**Java Hibernate JPA Annotations**

**1. Overview of JPA and Hibernate**

Let’s understand some fundamentals about JPA and Hibernate before writing code.

**Java Persistence API (JPA):**

JPA is a Java API specification for relational data management in applications using Java SE and Java EE. JPA defines a standard way for simplifying database programming for developers, reduce development time and increase productivity.

When using JPA, you have to import interfaces and classes from the package javax.persistence.

JPA defines Java Persistence Query Language (JPQL) which is an object-oriented query language. The syntax of JPQL is similar to SQL but it operates against Java objects rather than directly with database tables.

Remember JPA is a specification, and Hibernate is one of its implementations, among others such as EclipseLink and OpenJPA.

**Hibernate Framework:**

Hibernate is a popular Object Relational Mapping (ORM) framework that aims at simplifying database programming for developers.

Hibernate was developed before JPA. And after JPA becomes a standard, Hibernate restructures itself to become an implementation of JPA.

The Hibernate framework consists of several components: Hibernate ORM, Hibernate Search, Hibernate Validator, Hibernate CGM and Hibernate Tools.

**2. Create MySQL Database**

Use the following statement to create a database named **usersdb**using MySQL Workbench or MySQL Command Line Client:

|  |  |
| --- | --- |
| 1 | create database usersdb; |

Then create a table name **users** with 4 columns: user\_id, fullname, email and password, using the following script:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE TABLE `users` (    `user\_id` int(11) NOT NULL AUTO\_INCREMENT,    `fullname` varchar(45) NOT NULL,     `email` varchar(45) NOT NULL,    `password` varchar(45) NOT NULL,    PRIMARY KEY (`user\_id`)  ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |

Using **desc users** command in MySQL Command Line Client, the structure of the table looks like this:



Note that the column user\_id is the table’s primary key and it is auto-increment.

**3. Setup Java Maven Project in Eclipse**

In Eclipse IDE, click **File > New > Project…** and select **Maven > Maven Project** in the *New Project* dialog. Then click **Next**.

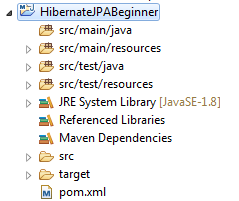
In the next screen, check the option ‘*Create a simple project (skip archetype selection)*’, and then click **Next**.

In the *New Maven Project* screen, enter the project’s information as follows:

- Group Id: net.codejava.hibernate

- Artifact Id: HibernateJPABeginner

Leave other things as they are and click **Finish**. In the *Project Explorer* view, you see the project gets created with the following structure:

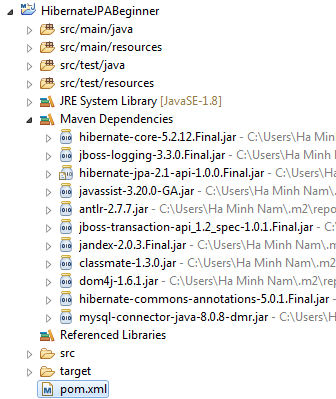


**Configure Maven Dependencies:**

Next, we need to add dependencies in Maven’s Project Object Model (pom.xml) for Hibernate, JPA and MySQL Connector Java. Open the pom.xml file in XML mode and insert the following XML just before the </project> tag:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <dependencies>      <dependency>          <groupId>org.hibernate</groupId>          <artifactId>hibernate-core</artifactId>          <version>5.2.12.Final</version>      </dependency>      <dependency>          <groupId>mysql</groupId>          <artifactId>mysql-connector-java</artifactId>          <version>8.0.8-dmr</version>      </dependency>  </dependencies> |

You see, here we add two dependencies for the project: hibernate-core and mysql-connector-java. Maven automatically downloads the required JAR files which are shown under the *Maven Dependencies* node in the project:



You see, we just specify the dependency hibernate-core, but Maven can analyze and download all the dependencies of hibernate-core as well. That’s really helpful, right?

Create a new Java package name net.codejava.hibernate under the src/main/java folder. We’ll put our Java classes in this package.

