Lesson 03 Demo 01

Create Microservice with JDBC

Objective: To create a microservice using Spring Boot and JDBC to interact with a MySQL database

Tool required: Eclipse IDE, MySQL, and Postman

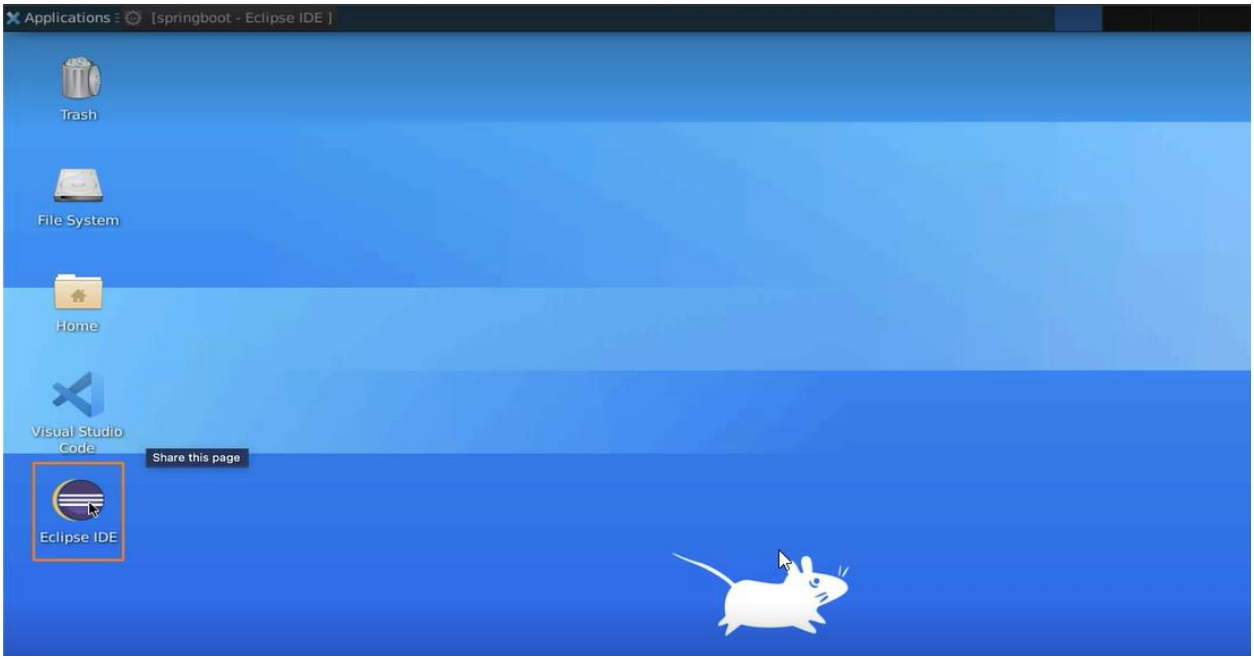
Prerequisites: None

Steps to be followed:

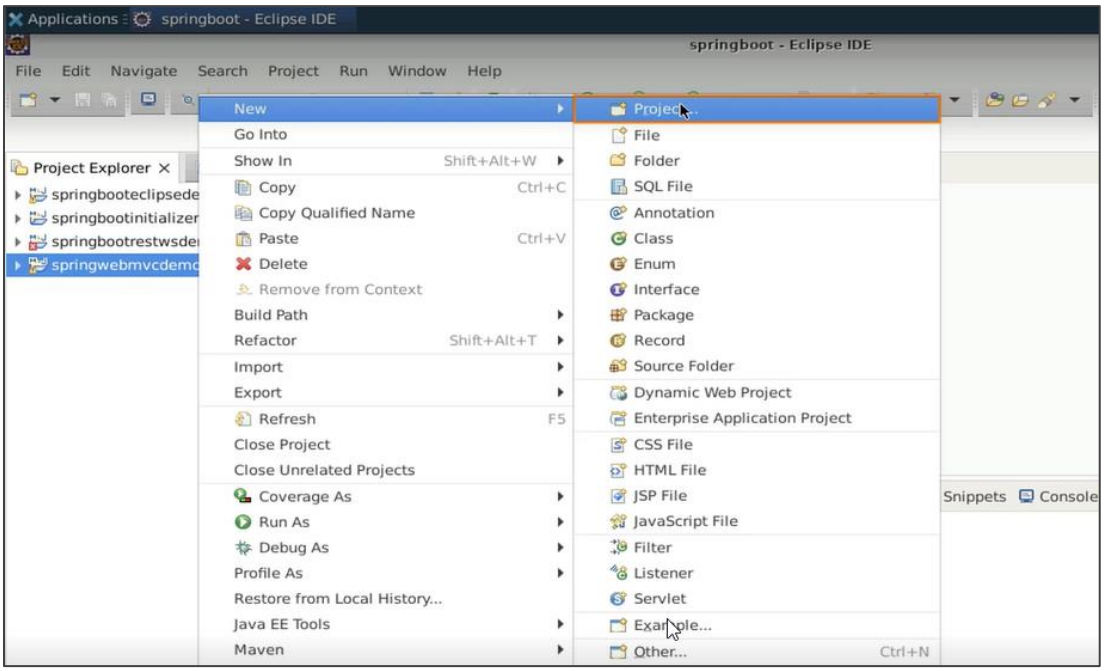
1. Creating a new Spring Starter project
2. Creating a welcome page
3. Creating the Product model class
4. Setting up the database configuration
5. Creating the ProductRepository interface
6. Creating the ProductController class
7. Creating the Response class
8. Configuring the CRUD methods
9. Running and testing the application

