

Tutorial/Mini-Lab

Hibernate

Learn how to use Hibernate

In this tutorial you will learn how to annotate a file with Hibernate annotations. Then you will use those annotations to create, read, update, and delete a Java object from your MySQL database. Finally, you will create your own Java Class and practice what you learned.

1. Create a new Maven project and add the following dependencies
 - a. <https://mvnrepository.com/artifact/org.hibernate/hibernate-core> (Latest Final)
 - b. <https://mvnrepository.com/artifact/mysql/mysql-connector-java> (Latest)
 - c. <https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.6.0>
2. Download [Book.java](#) and place it in your main>java folder
3. Open the Book.java class.
4. In this example we will be annotating this Book object so that it can be saved inside our database.

```
public class Book {  
    private long bookId;  
    private String title;  
    private String author;  
    private float price;
```

5. First we need to make our Java object a persistent class by adding the following (you can do this by right clicking and selecting **generate**)
 - a. A default constructor
 - b. A getter and setter for each private attribute

- c. Persistent classes also need an attribute id (a field that will be unique for each object in our database) in our case this is our **bookId**, which already exists.
- 6. Next we will annotate our file so that Hibernate knows how to save our object in the database.
- 7. The first step is to annotate our Book class with `@Entity` and `@Table`
 - a. `@Entity`
 - b. `@Table(name="book")`

```
@Entity
@Table(name="book")
public class Book {
    private long bookId;
    private String title;
    private String author;
    private float price;
```

- 8. This will tell hibernate that we want to save Book objects inside a table called "book" inside our database.

```
@Entity
@Table(name="book")
public class Bo
    private lon
    private Str
    private String author;
```

Cannot resolve table 'book'

Assign Data Sources Alt+Shift+Enter More actions... Alt+Enter

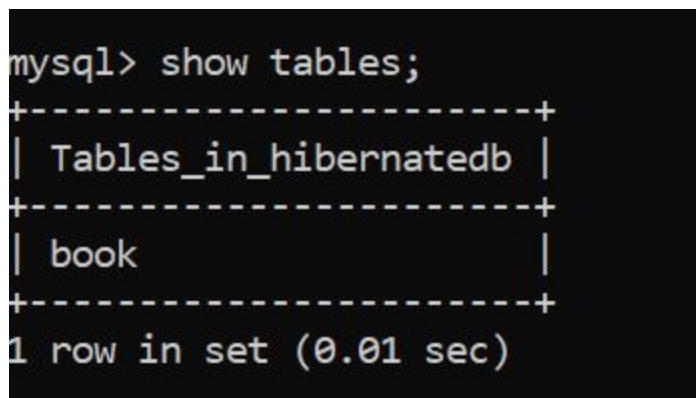
This error is caused by IDEA. It means that IntelliJ isn't configured for tables names validation. It will still compile and work, but the IntelliJ code completion will say there is an error.

- 9. Now we will create a table in our database called "book"
 - a. Connect to your MySql Database (you should know how to do this from the pre-class assignment)

- b. Run the following SQL script, it will create a new table in your database for storing our Book objects. If you need a refresher on SQL, [here](#) is a helpful resource.

```
USE `hibernateDB`;  
  
CREATE TABLE `book` (  
  `book_id` int(11) NOT NULL AUTO_INCREMENT,  
  `title` varchar(128) NOT NULL,  
  `author` varchar(45) NOT NULL,  
  `price` float NOT NULL,  
  PRIMARY KEY (`book_id`),  
  UNIQUE KEY `book_id_UNIQUE` (`book_id`)  
);
```

10. Type **show tables;** and check that your output looks like this



```
mysql> show tables;  
+-----+  
| Tables_in_hibernatedb |  
+-----+  
| book                   |  
+-----+  
1 row in set (0.01 sec)
```

11. Now we need to create annotations above the properties in our file

- a. Add the following annotations above your **getBookId()** method
- @Id** - marks the attribute as an ID
 - @Column(name="book_id")** - says what column to store the data in

