UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Breddit

Software Engineering

Authors: Borza Andrei
Group: 30236

Faculty of Automation and Computer Science

22 March 2024

Contents

1 Introduction

Welcome to the documentation for my innovative new project – Breddit – a simplified Reddit clone designed to offer a seamless user experience. Drawing inspiration from the popular platform, Reddit, Breddit aims to replicate its core functionalities while introducing innovative features to enhance user engagement and satisfaction.

The primary objective of Breddit is to create a user-friendly platform that caters to the diverse preferences of our community. By adopting a familiar interface reminiscent of Reddit, our goal is to facilitate a smooth transition for users while introducing unique functionalities that distinguish Breddit from its counterparts.

My mission with Breddit is to revolutionize the community-driven content experience. Some of our key features include profile enhancements, the ability to interact with friends, and access to a wide range of content, all in one convenient location – Breddit.

2 Deliverable 1

2.1 Project Specification

2.1.1 Overview

The purpose of this section is to outline the specifications and requirements of the Breddit project. This includes defining the scope, objectives, and key features of the platform.

2.1.2 Scope

Breddit aims to provide a simplified Reddit-like platform that allows users to discover, share, and engage with a variety of content in a user-friendly environment. The platform will include features such as user profiles, subreddit creation, voting, commenting, and content sharing.

2.1.3 Objectives

The main objectives of the Breddit project are as follows:

- To create a user-friendly interface that facilitates easy navigation and content discovery.
- To implement core features such as user registration, content submission, voting, commenting, and user interaction.
- To ensure scalability and performance to accommodate a growing user base and increasing content volume.
- To prioritize user privacy and security by implementing robust authentication and data protection measures.
- To foster community engagement and interaction through features like user profiles, direct messaging, and content sharing.

2.1.4 Key Features

Breddit will offer the following key features to its users:

- User Registration and Authentication: Users will be able to create accounts, log in securely, and manage their profiles.
- Subreddit Creation and Management: Users can create and moderate their own communities (subreddits) based on their interests.

- Content Submission: Users can submit various types of content, including text posts, links, images, and videos.
- Voting and Ranking: Users can vote on posts and comments, influencing their visibility and ranking within the platform.
- Commenting: Users can engage in discussions by commenting on posts and replying to other users' comments.
- Content Filtering and Search: Users can filter content based on preferences and search for specific topics or keywords.

2.1.5 Constraints

The Breddit project will operate within the following constraints:

- Time Constraints: The project must be completed within the specified timeline to meet the target release date.
- Resource Constraints: The project will be developed using available resources, including technology stack, personnel, and budget.
- Compatibility Constraints: The platform must be compatible with a variety of devices and browsers to ensure accessibility for all users.

2.1.6 Assumptions

The following assumptions have been made in the development of the Breddit project:

- Users have basic internet connectivity and access to modern web browsers.
- Users are familiar with common social media and content-sharing platforms, such as Reddit.
- The platform will initially target a general audience but may later expand to cater to specific niches or communities.

2.2 Functional Requirements

2.2.1 User Authentication

- **Description:** Users should be able to register an account with Breddit, providing necessary information such as username, email address, and password.
- **Acceptance Criteria:** Users can successfully register an account and log in using their credentials. Passwords are securely stored and encrypted.

2.2.2 User Profile

- **Description:** Users should have the ability to customize their profiles by adding personal information, profile pictures, and bio.
- **Acceptance Criteria:** Users can access and update their profiles, including adding or changing profile pictures, updating personal information, and editing their bio.

2.2.3 Content Submission

- **Description:** Users should be able to submit various types of content, including text posts, links, images, and videos.

- **Acceptance Criteria:** Users can create new posts with the option to include text, links, images, or videos. Posts are displayed within the appropriate subreddit or on the user's profile.

2.2.4 Voting and Ranking

- **Description:** Users should be able to vote on posts and comments to influence their visibility and ranking within the platform.
- **Acceptance Criteria:** Users can upvote or downvote posts and comments. The ranking algorithm takes into account the number of votes, recency, and other factors to determine post visibility and ranking.

2.2.5 Commenting

- **Description:** Users should be able to engage in discussions by commenting on posts and replying to other users' comments.
- **Acceptance Criteria:** Users can submit comments on posts and replies to existing comments. Comments are displayed in threaded format for easy readability and navigation.

2.2.6 Subreddit Creation and Management

- **Description:** Users should have the ability to create and manage their own communities (subreddits) based on their interests.
- **Acceptance Criteria:** Users with sufficient privileges can create new subreddits, set subreddit rules, and moderate content within their communities. Subreddit creators have access to tools for managing moderators, banning users, and enforcing subreddit rules.

2.2.7 Content Filtering and Search

- **Description:** Users should be able to filter content based on preferences and search for specific topics or keywords.
- **Acceptance Criteria:** Users can filter content by categories such as "hot," "new," and "top," and apply additional filters based on factors like time range, subreddit, and content type. A search feature allows users to search for posts, subreddits, or users using keywords.

2.2.8 Direct Messaging

- **Description:** Users should have the ability to send direct messages to other users for private communication.
- **Acceptance Criteria:** Users can send and receive direct messages to and from other users. Messages are private and only visible to the sender and recipient(s).

2.2.9 Notification System

- **Description:** Users should receive notifications for relevant activities, such as new messages, replies to their comments, and mentions.
- **Acceptance Criteria:** Users receive real-time notifications for various activities within the platform. Notifications are customizable, allowing users to specify their preferences for receiving notifications.

2.2.10 Content Moderation

- **Description:** The platform should include tools for content moderation to ensure compliance with community guidelines and prevent abuse.
- **Acceptance Criteria:** Moderators have access to tools for removing or hiding content that violates community guidelines, banning users, and managing reported content. Users can report inappropriate or abusive content for review by moderators.

2.3 Use Cases Identification

As a large-scale project, there rises a need for identifying the main flows and ensure that they are correct and do not impact the user experience at all. So, for that, I came up with some of the app's core flows, presented as following:

