Challenge module 6: Building a full fledged REST API

As the second part of the training, we are going to build a REST API. It allows you to register temperatures and retrieve a list of all registered temperatures.

Getting started

To get started, head to spring initializr and start by creating a boilerplate project. Make sure to choose Java 14 so that we can make use of the latest features.

Select the following dependencies:

- Lombok
- Spring web
- Spring data JPA
- H2 Database
- Validation (JSR-303)

This generates a skeleton that we can use. It has everything correctly configured to get started right away.

Lombok

Lombok is a project that reduces the amount of boilerplate you have to write for Java classes. It provides annotations like @AllArgsContructor, @Data, @NoArgsContructor that makes sure you don't have to write it. If you use Lombok, make sure to add the lombok plugin to IntelliJ in order to get proper syntax highlighting. Using Lombok is optional, maybe even not preferred if you don't have a lot of experience writing Java.

Spring web

Spring web provides us with everything we need for writing RESTful webservices. It also gives us jackson, a JSON marshaller for decoding and encoding Java objects to JSON and the other way around. It uses an embedded tomcat server that allows us to easily start our application.

Spring data JPA

This gives us annotations that we can use to map database columns / tables to Java objects. It provides us with ways to easily interact with a database and generate implementations for interacting with tables.

H2 Database

An in-memory database we are going to use. It can also write to a file, so that we can have real persistence if we want.

Validation

A project that uses annotations in order to put constraints on our input parameters, so that we can properly validate business constraints in our requests.

Implementing the REST endpoints

We start with two endpoints:

- An endpoint to register a new temperature. (PUT request)
- An endpoint to retrieve all registered temperatures. (GET request)

Use the following Spring documentation to setup the endpoints: Building a RESTful web service

For now, the controllers can use an in memory data structure like a List or HashMap. Make sure that if you register a new temperature and than retrieve all the temperature, you also retrieve the newly registered temperature. Also make sure there is proper request validation. A temperature cannot be lower than -50 and higher than 50. For this, you can use the validation package that we have initialized. Lastly, we want to know when the temperature / measurement was registered.

After implementing these two endpoints, add unit tests. Preferrably, think beforehand about the implementation. It is probably best to move the im memory data structure to a service or repository class. This allows us to use a mock in the test and easily swap it later for a database.

Writing integration tests with RestAssured.

For integration tests, we are going to use RestAssured. Follow the getting started guide on the page Getting started. Add the required dependencies to the pom.xml file so that maven can resolve the dependencies for us.

Create a Java test class and call it SpringIntegrationTest. Add the following annotation on top of the class:

```
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
```

Add the following property to the class:

```
@LocalServerPort
private int port;
```

We need this to know the port the API is started on.

Now, write an integration test. An example test could be that posting twice and than retrieving all measurements result in a list of two items with the recently posted temperatures. RestAssured provides a convenient given-when-then syntax, have a look at the docs to see how to use it.

Adding a persistence layer

To actually store the temperatures, we are going to add an H2 database. Use the following documentation to setup H2 with Spring: Set up H2

Right now, you have two options: Use spring-data-jdbc Spring data JDBC or JPA Spring data JPA. Jdbc is lower level and will let you write queries yourself.

The Java Persistence API provides a specification for persisting, reading, and managing data from your Java object to relational tables in the database. JDBC is a standard for Database Access. If necessary, this can be done as a group exercise.

If you chose JPA, you can generate the database scheme using the annotations, or providing a custom SQL file in resources called data.sql, which will be used to initialize on startup.

Adding mail functionality

Our product owner wants to receive an email every time a new temperature is registered. To enable this, we can use Simple Java Mail. It's a library that makes it easy to send email. As an SMTP service, we are going to use MailGun.

You need to take the following steps:

- Add the simple java mail dependency.
- Make a free account at mailgun.
- Go to sending and click on the sandbox domain
- Add your email as authorized receiver on the right
- Select SMTP for the settings (we need to configure simple java mail with these settings)
- Create a new Service class called MailService (which you need to annotate with @Service, so that it can be resolved by Spring).
- Add the following to the Main class:

```
@Bean
public Mailer mailer(
    @Value("${mailer.host}") String host,
    @Value("${mailer.port}") Integer port,
    @Value("${mailer.username}") String username,
    @Value("${mailer.password}") String password
) {
    // Return an instance of Mailer here. Take a look at the documentation
on how to do that.
    // Don't forget to provide the username and password for mailgun.
}
```

Since this is an external dependency, we need to add a bean (dependency) so that spring knows how to resolve the Mailer class whenever a class needs it as a dependency. The @Value annotation is used to read parameters from the application.properties file.

In the MailService class, implement a method that receives an email object and then sends the mail. Tip: Use the EmailBuilder provided by `Simple java mail`.

Now, as soon as the MailService is ready, we can use this service in other services. Add the MailService as a constructor argument to the temperature service and add it as a property. Create an instance of Email and send call the mailservice to send the mail.

Update the unit tests for this service layer and add tests for the MailService.

Upgrading Integration Tests to test end-to-end mailing.

Mailing can be difficult to integration test. We do not want to mock the mailer, because then we don't know if the integration actually works. We are going to make use of GreenMail. First, we need to install greenmail (or you can make use of the provided docker-compose.yaml in the root of the project. As soon as you reach this step, you can ask the trainer for help). Make use of the starting page of greenmail to set it up properly GreenMail homepage.

Extending the application

For some reason, it happends that a mismeasurement is recorded in the system, so we want the ability to manually correct them. Allow the system to be able to change temperatures. Alter the put request to be able to update an existing resource whenever it already exists.

Also, for our frontend it would be nice if our PUT request returns the new resource. You can imagine a frontend that allows you to update data afterwards. However, if we create a new resource and we don't know the id of the new temperature, we cannot update it afterwards.

Update the PUT endpoint to return the newly created / updated resource.