JPA and Spring Data Exercise

JPA

JPA is a Java standard for mapping Java objects to database tables. It is an ORM framework (Object to Relational Mapping).

A pioneer in Java ORM is the open source project Hibernate. It was often used as an ORM solution in Java before the JPA standard specification. Now it still has its own API, but is also an implementation of the JPS specification.

Read about what Hibernate says about ORM:

http://hibernate.org/orm/what-is-an-orm/

JPA is the Java standard specification of Object Relation Mapping. Read about JPA here (read a little bit now, read all of it as a stretch task):

https://www.javacodegeeks.com/2015/02/jpa-tutorial.html

Spring Data

Spring Data is a Spring Framework project for integrating with databases. Read a little about Spring Data here:

http://projects.spring.io/spring-data/

Spring Data is for many different types of databases, a lot of them NOSQL databases. What we are using in this exercise is really Spring Data JPA for integrating with SQL databases. Read about Spring Data JPA here:

http://projects.spring.io/spring-data-jpa/

Exercise 1 – Create a Spring Boot project that uses Spring Data JPA

Create a new Spring Boot project.

Select the following dependencies:

Web – Web
SQL – JPA
SQL – H2

Create a new Java class with the name **AdminController**.

Annotate the class with **@RestController**.

Create a new method in **AdminController** like this (just to be able to see if the application is running ok when not interacting with the database):

```
@GetMapping("/")
public String hello() {
    return "App is running!";
}
```

Create a new class with the name **Admin**.

Add these variables to the **Admin** class:

```
private long id;
private String username;
private String password;
```

Create getters and setter for these variables and also add these constructors:

```
public Admin() {};

public Admin(String username, String password) {
    this.username = username;
    this.password = password;
}
```

For JPA to work, we have to add a few annotations to the Admin class.

Add the annotation **@Entity** to the class (above the row with **public class Admin**…). This tells JPA that the **Admin** class is an Entity. This means there is a table in the database called Admin and that this table has fields for the variables in the Admin class (fields for id, username and password).

Add the following annotations above the declaration of the **private long id** variable:

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

These annotations mean that the id variable is the primary key of the table and that the value of id should be generated automatically.

Add these controller methods with GetMappings to the **AdminController** class:

```
@GetMapping("/admins")
public List<Admin> getUsers() {
    List<Admin> admins = (List<Admin>)repository.findAll();
    return admins;
}

@GetMapping("/add/{username}/{password}")
public String add(@PathVariable String username, @PathVariable String
password) {
    repository.save(new Admin(username, password));
    return "ok";
}

@GetMapping("/delete/{id}")
public Iterable<Admin> delete(@PathVariable long id) {
    repository.delete(id);
    return repository.findAll();
}

@GetMapping("/deleteAll")
public Iterable<Admin> deleteAll() {
    repository.deleteAll();
    return repository.findAll();
}
```

The repository variable will be marked red since it does not exist yet.
Add this autowired Repository to the **AdminController** class (inside the public class **AdminController** scope):

```
@Autowired
private AdminRepository repository;
```

The **AdminRepository** will be red and repository will be underlined since it does still not exist. We will fix that soon.

Remember when we used an autowired Interface before when adding a reference to a Repository in the Controller? And then we created a class that implemented that Interface and had implementations of all its methods where we put all the JDBC code and SQL statements inside those methods?

We will create an Interface here too, but we will not create a class that implements that Interface, and we will not create any methods with JDBC code thanks to the magic of Spring Data.

Create an Interface with the name **AdminRepository** that extends **CrudRepository<Admin, Long>**. Don't add any methods to this Interface.

Now the **AdminController** should be ok!

Now when the Interface **AdminRepository** exists, it will no longer be marked red in **AdminController**, and all the method references like **findAll**, **save**, **delete** and **deleteAll** will no longer be marked red because they now exist without us creating them!

By extending the **CrudRepository** with the **Admin** class and a **Long**, Spring Data has implemented all these methods automatically!

By using the **Admin** class when extending the Interface with **CrudRepository** we tell Spring Data that it should implement the methods for the **Admin** class, and expect a table in the database that corresponds to the variables in the **Admin** class. By also using a **Long** as the second argument, we tell Spring Data that the primary key in this table is of type **Long**.

This is enough for Spring Data to create the necessary methods to find, save, delete **Admin** objects in the **Admin** table in the database with the help of JPA.