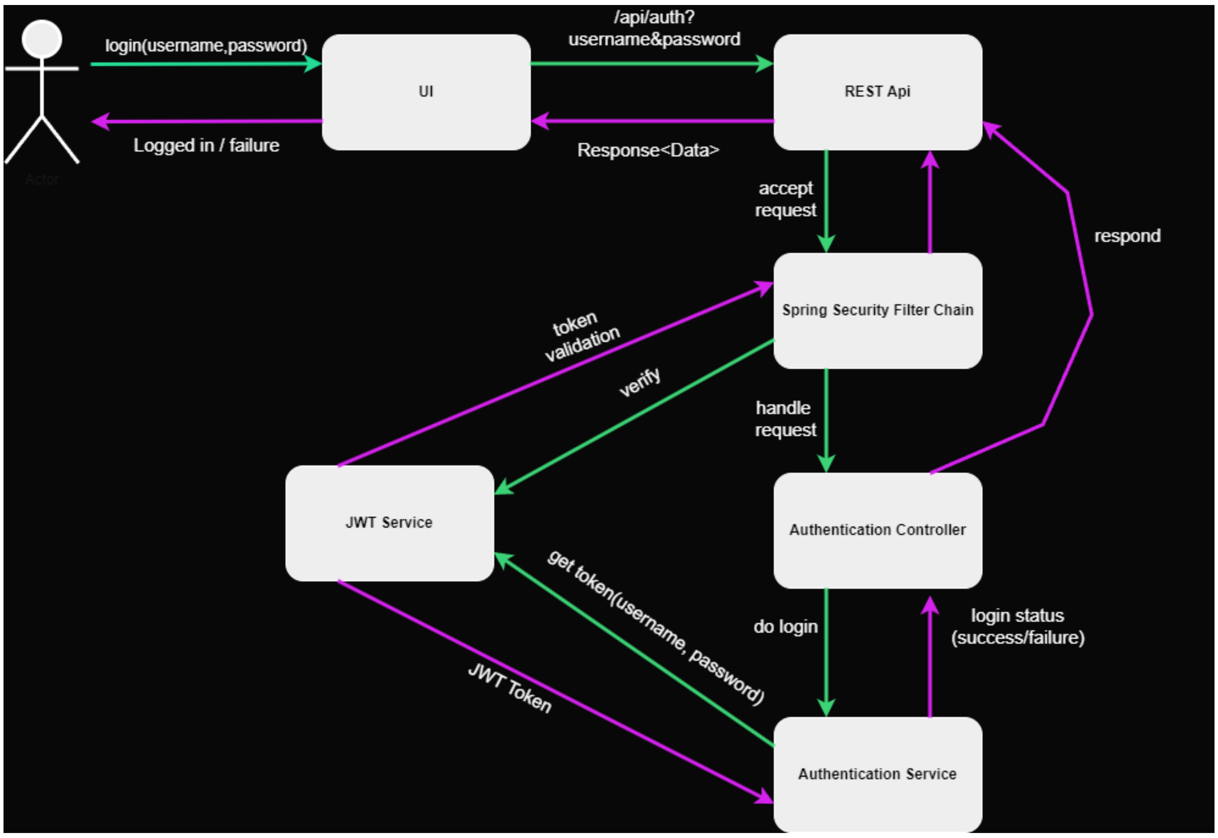


Figure 1: Communication Diagram for Authentication Flow

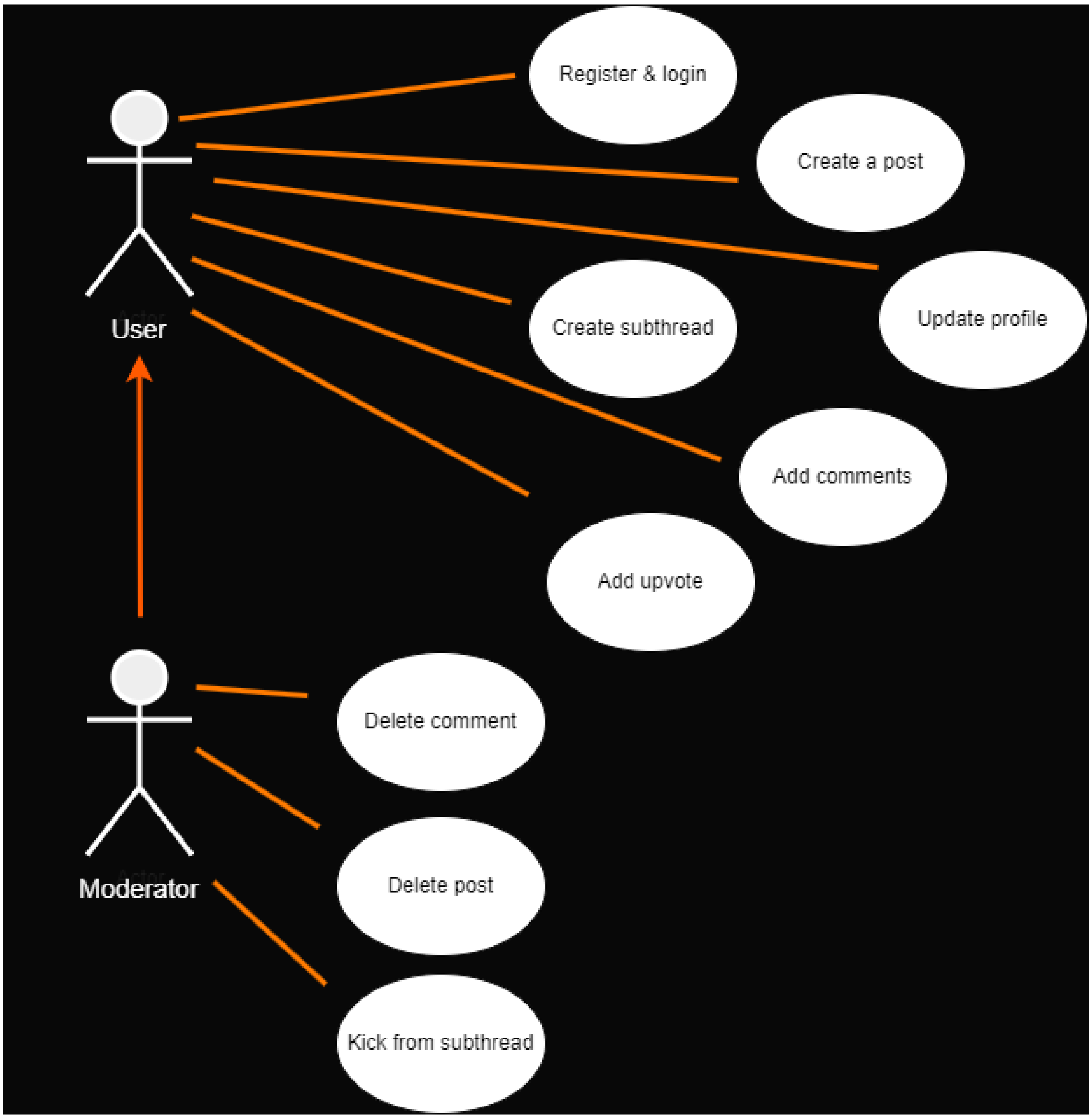


Figure 2: Use cases diagram

2.4 Supplementary Specification

2.4.1 Non-functional Requirements

Performance: Breddit should provide a responsive and fast user experience, ensuring that users can access and interact with content without significant delays. Pages should load quickly, and actions such as posting, commenting, and voting should be processed efficiently. The platform should be able to handle a large number of concurrent users without experiencing performance degradation.

Scalability: As the user base grows, Breddit should be able to scale its infrastructure to accommodate increasing traffic and content volume. The platform should be designed with scalability in mind, allowing for easy horizontal scaling by adding more servers or resources as needed. Additionally, Breddit should support auto-scaling mechanisms to dynamically adjust resources based on demand fluctuations.

Security: Breddit must prioritize the security of user data and prevent unauthorized access to sensitive information. The platform should implement robust authentication mechanisms, such as secure password hashing and encryption, to protect user credentials. Additionally, Breddit should enforce strict access controls to ensure that users can only access content and perform actions appropriate to their roles and permissions. Measures should also be in place to mitigate common security threats, such as cross-site scripting (XSS) and SQL injection.

Availability: Breddit should strive for high availability, ensuring that the platform is accessible to users at all times. This includes minimizing downtime due to maintenance or unexpected outages. Breddit should implement redundancy and failover mechanisms to mitigate the impact of hardware failures or other disruptions. Additionally, the platform should have a comprehensive monitoring system in place to promptly detect and address any issues that may affect availability.

By adhering to these non-functional requirements, Breddit can provide users with a reliable, secure, and seamless experience while using the platform.

2.4.2 Design Constraints

The app has some core components, which offer the functionality I want to deliver. The first and most important component is the authentication page, where users can create an account or log in to their existing one and start enjoying our platform.

The second most important part is, of course, the main page, where you can see all your available options in terms of posts, profile and customization

And last but not least, the supporting component which is our server, where all the validation and logic is placed, to ensure that all users have a seamless experience, and also their private data is secured.

I have used the three-layered architecture, comprised of the Repository layer, Service layer, and Controller layer, representing a well-organized and scalable approach to designing modern software applications.

At the foundation lies the Repository layer, responsible for handling data access and interaction with the database. This layer encapsulates the logic for reading and writing data, ensuring a clean separation between the application's business logic and its data storage concerns.

Above the Repository layer is the Service layer, where the core business logic resides. Services orchestrate and manage the application's functionality, utilizing data retrieved from the repositories. This layer acts as a mediator between the user interface and the data storage, facilitating a modular and maintainable codebase.

Finally, the Controller layer sits at the top, serving as the entry point for user requests and managing the flow of information between the user interface and the underlying services. This architecture fosters a clear separation of concerns, enhances code readability, and promotes flexibility and maintainability in the development process.

Unit testing and integration testing are crucial components of a robust testing strategy in software development, each serving distinct purposes within the software testing life cycle. So, I have tested our components using Mockito API, which is basically another proxy between the mocked and the real implementation. Using the three-layered architecture really came in handy, as we were able to easily test in isolation each layer and how it communicates with the one above, and the one below.

2.5 Applied techniques

Throughout the development cycles, the need for for structure and well-written, highly available and reusable code became a necessity. For that, I have decided to use some conventional code design patterns to ensure that everything is stable and built with quality.

