Table Content

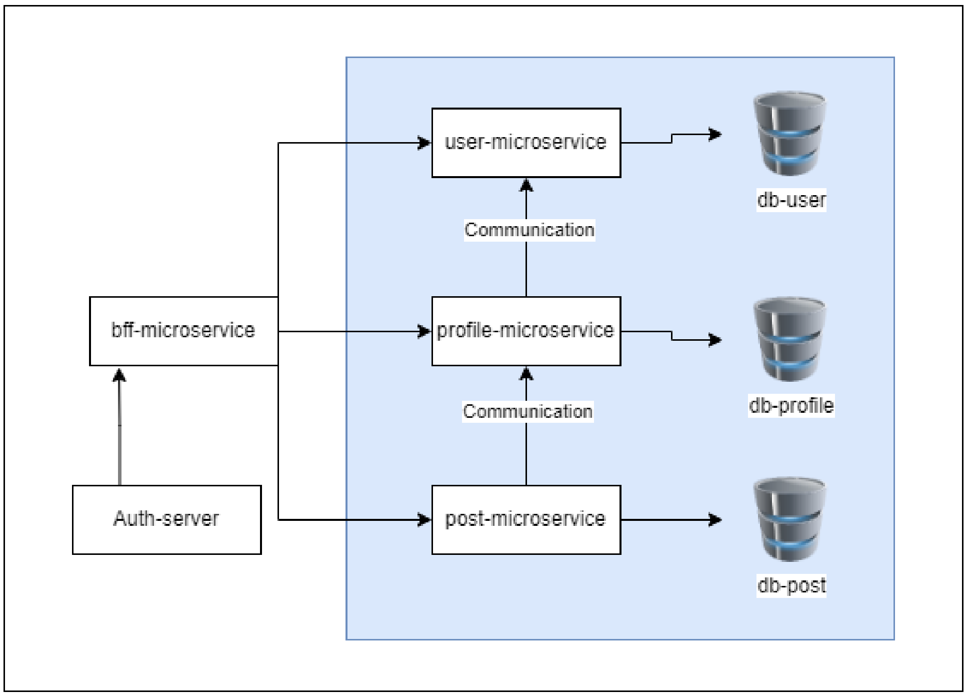
Ecole-IT

4AMS Microservices

Par : Goulmemei B. Amram

1) Project Schema

The project is about a LinkedIn project with 3 microservices (user-microservice, profile-microservice and post-microservice). The microservices have different databases and can communicate among themselves via an http link. The diagram of the project is as follows:



2) User Microservice

On port **8081,** Database name: **db-user**

The User Microservice was named “user-service”. The database is called “db-user” This section had some sections:

A- AREA 1: Do not create User under 18 years

In order for this to be achieved, a testing function was created and takes in an integer and verifies if the integer is less than 18. If the integer is less than 18, it returns “*true*” else it returns “*false*”. This function is then called at the Creation or Updating of a user. If the functions return a confirmation of user age under 18 years, the user is automatically not registered. The function is at follows :

```
// Test if user is aged >= 18
public boolean ageValid(int age){
    if (age < 18) {
        log.info("User with age " + age + " Rejected. User should be at least 18 years
old");
        return false;
    } else {
        return true;
    }
}
```

B- AREA 2: Authentification

This was made with a function which checks the existence of an “username” and a corresponding “password” in the database

The CRUD (Create, Read, Update, Delete) of user are available at the links :

Link	Method	State

http://localhost:8083/api/users	POST	Create a new user
http://localhost:8083/api/users	GET	Read all users
http://localhost:8083/api/users/{id}	GET	Read data of a single user
http://localhost:8083/api/users/{id}	PUT	Update data of a single user
http://localhost:8083/api/users/{id}	DELETE	Delete the data of a user
http://localhost:8083/api/users/authenticate	POST	Authenticate a user

3) Profile Microservice

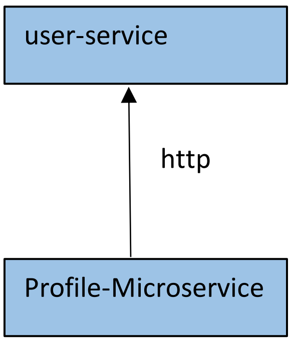
On port **8082,** database name: **db-profile**

The Profile microservice project was named “profile-microservice”.

