Practical 04

Project 02 - Spring Web Application (Configuring Hibernate)

‼️Important Note (Before Attempting this Project)

1. Make sure you complete **Practical 03 - Project 01** because this is a continuation from that project.
2. Since this is a continuation from the previous project, the Git remote is pointing to Practical 03 Github classroom. **You need to change the Git remote to Practical 04 Github classroom**
    1. Check the Git remote name in your repository.

        ```
        git remote -v
        ```

        The output will look something like this:

        ```
        origin  https://github.com/user/repo_name.git (fetch)
        origin  https://github.com/user/repo_name.git (push)
        ```

    2. Assuming the remote name is `origin`. Remove the remote URL pointing to the Practical 03 Github classroom based on the remote name. **If your remote name is different, change `origin` to the appropriate remote name stated in `git remote -v`**.

        ```
        git remote remove origin
        ```

3. Go to Practical 04 Github classroom repository and add a new remote URL to the project. **Make sure the URL is for Practical 04, you can accept the invitation to Practical 04 in PBLMS**.

```
git remote add origin <URL>
```

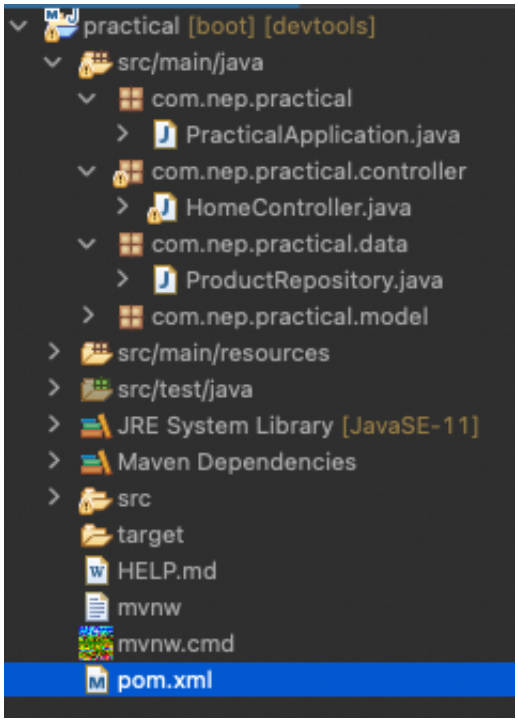
❗ Change `<URL>` to your specific Practical 04 URL.

4. Double check the URL and you can try `git push origin` to the new URL.
5. Then got to your Practical 04 Github classroom repository to check if the files are uploaded there.

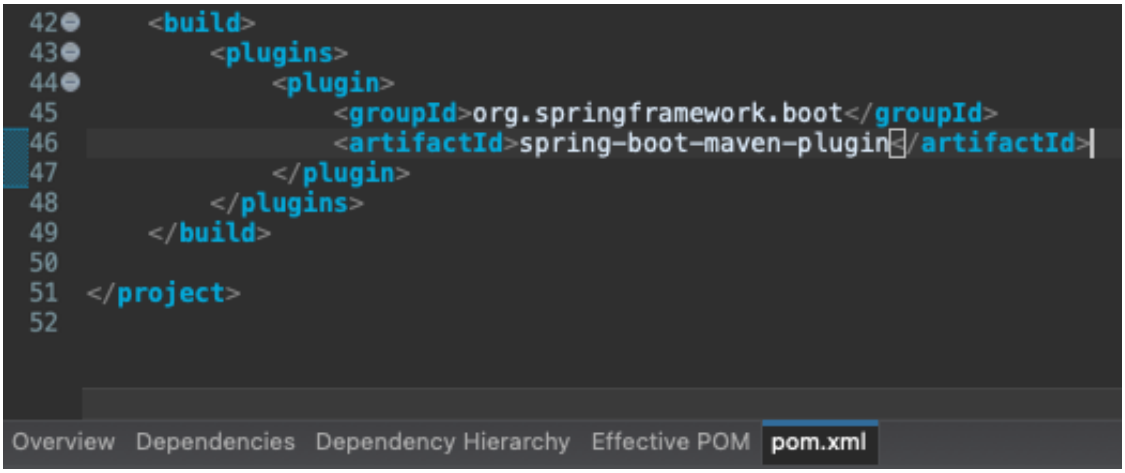
Lesson 2.1 - Setting Up Spring Data JPA and MySQL Connector Java