The most used patterns include the Singleton, Repository, Chain of Responsibility, Controller, and Builder. I have also aimed at simplifying the development process, designing abstract high-level classes that help throughout the development cycle, such as a custom response template for our REST Api, unit testing methods to easily assert wanted data and custom hooks for fetching the responses.

As more of the applied techniques, we can add:

- State management solution like Zustand to efficiently manage the application's state, especially when dealing with global data such as user authentication.
- Responsive design that adapts to various screen sizes and devices, utilizing media queries and responsive design principles to optimize the user interface for both desktop and mobile users.
- Lazy loading and code splitting to optimize the application's performance, loading components and resources asynchronously to improve initial load times and reduce the overall bundle size.
- Effective integration with the backend API (provided by Spring Boot) to fetch and update data utilizing asynchronous programming and error handling to enhance the reliability of API calls.
- React Router for client-side routing to create a smooth and intuitive navigation experience, defining routes for different sections of the application, such as threads, search, and user profiles.
- Secure user authentication and authorization mechanisms, considering features like user registration, login, and access control to ensure data privacy and security.
- Modular and reusable UI components leveraging React's component-based architecture.
- RESTful API to expose endpoints for frontend interactions, following the best practices for RESTful design, including meaningful resource naming, proper HTTP methods usage, and consistent error handling.
- Spring Security for robust authentication and authorization. Configure security settings to protect endpoints, handle user roles, and secure sensitive operations.
- Spring Data JPA for simplified and efficient interaction with the PostgreSQL database, defining JPA entities to represent database tables and use repositories for CRUD operations.
- Following database normalization principles to organize data and reduce redundancy, designing a schema that reflects the relationships between entities in the application, such as users, playlists, and tracks.
- Implementing data encryption mechanisms, especially for sensitive user information, such as password encryption.
- JWT Tokens for full control in our authorization system.

By applying these techniques, I have built an app that is not only feature-rich but also performs efficiently, ensures data security, and provides a seamless user experience, remembering to continuously test, iterate, and optimize throughout the development process.

2.6 Code snippets

2.6.1 Repository pattern

```
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    @Query(value = "select u from User u where u.username=:username")
    Optional<User> findByUsername(final String username);

    @Query(value = "select u from User u where u.firstName=:firstName")
    Optional<User> findByFirstName(final String firstName);

    @Query(value = "select u from User u where u.lastName=:lastName")
    Optional<User> findByLastName(final String lastName);

    @Query(value = "select u from User u where u.email=:email")
    Optional<User> findByEmail(final String email);

    @Query(value = "select u from User u where u.country=:country")
    List<User> findAllByCountry(final String country);

    @Query(value = "select u from User u WHERE u.username like %:pattern%")
    List<User> findByUsernameContaining(final String pattern);

    @Query(value = "select u from User u where u.userId in :ids")
    List<User> findMultipleById(final List<Integer> ids);

    @Transactional
    @Query(value = "delete from Users u where u.user_id=:id returning *",
     ↪  nativeQuery = true)
    Optional<User> deleteUserById(final Integer id);
}
```

2.6.2 Global exception handling

```
@RestControllerAdvice
@ControllerAdvice
public class ApiExceptionHandler extends ConcreteMessageController {

    @ExceptionHandler(value = ApiRequestException.class)
    public Response handleApiRequestException(final ApiRequestException
     ↪  exception) {
        return failureResponse(exception.getMessage(), exception.getStatus());
    }

    @ExceptionHandler(value = Exception.class)
    public Response handleException(final Exception exception) {
        return failureResponse(exception.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

2.6.3 Spring Security Configuration

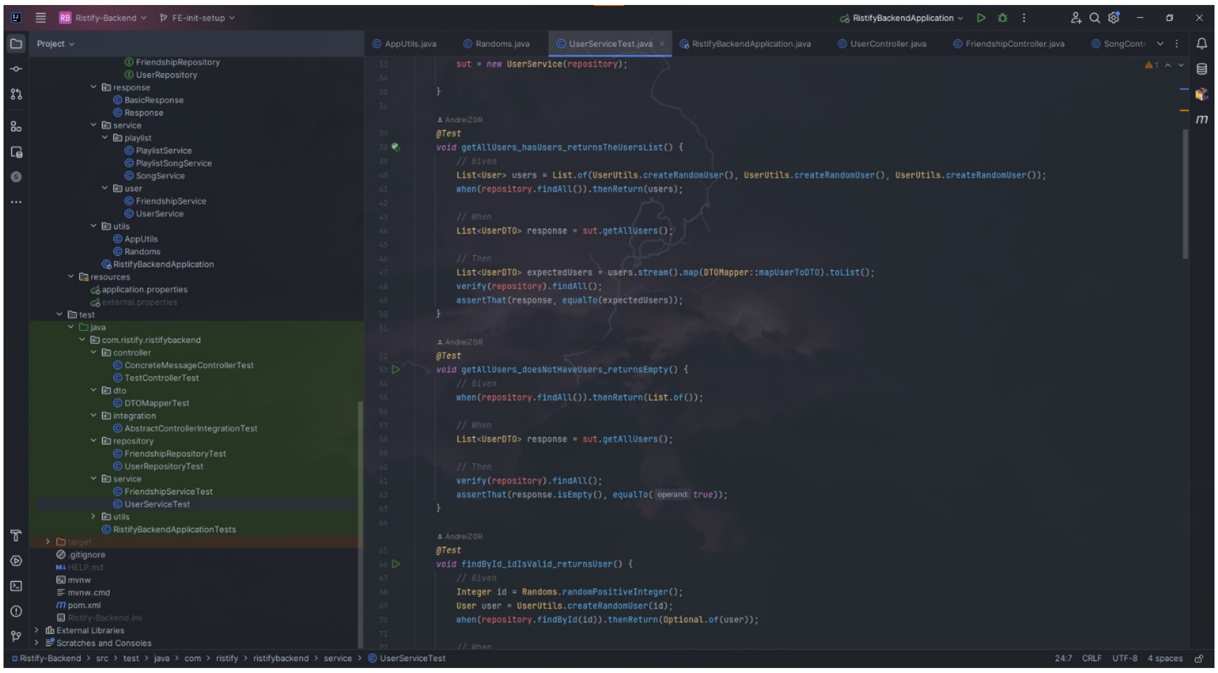
```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private final JwtAuthenticationFilter jwtAuthFilter;
    private final AuthenticationProvider authenticationProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(final HttpSecurity
    ↪   httpSecurity) throws Exception {
        httpSecurity
                .cors(AbstractHttpConfigurer::disable)
                .csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(authorizeCustomizer ->
                ↪   authorizeCustomizer
                        .requestMatchers(AppUtils.WHITE_LIST_URLS).permitAll()
                        .requestMatchers(HttpMethod.DELETE,
                        ↪   "/api/songs/**").hasRole(UserRole.Artist.name())
                        .anyRequest()
                        .authenticated()
                )
                .exceptionHandling(exceptionCustomizer -> exceptionCustomizer
                        .accessDeniedHandler(((request, response,
                        ↪   accessDeniedException) ->

