19-Spring Data JPA with JPA Repository

Checking the H2 Database

To make sure that our H2 in-memory database is set up correctly:

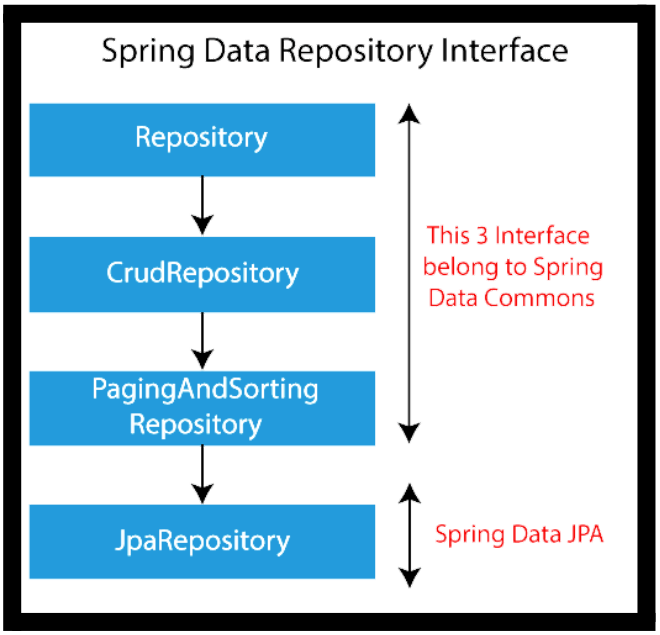
Go to http://localhost:8080/h2-console in your browser.
To access the H2 console, press "Enter."
When you go to the console layout, you'll see that no tables have been made yet; only the default tables are there. So, we need to look into the repository layer for creating classes and interfaces that interact with the database.

Repository Layer:

Do we need to create a class as we did in previous all layers? No, we don't need. Instead, we will make an interface called "ProductRepo" and annotate it with **@Repository** having repo as reference. The rest is taken care of by Spring Data JPA. The JpaRepository interface expects **Product** as the entity class and **Integer** as the primary key.

```
@Repository
public interface ProductRepo extends JpaRepository<Product, Integer> {

}
```

Hierarchy of JPA repository:



@Repository:

The **@Repository** annotation is a specialized form of the **@Component** annotation, indicating that the class can **save, fetch, update, delete**, and **search** for objects. Spring automatically detects repository classes by scanning the class path. This annotation works.

similar to that of the DAO pattern, where DAO classes handle CRUD operations on database tables.

Service Layer:

In the service layer, we will inject and use the repository interface as a reference variable, **ProductRepo,** and add **the @Autowired** annotation to it. Also, we don't have class for this Spring Data JPA is responsible for creating it, and Spring will manage it.

```
 1 package com.telusko.SimplewebApp.service;
 2
 3⊕import java.util.List;
10
11 @Service
12 public class ProductService {
13
14•     @Autowired
15      ProductRepo repo;
16
```

From now on, you can comment out the previous code data that was hard-coded because we're not using a database and added products manually.

Implementation of Code:

In the service layer, you don't have to write implementations for methods that are already there because JpaRepository gives them to you. There are many methods that you can use when you access the repo object.

Examples of Methods:

1) getProduct():
   The **findAll ()** method shows a list of all the products in the database.

```
public List<Product> getproducts() {
    return repo.findAll();

}
```

2) getProductById (): Here the **findById ()** method will get the product-based ID, and it returns optional.

```
public Product getProductById(int prodId) {
    return repo.findById(prodId).orElse(new Product());
}
```

3) addProduct(): The **save ()** method saves the product object and stores it in the database.

```
public void addProduct(Product prod) {
    repo.save(prod);
}
```

4) updateProduct (): For updating, we don't have any specific method. Products are updated by the same **save()** method; if the product is not available, then it will create a new product.

```
public void updateProduct(Product prod) {

    repo.save(prod);
}
```

5) deleteProduct (): For deleting the product by id we have the same method **deleteById ()** available in the jpa, which will delete the corresponding object based on its id.

```
public void deleteProduct(int prodId) {
    repo.deleteById(prodId);
}
```

Putting the project together
Run the project to see if there are any mistakes.