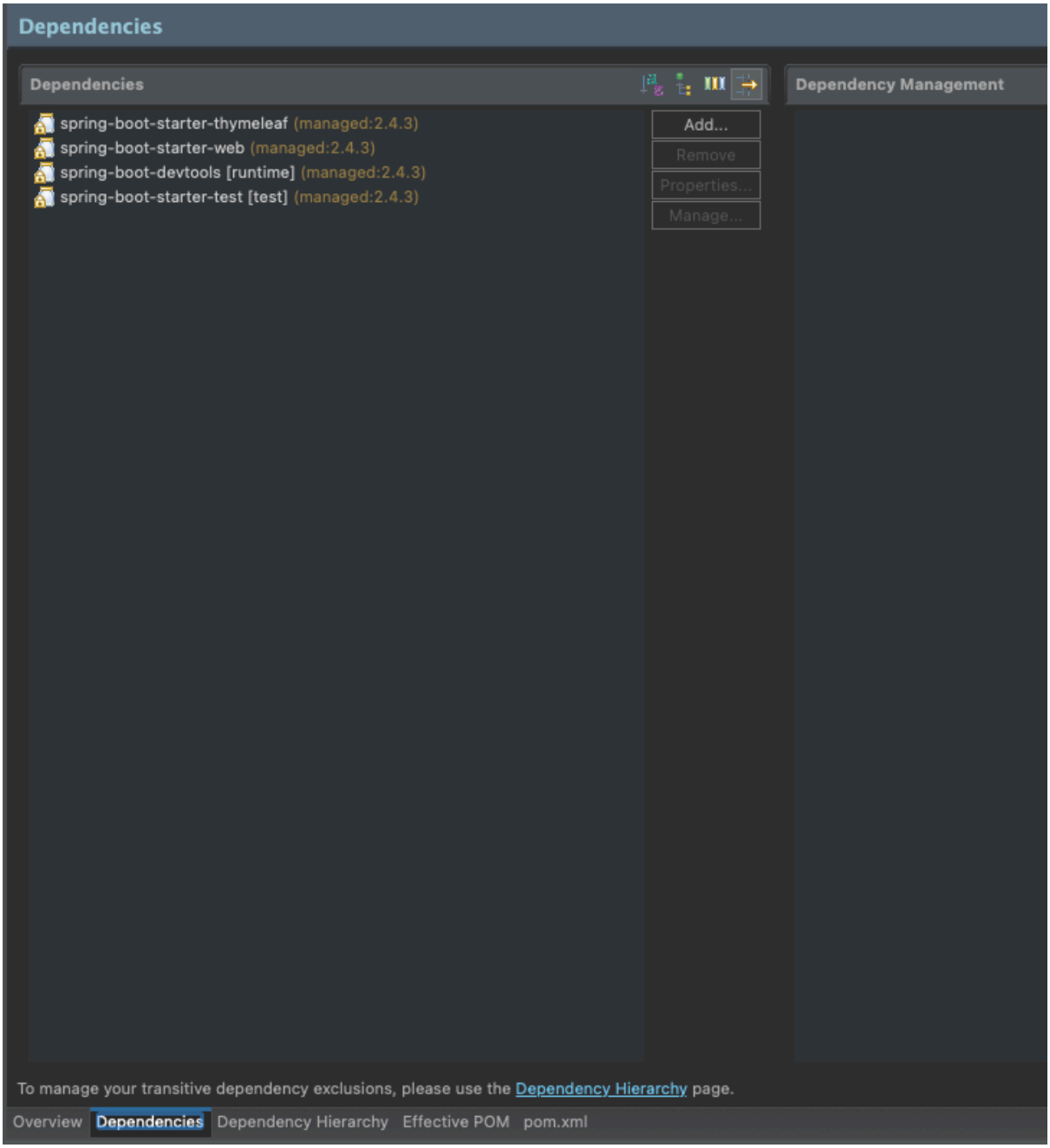
‼️Important Note: Make sure your computer is connected to the internet.

1. First we need to add two more dependencies to the application. These two need to be added to the pom.xml dependencies.
    1. Spring Data JPA: part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This will significantly improve the implementation of data access layers by reducing the effor to the amount that is actually needed.
    2. MySQL Connector Java: Add JDBC driver for MySQL to your application. JDBC provides Java programmers with a uniform interface for accessing and manipulating relational databases.

