UeK 223

Thema: Implementing a multi-user application in an object-oriented manner

Dokumentinformationen

Dateiname: uek223_documentation_group2

Autoreninformationen

Autor: Shawn Lacarta, Nuwera Mohammad, Matijas Polazarov

Table of Content

1 Introduction

This document describes our group work for module 223. Our mission is to implement a Spring Boot application with a Postgres database that can create, delete, and edit multiple users. In addition, we also need to maintain the security vulnerabilities to prevent false login attempts (authentication and authorization). Our group specific tasks are to display additional information about a user for administrators and the user himself and to arrange it according to arbitrary parameters. Special attention should be paid to the fact that we do not violate the ACID principles. In addition, we need to create a domain-, class- and sequence diagram. Towards the end we need to test our web application and document everything.

2 Planning

2.1 GANTT

	KW 44	KW 45	KW 46
Planning and Conventions			
Creating Git Issues & ReadMe			
Create Domain Model			
Work on Endpoints			
Implement Swagger & Logger			
Create Use Cases & Sequence Diagram			
Do Testing			
Work on Documentation			
Final Check			

2.2 Convention

2.3 Naming Convention

Before starting the code, we agreed on following these Convention for the naming of files and methods. This tables contains the conventions:

Database

Topic	Convention	Example
table name	Snake casing	role_authorities
column name	Snake casing	last_name

Code

Topic	Convention	Example
package name	Camel casing	userProfile
class name	Pascal casing	UserProfileController
methods	Camel casing	addUserProfile
fields	Camel casing	birthDate

2.4 GIT Convention

For the Git convention, we decided to write the following for each commit message: first, a verb describing exactly what you did with the commit. For example: "implemented test function", "fixed bug return value from function test", etc. For the language of the commits we decided to use English, because we did the whole project in English (JavaDoc, Documentation and Planning.). For the Git Branches etc. we have defined that we always create a new branch for each "issue" / task. This branch is named either "feature/..." or "bugfix/... we use for programming a function, extending a function / class, etc. "Bugfix/..." we use for bug fixes, handling of errors, etc. These branches are later merged into the develop branch if we have approved the whole thing in the team. The develop branch is the branch that comes directly after the main, so you don't break everything in the main. The smartest way is to merge everything into the develop branch and then merge into the main branch at the end of the project.

In short:

Commit

Topic	Example
Endpoints	“added method in class <insert> and <insert> for <reason>”
Fixed bug	“fixed bug in method <insert> so method (for example) returns right output”
Merge conflicts	“solved merge conflicts with branch <insert>”

Branch

Topic	Example & Description
Adding something new	“feature/<insert>” → The feature that you are going to add to the method is meant to be in the braces.
Fixing Bugs	“bugfix/<insert>” → The feature that you are going to edit or change completely is meant to be in the braces.

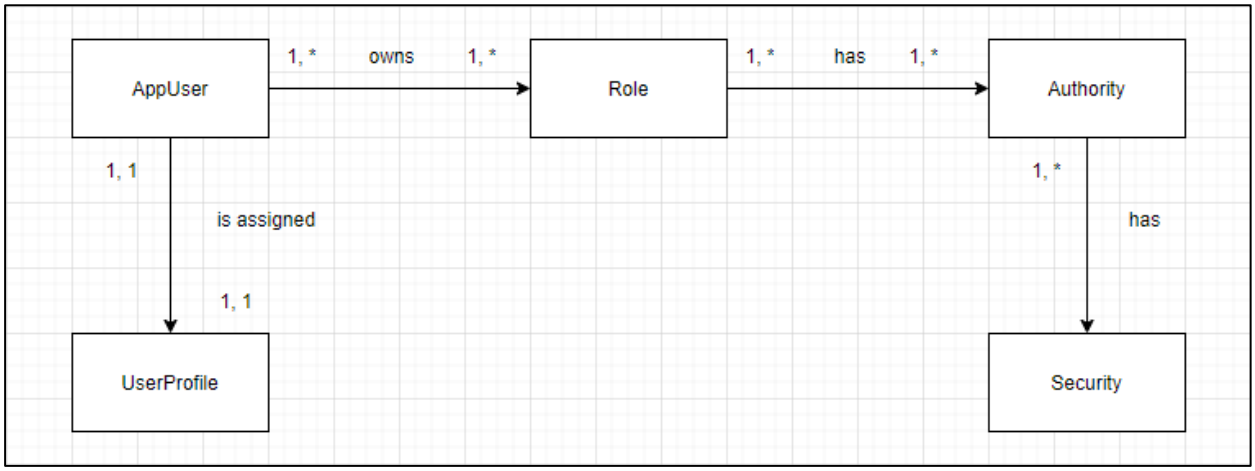
3 Authorization matrix

	Create	ReadOwn	ReadAll	UpdateOwn	UpdateAll	Delete
Apprentice						
Office worker						
Instructor						

The authorization matrix shows all the roles and their authorities in the table above. As you can see, on the horizontal column we have the different roles and on the vertical row the different authorities. The instructor has all the authorities which is why he can execute all the CRUD methods. The office worker, on the other hand, is not allowed to use the delete function. Otherwise, the office worker can do everything, like the instructor, except for deleting a user profile. The apprentice can only read his own data and edit them as well. All the other CRUD methods are forbidden for the apprentice.