Step 1: Creating a new Spring Starter project

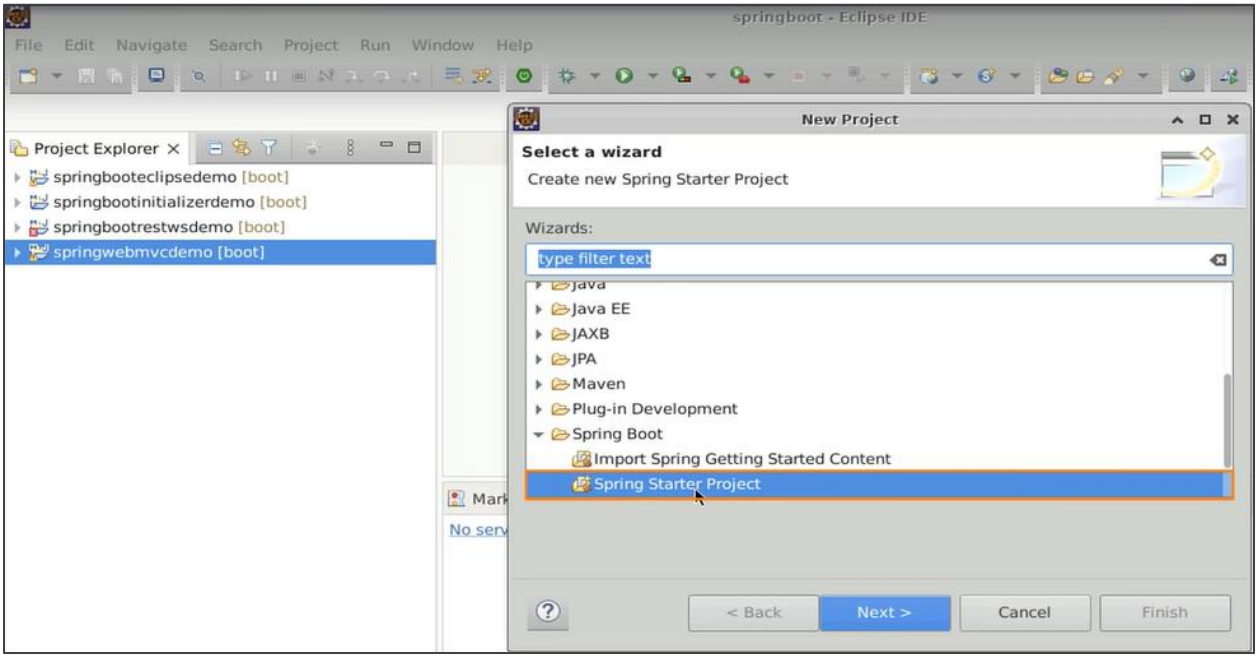
1.1 Open **Eclipse IDE**



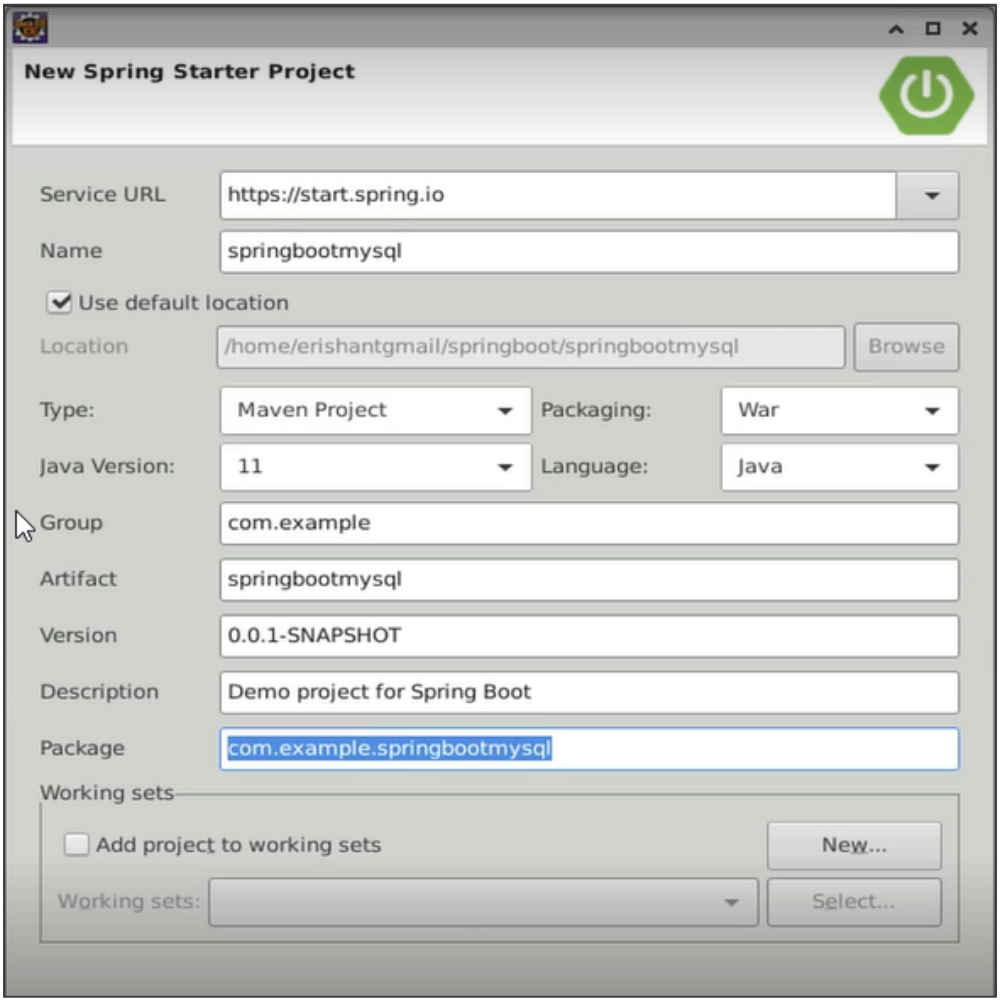
1.2 In the Project Explorer, right-click and select **New** > **Project**



1.3 Select **Spring Starter Project** and click **Next**

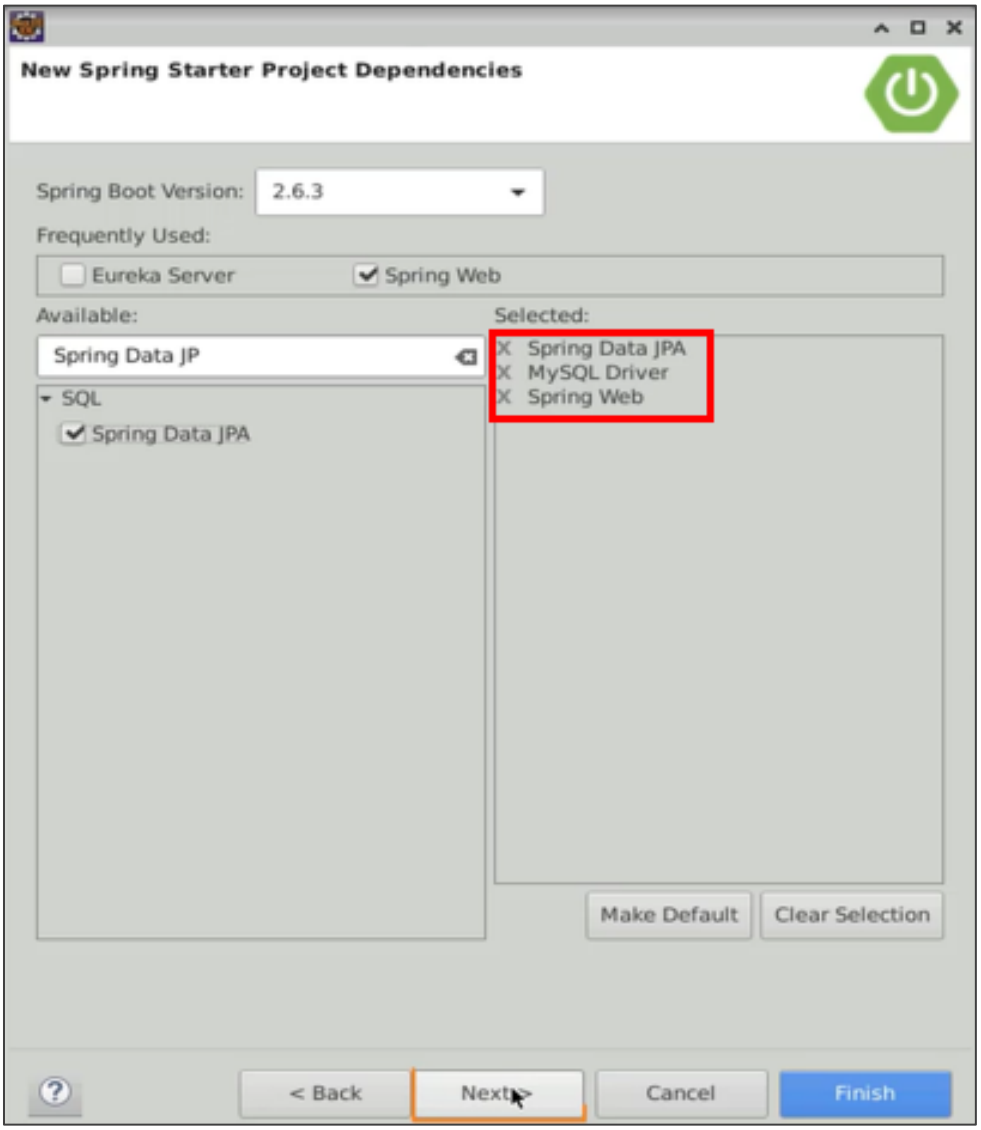


1.4 Provide a project name, such as **springbootmysql**, configure the project with Maven as the build tool, and choose the packaging as **War**



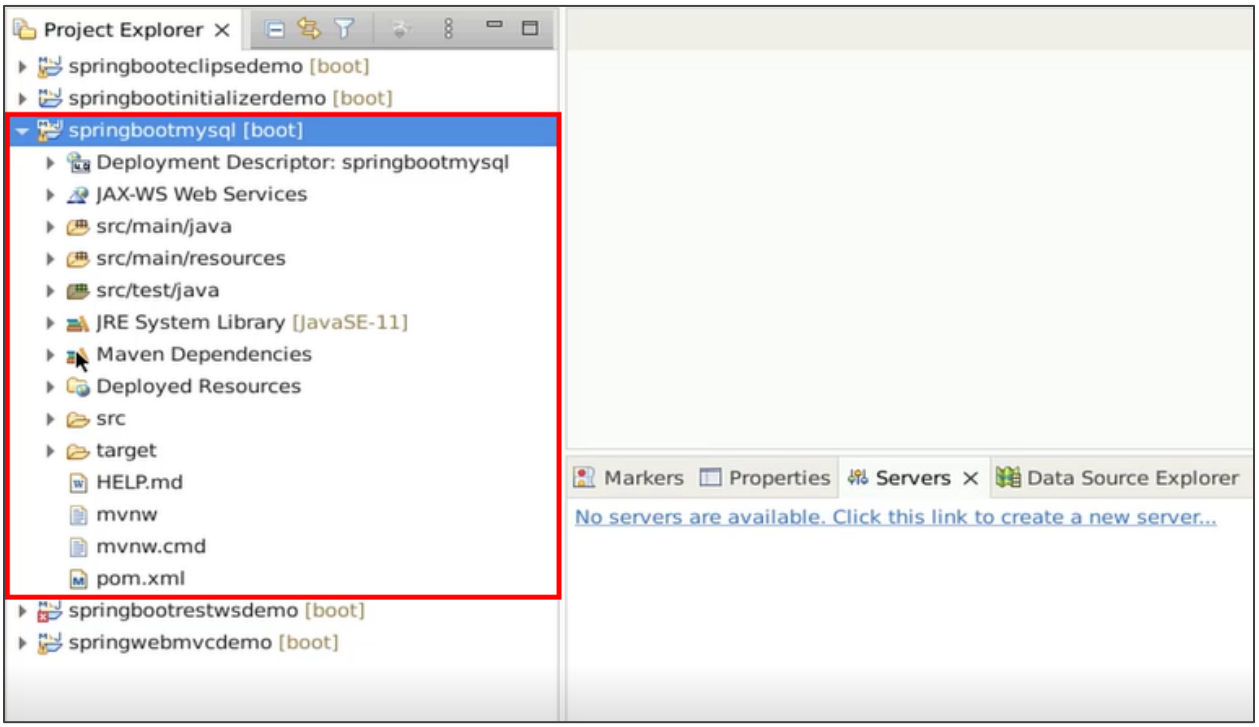
1.5 Click **Next** and add the following dependencies:

- Spring Data JPA
- MySQL Driver
- Spring Web



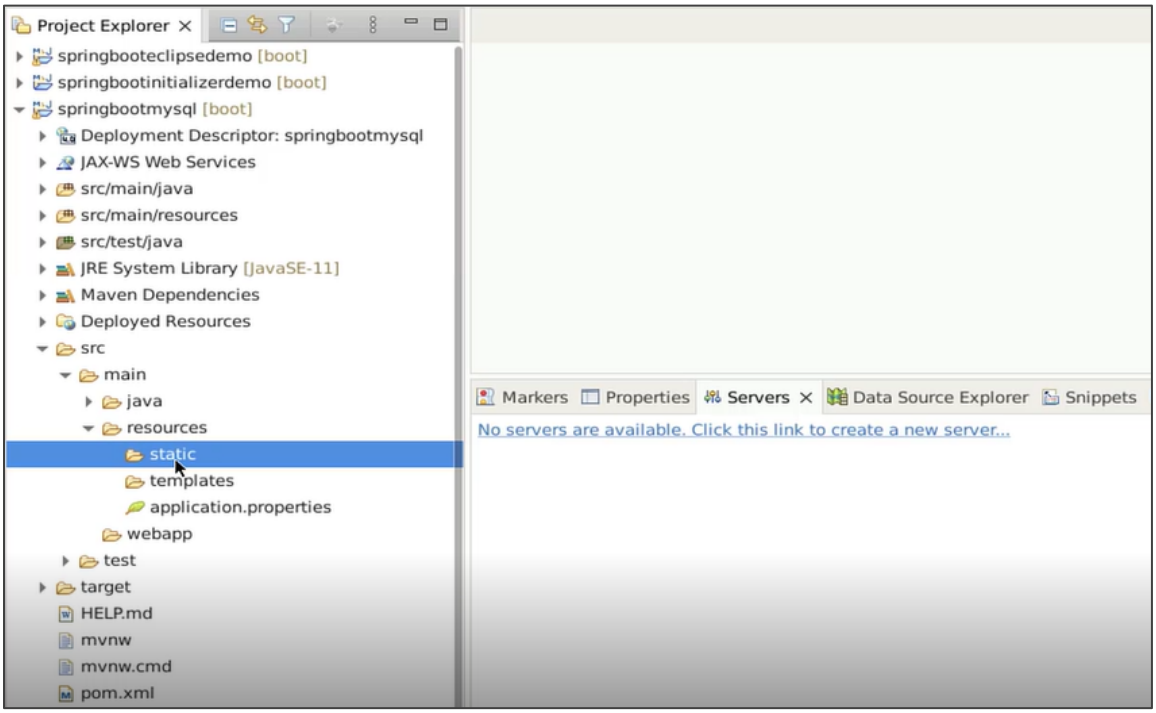
1.6 Click **Finish** to create the project with the specified dependencies

The eclipse will generate the project structure and download the necessary dependencies.