                                ↪   ConcreteMessageController.setServletResponse(response,
                                ↪   AppUtils.ACCESS_DENIED,
                                ↪   HttpServletResponse.SC_FORBIDDEN)))
                )
                .sessionManagement(management ->
                ↪   management.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .authenticationProvider(authenticationProvider)
                .addFilterBefore(jwtAuthFilter,
                ↪   UsernamePasswordAuthenticationFilter.class);

        return httpSecurity.build();
    }
}
```

2.6.4 Response structure

```
@Getter
@EqualsAndHashCode(callSuper = false)
@ToString
```

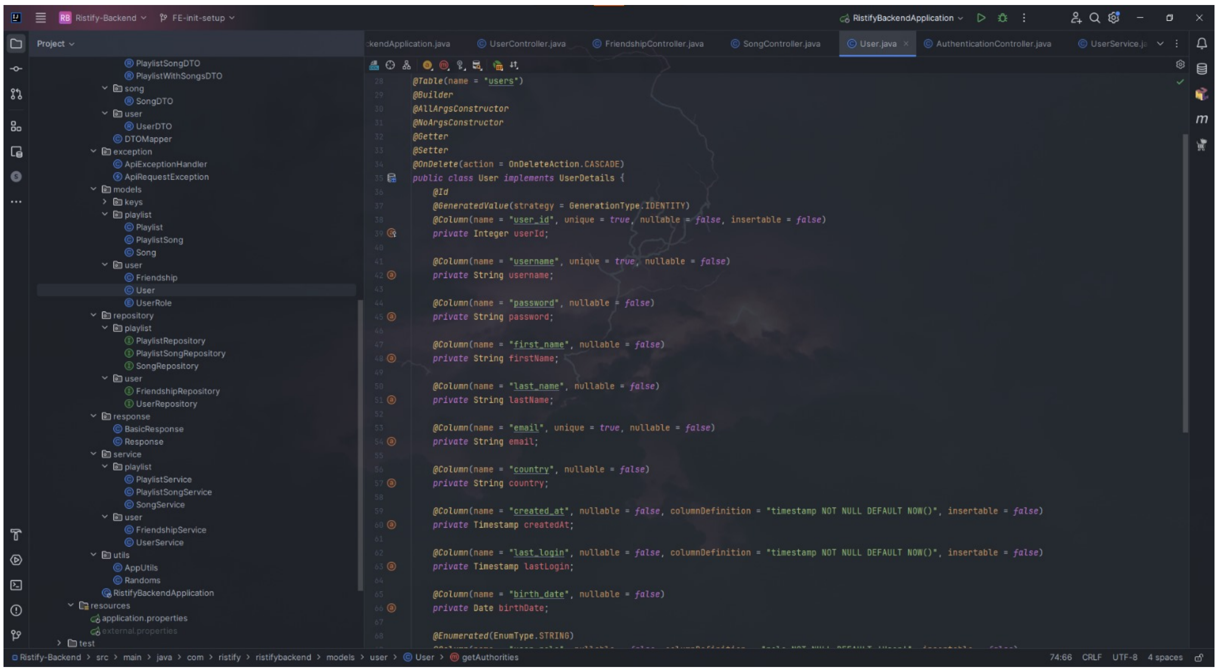
```java
public class Response extends ResponseEntity<Object> implements Serializable {
    private final Map<String, Object> body;
    private final String message;
    private final HttpStatus status;
    private final HttpHeaders headers;

    public Response(final Builder builder) {
        super(builder.body, builder.headers, builder.status);

        this.message = builder.message;
        this.status = builder.status;
        this.headers = builder.headers;
        this.body = builder.body;
    }

    public static class Builder {
        private String message;
        private HttpStatus status;
        private final Map<String, Object> body = new HashMap<>();
        private final HttpHeaders headers = new HttpHeaders();

        public Builder withMessage(final String message) {
            this.message = message;
            return this;
        }

        public Builder withStatus(final HttpStatus status) {
            this.status = status;
            return this;
        }

        public Builder withField(final String fieldName, final Object field) {
            body.put(fieldName, field);
            return this;
        }

        public Builder withMultipleFields(final Map<String, Object> fields) {
            body.putAll(fields);
            return this;
        }

        public Builder withBody(final Object body) {
            this.body.put(AppUtils.PAYLOAD, body);
            return this;
        }

        public Builder withHeader(final String headerName, final String
         ↪  headerValue) {
            headers.set(headerName, headerValue);
            return this;
        }

        public Builder withHeaders(final HttpHeaders headers) {
            this.headers.putAll(headers);
            return this;
        }

        public Response build() {
            body.put(AppUtils.MESSAGE, message);
            body.put(AppUtils.STATUS, status);
            headers.set(HttpHeaders.CONTENT_TYPE, AppUtils.APPLICATION_JSON);

            return new Response(this);
        }
    }
}
```

2.7 Featuring screenshots



```
        sut = new UserService(repository);

    }

    @Test
    void getAllUsers_hasUsers_returnsTheUsersList() {
        // Given
        List<User> users = List.of(UserUtils.createRandomUser(), UserUtils.createRandomUser(), UserUtils.createRandomUser());
        when(repository.findAll()).thenReturn(users);

        // When
        List<UserDTO> response = sut.getAllUsers();

        // Then
        List<UserDTO> expectedUsers = users.stream().map(DTOMapper::mapUserToDTO).toList();
        verify(repository).findAll();
        assertThat(response, equalTo(expectedUsers));
    }

    @Test
    void getAllUsers_doesNotHaveUsers_returnsEmpty() {
        // Given
        when(repository.findAll()).thenReturn(List.of());

        // When
        List<UserDTO> response = sut.getAllUsers();

        // Then
        verify(repository).findAll();
        assertThat(response.isEmpty(), equalTo( operand: true));
    }

    @Test
    void findById_idIsValid_returnsUser() {
        // Given
        Integer id = Randoms.randomPositiveInteger();
        User user = UserUtils.createRandomUser(id);
        when(repository.findById(id)).thenReturn(Optional.of(user));

        // When
```

Figure 3: Server-side testing



```java
@Table(name = "users")
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@OnDelete(action = OnDeleteAction.CASCADE)
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id", unique = true, nullable = false, insertable = false)
    private Integer userId;

    @Column(name = "username", unique = true, nullable = false)
    private String username;

    @Column(name = "password", nullable = false)
    private String password;

    @Column(name = "first_name", nullable = false)
    private String firstName;

    @Column(name = "last_name", nullable = false)
    private String lastName;

    @Column(name = "email", unique = true, nullable = false)
    private String email;

    @Column(name = "country", nullable = false)
    private String country;

    @Column(name = "created_at", nullable = false, columnDefinition = "timestamp NOT NULL DEFAULT NOW()", insertable = false)
    private Timestamp createdAt;

    @Column(name = "last_login", nullable = false, columnDefinition = "timestamp NOT NULL DEFAULT NOW()", insertable = false)
    private Timestamp lastLogin;

    @Column(name = "birth_date", nullable = false)
    private Date birthDate;