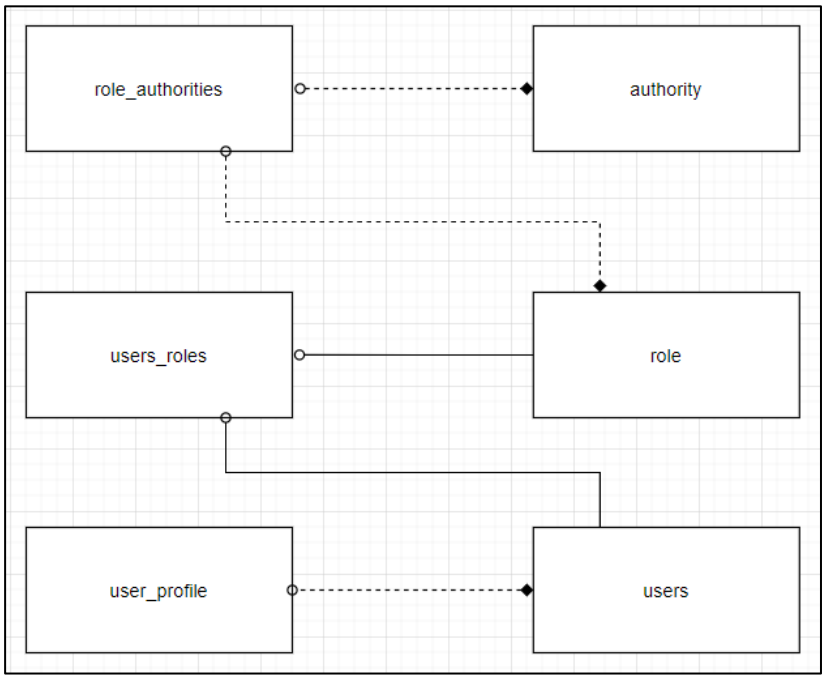
4 UML

4.1 Domain Model



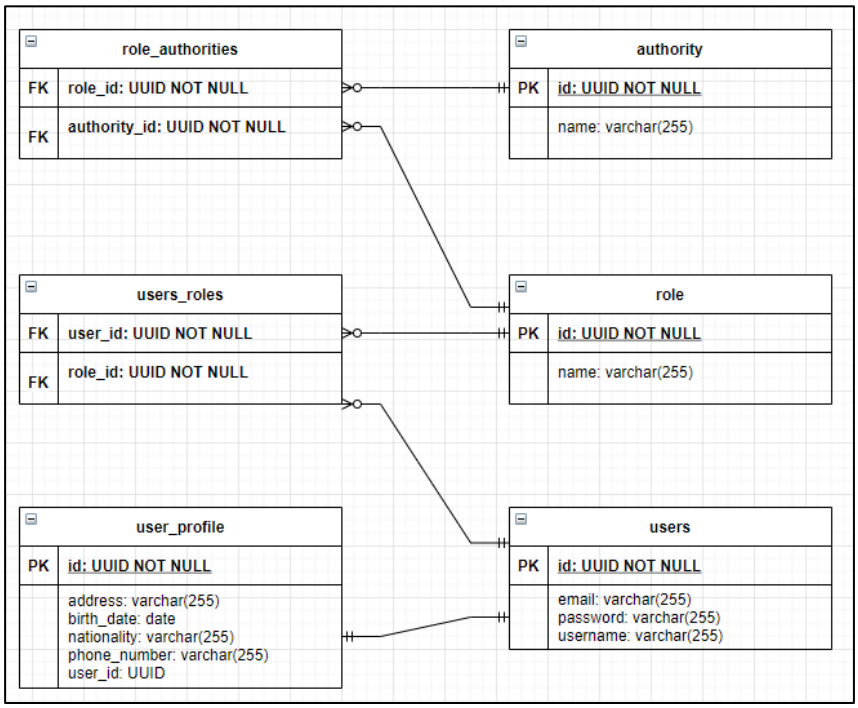
Our thoughts in the design of the domain model was first to list all the domains. In our program there are five domains. On the one hand there are userProfile, AppUser, Role, Authority and Security. The next step was to connect the individual domains. For example, AppUser has a connection to UserProfile. AppUser has a connection to Role, Role has a connection to Authoriy and Authority has a connection to Security. After the connections, we thought about the next step. We came to the conclusion that inserting the cardinalities is the next step. UserProfile is associated to exactly one AppUser. Conversely, exactly one AppUser is assigned to exactly one UserProfile. An AppUser owns one or more roles. In the other direction it looks the same: A role owns have one or more AppUser/s. A role has one or more authorities and an authority can have one or more roles. The Security domain is independent, but a Security can have one or more Authorities.

4.2 Entity-Relationship Model



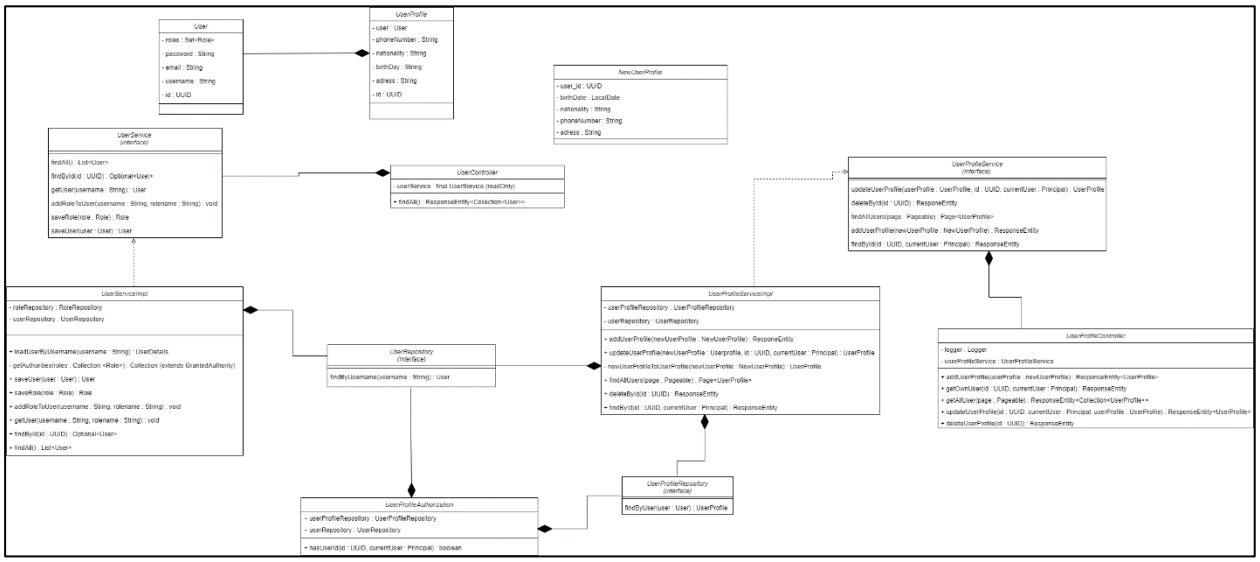
Here is our entity-relationship model (ERM). As you can see, we work with six tables. On the ERM you can see a rough view of how the tables are connected from our database. As you can see on the ERM, we have represented them with two different lines. The broken line means that it has been implemented by an interface.

4.3 Entity-Relationship-Diagram



Here is our entity-relationship-diagram (ERD). In the ERD you can see the more detailed connections to the tables (cardinality) and their attributes. As you can see, we choose a UUID as our key type. The reason behind that is so that we can keep our safety level high.

