

Contents

Course Overview	3
Object-relational Mapping.....	4
Databases for Structured Datastorage	4
Programmatic Access.....	5
ODBC for Programmatic Access	6
JDBC for Programmatic Access from Java.....	6
Relational Database vs Java Class.....	7
Object-Relational “Impedance Mismatch”	8
Object Relational Mapping	8
ORM for Programmatic Access from Java.....	9
Several ORM Frameworks.....	10
The Hibernate Framework.....	11
JPA – Java Persistence API	11
Hibernate	11
Reasons for the Popular of Hibernate	13
Why is hibernate easy to use	13
Performance and Scalability	14
Hibernate Query Language (HQL)	14
Hibernate Architechture	15
The JPA Framework.....	16
What is JPA?	16
JPA in Typical Architecture.....	17
JPA Annotations	17
Class Relationships in JPA	18
JPA Mapping.....	19
Entity Relationship in JPA.....	20
JPA and Hibernate	20
Set Up JPA and Hibernate Dependencies with Maven Project.....	21
1. Generate Project Template using Maven	21
2. Import Generated Maven Project into Eclipse IDE	23
3. Update Dependency Library on pom.xml	27

4. Start Creating Entity Class	28
5. Create Persistence.xml File	32
6. Persisting Entity Using the Entity Manager	34
Configuring Database Actions	41
• Schema Generation Database Action Create	41
• Schema Generation Database Action None	45
• Schema Generation Database Action Drop and Create	48
Drop and Create Actions Using Scripts	50
Summary	53

Course Overview

JP or the Java Persistence API is focused on persistence. Persistence can refer to any mechanism by which Java objects outlive the applications that create them. JP is not a tool or a framework or an actual implementation. It defines a set of concepts that can be implemented by any tool or framework.

JP supports object relational mapping, a mapping from objects in high level programming languages to tables which are the fundamental units of databases. JPA's model was originally based on Hibernate and initially only focused on relational databases. Today there are multiple persistent providers that support JPA but Hibernate remains the most widely used. Because of their intertwined relationship and JPA's origins from Hibernate, they're often spoken off in the same breath. JPA today has many provider implementations and Hibernate is just one amongst them.

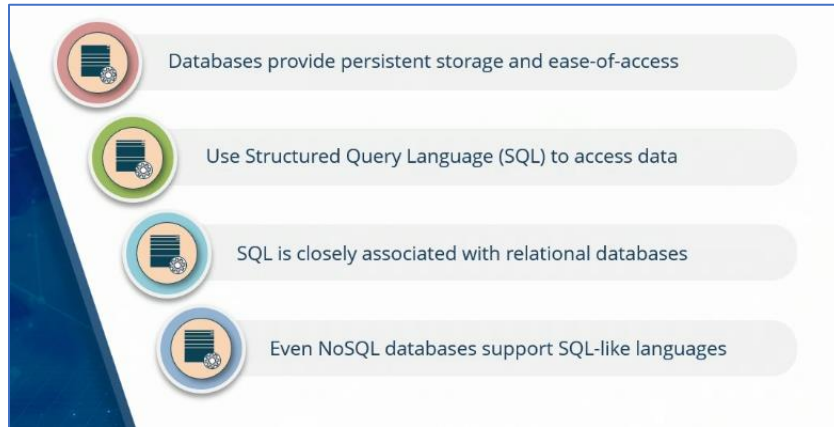
This course introduces the basic idea behind object relational mapping where entities and relationships expressed in an object oriented framework are mapped to records and tables in a relational database.

We'll discuss the basic features of JPA and the Hibernate framework and get set up with the MySQL database, the MySQL Workbench, and Apache Maven to manage dependencies in our Java application. Once you're done with this course, you will be able to use the persistence.xml file to connect to a database from a Java application, create entities and use the JPA entity manager to persist these entities in the underlying tables.

Object-relational Mapping

In order to understand the Java Persistence API and the Hibernate implementation of this API, why these are important and why they make our life very useful while writing applications that collect to databases, we need to go back a little bit and understand object relational mapping. You're in an organization and you're building something for that company or maybe you're writing your own prototype.

Databases for Structured Datastorage



- No matter what, it's quite likely that your application uses databases to provide persistence storage and ease of access to your data. There's practically no useful real world application that does not have an underlying database. And when you write code, you write code to access this data, retrieve it, display it, and so on.
- In order to update retrieve, access and delete data stored in a database, it's quite likely that you're using a programming language known as the Structured Query Language or SQL to access this data.
- SQL is closely associated with relational databases. Databases that are made up of tables. Tables have rows and columns in which your data is stored. Structured Query Language is the query language of choice for engineers, and others to access data in a database.
- SQL is so popular that it's used beyond relational databases. Even NoSQL databases support SQL-like languages.

Programmatic Access



- SQL is extremely useful for interactive operations by a human user. You'll log in and connect to your database using some kind of user interface or maybe the command line or terminal window and you'll run SQL queries to retrieve data.
- SQL however, doesn't really lend itself well for programmatic access. It's not really that convenient. SQL is actually highly optimized for human users. It's not really built for programming language access, such as using Python, Java or any other higher level programming language. But you can't get away from working with databases from within your application where you write code.
- All meaningful application programs have state, and
- this state is often best stored in a database, which makes it extremely important to have a good way to get programmatic access to your data stored in your database, so your data is accessible from within your application.
- This need was understood and felt acutely many years ago, which means over the years, various paradigms for programmatic database access from within your application have cropped up.
- For example, the ODBC specification ODBC stands for Open Database Connectivity, is a standard application programming interface for accessing database management systems. The designers of ODBC aim to make it independent of database systems and operating systems. The ODBC is a general API.
- If you're working in Java, you're likely to use JDBC, this stands for Java Database Connectivity. And it's a standard API written in Java allowing you to access underlying relational databases from Java code.
- The basic idea here is that both APIs are completely standard, well defined, well known and they abstract the application code that you write from the inner workings of the database.

ODBC for Programmatic Access

It doesn't matter what kind of database you're working with, whether it's a PostgreSQL database, a MySQL database, maybe even SQL Server, these APIs serve as a complete abstraction.



- The ODBC serves as a standard API for programmatic DBMS access.
- Databases can be changed under the hood without actually changing your code.
- There are ODBC drivers available in different programming languages
- Allow you to connect to the database, run queries, commands, retrieve data from a database and so on.

JDBC for Programmatic Access from Java

The ODBC programming API is not really language specific.



- JDBC is what you'd use to connect to a database from within a Java application. It's a standard Java API.
- It's similar in concept to ODBC.
- The basic idea remains the same. It has a way to set up connections, run queries, store procedures etc, transactions and other commands, except that it's written in Java and meant for a Java app.
- Both ODBC and JDBC were primarily developed for relational databases where data is stored in tables with rows and columns.

Relational Database vs Java Class

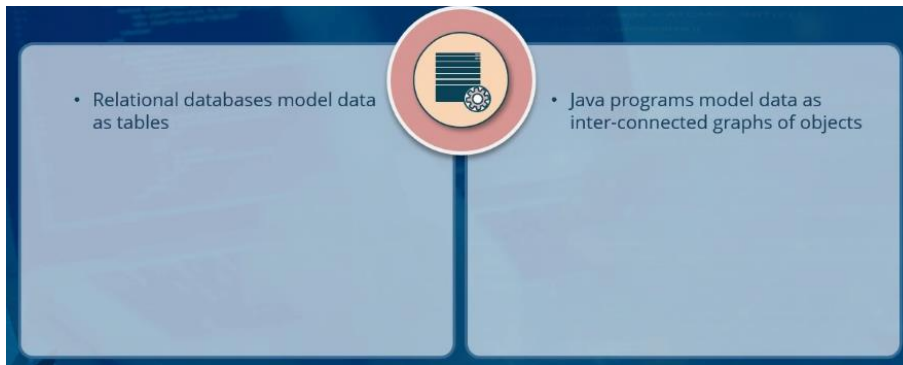
Now that we've understood what the Java Database Connectivity API is all about, let's talk about the relationship between relational databases and Java classes.



- Now in a relational database you know that the table is a fundamental unit. In an object oriented programming such as Java, the fundamental building block that you use in your code are objects. Objects are instantiated using classes, which serve as templates for the object.
- Tables are a fairly intuitive and straightforward structure. Tables consists of rows and columns. There are no nested or repeated types within a table. Classes are fairly complex. They contain complex member variables, dictionaries, generics, nested objects, other references and so on.
- When you define a table in a relational database, every field of your record corresponds to a column in this table. You set up your columns according to a fixed schema. Columns can be of different types and you have different constraints on your columns. In the case of classes, the class definitions change over time. You have member variables of a class, you may tweak those member variables. You don't really apply constraints to your member variables. Rather, you express constraints in terms of code.
- In a relational model, a record in a table represents an entity and relationships between entities are between tables via constraints such as foreign key constraints. That's pretty straightforward. Relationships between objects can be fairly complex. You can define these relationships via inheritance composition and a number of other design pattern.

Object-Relational “Impedance Mismatch”

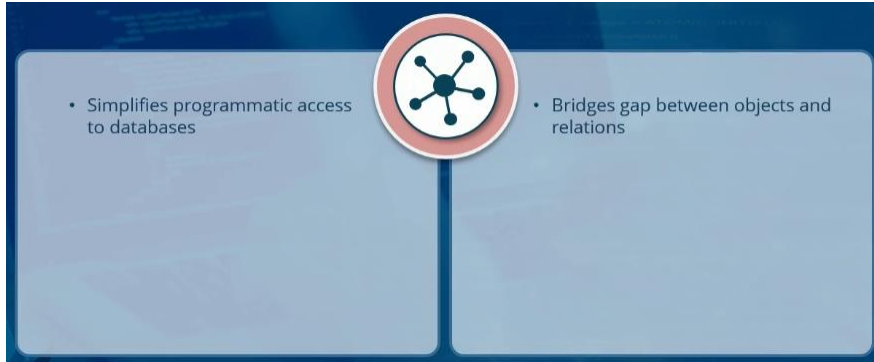
When you sit down to think about it, the relational model that an underlying database uses and the object model that you'd use within a programming language, well, there is an impedance mismatch between the two.



Relational databases model data in the form of tables, Java programs, where Java is a higher level object oriented programming language, model data as inter-connected graphs of objects. If you're working in Java and you need a way to work with your relational database under the hood, it's pretty clear that you need some additional help.

Object Relational Mapping

You need some way to map your objects to underlying database tables. And that's where Object-relational Mapping comes in.



The whole idea behind Object-relational Mapping is to simplify programmatic access to databases. It allows you to move seamlessly between an object oriented programming world and the relational world of your databases. It bridges the gap between objects and relations.

ORM for Programmatic Access from Java



Object-relational Mapping is provided using Object-relational Mapping frameworks or ORM frameworks, as they're called.

- ORM frameworks basically free you up from thinking about how objects within your programming world map to relations and how they're mapped to the underlying database tables.
- Object relational frameworks in Java are based on JDBC under-the-hood. ORM frameworks typically use JDBC to manage connections to the underlying database, manage transactions, queries, access, and so on.
- ORM frameworks serve to present you with an object-oriented view of your underlying data. It can cope with inheritance, object equality, associations and references between objects.
- In addition, it will deal with the database, perform transaction management, primary key generation and so on.

ORM frameworks understand database constructs as well as object-oriented programming constructs. And when you work with ORM frameworks, all you have to worry about is your object model.

Several ORM Frameworks

ORM frameworks have been around for a while, the need for them is acute. There are a number of different ORM frameworks available.



- The most popular of which is Hibernate part of the spring framework.
- But there are other frameworks as well, Enterprise JavaBeans Entity Beans, that is an ORM framework,
- Castor,
- the Spring Data Access Objects,
- Java Data Objects that is JDO.

All of these are different implementations of ORM frameworks in Java. Now in the context of ORM frameworks, you might have heard of the JPA or the Java Persistence API.

- The Java Persistence API is not a framework by itself, instead, it provides a standard API for dealing with ORM frameworks, such as Hibernate and EclipseLink.

The Hibernate Framework

In this learning path, we won't really worry about the other ORM frameworks that we discussed. We'll focus our attention on JPA that is the Java Persistence API, and Hibernate framework that implements the JPA model.

JPA – Java Persistence API

The first thing to understand about JPA is that it stands for a Java Persistence API. And it's a specification for persistence providers implemented using object relational mapping frameworks.



Remember ORMs know how to map objects from an object oriented programming world to database entities. ORMs are also known as persistence providers because they take care of persisting your Java objects to the underlying database. JPA is not an implementation, it doesn't actually work with databases, it's only an API.

Hibernate

The specification for the API that is implemented by many popular persistence providers specifically hibernate. Hibernate is an important Java Persistence framework that implements the Java Persistence API. I should tell you that Hibernate has been around for almost 20 years.



Hibernate in fact existed before the Java Persistence API specification came into being. Hibernate in fact served as the basis for the initial specification of the JPA.



- Primarily Hibernate can be thought of as an ORM framework. It provides object relational mapping, allowing you to work in a high level programming language such as Java with objects inheritance and other relationships. But it'll map these entities to the underlying relational database. But Hibernate has evolved over the years and it offers many more additional features.
- Hibernate Search allowing you to search within your database is a popular extension.
- You can use Hibernate Validators to check for constraints on your data.
- As NoSQL databases to store huge amounts of data got more popular. Hibernate OGM was developed to allow you to work with the Hibernate framework, but with NoSQL databases such as MongoDB.
- Hibernate implements the JPA spec, the Java Persistence API spec.
- The JPA spec is typically expressed using Java annotations. Hibernate annotations also exists, but today they typically not used. Many of them have been deprecated. When you're working with Hibernate, you'll use JPA annotations. The annotations will be part of the Java Persistence API. But the actual implementation of the mapping from your Java objects to the underlying databases will be Hibernate.

So Hibernate offers the implementation, JPA offers the API.