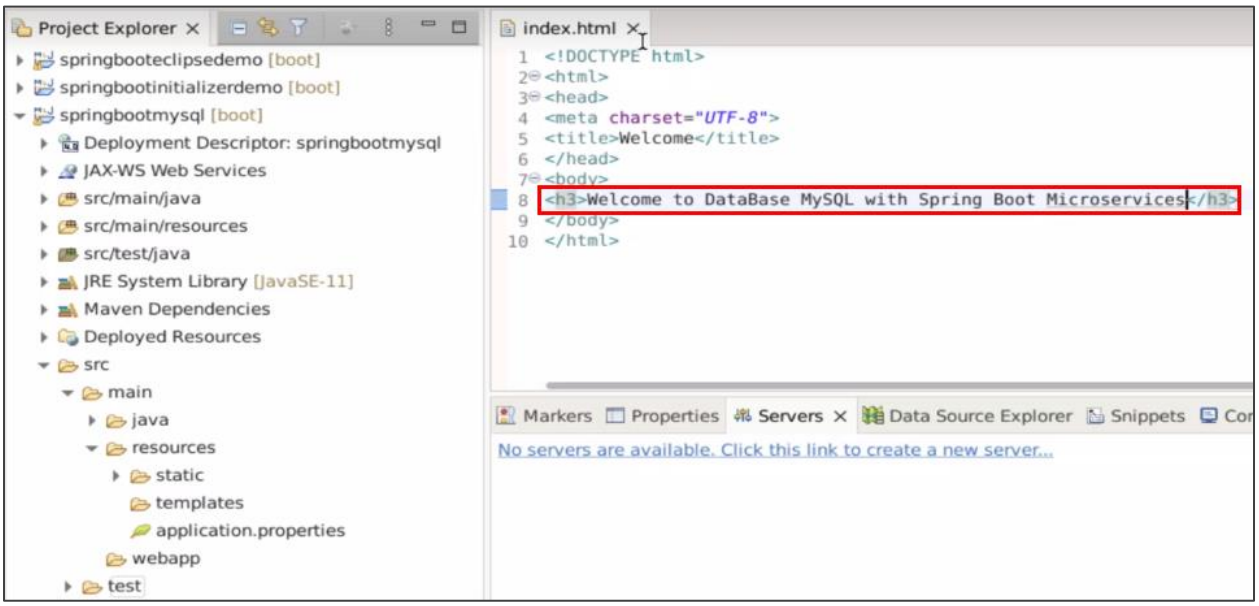
Step 2: Creating a welcome page

2.1 Navigate to the **src/main/resources/static** directory



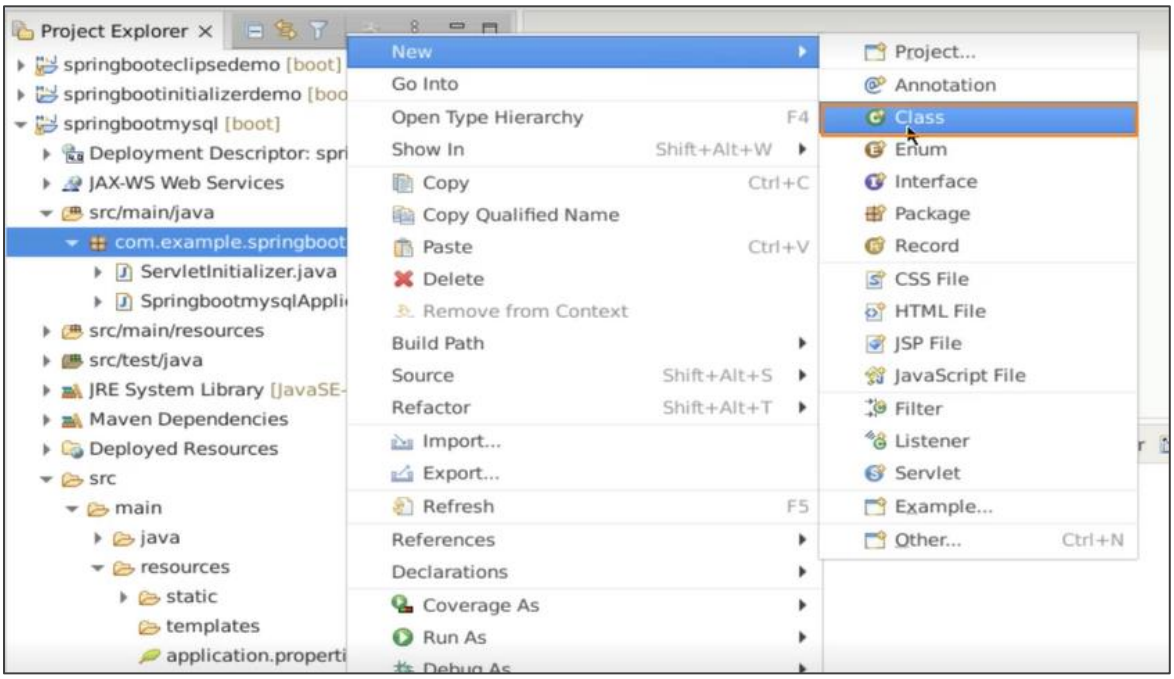
2.2 Create a new HTML file, such as **index.html**, and add the following code under the **<body>** tag:

<h3>Welcome to Database MySQL with Spring Boot Microservices</h3>

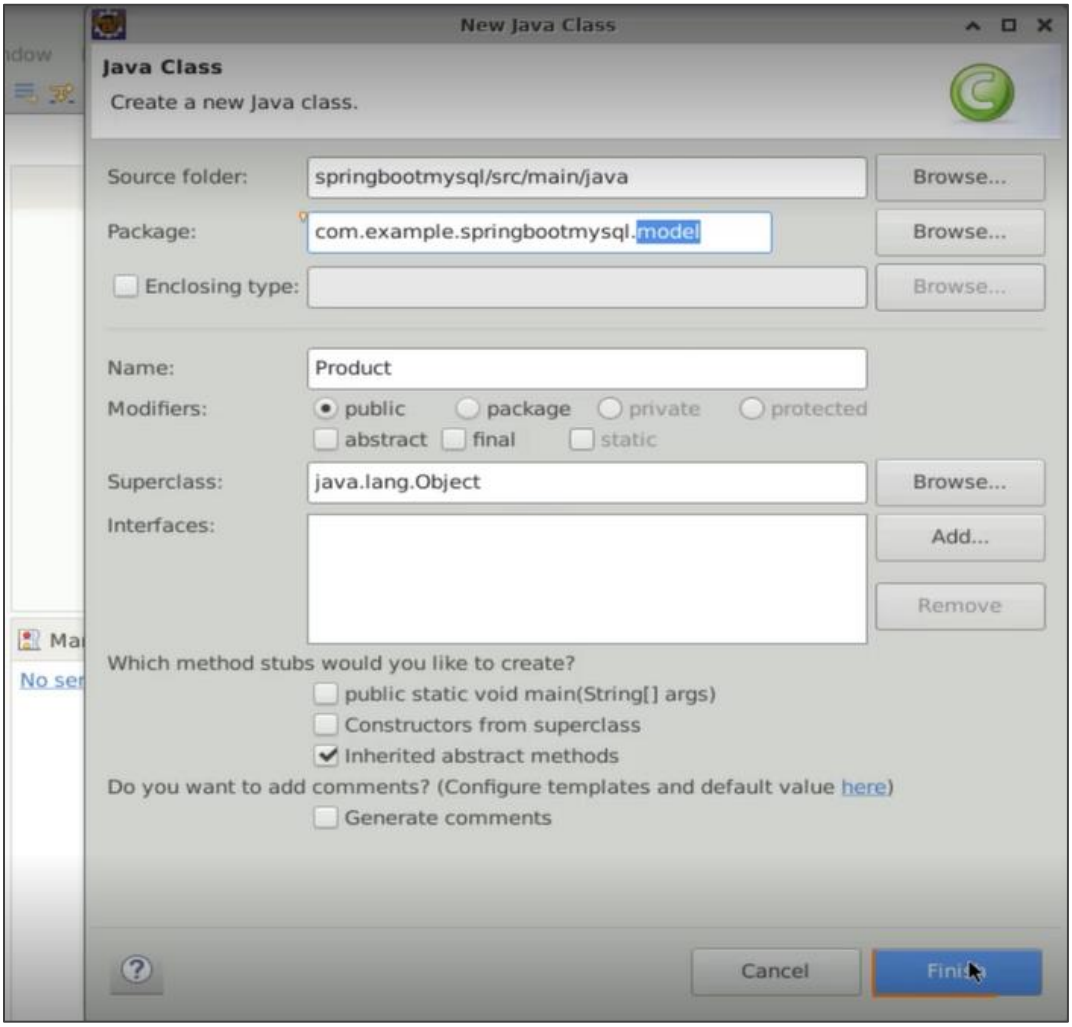


Step 3: Creating the Product model class

3.1 Right-click on the source package and select **New** > **Class**



3.2 Name the class **Product** and add **.model** to the package name and click **Finish**



3.3 Add the required attributes for the Product, such as **pid**, **name**, **brandName**, and **price**

```java
Product.java ×
package com.example.springbootmysql.model;

public class Product {

    Integer pid;

    String name;
    String brandName;
    Integer price;

}
```