4.4 Class-Diagram

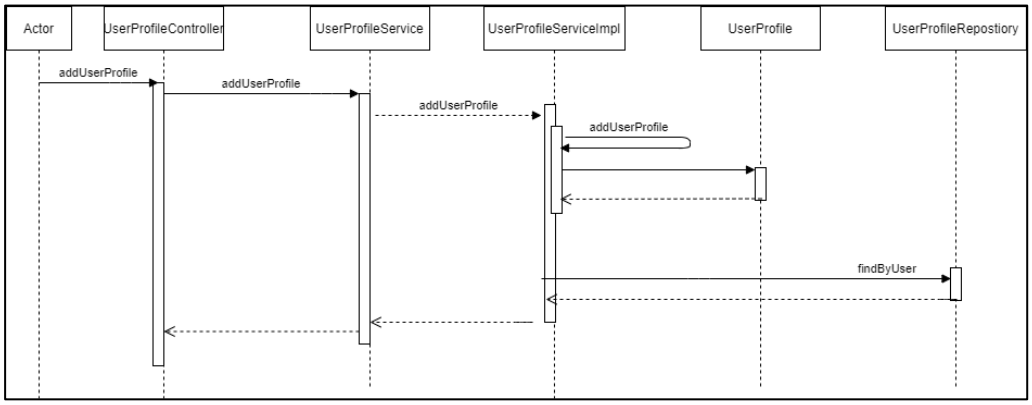


For the class diagram we have considered the following steps. The first step to create our class diagram was to create the individual classes in the diagram. Since we have to do only with the packages User and Roles, we have renounced to write down all classes and only the classes from the packages user and userProfile. There are the following classes which we have entered first: User, UserProfile, NewUserProfile, UserService, UserController, UserServiceImpl, UserProfile, UserProfileController, UserProfileService, UserProfileServiceImpl, UserProfileAuthorization and the UserRepository class. The User class has a relation to UserProfile. The User Class has a constructor with an instance of the Class UserProfile. If the class got a constructor with an instance of another class, the connection is called composition. In the diagram it's marked with a rhomb, which is filled in black. A composition between two objects associated with each other exists when there is a strong relationship between one class and another. The classes cannot exist without the owner or parent class. For example, A dog is a composition of kidney and diaphragm. For the class diagram we have considered the following. The first step to create our class diagram was to create the individual classes in the diagram. Since we have to do only with the packages User and Roles, we have renounced to write down all classes and only the classes from the package's user and userProfile. There are the following classes which we have entered first: User, UserService, UserController, UserRepository, UserService, UserServiceImpl, UserProfile, UserProfileController, UserProfileService, UserProfileServiceImpl and the UserProfileController class. The User and UserProfile classes have a connection with each other. The class UserProfileServiceImpl has a connection to the interface UserProfileService. The interface UserProfileService has a connection to the class UserProfileController, in which a constructor with the instance of the class UserProfileController was created and thus it is an aggregation. The UserServiceImpl class has a connection to the UserRepository interface and the UserService interface has a connection to the UserController class, which also has a constructor with the instance of the UserService interface and so it is an aggregation.

4.5 Sequence-Diagram

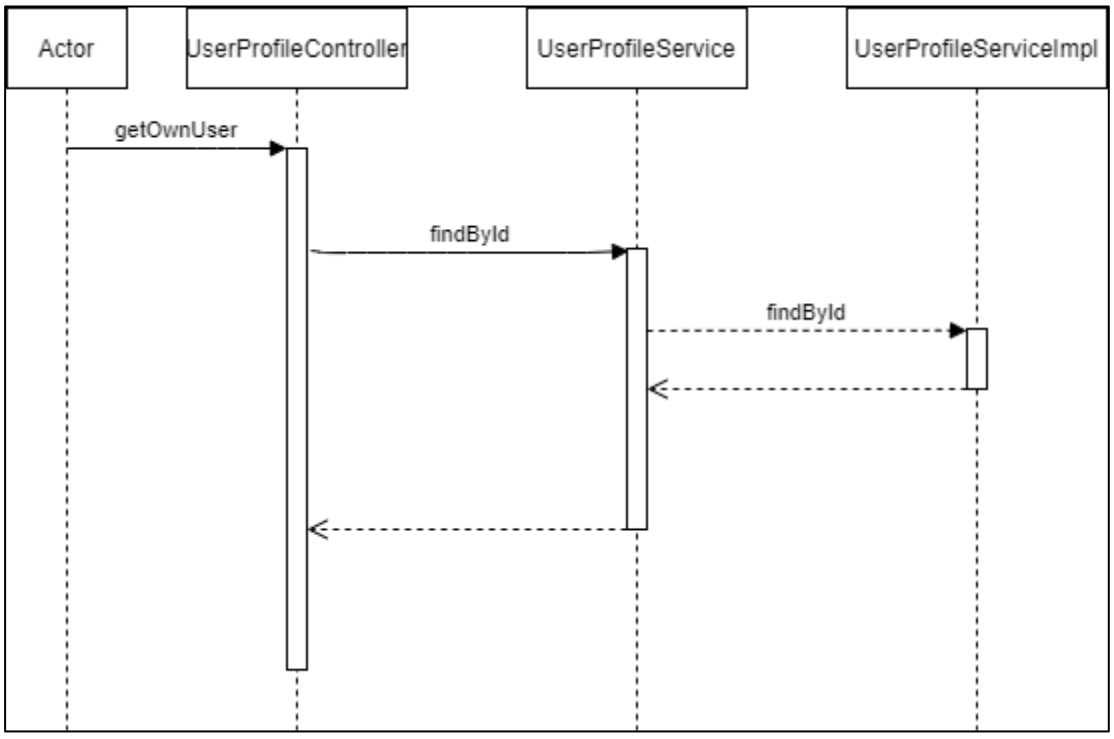
We decided to make a single Use Case for each endpoint because we think it is important to note the process for each endpoint and to record the whole thing in a Use Case.

4.5.1 Sequence Diagram addUserProfile



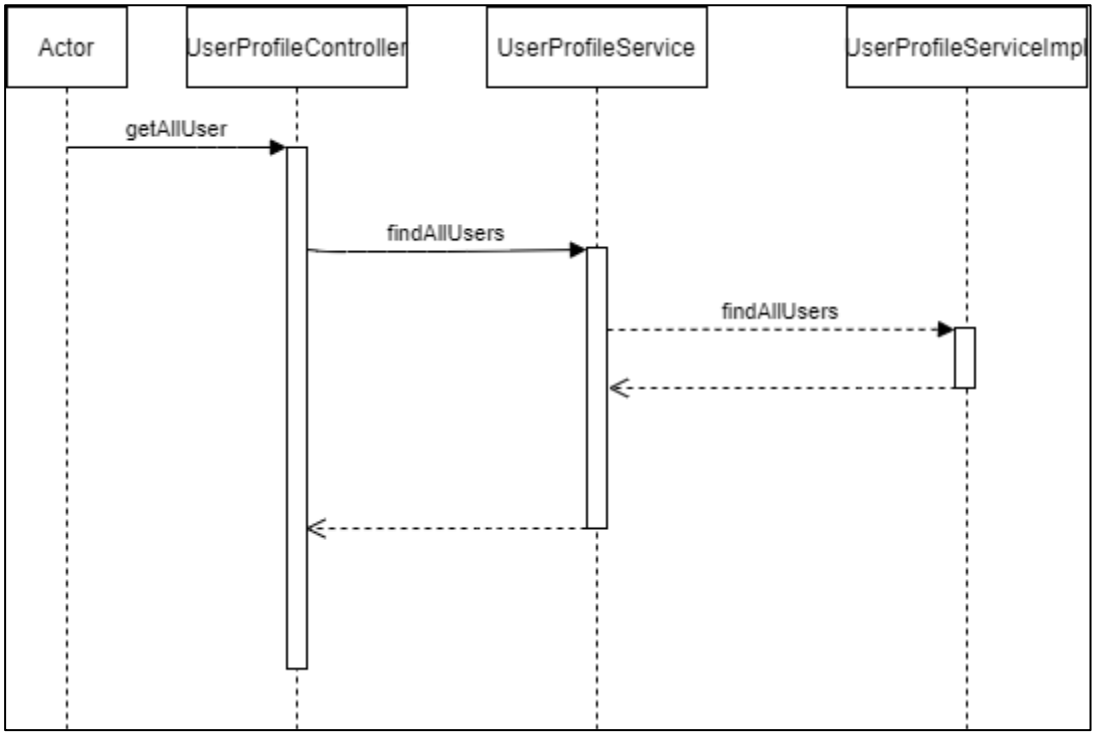
Our steps for the sequence diagram of addUserProfile were, first of all to write down the classes which are involved when you start the function. The classes are UserProfileController, UserProfileService, UserProfileServiceImpl, UserProfile and UserProfileRepository.