**4. Code Model Class**

Next, let’s create a domain model class named User. Then we will use JPA annotations to map this table to the corresponding table in the database.

Here’s the initial code of the User class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | package net.codejava.hibernate;    /\*\*   \* User.java   \* Copyright by CodeJava.net   \*/  public class User {      private Integer id;      private String fullname;      private String email;      private String password;        public Integer getId() {          return id;      }        public void setId(Integer id) {          this.id = id;      }        public String getFullname() {          return fullname;      }        public void setFullname(String fullname) {          this.fullname = fullname;      }        public String getEmail() {          return email;      }        public void setEmail(String email) {          this.email = email;      }        public String getPassword() {          return password;      }        public void setPassword(String password) {          this.password = password;      }  } |

You see, this is just a POJO (Plain Old Java Object) class with some instance fields and its getter and setter methods. Now, let’s use some annotations provided by JPA to map this model class to the users table in the database.

**@Entity**

This annotation indicates that the class is mapped to a database table. By default, the ORM framework understands that the class name is as same as the table name. The @Entity annotation must be placed before the class definition:

|  |  |
| --- | --- |
| 1  2  3  4 | @Entity  public class User {  …  } |

**@Table**

This annotation is used if the class name is different than the database table name, and it is must placed before the class definition. Since the class name is User and the table name is Users, we have to use this annotation:

|  |  |
| --- | --- |
| 1  2  3 | @Entity  @Table(name = "USERS")  public class User { |

**@Column**

This annotation is used to map an instance field of the class to a column in the database table, and it is must placed before the getter method of the field. By default, Hibernate can implicitly infer the mapping based on field name and field type of the class. But if the field name and the corresponding column name are different, we have to use this annotation explicitly. This

In our model class, the field name id is different than the column user\_id, so we have to use the @Column annotation as follows:

|  |  |
| --- | --- |
| 1  2  3  4 | @Column(name = "USER\_ID")  public Integer getId() {      return id;  } |

The other fields (fullname, email and password) have identical names as the corresponding columns in the table so we don’t have to annotate those fields.

**@Id**

This annotation specifies that a field is mapped to a primary key column in the table. Since the column user\_id is a primary key, we have to use this annotation as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Column(name = "USER\_ID")  @Id  public Integer getId() {      return id;  } |

**@GeneratedValue**

If the values of the primary column are auto-increment, we need to use this annotation to tell Hibernate knows, along with one of the following strategy types: AUTO, IDENTITY, SEQUENCE, and TABLE. In our case, we use the strategy IDENTITY which specifies that the generated values are unique at table level, whereas the strategy AUTO implies that the generated values are unique at database level.

Therefore, the getter method of the field id is annotated as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Column(name = "USER\_ID")  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  public Integer getId() {      return id;  } |

Finally, we have the model class User is annotated as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.cg.hibernateexample.entity;  import javax.persistence.\*;    @Entity  @Table(name = "USERS")  public class User {      private Integer id;      private String fullname;      private String email;      private String password;        @Column(name = "USER\_ID")      @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      public Integer getId() {          return id;      }        // other setters and getters are not shown for brevity  } |

**5. Specify database connection properties**

If you use hibernate.cfg.xml file, specify connection properties for Oracle database as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <?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>  <property name="connection.driver\_class">oracle.jdbc.OracleDriver</property>  <property name="connection.url">jdbc:oracle:thin:@localhost:1521:dbname</property>  <property name="connection.username">user</property>  <property name="connection.password">pass</property>  <property name="dialect">org.hibernate.dialect.Oracle8iDialect</property>  <property name="show\_sql">true</property>  <mapping class="package.ModelClass1" />  <mapping class="package.ModelClass2" />  <mapping class="package.ModelClass3" />  </session-factory>  </hibernate-configuration> |

Remember that the hibernate.cfg.xml file must be put in the src/main/resources folder of your project.

