**Lab 5**

**Database interaction with Spring Boot & JPA**

In the previous SIG, we made the ProductService which let us get, put and and post drinks to a repository. This repository was however not defined in a database, but kept in a Java class. In the next two labs we will modify the ProductService to interact with a database instead.

We will be using Spring Data JPA. JPA stands for Java Persistance API. JPA is quite extensive and complex, but fortunately Spring Boot does a lot of the actual JPA work for us. In fact, our first basic implementation will have hardly any JPA code at all.

This lab will cover the basics of Spring Data JPA. The following link provides a comprehensive reference guide, and is a good starting point if you ever want to do something more complex: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

**Setting up dependencies**

Open /lab5/ProductService-start in IntelliJ or your favorite editor. This start situation is a slightly modified version of the end situation of Lab 3 from the previous SIG.

For our database interaction, we will need to add two maven dependencies in the pom.xml. Make sure to add them inside the “dependencies” element.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
</dependency>

**Preparing the Product Class**

If we want to store products in the database, we first need to let Spring know that Product is a persistable class. Additionally, we need to add some extra information about how a Product java object should map to a database. At the very least, we should specify a primary key.

We can do all this in the Product class using annotations in the Product class. Replace the first part of the class (everything up to and including “private string productId”) with the following code (goes on on the next page!):

import javax.persistence.\*;  
@Entity  
public class Product {  
  
 //private static final Long serialVersionID = 1L;  
 @Id  
 @GeneratedValue(strategy= GenerationType.*AUTO*)  
 private Long id;  
 @Column(unique=true)   
 private String productId;

We added an annotation @Entity. This annotation lets Spring know that Product is a persistable class.

We added a new variable id and added annotations to make it the primary key and have the value be generated.

Finally, we added an annotation to productId to specify that the corresponding column should have a unique constraint attached.

There are many more annotations that we can put on our variables to influence what the corresponding datamodel will look like. For instance, we can modify the column names, data types, and add constraints. If we have other entities that correspond to other tables in the database, we can specify relations between those entities. We won’t go into detail here, but there are ample examples online.

**Creating the Repository**

We now need to provide some functionality that actually lets us store and retrieve products from a database. This is surprisingly easy. Create a new Interface (not a class!) in the Domain package. Call it ProductRepository. The code should look like this:

package nl.groothandel.service.domain;  
  
import java.util.Optional;  
import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface ProductRepositoryDb extends JpaRepository<Product, Long>{  
 Optional<Product> findByProductId(String productId);  
}

We are extending the interface JpaRepository, which contains some basic CRUD methods. In addition to these method, we added a method to find a product by productId.

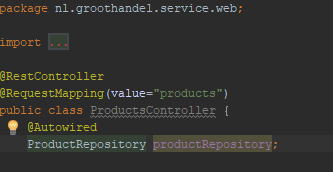
For those unfamiliar with Java: An interface in Java is somewhat like a package specification in PL/SQL. It specifies the exposed methods, but doesn’t provide an actual implementation.

You might expect that we now have to write an implementation for this interface, but Spring Boot can actually do that for us. Even our custom method gets implemented automatically because Spring can infer what we want it to do by its name (more on this later!). We now actually have all we need to interact with a database. All we need to do now is modify the controller to call our newly defined repository.

**Modifying the Controller**

All we need to do now is modify our Rest controller so that it actually calls our new Repository. Currently,it still uses the class ProductRepositoryImpl. We won’t be using this class anymore, so start by deleting it entirely.

Open the ProductController class. It should have a lot of errors, since we just deleted the ProductRepositoryImpl. Change the type of the variable productRepository to ProductRepository and add the @Autowired annotation.



There are still a lot of errors. The ProductRepository interface doesn’t expose the same methods as the ProductRepositoryImpl used to, so we need to replace those. Replace getProducts().values() with getAll() and getProducts().get(productId) with findByProductId(productId).get(). This takes care of the GET methods.