4.5.2 Sequence Diagram getOwnUser



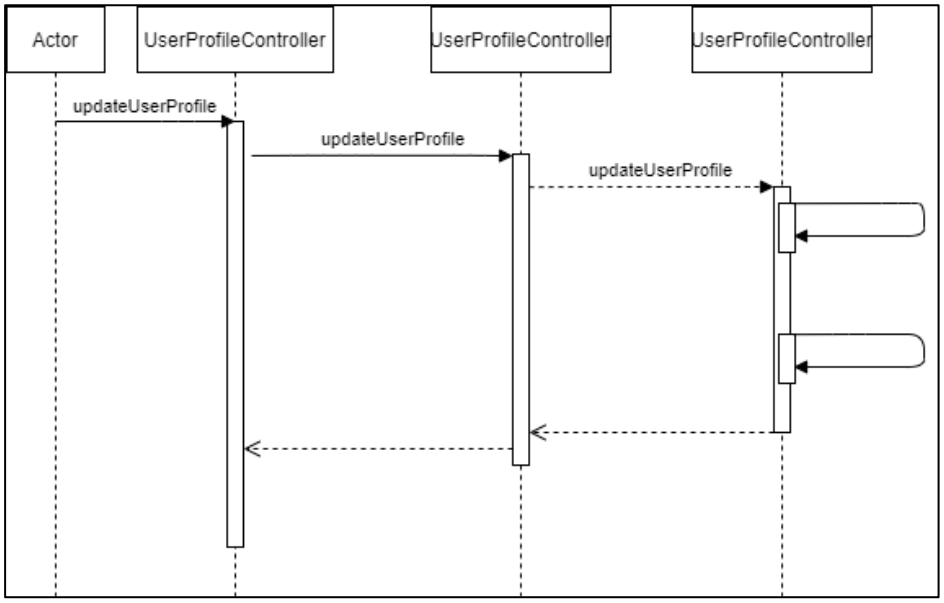
Special about this sequence diagram the function findById is involved in this Endpoint. At first it runs the endpoint getOwnUser, then the function searches for the id with the findById method. The function findById is required to check if there is an user profile to get data from the user profile and print it out.

4.5.3 Sequence Diagram getAllUser



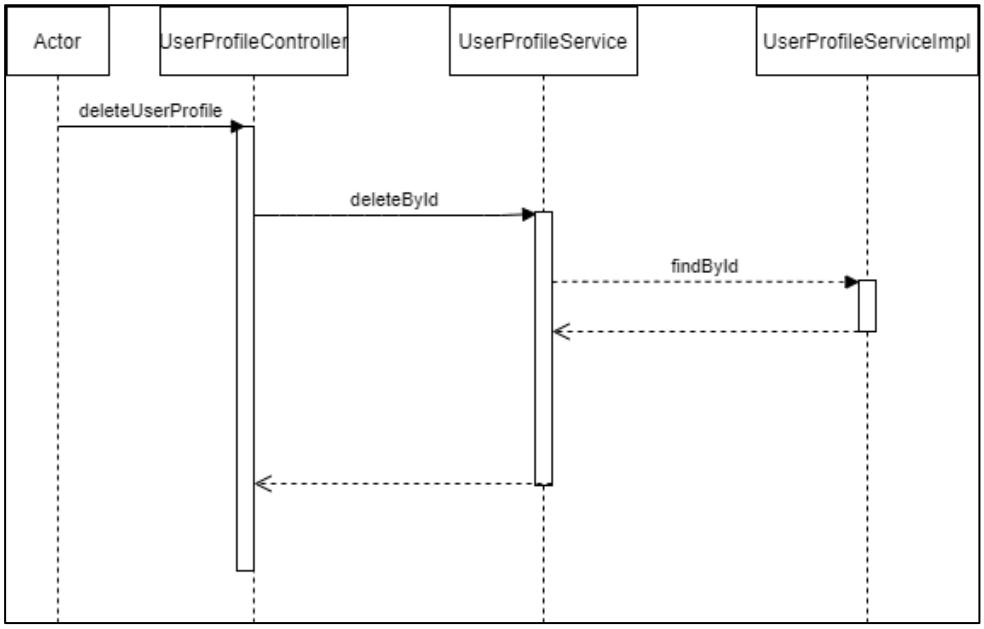
The endpoint getAllUser includes the findAllUsers method. It searches for all possible users and returns it to the function getAllUser. Then the endpoint getAllUser returns all users. The function findAllUsers is required to output every single user.

4.5.4 Sequence Diagram updateUserProfile



The special thing about the sequence diagram of the endpoint updateUserProfile is that the function updateUserProfile is self-referenced in the class UserProfileController. This is shown by the self-referenced arrows.

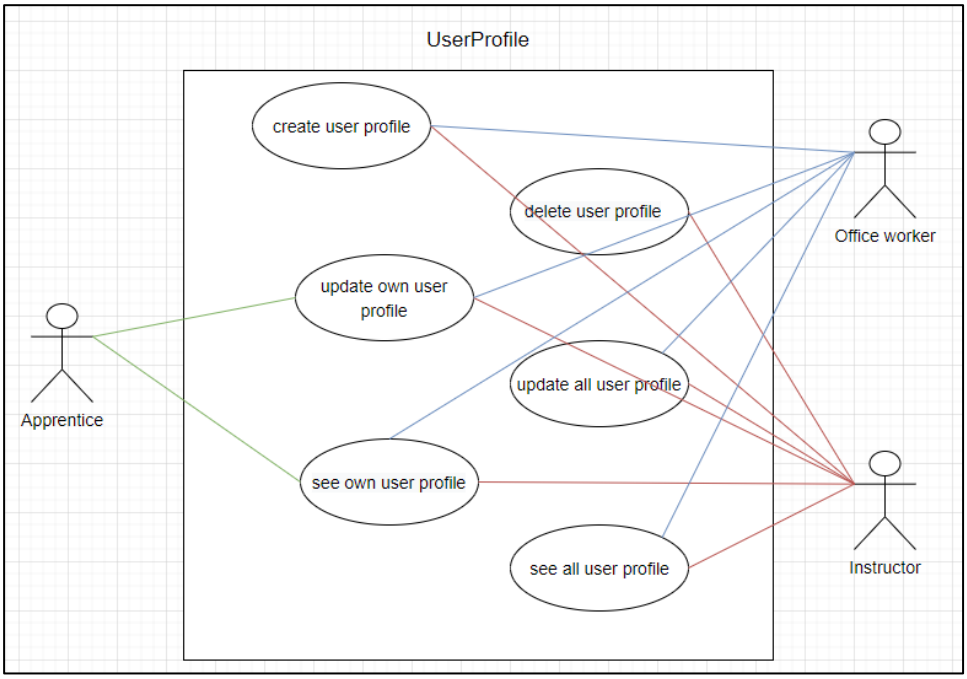
4.5.5 Sequence Diagram deleteUserProfile



The endpoint deleteUserProfile has two methods that are executed when you call the endpoint. The first method is deleteById. The function deleteById deletes something by the id the user gives. The second method is findById and the function searches for the user by the id. The function findById is required to search if there is a valid user profile to delete.

