Study Buddy

A student-oriented meeting application to find study groups.

Software Design project

Table of Contents

1. Project Objective

The objective of this project is to create a functioning web application where students can join study groups with people with similar interests. The application should implement useful functionalities for the end users and while using software design principles and appropriate architectures. The idea is to create a product that facilitates how students find resources when studying by pairing them with other students that have similar interests and academic needs. Thus, the main objective of this project is to create a social network focused mainly on student study groups.

2. Use cases and functionalities

a. Use cases:

- Finding study partners for group assignments: Users can create a group or join existing ones to find other students to work with on group assignments
- Organizing study sessions: Students can use Study Buddy to organize study sessions with other members of the group, helping each other to prepare for exams and assignments
- Finding like-minded peers: Students can use Study Buddy to find and connect with like-minded peers who are studying the same subjects, making the learning process more enjoyable
- Finding study groups for specific topics: Users can search for existing study groups or create their own for specific subjects or topics, making it easier to find other students who are studying the same things
- Time management: Users can schedule study sessions or create task lists to help them manage their time and stay on track with their studies
- Geolocation tracking: a student can see what the groups' location are and can export the location in Google Maps to get directions

b. Functionalities:

- A user can register and login with an account that has a unique email and username
- The application returns client errors and displays them as pop-up messages which contain useful information for the user
- Users can have a normal role, or a admin role which lets the admin manipulate data regarding other users, such as locking or deactivating accounts
- Users should be able create, view, join and leave groups
- If a user creates a group, that user is automatically the admin of the group, and therefore, can update the group name, description and the location of the meetings
- A group admin can set new meetings using a calendar
- The meetings of a group are ordered chronologically and highlighted with red if they are in the past
- The admin of the group can set which topics the study group will focus on (for example *Math* or *Biology*)
- The admin of a group can also kick members of the group or promote other members to become the admin

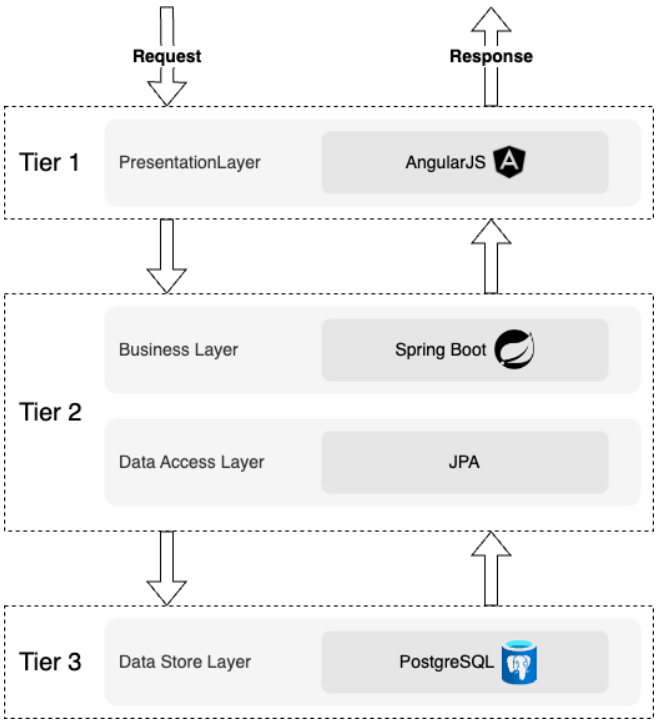
- The application shows notifications when an action is performed
- A user can see clearly on a web page the names, descriptions, topics, location and next meetings of all the groups
- The user can also filter the groups displayed in the main web page based on where that user is either an admin or a member of
- The user has a meetings calendar page where all the meetings scheduled are shown from all the groups where that user is a member of ordered chronologically
- The user can filter past meetings from the calendar, as well as applying the same filters as in the main group page: see meetings where user is admin, see meetings where user is just a member of the group
- Like the calendar page, each user has a map page where it a map displays the locations of each group place, show as markers on a Google Maps
- The user can also filter the locations shown on the map by where a user is admin or just a member of the group

c. Non-Functional requirements:

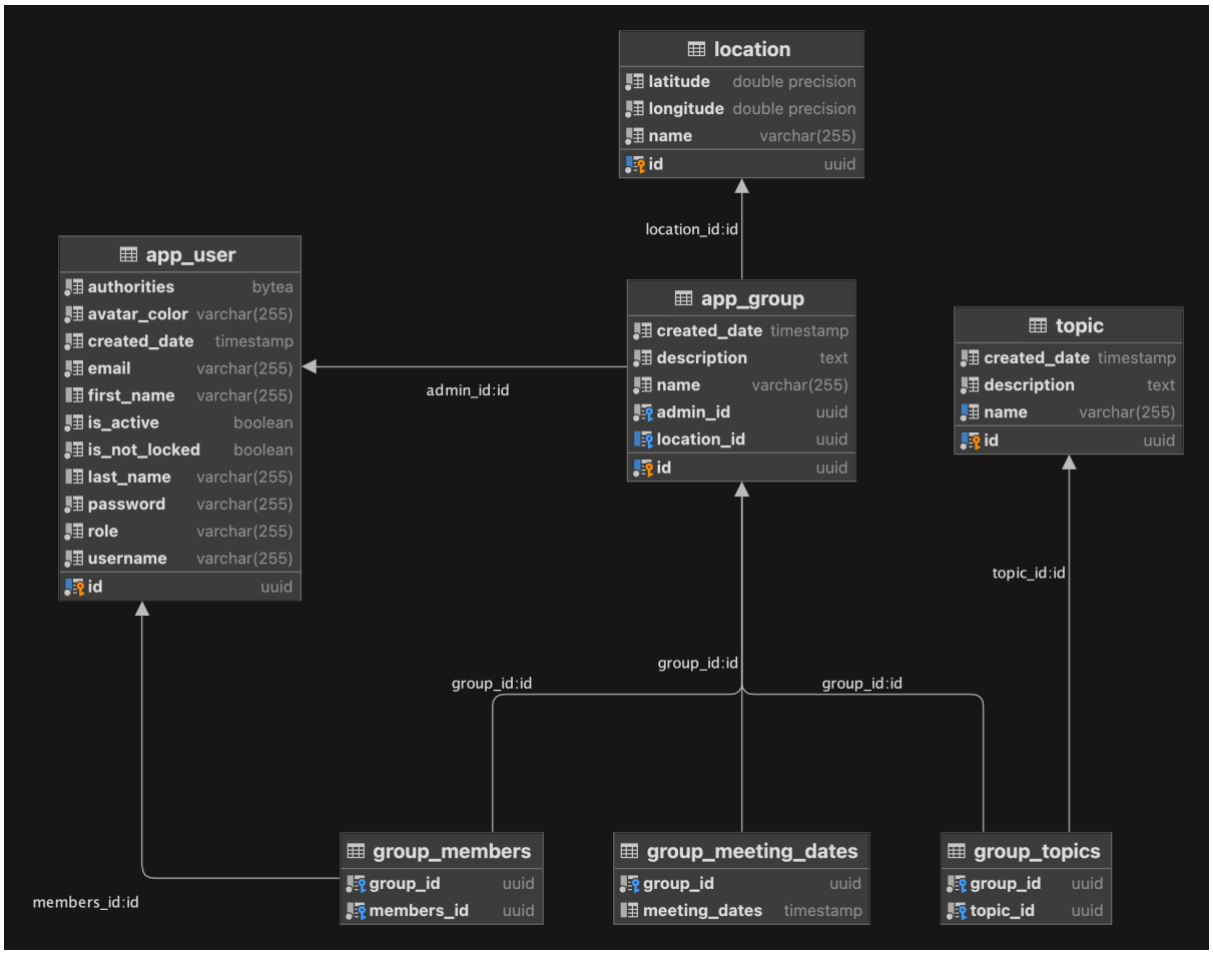
- The application should have strong security by securing transactions with JWT
- The application should encrypt all passwords and authorities of a user
- The application should be responsive
- The app should have a user-friendly design with intuitive features
- The application should be well documented with Javadoc