This microservice contains the crud for profiles and profile connections.

A- Profile

For a profile to be created, we need to have the user existing. The table in database is called “*profiles*”. This has been done through the communication between the microservice user-service and profile-microservice.



In the profile-microservice, a user DTO was created and a user service on the name “userServiceE”. This service contains a function to test if the introduced ID exists in the database of the external service (user-service). The sample function is as follows :

```
// Find a user if it exists
public UserDTOE findUser(Long profileId) {
    // Use the service name as hostname
    String url ="http://localhost:8081/api/users/" + profileId;
    return restTemplate getForObject(url  UserDTOE class);
}

public boolean userTest(Long id){
    if (id == null){
        log.info("User ID CANNOT be null");
        return false;
    } else {
        if (findUser(id) == null){
            log.info("No user found with ID : " + id);
            return false;
        } else {
            return true;
        }
    }
}
```

This function is later called in the "ProfileService.java" file and class to test if the user is existent. The function returns "true" if the profile is found and "false" if it is not found. If the function returns true, the profile will go ahead and be saved. If it returns false, Post won't be saved and will give a message "No userfound with ID : *data_given*"

The CRUD (Create, Read, Update, Delete) of profiles are available at the links :

Link	Method	State
http://localhost:8083/api/profiles	POST	Create a profile
http://localhost:8083/api/profiles	GET	Read all profiles
http://localhost:8083/api/profiles/{id}	GET	Read data of a single profile
http://localhost:8083/api/profiles/{id}	PUT	Update data of a single profile
http://localhost:8083/api/profiles/{id}	DELETE	Delete the data of a profile

B- ProfileConnections

For a profile connection in database as table is called : "*profiles_connections*". For a profile_connection to be created, the profiles_id given must be valid in database. To do this, the profile Repository was called in the profileConnection Service file and a function injected at every manipulation to test if the profiles are valid. If they aren't, the connection won't be created and a message error will be sent.

The CRUD (Create, Read, Update, Delete) of profiles are available at the links :

Link	Method	State
http://localhost:8083/api/profile_connections	POST	Create a profileConnection
http://localhost:8083/api/profile_connections	GET	Read all profileConnections
http://localhost:8083/api/profile_connections/{id}	GET	Read data of a single profileConnection
http://localhost:8083/api/profile_connections/{id}	PUT	Update data of a single profileConnection
http://localhost:8083/api/profile_connections/{id}	DELETE	Delete the data of a profileConnection

4) Post Microservice

On port **8083,** database name: **db-post**

The post microservice project was named "post-microservice". This microservice is for the management of posts and comments of posts.

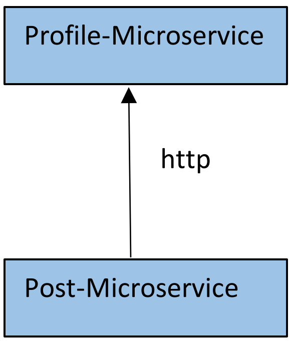
A- Post

The Post class in model is called "Post" and in the database, the table is called "posts".

The attributes placed in the code are:

```
private Long id;
private Long profile_id;
private String title;
private String content;
private Date creation_date;
```

For a post to be made, there is an attribute called "*profile_id*" which refers to the id of the profile making the post (which is required*). A post can't be made if the profile_id is non existant. Inorder to achieve this, the communication link between "*profile_microservice*" and "*post_microservice*" was managed as follows :



In the post-microservice, a profile DTO was created and a profile service on the name “profileServiceE”. This service contains a function to test if the introduced ID exists in the database of the external service (profile-microservice). The sample function is as follows :

```
// Find a profile if it exists
public ProfileDTOe findProfile(Long profileId) {
    // Use the service name as hostname
    String url = "http://localhost:8082/api/profiles/" + profileId;
    return restTemplate getForObject(url, ProfileDTOe class);
}

public boolean profileTest(Long id){
    if (id == null){
        log.info("Profile ID CANNOT be null");
        return false;
    } else {
        if (findProfile(id) == null){
            log.info("No profile found with ID: " + id);
            return false;
        } else {
            return true;
        }
    }
}
```

This function is later called in the “PostService.java” file and class to test if the profile is existent. The function returns “true” if the profile is found and “false” if it is not found. If the function returns true, the post will go ahead and be saved. If it returns false, Post won’t be saved and will give a message “No profile found with ID : *data_given*”.