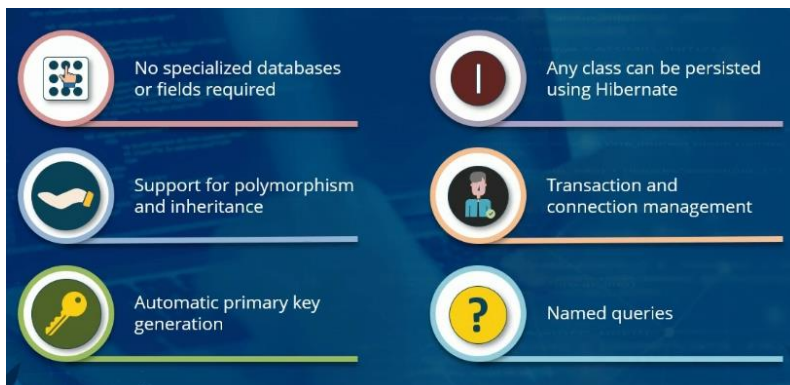
Reasons for the Popular of Hibernate



Of all of the persistence providers that implement the JPA spec, Hibernate is the most popular.

- It's very easy to use,
- it offers performance and scalability, and
- it has its own query language known as the Hibernate Query Language, that is compatible with JPQL. The query language that you'll use from within your JPA programming interface.

Why is hibernate easy to use



- It's not like you have to use a specialized database or specialized fields in your database. Hibernate works with anything.
- All classes, all objects within your Java code can be persisted using Hibernate.
- There is support for polymorphism and inheritance and natural ways of expressing relationships in OO code.
- Hibernate gives you transaction and connection management,
- Automatic primary key generation for your tables.
- And it also allows you to specify named queries. Queries that you run over and over again on your underlying tables. Hibernate gives you performance and scalability.

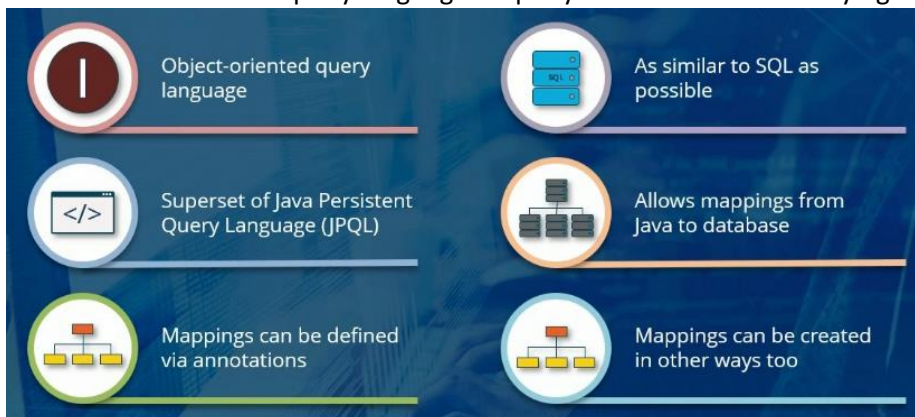
Performance and Scalability



- You can lazily load your data from the underlying database using lazy loading or lazy initialization.
- Hibernate offers multiple levels of caching for your data, so you don't go back to your tables over and over again.
- There is first-level caching within your Hibernate session.
- And there is second level caching that works across Hibernate sessions.
- There are different caching providers that you can plug in and use and configure different caching strategies.
- Hibernate also offers you optimistic locking, where you can lock your database table before you perform operations on it.

Hibernate Query Language (HQL)

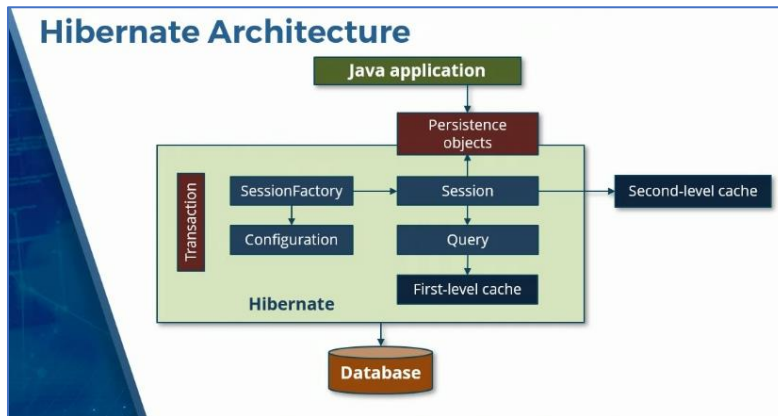
Hibernate uses its own query language to query the data in the underlying database tables.



- This is an object-oriented query language,
- but it's very, very similar to SQL. One that all database developers are familiar with.
- The Hibernate Query Language, in fact, is a superset of JPQL. The Java Persistence Query Language. This is the query language that we'll work with when we use JPA with Hibernate.
- The HQL allows mappings from Java to the database.
- Mappings can be defined via annotations.
- And mappings can be created using other techniques, such as XML files. Here's the basic architecture of the Hibernate framework.

Hibernate Architecture

You have your Java application and within your Java application, you only deal with objects. These objects can be tagged as persistence objects.



Now persistent objects within Hibernate are managed by a session. A session is created from a session factory. And you can specify the configuration of your session factory, indicating what database you want to connect to the username, password, and other aspects of your connection. You will instantiate a session from the session factory and you'll use that to run queries on your database tables.

Hibernate will perform first level caching within each session. A second level cache will work across sessions.

Hibernate existed before the Java Persistence API's specification was defined. So often users of Hibernate use Hibernate API's directly. A more common practice follow today is to use the Hibernate ORM framework with the Java Persistence API. And that's what we'll be doing in the rest of this learning path in our demos.

The JPA Framework

A popular way of working with persistence providers today is to use the Java Persistence API.



This is an API, that provides a standard way of interacting with persistence providers. In fact, when you use JPA, you can change your persistence providers and keep your code exactly the same. We have the database at the very bottom, we have an ORM framework and on top of the ORM framework you'll use the JPA, this means you're only interacting with a common interface.

So you can change your persistence provider, that is your ORM framework from say, Hibernate to EclipseLink, but your code remains the same.

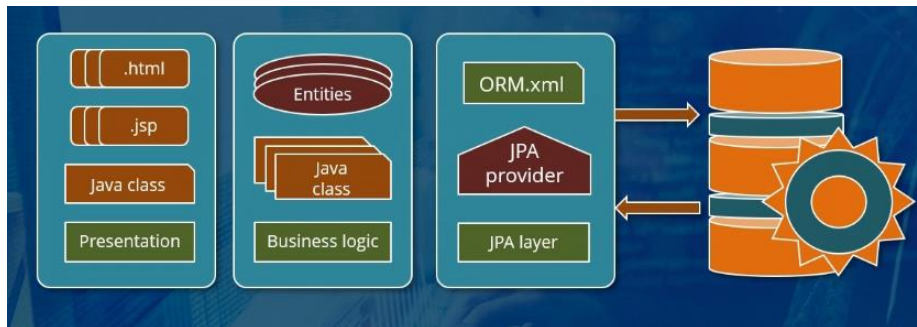
What is JPA?



- JPA is what is widely used today, JPA is an acronym that stands for, the Java Persistence API, you already knew this.
- JPA is a specification to standardize ORM behavior, object-relational mapping frameworks, are extremely important. And we needed some way that we could interoperate with multiple ORMs, and that's what JPA provides.
- In fact, the Hibernate object-relational mapping framework came first, JPA came after Hibernate was implemented, and was heavily inspired by Hibernate.
- Hibernate today is the most popular, the most robust, and the most mature JPA implementation, which is why JPA is often, used along with Hibernate.
- The JPA specification is considered to be intuitive and easy to use. The Hibernate OGM that works with NoSQL databases uses the Hibernate NoSQL JPA.
- There are other ORMs beyond Hibernate that implement the JPA specification. Another popular ORM is EclipseLink, this is an alternative implementation of the Java Persistence API.

JPA in Typical Architecture

Let's see where exactly in a typical ORM architecture JPA fits in, on the very right we have our database, typically a relational database, where our data is persisted.



Our JPA provider, that is the actual implementation interacts directly with the database. The JPA layer interacts with the JPA provider, and provides to the business logic Java objects, that can be modeled at entities stored in the underlying database.

These Java classes tagged as entities, which are persisted in the underlying database, make up the business logic of your application. And this business logic, is what the presentation layer uses to display your data to users.

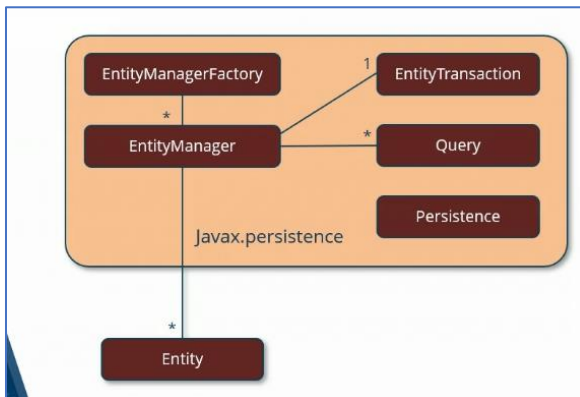
JPA Annotations



- The JPA programmatic interface is in the form of metadata annotations. These metadata annotations can be applied to classes, member variables of classes, getters, and setters, constructors and so on.
- And these annotations help map from your Java code, to the underlying Database Management System.
- All JPA implementations, whether it's Hibernate or EclipseLink, support these metadata annotations.
- The JPA spec also provides two additional classes, these are actual implementations.
- These classes are the PersistenceManager and the EntityManager.
- These classes contain the business logic, to manage entities and their persistence life cycle.

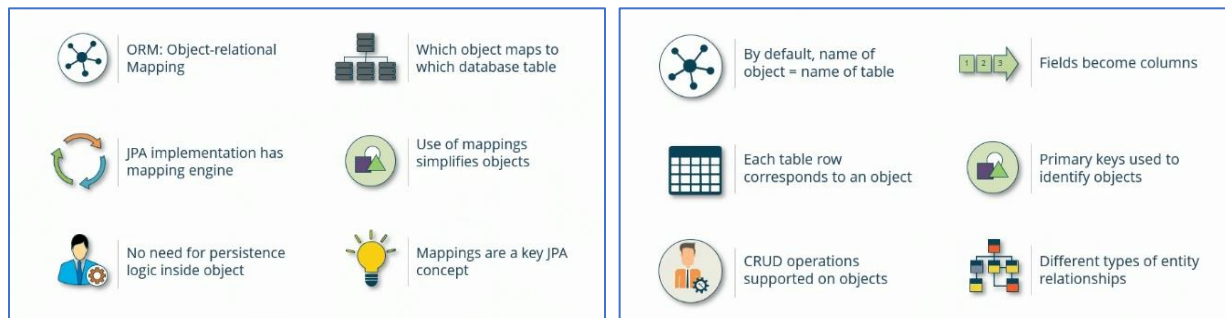
Class Relationships in JPA

When you work with the JPA, here is a high-level picture that you need to keep in mind, you'll use an entity manager factory to create an entity manager.



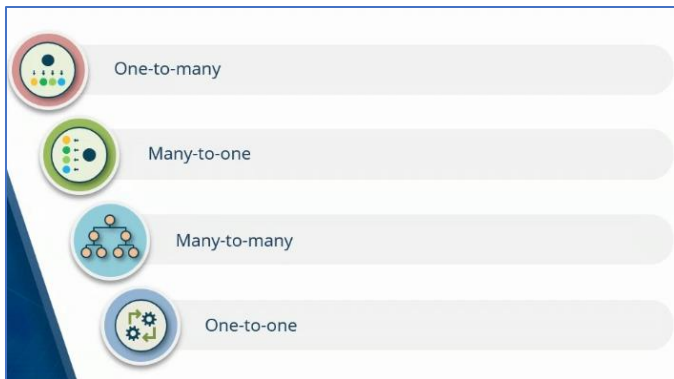
The entity manager is what you use to work with entities, that are persisted in your underlying database. In order to perform operations with the database, you will start off an EntityTransaction, run queries, maybe persist data.

JPA Mapping



- The JPA specification allows you to specify mappings, this is what facilitates the ORM framework, the Object-relational Mapping framework.
- This is what determines, what objects within your Java programming code, maps to which database table.
- The JPA implementation has a mapping engine, that manages these mappings for you, an example of a JPA implementation, as you know is Hibernate.
- Now, using these mappings simplifies the creation of the objects that you work with, within your Java code.
- You don't need to have persistence logic inside the object, you'll simply specify annotations and the mapping engine, and your JPA provider will take care of persistence.
- The mappings that you specify are a key JPA concept.
- JPA has a number of default specifications, by default, the name of your Java object corresponds to the name of the underlying table.
- Your fields or member variables within your Java object, become columns in that table,
- Every table row corresponds to one object that you instantiate.
- You have to specify primary keys used to identify objects in your Java code, these primary keys are applied on the underlying database table.
- Using the EntityManager class in JPA, you can have CRUD operations CREATE, READ, UPDATE, and DELETE operations on objects.
- And you can also specify different types of entity relationships using JPA.

Entity Relationship in JPA



When you model real world entities, entities may be involved in different kinds of relationships. These can be One-to-many, "Many-to-one", "Many-to-many", "One-to-one" relationships.

All of these relationships, can be modeled using the Java Persistence API.

JPA and Hibernate

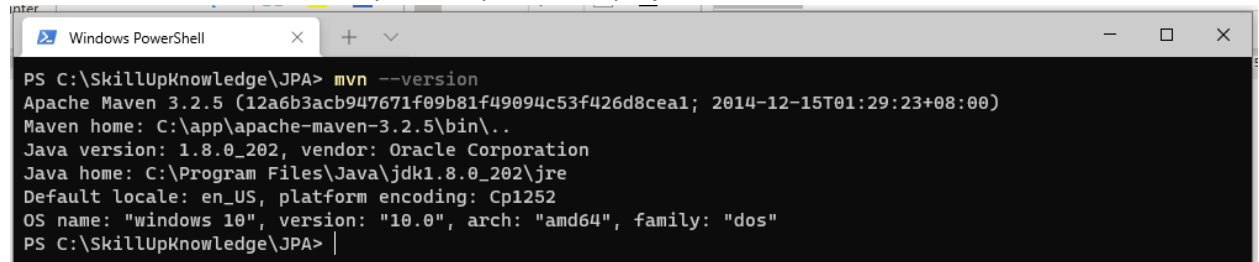


We are now ready to move forward to working with JPA and Hibernate. Remember, the Java Persistence API is used the most often along with the Hibernate implementation, which is why we use this combination for all of the demos in our learning path. So long as you're using JPA, rather than the Hibernate API directly, it gives you the option to switch your ORM framework if you want to. From Hibernate, you can move over to another implementation of JPA such as EclipseLink.