* . <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>: This property helps hibernate to generate database-specific SQL statements. This is an optional property. According to hibernate documentation, hibernate will be able to choose the correct implementation of dialect automatically using the JDBC metadata returned by the JDBC driver.
* <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>: Using this property, we can provide the **Fully Qualified Name** (**FQN**) of the java driver name for a particular database. The driver class is implemented using Java and resides in the JAR file and contains the driver that should be placed in our classpath.
* <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/kode12</property>: Using this property, we can provide the physical location of the database; however, the connection URL may vary from database to database. Here, we will use the MySQL database, so the URL shows jdbc:MySQL://<host/computer-name/ip>:<port>/<database name to connect>.
* <property name="hibernate.connection.username">root</property>: Using this property, we can provide the username to access a particular database.
* <property name="hibernate.connection.password">root</property>: Using this property, we can provide the password to access a particular database.
* <property name="show\_sql">true</property>: The possible value for this property is either true or false. This is an optional property. Hibernate logs all the generated queries that reach the database to the console if the value of show\_sql is set to true. This is useful during basic troubleshooting. Hibernate will use the prepared statement so that it does not display the parameter in the output window. If you want to see this parameter as well, you will have to enable the detailed log. Log4j is preferred for the detailed log.
* <property name="hbm2ddl.auto">create</property>: The possible values are validate, update, create or create-drop. This is also an optional property. Here, we will set value=create so that it will remove all the schemas and create a new one using the hibernate mapping on each build of sessionfactory. For value=update, hibernate will update the new changes in the database.

Note

Do not use the hbm2ddl.auto property in the production environment because it may remove all of the data and schema. So, it's best practice to avoid it in the production environment.

* <mapping resource="Employee.hbm.xml"/>: All of the mapping file is declared in the mapping tag, and the mapping file is always named xx.hbm.xml. We can use multiple mapping tags for multiple mapping files.

Here is an example:

<mapping resource="Employee.hbm.xml"/>

<mapping resource="Department.hbm.xml"/>Copy

There's more…

Here are some useful properties used in hibernate:

* hibernate.format\_sql:
  + The possible values are true and false
  + It shows the hibernate-generated queries in the pretty format if set as true
* hibernate.connection.pool\_size:
  + The possible value is always greater than 1 (value >= 1)
  + It limits the maximum number of pooled connections
* hibernate.connection.autocommit:
  + The possible values are true and false
  + It sets the autocommit mode for JDBC

**6. Hibernate Example Program**

For your reference, the following example program uses Hibernate to persist a Customer object to the Oracle database:

|  |  |
| --- | --- |
|  |  |

package com.cg.hibernateexample;

 import org.hibernate.\*;

 import org.hibernate.boot.\*;

 public class HibernateOracleTestXML {

    public static void main(String[] args) {

        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()

                .configure() // configures settings from hibernate.cfg.xml

                .build();

        try {

            SessionFactory factory = new MetadataSources(registry).buildMetadata().buildSessionFactory();

            Session session = factory.openSession();

            Transaction transaction = session.beginTransaction();

            Customer customer = new Customer("Alexander", "alexander@gmail.com");

            session.save(customer);

            transaction.commit();

            session.close();

            factory.close();

        } catch (Exception ex) {

            StandardServiceRegistryBuilder.destroy(registry);

        }

    }

 }

**Sessionfactory**:

1. It is one instance per datasource/database.
2. It is thread-safe.
3. It is a heavyweight object, because it maintains data sources, mappings, hibernate configuration information etc.
4. Sessionfactory will create and manage the sessions.
5. If you have say, 4 datasources/databases, then you must create 4 session factory instances.
6. sessionfactory is an immutable object and it will be created as a singleton while the server initializes itself.

**Session**:

1. It is one instance per client/thread/one transaction.
2. It is not thread-safe.
3. It is lightweight.
4. sessions will be opened using sessionfactory.openSession() and some database operations will be done finally session will be closed using session.close().

**Transaction Interface in Hibernate:**

In hibernate framework, we have Transaction interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

A transaction is associated with Session and instantiated by calling session.beginTransaction().