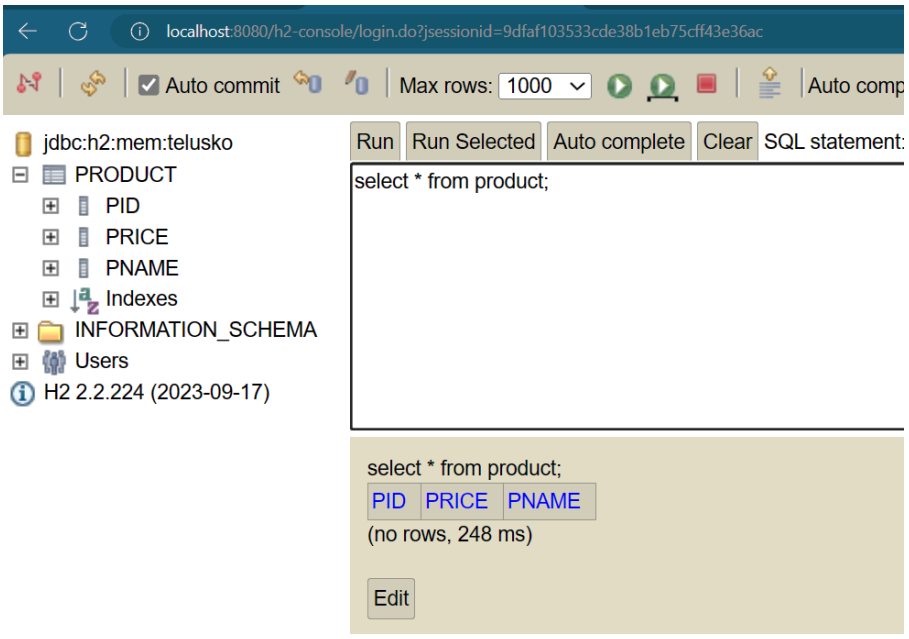
After running the project, we get this error, which specifies our product model class object is not managed. It's likely a Hibernate error if you get a message that also it can't find an identifier for the class to make the table. Hibernate is used by Spring Data JPA following the

JPA standards, even though we didn't use it directly.

To resolve this error in the model class, make the product ID the primary key, add the **@Entity** annotation to the class declaration, and add the **@Id** annotation to the product ID.

```java
package com.telusko.SimplewebApp.model;

import org.springframework.stereotype.Component;

@Component
@Entity
public class Product {

    @Id
    private int pid;
    private String Pname;
    private long price;

```

After the successful testing, our project is running with no errors, i.e., Tomcat is running on port 8080. Also, we got our table with product as the table name and other variables as parameters.

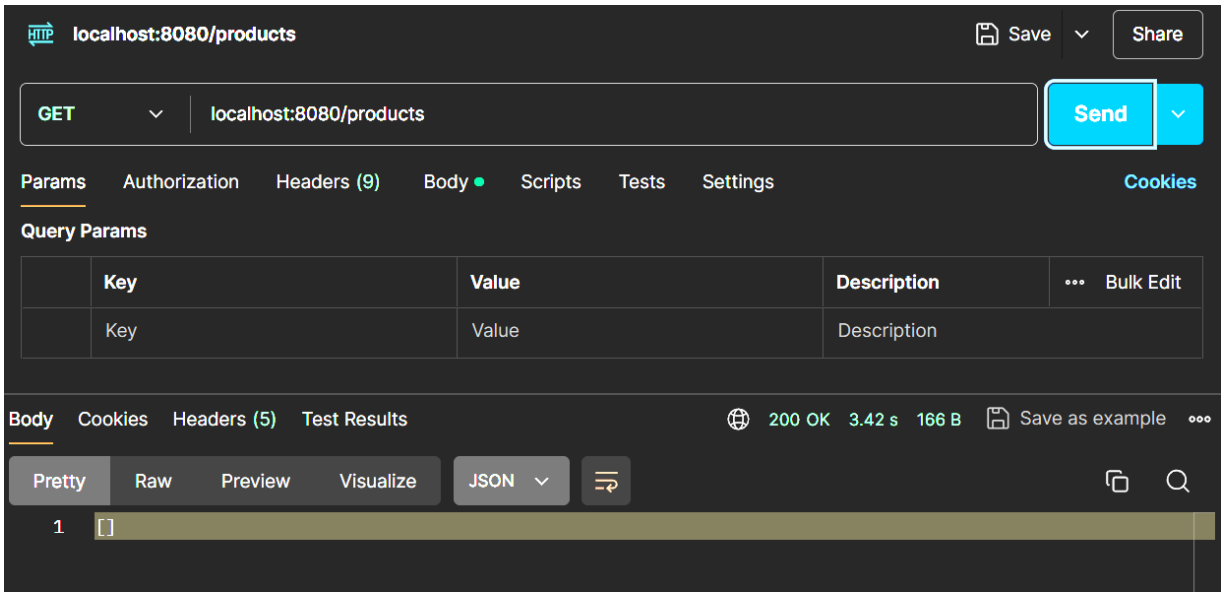


Trying out the H2 Database

To see the products in the table, go back to the H2 database by firing a SELECT query.

