Documentation

üK 223

Multi User Application

Andrin Rüegg, Giulia Villiger, Naima Cavegn

21.02.2025

Sources

Frontend: https://github.com/naica922/223_group3_blogProject_frontend/

Backend: https://github.com/naica922/223_group3_blogProject_backend

Table of contents

Project description

This is our full-stack application project for üK 223. We had to create a multi-user fullstack application for a blog website within 5 days. Each group got an individual entity, ours was "group". we had to think about which methods and attributes belong to it and structure for the database. A big part was also the testing and this documentation. Here you can find all our ideas and information about the project.

Initial situation

At the beginning of the project, we received a full-stack application with the functionality to log in as a user and to create, add and change users. Part of the routing was also implemented.

Goals and Task

The goal of the project was to implement a functional multi-user full-stack application with a Java Spring Boot backend, a React Typescript frontend and PostgreSQL as database. We focused on the logic and not the UI to ensure that our application was secure and contained all the required features.

Our individual entity was "groups", which have special conditions. For example, a user can only be in one class and does not have access to groups they do not belong to. However, the administrator should be able to perform CRUD operations on the groups.

Requirements

Functional

1. Extend existing roles and authorizations in the backend
2. A users personal information is only accessible to admins and themselves
3. Admins can create, modify and delete
4. Publicly accessible login page and homepage
5. Homepage for all logged-in users
6. Admin page only accessible to admins
7. Component "groups"
    - attributes members, name, motto, logo attributes
    - a user can only belong to one group
    - CRUD operations on endpoints
    - admin has rights to create, read, update, delete
    - only admins can see members and groups
    - users only have access to groups they belong to
8. REST endpoint with meaningful permissions
9. Part of the frontend should only be accessible for logged-in users and another part only for admins
10. Authentication implemented with a JSON web token

