# Blog-site

User Manager

Alisha Khalid, Janna Esteban, Jannik Böppli

# Content of Table

# Introduction

This documentation is all about our Spring Boot project in which we will do a user management. In the following pages we will document the planning up to the implementation. To better understand our way of thinking and why we have implemented the code this way, some code snippets are shown and described. We will also show you our way of working together for this we have decided to define commits to get a clear view of what changed in the code. We decided to work with git issues, so that we can assign tasks to each other. To get a better overview of the project, we created a Domain Model, Class Diagram, Sequence Diagram and use cases, so that the user has a better knowledge about the usage of the application. For a better orientation of our endpoints, we included to our source code Swagger, you can get the link in the readme file.

The goal of this project was to use our newly learned knowledge and be able to apply it practically

# Planning

## Gantt

| | KW44 | KW45 | KW46 |
|---|---|---|---|
| Planning | | | |
| Create Git issues | | | |
| Do Readme File | | | |
| Create Domain Model | | | |
| Code | | | |
| Work on endpoints | | | |
| Do Javadoc | | | |
| Add logger | | | |
| Documentation | | | |
| Create Use cases | | | |
| Create Domain Model | | | |
| Create ERM | | | |
| Create Class Diagram | | | |
| Create sequence diagram | | | |
| Describe Endpoints | | | |
| Do Component Tests | | | |

# Conventions

## Commit

For each commit we defined a shortcut, and you must write down what you did or changed after the shortcut. We summarized everything in the table below. With the commit you can tell what you did and in the following layer by the shortcut.

| Commit message | Description |
|---|---|
| <SL-WHAT> | SL stands for the service layer |
| <RL-WHAT> | RL stands for the repository layer |
| <WL-WHAT> | WL stands for the web layer / Controller |
| <ALL-WHAT> | ALL stands for the three layers (SL, RL and WL) |
| <MODEL> | MODEL stands for the object |
| <LAYER -IMP> | IMP stands for the improvements, and for quick changes |
| <LAYER-BC> | BC stands for breaking change, when you made a major change |
| <LAYER-JD> | JD stands for JavaDoc |
| <SC-WHAT> | SC stands for security |
| <DB> | DB stands for database |
| <LAYER-FEAT> | FEAT stands for features, when you add a new functionality. |
| <LAYER-TEST> | TEST stands for all tests and the layer in front is the layer which was tested |
| <MC> | MC stands for merge conflict |
| <CO> Message of latest change | CO stands for checkout, and this is our last commit message of the day. You also write, what you did in the code. |

## Branch

| Branch name | Description |
|---|---|

| | |
|---|---|
| feature/WHAT | When you add a new feature. At the ending (WHAT) you describe in small words for what the branch is in camel casing. For example, *feature/deleteUserEndpoint* |
| bugFix/WHAT | When you fix a bug. At the ending (WHAT) you describe in small words for what the branch is in camel casing. For example, *bugFix/deleteUserEndpoint* |
| improvement/WHAT | When you have to expand a code source or to improve the code. |
| documentation/WHAT | This is mainly for Javadoc purpose |

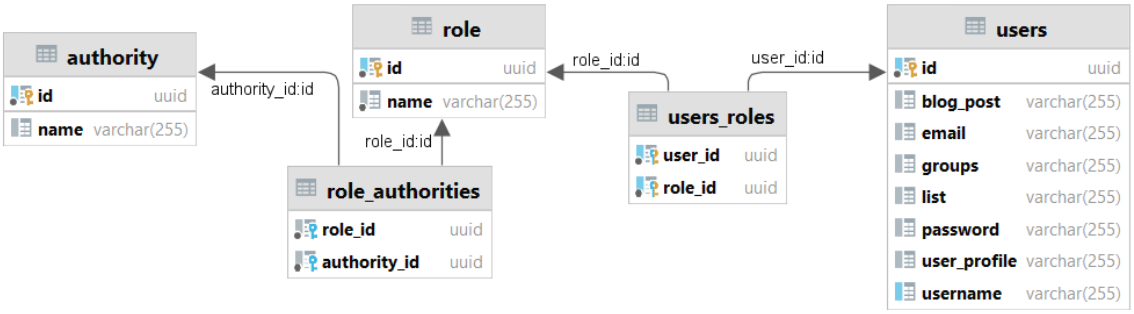
## Naming

| Essentials in code | Description |
|---|---|
| methods | Camel casing |
| attributes | Camel casing |
| class | Camel casing |

