example to use mysql one must add `mysql-connector-java-5.1.24-bin.jar` to the project dependencies.

In NetBeans select the project `Properties`, and then `Libraries` (as in the previous step). Then:

Figure 5: Select `Add JAR/Folder`.

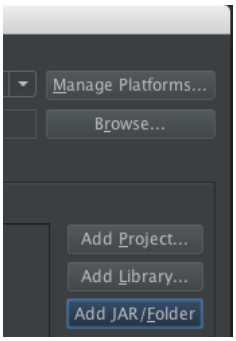
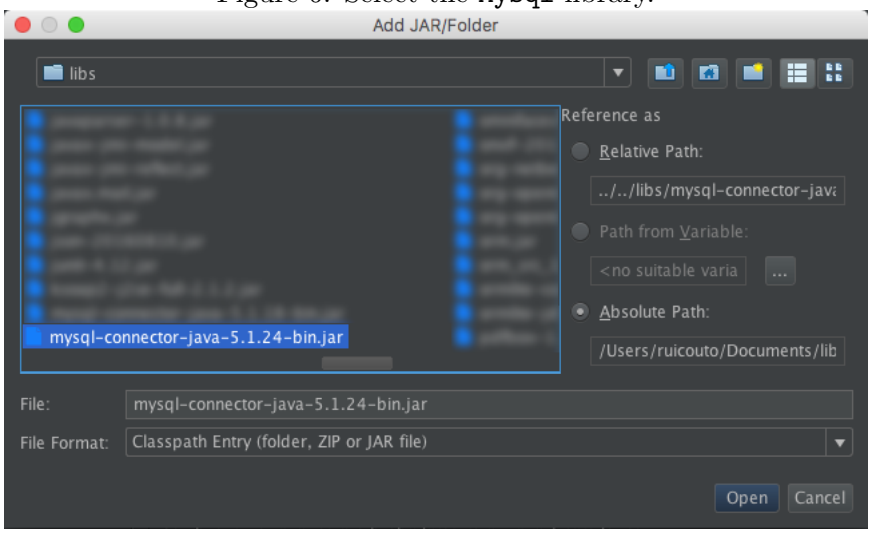


Figure 6: Select the `Mysql` library.



Tasks

1. Create a new Java project.
2. Add the required dependencies (i.e. libraries).

3 Hibernate configuration file

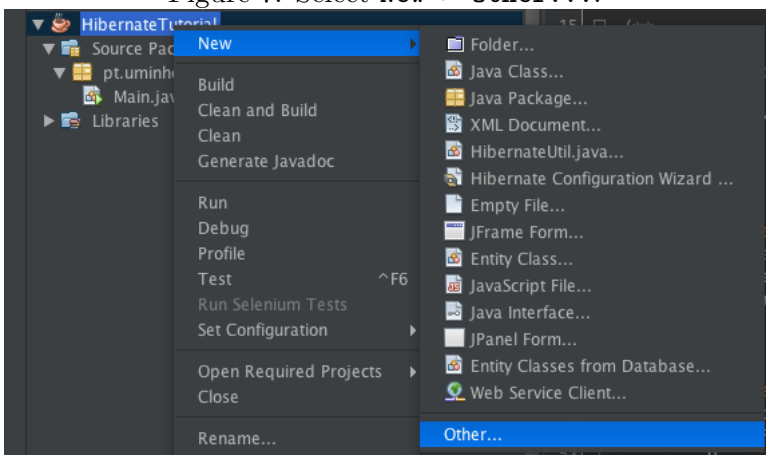
Create a hibernate configuration file, `hibernate.cfg.xml`, in your project. This file contains the hibernate properties such as mapped entities and connection settings. A minimal example is:

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://server:port/database?
                                          zeroDateTimeBehavior=convertToNull</property>
    <property name="hibernate.connection.username">uname</property>
    <property name="hibernate.connection.password">password</property>
  </session-factory>
</hibernate-configuration>
```

Change the fields `server`, `port`, `database`, `uname` and `password`, according to your configuration.

In NetBeans the file can be added as follows.

Figure 7: Select `New > Other...`.



Tasks

1. Setup the database connection.

Figure 8: Select `Hibernate` in the categories, and `Hibernate Configuration Wizard` in the `File Types`.

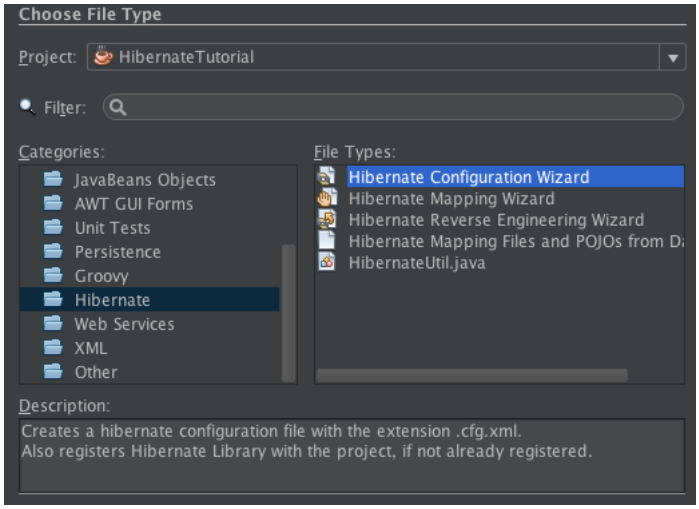


Figure 9: In the next step, select `New Database Connection...` in the dropdown menu.

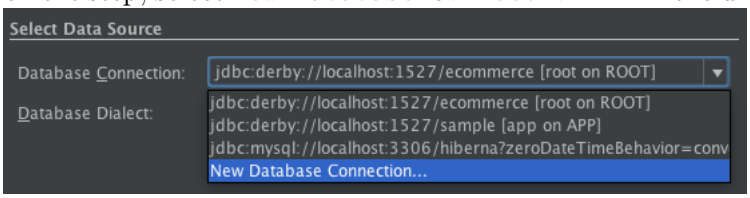


Figure 10: Choose `MySQL` (Connector/J driver).

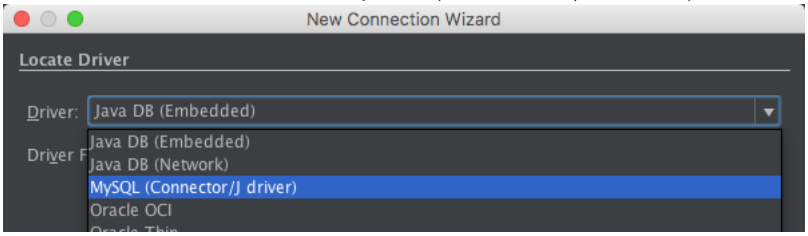


Figure 11: Enter your database connection settings.

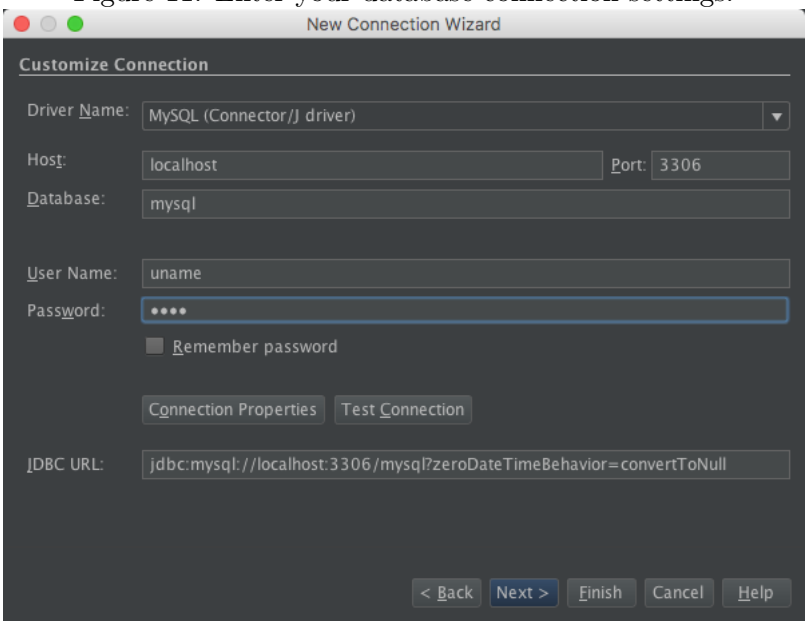
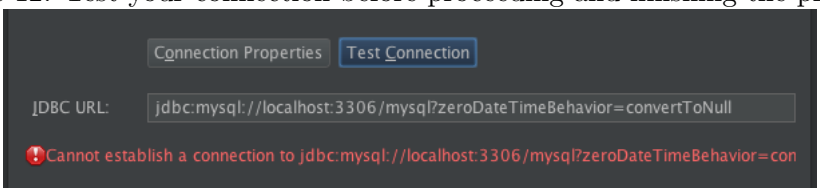


Figure 12: Test your connection before proceeding and finishing the process.



4 Enable automatic creation of the database schema

Hibernate can automatically generate the database schema for the properties. In order to do such generation one must add the following property to the `hibernate.cfg.xml` :

```
1 <property name="hibernate.hbm2ddl.auto">update</property>
```

In the case that you want to recreate the database schema, it's necessary to change the property to `create` . Note however that this will recreate the schema every time a connection is established.