    @Enumerated(EnumType.STRING)
```

Figure 4: Spring Data JPA

3 Deliverable 2

3.1 Domain Model

The domain model for Breddit encapsulates the key entities and their relationships within the platform's ecosystem. At the core of the model are entities such as User, Post, Comment, Subthread and Message, which represent the fundamental building blocks of the platform. Users interact with the system by registering accounts, submitting posts, commenting on content, and creating or joining subreddits.

Posts serve as the primary content units, encompassing various formats such as text, links, images, and videos, while comments enable discussions and interactions between users. Subthreads function as thematic communities, allowing users to explore and engage with content tailored to their interests.

The domain model reflects the interconnected nature of Breddit, facilitating seamless navigation and interaction within the platform.

3.2 Architectural Design

3.2.1 Conceptual Architecture

The architectural design is structured around a modern and scalable architecture, leveraging a RESTful API approach to deliver a robust and responsive user experience. The system is built upon a monolithic architecture, where different components are integrated into a single cohesive unit. The back-end of Breddit consists of a set of RESTful APIs responsible for handling user authentication, content management, and other core functionalities.

These APIs expose well-defined endpoints, enabling efficient communication between the client-side application and the server. The front-end client is developed using responsive web design principles to ensure cross-device compatibility.

Additionally, Breddit incorporates caching mechanisms to optimize performance and reduce latency. The architectural design emphasizes scalability, reliability, and maintainability, enabling it to accommodate a growing user base and evolving requirements while maintaining high availability and responsiveness.

3.2.2 Package Design

In Breddit, the package design adheres to a three-layered architecture consisting of repository, service, and controller layers, facilitating modularity, maintainability, and separation of concerns.

At the lowest layer, the repository layer encapsulates data access logic and interacts directly with the underlying database, abstracting away the details of data storage and retrieval. Repositories are responsible for querying and persisting data entities, ensuring data integrity and consistency.

Above the repository layer, the service layer contains business logic and application-specific functionalities. Services orchestrate interactions between repositories and provide a high-level interface for performing operations such as user authentication, content management, and business rule enforcement. They encapsulate complex business logic, promoting reusability and testability.

Finally, at the topmost layer, the controller layer serves as the entry point for incoming requests from clients. Controllers handle request processing, route requests to the appropriate service methods, and orchestrate the flow of data between the client and the application. This layered architecture promotes loose coupling between components, enabling easier maintenance, scalability, and evolution of the platform.

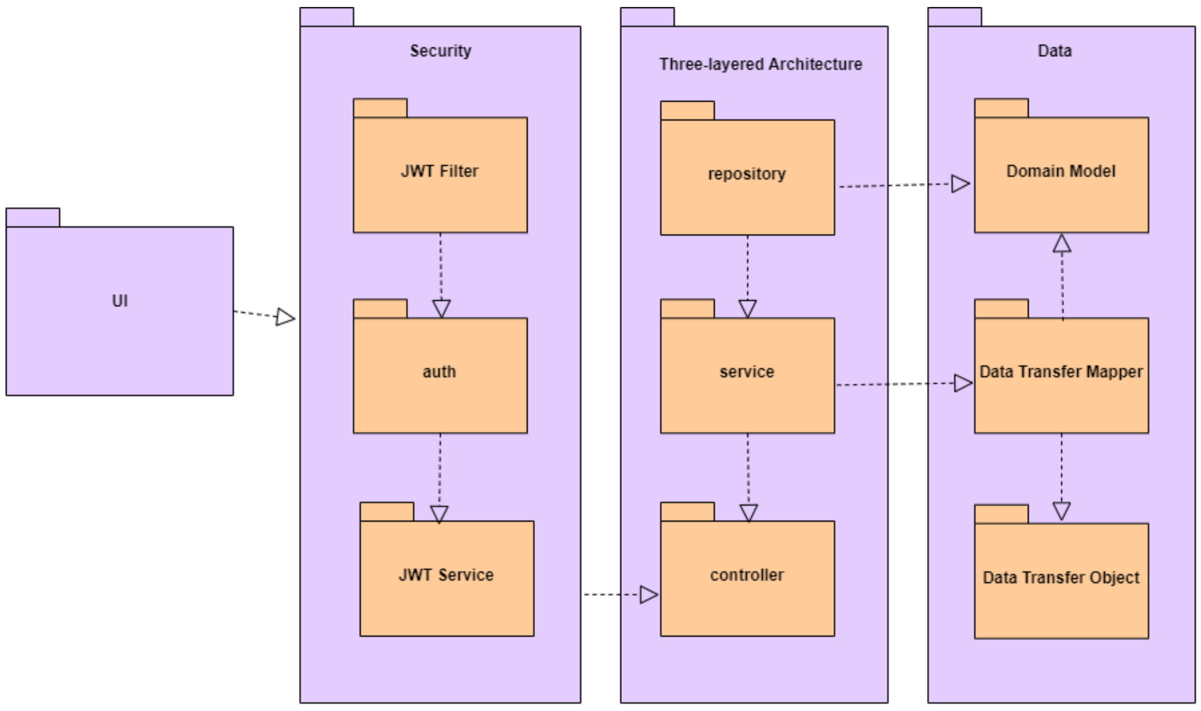


Figure 5: Package Diagram

3.2.3 Component and Deployment Diagram

In Breddit, the component architecture is designed to promote modularity, reusability, and scalability. The system is composed of various components, each responsible for specific functionalities within the platform. Components are encapsulated units of code that can be independently developed, tested, and deployed.