# Unified Modeling Language (UML)

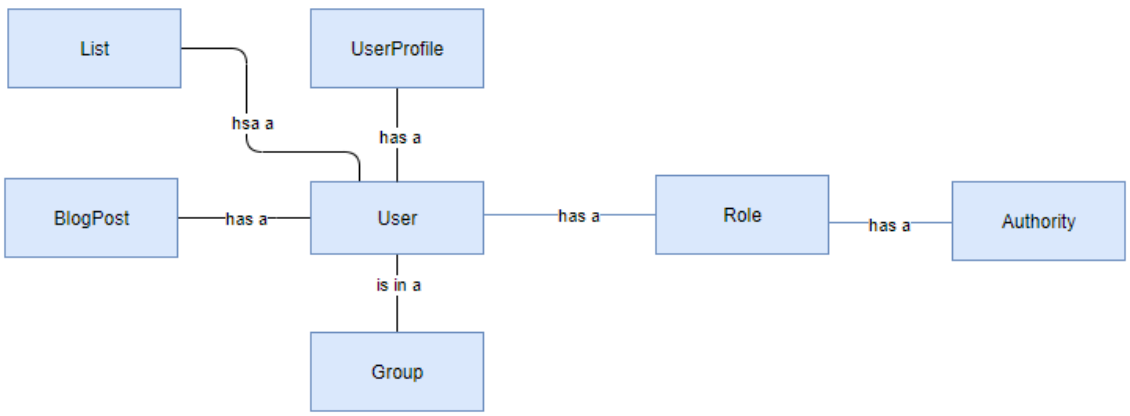
## Entity-Relationship-Modell (ERM)

## Entity-Relation-Diagram (ERD)




## Domain Model

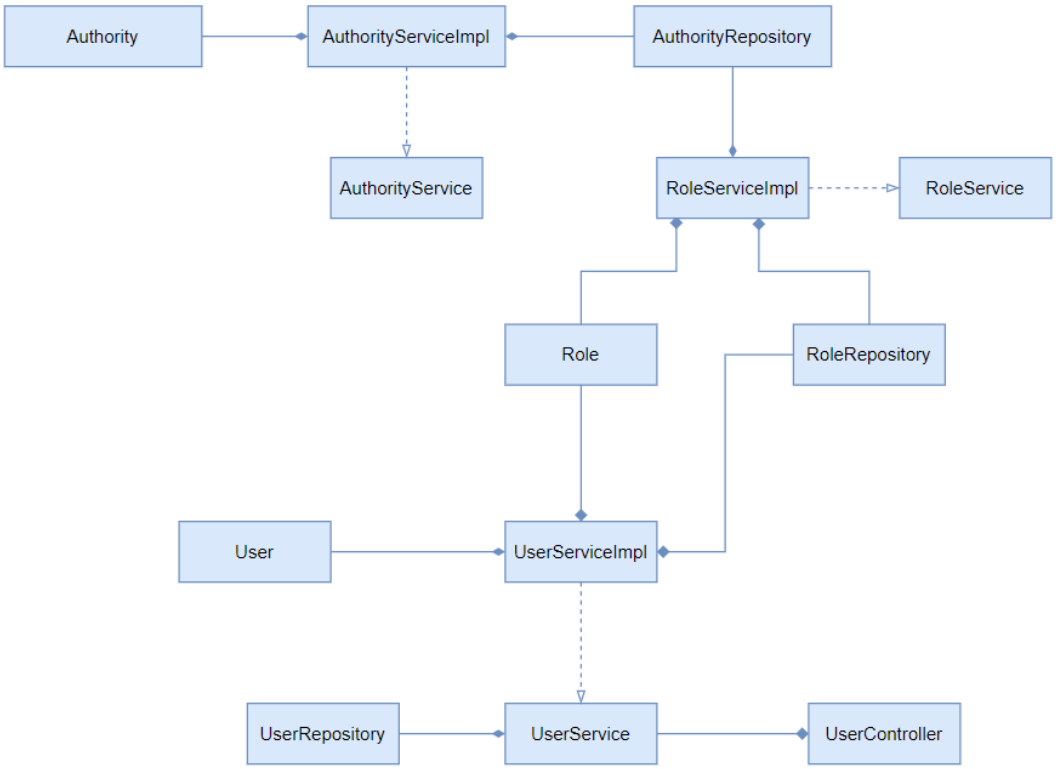
### ROUGH DESIGN



This is our domain model which gives a brief overview of the whole project. We have defined the domains / also known as "entities" for which we figured out what relationship they have to each other.