```
1 <property name="hibernate.hbm2ddl.auto">create</property>
```

Tasks

1. Configure Hibernate to automatically generate the database.

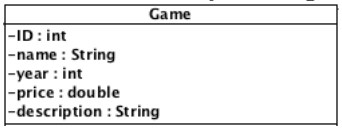
5 Create and annotate the classes

This section addresses the required steps to persist an entity. The effort is mainly performed through annotation in the source code of the entithy definition..

5.1 Game Management System

As a practical example, we will create a system to store games information. Specifically, the already presented *Game Management System*. In this session, all the classes and configurations will be created manually.

Figure 13: Games' Library - the game entity.



5.2 Creating the classes

All the entities to persist **must** have:

- An empty constructor;
- A getter and a setter for each property (following the standard convention);

As an example, consider the `Game` class shown next.

```
1  public class Game {
2      private int id;
3      private String name;
4      private int year;
5      private double price;
6      private String description;
7
8      public Game() {
9      }
10
11     public int getId() {
12         return id;
13     }
14
15     public void setId(int id) {
16         this.id = id;
17     }
18
19     // remaining getters and setters
20 }
```

5.3 Annotating the classes

Each class to persist should be annotated with `@Entity` . That information gives Hibernate the indication how to process this class.

```
1 @Entity
2 public class Game {
```

Also, each `Entity` needs to have an `ID`, corresponding to an integer attribute. The attribute should be annotated with **@Id**. In this case, we want the `ID` to be automatically set by the database (i.e. *autoincrement*) , so we need to add: `@GeneratedValue(strategy=GenerationType.AUTO)` .

```
1     @Id
2     @GeneratedValue(strategy=GenerationType.AUTO)
3     private int id;
```

It is also possible to configure the mapping process. The `@Table` annotation provides the possibility to specify, for instance, the name of the table to persist the entity.

5.4 Updating the mapping configuration

After annotating the entities, the information regarding the entities to persist should be specified in the hibernate configuration file. So, each entity should be declared in `hibernate.cfg.xml` , as:

```
1 <mapping class="pt.uminho.di.aa.Game"/>
```

This entry should be stated in the `<session-factory>` section. Note that it is needed to include the information about the package and the class name (ie., the qualified name of the class).

Tasks

1. Create the `Game` †Java class.
2. Annotate it properly so that it can be persisted.

6 Run the code

After the configuration is done, it is now possible to persist and load entities. Several steps need to be carried out to configure hibernate in Java and to open a session in order to perform some actions.

1. Create a configuration - this loads the `hibernate.cfg.xml` configurations.
2. Create a session factory - this enables the possibility to create sessions.
3. Create and open a session - the sessions allows to perform the persistency operations.
4. Begin a transaction - the persistency operations should be performed inside a transaction.
5. Perform the persistency actions - such as save or load.
6. Commit or rollback the transaction - once the commit is performed, the database is updated.
7. Close the session - this ends the process.

```
public static void main(String[] args) throws Exception {
  try {
    //1 - Configuration
    Configuration configuration = new Configuration().configure();
    StandardServiceRegistry sr = new StandardServiceRegistryBuilder()
              .applySettings(configuration.getProperties()).build();
    //2 - SessionFactory
    SessionFactory sf = configuration.buildSessionFactory(sr);
    //3 - Session
    Session s = sf.openSession();
    s.setFlushMode(FlushMode.COMMIT); //propagate changes on commit
    //4 - start the transaction
    Transaction t = s.beginTransaction();

     //Create a new object
    Game g = new Game();
    g.setName("GTA V");
    g.setPrice(60.0);
    //g.set...
```

```
20
21      //5 - save the object
22      s.save(g);
23
24      try {
25        //6 - commit the transaction
26        t.commit();
27      } catch (Exception e) {
28        //6 - rollback in case of exception
29        t.rollback();
30        e.printStackTrace();
31        System.out.println("Unable to commit changes");
32      }
33
34      //7 - Close the session and end process
35      s.close();
36      StandardServiceRegistryBuilder.destroy(sr);
37    } catch (Exception e) {
38      e.printStackTrace();
39      System.out.println("Unable to connect to hibernate");
40    }
41  }
```

The process to read requires the same configurations, but the method used in the session is the `get(Class, id)` .

```
1  Game g = (Game) s.get(Game.class, 1);
2  System.out.println(g.getName());
```

The configuration steps should be abstracted in another method, and the session can be reused if required.

Tasks

1. Create a new `Game` (by code) and save it in the database.

7 Performing Hibernate queries (HQL)

The HQL queries are executed through the session object, as follows.