Set Up JPA and Hibernate Dependencies with Maven Project

1. Generate Project Template using Maven

Make sure is Maven successfully installed on our machine, whether it's a Mac machine or a Windows machine, we're ready to set up a Maven project.



```
PS C:\SkillUpKnowledge\JPA> mvn --version
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-15T01:29:23+08:00)
Maven home: C:\app\apache-maven-3.2.5\bin\..
Java version: 1.8.0_202, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_202\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"
PS C:\SkillUpKnowledge\JPA> |
```

Now projects are going to live within this `C:\SkillUpKnowledge\JPA` directory. So I cd into that directory and generate a new Maven project using the command line. The command to create a new Maven project is a standard one with standard parameters.

I use the `mvn archetype:generate` command. The `groupId` is `com.mytutorial.jpa` that's going to be the namespace for my package. The `artifactId` is the name of my project, `my-jpa-app`. The `artifactId` is the artifact that I use as a template to create my project and the standard one to use here is the `maven-archetype-quickstart`.

```
mvn archetype:generate -DgroupId=com.mytutorial.jpa -DartifactId=my-
jpa-app \
-DarchetypeArtifactId=maven-archetype-quickstart -
DinteractiveMode=false
```

This is the standard Maven template sufficient for most basic programs, go ahead and hit enter. And this will generate your Maven project. You might have to wait for a little bit it took about 15 seconds or so for Maven to run through to completion.

Java Persistence API: Getting Started With JPA & Hibernate

```
Windows PowerShell
PS C:\SkillUpKnowledge\JPA> mvn "archetype:generate" "-DgroupId=com.mytutorial.jpa" "-DartifactId=my-jpa-app"
"-DarchetypeArtifactId=maven-archetype-quickstart" "-DinteractiveMode=false"
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO] >>> maven-archetype-plugin:3.2.0:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO] <<< maven-archetype-plugin:3.2.0:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO] --- maven-archetype-plugin:3.2.0:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] -----
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] -----
[INFO] Parameter: basedir, Value: C:\SkillUpKnowledge\JPA
[INFO] Parameter: package, Value: com.mytutorial.jpa
[INFO] Parameter: groupId, Value: com.mytutorial.jpa
[INFO] Parameter: artifactId, Value: my-jpa-app
[INFO] Parameter: packageName, Value: com.mytutorial.jpa
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\SkillUpKnowledge\JPA\my-jpa-app
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.111 s
[INFO] Finished at: 2021-12-01T10:54:33+08:00
[INFO] Final Memory: 14M/198M
[INFO] -----
PS C:\SkillUpKnowledge\JPA>
```

If you see BUILD SUCCESS within your terminal window, you know your Maven project has been successfully created. Let's run an *dir* command here and you should see a new subfolder called *my-jpa-app*. And if you cd into this subfolder, you will find within here the **pom.xml** file. This is the XML file where you will configure the dependencies of your project.

```
Windows PowerShell
PS C:\SkillUpKnowledge\JPA> dir

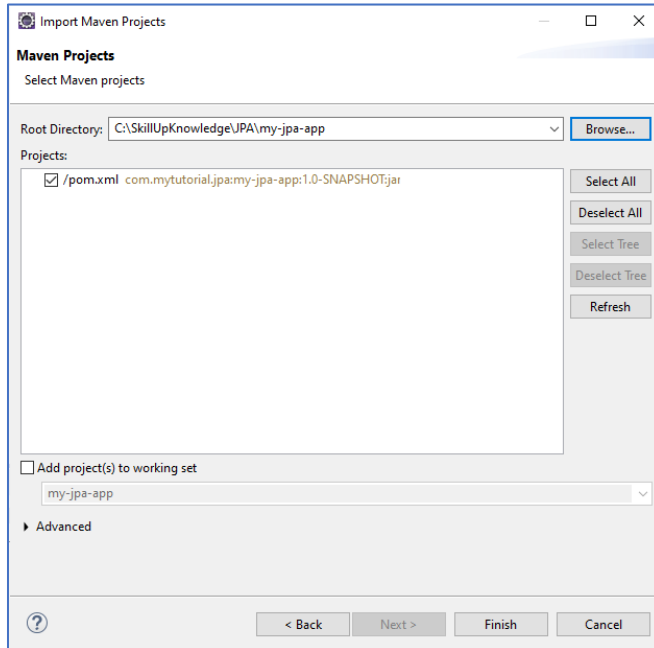
Directory: C:\SkillUpKnowledge\JPA

Mode                LastWriteTime         Length Name
----                -
d-----          12/1/2021  10:54 AM                my-jpa-app

PS C:\SkillUpKnowledge\JPA> tree /F
Folder PATH listing for volume OSDisk
Volume serial number is E6DA-7971
C:.
|-- my-jpa-app
|   |-- pom.xml
|   |-- src
|   |   |-- main
|   |   |   |-- java
|   |   |   |   |-- com
|   |   |   |   |   |-- mytutorial
|   |   |   |   |   |   |-- jpa
|   |   |   |   |   |       App.java
|   |   |-- test
|   |   |   |-- java
|   |   |   |   |-- com
|   |   |   |   |   |-- mytutorial
|   |   |   |   |   |   |-- jpa
|   |   |   |   |   |       AppTest.java
PS C:\SkillUpKnowledge\JPA>
```

2. Import Generated Maven Project into Eclipse IDE

The IDE that I'm going to use for all of the demos in this learning path is the Eclipse IDE. Go ahead and open up Eclipse and let's import the Maven project that we created using the terminal that is the command line. Click on the File option and choose the Import option. We are going to import our Maven project into our eclipse IDE. I have the Java Enterprise edition of Eclipse here. And you can see in the options below that you have an option to import a maven project. I'm going to select Existing Maven Projects because that's what I'm going to import here within Eclipse.



We'll import the Existing Maven Projects that we've created using the command line, click on the Next button here. And we'll use this Browse button to specify the root directory of our Maven project. So go to Project, Skillsoft, and within this subfolder you will find my-jpa-app, the folder that was created when we created our Maven project. In this subfolder, you'll find the pom.xml file and additional subfolders holding the source code. Let's go ahead and hit Open, and click on the Finish button here to complete the import of this Maven project into our Eclipse IDE. You can see the project is now visible in the left navigation pane.

Every Maven project has a pom.xml, which we use to configure the settings for our current project, our Java project here. Here is what the default **pom.xml** generated by Maven looks like.



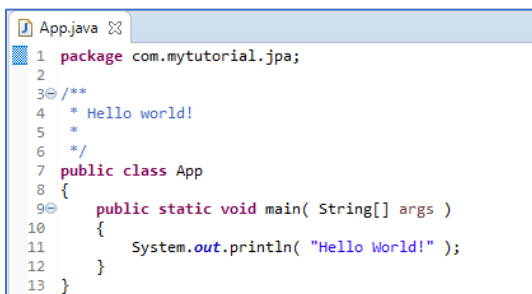
The **groupId** is `com.mytutorial.jpa`, the **artifactId** is `my-jpa-app`. Now before you edit the **pom.xml**, you might have to wait for a bit for Eclipse to load in all of the dependencies specified by **pom.xml**. Once that is complete, let's go ahead and specify a few properties. These are properties that tell Maven what Java compiler we want Maven to use. Our **JDK version is 1.8**, so make sure you specify these within properties here.

A screenshot of the Eclipse IDE showing the `my-jpa-app/pom.xml` file. The XML content is as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <groupId>com.mytutorial.jpa</groupId>
6   <artifactId>my-jpa-app</artifactId>
7   <packaging>jar</packaging>
8   <version>1.0-SNAPSHOT</version>
9   <name>my-jpa-app</name>
10  <url>http://maven.apache.org</url>
11  <properties>
12    <java.version>1.8</java.version>
13  </properties>
14  <dependencies>
15    <dependency>
16      <groupId>junit</groupId>
17      <artifactId>junit</artifactId>
18      <version>3.8.1</version>
19      <scope>test</scope>
20    </dependency>
21  </dependencies>
22 </project>
```

A red rectangle highlights the `<properties>` section, specifically the `<java.version>1.8</java.version>` line.

We'll set up the Java source code files in `src/main/java`, you can expand this directory structure. And you will find within there that **App.java**, the entry point for your Maven application has already been created.

A screenshot of the Eclipse IDE showing the `App.java` file. The Java code is as follows:

```
1 package com.mytutorial.jpa;
2
3 /**
4  * Hello world!
5  *
6  */
7 public class App
8 {
9     public static void main( String[] args )
10    {
11        System.out.println( "Hello World!" );
12    }
13 }
```

By default **App.java** contains a simple Hello World! Under `src/test/java`, you will find a default test file setup, `AppTest.java` containing a unit test for your `App.java`. If you wish to write unit test for whatever application you'll build, this is where you place the test files, here this `AppTest.java`.


```

AppTest.java
1 package com.mytutorial.jpa;
2
3 import junit.framework.Test;
4
5
6
7 /**
8  * Unit test for simple App.
9  */
10 public class AppTest
11     extends TestCase
12 {
13     /**
14      * Create the test case
15      *
16      * @param testName name of the test case
17      */
18     public AppTest( String testName )
19     {
20         super( testName );
21     }
22
23     /**
24      * @return the suite of tests being tested
25      */
26     public static Test suite()
27     {
28         return new TestSuite( AppTest.class );
29     }
30
31     /**
32      * Rigorous Test :-)
33      */
34     public void testApp()
35     {
36         assertTrue( true );
37     }
38 }

```

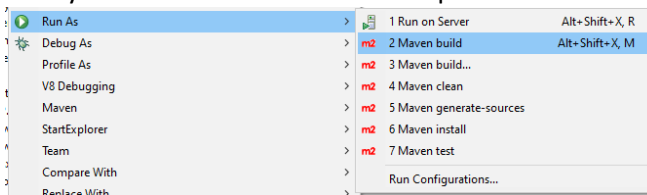
Once you have everything configured, all you need to do to run your App.java main method is to select the project, right-click on it, choose Run As and choose the Java Application option. If you see a dialogue pop up, this will ask you to select the application that you want to run App at *com.mytutorial.jpa* is what we want to execute, select OK.

```

JUnit Console
<terminated> App (2) [Java Application] C:\Program Files\Java\jdk1.8.0_20
Hello World!