## *Class Diagram*

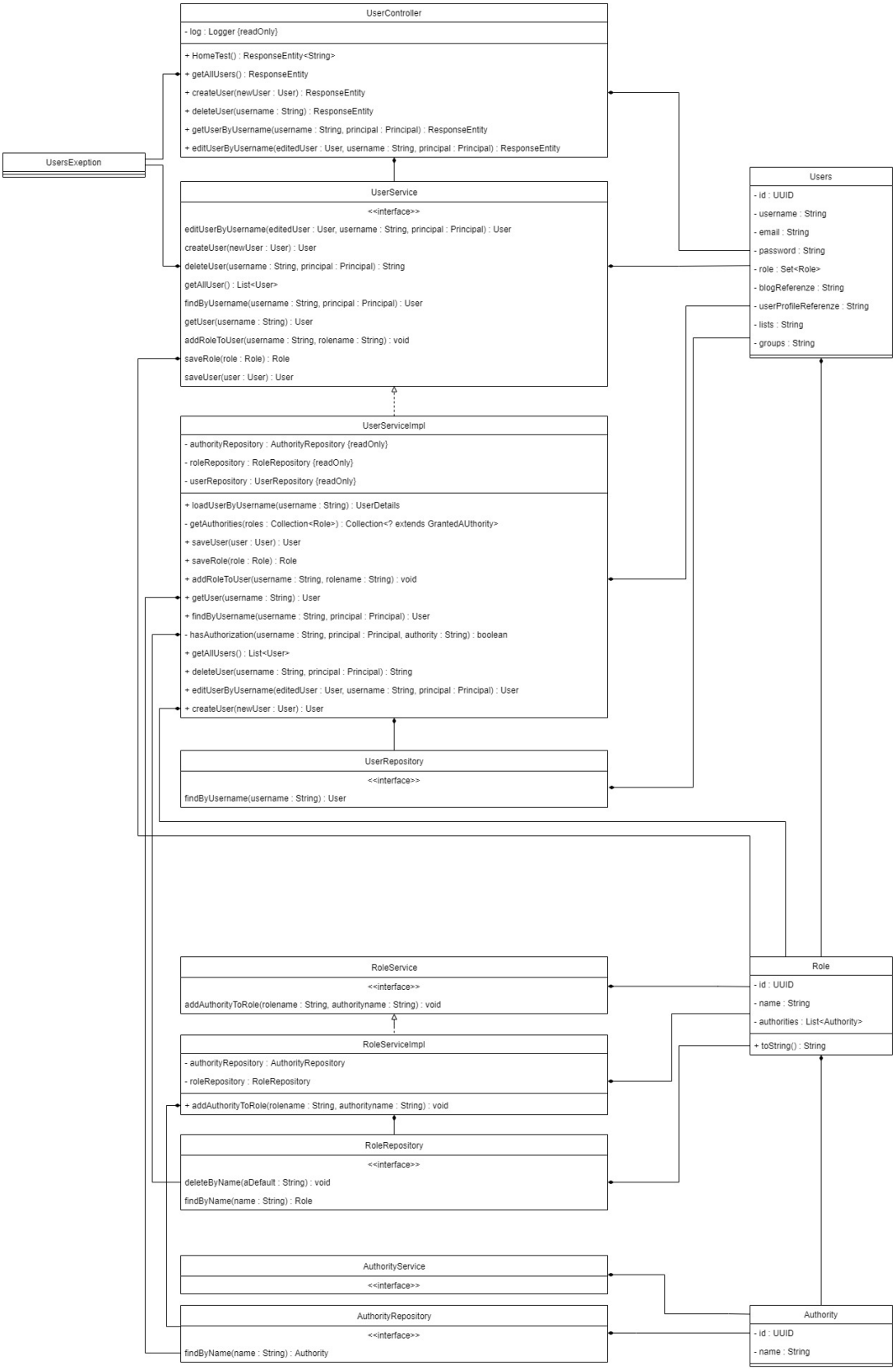
### ROUGH DESIGN




In comparison to the domain model, the relationships between the entities can be seen here. The empty diamond stands for aggregation, the full diamond for composition and the dotted lines for implementation because the UserServiceImpl. implements from UserService.

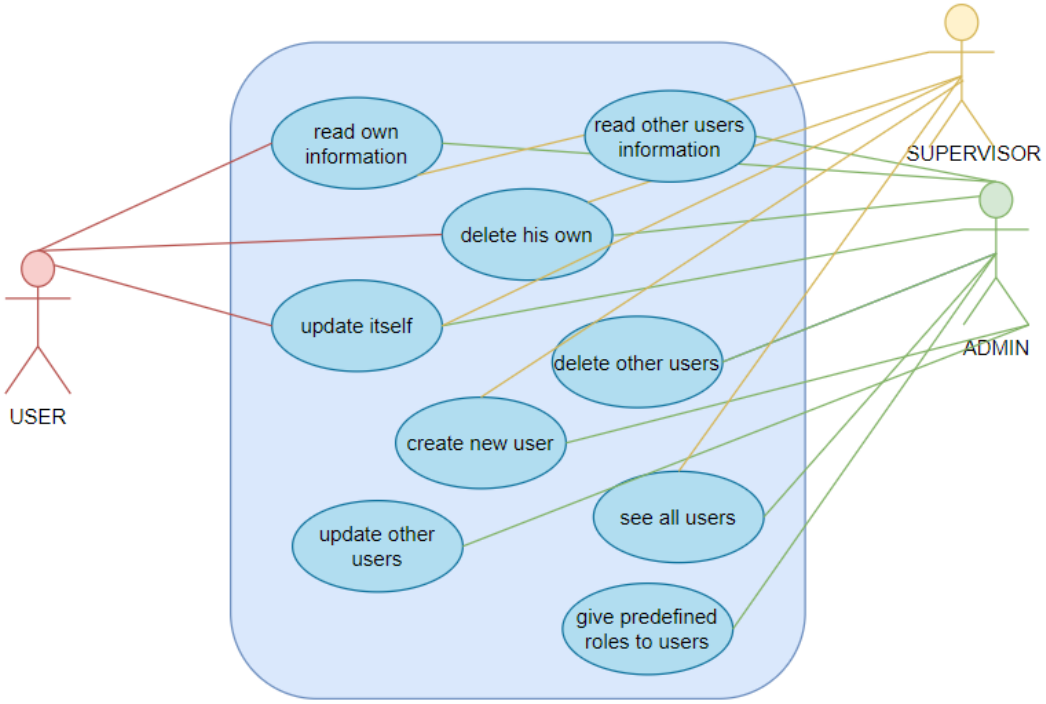
The Cardinalities as well as the methods and attributes of the individual classes are visible in the Detailed Design of the Class Diagram.