3.4 Generate the default constructor, parameterized constructor, getters, setters, and a **toString()** method for the Product class

```java
*Product.java ×
package com.example.springbootmysql.model;

public class Product {

    Integer pid;

    String name;
    String brandName;
    Integer price;

    public Product() {
        // TODO Auto-generated constructor stub
    }

    public Product(Integer pid, String name, String brandName, Integer price) {
        this.pid = pid;
        this.name = name;
        this.brandName = brandName;
        this.price = price;
    }
```

```java
*Product.java ×
    public void setPid(Integer pid) {
        this.pid = pid;
    }

    public String getName() {
        return name;
    }

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

    public String getBrandName() {
        return brandName;
    }

    public void setBrandName(String brandName) {
        this.brandName = brandName;
    }

    public Integer getPrice() {
        return price;
    }

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

    @Override
    public String toString() {
        return "Product [pid=" + pid + ", name=" + name + ", brandName=" + brandName + ", price=" + price + "]";
    }
```

3.5 Annotate the **Product** class with **@Entity** to mark it as an entity for JPA

```java
*Product.java ×
package com.example.springbootmysql.model;

import javax.persistence.Entity

@Entity
public class Product {


    Integer pid;

    String name;
    String brandName;
    Integer price;

    public Product() {
        // TODO Auto-generated constructor stub
    }

    public Product(Integer pid, String name, String brandName, Integer price) {
        this.pid = pid;
        this.name = name;
        this.brandName = brandName;
        this.price = price;
    }

    public Integer getPid() {
        return pid;
    }
```

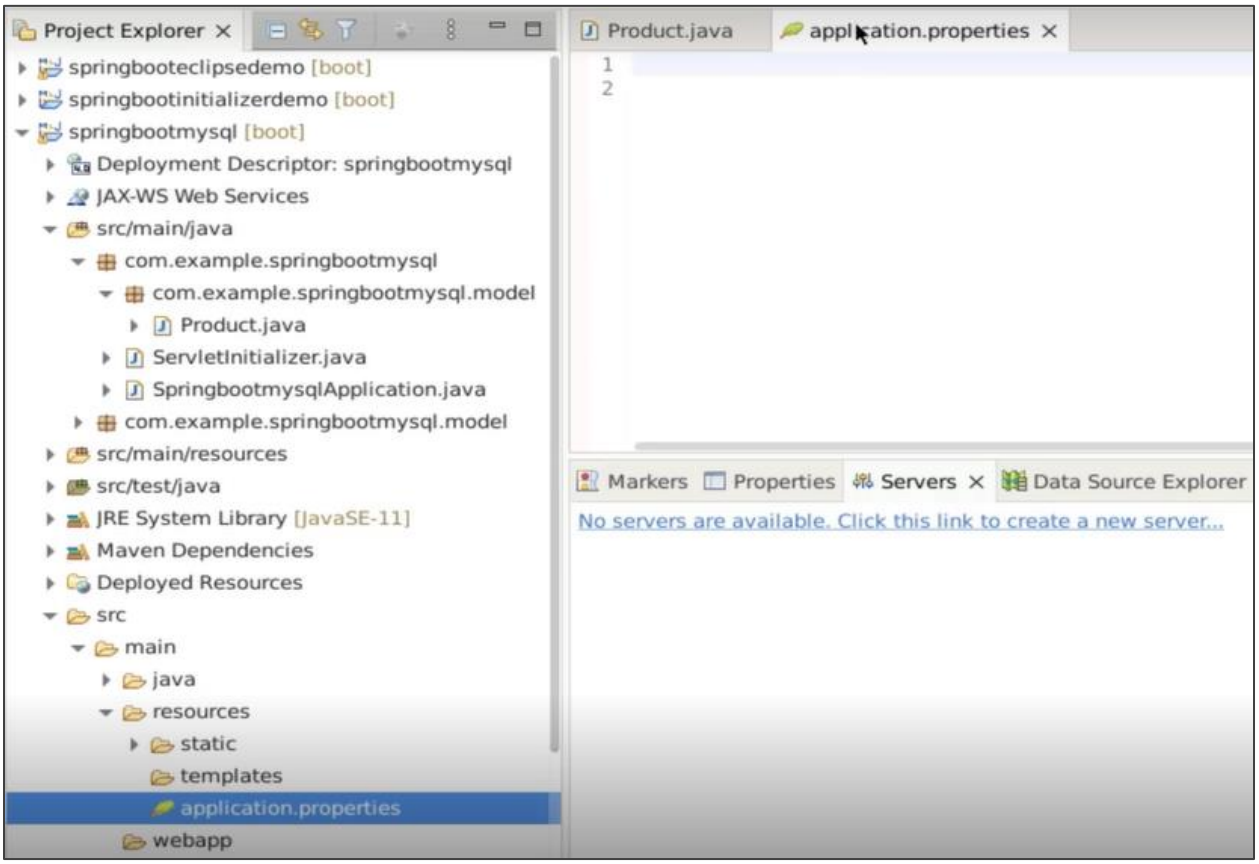
3.6 Use JPA annotations like **@Id** and **@GeneratedValue** to define the primary key and auto-increment strategy, respectively

```java
Product.java ×
package com.example.springbootmysql.model;

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

@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    Integer pid;

    String name;
    String brandName;
    Integer price;

    public Product() {
        // TODO Auto-generated constructor stub
    }

    public Product(Integer pid, String name, String brandName, Integer price) {
        this.pid = pid;
        this.name = name;
        this.brandName = brandName;
        this.price = price;
    }

```

The table name will automatically derive from the class name.