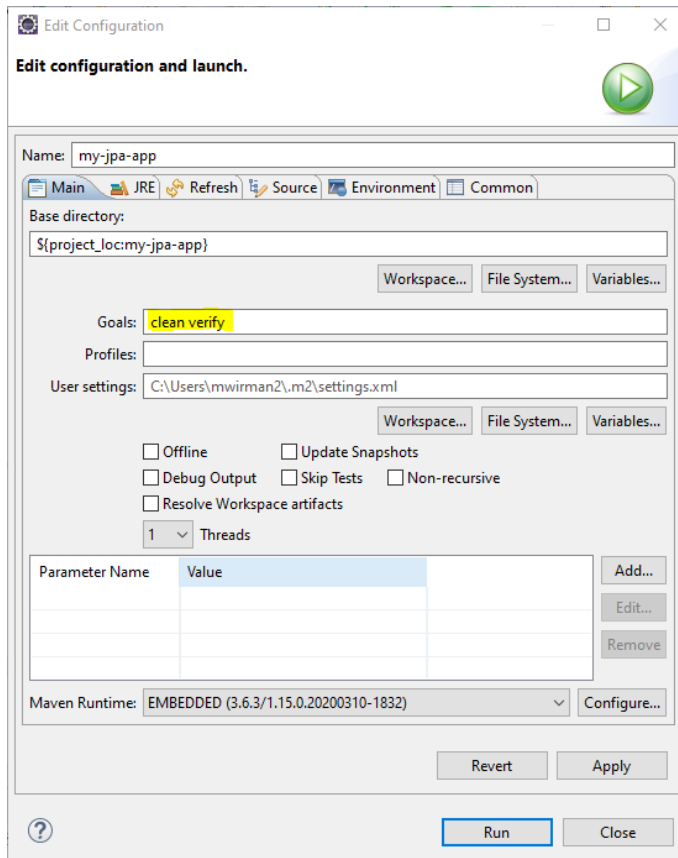
```

If you have any unsaved resources, you may be prompted for that as well. Click on the OK button here. And observe Hello World! printed onto screen, you successfully run your Maven app. If you want to explicitly do a Maven build, right click on your project, choose Run As, and then you'll have a bunch of Maven options.

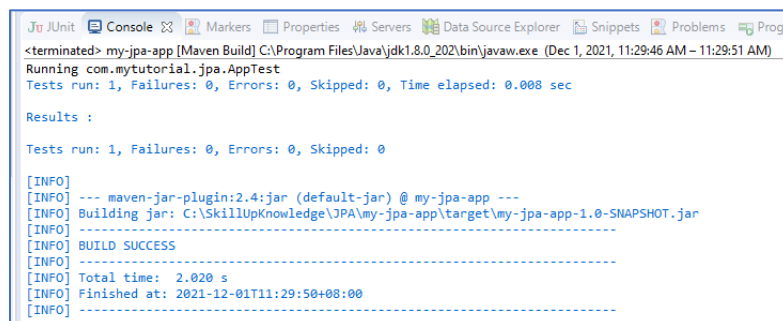


Choose **Maven > build...**, and this will bring up a dialog which you can use to configure your Maven build. Select the Workspace button and configure the current workspace as the one that you want to build *my-jpa-app*. Click on **OK**. Notice the base directory changes. You will now need to specify the goals of your build.

Java Persistence API: Getting Started With JPA & Hibernate



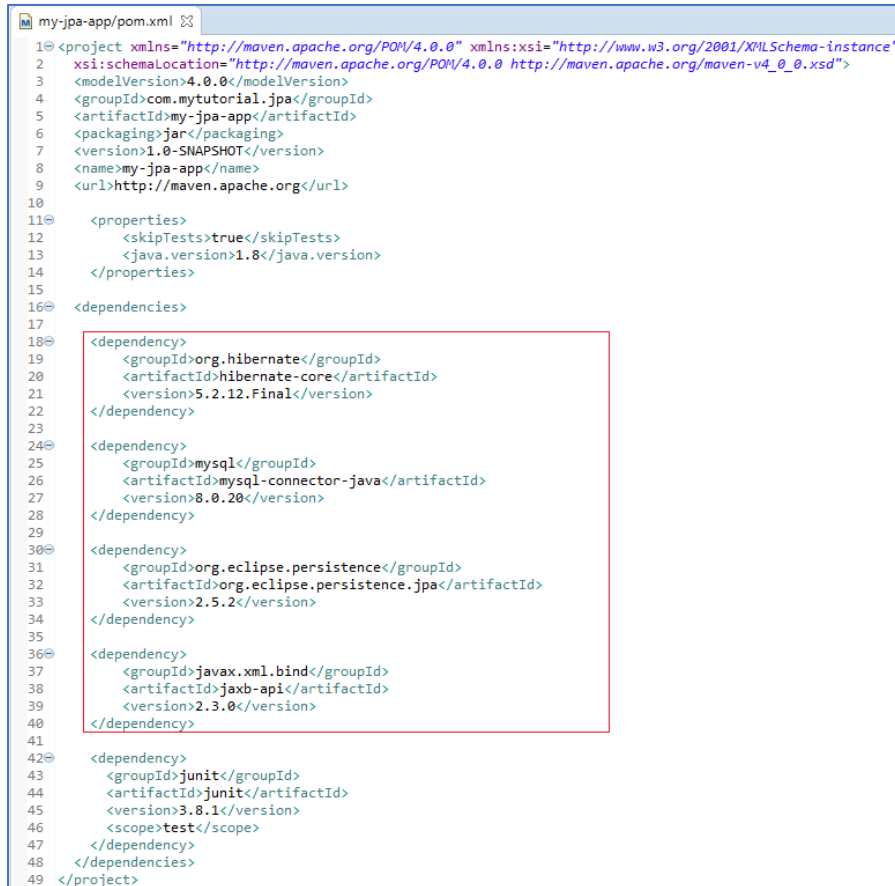
What Maven operation do you want to run? I want to build **clean, verify**. Once you've specified the goals, go ahead hit Apply and then hit Run. Your project should build successfully using Maven.



If you scroll down below you should see BUILD SUCCESS at the very end. Now we know that our Maven project works just fine. It's now time for us to specify the dependencies for JPA and Hibernate.

3. Update Dependency Library on pom.xml

So we can connect to our relational database from within Java. And these dependencies will be specified in the **pom.xml** file of our Maven project. I'm going to adding new dependency in the pom.xml.



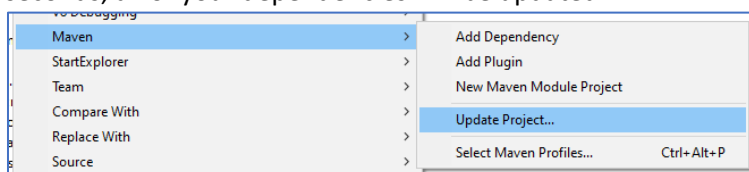
```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>com.mytutorial.jpa</groupId>
5   <artifactId>my-jpa-app</artifactId>
6   <packaging>jar</packaging>
7   <version>1.0-SNAPSHOT</version>
8   <name>my-jpa-app</name>
9   <url>http://maven.apache.org</url>
10
11   <properties>
12     <skipTests>true</skipTests>
13     <java.version>1.8</java.version>
14   </properties>
15
16   <dependencies>
17
18     <dependency>
19       <groupId>org.hibernate</groupId>
20       <artifactId>hibernate-core</artifactId>
21       <version>5.2.12.Final</version>
22     </dependency>
23
24     <dependency>
25       <groupId>mysql</groupId>
26       <artifactId>mysql-connector-java</artifactId>
27       <version>8.0.20</version>
28     </dependency>
29
30     <dependency>
31       <groupId>org.eclipse.persistence</groupId>
32       <artifactId>org.eclipse.persistence.jpa</artifactId>
33       <version>2.5.2</version>
34     </dependency>
35
36     <dependency>
37       <groupId>javax.xml.bind</groupId>
38       <artifactId>jaxb-api</artifactId>
39       <version>2.3.0</version>
40     </dependency>
41
42     <dependency>
43       <groupId>junit</groupId>
44       <artifactId>junit</artifactId>
45       <version>3.8.1</version>
46       <scope>test</scope>
47     </dependency>
48   </dependencies>
49 </project>

```

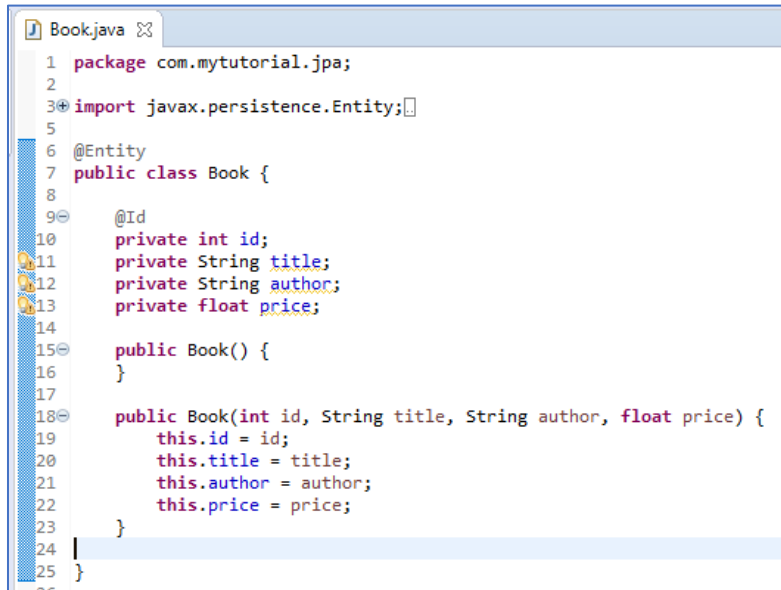
- I've specified the dependency of this project on the Hibernate implementation of JPA, **groupId** is [org.hibernate](#) **artifactId** [hibernate-core](#). And the version, the latest at the time of this recording is [5.2.12.Final](#).
- The Hibernate implementation is what we are going to use as our JPA provider. Now, in addition to Hibernate, we need a dependency on the actual **Java Persistence API**. This is the dependency specified on this line [org.eclipse.persistence.jpa](#) version [2.5.2](#).

These dependencies are present in my project. Our project requires an update and needs to load in Maven dependencies. Select the project, right click, choose the Maven option and choose the option here to Update Project. This will ensure that all of the references within your source code reference the right dependencies, click on OK. And in just about under 10 or so seconds, all of your dependencies will be updated.



4. Start Creating Entity Class

I'm now going to set up a brand new class which uses JPA annotations to identify entities where entities are persistent objects in our database. Now we'll discuss in detail what an entity is in just a bit in the next video. Here is a new class that I've set up called **Book.java**. I'm going to annotate **Book.java** so that it represents a table in the underlying database.



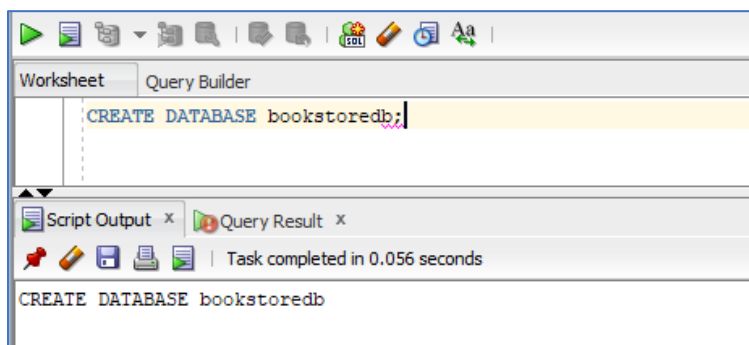
```

1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4
5
6 @Entity
7 public class Book {
8
9     @Id
10    private int id;
11    private String title;
12    private String author;
13    private float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24 }

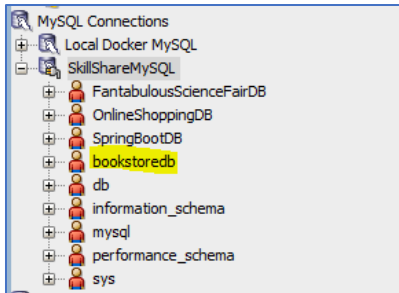
```

Here's the basic structure of Book.java with a few member variables and some constructors. And I'm going to add in an **@Entity** annotation for this Book class. **@Entity** is a JPA annotation, identifying this book class as a persistent object. And this **@Entity** annotation is present in the namespace **javax.persistence**. In addition, I'm going to set up a **primary key** for this Book entity, that is the ID field, which I tag using the **@Id** annotation.

And we are now ready to write code using the Java Persistence API. But before we do that, let's head over to our SQL workbench and create a database for us to work with. This database is called bookstoredb. Go ahead and run this command using the lightning icons here. And a new database called bookstoredb will be created within your MySQL Server.



If you refresh the schema pane off to the left of your screen, you'll see that the new database is listed there, bookstoredb has been successfully created.



And this is the database that we are going to use in our very first JPA example. Here we are within our Java project in this Eclipse editor. Observe that next to App.java, I've created a new Java file called **Book.java**. This Book.java file contains a bunch of code, and observe that I have tag this code using the **@Entity** annotation.

```
6 @Entity
7 public class Book {
8
9     @Id
10    private int id;
11    private String title;
12    private String author;
13    private float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24 }
```

What does this mean? An entity in JPA corresponds to a persistence domain object. What does that mean? This is the object whose fields will be persisted within a database as a table.

Now, there is a very standard format that an entity has, the name of the class that we've annotated using the **@Entity** annotation, which is the Book class here corresponds to the name of the table within our database. So the way you think about this entity is that objects of this entity contain records, which are inserted into the book table in our relational database. Now observe that this class has fields within it.

```
6 @Entity
7 public class Book {
8
9     @Id
10    private int id;
11    private String title;
12    private String author;
13    private float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24 }
```

Every field or member variable within this Entity object, corresponds to the columns in our table, within our relational database. So the Book table has four columns as defined by this

entity object, id, that is the ID of a book, title, which is a string field, author, that is a string field, and a floating point variable called price. This will correspond to a column in our book table containing floating point data.

Now observe that our id member variable has an annotation called **@Id**. The **@Id** annotation tells JPA that this is the unique primary key of the Book table. The **id** column is the **primary key**. And as is the case for any primary key within a relational database, this id variable should contain unique values for the different book objects.

Classes in JPA that have been annotated using the **@Entity** annotation **have to HAVE** a field with the **@Id** annotation. If by some chance you specify an entity without an ID, well, JPA won't let you. When you try and run this code, you will encounter an error.

*“So make sure that all of your **entities** have a **primary key** specified by **@Id**. ”*

You might have observed that the fields in your Entity object that corresponds to the column in your Book database table, can be either primitive fields or Java classes. So String is an example of a Java class, int and float are primitive fields.

Primitive fields, by default, are set to not null by hibernate under the hood. Whenever you specify primitive fields within your entities, those primitive fields will be columns that cannot hold null values.

Fields whose data types are Java classes can hold null values. However, primary keys cannot be null. So anything with the **@Id** annotation cannot be null.

In JPA, when you annotate a class using the **@Entity** annotation, make sure that it has a default **no argument constructor**, which you can see here on line 15 and 16.

```
1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Id;
5
6 @Entity
7 public class Book {
8
9     @Id
10    private int id;
11    private String title;
12    private String author;
13    private float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24 }
```

In addition, you can specify other additional constructors to initialize the fields within your class. I have an additional constructor on line 18, which takes in four input arguments, the unique id, that is the primary key, the string, title and author, and the floating point price of each book.

The Java Persistence API uses these annotations, indicate to the Hibernate object relational mapping provider how exactly all of our object oriented classes map to the underlying database structure. Hibernate predates JPA, and in fact, when Hibernate was originally launched, it had its

own set of annotations. Many of those have been deprecated in favor of JPA annotations. All JPA annotations are in the namespace **javax.persistence**.

You can see that our import statements here import **javax.persistence.Entity** and **javax.persistence.Id**. This is important. Don't mix up JPA annotations with Hibernate annotations unless you know exactly what you're doing. They do work together. But the best practice followed today is to use Hibernate using the Java Persistence API, and not Hibernate annotations directly.

5. Create Persistence.xml File

Now we know that this Book class represents a table, and Book objects represent records in that table.

But we now need a way to tell JPA that this Book table needs to be created or exists within the bookstordb that we set up in our MySQL database. How do we make this connection? Well, using a **persistence.xml** file, and this **persistence.xml** file is something that you need to explicitly set up.

Under your main Project, in the source main directory, create a new folder named **resources**. This is the resources folder that will contain all of your XML files and also SQL commands that you might want to execute when you run this Maven hibernate project. Observe the path to this resources folder.

We have *my-jpa-app*, under that we have the source folder, then main, and then resources. Once this resources folder has been created, you will need to create another subfolder within resources, and this is called the META-INF folder. By default, your hibernate implementation of the Java Persistence API looks for the **persistence.xml** file in this **META-INF** subfolder under your resources folder. So once META-INF has been created, go ahead and right click on it, choose the New option and select Other. This option will bring up a dialogue allowing to choose the kind of file that you want to create.

I want to create an XML File which holds my persistence information. This XML File is what tells my hibernate implementation of JPA how exactly to connect to the database where objects will be persisted. Select XML File here, click on the Next button, and give your XML file a name. **persistence.xml** is what our hibernate implementation of JPA expects. Click on the Finish button and a new persistence.xml file is created within the resources META-INF folder.



This persistence file contains information of how exactly hibernate should connect to the underlying MySQL database that we have set up. Now observe the structure of this **persistence** file, it's an **XML File**. And within the outer **persistence** tag, we have a **persistence-unit**. The persistence-unit is a logical grouping of user defined classes whose objects will be persisted within our database.

The persistence-unit contains a bunch of configurable properties telling this JPA hibernate implementation how exactly we should connect to this particular database in our database server, and how exactly objects should be persisted.



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <class>com.mytutorial.jpa.Book</class>
8     <properties>
9       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11      <property name="javax.persistence.jdbc.user" value="root" />
12      <property name="javax.persistence.jdbc.password" value="password" />
13
14      <property name="javax.persistence.schema-generation.database.action" value="create"/>
15      <property name="hibernate.show_sql" value="true"/>
16      <property name="hibernate.format_sql" value="true"/>
17    </properties>
18  </persistence-unit>
19
20 </persistence>

```

Every persistence-unit is identified by unique name. Here the name that I've specified, which references our bookstore database is `BookstoreDB_Unit`. This is the name that we'll use within our code, so keep track of how this name has been specified and all the upper casing and the capitalization in this name.