## DETAILED DESIGN



# *Use Case*

## Drawing




## Detailed description of update user

| | |
|---|---|
| **Use Case:** | Update a user's information |
| **Use Case ID:** | 1 |
| **Small description:** | The api updates a user's information, depending on his authority. |
| **Precondition:** | A valid user is logged in to the system and can access the page |
| **Actor (Primary):** | Logged in user |
| **Actor (Secondary):** | User who get's updated by the logged user |

**Main Flow:**

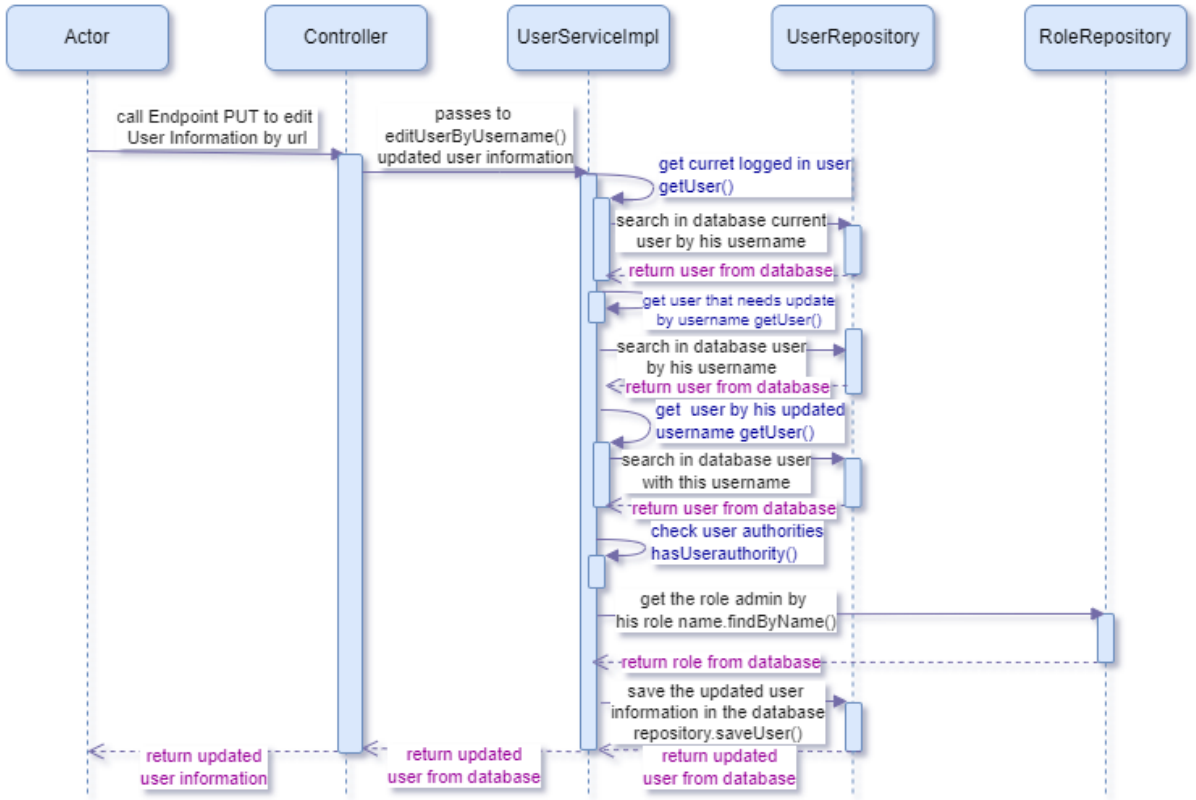
1. This use case begins when the user accesses the endpoint put by accessing to localhost:8080/Blog-Site/user/{username}
2. The user logs in successfully
3. Current user has the authentication to access
4. Redirect to the requested page
5. Entered the user information as a JSON format in postman, which contains new/updated information about the user, which you want to update
6. Get updated information from the controller
7. Get from the database the current logged in user's information by searching him by his username
8. Get updated username
9. Changed username to a unique one
10. Username is writing without any spaces
11. No one is assigned with this username
14. Current user has the authority to update the wanted user
15. Email updated
16. Password updated
17. Encoded updated Password
18. The role of the user is changed
12. After all this validation, the new information will overwrite the previous data
13. The processed data will be updated and stored in the database
14. The user has been updated successfully