Step 4: Setting up the database configuration

4.1 Open the **application.properties** file located in the **src/main/resources** directory

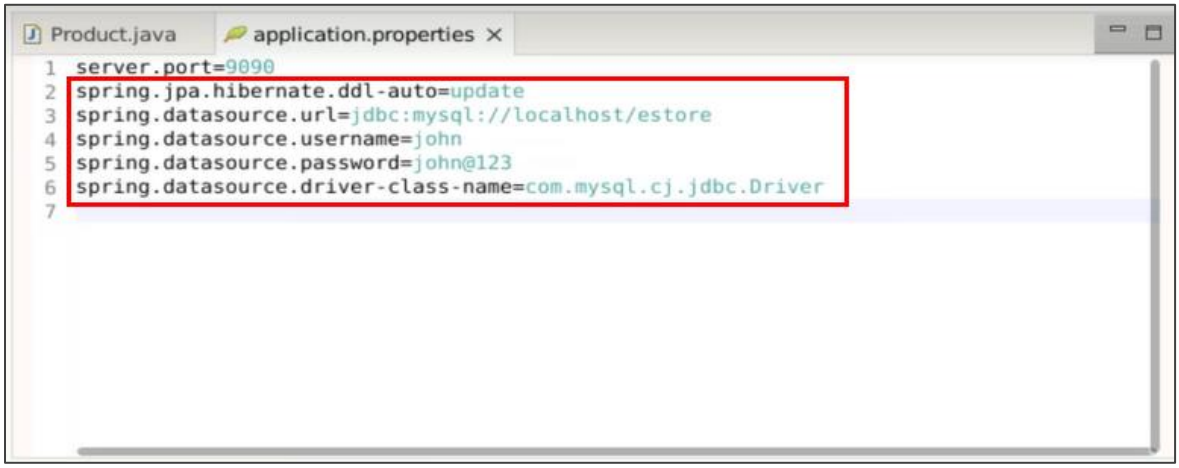


4.2 Configure the server port by adding the property **server.port=9090** (you can choose any port number)

4.3 Set the database configuration properties as follows:

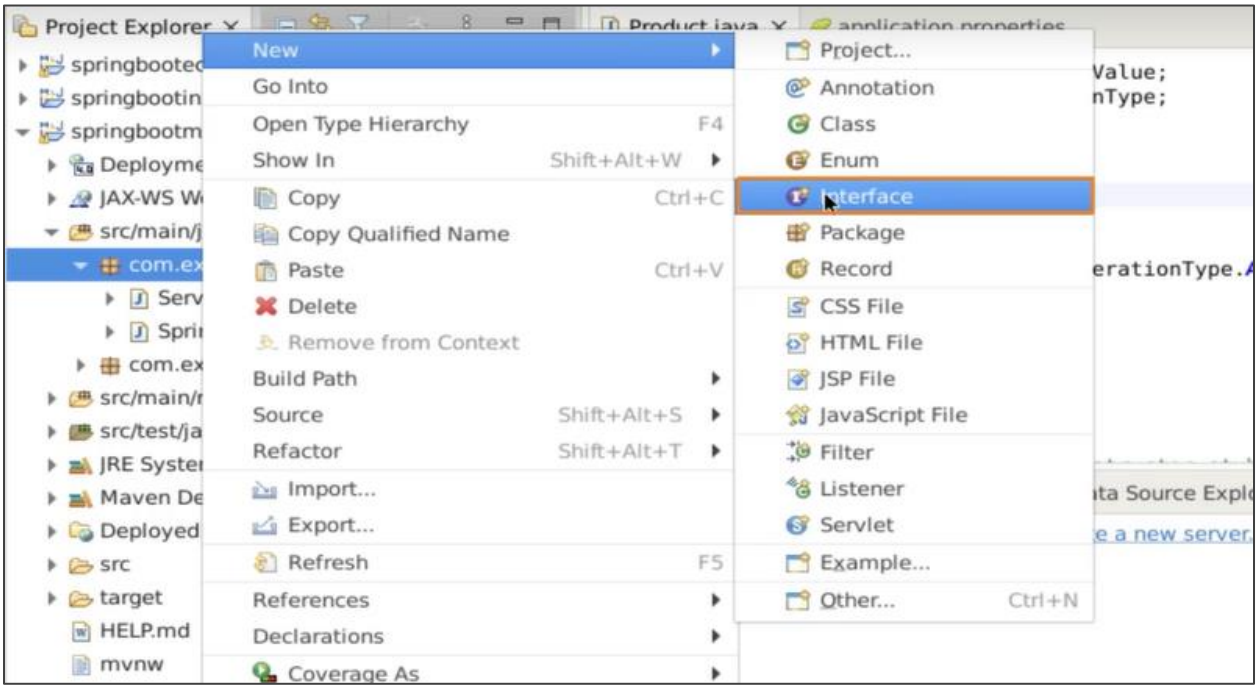
- **spring.jpa.hibernate.ddl-auto=update** to automatically create or update the tables in the database
- **spring.datasource.url=jdbc:mysql://localhost/estore** to specify the MySQL database URL
- **spring.datasource.username=john** and **spring.datasource.password=john123** to provide the database credentials

- **spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver** to specify the MySQL driver class

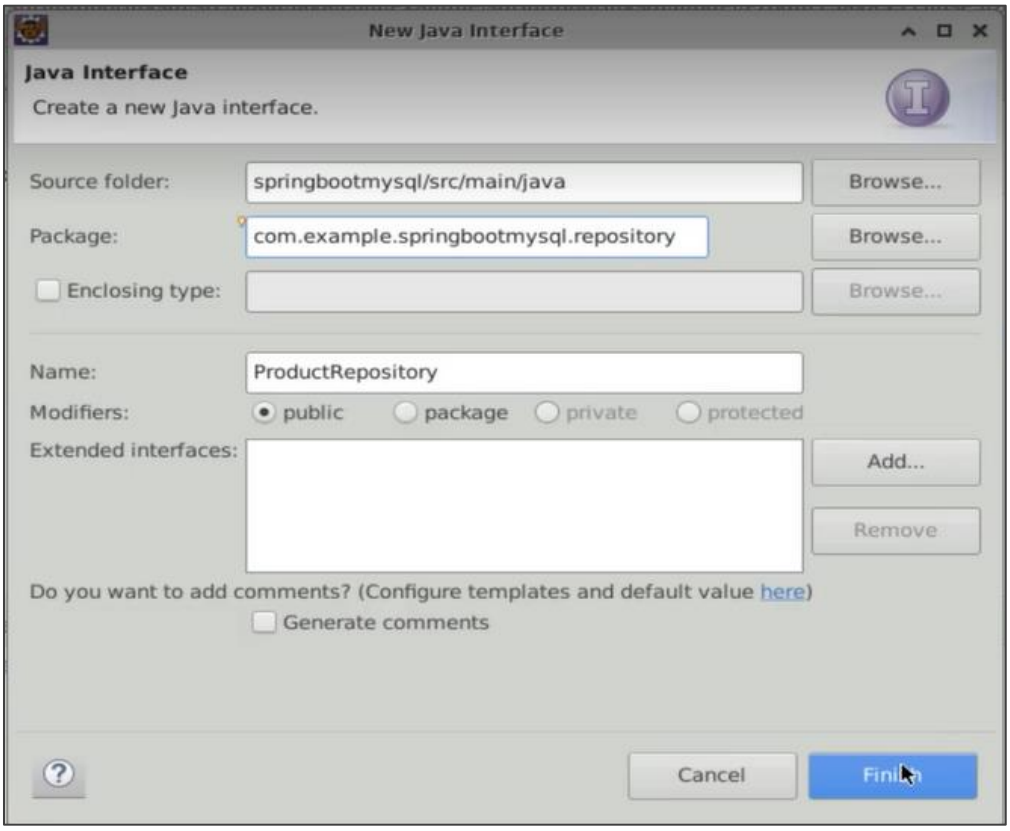


Step 5: Creating the ProductRepository interface

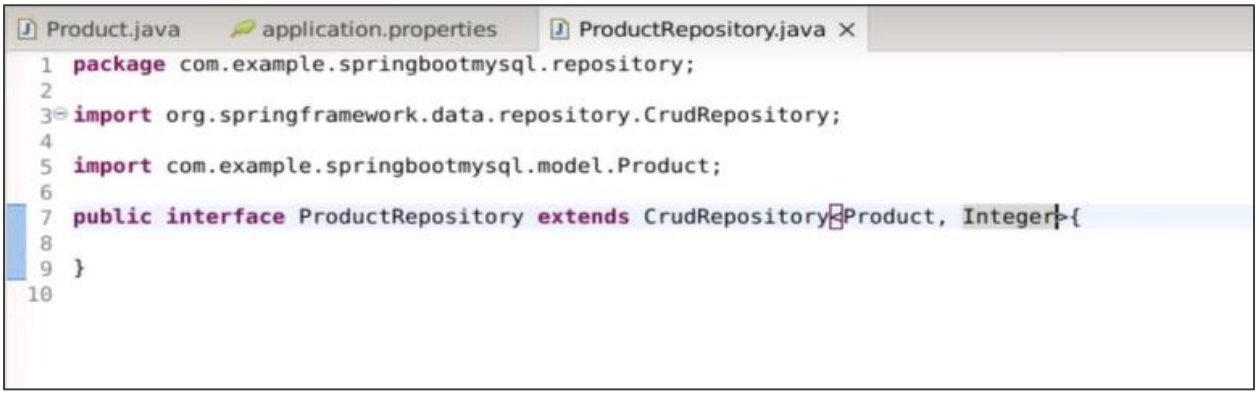
5.1 Right-click on the source package and select **New** > **Interface**