The entity classes that you define within your JPA project should be defined in the form of class mappings within your persistence.xml. For example, I have a class tag with `com.mytutorial.jpa.Book`. This references the complete path to the Book entity that I had set up earlier. But note though, it's not really necessary for us to have this mapping. As long as you've annotated your user defined persistent objects using the **@Entity** annotation, you don't need an explicit entry in persistence.xml for your entities as you shall see in just a bit.

For each persistent-unit that you define within persistence.xml, you need to specify properties. And these properties indicate how you should connect to the underlying database to work with this persistence-unit. Observe that we have a **jdbc.url**, which tells hibernate how it can use jdbc to connect to our bookstoredb within our MySQL database. The value of the jdbc.url property is basically the way we reference our MySQL database running on localhost 3306. If your port number is different, make sure you specify the right port number here. And then we use a / and specify the name of the database, which in our case in this example is bookstoredb.

In addition, you need to specify a user and a password which hibernate will use in order to connect to our database. We are using the `"root"` user and the password for our `root` user happens to be `"password"`. It's pretty clear from this persistence.xml file that our hibernate implementation of the Java Persistence API uses **jdbc** under the hood to connect to the SQL database. The jdbc driver is also specified here `com.mysql.jdbc.driver`. And then, we have three additional properties that have to do with how hibernate works with the underlying database.

The first of these properties `javax.persistence.schema-generation.database.action` is extremely important. Because it tells your project how to deal with the underlying database when you run your code. The value `create` that we've specified for the database action here tells hibernate to create database tables corresponding to our entity objects if the tables do not already exist in our database. If the tables already exist, that will be an error.

With this configuration setting, when you run this code, tables will be automatically created for you provided they don't exist. The two additional properties are `hibernate.show_sql`, and `hibernate.format_sql` which will display the SQL commands that hibernate runs under the hood each time we run our code. These properties are great for debugging.

6. Persisting Entity Using the Entity Manager

Before our Java code is entirely set up to work with our MySQL database, there is some additional wiring up that we need to do in order to get our Hibernate implementation of the Java persistence API to create tables and records within those tables.

This last bit of wiring up, I'm going to do within the main code of this project. I'm going to set up an `EntityManager` that will work with the entities in our database. The first step here is to set up the import statement for the libraries that we'll use in our code. The **EntityManager** is what we'll use to perform CRUD or Create, Read, Update and Delete operations on entities in our database. The `EntityManager` is created using the `EntityManagerFactory`, which in turn is created using the `Persistence` object.

Here is our **App.java** containing our main code. Instead of the Hello World! application that we had earlier, I'm going to delete this code and set up the code to actually use the `EntityManager` to persist entities in our database. The **EntityManager** that we've instantiated here, which we'll use through all of the demos in this learning part, is the Application-managed **EntityManager**. The Application-managed `EntityManager` can be used by the Java Enterprise Edition environments as well as the Java Standard Edition environment.

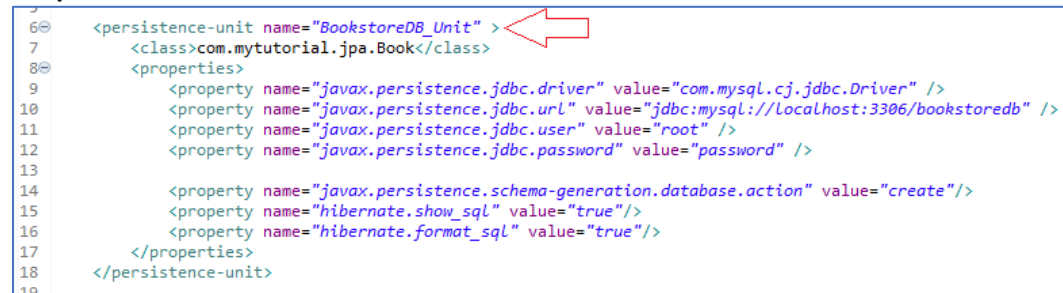
```
App.java
1 package com.mytutorial.jpa;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityManagerFactory;
5 import javax.persistence.Persistence;
6
7 /**
8  * Hello world!
9  */
10
11 public class App
12 {
13     public static void main( String[] args )
14     {
15         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
16         EntityManager entityManager = factory.createEntityManager();
17
18         try {
19             entityManager.getTransaction().begin();
20
21             Book firstBook = new Book(1234, "The Java Language Specification",
22                                     "Gilad Barcha", 99f);
23             Book secondBook = new Book(2222, "The Java Language Specification Second Edition",
24                                     "Gilad Barcha, James gosling", 119f);
25
26             entityManager.persist(firstBook);
27             entityManager.persist(secondBook);
28
29         } catch (Exception ex) {
30             System.err.println("An error occurred: " + ex);
31         } finally {
32             entityManager.getTransaction().commit();
33             entityManager.close();
34             factory.close();
35         }
36     }
37 }
38
39 }
```

In the Java EE environment, it's possible for you to set up a dependency injection framework to inject the `EntityManager` in. It's quite possible that within your organization, that's how you will get access to the `EntityManager`. But when we're focused on learning JPA, instantiating the `EntityManager` as an application managed one makes a lot of sense. And that's exactly what we'll do here.

The first thing is to use the Persistence class to create an **EntityManagerFactory**.

```
EntityManagerFactory factory =
    Persistence.createEntityManagerFactory("BookstoreDB_Unit");
```

The `EntityManagerFactory` is always associated with the Persistence unit that we've specified in the **persistence.xml** file.



```

6  <persistence-unit name="BookstoreDB_Unit" >
7      <class>com.mytutorial.jpa.Book</class>
8      <properties>
9          <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10         <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11         <property name="javax.persistence.jdbc.user" value="root" />
12         <property name="javax.persistence.jdbc.password" value="password" />
13
14         <property name="javax.persistence.schema-generation.database.action" value="create"/>
15         <property name="hibernate.show_sql" value="true"/>
16         <property name="hibernate.format_sql" value="true"/>
17     </properties>
18 </persistence-unit>
19

```

The only persistence unit that we have configured is the **BookstoreDB_Unit**. Make sure you specify the name of the persistence unit correctly. Otherwise you won't get access to the `EntityManagerFactory`.

On line 16, we create the `EntityManagerFactory` using the Persistence bootstrap class. The Persistence class has access to our **persistence.xml** configuration properties, which we'll use to connect to the underlying database using JDBC. All of that is abstracted away from us.

Now `BookstoreDB_Unit` is the persistence unit to which this **EntityManagerFactory** is tied. We then use this factory on line 17 to create an **EntityManager**.

```
EntityManager entityManager = factory.createEntityManager();
```

And this **EntityManager** is what we'll use to interact with the underlying persistence unit and manage our entities life cycle. An **EntityManager** is **not thread safe**, so the same `EntityManager` should not be shared across multiple threads.

We'll now perform a bunch of create operations on entities, which means we are going to be inserting rows into a newly created database table. Now this we'll do within a transaction, that is why on line 20, I have **entityManager.getTransaction().begin**, which begins our transaction.

```
entityManager.getTransaction().begin();
```

We've already defined our entity class earlier, which is the book class in `Book.java`. I'm now going to instantiate two book objects. The first of these books is the Java language specification by *Gilad Bracha*, with the price of \$99. Observe that this book has a unique ID 1234.

```
Book firstBook = new Book(1234,
```

```
"The Java Language Specification", "Gilad Bracha", 99f);
```

The second book is the one with the unique ID 2222, this is The Java Language Specification, Second Edition by Gilad Bracha and James Gosling.

```
Book secondBook = new Book(2222,  
    "The Java Language Specification Second Edition",  
    "Gilad Bracha", 119f);
```

These book objects are JPA entities which means they can be persisted to our database. And the way we do this is by invoking the `entityManager.persist()` method.

```
entityManager.persist(firstBook);  
entityManager.persist(secondBook);
```

Persist the first book as well as the second book and this persistence will be committed to our database when you call `entityManager.getTransaction().commit()`.

```
entityManager.getTransaction().commit();
```

Now, this entire series of operations has opened up a JDBC connection under the hood. Hibernate has done that for us. And our code here will insert two book records into the book table. Now because our database action in the `persistence.xml` file has been set to **create**, this table will be created if it doesn't already exist.

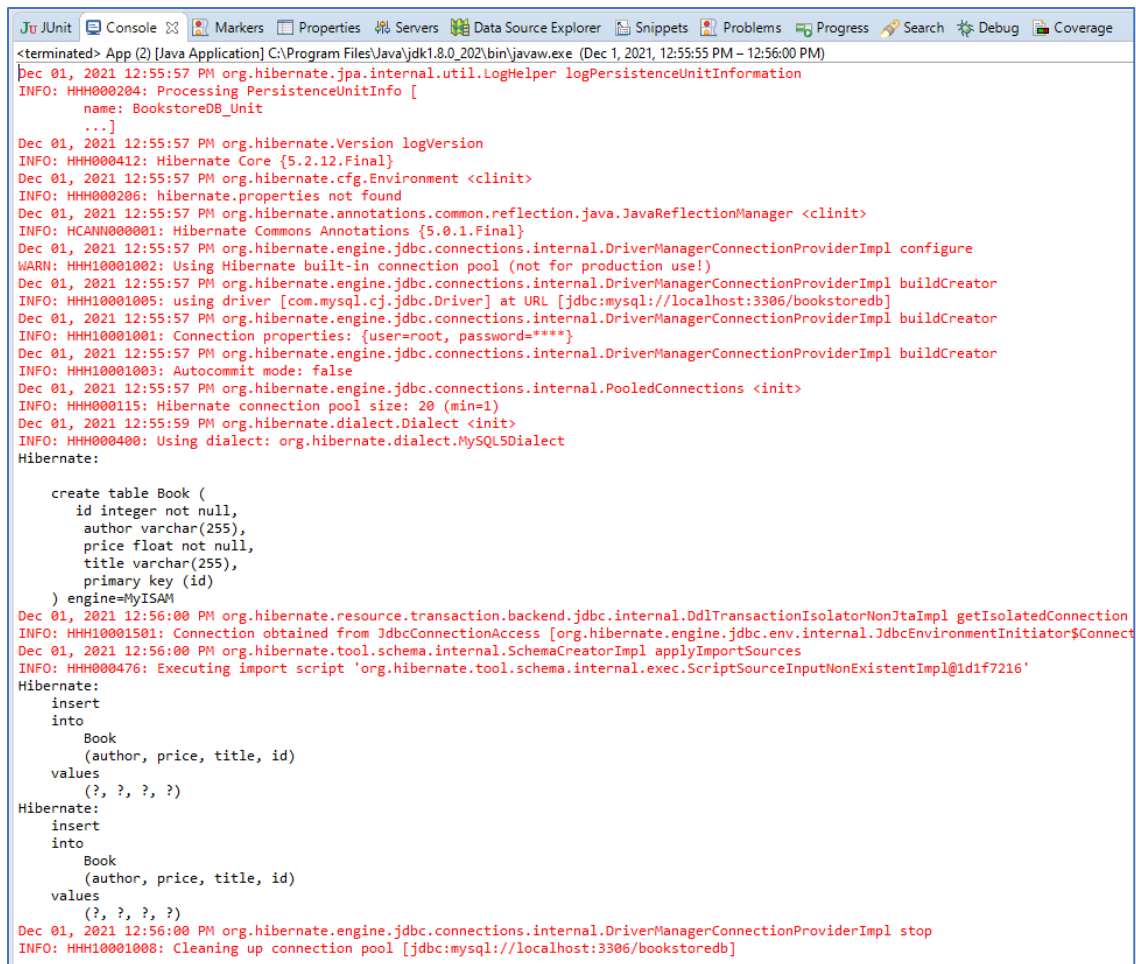
But in order to commit all of these operations within our transaction, we need to call `entityManager.getTransaction().commit()`. Once you've completed your operation, it's good practice to close any resources which you've been using. Which is why we'll call `entityManager.close()` and `factory.close()` on lines 33 and 34.

```
entityManager.close();  
factory.close();
```

And that's all you need to do to work with your database using JPA and Hibernate.

If you have a keyboard shortcut to run your application, you can use that. Or you can go to `run`, and `Run as Java Application`. This will run your Maven application and execute the code that you've written. And you can see the result of your execution here down below on the console window.

Java Persistence API: Getting Started With JPA & Hibernate



```
<terminated> App (2) [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Dec 1, 2021, 12:55:55 PM - 12:56:00 PM)
Dec 01, 2021 12:55:57 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
    name: BookstoreDB_Unit
    ...]
Dec 01, 2021 12:55:57 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.2.12.Final}
Dec 01, 2021 12:55:57 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Dec 01, 2021 12:55:57 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
Dec 01, 2021 12:55:57 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
Dec 01, 2021 12:55:57 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/bookstoredb]
Dec 01, 2021 12:55:57 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=root, password=****}
Dec 01, 2021 12:55:57 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
Dec 01, 2021 12:55:57 PM org.hibernate.engine.jdbc.connections.internal.PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Dec 01, 2021 12:55:59 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    create table Book (
        id integer not null,
        author varchar(255),
        price float not null,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 01, 2021 12:56:00 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$Connect
Dec 01, 2021 12:56:00 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@1d1f7216'
Hibernate:
    insert
    into
        Book
    (author, price, title, id)
    values
    (?, ?, ?, ?)
Hibernate:
    insert
    into
        Book
    (author, price, title, id)
    values
    (?, ?, ?, ?)
Dec 01, 2021 12:56:00 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

All of the logs generated by Hibernate are displayed in red. So don't be worried about that red output that you see on your console. Look for exceptions, though. Observe that the persistent unit that we are working with is the **BookstoreDB_Unit**, that is the name. You can scroll down below and see other bits of information that are useful.

You can see that we have a lot of **org.hibernate.log** statements indicating that JPA uses hibernate under the hood and hibernate uses JDBC to connect to our database.

We had set up the properties in our **persistence.xml** configuration file, have Hibernate display the SQL commands that are executed under the hood. Observe that Hibernate creates a new book table, because this table doesn't already exist.

The reason this book table is created is because of our **javax-persistence-schema-generation.database.action** property, which we've set to the value **create**. We can see the query that creates the book table, because we have **hibernate.show_sql** value **true**, and **hibernate.format_sql** value **true**.

This is in our persistence.xml file.