The POST and PUT methods are a bit more involved. Replace the methods with the following code (code block goes on on the next page!):

@RequestMapping(method=RequestMethod.*POST*)  
//@ResponseBody  
public ResponseEntity<Product> addProduct(@RequestBody Product product,  
 UriComponentsBuilder builder) {  
 Optional<Product> productDb = productRepository.findByProductId(product.getProductId());  
 if (!productDb.isPresent()) {  
 product = productRepository.save(product);  
 final HttpHeaders headers = new HttpHeaders();  
 final URI uri = builder.path("drink/{id}").buildAndExpand(product.getProductId()).toUri();  
 headers.setLocation(uri);  
 return new ResponseEntity<>(product, headers, HttpStatus.*CREATED*);  
 } else {  
 return new ResponseEntity<>(product, HttpStatus.*CONFLICT*);  
 }  
}

@RequestMapping(value = "/{productId}", method=RequestMethod.*PUT*)  
//@ResponseBody  
public ResponseEntity<Product> modifyProduct(@PathVariable("productId") String productId,  
 @RequestBody Product product,  
 UriComponentsBuilder builder) {  
 Optional<Product> productDb = productRepository.findByProductId(productId);  
 if (productDb.isPresent()) {  
 product.setId(productDb.get().getId());  
 product = productRepository.save(product);  
 final HttpHeaders headers = new HttpHeaders();  
 final URI uri = builder.path("drink/{id}").buildAndExpand(productId).toUri();  
 headers.setLocation(uri);  
 return new ResponseEntity<>(product, headers, HttpStatus.*OK*);  
 } else {  
 return new ResponseEntity<>(product, HttpStatus.*NOT\_FOUND*);  
 }  
}

Note that we use the save method in both cases. This is because the JpaRepository doesn’t expose a separate insert and update method. The save either inserts or updates, depending on whether a record with the same primary key already exists. This is also the reason we need to look up the id in the PUT method. This situation is not ideal. Later in this lab, we will extend our ProductRepository with actual update and insert methods.

**Run it!**

We should now have our first working version! Run the application.

Post some data at localhost:8081/products with postman or another tool. Here is a sample JSON body you can use. Note that we don’t provide an id, but we get the generated value in the response!

{"productId":"PUNKIPA",  
 "name":"Punk Ipa",  
 "type":"Bier",  
 "brand":"BrewDog",  
 "bottleSize":"33cl",  
 "abv":"123",  
 "price":"1"  
 }

Try out the get and put operations as well.

You might be wondering where the data is going, since we didn’t configure a connection to an actual database. The answer is that, unless we specify otherwise, Spring Boot spins up an in-memory database for us. This is a very nice feature for development purposes, but our data doesn’t actually get persisted; it will be dissappear upon a restart of the application. For most production purposes, this will be insufficient. We will go into how to connect to an actual database later.

**Extending the repository with custom queries**

The ProductRepository lets us do some basic CRUD stuff, but it is still fairly limited. We already extended the repository a bit with the findByProductId method, which Spring Boot can implement for us because it adheres to a naming convention. Let’s explore what else we can do.

List<Product> findByBrandIgnoreCaseAndPriceBetween(String productId, Double priceFloor, Double priceCeiling);  
  
@Query("Select p from Product p where (p.price >= ?1 or ?1 = null) and (p.price <= ?2 or ?2 = null)")  
List<Product> customQuery(Double priceFloor, Double priceCeiling);  
  
@Query(value = "select \* from product where (price >= ?1 or ?1 is null) and (price <= ?2 or ?2 is null)", nativeQuery = true)  
List<Product> nativeQuery(Double priceFloor, Double priceCeiling);

All three methods do similar things, but they are constructed in different ways.

* The first method shows of the strength and extensiveness of the naming convention way.
* The second method uses the @Query annotation to specify a custom JPQL query(java persistence query language).
* The third method also uses @Query, but specifies nativeQuery=true. This lets us write pure sql.