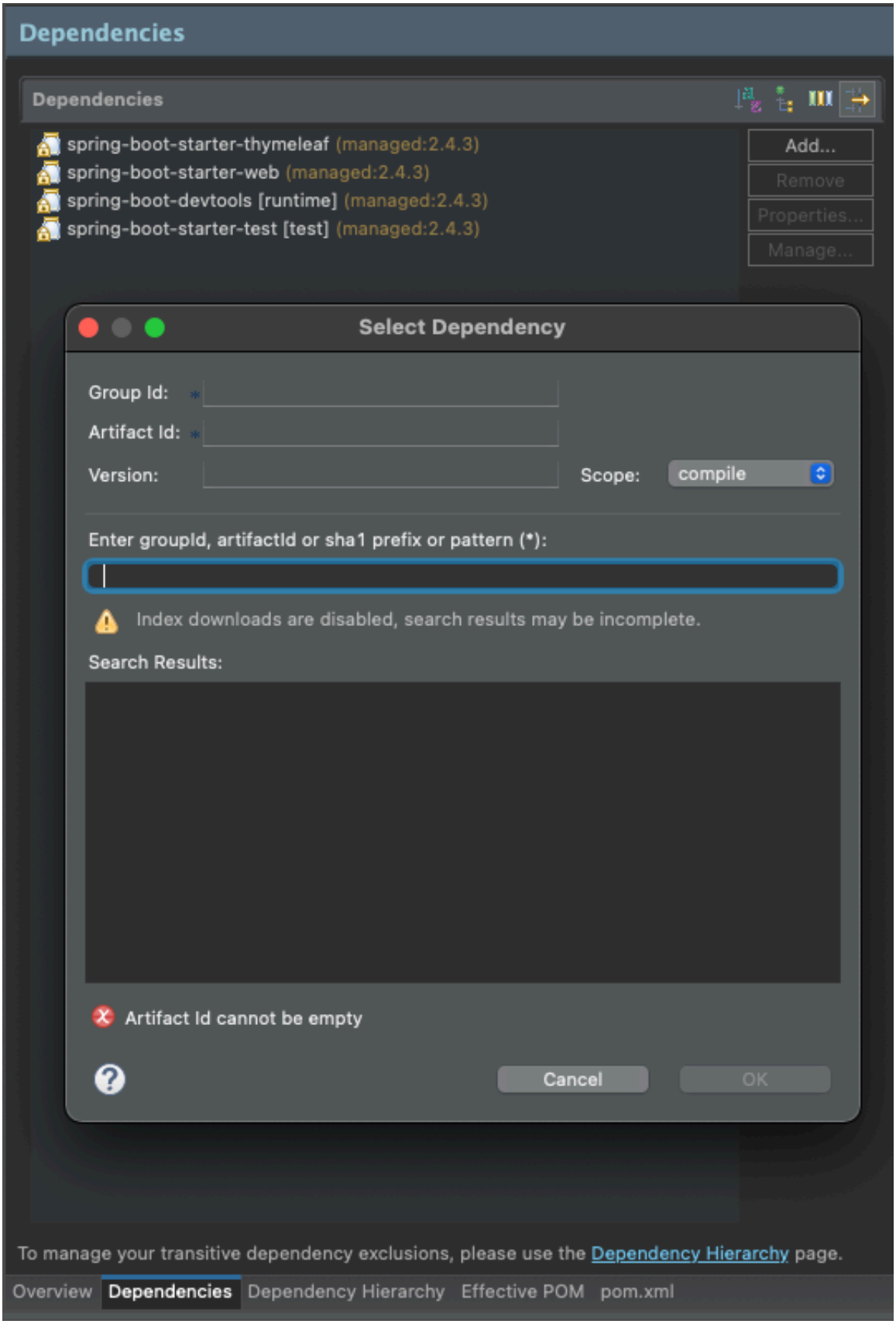


2. Instead of directly editing the xml file. It is recommended to use the user interface provided in Eclipse to add the dependencies. Open `pom.xml` and click the `Dependencies` tab.



