Web Applications 2

Lab 2: Setting up a basic service

Learning Objectives

By successfully completing this lab, students will develop the following skills:

- Use GitHub as a repository and a collaboration tool
- Create and deploy a simple web application
- Check input parameters
- Manage runtime exceptions and maps them to error descriptions
- Write a React client application and check it can properly access the server

Description

In order to support customers that have opened a ticket to receive support for a given product, several preliminary pieces of information need to be available: among these, there are product details and customer profiles.
Thinking of the overall problem and of the specific use cases that have been identified during the event storming process, define the basic set of data that these two structures should contain (exclude from these all security-related information). Then, create a basic service, consisting of all the relevant layers of a web application (data, back-end, front-end), capable of managing that information.

Steps

1. Select one group member as coordinator and ask her/him to create a GitHub private repository and to grant access permissions to all other group members. You may need to create free personal accounts, if you don't have one, yet. All produced code has to be committed to this repository.

2. Run IntelliJ IDEA and select File → New → Project from Version Control…: Select the repository URL in the dialog box and proceed.
   The repository will contain two modules: a Spring Boot web application named "server" and a React application named "client". Create the first module via File → New → Module…: choose Spring Initializr, name it "server", opt for the Kotlin language and the Gradle build tool. In the next screen select, as dependencies, Spring Web, Spring Data JPA, PostgresSQL Driver. Commit following the Conventional Commits specification: https://www.conventionalcommits.org/en/v1.0.0/
   Then push the commit to the Github repository.
   Create a second module: choose JavaScript / React and name it "client". Commit and push.

3. Start a docker container with the image postgres:latest and check that it properly exports port 5432 to the corresponding port of the host.

Using the IntelliJ Idea Database view, check that the database is accessible and create a couple of tables aimed at storing the product and customer profile data.

4. In the server module, edit the src/resources/application.properties file and add entries for the database connection url and credentials, and set the "spring.jpa.show-sql" key to true and "spring.jpa.hibernate.ddl-auto" to "validate".
   Divide the group in two subgroups of two people each and proceed doing concurrently points 5 and 6

5. Implement the server module, managing REST requests targeted to the following endpoints:
   - GET /API/products/ -- list all registered products in the DB
   - GET /API/products/{productId} -- details of product {productId} or fail if it does not exist
   - GET /API/profiles/{email} -- details of user profiles {email} or fail if it does not exist
   - POST /API/profiles -- convert the request body into a ProfileDTO and store it into the DB, provided that the email address does not already exist
   - PUT /API/profiles/{email} -- convert the request body into a ProfileDTO and replace the corresponding entry in the DB; fail if the email does not exist.

   All endpoints should return a valid JSON object and a corresponding HTTP status code.
   The application code should throw custom subclasses of RuntimeException to notify any error it may detect. In order to manage these exceptions, add a class similar to the following one:

```
@RestControllerAdvice
class ProblemDetailsHandler: ResponseEntityExceptionHandler() {
    @ExceptionHandler(ProductNotFoundException::class)
   fun handleProductNotFound(e: ProductNotFoundException) = ProblemDetail
        .forStatusAndDetail( HttpStatus.NOT_FOUND, e.message!! )
    @ExceptionHandler(DuplicateProductException::class)
   fun handleDuplicateProduct(e: DuplicateProductException) = ProblemDetail
        .forStatusAndDetail(HttpStatus.CONFLICT,  e.message!! )
}
```

   Commit code and push it to the repository.

6. Implement the client module, providing a minimal interface that allows to verify the exposed endpoints.
   Since the client is normally run from port 3000, and the server is running on port 8080, add, in the "package.json" file, the following line, before the last "}":

```
"proxy": "http://localhost:8080"
```

   This will forward the requests aimed at the exposed urls to the server module avoiding CORS error.
   Verify that the application works, then commit and push.

7. Once both the client and the server module are stable, pack the React application using command "npm run build": this will create, in the client module, a new subfolder named "build". Copy all the content of this subfolder into the server module, in the

folder named "src/main/resources/static". Stop the client module, run the server one and verify that opening the URL http://localhost:8080/ the client application is loaded and that it can properly interact with the API server

Submission rules

Download a copy of the zipped repository and upload it in the “Elaborati” section of “Portale della didattica”. Label your file “Lab2-Group<N>.zip” (one copy, only - not one per each team member). Work is due by Friday April 14, 23.59 for odd numbered teams, and by Friday April 21, 23.59 for even numbered ones.