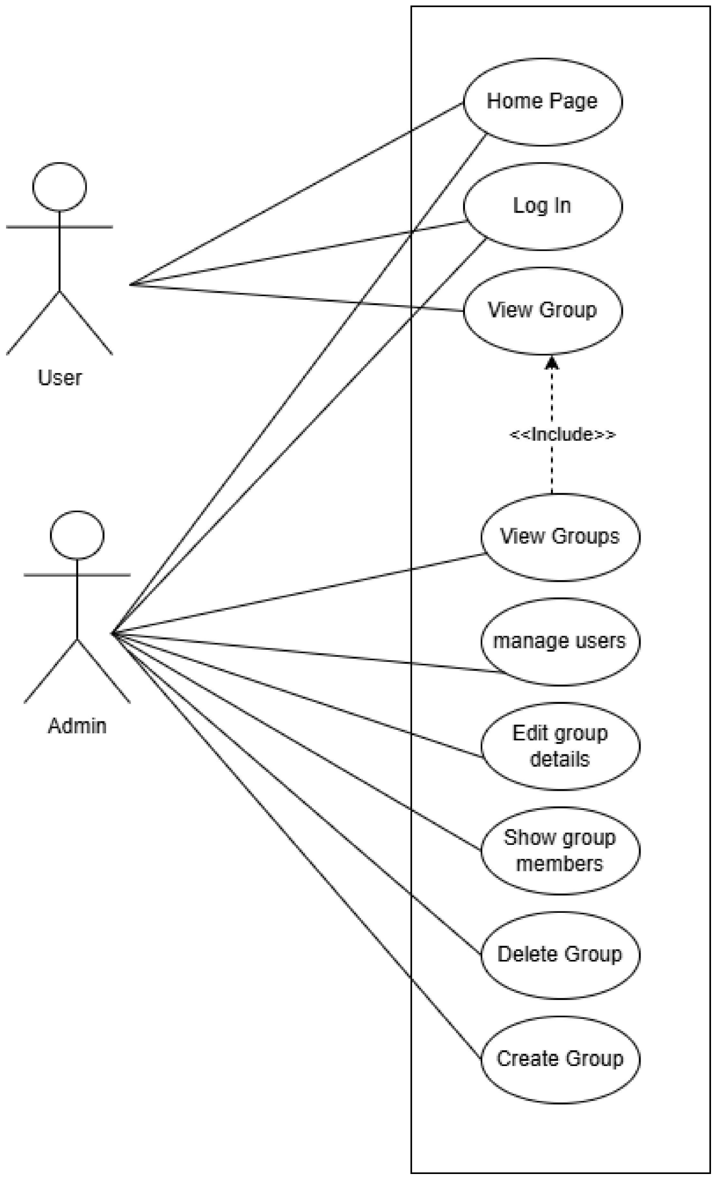
Non functional

1. Persisted data in PostgreSQL database, OR-Mapping with JPA
2. React Typescript Frontend and Spring Boot Java backend
3. Source Code uploaded to Git Repo
4. Testing with Cypress, Postman/JUnit
5. Test authorities at least one use case with cypress
    - multiple roles and users
    - at least one error and success test
    - use cases described with UML standard
    - ACID principles used

Diagrams

Use Case

The use case diagram shows two roles: user and admin, and what they can do. A standard user can only see the groups they belong to. An administrator ,on the other hand, can see all groups and has the right to manage users, edit and view group information, create new groups and delete groups.



Domain

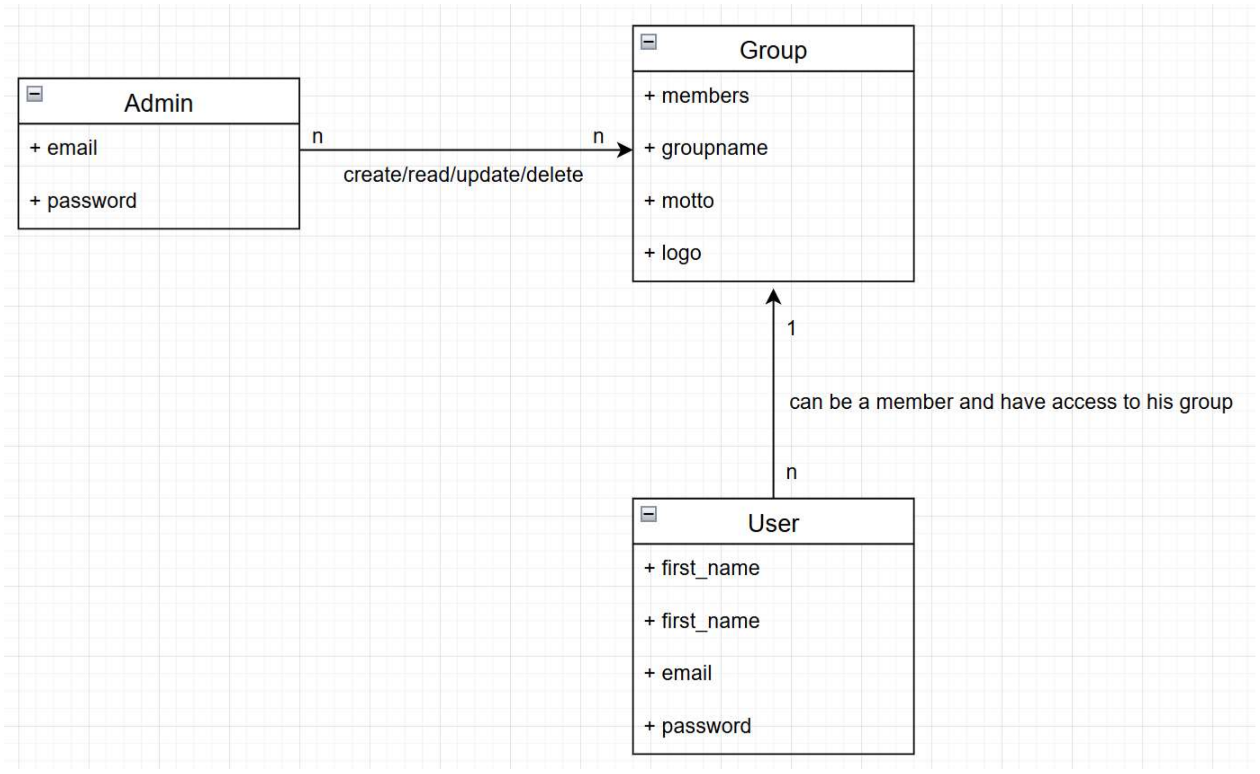
Our domain model represents a user, an administrator, our entity, group, and their members. A group has several attributes, such as members, group name, motto and logo. These attributes have already been assigned in the beginning.