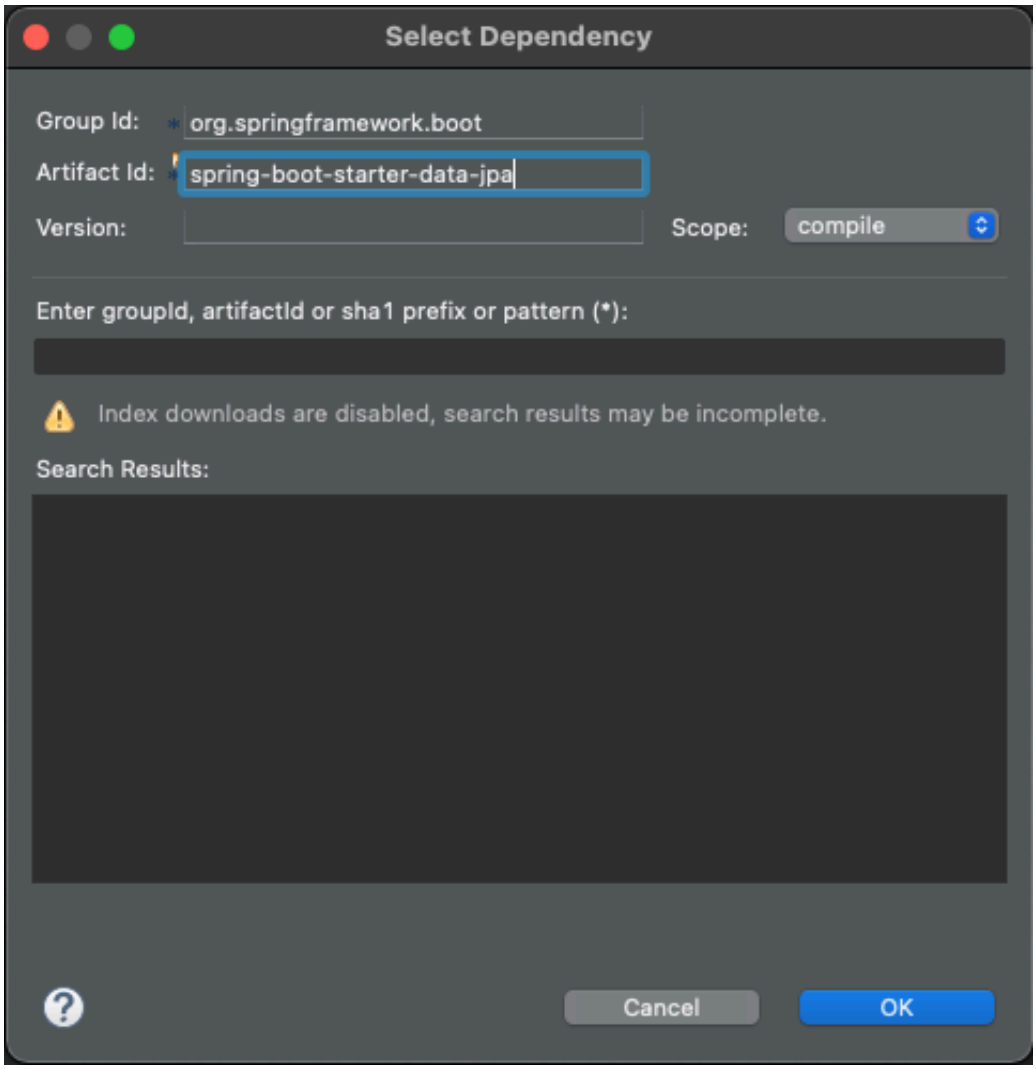


3. Click the `Add` button under the Dependencies column. A window should appear. Here is where you need to fill in the details for Spring Data JPA and MySQL Connector Java.