```
6 <persistence-unit name="BookstoreDB_Unit" >
7   <class>com.mytutorial.jpa.Book</class>
8   <properties>
9     <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10    <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11    <property name="javax.persistence.jdbc.user" value="root" />
12    <property name="javax.persistence.jdbc.password" value="password" />
13
14    <property name="javax.persistence.schema-generation.database.action" value="create"/>
15    <property name="hibernate.show_sql" value="true"/>
16    <property name="hibernate.format_sql" value="true"/>
17  </properties>
18 </persistence-unit>
```

These are great debugging statements in order to really help us understand how JPA and Hibernate work with the underlying database.

If you scroll down a little bit here, you'll be able to see the columns that have been set up for this book table.

```
INFO: HHH10001003: Autocommit mode: false
Dec 01, 2021 12:55:57 PM org.hibernate.engine.jdbc.connections.internal.PooledConnections <
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Dec 01, 2021 12:55:59 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    create table Book (
      id integer not null,
      author varchar(255),
      price float not null,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 01, 2021 12:56:00 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransa
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc
Dec 01, 2021 12:56:00 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSe
```

Observe that these columns correspond exactly with the names of the member variables of our book entity class.

```
6 @Entity
7 public class Book {
8
9   @Id
10  private int id;
11  private String title;
12  private String author;
13  private float price;
14
15  public Book() {
16  }
17
18  public Book(int id, String title, String author, float price) {
19    this.id = id;
20    this.title = title;
21    this.author = author;
22    this.price = price;
23  }
24}
```

You can see that we have the **id** column that is an **integer**, it's **not null**. That's because we have specified the **id** as the primitive **int** type. The author and the title columns are both **String**. You can see that they've been defined by default as **varchar(255)**.

The Price column is of type **floating point**. It's not null once again, because it's the primitive type **float**. Because of the **@Id** annotation in the book id column, you can see that the book id has been designated the **primary key** of this table. The primary key reference is id.

If you scroll down further, you'll see other SQL statements that have been run under the hood by Hibernate.

```
Dec 01, 2021 12:56:00 PM org.hibernate.tool.schema.internal.S
INFO: HHH000476: Executing import script 'org.hibernate.tool.
Hibernate:
insert
into
Book
(author, price, title, id)
values
(?, ?, ?, ?)
Hibernate:
insert
into
Book
(author, price, title, id)
values
(?, ?, ?, ?)
Dec 01, 2021 12:56:00 PM org.hibernate.engine.jdbc.connection
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://
```

Here is an insert statement inserting the first book into this table, author, price, title and id. And if you scroll down further, you'll see the second insert statements as well. These insert statements correspond to the persist method that we invoke. This is our Java code on lines 27 and 28.

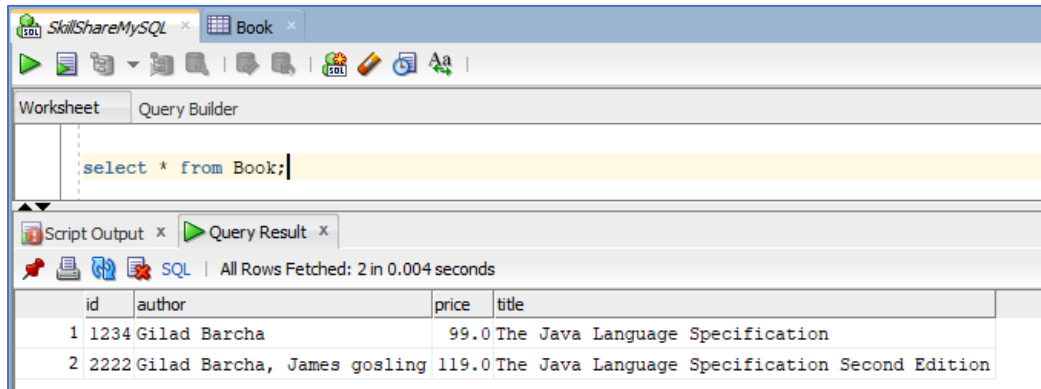
```
13 public static void main( String[] args )
14 {
15
16     EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17     EntityManager entityManager = factory.createEntityManager();
18
19     try {
20         entityManager.getTransaction().begin();
21
22         Book firstBook = new Book(1234, "The Java Language Specification",
23             "Gilad Barcha", 99f);
24         Book secondBook = new Book(2222, "The Java Language Specification Second Edition",
25             "Gilad Barcha, James gosling", 119f);
26
27         entityManager.persist(firstBook);
28         entityManager.persist(secondBook);
29
30     } catch (Exception ex) {
31         System.err.println("An error occurred: " + ex);
32     } finally {
33         entityManager.getTransaction().commit();
34         entityManager.close();
35         factory.close();
36     }
37 }
```

Now it seems like everything seems to have run through, but let's confirm this by switching over to MySQL workbench. I'm now going to describe bookstoredb.book. Allowing us to see whether this table has been created. Run this, execute the SQL command and you can see that this table is now present within our SQL database.

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
id	1	(null)	NO	int	10	0	(null)
author	2	(null)	YES	varchar	(null)	(null)	(null)
price	3	(null)	NO	float	12	(null)	(null)
title	4	(null)	YES	varchar	(null)	(null)	(null)

This table has the four columns that we expect, id, author, price and title. You can see the type of each of these columns also map correctly to our Book entity fields. You can see that id is the primary key.

Let's go ahead and run a select statement in order to see the records within this bookstore db. Run the SQL command and this will print out the two entities that we persisted using JPA and Hibernate. There are the two books here, Java Language Specification and Java Language Specification, the Second Edition.



The screenshot shows a MySQL IDE window with a tab titled 'SkillShareMySQL'. Inside, there's a 'Query Builder' tab and a 'Worksheet' tab. The 'Query Builder' tab is active, showing a SQL query: `select * from Book;`. Below the query, there's a 'Script Output' tab and a 'Query Result' tab. The 'Query Result' tab is active, showing the results of the query. The results are displayed in a table with 5 columns: 'id', 'author', 'price', and 'title'. There are 2 rows of data.

	id	author	price	title
1	1234	Gilad Barcha	99.0	The Java Language Specification
2	2222	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition

Configuring Database Actions

- Schema Generation Database Action Create

Just to remind you here is how we have set up our persistence.xml file. The **database.action** property has been set to the value, **create**.

```

6  <persistence-unit name="BookstoreDB_Unit" >
7    <class>com.mytutorial.jpa.Book</class>
8    <properties>
9      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10     <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11     <property name="javax.persistence.jdbc.user" value="root" />
12     <property name="javax.persistence.jdbc.password" value="password" />
13
14     <property name="javax.persistence.schema-generation.database.action" value="create"/>
15     <property name="hibernate.show_sql" value="true"/>
16     <property name="hibernate.format_sql" value="true"/>
17   </properties>
18 </persistence-unit>

```

This means that tables corresponding to your entities will be created if needed. Now, with this value for **javax.persistence.schema-generation.database.action** value="create", let's go ahead and run the same code once again using Run As Java Application. And observe that you can see a few exceptions within the console output.

```

INFO: HH000400: Using dialect: org.hibernate.dialect.MySQLDialect
Hibernate:
    create table Book (
      id integer not null,
      author varchar(255),
      price float not null,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 01, 2021 1:49:11 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HH010001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$Connecti
Dec 01, 2021 1:49:11 PM org.hibernate.tool.schema.internal.ExceptionHandlerLoggedImpl handleException
WARN: GenerationTarget encountered exception accepting command : Error executing DDL via JDBC Statement
org.hibernate.tool.schema.spi.CommandAcceptanceException: Error executing DDL via JDBC Statement
    at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:67)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.applySqlString(SchemaCreatorImpl.java:440)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.applySqlStrings(SchemaCreatorImpl.java:424)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.createFromMetadata(SchemaCreatorImpl.java:315)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.performCreation(SchemaCreatorImpl.java:166)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.doCreation(SchemaCreatorImpl.java:135)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.doCreation(SchemaCreatorImpl.java:121)
    at org.hibernate.tool.schema.spi.SchemaManagementToolCoordinator.performDatabaseAction(SchemaManagementToolCoordinator.java:129)
    at org.hibernate.tool.schema.spi.SchemaManagementToolCoordinator.process(SchemaManagementToolCoordinator.java:72)
    at org.hibernate.internal.SessionFactoryImpl.<init>(SessionFactoryImpl.java:313)
    at org.hibernate.boot.internal.SessionFactoryBuilderImpl.build(SessionFactoryBuilderImpl.java:452)
    at org.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderImpl.build(EntityManagerFactoryBuilderImpl.java:889)
    at org.hibernate.jpa.HibernatePersistenceProvider.createEntityManagerFactory(HibernatePersistenceProvider.java:58)
    at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:55)
    at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:39)
    at com.mytutorial.jpa.App.main(App.java:16)
Caused by: java.sql.SQLException: Table 'Book' already exists
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:120)
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:97)
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122)
    at com.mysql.cj.jdbc.StatementImpl.executeInternal(StatementImpl.java:764)
    at com.mysql.cj.jdbc.StatementImpl.execute(StatementImpl.java:648)
    at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:54)
    ... 15 more

```

The first exception here is because the table book already exists within our bookstore database. Because of the database action configuration setting set to **create**, Hibernate tries to create this book table. But the table already exist and that gives us this first exception.

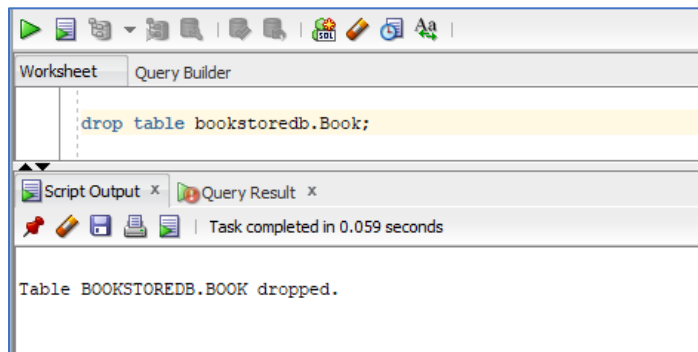
If you scroll down below, you'll see other exceptions here.

```
Hibernate:
insert
into
    Book
(author, price, title, id)
values
    (?, ?, ?, ?)
Dec 01, 2021 1:49:11 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper logExceptions
WARN: SQL Error: 1062, SQLState: 23000
Dec 01, 2021 1:49:11 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper logExceptions
ERROR: Duplicate entry '1234' for key 'PRIMARY'
Dec 01, 2021 1:49:11 PM org.hibernate.engine.jdbc.batch.internal.AbstractBatchImpl release
INFO: HHH000010: On release of batch it still contained JDBC statements
Dec 01, 2021 1:49:11 PM org.hibernate.internal.ExceptionMapperStandardImpl mapManagedFlushFailure
ERROR: HHH000346: Error during managed flush [org.hibernate.exception.ConstraintViolationException: could not execute statement]
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:77)
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:71)
    at com.mytutorial.jpa.App.main(App.java:33)
Caused by: javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException: could not execute statement
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:149)
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:157)
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:61)
    ... 2 more
Caused by: org.hibernate.exception.ConstraintViolationException: could not execute statement
    at org.hibernate.exception.internal.SQLExceptionTypeDelegate.convert(SQLExceptionTypeDelegate.java:59)
    at org.hibernate.exception.internal.StandardSQLExceptionConverter.convert(StandardSQLExceptionConverter.java:42)
    at org.hibernate.engine.jdbc.spi.SqlExceptionHelper.convert(SqlExceptionHelper.java:111)
    ... 1 more
Caused by: java.sql.SQLIntegrityConstraintViolationException: Duplicate entry '1234' for key 'PRIMARY'
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:117)
    at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:97)
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122)
```

And these other exceptions are because we try to re-insert the book entities, `firstBook` and `secondBook`.

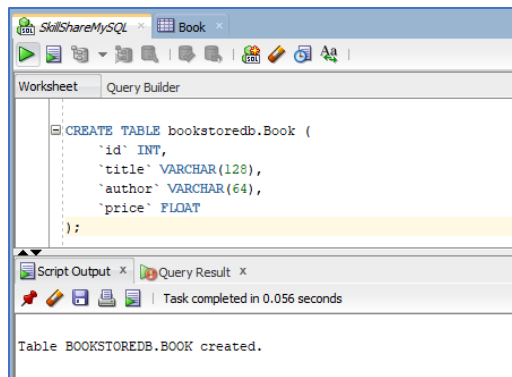
Books with these Ids already exist in our Book table. And that's why we get these additional constraint violation exceptions.

Let's head over to the MySQL Workbench and this time I'm going to drop the table that already exists here, drop table 'bookstoredb'. 'book'.



Once the table has been dropped, and you can see that the table drop action took place successfully here at the bottom of my screen. I'm now going to recreate the book table within this bookstore database with my own fields, and column specifications.

Java Persistence API: Getting Started With JPA & Hibernate



```
6 @Entity
7 public class Book {
8
9     @Id
10    private int id;
11    private String title;
12    private String author;
13    private float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24 }
```

Observe that my column names remain the same, id, title, author, and price. They correspond to the member variables of my book entity.

Observe that id is an integer. I don't have any not null constraints on any of these columns. And I haven't specified that id is the primary key for this table. That is the major difference. Go ahead and run this command using the lightning bolt icon, and this book table will be created within the bookstore database.

Now if you quickly run the describe book command for this book table, you'll see the constraints on the individual columns. All columns are nullable and there is no primary key.

The screenshot shows the 'Columns' tab for the 'Book' table in SQL Workbench. It displays a table with 8 columns: COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, NUMERIC_PRECISION, NUMERIC_SCALE, and COLUMN_COMMENT. The data for each column is as follows:

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
id	1	(null)	YES	int	10	0	(null)
title	2	(null)	YES	varchar	(null)	(null)	(null)
author	3	(null)	YES	varchar	(null)	(null)	(null)
price	4	(null)	YES	float	12	(null)	(null)

Remember that our persistence.xml still has database action set to `create`, but we no longer have the entries in the book table. The book table is now completely empty. Let's run this code and see what happens.

Java Persistence API: Getting Started With JPA & Hibernate

As you scroll down the console window, you'll see we still encounter the exception which says Table 'book' already exists.