```
1  Query query = s.createQuery("FROM Game where id>2");
2  List results = query.list();
3  System.out.println("Number of entries: " + results.size());
4  Game lg = (Game) results.get(0);
5  System.out.println(lg.getDescription());
```

See the HQL reference manual for further examples and documentation at: `https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html`.

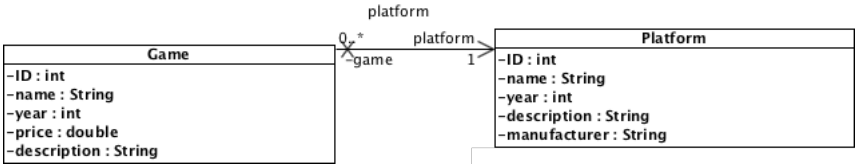
Tasks

1. Create the code to load the previously saved user.
2. Modify the user, persist it and reload in order to see the changes.

8 Object reference

Having the `Game` entity defined, one can next defined the `Platform` entity. Each `Game` references one `Platform`, which can be referenced by several games.

Figure 14: Games' Library - the game platform.



8.1 Creating a new entity

To represent the `Platform` a new entity must be created. Following the same process as we did for `Game` the class is defined as follows.

```
@Entity
public class Platform {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String name;
    private int year;
    private String description;
    private String manufacturer;

    //...
}
```

8.2 Adding a reference from class Game

Since `Platform` is a reference, it must be specified in the `Game` class with an annotation. In this case, with `@OneToOne`, stating that one `Game` contains one `Platform` instance.

```
@OneToOne
private Platform platform;
```

8.3 Adding mapping information

It is essential not to forget to update the mapping information, by declaring this new entity.

```
<mapping class="pt.uminho.di.aa.Platform"/>
```

8.4 Persisting information

Now that the mapping is specified, it is still not possible to save the information. The *child* entities should be saved first and then the *parent* entities can then be saved.

```
Platform p = new Platform();
p.setName("PS4");
//p.set...
s.save(p);

Game g = new Game();
g.setTitle("GTA V");
//g.set...
s.save(g);
```

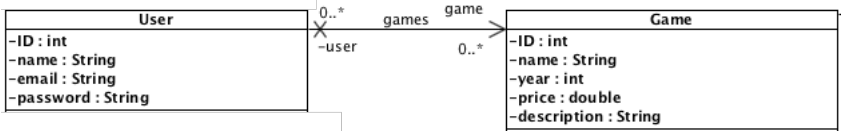
Tasks

1. Implement the `Platform` class.
2. Add the appropriate annotations.
3. Test the developed code.

9 Collections

Hibernate supports also persisting collections information. Consider for instance the entity `User` which owns a list of games.

Figure 15: Games' Library - the user.



9.1 The user entity

The user is an entity (similar to the previous one). The attribute which relates it with `Game` , `games` , is a collection. In Hibernate, collections are annotated with the annotation `@OneToMany` , stating that one `User` has many `Game` instances.

```java
@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String email;
    private String username;
    private String password;

    @OneToMany
    private List<Game> games;
    //...
```

9.2 Adding mapping information

Once again, the mapping information should be updated, declaring the `User` entity.

```xml
<mapping class="pt.uminho.di.aa.User"/>
```

9.3 Persisting information

The process of persisting information is similar to the previously presented ones.

```java
Platform p = new Platform();
p.setName("PS4");
//p.set...
s.save(p);

List<Game> games = new ArrayList<>();

Game g = new Game();
g.setName("GTA V");
//g.set...
g.setPlatform(p);
s.save(g);
games.add(g);

Game g2 = new Game();
g2.setName("Gran Turismo Sport");
//g2.set...
g2.setPlatform(p);
s.save(g2);
games.add(g2);

User u = new User();
u.setUsername("email");
//u.set...
```

```
25 u.setGames(games);
26 s.save(u);
```

9.4 Querying several tables

HQL supports querying several tables at once.

```
1 Query query = s.createQuery("from Game g, Platform p where g.platform = p and p.year = 2017");
2 List results = query.list();
3 //index 0 - Game; index 1 - Platform
4 Object[] os = (Object[]) results.get(0);
5 //cast
6 Game g = (Game) os[0];
7 System.out.println(g.getTitle());
```

Tasks

1. Implement the `User` class.
2. Add the appropriate annotations.
3. Test the developed code.

10 Tasks

1. Implement the persistence for the remaining classes. Remember the diagram from the previous tutorial (see Figure 16).
2. Implement the `GMS` façade. Remember the features proposed in the previous tutorial:
   (a) Register a user;
   (b) Register a game;
   (c) Register a platform;
   (d) List user games;
   (e) List all games;
   (f) Search a game;
   (g) Delete a game.

Figure 16: GMS proposed architecture.