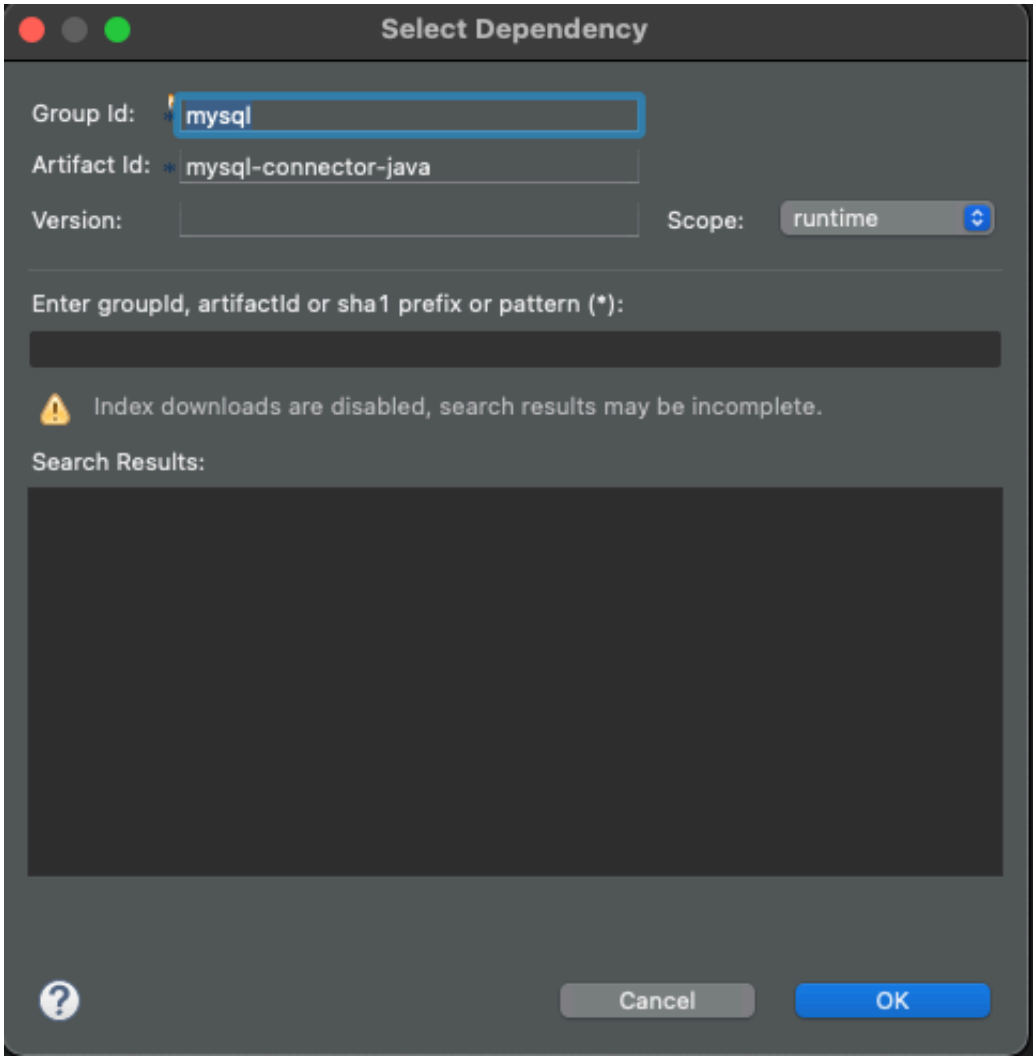
4. Lets add Spring Data JPA. These are details you need to fill in. Then press the Ok button.

```
Group Id: org.springframework.boot
Artifact Id: spring-boot-starter-data-jpa
Scope: compile
```