- iii. `@GeneratedValue(strategy = GenerationType.IDENTITY)` - says how we will want the IDs for the Book objects to be generated

```
@Id
@Column(name="book_id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
public long getBookId() { return bookId; }
```

- b. Annotate the **getTitle** method with `@Column(name="title")`
 - c. Annotate the **getAuthor** method with `@Column(name="author")`
 - d. Annotate the **getPrice** method with `@Column(name="price")`
 - e. Note that these Column names come from our SQL script in step 9.
 - f. (Optional) To learn about all the different Hibernate annotations click [here](#).
12. Now we will need to create a hibernate configuration file so that Hibernate can connect to our database and map our Book objects.
13. Download the provided [hibernate.cfg.xml](#)
14. Put the hibernate.cfg.xml file in main>resources



15. Now we will open the hibernate.cfg.xml file (also known as the hibernate configuration file) and go through it so you know what it does.

16. The first 4 lines of the document are needed for all hibernate configuration files and more info about them can be found [here](#)

```
<?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">
```

17. Next we have the configuration information for the session-factory. This section has the information for Hibernate to create a SessionFactory to produce Session objects that allow us to connect to our database (this is part of the Hibernate architecture that you read about)

```
5 <hibernate-configuration>
6 <session-factory>
7     <!-- Database connection settings -->
8     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
9     <property name="connection.url">jdbc:mysql://localhost:3306/hibernateDB</property>
10    <property name="connection.username">root</property>
11    <property name="connection.password">password</property>
12    <property name="show_sql">>true</property>
13
14    <!-- Mapping files -->
15    <mapping class="Book" />
16
17 </session-factory>
18 </hibernate-configuration>
```

- a. The first 5 properties are there for information on how to connect to your database. Update the username and password properties to match the information you gave while setting up your MySQL server.
- b. On line 15, there is a reference to our Book class. This tells hibernate that we have annotated our class and want it to be mapped to our database. If we had another class that we wanted to map to our database, we would copy and paste line 15 onto line 16 and replace "Book" with the name of the class.
 - i. Note that if Book.java was in a package that we would need to include package information, meaning "Book" should be written as "com.example.Book"

18. Now that we have created the configuration file, it is time to create our DAO

19. Download [Dao.java](#)

20. There are some methods in this file that are blank that you will be filling with code in this tutorial.

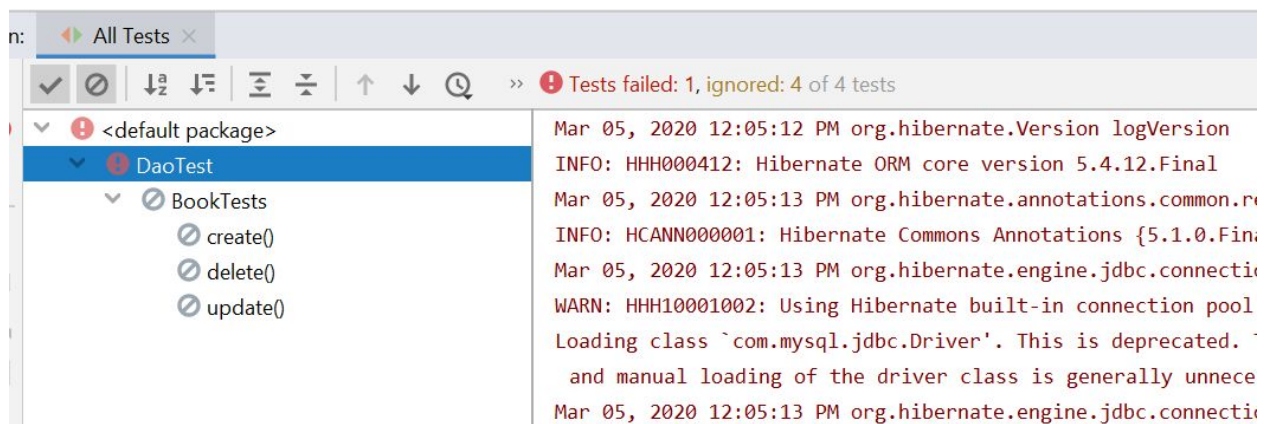
21. We have provided a few methods for you.

a. **setup** will create a sessionFactory from your hibernate configuration file.

b. **exit** will close the Hibernate sessionFactory

22. Download [DaoTest.java](#) and run the tests found in the file.

23. If your configuration files are **wrong OR** your DAO annotation is wrong you should see something like the image below. The read text is the Hibernate console running and can be useful for debugging errors.