```

Hibernate:
    create table Book (
        id integer not null,
        author varchar(255),
        price float not null,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 01, 2021 2:02:54 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHM100801501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$Connection
Dec 01, 2021 2:02:54 PM org.hibernate.tool.schema.internal.ExecutionHandlerLoggedImpl handleException
WARN: GenerationTarget encountered exception accepting command : Error executing DDL via JDBC Statement
org.hibernate.tool.schema.spi.CommandAcceptanceException: Error executing DDL via JDBC Statement
    at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:67)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.applySqlString(SchemaCreatorImpl.java:448)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.applySqlStrings(SchemaCreatorImpl.java:424)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.createFromMetadataData(SchemaCreatorImpl.java:315)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.performCreation(SchemaCreatorImpl.java:166)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.doCreation(SchemaCreatorImpl.java:135)
    at org.hibernate.tool.schema.internal.SchemaCreatorImpl.doCreation(SchemaCreatorImpl.java:121)
    at org.hibernate.tool.schema.spi.SchemaManagementToolCoordinator.performDatabaseAction(SchemaManagementToolCoordinator.java:129)
    at org.hibernate.tool.schema.spi.SchemaManagementToolCoordinator.process(SchemaManagementToolCoordinator.java:72)
    at org.hibernate.internal.SessionFactoryImpl.<init>(SessionFactoryImpl.java:313)
    at org.hibernate.boot.internal.SessionFactoryBuilderImpl.build(SessionFactoryBuilderImpl.java:452)
    at org.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderImpl.build(EntityManagerFactoryBuilderImpl.java:889)
    at org.hibernate.jpa.HibernatePersistenceProvider.createEntityManagerFactory(HibernatePersistenceProvider.java:58)
    at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:55)
    at javax.persistence.Persistence.createEntityManagerFactory(Persistence.java:39)
    at com.mytutorial.jpa.App.main(App.java:16)
Caused by: java.sql.SQLSyntaxErrorException: Table 'Book' already exists
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException:120)
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException:97)
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122)
    at com.mysql.cj.jdbc.StatementImpl.executeInternal(StatementImpl.java:764)
    at com.mysql.cj.jdbc.StatementImpl.execute(StatementImpl.java:648)
    at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase.java:54)
... 13 more

```

Hibernate tries to recreate this table and that fails. But this time around, we don't encounter the exception when we try to insert the books into this table. The table started off empty. So our insertion statements for the book with id 1234, and the book with id 2222, were both successful.

```
Dec 01, 2021 2:02:54 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HH0000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@5cc126dc'
Hibernate:
insert
into
    Book
    (author, price, title, id)
values
    (?, ?, ?, ?)
Hibernate:
insert
into
    Book
    (author, price, title, id)
values
    (?, ?, ?, ?)
Dec 01, 2021 2:02:54 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HH010001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

Switching over to the MySQL Workbench. If you now run a `select *` from the book table, you will find that the two books have been successfully inserted in here, Java language specification and the second edition as well.

The screenshot shows the IntelliJ IDEA IDE interface. At the top, there's a toolbar with various icons. Below it, the 'Worksheet' tab is active, displaying a SQL query in the 'Query Builder' pane:

```
SELECT * FROM bookstoredb.Book ;
```

Below the query, the 'Script Output' and 'Query Result' panes are visible. The 'Query Result' pane shows the results of the query, indicating 'All Rows Fetched: 2 in 0.003 seconds'.

id	title	author	price
1 1234	The Java Language Specification	Gilad Barcha	99.0
2 2222	The Java Language Specification Second Edition	Gilad Barcha, James gosling	119.0

- Schema Generation Database Action None

Now it's quite possible that you have an already existing table within your database, and you want to work with that table using JPA. In that case you'll head over to **persistence.xml**, and change your configuration setting here.

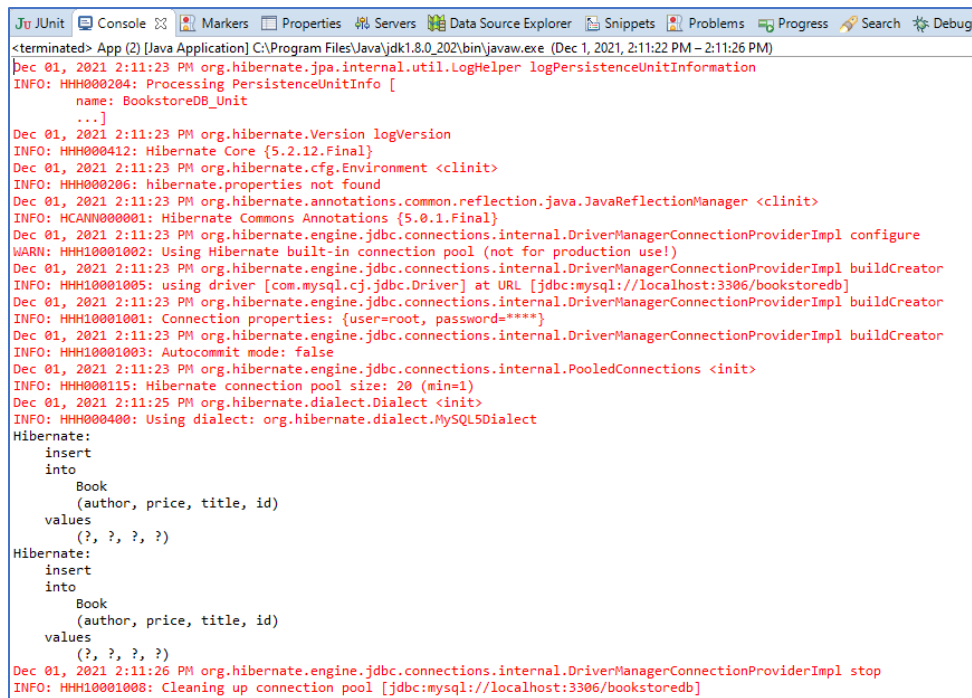
Instead of using the **create** action for your database, you'll set the value of your database action to be equal to **none**.

A value of **none** here indicates to Hibernate that it should not try and create tables within your database. It should only perform the CRUD operations or query operations that you run within Hibernate.

```
6  <persistence-unit name="BookstoreDB_Unit" >
7    <class>com.mytutorial.jpa.Book</class>
8    <properties>
9      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10     <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11     <property name="javax.persistence.jdbc.user" value="root" />
12     <property name="javax.persistence.jdbc.password" value="password" />
13
14     <property name="javax.persistence.schema-generation.database.action" value="none"/>
15     <property name="hibernate.show_sql" value="true"/>
16     <property name="hibernate.format_sql" value="true"/>
17   </properties>
18 </persistence-unit>
```

This value of **none** is what you'll use when you work with database tables that have been pre-created, and maybe even pre-populated.

Now save your **persistence.xml** file and let's head over to App.java. I'm going to run this App.java code which will try and create two entries in my book table.



```
<terminated> App (2) [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Dec 1, 2021, 2:11:26 PM)
Dec 01, 2021 2:11:23 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
    name: BookstoreDB_Unit
    ...]
Dec 01, 2021 2:11:23 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.2.12.Final}
Dec 01, 2021 2:11:23 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Dec 01, 2021 2:11:23 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCAN000001: Hibernate Commons Annotations {5.0.1.Final}
Dec 01, 2021 2:11:23 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
Dec 01, 2021 2:11:23 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/bookstoredb]
Dec 01, 2021 2:11:23 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=root, password=****}
Dec 01, 2021 2:11:23 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
Dec 01, 2021 2:11:23 PM org.hibernate.engine.jdbc.connections.internal.PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Dec 01, 2021 2:11:25 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
insert
into
  Book
(author, price, title, id)
values
(?, ?, ?, ?)
Hibernate:
insert
into
  Book
(author, price, title, id)
values
(?, ?, ?, ?)
Dec 01, 2021 2:11:26 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

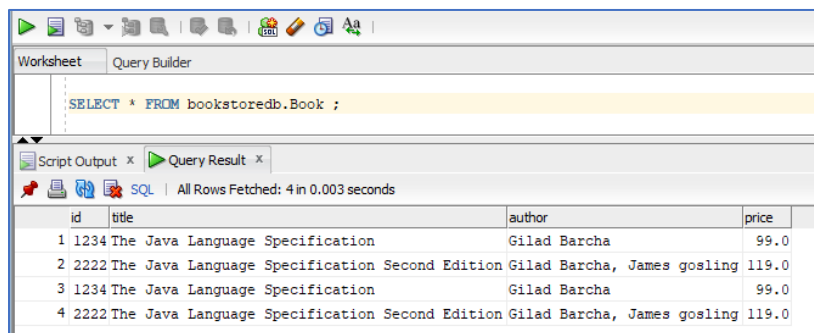
The book table already exists in my database, and it has already been populated with these two entries with the same ID.

```
15
16 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17 EntityManager entityManager = factory.createEntityManager();
18
19 try {
20     entityManager.getTransaction().begin();
21
22     Book firstBook = new Book(1234, "The Java Language Specification",
23                             "Gilad Barcha", 99f);
24     Book secondBook = new Book(2222, "The Java Language Specification Second Edition",
25                               "Gilad Barcha, James gosling", 119f);
26
27     entityManager.persist(firstBook);
28     entityManager.persist(secondBook);
29
30 } catch (Exception ex) {
31     System.err.println("An error occurred: " + ex);
32 } finally {
33     entityManager.getTransaction().commit();
34     entityManager.close();
35     factory.close();
36 }
```

But observe in the console, there is no exception, and everything seems to have run through successfully.

Now this was because when we created the book table earlier manually using the MySQL Workbench, we had not specified Id as the primary key.

If you run a select statement from the book table and run this command, you'll find that we have four entries here.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the SQL query: `SELECT * FROM bookstoredb.Book ;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 rows and 4 columns: id, title, author, and price. The first two rows represent the original entries, and the next two rows represent the new entries added by the Java code.

id	title	author	price
1 1234	The Java Language Specification	Gilad Barcha	99.0
2 2222	The Java Language Specification Second Edition	Gilad Barcha, James gosling	119.0
3 1234	The Java Language Specification	Gilad Barcha	99.0
4 2222	The Java Language Specification Second Edition	Gilad Barcha, James gosling	119.0

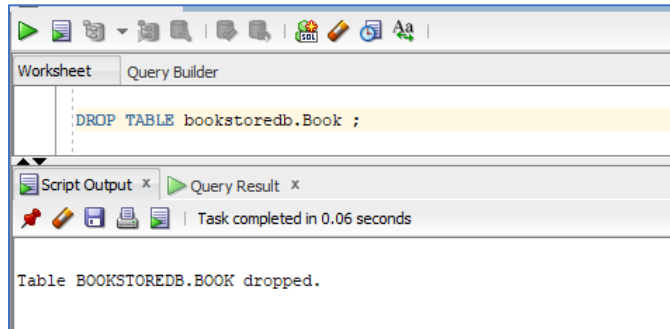
The original two entries which already existed in the table. The table wasn't recreated and we added two more entries with the same Id values. When we set up the database manually, Id was not a primary key. So if Id is not originally a primary key, Hibernate does not enforce it as a primary key.

“Hibernate adds the primary key constraint for Id fields within your entities only if Hibernate is the one creating the table”.

This is something you need to watch out for.

Java Persistence API: Getting Started With JPA & Hibernate

Here from within my MySQL Workbench, I'm going to drop the book table from my bookstore database.



So I've manually dropped the table, the table will no longer exist in the database once this command has run through. This command has run through successfully.

Back to Eclipse and our JPA Hibernate application. Remember our database action property has been set to `none`. That means tables will not be created by our Hibernate implementation. It'll work with tables that already exist, but it won't create database tables. Our database action is none, and the book table does not exist in our database.

So when we run this code, you'll encounter an exception.

```

Hibernate:
insert
into
  Book
(author, price, title, id)
values
  (?, ?, ?, ?)
Dec 01, 2021 2:22:39 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper logExceptions
WARN: SQL Error: 1146, SQLState: 42S02
Dec 01, 2021 2:22:39 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper logExceptions
ERROR: Table 'bookstoredb.Book' doesn't exist
Dec 01, 2021 2:22:39 PM org.hibernate.engine.jdbc.batch.internal.AbstractBatchImpl release
INFO: HH000010: On release of batch it still contained JDBC statements
Dec 01, 2021 2:22:39 PM org.hibernate.internal.ExceptionMapperStandardImpl mapManagedFlushFailure
ERROR: HH000346: Error during managed flush [org.hibernate.exception.SQLGrammarException: could not execute statement]
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:77)
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:71)
    at com.mytutorial.jpa.App.main(App.java:33)
Caused by: javax.persistence.PersistenceException: org.hibernate.exception.SQLGrammarException: could not execute statement
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:149)
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:157)
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:61)
    ... 2 more
Caused by: org.hibernate.exception.SQLGrammarException: could not execute statement
    at org.hibernate.exception.internal.SQLExceptionTypeDelegate.convert(SQLExceptionTypeDelegate.java:63)
    at org.hibernate.exception.internal.StandardSQLExceptionConverter.convert(StandardSQLExceptionConverter.java:42)
    at org.hibernate.engine.jdbc.spi.SqlExceptionHelper.convert(SqlExceptionHelper.java:111)
    ... 1 more
Caused by: java.sql.SQLException: Table 'bookstoredb.Book' doesn't exist
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:120)
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:97)
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedStatement.java:953)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1092)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1040)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeLargeUpdate(ClientPreparedStatement.java:1347)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedStatement.java:1025)
```

The exception clearly says, `bookstoredb.book` doesn't exist.

So if you have database action set to `none`, make sure the tables that you're working with already exist in your database.

- Schema Generation Database Action Drop and Create

Let's see another configuration for the database action that will prove very useful when you're running automated tests or developing prototypes, like we are in our demos here. I'm going to update the database action value and set it to **drop-and-create**.