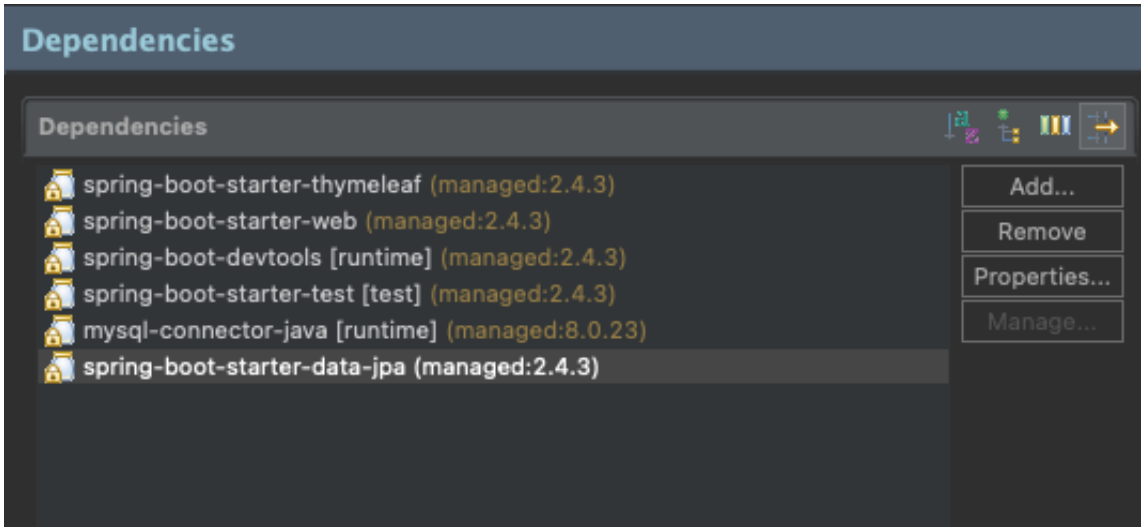


5. Then add MySQL Connector Java.

```
Group Id: mysql
Artifact Id: mysql-connector-java
Scope: runtime
```



6. Make sure you have internet, then save `pom.xml' file. It will automatically download the dependencies.



7. Wait for it to complete downloading the dependencies before continuing.

> # ⚠️ Milestone:
>
> Commit the changes made to the project folder with message "Lesson 2.1 - Setting Up Spring Data JPA and MySQL Connector Java".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 2.1 - Setting Up Spring Data JPA and MySQL Connector Java"`.

Lesson 2.2 - Setting Up Spring Web Application to Connect with Database (using Hibernate)

1. To make the Spring Web Application to connect with MySQL database. Open `application.properties`. Depending on your configuration in Practical 03 - Project 01. You might have a property for setting the port number. If that the case, just add the following below it. We need to set up the following properties:
    1. spring.datasource.driver-class-name - The driver that will be used to connect to a database.
    2. spring.datasource.url - URL location the database is hosted and connect to that database.
    3. spring.datasource.username - Username used to connect to the database server
    4. spring.datasource.password - Password used to connect to the database server

    ```
    spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
    spring.datasource.url=jdbc:mysql://<MySQL Server URL>/<MySQL Database Name>
    spring.datasource.username=<MySQL account name>
    spring.datasource.password=<MySQL account password>
    ```

    NOTE: If you're using this module's allocated server ~~`jailanirahman.com`~~. These are the default setting.

    `database.jailanirahman.com`

    ```
    spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
    spring.datasource.url=jdbc:mysql://database.jailanirahman.com/<Your Student ID>
    spring.datasource.username=<Your Student ID>
    spring.datasource.password=<Your Password>
    ```

2. Next we need to set the behaviour of Spring Data JPA. The properties are:
    1. spring.jpa.hibernate.ddl-auto - What you want to do to the tables in the database, everytime you start
       the web application. The value can be:
        1. create - creates the database tables, destroying previous data.
        2. create-drop - drop the database tables when the application is stopped.

    3. update - update the database tables if new columns or constraints added, but will not remove existing columns.
    4. validate - validate the database tables, makes no changes to the database.
2. spring.jpa.show-sql - To show or not the sql statement in the console.
3. spring.jpa.properties.hibernate.dialect - To tell what kind of SQL statement Hibernate should use to access and manipulate data in the database.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://<MySQL Server URL>/<MySQL Database Name>
spring.datasource.username=<MySQL account name>
spring.datasource.password=<MySQL account password>

spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

NOTE:

- When designing your Spring Model (to represent database table), it is recommended to set `spring.jpa.hibernate.ddl-auto` to `create-drop`. Once you are satisfied with the design, change it to `create`. Then onwards set it to `update`. Finally when it is in production set it to `validate`.
- Additionally when it is in production set `spring.jpa.show-sql` to `false`.

3. Run the application to make sure your configuration is correct. If the application runs appropriately, that means you have configured them correctly.

> ## ⚠️ Milestone:
>
> Commit the changes made to the project folder with message "Lesson 2.2 - Setting Up Spring Web Application to Connect with Database (using Hibernate)".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 2.2 - Setting Up Spring Web Application to Connect with Database (using Hibernate)"`.