User -> The user has fewer rights than an admin. Users have a first name, lastname and also an email with a password to log into the application. A user can belong to a maximum of one group but does not have to belong to any group. They can also only see the group to which they belong and not all others.

Admin-> The admin has a role with more rights than a simple user. It can perform the CRUD operations Create, Read, Update, Delete for all groups. It also has an email and a password.

Group -> Group entity is our individual part with certain attributes such as members, group name, motto and logo. A group can have multiple members and multiple administrators can also perform operations on it.

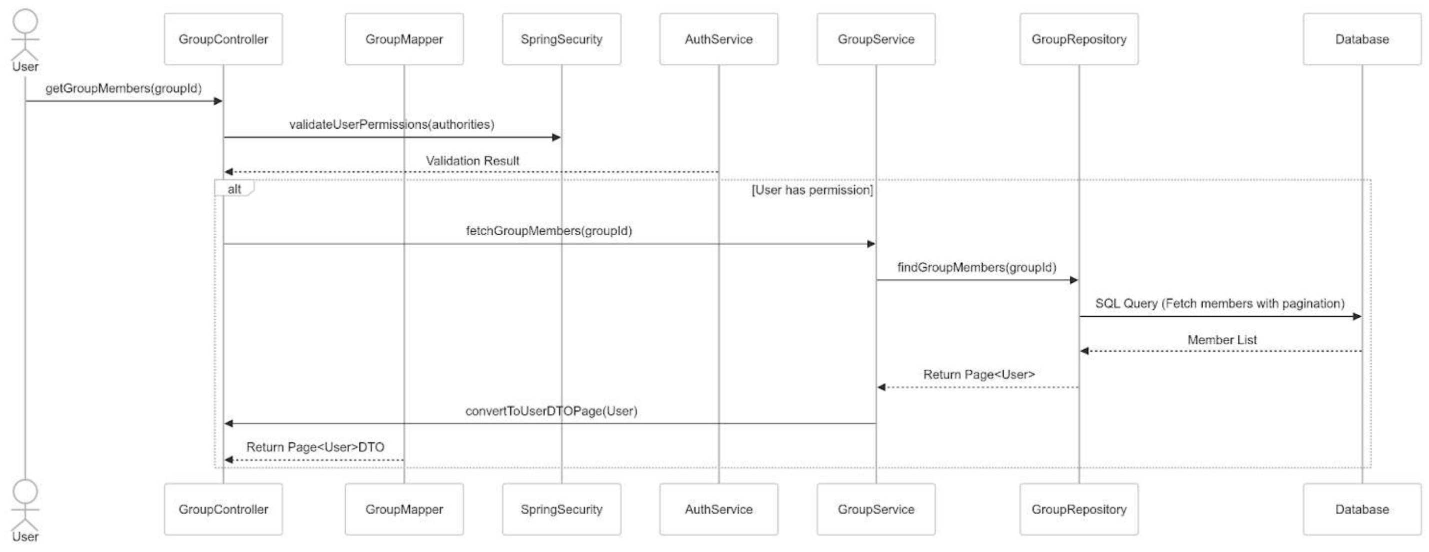
Member -> A member is a user who belongs to a group. Members can only read the attributes such as name, logo, etc., but cannot create, update or delete them.


Admin
+ email
+ password
n
n
create/read/update/delete
Group
+ members
+ groupname
+ motto
+ logo
1
can be a member and have access to his group
n
User
+ first_name
+ first_name
+ email
+ password