5.2 Name the interface **ProductRepository** and add **.repository** to the package name and click **Finish**

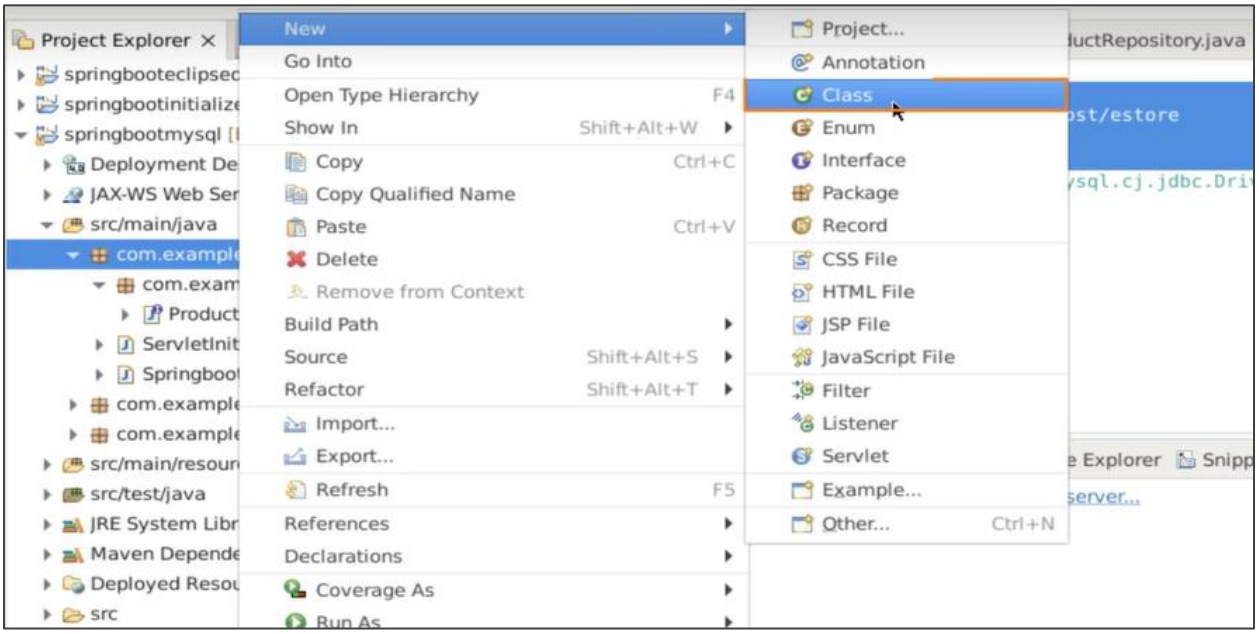


5.3 Extend the **CrudRepository<Product, Integer>** interface to inherit the CRUD operations for the **Product** entity

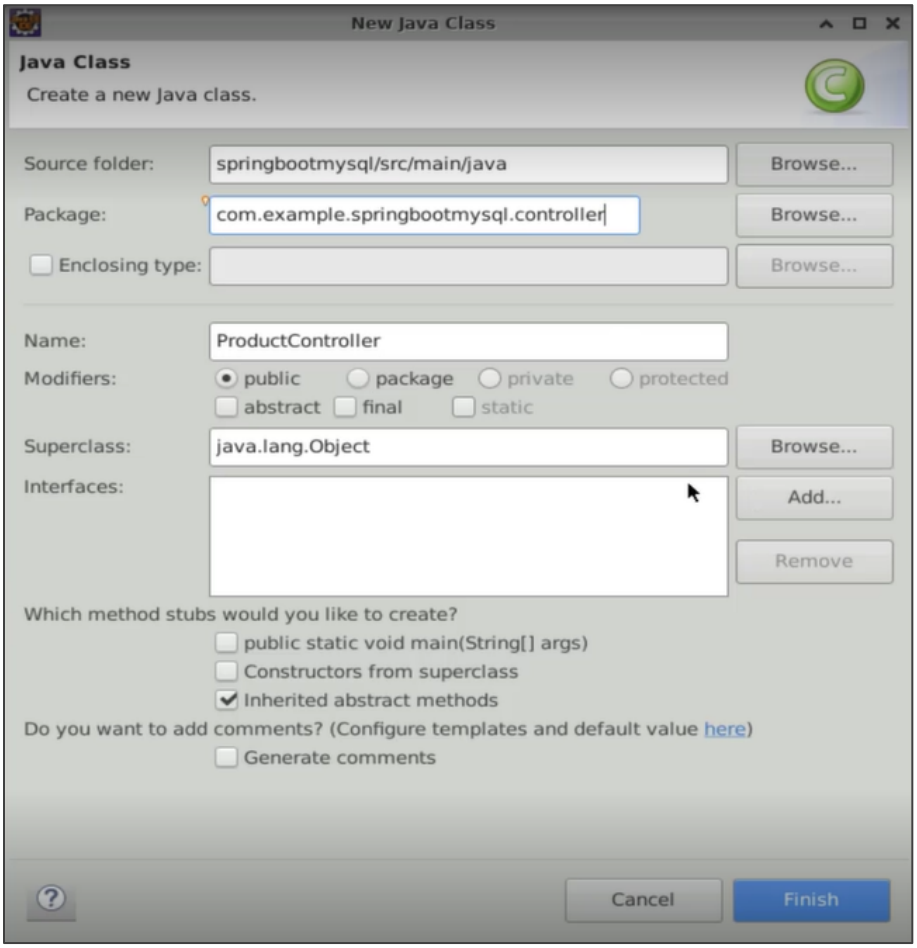


Step 6: Creating the ProductController class

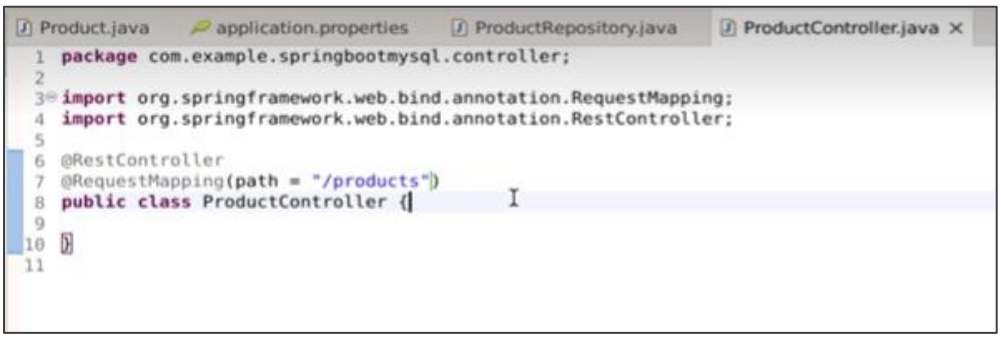
6.1 Right-click on the source package and select **New** > **Class**



6.2 Name the class **ProductController** and add **.controller** to the package name and click **Finish**



6.3 Annotate the class with **@RestController** to indicate it's a controller for handling RESTful requests. Add **@RequestMapping** with the path set to **/products**

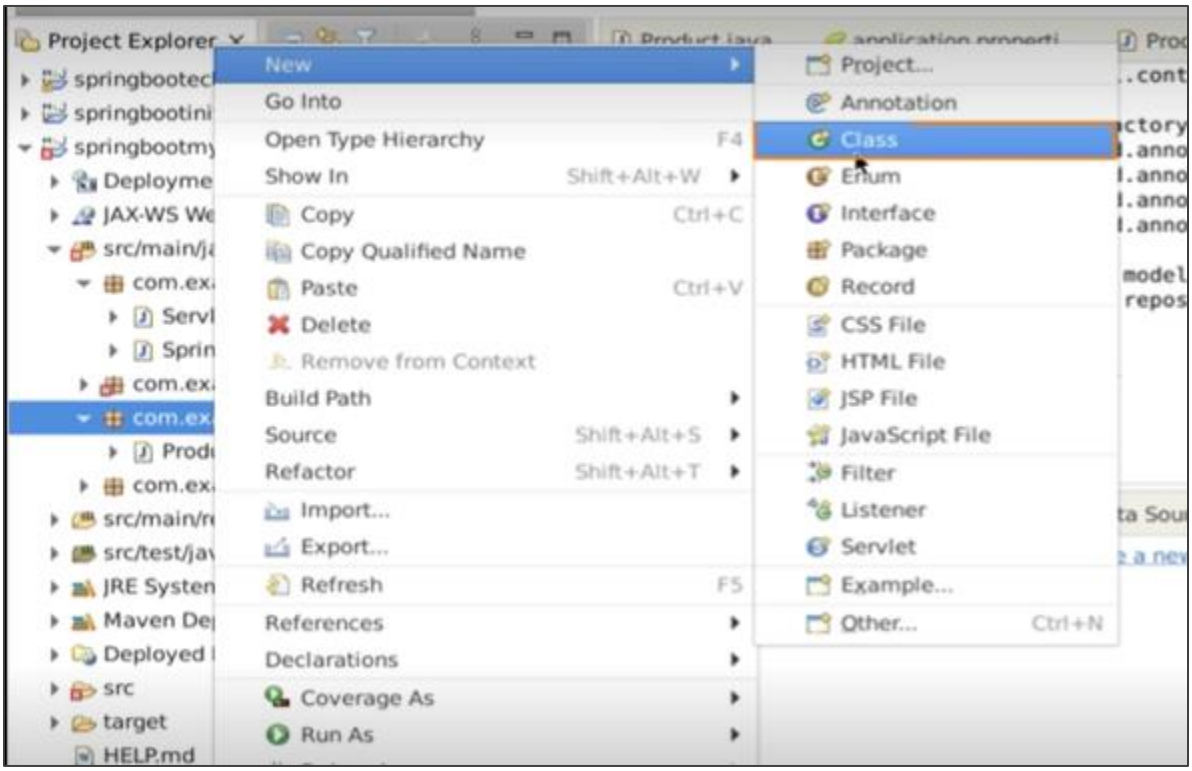


6.4 Autowire the **ProductRepository** into the controller

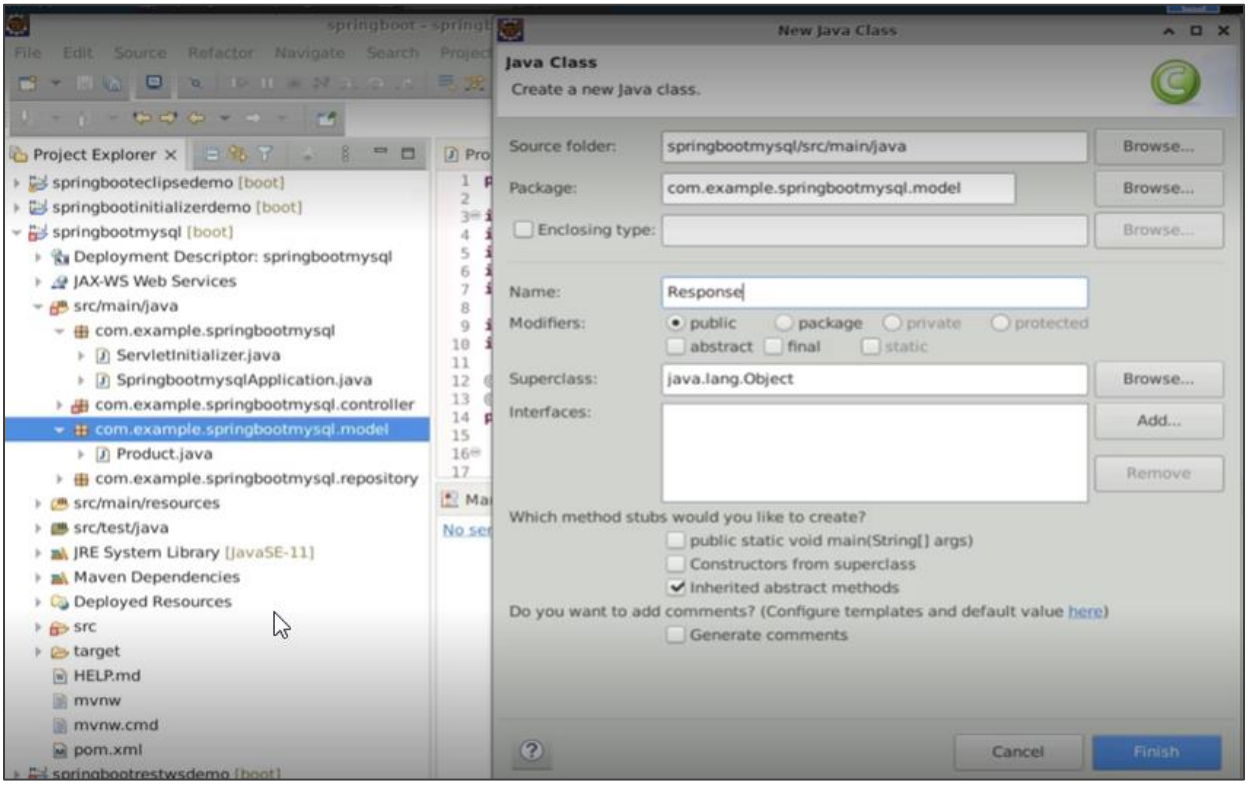
```
Product.java   application.properties   ProductRepository.java   ProductController.java ×
1  package com.example.springbootmysql.controller;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.bind.annotation.RestController;
6
7  import com.example.springbootmysql.repository.ProductRepository;
8
9  @RestController
10 @RequestMapping(path = "/products")
11 public class ProductController {
12
13     @Autowired
14     ProductRepository repository;
15 }
16
```