4.6 Use-Cases

Here you can see our basic use-case model. To make it clearer, the apprentice has a green connection, the office worker a blue one and the instructor a red one. With these connections you can see which functions a specific user can do. For this model we have been inspired by our Authorization matrix.



4.6.1 Successfull-Cases

Use Case deleteUserProfile()

Use Case:	Delete userprofile
Use Case ID:	1
Short description:	This use case deletes a user profile from the database.
Preconditions:	1. A valid user is logged in to the system. 2. The user has the correct authorization 3. The user enters a correct / an existing id
Actor (Primary):	The registered user
Actor (Secondary):	//Liste der Akteure, welche Informationen an einen Use Case übergeben oder Informationen von einem Use Case erhalten, diesen aber nicht anstossen.
Main process: 1. The UseCase starts when the user comes into the System. 2. The program checks if the entered id corresponds to an id for a userProfile in the database 3. If the id corresponds to an id for an userProfile in the database, the user will be deleted 4. If the id does not match the id of the userProfile in the database an error message is displayed	
Postconditions:	1. The userprofile has been deleted from the database 2. The id of the userprofile has been deleted 3. The program will display an error if the id is not equal to the id from the database
Alternative Flows: 1. If there is no userprofile in the database, the program outputs an error message 2. If the user gives the wrong id an error message (user not found) will be printed out	

Use Case addUserProfile()

Use Case:	Add userprofile
Use Case ID:	2
Short description:	This use case persisitates new userprofiles into the database.
Preconditions:	1. A valid user is logged in to the system. 2. The user has the Authority “CREATE” 3. A User is already created to add a userprofile to the user
Actor (Primary):	The logged in user
Actor (Secondary):	

Main process:

1. The UseCase starts when the user gets into the System.
2. The program checks if the there is a valid user to add the userprofile to the selected user
3. If there isn’t any user to add the userprofile the program sends an error message.
4. If there is a valid user, the program checks if there is already a userprofile for the user when adding the userprofile. If there is already a userprofile, the system prints out an error.
5. If both arguments are correct the userprofile will be created and a confirmation message will be printed.

Postconditions:	1. The user profile will be created 2. A new entry will be created in the database 3. If there is no user for the userprofile or the userprofiles already got created the program will print out an error message.

Alternative Flows:

1. If the user got the wrong authority, an error message (404: forbidden) will be displayed
2. If a userprofile for the user is already created

Use Case getOwnUser

Use Case:	Get own userprofile
Use Case ID:	3
Short description:	This endpoint gets the userprofile from the logged in user (own userprofile)
Preconditions:	1. A valid user is logged in to the system. 2. The user has the Authority “READ_ALL” 3. It’s a valid user which has a userprofile
Actor (Primary):	The user who is logged in in the system
Actor (Secondary):	
1. The Use Case starts when the user enters the system and writes in the right URL 2. This function checks if the user got the right authority to read the userprofiles 3. The function checks if there is a valid user 4. The method checks if there is a userprofile for the user which is logged in	
Postconditions:	1. The user got the userprofile from himself 2. The user knows that there is an userprofile of the user 3. The user can only read his own userprofile
Alternative Flows: 1. If there’s no userprofile an error message will be displayed 2. If the user got the wrong authority an error message (404: forbidden) will be outputted	

Use Case getAllUser()

Use Case:	Get all userprofiles
Use Case ID:	4
Short description:	This use case gets and prints out userprofiles
Preconditions:	1. A valid user is logged in to the system 2. The user got the “READ_ALL” authority 3. There are existing userprofiles
Actor (Primary):	The logged in user
Actor (Secondary):	

Main process:

1. The program checks if the user has the right authority to read all userprofiles
2. This endpoint outputs the userprofiles in a page and sorted
3. The user can decide how much userprofiles (pages) he wants to see / output and if the entries of the userprofiles should be sorted or not. The user can write it in the URL.
4. The program outputs the userprofiles

Postconditions:	1. The userprofiles were ouputted 2. If everything has worked out the user got the authority “READ_ALL” 3. There are existing userprofiles to read

Alternative Flows:

1. If there are no userprofiles the method will display an error message
2. On condition that the user got the wrong authority an error message (404 : forbidden) will be displayed

Use Case updateUserProfile()

Use Case:	Update UserProfile
Use Case ID:	5
Short description:	This use case can update a userprofile
Preconditions:	1. A valid user is logged in to the system. 2. The user has the right Authority (“UPDATE_ALL”) 3. The user enters the correct id for the user to edit the userprofile
Actor (Primary):	The current user is the primary actor
Actor (Secondary):	
Main process: 1. This use case starts when the user logs into the system 2. The program updates the userprofile from the user 3. The program has the variables address, birth date, nationality, and phone number 4. If there is no userprofile to update the system will create a new userprofile	
Postconditions:	1. If the id from the user was correct, the user should be able to edit the userprofile 2. If there was an existing userprofile, the userprofile got updated 3. The userprofile got created, if there wasn’t a userprofile for the user
Alternative Flows: 1.	

4.6.2 Failed-Cases

Use Case updateUserProfile()

Use Case:	Update UserProfile
Use Case ID:	6
Short description:	This use case can update a userprofile
Preconditions:	1. A valid user is logged in to the system. 2. The user has the right Authority (“UPDATE_ALL”) 3. The user enters the correct id for the user to edit the userprofile
Actor (Primary):	The current user is the primary actor
Actor (Secondary):	
Main process: 1. The user logs in with an account which has the Role(“APPRENTICE”) 2. The function recognizes it and displays the 404 forbidden HTTP message 3. The User cannot update the userprofiles	
Postconditions:	1. The user got the wrong authority to update any userprofile 2. The user can only read all userprofiles but cannot edit them
Alternative Flows: 1. If the user had registered with the account of the right authority, he could have updated it	

5 Endpoints

5.1 Get All UserProfiles

5.1.1 Controller

```
@GetMapping("/")
    @PreAuthorize("hasAuthority('READ_ALL')")
    public ResponseEntity<Collection<UserProfile>> getAllUser(Pageable page) {
        logger.trace("GET ALL USERPROFILES ENDPOINT ACCESSED");
        return new ResponseEntity(userProfileService.findAllUsers(page),
            HttpStatus.OK);
    }
```

The first Endpoint from the Controller-Layer to the Frontend is the getAllUsers Endpoint. At first, we specified the path in the @GETMapping.Annotation. This line also determines the kind of operation, which we will access trough. For the next part we added the annotation “PreAuthorize”. This line is necessary to prevent unauthorized users to get all users. The reason why we are using the authority “READ_ALL” is because only two of three roles are authorized to see, read, all the user profiles. Next, we are adding “Pageable” as the parameter so we can decide how many user profiles should be shown, and how (sort). Now that we are in the Endpoint, we can tell the Log and trace the transaction and return the method “findAllUsers” from the “Service-Layer” as a Response Entity.