Note: Your Test Cases will all Fail because you haven't implemented the DAO yet, But if the test cases are ignored, then you have a problem with your configuration file

24. Now we will implement the **create** and **read** methods

25. All of your methods will have the following code, which is explained below

```
public void create(T object) {  
    Session session = sessionFactory.openSession();  
    Transaction transaction = session.beginTransaction();  
  
    |  
  
    transaction.commit();  
    session.close();  
}
```

- a. These are the steps that will happen for each of your methods
 - i. Create a new hibernate session from the sessionFactory that was created from your hibernate configuration file in the provided **setup** method.
 - ii. Begin a new database transaction
 - iii. (Code that changes depending on the method)
 - iv. Commit the transaction to the database
 - v. Close the hibernate session

26. Implement the **create** method

- a. For the **create** method type the code in the step above into your create method

- b. Call **session.save** and pass in your Book object, as shown below.

```
public void create(T object) {  
    Session session = sessionFactory.openSession();  
    Transaction transaction = session.beginTransaction();  
  
    session.save(object);  
  
    transaction.commit();  
    session.close();  
}
```

- c. Your method is now complete. Hibernate takes care of generating the SQL statement to send to your database by using your annotations that you created in your book file.

27. Next implement the **read** method







- a. Copy the code from the **create** method, but make the modifications so that it looks like the code below.

```
public Type read(Class<Type> type, Id id){  
    Session session = sessionFactory.openSession();  
    Transaction transaction = session.beginTransaction();  
  
    Type object = session.get(type, id);  
  
    transaction.commit();  
    session.close();  
  
    return object;  
}
```

- b. You may be wondering what a `Class<Type>` is. `Type` is the generic type of our Dao class. A `Class<Type>` is the class of object that gets returned when you do

.getClass() on any Java object. A Hibernate Session needs this variable because it has access to any of the classes that we want to map to our database. For example, if we had a “Person” class then a single Hibernate Session could store both a Person object and a Book object into the database. So, a Session object needs to know what table and what object we want it to retrieve from the database.

28. Run **DaoTest.java** and check to see that the create test passes

>  <default package>	367 ms	Hibernate: insert into BOOK (author, price, title) values (?, ?, ?)
>  DaoTest	367 ms	Hibernate: select book0_.book_id as book_id1_0_0_, book0_.author as a
>  BookTests	367 ms	book0_.book_id=?
 create()	316 ms	
 delete()	30 ms	
 update()	21 ms	

29. The black text in the console is the Hibernate console showing the SQL statements that it is running. You can turn off this text by setting the **show_sql** property in **hibernate.cfg.xml** to **false**. But it is useful during development, so right now we will leave it on.

30. Implement the **update** method

- Copy the **create** method and make the modifications so that it looks like the code below.

```
public void update(T object){
    Session session = sessionFactory.openSession();
    Transaction transaction = session.beginTransaction();

    session.update(object);

    transaction.commit();
    session.close();
}
```

- This will update the information for the book object in our database that has the same bookId as book.