3. Design

The *Study Buddy* application was designed with the layered architecture in mind. The client-side of the app was designed in Angular; the server side of the app was developed in Spring Boot and the database was created using PostgreSQL. This diagram shows the layered architecture of the application with the frameworks and technologies that were used:



In order to implement the functionalities mentioned at the previous chapter, several relationships between data entities must be established. The database diagram shows what relationships must be had for the application to implement all functionalities:

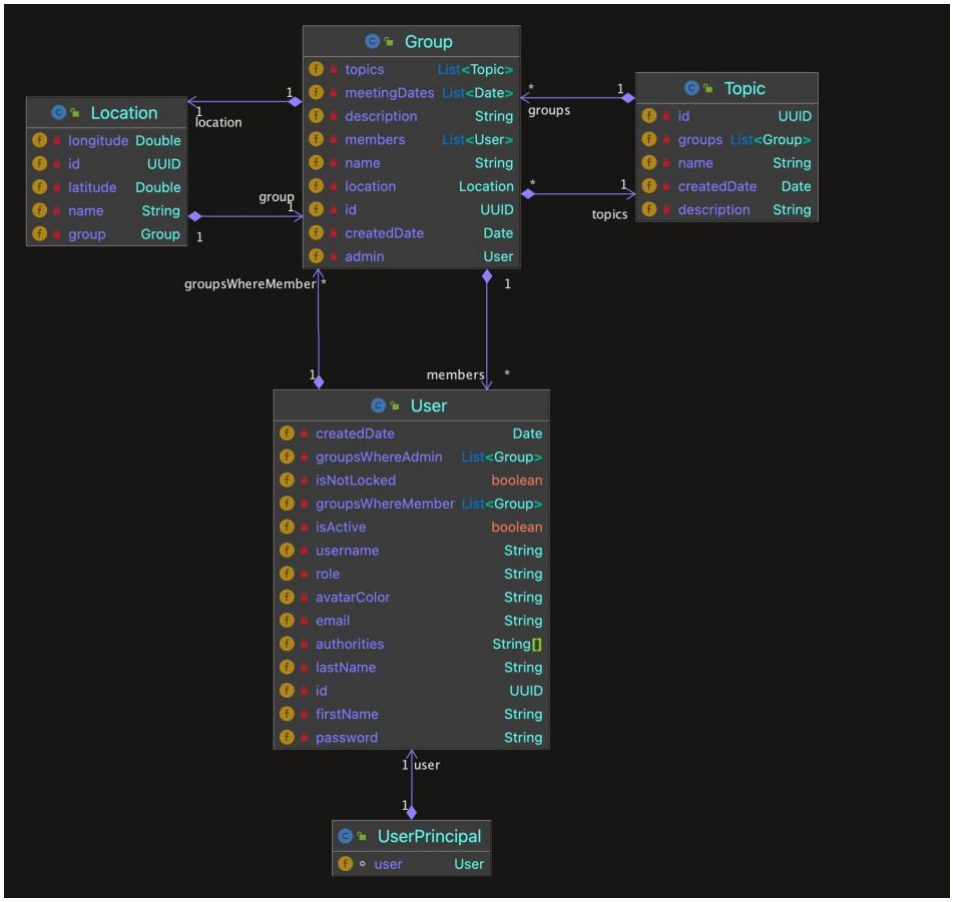


Here we can see four main entities that define the application: ***app_user***, ***app_group***, ***location*** and ***topic***. The *User* and *Group* tables have been renamed to “app_user” and “app_group” because using the default names was forbidden in PostgreSQL, those being keywords in Postgres. The relationships between these entities are as follows:

- The *User* entity (defined *app_user*) contains:
  - Id (UUID type)
  - Created date (Date type)
  - First name
  - Last name
  - Username
  - Email
  - Avatar color
  - Password (encrypted byte string)
  - Role (String type)
  - Authorities (encrypted byte string)
  - Active parameter (Boolean type)
  - Locked parameter (Boolean type)

    - A list of groups where user is member (Many-To-Many relationship with Group table through “group_members” join table)
    - A list of groups where the user is an admin (One-To-Many relationship with Group table through “admin_id” foreign key parameter)
- The *Group* entity (defined by *app_group*) contains:
    - Id (UUID type)
    - Created date (Date type)
    - Name
    - Description
    - Admin (Many-To-One relationship using “admin_id” foreign key column)
    - Location (One-To-One relationship with Location Table through “location_id” foreign key)
    - A list of meeting dates (One-To-Many relationship with collection table of timestamps, mapped by “group_id” foreign key)
    - A list of member users (Many-To-Many relationship with User table through “group_members” join table)
    - A list of topics (Many-To-Many relationship with Topic table through “group_topics” join table)
- The *Location* entity contains:
    - Id (UUID type)
    - Name
    - Latitude
    - Longitude
    - Group (One-To-One relationship with Location Table through “location_id” foreign key)
- The *Topic* entity contains:
    - Id (UUID type)
    - Create date (Date type)
    - Name
    - Description
    - A list of groups where topic is present (Many-To-Many relationship with Group table through “group_topics” join table)

In Java the relationships in between the entities defined in the domain package are:



4. Implementation

a. Entities

In order to reduce boiler plate code, the Lombok package was used in the backend application, thus reducing writing a lot of getters, setters, constructors and other repetitive methods such as “toString()”, “equals()” or “hashCode()”. This package also implemented the *Builder* design pattern when using the corresponding annotation. The *Builder* design pattern was used throughout all the entity packages and in all the data transfer object classes. Another implementation that was possible was by using the Java records introduced in Java 14, however the by using Lombok annotations we allow compatibility with older versions of Java.

Here is an example of the User entity implementation:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@ToString
@Builder
@Entity
@Table(name = "app_user")
public class User implements Serializable {
    @Id
    @Type(type = "pg-uuid")
    private UUID id;
```

```
    // ...
}
```

b. Security

This application was developed with JWT security in mind, therefore the security package of the contains filters, an entry point, a token provider, security constants, roles and authorities that define who has access to what information. It should be mentioned that security is doubled both on the server side and on the client side by guards. On the server side however there are classes which verify if the required HTTP headers are present, if the correct HTTP method is allowed, if the token is valid and if the authorities permit the user to execute the requested action.

In the security package, class contains the possible roles the user can take and the associated authorities as constants:

```
public class Authority {
    public static final String[] USER_AUTHORITIES = {
            "user:read",
            "topic:read",
            "topic:create"
    };
    public static final String[] SUPER_ADMIN_AUTHORITIES = {
            "user:read",
            "user:update",
            "user:create",
            "user:delete",
            "topic:read",
            "topic:update",
            "topic:create",
            "topic:delete",
    };
}
```

There are only two roles in the current version, these roles are “USER” and “SUPER_ADMIN”. The *SUPER_ADMIN* is supposed to have all the authorities a user can have, thus enabling the admin to execute CRUD operations of all entities, while the normal user should have the minimum number of authorities that allow a consumer interaction with the system.

c. Controllers

The backend endpoints for this application were organized in the “controller” package. Here, all the classes annotated with “@Controller” annotation implement a *Singleton* design pattern. The methods in each controllers associate the URL of the endpoints with the methods in the services that execute changes in the system. For example, the Topic controller can call the CRUD operations of the Topic entity in the service:

```
@RestController
@RequestMapping(path = {"/topics"})
public class TopicController {
    private final TopicService topicService;

    @Autowired
    public TopicController(TopicService topicService) {
        this.topicService = topicService;
    }