5.1.2 ServiceImpl

```
@Override
public Page<UserProfile> findAllUsers(Pageable page) {
    return userProfileRepository.findAll(page);
}
```

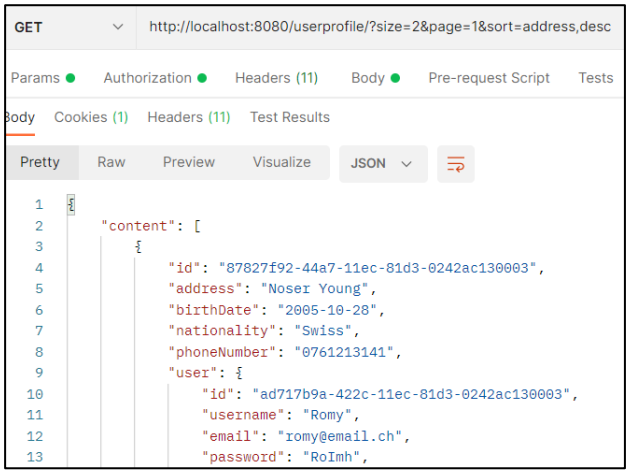
This method gets the methods form our “user profile repository” and forwards it to our user profiles controller.

5.1.3 Postman Result

GET http://localhost:8080/userprofile/?size=2&page=1&sort=address,asc


```
{
    "content": [
        {
            "id": "5b9b7994-4306-11ec-81d3-0242ac130003",
            "address": "Noser",
            "birthDate": "1999-12-12",
            "nationality": "GERMAN",
            "phoneNumber": "0761213141",
            "user": {
                "id": "2aa70872-421a-11ec-81d3-0242ac130003",
                "username": "Luca",
                "email": "instructor@noseryoung.ch",
                "password": "LuWid",
```

This is our postman request. Since we are returning a page, we, the postman-user, can decide how many profiles per page postman should show. The other benefit of using pageable is that is can also sort the user profiles by any given field. Like shown above with the address of the user profiles. Now if we change the sort to ascending instead of descending then the output looks like that:



Here is a list of sorting algorithms which you can use to:

Description	Parameter
If you only want to print three user profiles, without any sorting/filter, then this is the way to go:	?size=3
If your demand is to have multiple pages, starting with one of user profiles with only two user profiles per page, then your path should contain this: **Attention: A page begins at the index 0**	?size=2&page=1
At last, you want to sort the pages and user profiles by their address,ascending, then add this:	?size=3&page=1&sort=address,asc

5.2 Get UserProfile By ID

5.2.1 Controller

```
@GetMapping("/{id}")
    @PreAuthorize("@userProfileSecurity.hasUserId(#id, #currentUser) || hasAuthority
        ('READ_ALL')")
    public ResponseEntity getOwnUser(@PathVariable UUID id, Principal currentUser) {
        logger.trace("GET ONE USERPROFILE ENDPOINT ACCESSED");
        try{
        return ResponseEntity.ok().body(this.userProfileService.findById(id, currentUser));
    } catch(NullPointerException e){
            return ResponseEntity.status(404).body(e.getMessage());
        }
    }
```

Starting with this Endpoint. We first changed the path to "/{id}". Therefore, the two Get-Request won't crosswise each other, but the more important task of that is so we can access a certain profile by the ID of it. Now for the authorities. We decided on letting the user see his own user profile along the higher roles in the matrix. To see how the component works see the section below. In the Endpoint itself we are tracing again and return the body of the response, and also the method from the service layer.

5.2.2 Config

```
public boolean hasUserId(UUID id, Principal currentUser){
    UUID idCurrent = userProfileRepository.findByUser(userRepository
        .findByUsername(currentUser.getName())).getId();
    if(idCurrent.equals(id))
        return true;
    return false;
}
```

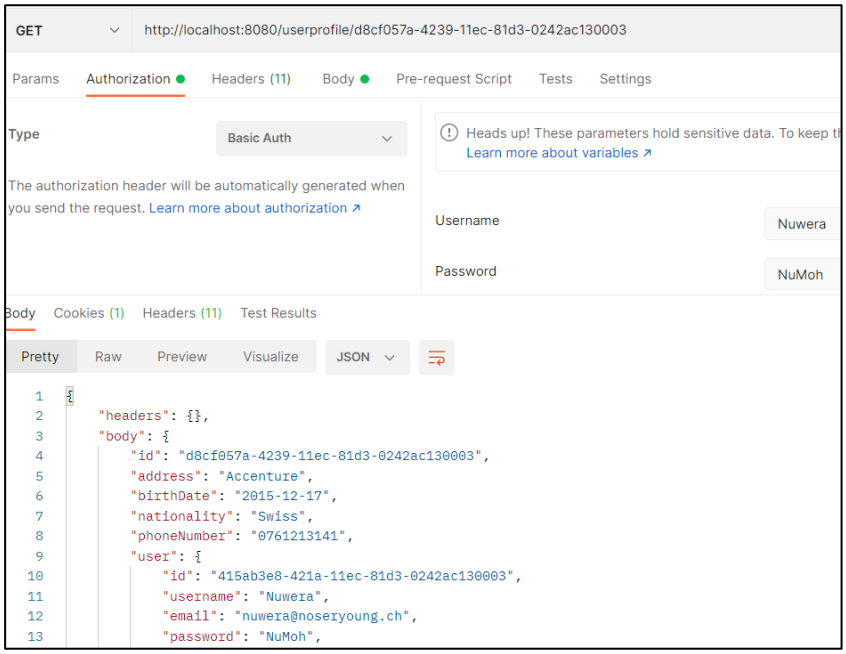
This file is in the security domain. As you can tell we are expecting the Id of the user profile which the user is working with and the principal (current user who is logged in). The following code checks if the user who is logged has the same id as the foreign key of the user profile which is being edited. If that is the case, then the parameters are forwarded to the controller Endpoint and hence the output is true the user is allowed to get his own profile.

5.2.3 ServiceImpl

```
@Override
    @Transactional(rollbackFor = NullPointerException.class)
    public UserProfile findById(UUID id, Principal currentUser) throws NullPointerException
        {
        Optional<UserProfile> optionalUserProfile = this.userProfileRepository.findById(id
            );
        if (optionalUserProfile.isPresent()) {
            return optionalUserProfile.get();
        } else {
            throw new NullPointerException("USERPROFILE NOT FOUND");
        }
    }
```

This method which is the intermediate step of the two other layers has more that only that job. As you may have already seen in the upper section this method throws an Exception if the user profile is empty. But in the successful case the user profile will be returned.

5.2.4 Postman Result



Now if the user accesses the endpoint with their own id, then the person will be able to see their profile.