Now, try out the web application by calling the methods in the **AdminController** like this:

At first, a call to:

http://localhost:8080/admins

will give a response with an empty list of Admin objects in JSPN format like this:

```
[]
```

A call to the add method, to add an Admin object with the username Admin1 and a password of 123 like this:

http://localhost:8080/add/Admin1/123

Will give **ok** as a response when the method uses Spring Data and JPA to insert a row into the **Admin** table in the database.

Try to look at the table again with a request to http://localhost:8080/admins and you will se a JSON response like this:

```
[{"id":1,"username":"user1","password":"pass1]
```

Try to add more admins, and try to delete an admin by calling the delete method with an id of an admin object like this:

http://localhost:8080/delete/1

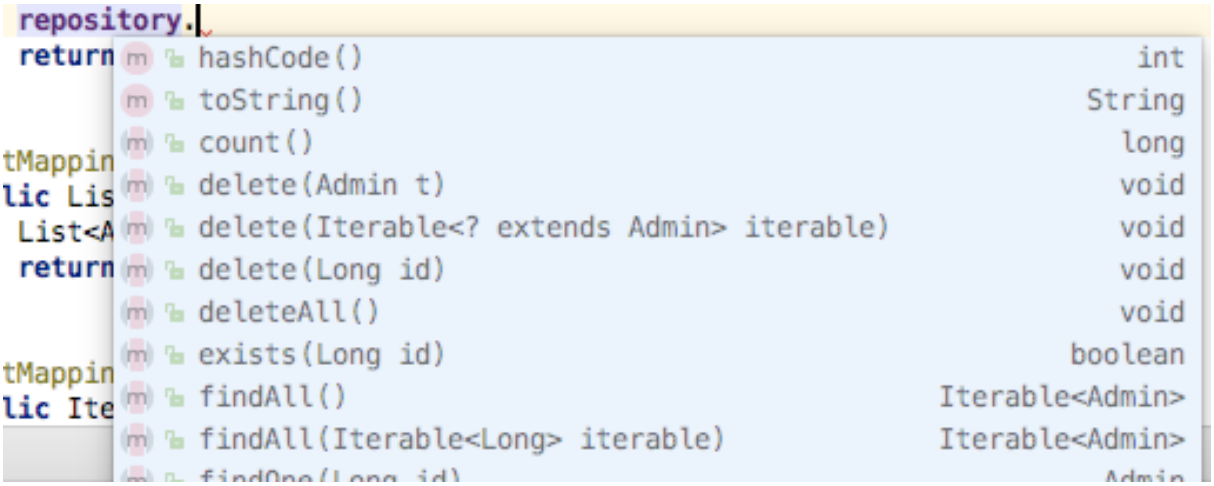
Or delete all rows in the table by calling:

http://localhost:8080/deleteAll

Look at all the automatically implemented methods you can use by entering this into any Controller method in **AdminController**:

repository.

The IntelliSense of IntelliJ IDEA will give you a list of all the methods that look like this:



Here you can see all the methods that are automatically created and that handles how the Admin object is found, created and saved to the Admin table in the database.
But what if you need another way to find an Admin object in the database?

Exercise 2 – Extend the Interface with your own methods

What if you need to find an Admin object by its **username**, like the SQL syntax **WHERE USERNAME = "Admin1"?**

Just create a method signature in the **AdminRepository** Interface like this:

```
List<Admin> findByUsername(String username);
```

Now you have given Spring Data a hint of what you want implemented, and Spring Data will automatically implement this method for you.

Try to use this new method by calling it with a **username** that exists in the database, and it will find all the results and return them in a List of Admin objects.

(Ok, a **username** field is usually unique, and a lookup for a unique **username** should normally only result in one unique object, but this is only a simple example and we haven't specified that this field should be unique)

Stretch task 1 – Read more about Spring Data JPA

Read more about Spring Data here:

http://docs.spring.io/spring-data/jpa/docs/1.11.1.RELEASE/reference/html/

Stretch task 2 – Try to create custom methods by using Spring Data keywords

You can create new methods by adding method signatures in the repository Interface with certain keywords.

The example in this exercise was **FIND BY username**, and this method is automatically created with this method signature in the **AdminRepository** Interface:

```
List<Admin> findByUsername(String username);
```

You could also try for example:

```
List<Admin> findByPassword(String password);
```

Stretch Task – create more methods using the keywords of Spring Data

Read more about the keywords here:

http://docs.spring.io/spring-data/jpa/docs/1.11.1.RELEASE/reference/html/#repository-query-keywords

Then try to add more methods with the help of these keywords and check if you can make them work by calling them from Controller methods.