Key components in Breddit include the user authentication module, content management module, notification module, and user interface components. These components interact with each other through well-defined interfaces, enabling loose coupling and facilitating easier maintenance and evolution of the platform.

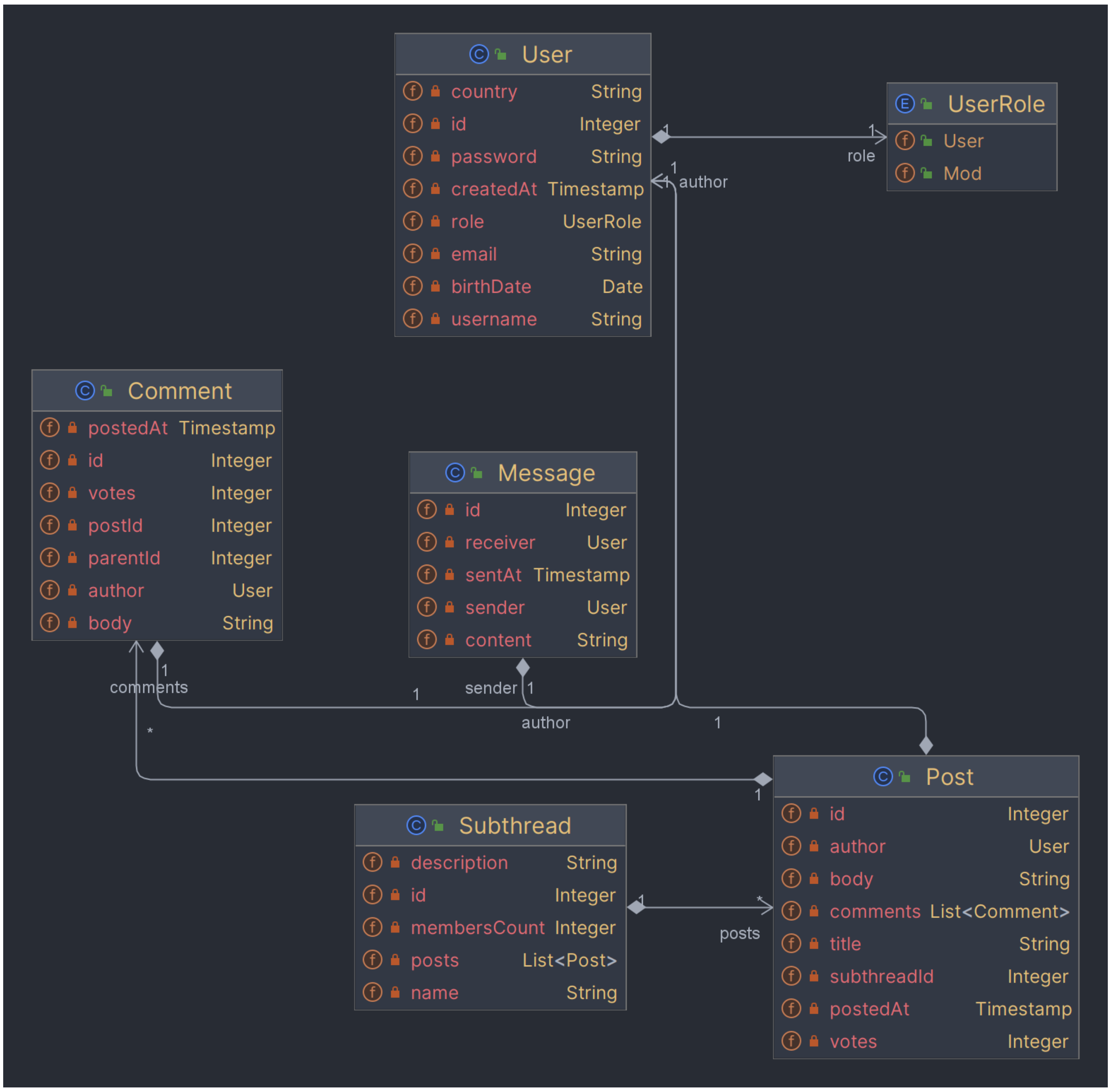


Figure 6: Model Layer Diagram

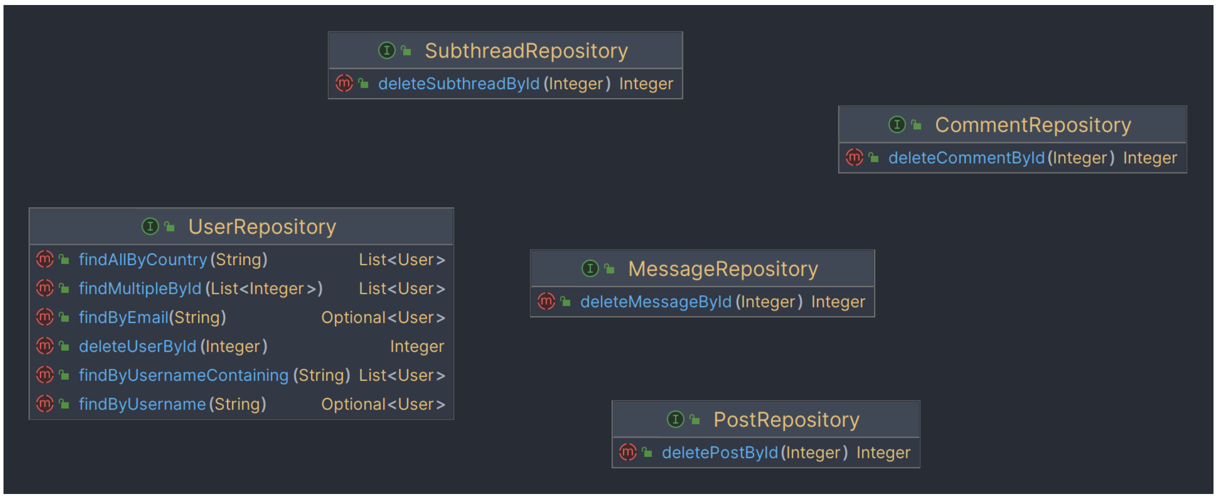

SubthreadRepository
deleteSubthreadById(Integer) Integer
CommentRepository
deleteCommentById(Integer) Integer
UserRepository
findAllByCountry(String) List<User>
findMultipleById(List<Integer>) List<User>
findByEmail(String) Optional<User>
deleteUserById(Integer) Integer
findByUsernameContaining(String) List<User>
findByUsername(String) Optional<User>
MessageRepository
deleteMessageById(Integer) Integer
PostRepository
deletePostById(Integer) Integer

Figure 7: Repository Layer Diagram

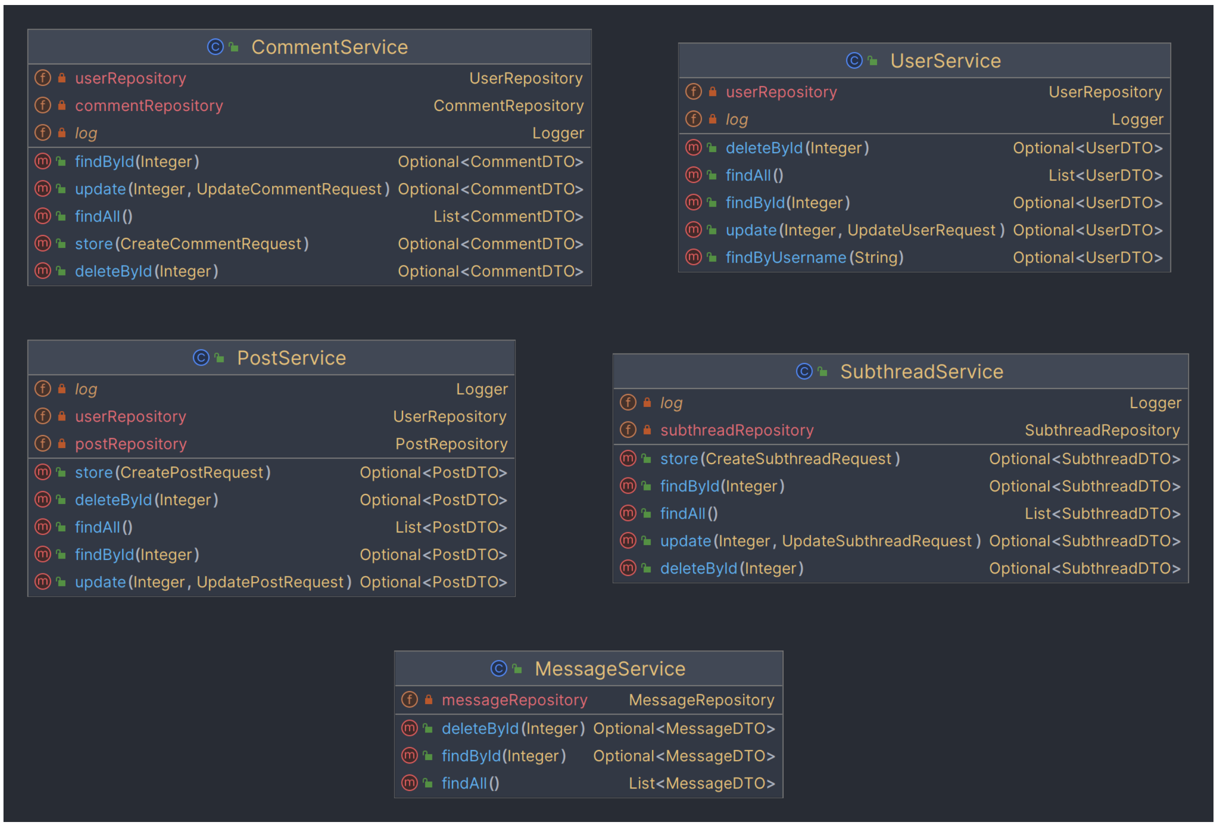

CommentService
userRepository UserRepository
commentRepository CommentRepository
log Logger
findById(Integer) Optional<CommentDTO>
update(Integer, UpdateCommentRequest) Optional<CommentDTO>
findAll() List<CommentDTO>
store(CreateCommentRequest) Optional<CommentDTO>
deleteById(Integer) Optional<CommentDTO>
UserService
userRepository UserRepository
log Logger
deleteById(Integer) Optional<UserDTO>
findAll() List<UserDTO>
findById(Integer) Optional<UserDTO>
update(Integer, UpdateUserRequest) Optional<UserDTO>
findByUsername(String) Optional<UserDTO>
PostService
log Logger
userRepository UserRepository
postRepository PostRepository
store(CreatePostRequest) Optional<PostDTO>
deleteById(Integer) Optional<PostDTO>
findAll() List<PostDTO>
findById(Integer) Optional<PostDTO>
update(Integer, UpdatePostRequest) Optional<PostDTO>
SubthreadService
log Logger
subthreadRepository SubthreadRepository
store(CreateSubthreadRequest) Optional<SubthreadDTO>
findById(Integer) Optional<SubthreadDTO>
findAll() List<SubthreadDTO>
update(Integer, UpdateSubthreadRequest) Optional<SubthreadDTO>
deleteById(Integer) Optional<SubthreadDTO>
MessageService
messageRepository MessageRepository
deleteById(Integer) Optional<MessageDTO>
findById(Integer) Optional<MessageDTO>
findAll() List<MessageDTO>

Figure 8: Service Layer Diagram

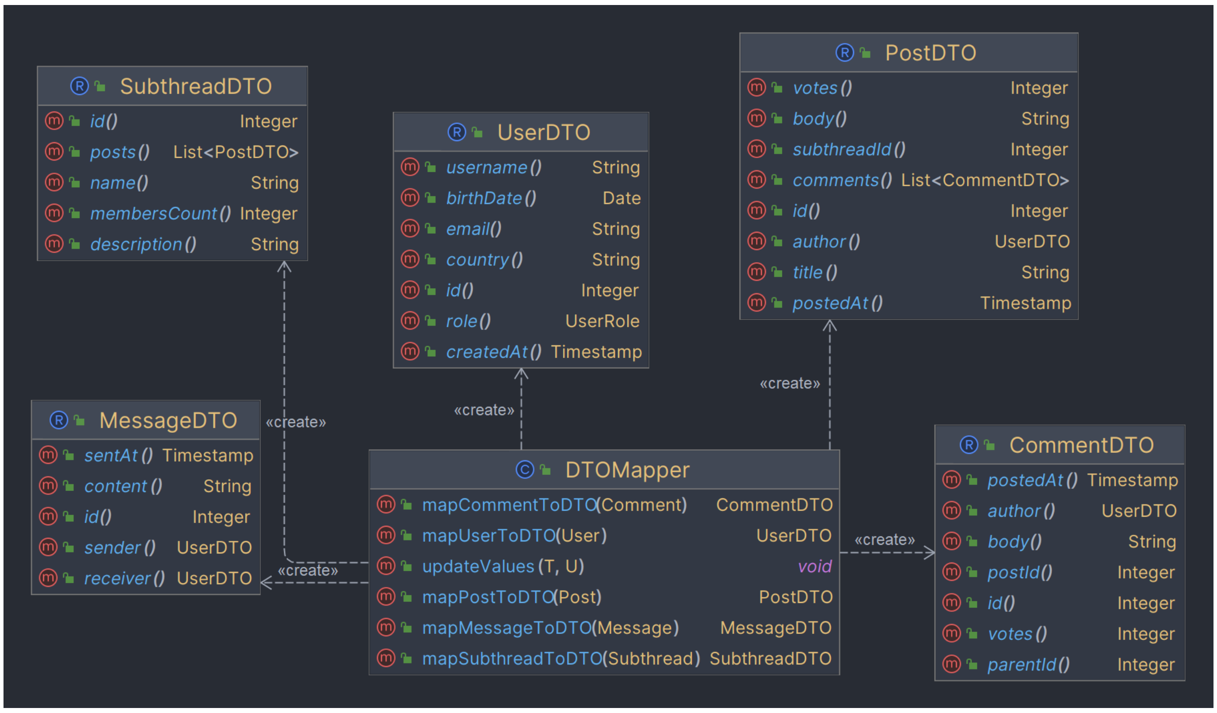


Figure 9: DTO Layer Diagram

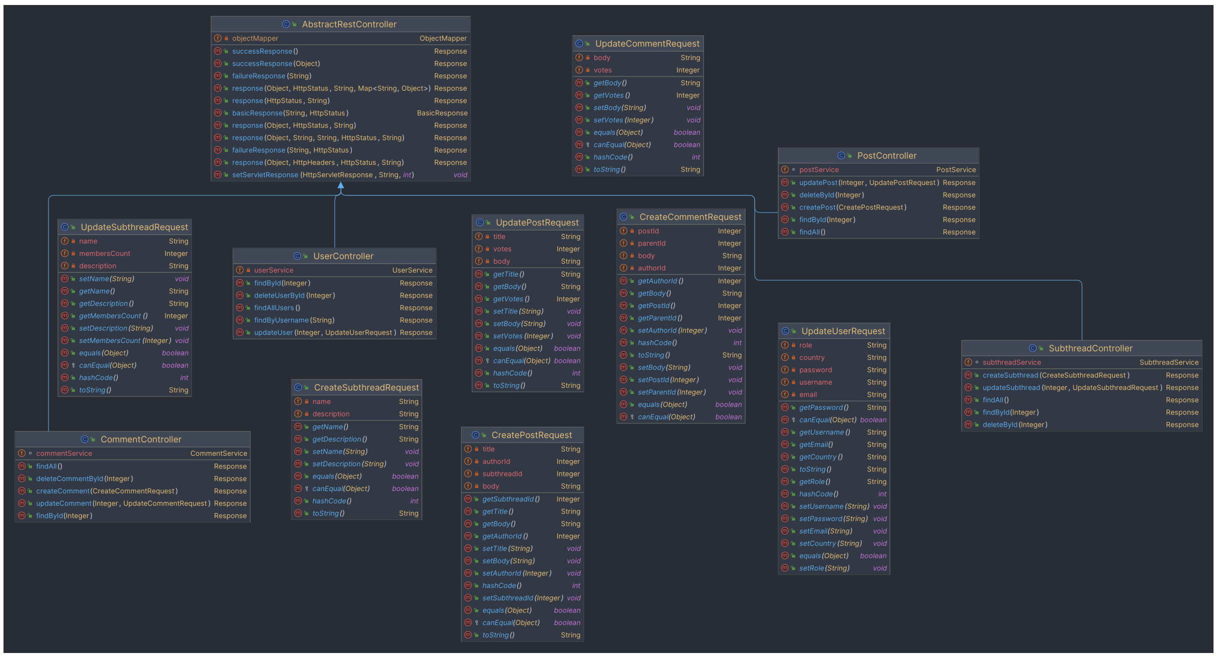


Figure 10: Controller Layer Diagram

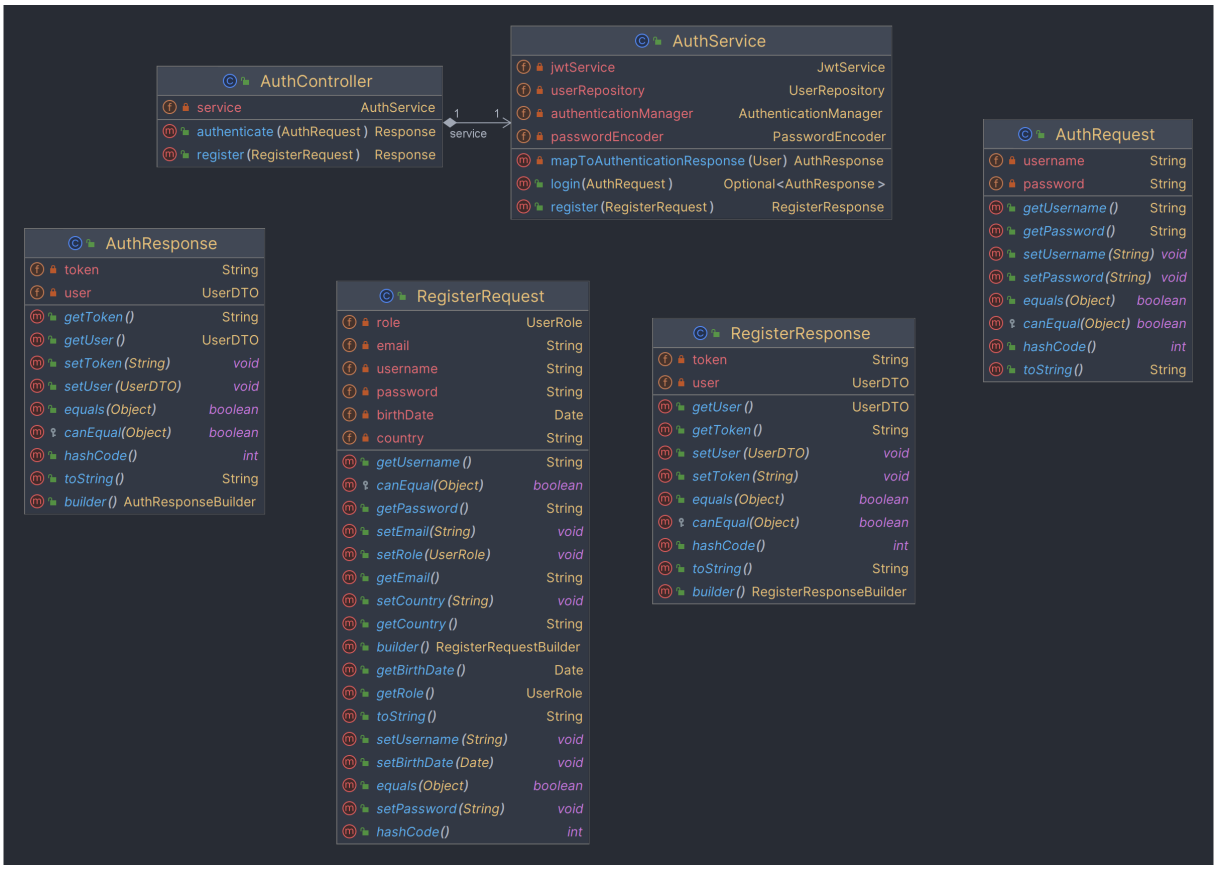


Figure 11: Auth Diagram

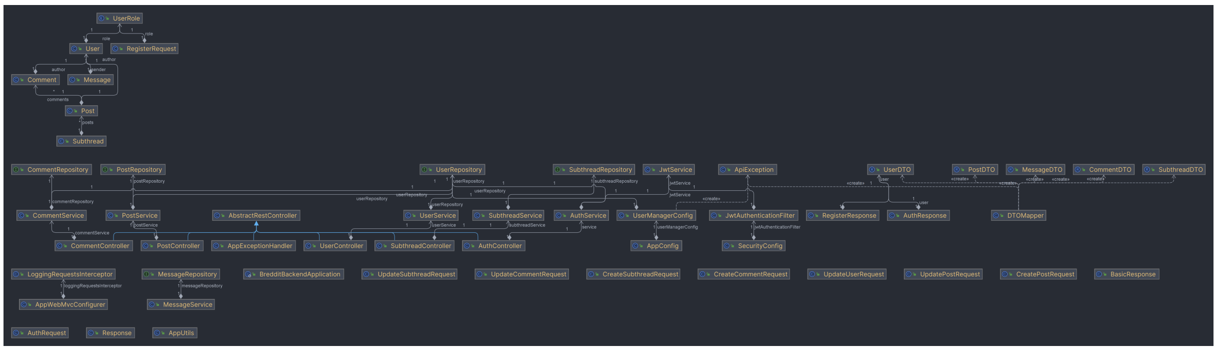


Figure 12: Overall REST Diagram

4 Deliverable 3

4.1 Design Model

The design model encompasses various architectural and structural decisions made during the design phase. At the core of our design model is the architectural pattern chosen to govern the system's overall structure. I've opted for a layered architecture, which organizes the system into distinct layers, each responsible for a specific set of functionalities. This architectural pattern promotes modularity, separation of concerns, and ease of maintenance, aligning well with the requirements and goals of the project.

In terms of component design, I've decomposed the system into smaller, manageable components, each with well-defined responsibilities and interfaces. These components interact with each other to fulfill the system's functionalities. The component design emphasizes cohesion and low coupling, facilitating easier development, testing, and maintenance of the system. Component diagrams provide a visual representation of the relationships between components and their interactions.

At the class level, I've designed a coherent class structure that encapsulates data and behavior within the system. Classes are organized into meaningful hierarchies, with clear relationships and dependencies. Our class design emphasizes encapsulation, inheritance, and polymorphism, enabling flexibility and extensibility in the system's implementation. Class diagrams serve as a visual aid to illustrate the class structure and relationships within the system.