Step 7: Creating the Response class

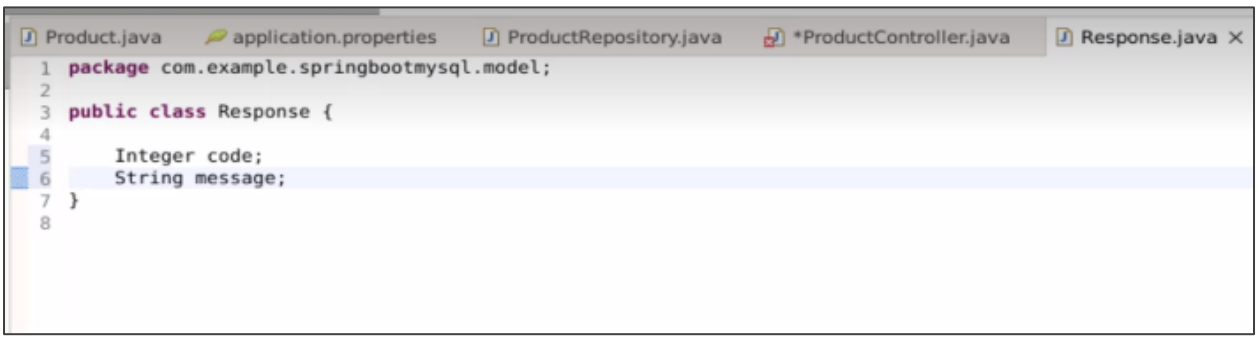
7.1 Right-click on the controller package and select **New** > **Class**



7.2 Name the class **Response** and click **Finish**



7.3 Define the class with the necessary fields for the response, such as **code** and **message**



7.4 Generate the default constructor, parameterized constructor, getters, setters, and a **toString()** method for the Response class

```
Product.java   application.properties   ProductRepository.java   *ProductController.java   *Response.java

4
5       Integer code;
6       String message;
7
8       public Response() {
9           // TODO Auto-generated constructor stub
10      }
11
12      public Response(Integer code, String message) {
13          this.code = code;
14          this.message = message;
15      }
16
17      public Integer getCode() {
18          return code;
19      }
20
21      public void setCode(Integer code) {
22          this.code = code;
23      }
24
25      public String getMessage() {
26          return message;
27      }
28
29      public void setMessage(String message) {
30          this.message = message;
31      }
32
33      @Override
34      public String toString() {
35          return "Response [code=" + code + ", message=" + message + "]";
36      }
```

Step 8: Configuring the CRUD methods

8.1 In the **ProductController.java** class, add a method called **addProduct** and annotate it with **@PostMapping** to set the endpoint as **/add**

```java
package com.example.springbootmysql.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.example.springbootmysql.repository.ProductRepository;

@RestController
@RequestMapping(path = "/products")
public class ProductController {

    @Autowired
    ProductRepository repository;

    @PostMapping(path = "/add")
    public String addProduct(@RequestParam String name, @RequestParam String brandName, @RequestParam Integer price) {

    }
}
```

8.2 Implement the logic to save the product details in the database and return a status code of **101** if successful. If any exceptions occur, catch them using a try-catch block and return a status code of **701**.

```java
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.example.springbootmysql.model.Product;
import com.example.springbootmysql.model.Response;
import com.example.springbootmysql.repository.ProductRepository;

@RestController
@RequestMapping(path = "/products")
public class ProductController {

    @Autowired
    ProductRepository repository;

    @PostMapping(path = "/add")
    public ResponseEntity<Response> addProduct(@RequestParam String name, @RequestParam String brandName, @RequestParam Integer price) {

        Product product = new Product(null, name, brandName, price);
        System.out.println("Product: "+product);

        try {
            repository.save(product);

            Response response = new Response(101, "Product "+name+" Saved Successfully");
            return new ResponseEntity<Response>(response, HttpStatus.OK);

        }catch(Exception exception) {
            Response response = new Response(701, "Product "+name+" Not Saved Successfully. Exception: "+exception.getMessage());
            return new ResponseEntity<Response>(response, HttpStatus.OK);
        }
```

Similarly, you can implement other methods like update, delete, and read operations for the product table.

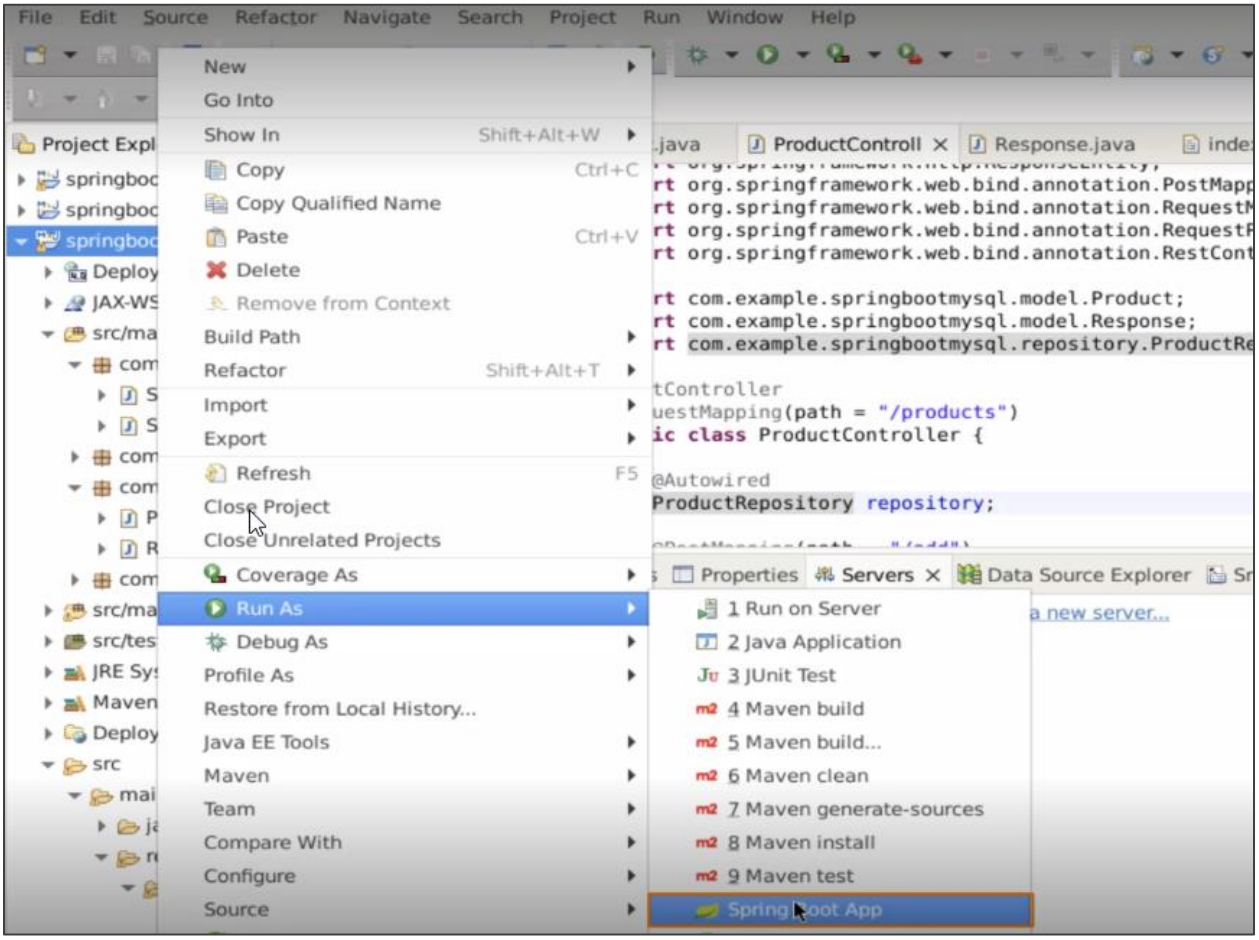
Step 9: Running and testing the application

9.1 Before running the application, open the terminal and run the command **show tables;** to verify if the product table is available or not

```
erishantgmail@ip-172-31-84-97: ~
File  Edit  View  Search  Terminal  Help

Database changed
mysql> show tables;
+------------------+
| Tables_in_estore |
+------------------+
| Dish             |
| Product          |
+------------------+
2 rows in set (0.00 sec)

mysql> drop table Dish;
Query OK, 0 rows affected (0.01 sec)

mysql> drop table Product;
Query OK, 0 rows affected (0.01 sec)

mysql> show tables;
Empty set (0.00 sec)

mysql> show tables;
Empty set (0.00 sec)

mysql>
```

There are currently no tables present within the **estore** database that has already been created.