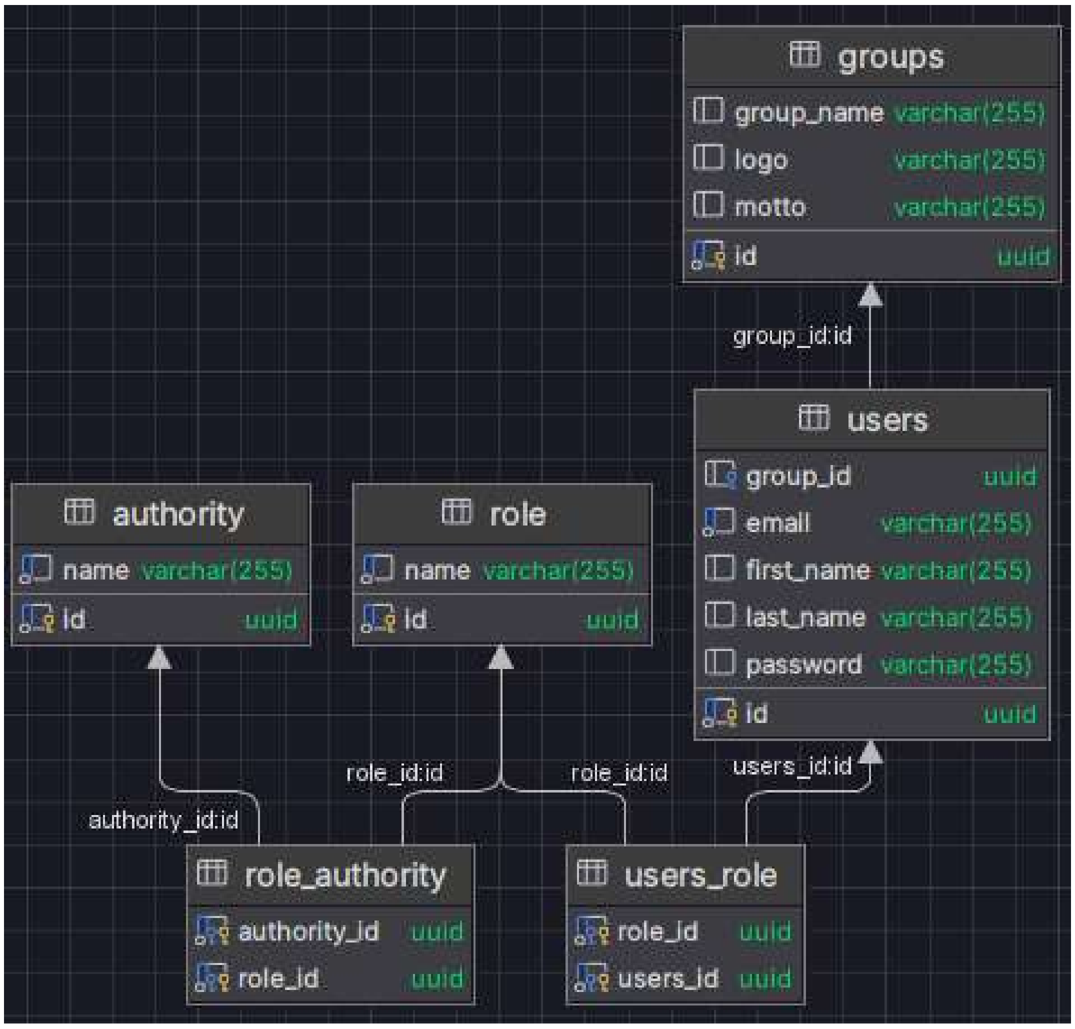
Sequence

This sequence diagram demonstrates the use case in which a user wants to access information about the group to which they belong.

1. User sends a GET request to the *GroupController* class.
2. The *GroupController* sends a request to the *Service* and checks if the user has the authority to get this data. He checks if the user has the correct role and belongs to the group.
3. The *GroupRepository* is called from the service to load the information of the group from the *database*.
4. To get the group and membership data, the *database* is called with a query to receive the data.
5. Our *Database* sends back the requested data to the *repository* and this sends it to the service.
6. After the service receives data, the information is sent to the *controller* which then represents it for the UI.
7. The frontend receives data and displays it on the application.



ERD


groups
group_name varchar(255)
logo varchar(255)
motto varchar(255)
id uuid
group_id:id
users
group_id uuid
email varchar(255)
first_name varchar(255)
last_name varchar(255)
password varchar(255)
id uuid
authority
name varchar(255)
id uuid
role
name varchar(255)
id uuid
role_id:id
role_id:id
users_id:id
authority_id:id
role_authority
authority_id uuid
role_id uuid
users_role
role_id uuid
users_id uuid

Use-Case descriptions

This is the detailed use-case description from a user that wants to access the data from the group he belongs to. It is the same scenario as in the sequence diagram.

Actor:	User (member of a group)
Description:	The user wants to access the information of the group he belongs to
Preconditions:	- User must be logged in and be member of a group - Permissions have to be correct
Postconditions:	- Data of the group (motto, name, etc.) that the user wants to access is shown on the application
Normal course:	1. User logs into the application 2. Home page is visible 3. Clicks on the button “Visit my group details” 4. Gets redirected to another page where the data of his group is displayed
Alternative course:	1. User has no group - User sees no group in the group overview 2. Wrong credentials - User can only access the homepage for users that aren't logged in
Exceptions:	When the credentials are wrong, user can't log in and gets an error message

This use case diagram is the scenario of an admin that wants to create a new group in our application.