5.3 Post UserProfile

5.3.1 Controller

```
@PostMapping("/")
    @PreAuthorize("hasAuthority('CREATE')")
    public ResponseEntity addUserProfile(@RequestBody NewUserProfile userProfile) {
        logger.trace("POST USERPROFILES ENDPOINT ACCESSED");
        try {
            return ResponseEntity.ok().body(userProfileService.addUserProfile(userProfile
                ));
        } catch(NullPointerException e){
            return ResponseEntity.status(404).body(e.getMessage());
        } catch (InstanceAlreadyExistsException e){
            return ResponseEntity.status(409).body(e.getMessage());
        }
    }
```

When accessing the endpoint of POST, we need the authority of the admin, create. We use the annotation @RequestBody to enable automatic deserialization. Like in every of our endpoint we trace the operation. The return value is the Response Entity with, in the best case, the addUserProfile method.

5.3.2 ServiceImpl

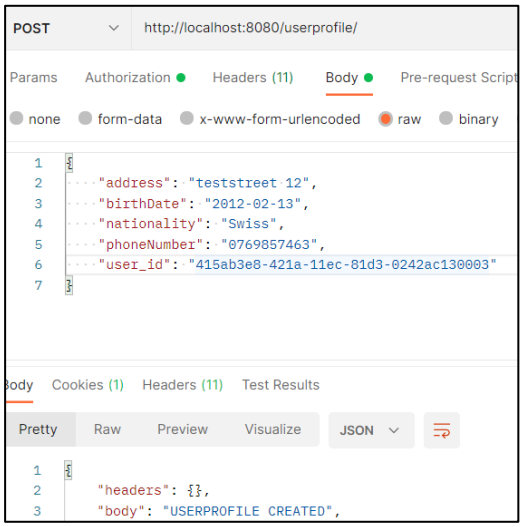
```
private UserProfile newUserProfileToUserProfile(NewUserProfile newUserProfile) {
        UserProfile userProfile = new UserProfile();
        userProfile.setUser(userRepository.findById(newUserProfile.getUser_id()).orElse
            (null));
        userProfile.setNationality(newUserProfile.getNationality());
        userProfile.setAddress(newUserProfile.getAddress());
        userProfile.setPhoneNumber(newUserProfile.getPhoneNumber());
        userProfile.setBirthDate(newUserProfile.getBirthDate());

        return userProfile;
    }
```

```
@Override
    public UserProfile addUserProfile(NewUserProfile newUserProfile) throws
        InstanceAlreadyExistsException, NullPointerException {

        UserProfile userProfile = newUserProfileToUserProfile(newUserProfile);

        if (userProfile.getUser() == null) {
            throw new NullPointerException("USER NOT FOUND");
        } else if (userProfileRepository.findByUser(userProfile.getUser()) != null) {
            throw new InstanceAlreadyExistsException("USER ALREADY HAS USERPROFILE");
        } else {
            return userProfileRepository.save(userProfile);
        }
    }
```

In our project we have the whole user as an attribute in our UserProfile file. This causes that postman expects a whole user in the body which is not very user friendly. To overcome this problem, we get the user with their ID and set it already. Next in the “addUserProfile” method we call the upper listed method and set it in here. Next, we have the usual exception handling and the return of the save method with the new user profile.

5.3.3 Postman Result

This here is the output:



As you can see, we are using the “POST” operation and sending the data as json forward. After completing the transaction, we get the response that the profile was created.

5.4 Put UserProfile

5.4.1 Controller

```
@PutMapping("/{id}")
    @PreAuthorize("@userProfileSecurity.hasUserId(#id, #currentUser) || hasAuthority
        ('UPDATE_ALL')")
    public ResponseEntity updateUserProfile(@PathVariable UUID id, Principal currentUser,
        @RequestBody UserProfile userProfile){
        logger.trace("PUT USERPROFILE ENDPOINT ACCESSED");
        try {
            return ResponseEntity.ok().body(userProfileService.updateUserProfile
                (userProfile, id, currentUser));
        } catch(NullPointerException e){
            return ResponseEntity.status(404).body(e.getMessage());
        }
    }
```

On top of this method, we are going to use our component again (authorize). Next, we are going to use the annotation @RequestBody. This is necessary so we can give the id with the path. The last part is returning the actual "**newUserProfile**" → updated user profile, in the best case, with the appropriate HTTP. Status code.

5.4.2 ServiceImpl

```
@Override
    public UserProfile updateUserProfile(UserProfile newUserProfile, UUID id, Principal
        currentUser) throws NullPointerException {
        Optional<UserProfile> optionalUserProfile = this.userProfileRepository.findById(id);

        if(optionalUserProfile.isEmpty()) {
            throw new NullPointerException("USERPROFILE DOESN'T EXIST");
        } else if(!userProfileRepository.existsById(id)) {
            throw new NullPointerException("USER DOESN'T EXIST");

        }else
        {
            return userProfileRepository.findById(id)
                    .map(updatedUserProfile -> {
                        updatedUserProfile.setAddress(newUserProfile.getAddress());
                        updatedUserProfile.setNationality(newUserProfile.getNationality());
                        updatedUserProfile.setBirthDate(newUserProfile.getBirthDate());
                        updatedUserProfile.setPhoneNumber(newUserProfile.getPhoneNumber());
                        return userProfileRepository.save(updatedUserProfile);
                    }).orElseGet(() -> {
                        return userProfileRepository.save(newUserProfile);
                    });
        }
    }
```

Here is the Implementation of the method which is in the file "UserProfileServiceImpl". First, we check if the user profile is empty, depending in that we throw an exception. On condition that the profile exists then the new data should be set and saved.

5.4.3 Postman Result

5.5 Delete UserProfile

5.5.1 Controller

```
@DeleteMapping("/{id}")
    @PreAuthorize("hasAuthority('DELETE')")
    public ResponseEntity deleteUserProfile(@PathVariable("id") UUID id) {
        logger.trace("DELETE USERPROFILE ENDPOINT ACCESSED");
        try {
            userProfileService.deleteById(id);
            return ResponseEntity.ok().body("USERPROFILE DELETED");

        } catch (NullPointerException e) {
            return ResponseEntity.status(404).body(e.getMessage());
        }
    }
```

First, like in front of every endpoint, the authority is checked. Depending on that the user can operate with this method or not. After tracing the entrance of the endpoint, the user profile is being deleted or the suitable HTTP status is returned.

5.5.2 ServiceImpl

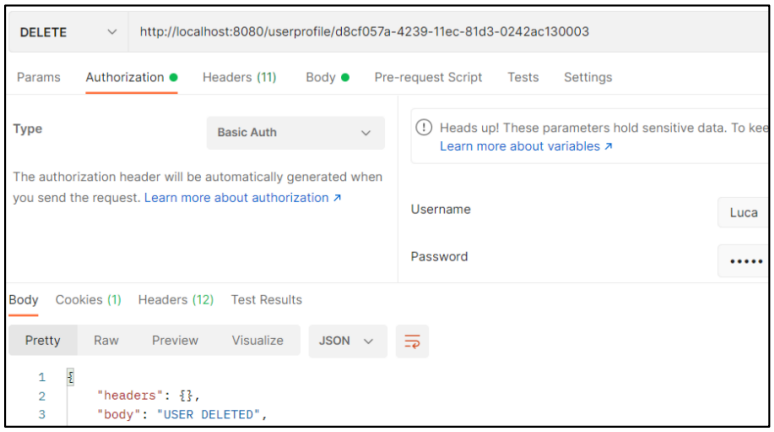
```
@Override
    @Transactional(rollbackFor = NullPointerException.class)
    public UserProfile deleteById(UUID id) throws NullPointerException {
        Optional<UserProfile> optionalUserProfile = this.userProfileRepository.findById(id);
        if (optionalUserProfile.isEmpty()) {
            throw new NullPointerException("USERPROFILE DOESN'T EXIST");
        }else {
            userProfileRepository.deleteById(id);
        }

        return null;
    }
```

Now in the service side of the backend the user profile is being searched first → with findById() and depending on the income an exception will be thrown or the method of the JPARepository will be called.