Lesson 2.3 - Make Spring Model as Database Entity

1. From previous practical we have created Spring Model called `Product.java`.

```
package com.nep.practical.model;
```

```java
public class Product {

  private String name;
  private double price;
  private String file;
  private boolean inStock;

  public Product(String name, double price, String file, boolean inStock) {
    this.name = name;
    this.price = price;
    this.file = file;
    this.inStock = inStock;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public double getPrice() {
    return price;
  }

  public void setPrice(double price) {
    this.price = price;
  }

  public String getFile() {
    return file;
  }

  public void setFile(String file) {
    this.file = file;
  }

  public boolean isInStock() {
    return inStock;
  }

  public void setInStock(boolean inStock) {
    this.inStock = inStock;
  }

}
```

2. Before we make this as a Database Entity, we need to add further implementation to this Spring Model Class. We need to implement a no-arg constructor. We don't need to implement anything inside the constructor.

```
package com.nep.practical.model;

public class Product {
  ...
  private boolean inStock;

  public Product() { }

  public Product(String name, double price, String file, boolean inStock) {
...

}
```

3. Next we need to add additional annotations to the Spring Model Class to make it into a Database Entity. Make sure you import them from `javax.persistence` package.

```
package com.nep.practical.model;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="products")
public class Product {
...
}
```

4. We need to identify the Primary Key (column that make the data unique) in the Spring Model Class. Lets add a new field into the Spring Model Class to represent a running number for identifying products. We need to add getter for this.

```
package com.nep.practical.model;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="products")
public class Product {

  private int id;
```

```
  private String name;
  ...

  public Product(String name, double price, String file, boolean inStock) {
    ...
  }

  public int getId() {
    return id;
  }

  public String getName() {
    ...
}
```

5. Since the newly added field will be a Primary Key, add annotation `@Id` on top of it and since we want it to be auto incremented, we need to set annotation `@GeneratedValue(strategy=GenerationType.IDENTITY)`. The strategy means it will auto increment. Make sure you import them from `javax.persistence` package.

```
package com.nep.practical.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="products")
public class Product {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  ...
}
```

6. When the Application started, in the console you will see. You will see `id` is set as primary key and auto increment.

```
Hibernate: create table products (id integer not null auto_increment, file
varchar(255), in_stock bit not null, name varchar(255), price double
precision not null, primary key (id)) engine=InnoDB
```

7. As you can see, `file` and `name` can store null value. If you want to prevent null value to a column. You can set it using `@Column(nullable=false)`.

```
package com.nep.practical.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="products")
public class Product {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  @Column(nullable=false)
  private String name;
  private double price;
  @Column(nullable=false)
  private String file;
  private boolean inStock;

  ...
}
```

8. When the Application started, in the console you will see `file` and `name` set to not null.

```
Hibernate: create table products (id integer not null auto_increment, file
varchar(255) not null, in_stock bit not null, name varchar(255) not null,
price double precision not null, primary key (id)) engine=InnoDB
```

9. Sometimes you want the double data to be precise. Lets say the price of each product should not be more than $ `100,000.00`. That means the maximum $ `99,999.99` that means it should contain 7 digits including the decimal values and it decimal value should only be 2 decimal values. That means `precision=7` and `scale=2`. Since the data type for price is double, we need to convert this in MySQL as `DECIMAL`. So that we can set the precision.

```
...
public class Product {

  ...
  private String name;
  @Column(columnDefinition="DECIMAL(7,2) NOT NULL")
  private double price;
  ...

  ...
}
```

10. This will set the create the table as follows:

```
Hibernate: create table products (id integer not null auto_increment, file
varchar(255) not null, in_stock bit not null, name varchar(255) not null,
price DECIMAL(7,2) NOT NULL, primary key (id)) engine=InnoDB
```

    In the database, your products table will be structured as such:

```
mysql> describe products;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| id       | int          | NO   | PRI | NULL    | auto_increment |
| file     | varchar(255) | NO   |     | NULL    |                |
| in_stock | bit(1)       | NO   |     | NULL    |                |
| name     | varchar(255) | NO   |     | NULL    |                |
| price    | decimal(7,2) | NO   |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
5 rows in set (0.01 sec)
```

11. We are done designing our Spring Model (as Database Entity), now lets change our `application.properties` for property to `create` first.

```
...

spring.jpa.hibernate.ddl-auto=update
...
```

12. **Run the application, it will create the table.**. If you stop the application, it will no longer drop the table.

> ## ⚠️ Milestone:
>
> Commit the changes made to the project folder with message "Lesson 2.3 - Make Spring Model as Database Entity".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 2.3 - Make Spring Model as Database Entity"`.