Actor:	Admin
Description:	Admin wants to create a new group with the data group name, motto, logo.
Preconditions:	- Admin must be logged in and as an admin
Postconditions:	- Group gets created and values inserted into the database - Group is visible in the list for admins and users apart of that group
Normal course:	1. Admin logs into the page with the correct credentials 2. When successfully logged in get redirected to the homepage for logged in users 3. Navigate to the menu to create groups and enter requested information 4. Add members to the group 5. Admin clicks on “create group” 6. If necessary information is provided group gets created 7. System shows a confirmation 8. Group is now visible in the list
Alternative course:	1. Missing data: - system shows an error for the empty field/fields 2. Users that were in another group before get removed from the old one and added to the new one
Exceptions:	-

Testing strategy

We defined our test methods and test use cases right at the start of the project to ensure that all cases were completed and the tools defined.

Functional tests

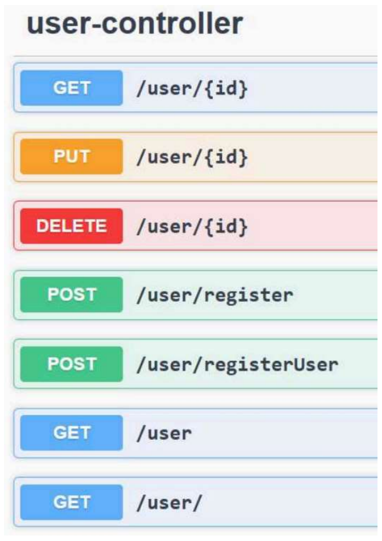
Component Tests - We chose Postman for the backend tests for several reasons. One reason was that we were already familiar with it, as we had used it in other projects. Another advantage of Postman is the user-friendly interface.
With postman we can test all endpoints of our backend. We will use postman to verify that the data we receive is the data we want different users to receive.
The only disadvantage of this strategy is that we need to check if the data we receive is the data that a user with a specific role should receive and if other operations such as create, update or delete are also performed correctly. We have to manually check whether our objects in the database are actually changed.

E2E Cypress - We plan to implement the end-to-end tests with the Cypress tool. These tests will help us to check whether the process of using the application by our user and the interaction with the backend is correct.
This will also allow us to test our second use case description test. We will also test whether the functionality works with the different roles.

Non functional

Lighthouse - The Lighthouse tool is used in our project to test performance. We can analyze how accessibility, scalability, etc. are doing and improve the application if we think it makes sense.

Swagger


user-controller

GET /user/{id}
PUT /user/{id}
DELETE /user/{id}
POST /user/register
POST /user/registerUser
GET /user
GET /user/

Description Endpoints entity Group

GET /groups - returns a list of all existing groups. Admins and users can access it, but only admins receive all groups. Users will only receive the groups they are a part of.

GET /groups/{groupId} - calls a group with the id. The endpoint is used to retrieve the data of a specific group such as name, motto and logo. Only users that are a part of this group and admins will receive the details of the group when calling this endpoint.

POST /groups - creates a new group, only admins can perform this operation. They must provide the data for the new group.

PUT /groups - only admins can access this page, as users are not allowed to change groups.

DELETE /groups/{groupId} - deletes a group by its id. Only admins are allowed to do this operation, as users do not have the rights to do so.

Project Set Up

Backend

1. To use our backend clone the repository first:

```
git clone https://github.com/naica922/223_group3_blogProject_backend.git
```

2. Download docker desktop and run this command in your terminal:

```
docker run --name postgres_db -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -p 5432:5432 -d postgres
```

Open the Project in IntelliJ Ultimate and use gradle to build and then bootRun the application.

Frontend

1. Clone Repository local

```
git clone https://github.com/naica922/223_group3_blogProject_frontend.git
```

2. Yarn

The second step is installing yarn (note: you should be in the react_frontend folder):

```
npm install - - global yarn
```

```
yarn install
```

3. Run frontend

You can start our frontend with the following command and open localhost:3030.

```
yarn start
```