9.2 Right-click on the project and select **Run As** > **Spring Boot App** to start the application



Spring Boot will automatically start the embedded Tomcat Server and deploy the application at **localhost:9090**

9.3 Return to the terminal and verify the table creation by running the following commands:

show tables;

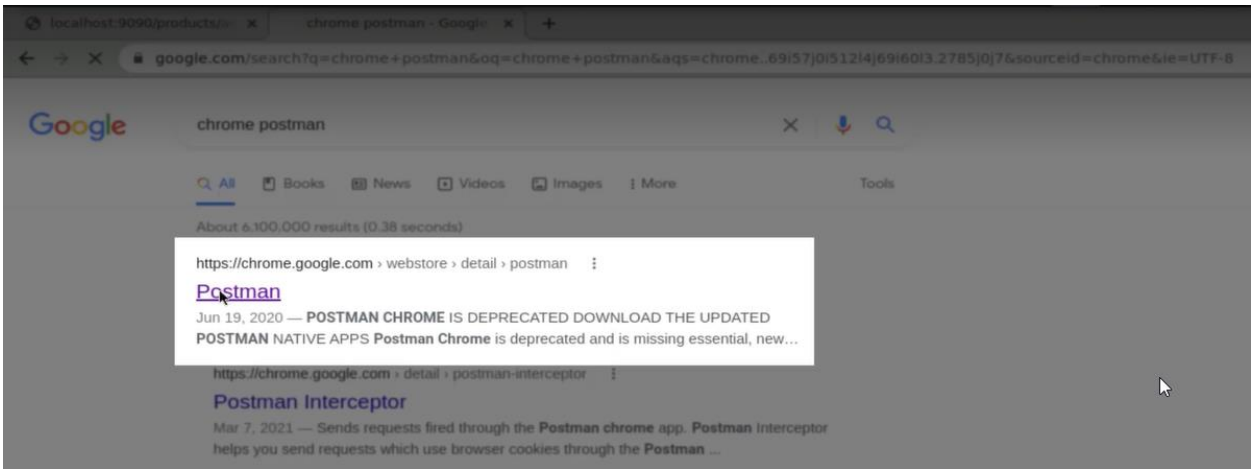
describe product;

```
erishantgmail@ip-172-31-84-97: ~
File  Edit  View  Search  Terminal  Help
2 rows in set (0.00 sec)

mysql> drop table Dish;
Query OK, 0 rows affected (0.01 sec)

mysql> drop table Product;
Query OK, 0 rows affected (0.01 sec)

mysql> show tables;
Empty set (0.00 sec)

mysql> show tables;
Empty set (0.00 sec)

mysql> show tables;
+--------------------+
| Tables_in_estore   |
+--------------------+
| hibernate_sequence |
| product            |
+--------------------+
2 rows in set (0.00 sec)

mysql>
```

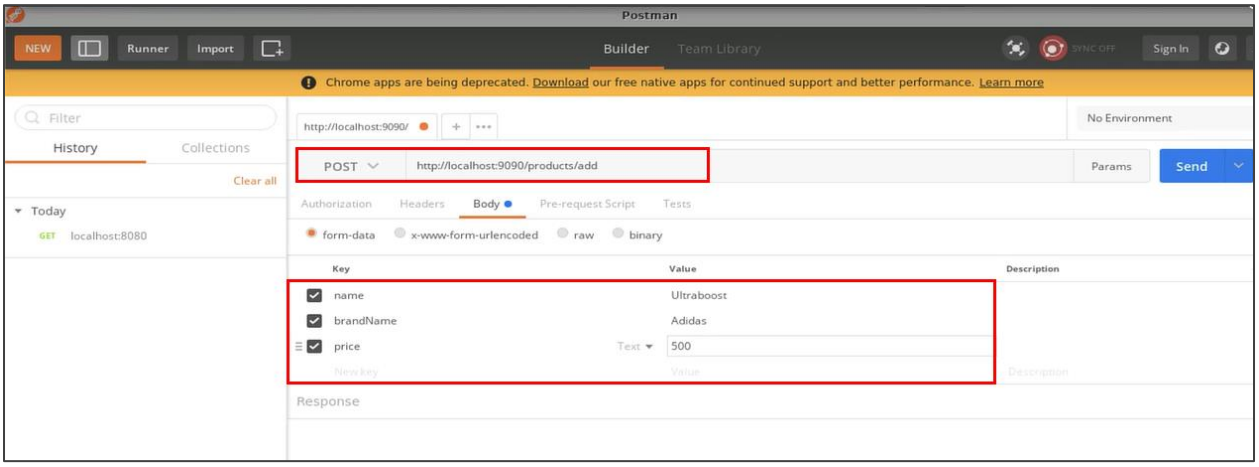
```
erishantgmail@ip-172-31-84-97: ~
File  Edit  View  Search  Terminal  Help
mysql> show tables;
Empty set (0.00 sec)

mysql> show tables;
+--------------------+
| Tables_in_estore   |
+--------------------+
| hibernate_sequence |
| product            |
+--------------------+
2 rows in set (0.00 sec)

mysql> describe product;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| pid        | int          | NO   | PRI | NULL    |       |
| brand_name | varchar(255) | YES  |     | NULL    |       |
| name       | varchar(255) | YES  |     | NULL    |       |
| price      | int          | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql>
```

You will notice that the product table is created by the Spring application.

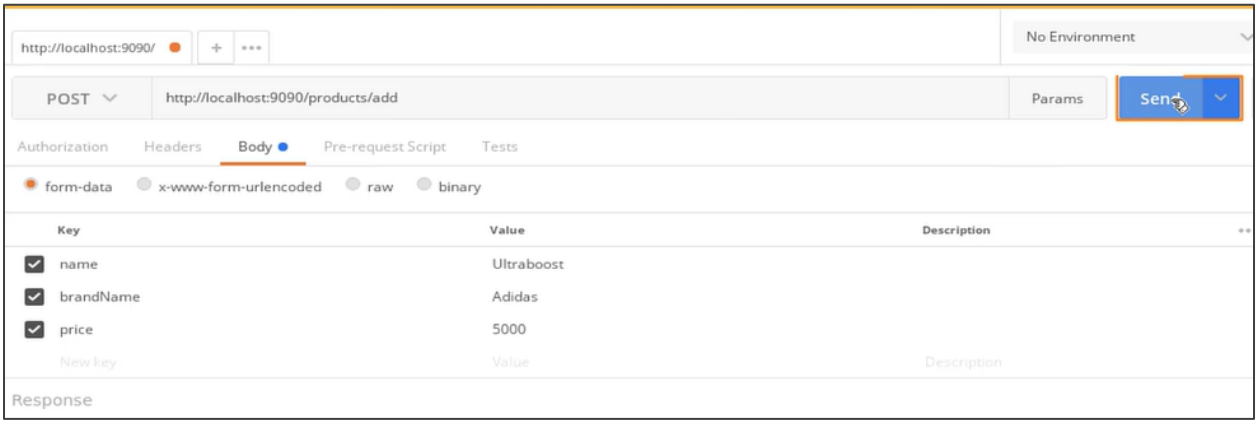
9.4 To test the **addProduct** method, open Postman in a web browser

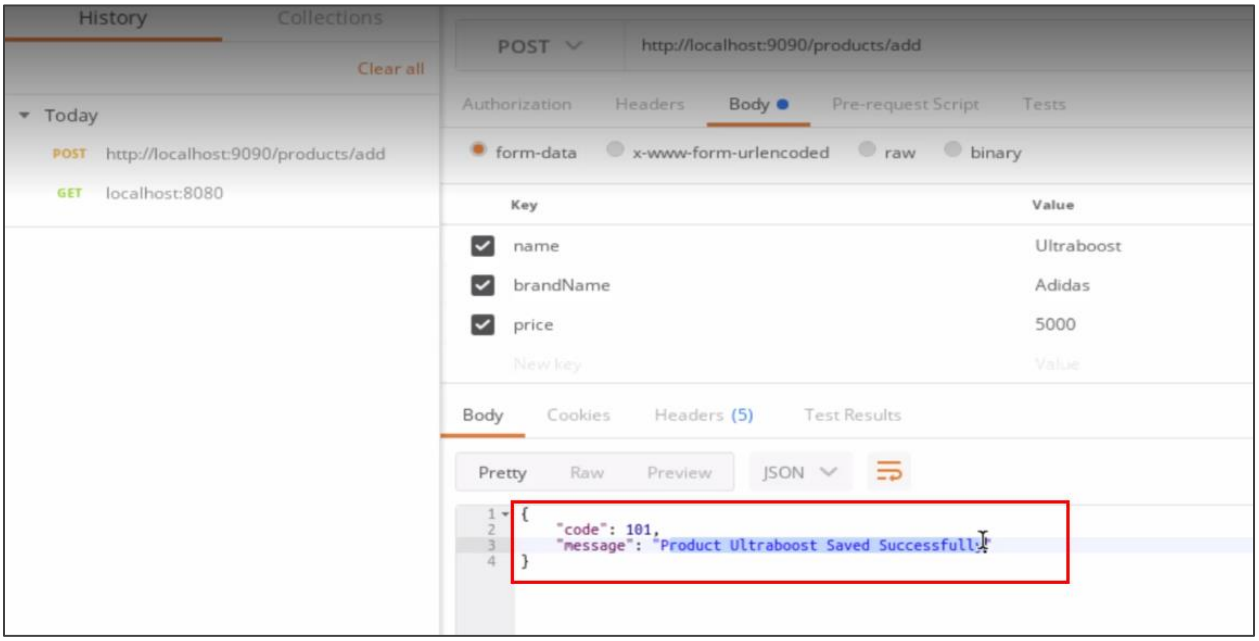


9.5 On the main page, send a POST request to the URL **http://localhost:9090/products/add**, which forwards the request to the **addProduct()** controller method. Define the data for the product table using key-value pairs under the **Body** section



9.6 Click **Send**





You will see the JSON output with a message stating **Product Ultraboost Saved Successfully** and a status code of **101**.

9.7 In the terminal, run the following query to check if any records are added to the product table:

select * from product;

```
erishantgmail@ip-172-31-84-97: ~
File  Edit  View  Search  Terminal  Help
| brand_name | varchar(255) | YES  |     | NULL    |       |
| name       | varchar(255) | YES  |     | NULL    |       |
| price      | int          | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> select * from product;
+-----+------------+-----------+-------+
| pid | brand_name | name      | price |
+-----+------------+-----------+-------+
|   1 | Apple      | iPhone 11 | 70000 |
+-----+------------+-----------+-------+
1 row in set (0.00 sec)

mysql> select * from product;
+-----+------------+------------+-------+
| pid | brand_name | name       | price |
+-----+------------+------------+-------+
|   1 | Apple      | iPhone 11  | 70000 |
|   2 | Adidas     | Ultraboost |  5000 |
+-----+------------+------------+-------+
2 rows in set (0.00 sec)

mysql>
```

You can see that the **Adidas Ultraboost** product has been added to the product table as specified in Postman.

Similarly, you can implement other methods to read, update, and delete products in the **ProductController.java** class.