Lesson 2.4 - Preparing Spring Application for Create, Read, Update and Delete (CRUD) operations.

1. Previously we implemented a class called `ProductRepository`.

```
package com.nep.practical.data;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.springframework.stereotype.Component;

import com.nep.practical.model.Product;

@Component
public class ProductRepository {

  private static final List<Product> ALL_PRODUCTS = Arrays.asList(
      new Product("Apple iPhone 6s", 1210.50, "/images/iphone6s.png",
true),
      new Product("Apple iPad Pro", 1310.99, "/images/ipadpro.png", true),
      new Product("Samsung Galaxy S7 Edge", 835.92,
"/images/samsungs7edge.png", false),
      new Product("Samsung Galaxy Note 7", 1035.92,
"/images/samsungnote7.png", true),
      new Product("HTC One X9", 850.20, "/images/htconex9.png", false),
      new Product("Oppo F1s", 400.28, "/images/oppof1s.png", true)
      );

  public static List<Product> getAllProducts() {
    return ALL_PRODUCTS;
  }

  public Product findByName(String name) {
```

```
    for(Product product : ALL_PRODUCTS) {
      if(product.getName().equals(name)) {
        return product;
      }
    }
    return null;
  }

  public List<Product> findContainName(String name) {
    List<Product> products = new ArrayList<Product>();
    for(Product product : ALL_PRODUCTS) {
      if(product.getName().toLowerCase().contains(name.toLowerCase())) {
        products.add(product);
      }
    }
    return products;
  }

}
```

2. First lets remove all the implementations made to this class, since we want to make this class to do CRUD operations to our Database.

```
package com.nep.practical.data;

import org.springframework.stereotype.Component;

@Component
public class ProductRepository {

}
```

3. Next, lets convert this from a Java class to Java `interface`.

```
...
public interface ProductRepository {

}
```

4. To allow this interface to be use to do CRUD operations to products table. We need to extend this interface with `CrudRepository` interface.

```
package com.nep.practical.data;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;

@Component
public interface ProductRepository extends CrudRepository {

}
```

5. Finally, we need to specify which Model class and the data type for the primary key of the Model class to `CrudRepository<Model Class, Primary Key Data Type>`. Since our Model class is `Product` and the field with `@Id` annotation is an integer. We need to specify it as `CrudRepository<Product, Integer>`.

```
package com.nep.practical.data;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Component;

import com.nep.practical.model.Product;

@Component
public interface ProductRepository extends CrudRepository<Product, Integer>
{

}
```

Once you have done this, these methods will be available to you (We will cover some of them, but actually there are more methods available).

The following are the methods that we will learn in the practical.

- delete(ID id) – Deletes the entity with the given id.
- exists(ID id) – Returns whether an entity with the given id exists.
- findAll() – Returns all entities.
- findOne(ID id) – Retrieves an entity by its id.
- save(S entity) – Saves a given entity.

6. There will be an error in the `HomeController` class. Lets remove the error first by commenting the lines that contain the error.

```
package com.nep.practical.controller;

import java.util.List;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.nep.practical.data.ProductRepository;
import com.nep.practical.model.Product;

@Controller
public class HomeController {

  @Autowired
  ProductRepository productRepository;

  @RequestMapping(value="/")
  public String home(ModelMap modelMap) {
//    List<Product> products = productRepository.getAllProducts();
//    modelMap.put("products", products);
    return "index";
  }

  @RequestMapping(value="/product/{name}")
  public String product(@PathVariable String name, ModelMap modelMap) {
//    Product product = productRepository.findByName(name);
//    modelMap.put("product", product);
    return "product";
  }

  @RequestMapping(value="/search")
  public String search(@RequestParam(required=false) String productName,
ModelMap modelMap) {
//    List<Product> products = null;
//    if(productName == null) {
//      products = productRepository.getAllProducts();
//    } else {
//      products = productRepository.findContainName(productName);
//    }
//    modelMap.put("products", products);
    return "search";
  }

}
```

We will start integrating this repository with our controller on the next exercise

> # ⚠️ Milestone:
>
> Commit the changes made to the project folder with message "Lesson 2.4 - Preparing Spring Application for Create, Read, Update and Delete (CRUD) operations.".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 2.4 - Preparing Spring Application for Create, Read, Update and Delete (CRUD) operations."`.