For the database design, it was a carefully crafted schema that efficiently stores and manages the system's data. The schema includes tables, columns, relationships, and constraints, designed to optimize data organization and retrieval. Entity-relationship diagrams (ERDs) provide a graphical representation of the database schema, illustrating the relationships between entities and their attributes.

In terms of user interface design, the focus was on creating intuitive and user-friendly interfaces that enhance user experience. The interface design emphasizes usability, accessibility, and responsiveness, ensuring that users can easily navigate and interact with the system. Wireframes and mockups help visualize the layout, navigation, and interaction patterns of the user interface.

Finally, security design considerations are integrated into the system to safeguard against potential risks and vulnerabilities. Authentication, authorization, encryption, and data protection measures are implemented to ensure the security and integrity of the system's data and resources. Security mechanisms are seamlessly integrated into the system's architecture, mitigating potential security threats and vulnerabilities.

4.1.1 Dynamic Behaviour

In the design model, dynamic behavior refers to how the components of the system interact and behave at runtime to fulfill various user scenarios and system functionalities. At the core of dynamic behavior is the flow of control and data between different components of the system. Within our layered architecture, dynamic behavior is facilitated by the interactions between the layers, with each layer responsible for specific aspects of request processing and data manipulation.

For instance, in a typical request-response cycle, the presentation layer interacts with the user interface, collecting user input and forwarding it to the business logic layer. In turn, the business logic layer processes the request, orchestrating interactions with the data access layer to retrieve or modify data as necessary before presenting the response back to the user through the user interface. Additionally, within each layer, dynamic behavior is governed by the interactions between individual components.

For example, in the business logic layer, service components orchestrate interactions between domain entities and data access components to perform business operations. These interactions are governed by business rules and logic, ensuring that the system behaves correctly and consistently in response to user requests.

4.1.2 Class Diagram

In the context of the Design model, the class diagram serves as a visual representation of the structure and relationships between classes in the system. It depicts the various classes, interfaces, attributes, and methods within the system, along with their associations, dependencies, and inheritance relationships.

The class diagram provides a high-level overview of the system's class structure, facilitating communication and understanding among stakeholders, including developers, architects, and project managers. It serves as a blueprint for the implementation phase, guiding developers in writing code and designing software components that align with the system's design.

Additionally, the class diagram helps identify potential design flaws, inconsistencies, or ambiguities early in the development process, allowing for timely corrections and refinements. Overall, the class diagram is an essential artifact within the Design model, providing valuable insights into the organization and structure of the system's classes and their interactions.