```

6  <persistence-unit name="BookstoreDB_Unit" >
7    <class>com.mytutorial.jpa.Book</class>
8    <properties>
9      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10     <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11     <property name="javax.persistence.jdbc.user" value="root" />
12     <property name="javax.persistence.jdbc.password" value="password" />
13
14     <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
15     <property name="hibernate.show_sql" value="true"/>
16     <property name="hibernate.format_sql" value="true"/>
17   </properties>
18 </persistence-unit>

```

For all of the database tables that you've specified as entities within your JPA Hibernate application, **drop-and-create** will drop those tables at the very beginning when you run your application.

So before any code is executed, the tables that you've configured as entities will be dropped. And then will be recreated each time you run your application. This means each time you run your application, you get a fresh start, which is great for running automated tests.

After having updated our database action to **drop-and-create**, let's head over to App.java and run this Java application. And when we run this, you'll notice something interesting within the console window.

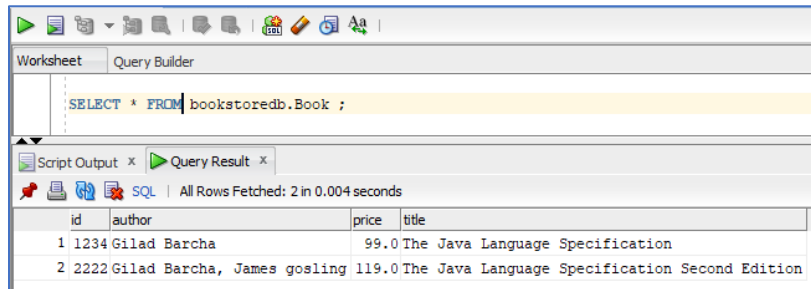
```

INFO: HH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    drop table if exists Book
Dec 01, 2021 2:26:22 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HH0001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionPool$ConnectionImpl$DefaultImplt@6256ac4f]
Hibernate:
    create table Book (
      id integer not null,
      author varchar(255),
      price float not null,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 01, 2021 2:26:22 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HH0001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionPool$ConnectionImpl$DefaultImplt@6256ac4f]
Dec 01, 2021 2:26:22 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@6256ac4f'
Hibernate:
    insert
    into
      Book
      (author, price, title, id)
    values
      (?, ?, ?, ?)
Hibernate:
    insert
    into
      Book
      (author, price, title, id)
    values
      (?, ?, ?, ?)
Dec 01, 2021 2:26:22 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HH0001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]

```

Observe that Hibernate first tries to **drop** the table Book **if the table exists**. If the table does not exist, it won't do anything. After dropping an existing table, it'll recreate the table Book based on the entity specifications for the Book class. id here will be the primary key, not null integer.

And after it has created the Book table it'll insert the entities that we have specified within our App.Java code. We can confirm that things have run through successfully by running a select * from our book table.



The screenshot shows the MySQL Workbench Query Builder interface. The query 'SELECT * FROM bookstoredb.Book ;' is entered in the SQL editor. Below the editor, the 'Query Result' tab is active, displaying a table with 4 columns: id, author, price, and title. Two rows of data are shown.

id	author	price	title
1 1234	Gilad Barcha	99.0	The Java Language Specification
2 2222	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition

Run this code, and you'll see that the book table has been successfully created. And our two book entries have been successfully inserted into this table.

Back to our Eclipse IDE App.Java, I'm going to run this application once again. Remember, our database action has been set to [drop-and-create](#), which means each time we run this application, the table Book if it exists will be dropped, and then recreated, and the entries that we have specified will be reinserted.

```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    drop table if exists Book
Dec 01, 2021 2:28:26 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionPool$ConnectionPoolImpl$JdbcConnectionAccess@1536602f]
Hibernate:
    create table Book (
      id integer not null,
      author varchar(255),
      price float not null,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 01, 2021 2:28:26 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionPool$ConnectionPoolImpl$JdbcConnectionAccess@1536602f]
Dec 01, 2021 2:28:26 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@1536602f'
Hibernate:
    insert
    into
      Book
      (author, price, title, id)
    values
      (?, ?, ?, ?)
Hibernate:
    insert
    into
      Book
      (author, price, title, id)
    values
      (?, ?, ?, ?)
Dec 01, 2021 2:28:26 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

Lets head over to the MySQL Workbench to double check that everything works just fine. So drop-and-create is a great setting for automated tests and for testing out your prototypes.

Drop and Create Actions Using Scripts

We've seen that the drop and create database action is extremely useful when you want to recreate your tables each time you run your JPA Hibernate application.

Now, **drop-and-create** can be made even more useful by specifying some additional parameters. Here we've specified a script that we want to run each time we create our tables as well as when we drop our tables.



We have a

```
<property name="javax.persistence.schema-generation.create-source" value="script"/>
```

that we've set to the value script, indicating that a script should be run.

And we'll specify the script that we want executed using

```
<property name="javax.persistence.schema-generation.create-script-source"
value="META-INF/create-book.sql"/>
```

This create-script-source points to *META-INF/create-book.sql*.

Similarly, the

```
<property name="javax.persistence.schema-generation.drop-source" value="script"/>
```

we've specified as value equal to script. Which means when we drop our tables, we'll execute a script.

And the

```
<property name="javax.persistence.schema-generation.drop-script-source" value="META-
INF/drop-book.sql"/>
```

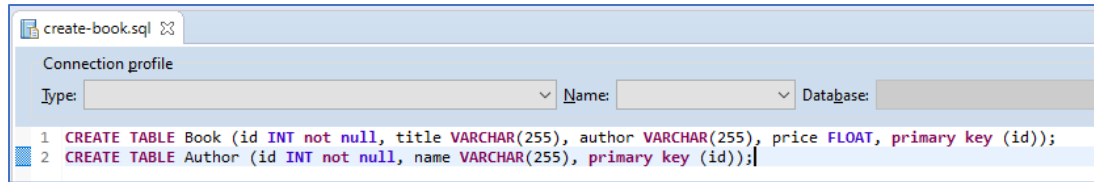
is *META-INF/drop-book.sql*.

Rather than using the entity table mapping specified within our Java code to create tables and drop tables, these scripts will be used instead, *create-book.sql* and *drop-book.sql*.

Now let's go ahead and specify these scripts.

The first filename that I specify is ***create-book.sql***, which will contain my creation scripts. This is a script that will run when I first load up my application.

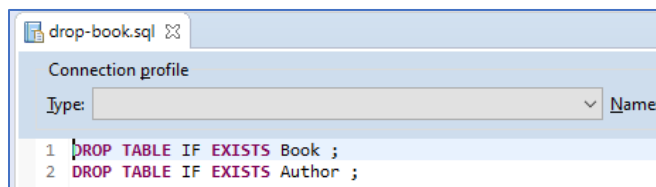
Java Persistence API: Getting Started With JPA & Hibernate



This is the script file where you'll set up your SQL commands to create the database you want to work with within your application. I've created two tables here, the book table and the author table.

The Java files that we are working with have just one entity specified, the book entity corresponding to the book table. We don't have an author entity yet. But I've created two tables, one corresponding to book and another corresponding to author.

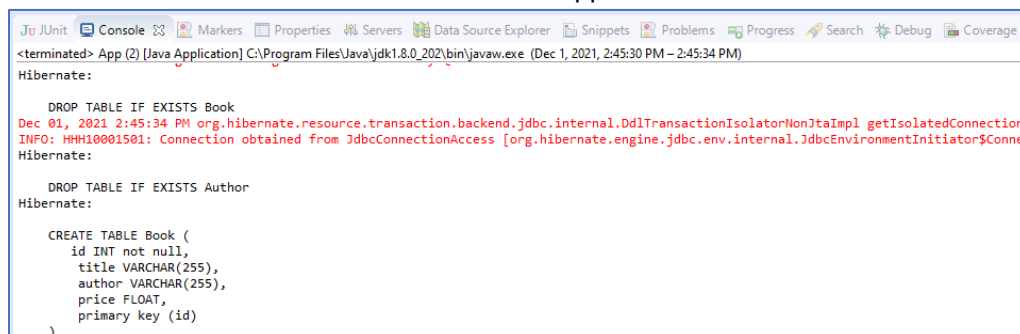
Once again, This is the second SQL File containing a SQL script. This will be the **drop-book.sql** file. This will contain the SQL commands that we'll run when we want to start afresh when we run our application.



This is the cleanup script that will clean up any existing tables that you already have within your database before your create script creates new ones. So I'm going to drop the bookstoredb book as well as author tables here in my **drop-book.sql** file.

Make sure you save all of the files that you've created. Let's head over to App.java and run our code. Thanks to our configuration in the **persistence.xml** file, the drop script will be executed first, and then the create script.

Let's take a look at the output on the console window. If you scroll down, you'll find that the drop SQL commands have been executed first. We've dropped the book table as well as the author tables.



And once the tables have been dropped, we've recreated those. Here are the commands from the create script creating our book table.

And if you scroll further down, you'll see the second command that creates the author table.

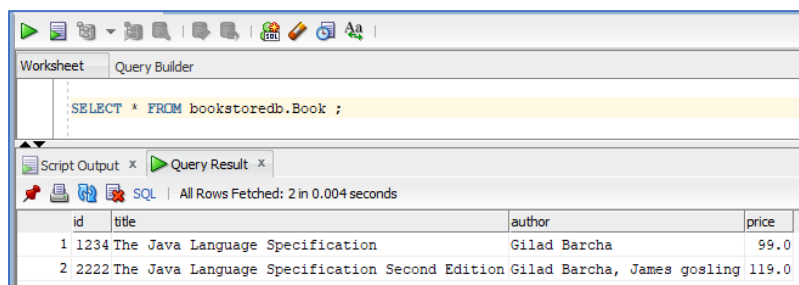
```
Dec 01, 2021 2:45:34 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNo
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.
Hibernate:
CREATE TABLE Author (
  id INT not null,
  name VARCHAR(255),
  primary key (id)
)
```

Notice that we don't have the author.java entity yet in our Java code, yet both of those tables have been created.

```
Dec 01, 2021 2:45:34 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH1000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInput
Hibernate:
insert
into
  Book
(author, price, title, id)
values
  (?, ?, ?, ?)
Hibernate:
insert
into
  Book
(author, price, title, id)
values
  (?, ?, ?, ?)
Dec 01, 2021 2:45:34 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProv
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

And the insert statements from our App.java code have been executed and two book entries are present now in our book table.

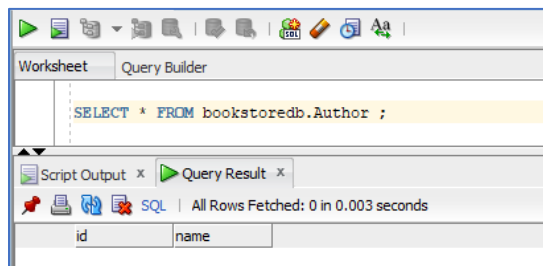
This is something that is easily confirmed using the MySQL Workbench. I'm going to run a select * command to select all records from the book table. Once this command runs through, you can see that the two book entries that we persisted using JPA code are now present in this table.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.Book ;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with four columns: id, title, author, and price. There are two rows of data.

id	title	author	price
1 1234	The Java Language Specification	Gilad Barcha	99.0
2 2222	The Java Language Specification Second Edition	Gilad Barcha, James gosling	119.0

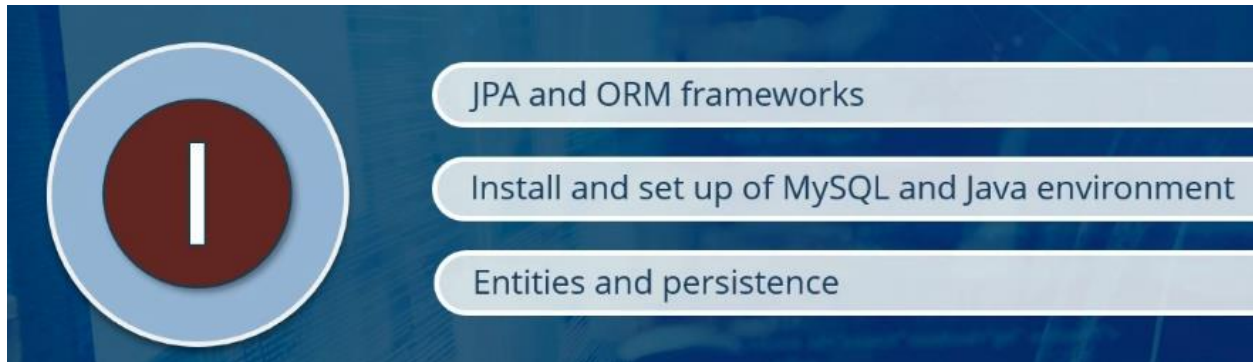
We can also confirm here that the author table has been created. I'm going to run a describe command on the author table. And once this command is executed, you'll see that the author table with two columns exists in our database thanks to our *create-book.sql* script.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.Author ;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with two columns: id and name.

id	name
----	------

Summary



In this course, we introduce the basic idea behind object-relational mapping. This is what allows entities and relationships, expressed using a high level object oriented programming language such as Java, to work with the fundamental building blocks of relational databases, that is records, tables, and constraints.

We started off by discussing the Java Persistence API specification of object relational mapping providers and its intertwined history with Hibernate which is a specific implementation of the JPA. JPA today is implemented by many persistence providers. One example beyond Hibernate is EclipseLink which also supports NoSQL databases.

We then got hands on and got our local machine environment set up to run our JPA applications. We install the MySQL database along with the graphical user interface for it, the MySQL Workbench. We also set up our Java project with our dependencies and builds managed using Apache Maven. We set up Hibernate which is part of the spring framework as our JPA provider. We finally got our first peek into using JPA by creating Java classes that represent persistent entities.

We use the **persistence.xml** file to connect to the underlying MySQL database to run queries, and persist our objects.