Before that, we will use the Postman tool as a client to get and add data to your database before running the SELECT query.

Using the Postman and hitting the GET request first, we will fetch all the products available in the database.

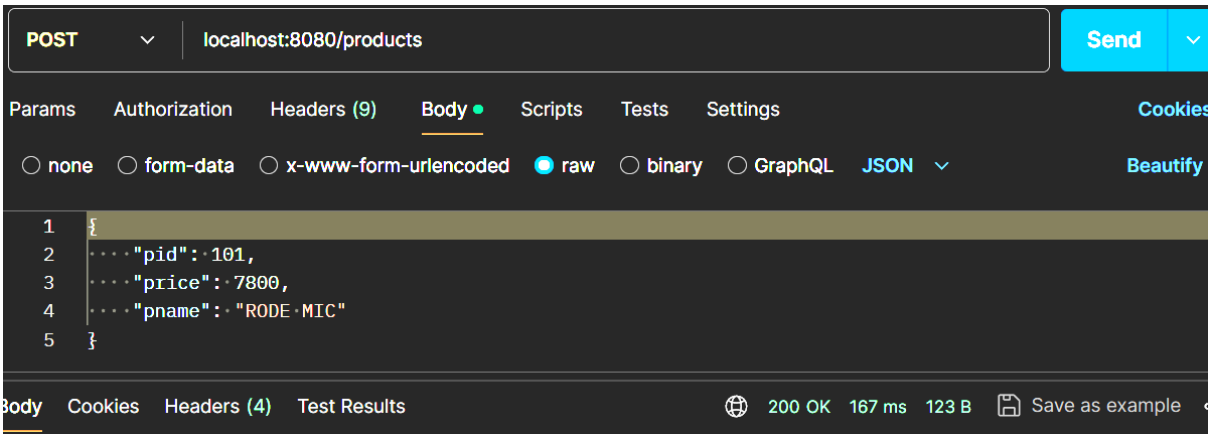


As we can see, we have an empty table with no data in it.

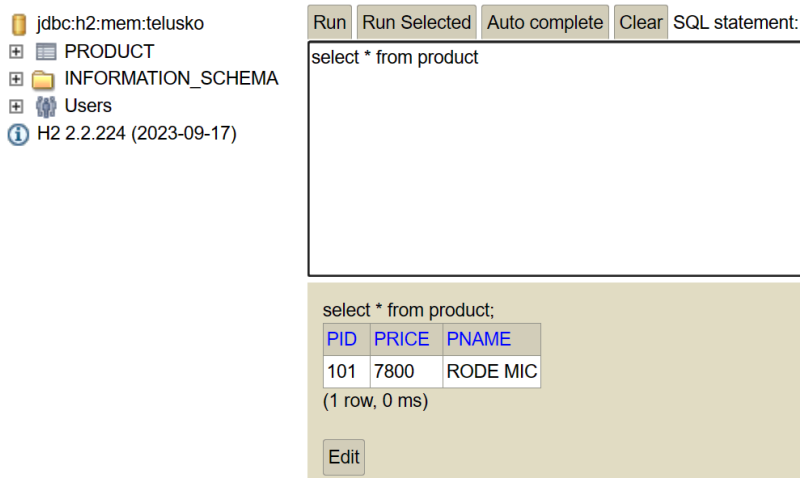
Let's perform some CRUD operations on it with the help of Postman acting as a client.

ADD Product:

By sending the POST request with JSON data into it, we will add the product..