31. Run **DaoTest.java** and check to see that the update test passes

<default package>	316 ms	Hibernate: insert into BOOK (author, price, title) values (?, ?, ?)
DaoTest	316 ms	Hibernate: update BOOK set author=?, price=?, title=? where book_id=?
BookTests	316 ms	Hibernate: select book0_.book_id as book_id1_0_0_, book0_.author as au
create()	251 ms	book0_.book_id=?
delete()	22 ms	
update()	43 ms	

32. Let's look in your database by sending the following command to your MySQL Server

a. **select * from `Book`;**

33. Depending on how many times it took you to implement the methods, you should see some entries in your database.

```
mysql> select * from `Book`;
```

book_id	title	author	price
1	Title	Author	0
2	Title	Author	0
3	Title	Author	0
4	Title	Author	0
5	Title	Author	0
6	Updated Title	Author	0

```
6 rows in set (0.01 sec)
```

34. Now, implement the **delete** method

- a. Copy the code from the **create** method and modify it so that it looks like the method below.

```
public void delete(T object){  
    Session session = sessionFactory.openSession();  
    Transaction transaction = session.beginTransaction();  
  
    session.delete(object);  
  
    transaction.commit();  
    session.close();  
}
```

- b. This will cause an object with the bookId of our book object to be removed from the database.

35. Run **DaoTest.java** and check to see that the delete method passes

✓ Tests passed: 3 of 3 tests – 360 ms		
✓ <default package>	360 ms	Hibernate: insert into BOOK (author, price, title) values (?, ?, ?)
✓ DaoTest	360 ms	Hibernate: select book0_.book_id as book_id1_0_0_, book0_.author as book0_.book_id=?
✓ BookTests	360 ms	
✓ create()	270 ms	Hibernate: delete from BOOK where book_id=?
✓ delete()	46 ms	Hibernate: select book0_.book_id as book_id1_0_0_, book0_.author as book0_.book_id=?
✓ update()	44 ms	

36. With Hibernate, we can also send custom SQL to our database which can be useful for clearing tables or querying data. You will learn how to clear a table and how to get all the objects in a table.

37. Now, implement the **clear** method

- a. Copy the method from **create** and make the modifications so that it looks like the code below.

```
public void clear(String tableName){
    Session session = sessionFactory.openSession();
    Transaction transaction = session.beginTransaction();

    Query query = session.createQuery("DELETE FROM " + tableName);
    query.executeUpdate();

    transaction.commit();
    session.close();
}
```

- b. This method is called from the **@AfterAll** method of your test class. It will execute the SQL query **DELETE FROM Book** which will clear the Book table of all its entries after all your tests ran.
- c. Run **DaoTest.java**
- d. Make sure your database was cleared by sending the command **select * from `Book`**; to your MySQL server.

```
mysql> select * from `Book`;
Empty set (0.00 sec)

mysql>
```

Mini-Lab

For this mini-lab, you will annotate a custom Java file and add a new section of Nested tests to verify that the Dao can save your new Class. This should also help solidify the concepts that you have learned in this tutorial.

1. Create your own class that is like the Book class. For example, I created a Person class (you can also create a Person class if you don't want to be creative)
 - a. Make sure it has at least 3 attributes. I did name, age, and personId.
 - b. Make sure one of the attributes is an id. Mine is personId
 - c. Follow the steps in the beginning of this tutorial for annotating your class.

- d. Don't forget to create a new table for your Class in your database. If you need a refresher, you can click [here](#).
- e. Don't forget to add your class to **hibernate.cfg.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>
      <!-- Database connection settings -->
      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
      <property name="connection.url">jdbc:mysql://localhost:3306/hibernateDB</property>
      <property name="connection.username">root</property>
      <property name="connection.password">password</property>
      <property name="show_sql">>true</property>

      <!-- Mapping files -->
      <mapping class="Book"/>
      <mapping class="Person"/>
    </session-factory>
  </hibernate-configuration>
```

- 2. Create a new Nested class for testing your object, mine was called PersonTests, it must test that.
 - a. Create and read works
 - b. Update works
 - c. Delete works

Submission

Notice that you didn't need to create a new Dao class for your new Java Class!
Submit the following files to Canvas (not in a zip!)

- 1. Dao.java
- 2. Your custom java class (mine was Person.java)
- 3. DaoTest.java
- 4. A screenshot of your tests passing