The CRUD (Create, Read, Update, Delete) of post are available at the links :

Link	Method	State
http://localhost:8083/api/posts	POST	Create a post
http://localhost:8083/api/posts	GET	Read all posts
http://localhost:8083/api/posts/{id}	GET	Read data of a single post
http://localhost:8083/api/posts/{id}	PUT	Update data of a single post
http://localhost:8083/api/posts/{id}	DELETE	Delete the data of a post

B- Comments :

The class model in the post microservice is called “Comment” and in the database it is called “comments”.

The attributs are :

```
private Long id;
private Long post_id;
private Long profile_id;
private String content;
private Date creation_date;
```

For a comment to be made, the constraint is that the profile_id and the post_id MUST be existent. So the same process described above in the post section for profile verification is the same that was made to verify an existing profile at the creation of the comment. The second verification to do is the verification of the post if it really exists. For this, the "PostRepository" was called in the CommentService class and a verification function was applied as follows :

```
// Calling the post Repository to test if the post injected exists
private final PostRepository postRepository;

// Testing if profile is not null and exists
// it shall be done with the code -> "profileServiceE.profileTest(data)"

    // Testing if id exist
    public boolean dataPostExists(Long id) {
        if (id == null){
            log.info("The Post ID CANNOT be NULL");
            return false;
        } else {
            if (postRepository existsById(id)) {
                return true;
            } else {
                log.info("Post " + id + " does not exist");
                return false;
            }
        }
    }
```

With this function, at the creation of a comment or at the updating of a comment, if the post id is modified to a non existing id in the table post, an error will be thrown and the comment will not be created and the message given will be "Post *data_given* does not exists".

The CRUD (Create, Read, Update, Delete) of comments are available at the links :

Link	Method	State
http://localhost:8083/api/comments	POST	Create a comment
http://localhost:8083/api/comments	GET	Read all comments
http://localhost:8083/api/comments/{id}	GET	Read data of a single comment
http://localhost:8083/api/comments/{id}	PUT	Update data of a single comment
http://localhost:8083/api/comments/{id}	DELETE	Delete the data of a comment

5) Backend For Frontend (BFF) Microservice

6) Documentation

The documentation of the API was made with Swagger. In order for this to be achieve, the dependencies of swagger were copied to the "pom.xml" file of each microservice as follows :

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
```

Inside the folders to be documented, the documentations were now done using the annotation "***@Tag(...)***" or "***@Operation(...)***" or "***@ApiResponse(...)***".

The documentation is available and can be viewed via the following links for the different microservices

Link	Corresponding microservice
localhost:8081/swagger-ui/index.html	Swagger documentation for "user-microservice"
localhost:8082/swagger-ui/index.html	Swagger documentation for "profile-microservice"
localhost:8083/swagger-ui/index.html	Swagger documentation for "post-microservice"
localhost:8080/swagger-ui/index.html	Swagger documentation for "bff-microservice"

7) Docker

Started but not finished.

In all the microservices, a docker file called "Docker" was placed having a code permitting to locate the .jar file. An example of the code in the file is shown below with what was in the microservice user:

```
# Dockerfile for user-microservice

FROM adoptopenjdk:21-jre-hotspot

WORKDIR /app

COPY target/user-microservice-0.0.1-SNAPSHOT.jar /app/user-microservice.jar

EXPOSE 8081

CMD ["java", "-jar", "user-microservice.jar"]
```

All the microservices were placed in one directory named "linkedIn". In that directory, a docker file named "docker-compose.yml" was created and a code inserted for parameter.

8) Examples

Examples on how to insert are founded in every microservice in the file **"src/main/resources/namerequette.http"**