| Post-Condition: | The updated user will be displayed |
|---|---|

Alternative Flows:
- User can't log in
- User has not the Authentication to access the page
- User doesn't have any authority to get the put endpoint, the http status 401 will be given with a user-friendly message
- User couldn't be found, the http status 404 will be given with a user-friendly message
- Username is already taken, the http status 409 will be given with a user-friendly message
- Username has spaces, it will be replaced with "_"
- Email or Passwort or Username or Role is null, changes won't be apply, it will keep old ones
- The current logged in user doesn't has the authority to update this user, the http status 401 will be given with a user-friendly message
- If the current logged in user hasn't the role admin, the roles won't be updated/edited

## *Sequence Diagram for Endpoint Put*

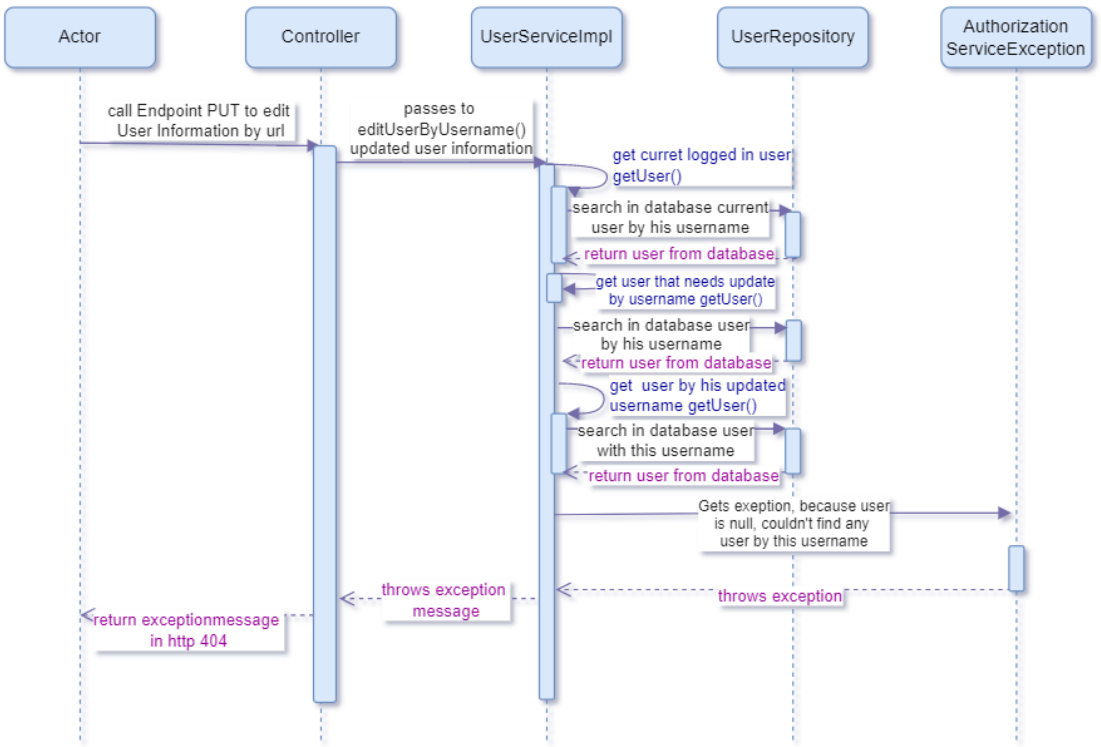
When everything goes well:



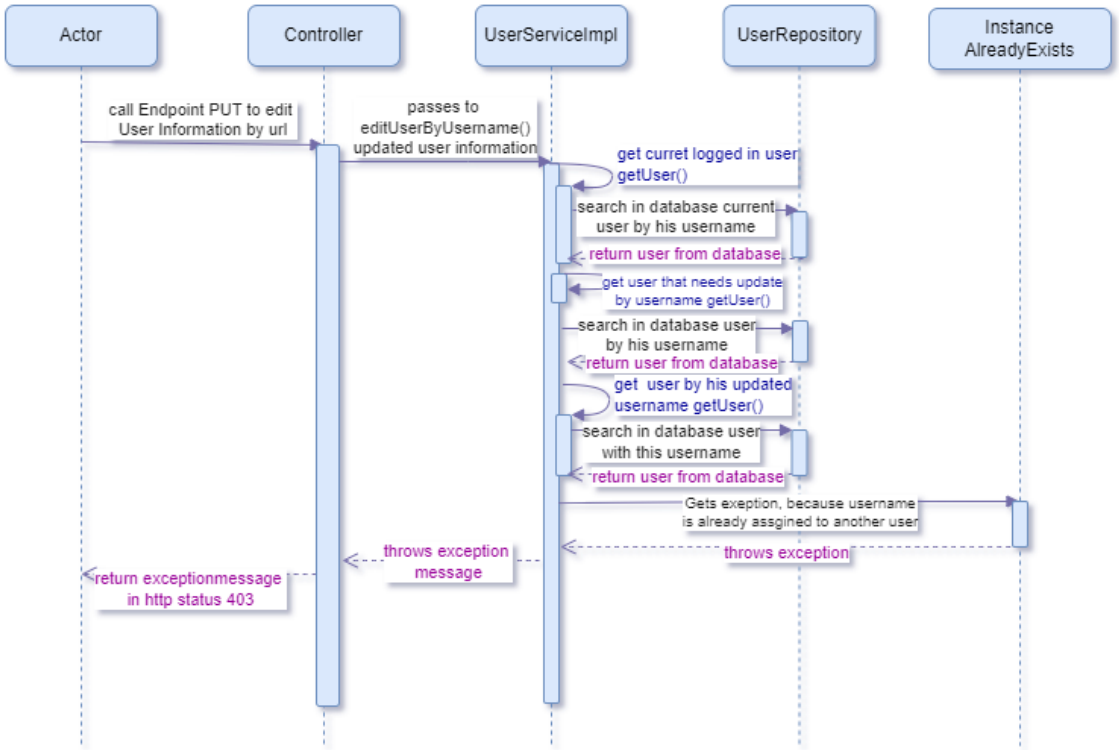

Requirement:
- Current logged in user has authentication
- Current logged in user has authority UPDATE_OTHERS if the user you want to update isn't himself
- User that you want to update exists
- The new username isn't assigned to anyone

## When exceptions are thrown:

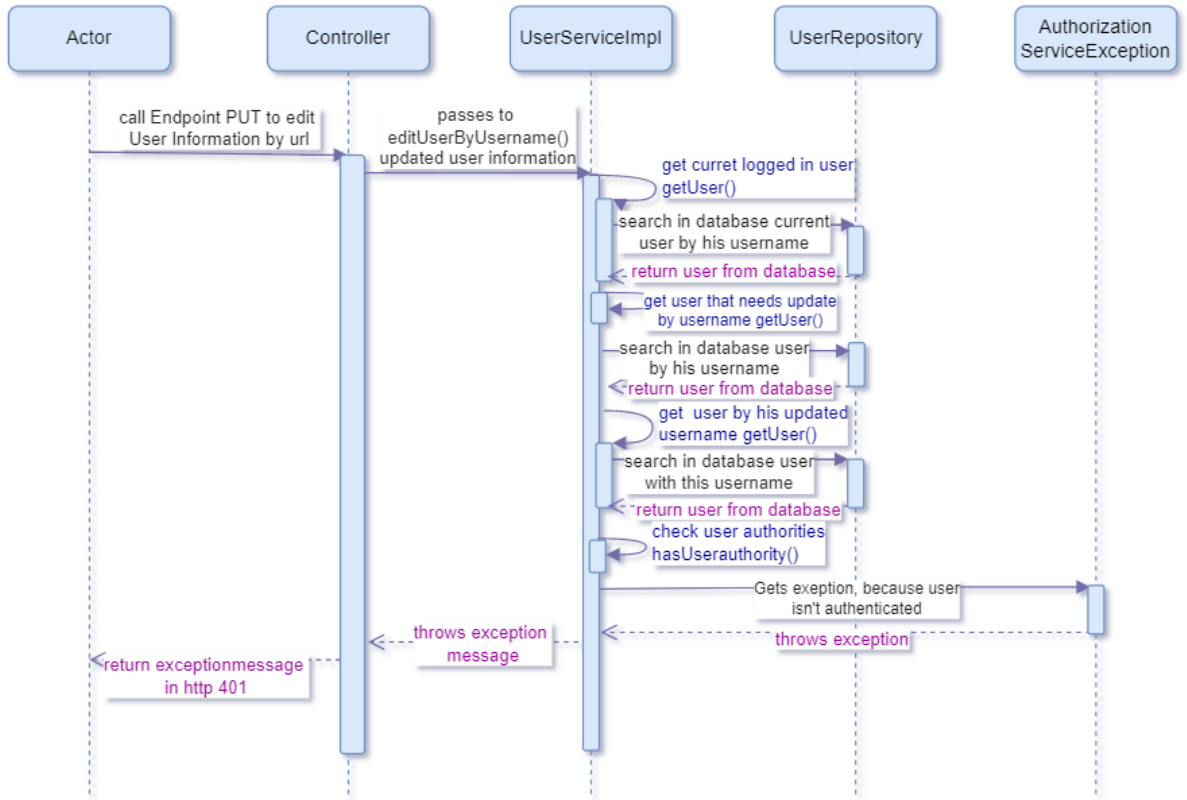
- User doesn't exist




- Username is already taken




- User hasn't authority to change/update user's information




# Authorization Matrix

| ROLES | ADMIN | USER | SUPERVISOR |
|---|---|---|---|
| READ | | | |
| Read own information | X | X | X |
| Read other users' information | X | | X |
| CREATE | | | |
| Create a user | X | | X |
| DELETE | | | |
| Can delete itself | X | X | |
| Can delete others | X | | |
| UPDATE | | | |
| Can update own information | X | X | X |
| Can add or updated user roles | X | | |
| Can update other users' information | X | | |

The authorization matrix shows all the roles and their authorities in the table above. The admin has all the authorities which is why he can execute all the CRUD operations. In Comparison to the admin the Supervisor, on the other hand, can't delete any user as well update any. The user can only read his own data and edit them as well.