    @GetMapping()
```

```
    @PreAuthorize("hasAnyAuthority('topic:read')")
    public List<TopicDTO> getTopics() {
        return this.topicService.getAllTopics();
    }

    @PostMapping("/create")
    @PreAuthorize("hasAnyAuthority('topic:create')")
    public ResponseEntity<String> createTopic(TopicDTO topicDTO) throws
TopicExistException {
        this.topicService.createTopic(topicDTO);
        return new ResponseEntity<>(HttpStatus.OK);
    }

    @PostMapping("/update")
    @PreAuthorize("hasAnyAuthority('topic:update')")
    public ResponseEntity<String> updateTopic(TopicDTO topicDTO) throws
TopicNotFoundException {
        this.topicService.updateTopic(topicDTO);
        return new ResponseEntity<>(HttpStatus.OK);
    }

    @DeleteMapping("/{id}")
    @PreAuthorize("hasAnyAuthority('topic:delete')")
    public ResponseEntity<String> deleteTopic(@PathVariable("id") UUID id)
throws TopicNotFoundException {
        this.topicService.deleteTopic(id);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

In the method “updateTopic(TopicDTO topicDTO)” the update method from the service is called only if the user has the required authorities, in this case ”topic:update”. The body of the request is automatically mapped to the DTO.

d. Service

All main business operations are done in the dedicated services. For example, if an admin wants to kick a user from a group, the operation and verification is done in the group service. Therefore, in this case the service package represents the business layer of the system.

Each service also contains mapper methods that convert data level objects to data transfer objects. Further documentation of most methods lies in the Javadoc comments of each method.

e. Frontend Implementation

The frontend was developed using the Angular 15 framework, the project being structured in components defined by their packages. There are two types of pages in the application: pages that do not require authentication and pages that require authentication. The only pages that do not require authentication are the login page and the register page. The rest of the pages requires the user to log in in order to access the page. All pages that require authentication are defined by a component which can be accessed by a router outlet. The main component which defines this set of authentication only pages is called “main-app”, and the access to all child pages are protected by role and authentication guards.

All components that define pages can also contain nested components for the functionalities that are specific to that page. For example, the "*groups*" page component, which displays all groups, contains the component called "*individual-group*" which represents the page for an individual group with all its information and functionalities.

5. Future Improvements

Although as Study Buddy works as intended as a standalone application, during the development process, certain features were considered, but were not implemented as due to time restrictions and their lower priority for the system. Such functionalities were categorized as "nice-to-have" rather than "must-have", and they include:

- Page dedicated for CRUD operations for Topics
- Description parameter for Topics (not fully implemented)
- Courses entity that inherits Topic properties with special characteristics such as:
    - Professor name
    - Extra information regarding exams and midterms
- Calendar format for user's scheduler page
- Calendar format in individual group's page
- Dashboard page with useful information
- User settings page for updating information
- Admin page for managing users, especially the fields regarding the accessibility:
    - The "Active" field
    - The "Locked" field
- Improved UI:
    - Dynamic header
    - Spinner wheel service
    - Bug fixes regarding CSS and HTML
    - Consistent color scheme
- Improved Security
    - Automatically lock account against brute force login attempts
    - Add HTTPS communication

These are just a few suggestions identified as possible improvements during the development.

6. Conclusion

This application was developed with the purpose of applying theoretical aspects of software design and system architecture. *Study Buddy* presents a user friendly, functionality rich web application designed to enhance the studying experience for students. By providing a platform for creating and joining study groups with like-minded individuals, Study Buddy aims to foster collaboration, support, and knowledge sharing among students.

With *Study Buddy*, students can easily connect with peers who have similar interests, forming study groups that enable efficient group assignments, project collaboration, and effective exam preparation.

7. Bibliography

https://www.udemy.com/course/jwt-springsecurity-angular/
https://www.baeldung.com/get-user-in-spring-security
https://www.baeldung.com/jpa-many-to-many
https://refactoring.guru/