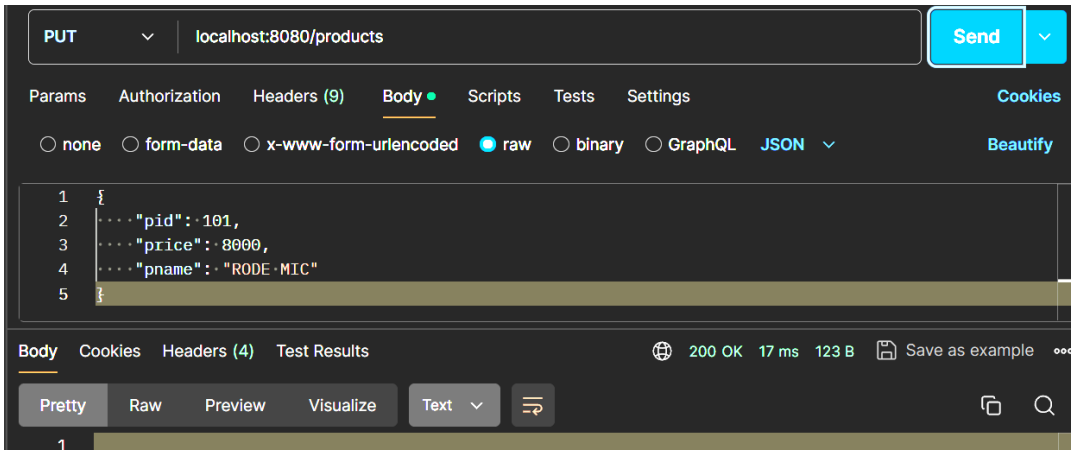


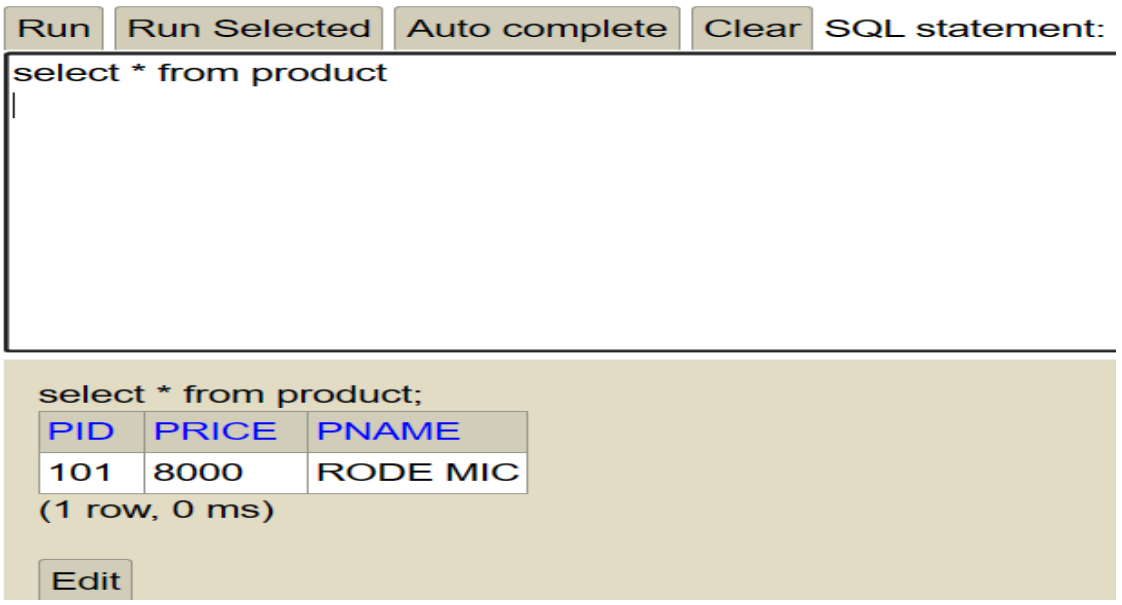
Now our product is added successfully in the database, as it shows status code 200. Let's check in the H2-Database.



Update Product:

Now we will update this existing product using Postman by sending the Put request and see the corresponding output in the H2 database to see if we are really getting those changes reflected into it as well or not.





You may be wondering how all of this query firing and product data addition and deletion in the database happen in the background. To see how queries are created, simply add more properties to the properties; you will then be able to see the queries that are being executed.

application.properties:

```
spring.datasource.username=navin
spring.datasource.password=telusko
spring.application.name=SimplewebApp
spring.datasource.url=jdbc:h2:mem:telusko
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.show-sql=true
```

If we want our in-memory database to be secure, we can also change the username and password as done in the above file, along with adding the property.

spring.jpa.show-sql=true

which will enable the queries that are visible in the backend of our project console.

```
Hibernate: select p1_0.pid,p1_0.pname,p1_0.price from product p1_0
Product [pid=101, Pname=RODE MIC, price=7800]
Hibernate: select p1_0.pid,p1_0.pname,p1_0.price from product p1_0 where
p1_0.pid=?
Hibernate: insert into product (pname,price,pid) values (?,?,?)
Hibernate: select p1_0.pid,p1_0.pname,p1_0.price from product p1_0 where
p1_0.pid=?
Hibernate: select p1_0.pid,p1_0.pname,p1_0.price from product p1_0 where
p1_0.pid=?
Hibernate: update product set pname=?,price=? where pid=?
Hibernate: select p1_0.pid,p1_0.pname,p1_0.price from product p1_0
```

The only drawback I could find with the in-memory databases in our case, H2, is that any small changes we made also had an impact on the database, and all of the data we had stored disappeared when we reloaded our project.

We will begin a fully functional project in the upcoming chapter, using React for the front end and Springboot for the backend, along with additional modules.

Up until then Continue to learn, and have fun coding 😊....