# Endpoints

## GET all Users

### Controller

```
@GetMapping("/users")
@PreAuthorize("hasAnyAuthority('READ_ALL')")
public ResponseEntity getAllUsers() {
    try {
        return new ResponseEntity<Collection<User>>(userService.getAllUsers(), HttpStatus.OK);
    }catch (UserException e){
        return ResponseEntity.status(200).body(e.getMessage());
    }
}
```

This Endpoint is to get all Users. At first, we defined the path with the Annotation @GetMapping which is */users*. Then we defined the Authorities for this Endpoint to restrict the access, as you can see everyone with the Authority *READ_ALL* can access the list of all the users. In our case the Admin and the Supervisor. We used ResponseEntitiy as a return value to use HTTP-Status Codes. As you can see, we also catch Exceptions.

### User Service Implementation

```
@Override
public List<User> getAllUsers() throws UserException {
    if (userRepository.findAll().isEmpty()) {
        throw new UserException("No users entries");
    }
    return userRepository.findAll();
}
```

```
@Override
public User getUser(String username) {
    return userRepository.findByUsername(username);
}
```

The whole logic behind this endpoint is in the User Service Implementation. We must Override this method because the User Service Implementation implements from the User Service. The Return value is a List of Users. We have to check if there is any users in the database for this case an exception is thrown and the message *"No users entries"* is displayed.

### Repository

Repository extends from JPARepository which has pre-written methods such as findAll() or findByUsername() that can be used in the Service.

## GET User by username

### Controller

```
@GetMapping("/user/{username}")
@PreAuthorize("hasAnyAuthority('READ_OWN', 'READ_ALL')")
public ResponseEntity getUserByUsername(@PathVariable String username, Principal principal) {
    try{
        return ResponseEntity.ok().body(userService.findByUsername(username, principal));
    }catch (InstanceNotFoundException e){
        return ResponseEntity.status(404).body("User not found");
    }catch (UserException e){
        return ResponseEntity.status(401).body(e.getMessage());
    }
}
```

This is the Endpoint to get a certain user by his username. For this again we had to define the path with the Annotation @GetMapping. The username in the curly braces stands as a variable so that any username can be given in the path. This time we defined more than one Authority for the

endpoint. Anyone with the authorities *READ_OWN* and *READ_ALL* can access the information of a user. We also catch exceptions and return them with HTTP status and a user-friendly message, that we get partly from the class UserServiceImpl.

### User Service Implementation

```
@Override
public User findByUsername(String username, Principal principal) throws InstanceNotFoundException, UserException {
    if (hasAuthority(username, principal, authority: "READ_ALL")) {
        User user = getUser(username);
        if (user != null) {
            return user;
        }
        throw new InstanceNotFoundException("User " + username + " not found.");
    }
    throw new UserException("You don't have the authority to display this user.");
}
```

There is the logic from the endpoint we check certain points one of them is if the current user has the authority to see this user information and the other is the existence of the user. When one of these if-statements aren't full filled it will throw an exception with a user-friendly message, so that we can relate what went wrong.

### Repository

```
User findByUsername (String username);
```

As you can see, we implemented a method, which allows us to get the user by his username.

## POST

### Controller

```
@PostMapping("/user")
public ResponseEntity createUser(@RequestBody User newUser) {
    try {
        return ResponseEntity.ok(userService.createUser(newUser));
    }catch (InstanceAlreadyExistsException e){
        return ResponseEntity.status(409).body(e.getMessage());
    }catch (UserException e){
        return ResponseEntity.status(428).body(e.getMessage());
    }
}
```

This is the Endpoint to create a User. For this we defined the path with the Annotation @PostMapping. For the parameter we give the User along, it tries to call the method from the service else depending on the case the exception is catched and a user-friendly message is printed to understand what went wrong. As you can see this Endpoint does not contain the @PreAuhtorize Annotation, which is why, it's possible for everyone to create a new user.