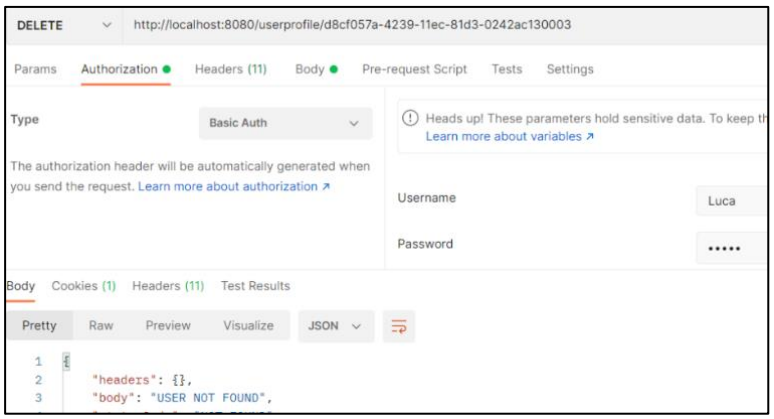
5.5.3 Postman Result

We created this endpoint to delete a user profile using the id. The method is accessed with localhost/8000/delete/{id}. Depending on the id you delete a different user. If this id does not exist, the function returns "USER NOT FOUND". We solved the whole thing with an if else statement. If the user profile of the user with the given id is empty it should return "USER NOT FOUND". If the user exists, the program should return "USER DELETED" and execute the function.

This is how it looks like in postman if you enter a correct id to delete the user:



If the id is not existing, this is what the output looks like:



6 Testing

6.1 ACID-Fulfillment

6.1.1 Atomicity

For this point we worked with Exceptions to prevent inconsistency, as you can see in the image below we **catch** the exceptions return it, so the transaction has a safe-state at the end:

```
@PostMapping("/")
    @PreAuthorize("hasAuthority('CREATE')")
    public ResponseEntity addUserProfile(@RequestBody NewUserProfile userProfile) {
        logger.trace("POST USERPROFILES ENDPOINT ACCESSED");
        try {
            return ResponseEntity.ok().body(userProfileService.addUserProfile(userProfile
                ));
        } catch(NullPointerException e){
            return ResponseEntity.status(404).body(e.getMessage());
        } catch (InstanceAlreadyExistsException e){
            return ResponseEntity.status(409).body(e.getMessage());
        }
    }
```

6.1.2 Consistency

We avoided this trap by only having one endpoint/method for one request. We tried to seperate the methods, so it is not possible to access the wrong operation.

6.1.3 Isolation

To check if the transaction is invisible to other transactions, we used the annotation «Transactional». In the parameter of the annotation, we work with a rollback (to the point where the exception was thrown):

```
@Override
    @Transactional(rollbackFor = NullPointerException.class)
    public ResponseEntity findById(UUID id, Principal currentUser) throws
        NullPointerException {
        Optional<UserProfile> optionalUserProfile = this.userProfileRepository.findById(id
            );
        if (optionalUserProfile.isPresent()) {
            return ResponseEntity.ok(optionalUserProfile.get());
        } else {
            return ResponseEntity.status(404).body("USER NOT FOUND");
        }
    }
```

6.1.4 Durability

We insured the durabillity by adding a logger trace in the endpoints:

```
@GetMapping("/")
    @PreAuthorize("hasAuthority('READ_ALL')")
    public ResponseEntity<Collection<UserProfile>> getAllUser(Pageable page) {
        logger.trace("GET ALL USERPROFILES ENDPOINT ACCESSED");
        return new ResponseEntity(userProfileService.findAllUsers(page), HttpStatus.OK);
    }
```

To prevent that the laptop suddenly shut down, then the logger is going to trace the acccess of the endpoint.

6.2 Component-Test (Postman)

Component tests, also called program or module tests, is done after the unit test. This type of testing allows the test object to be tested independently as a component without being integrated with other components (such as modules, classes, objects, and programs). These tests are performed by the development team. By us and are in the repository to be seen.

6.3 JUnit Test

```
@Test
    void addUserProfile() {

        int sizeOfUserProfile1 = userProfileRepository.findAll().size();
        try{
            userProfileService.addUserProfile(new NewUserProfile("teststr 32", LocalDate
                .parse("2012-09-09"), "CH", "0789876543", UUID.fromString("415ab3e8-421a
                -11ec-81d3-0242ac130003")));
        } catch (InstanceAlreadyExistsException e){
        }
        int sizeOfUserProfile2 = userProfileRepository.findAll().size();
        assertEquals(sizeOfUserProfile1 + 1, sizeOfUserProfile2);
    }
```

This test is checking if the post-method is working. First, we are saving the size of the data into a variable. Next, we're creating a user profile. Afterwards we are saving the size of the data into another variable. That means the first variable with the size of the data before adding a user profile + one should be the same size as the second variable.

Test result: passed

```
@Test
    void deleteById() {
        try{
            userProfileService.addUserProfile(new NewUserProfile("teststr 32", LocalDate
                .parse("2012-09-09"), "CH", "0789876543", UUID.fromString("415ab3e8-421a
                -11ec-81d3-0242ac130003")));
        } catch (InstanceAlreadyExistsException e){
        }
        userProfileRepository.deleteAll();
        assertEquals(true, userProfileRepository.findAll().size() == 0);
    }
```

This test is checking if the delete-method is working. First, we are creating a new user profile. After that we're deleting all data. Last, we are checking if the size of our data is 0, which is true.

Test result: passed

7 Problems

We as a group has the problem that the task looked quite big. Almost every input that we had was related to our group task. This made us start with the code too fast. But after we realized that our documentation also has a lot to say about our code, we took a step back and created all the issues and now had a better overview on our task. Everything looked more doable now. We split the task but that still caused us merge conflicts, which we as a group could handle pretty easily.

8 Conclusion

We all worked together to finish the project. We managed to implement every method/endpoint and that in a good manner (of time). We changed some of our methods a couple of times to improve the code quality and to avoid redundant code. We wrote more code than originally expected and documented better than we thought. Our endpoints are all complete and additionally as well some extra fulfillments. The documentation also is way more detailed than we thought we could do (in the given time frame). Everyone in our group did their best and attributed to the project. We all helped each other to solve errors, as well as generally when running into problems.