Especially the @query approach gives us full flexibility to write custom queries. More documentation on all these three methods, including a complete list of naming conventions, can be found in <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

If you want to try out the new methods, we need to call them in our controller. You can write some code yourself or just copy this:

@RequestMapping(value="/byBrand/{brand}", method=RequestMethod.*GET*)  
 public List<Product> retrieveProductByBrand (@PathVariable("brand") String brand, @RequestParam Map<String, String> queryParameters) {  
 Double priceFloor = queryParameters.get("priceFloor") == null? null : new Double(queryParameters.get("priceFloor"));  
 Double priceCeiling = queryParameters.get("priceCeiling") == null? null : new Double(queryParameters.get("priceCeiling"));  
 return productRepository.findByBrandIgnoreCaseAndPriceBetween(brand,priceFloor,priceCeiling); }  
  
 @RequestMapping(value="byPrice", method=RequestMethod.*GET*)  
 public List<Product> customQuery(@RequestParam Map<String, String> queryParameters) {  
 Double priceFloor = queryParameters.get("priceFloor") == null? null : new Double(queryParameters.get("priceFloor"));  
 Double priceCeiling = queryParameters.get("priceCeiling") == null? null : new Double(queryParameters.get("priceCeiling"));  
 return productRepository.customQuery(priceFloor,priceCeiling);  
 }  
  
 @RequestMapping(value="byPriceNative", method=RequestMethod.*GET*)  
 public List<Product> nativeQuery(@RequestParam Map<String, String> queryParameters) {  
 Double priceFloor = queryParameters.get("priceFloor") == null? null : new Double(queryParameters.get("priceFloor"));  
 Double priceCeiling = queryParameters.get("priceCeiling") == null? null : new Double(queryParameters.get("priceCeiling"));  
 return productRepository.nativeQuery(priceFloor,priceCeiling);  
 }  
}

Some sample requests:

<http://localhost:8081/products/byBrand/brewdog?priceFloor=1&priceCeiling=5>

<http://localhost:8081/products/byPrice?priceCeiling=5>

[http://localhost:8081/products/byPriceNative?priceFloor=1](http://localhost:8081/products/byPriceNative?priceFloor=1&priceCeiling=5)

**Extending the repository with custom DML**

The techniques in the previous chapter let us extend our repository with custom queries, but what about data manipulation? In this chapter we will write custom insert and update statements to use instead of the save method provided by the JpaRepository.

We can’t do this directly in our original ProductRepository. Instead, we will make a new interface ProductRepositoryCustom and extend ProductRepository with it. Make the new interface, and define methods insert and update in it:

package nl.groothandel.service.domain;  
  
public interface ProductRepositoryCustom {  
 public Product insert(Product product);  
 public Product update(Product product);  
}

Now extend the ProductRepository with the new ProductRepositoryCustom interface. Open ProductRepository and modify the line starting with “public interface”:

public interface ProductRepository extends JpaRepository<Product, Long>, ProductRepositoryCustom{

Unfortunately, Spring Boot can’t implement this for us automatically. We have to write an implementation of this interface, and implement the methods with JPA. The upside is that JPA is very extensive, which gives us a lot of flexibility. The downside is that we actually have to understand JPA. We will provide the implementation, but won’t go into the details.

Write the implanting class. Name it ProductRepositoryCustomImpl. Use the following code:

package nl.groothandel.service.domain;  
  
import org.springframework.stereotype.Repository;  
import org.springframework.transaction.annotation.Transactional;  
  
import javax.persistence.\*;  
import java.util.List;  
  
@Repository  
@Transactional  
public class ProductRepositoryCustomImpl implements ProductRepositoryCustom{  
  
 @PersistenceContext  
 EntityManager em;  
  
  
 @Override  
 @Transactional  
 public Product insert(Product product) {  
 try {  
 em.persist(product);  
 }  
 catch (Exception e){  
 throw e;  
 }  
 return product;  
 }  
  
 @Override  
 @Transactional  
 public Product update(Product product) {  
// Product productDb = em.find(Product.class, product.getId(), LockModeType.PESSIMISTIC\_WRITE);  
 Query q = em.createNativeQuery("select pdt.id from product pdt where pdt.PRODUCT\_ID = :productId for update", "IdValueMapping");  
 q.setParameter("productId",product.getProductId());  
 List<Long> ids = q.getResultList();  
 if (!ids.isEmpty()){  
 product.setId(ids.get(0).longValue());  
 product = em.merge(product);  
 return product;  
 }  
 else throw new EntityNotFoundException("Product with id "+product.getProductId()+" was not found");  
 }  
}

This code won’t work yet. Note that we provide a parameter “IdValueMapping” to em.createQuery in the update method. This mapping will specify what type the query result should be mapped to. We need to define this mapping in the product class using an annotation. Open the Product class and paste the following annotation directly under “@Entity”:

@SqlResultSetMapping(  
 name = "IdValueMapping",  
 classes = @ConstructorResult(  
 targetClass = Long.class,  
 columns = {  
 @ColumnResult(name = "id", type = Long.class)}))

Now our code should work. We just need to implement it in the controller. Open the ProductController and modify the code to call the newly defined update and insert statements:

Replace the body of the PUT method with the following:

try {  
 productRepository.update(product);  
 final HttpHeaders headers = new HttpHeaders();  
 final URI uri = builder.path("drink/{id}").buildAndExpand(productId).toUri();  
 headers.setLocation(uri);  
 return new ResponseEntity<>(product, headers, HttpStatus.*OK*);  
} catch (EntityNotFoundException e) {  
 return new ResponseEntity<>(null, new HttpHeaders(), HttpStatus.*CONFLICT*);  
}

and POST with the following:

try {  
productRepository.insert(product);  
final HttpHeaders headers = new HttpHeaders();  
final URI uri = builder.path("drink/{id}").buildAndExpand(product.getId()).toUri();  
headers.setLocation(uri);  
 return new ResponseEntity<>(product, headers, HttpStatus.*CREATED*);  
}  
catch(Exception e){  
 return new ResponseEntity<>(product, HttpStatus.*CONFLICT*);  
}

You should now be able to run your application again. Test the new insert and update statements. Do they work? How about the error handling? Try to update something that doesn’t exist. What http response status do you expect? What are you getting?

For some reason the try/catch construct in the controller doesn’t work as expected. We haven’t been able to figure out exactly why this is the case.

**Connecting to an actual database**

It’s nice to see all dml and queries working on an in-memory database, but the more interesting case is of course to connect to an actual database. Once connected, all the above will work as expected (test it out for yourself!).

Now for the actual connection. First we need a database. Provided you have docker installed (and running!), run:

docker run --name MySQLDB -p 3306:3306/tcp -e MYSQL\_ROOT\_PASSWORD=mysql -e MYSQL\_DATABASE=springboot -e MYSQL\_USER=springboot -e MYSQL\_PASSWORD=springboot -d mysql:latest

This will spin up a MySQL database, really really fast, and create user springboot with password springboot. Also, port 3306 in the docker container will be exposed to port 3306 on the local host (3306 is the default port on which the MySQL database is listening).

Next, we will need to connect the Spring Boot application to the now running database. We need an extra dependency in the maven pom to do so; add the following dependency to the pom:

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>

We’re not done just yet, since we still need to add connection information to application.properties file (host, user, password, driver). To do this, add the following lines to the application.properties file:

spring.jpa.hibernate.ddl-auto=create

spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useSSL=false&allowPublicKeyRetrieval=true  
spring.datasource.username=springboot  
spring.datasource.password=springboot  
spring.datasource.driver.class=com.jdbc.mysql.driver

The top line of the added configuration tells Spring Boot to always execute all ddl statements when spinning up the application (create will destroy all previous data in the schema; use update to keep the schema intact).

Use your previous tests to extensively test this new setup. What did you find?

The application is still not complete. For instance, we made column ‘id’ the primary key and added the following line of code to add a generator for the value:

@GeneratedValue(strategy= GenerationType.AUTO)

However, the generator created is not a database construct and therefore over multiple sessions the id-values wil get duplicated (which will result in errors due to the column being the primary key and thus unique).

We can use annotations to tell Spring Boot to create a generator (i.e. sequence in oracle, or another construct in MySQL) for you in the database. To do this replace the current ‘@GeneratedValue’ annotation with:

@GeneratedValue(  
 strategy= GenerationType.*AUTO*,  
 generator="native"  
)  
@GenericGenerator(  
 name = "native",  
 strategy = "native"  
)

This will work for both MySQL and Oracle. There are a lot of ways to steer the generation of generators (i.e. creating specifically named sequences in Oracle) in the database to your own liking. Examples of these are available through simple google-searches online.

One rather strange thing that we cannot as of yet explain is why the productId column is not unique when using MySQL, even though we explicitly added the unique annotation to the productId column definition. When using Oracle the unique key constraint is created.