### Service Implementation

```
@Override
public User createUser(User newUser) throws UserException, InstanceAlreadyExistsException {
    User user = new User();
    user.setEmail(newUser.getEmail().trim());
    user.setPassword(newUser.getPassword().trim());
    user.setUsername(newUser.getUsername().trim());
    user.setRoles(Set.of(roleRepository.findByName("USER")));

    if (!(user.getUsername().equals("") || user.getPassword().equals("") || user.getEmail().equals(""))) {
        if(getUser(user.getUsername()) == null){
            return saveUser(newUser);
        }
        throw new InstanceAlreadyExistsException("Username is already taken");
    }
    throw new UserException("All fields are required");

}
```

This is the logic behind the Put Endpoint. We created a method called createUser() which returns a User. This method catches two different Exception depending on the case because we had to check if the username is already defined or any of the required fields are empty if yes, a hint will be printed for the user else a new User is created. When creating a user, the role *USER* is automatically given and can be changed by the admin afterwards.

### Repository

Because the Repository extends from the JPARepository we can use the pre-written method in the service

## PUT

### Controller

```
@PutMapping("/user/{username}")
@PreAuthorize("hasAnyAuthority('UPDATE_OWN', 'UPDATE_OTHERS')")
public ResponseEntity editUserById(@RequestBody User editedUser,
                                   @PathVariable String username,
                                   Principal principal) throws InstanceNotFoundException {
    try {
        return ResponseEntity.ok().body(userService.editUserByUsername(editedUser, username, principal));
    } catch (UserException e) {
        return ResponseEntity.status(401).body(e.getMessage());
    } catch (InstanceAlreadyExistsException e) {
        return ResponseEntity.status(409).body(e.getMessage());
    } catch (InstanceNotFoundException e) {
        return ResponseEntity.status(404).body(e.getMessage());
    }
}
```

This is the Put Endpoint for which we also defined a path with the Annotation @PutMapping we also had to use @PreAuthorize to define the authorities for the following endpoint. The principal needed to be passed to get the current logged in user, because he can edit his personal information except his role. The admin on the other hand can edit anyone's information, he's the only one who can change the role as well. If anything goes wrong or doesn't match our requirements exceptions will be caught and the user gets a message to see what went did wrong.

### Service Implementation

```
@Override
public User editUserByUsername(User editedUser, String username, Principal principal)
        throws InstanceNotFoundException, UserException, InstanceAlreadyExistsException {
    User currentUser = getUser(principal.getName());
    User user = getUser(username);
    if (user == null)
        throw new InstanceNotFoundException("User " + username + " not found");
    if (!username.equals(editedUser.getUsername()) && getUser(editedUser.getUsername()) != null)
        throw new InstanceAlreadyExistsException("Username " + username + " is already taken");

    if (!hasAuthority(username, principal, authority: "UPDATE_ALL"))
        throw new UserException("You don't have the authority to edit user " + username);

    user.setEmail(editedUser.getEmail());
    user.setPassword(editedUser.getPassword());
    user.setUsername(editedUser.getUsername());
    if (currentUser.getRoles().contains(roleRepository.findByName("ADMIN"))) {
        user.setRoles(editedUser.getRoles());
    }
    return userRepository.save(user);
}
```

This is the logic behind the method editUserByUsername. We have looked at the different scenarios and caught all the possible exceptions. An if statement checks if the username is kept unique and only the admin would have access to change a role of a user. If not according to their actions a exception is thrown.

### Repository

Repository extends from the JPARepository which allow to use pre-written methods.

## DELETE

### Controller

```
@DeleteMapping("/user/{username}")
@PreAuthorize("hasAnyAuthority('DELETE_OTHERS', 'DELETE_OWN')")
public ResponseEntity deleteUser(@PathVariable String username, Principal principal) {
    try {
        return ResponseEntity.ok().body(userService.deleteUser(username, principal));
    }catch (InstanceNotFoundException e){
        return ResponseEntity.status(404).body(e.getMessage());
    }catch (UserException e){
        return ResponseEntity.status(401).body(e.getMessage());
    }
}
```

The Delete Endpoint has the same path as any other Endpoint which requires a certain username. This is defined with the @DeleteMapping annotation. The authorities are checked again, the user most have *DELETE_OTHERS* or *DELETE_OWN* to execute it. For certain use cases it will catch the exception and print out a user-friendly message to tell the user that the searched person was not found, or he doesn't has the authority.

### Service Implementation

```
@Override
public String deleteUser(String username, Principal principal) throws InstanceNotFoundException, UserException {
    User user = getUser(username);
    if (user != null) {
        if (hasAuthority(username, principal, authority: "DELETE_ALL")) {
            userRepository.deleteById(user.getId());
            return "User " + username + " has been deleted";
        }
        throw new UserException("You don't have the authority to delete the user " + username);
    }
    throw new InstanceNotFoundException("User not found");
}
```

In the logic of this endpoint, we used the principal again to get the current logged in user because he is allowed to delete his account and the admin again can delete anyone by the username. Like the other methods, we call our hasAuthority() method to check the authority of the user. If not a exception is thrown.

### Repository

Repository extends from JPARepository which allows to use the pre-written method.

## Exception

| Exception | Definition |
|---|---|
| 404 | Not found |
| 401 | Not Authorized |
| 403 | Not found |
| 409 | Conflict |
| 428 | Precondition Required |