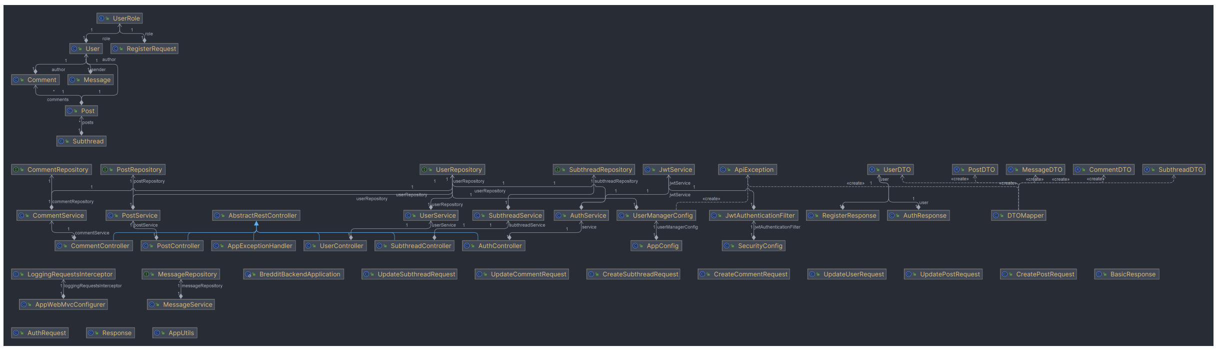


Figure 13: Class Diagram

4.2 Data Model

In the Design model, the data model defines the structure, organization, and relationships of the data within the system. It represents the logical view of the data and how it is stored, accessed, and manipulated by the system's components. The data model typically includes entities, attributes, relationships, and constraints that govern the data's behavior and integrity.

Entities represent the real-world objects or concepts that the system manages, such as users, posts, comments, and subthreads. Attributes define the properties or characteristics of entities, such as username, email address, and password for a user entity. Relationships describe the associations between entities, such as the relationship between a user and their posts or comments.

The data model may also include constraints to enforce data integrity and consistency, such as primary keys, foreign keys, unique constraints, and check constraints. These constraints ensure that the data remains accurate, valid, and consistent throughout the system.

In Breddit, the data model might consist of entities such as User, Post, Comment, Subreddit, Message, and others, along with their respective attributes and relationships. For example, the User entity have attributes such as username, email, and password, while the Post entity have attributes like title, content, and timestamp. Relationships between entities, such as a user's posts or comments, are represented through associations.

Overall, the data model serves as a foundation for designing the system's database schema and informs the implementation of data access and manipulation logic within the system. It ensures that the system effectively manages and processes data to support its functionalities and meet the requirements of its users.

4.3 System Testing

System testing is a crucial phase in the software development lifecycle where the entire integrated system is evaluated to ensure that it meets specified requirements and functions correctly as a whole. This testing phase focuses on verifying the behavior of the system as a complete and integrated entity, including all its components, modules, and interfaces.

The scope of system testing encompasses testing the system's features, functionality, performance, reliability, security, and usability to validate its compliance with functional and non-functional requirements specified during the requirements analysis phase. Various types of testing are performed during the system testing phase, including functional testing, performance testing, security testing, usability testing, and compatibility testing.

System testing begins with the development of a comprehensive test plan outlining the testing objectives, scope, approach, resources, schedule, and responsibilities. The test plan serves as a roadmap for conducting systematic and organized testing activities. Test cases are then executed against the system to validate its behavior and functionality. Test cases cover various scenarios, inputs, and usage patterns to ensure thorough coverage of the system's capabilities.

During test execution, defects or issues discovered in the system are logged, tracked, and managed using a defect tracking system. Defects are prioritized, assigned to appropriate team members for resolution, and retested once fixed to ensure closure. Regression testing is performed to ensure that existing functionalities remain unaffected by changes made to the system.

Throughout the system testing phase, documentation is updated to reflect test results, defect status, and any deviations from expected behavior.

4.4 Future Improvements

Looking ahead, there are several areas where Breddit can undergo future improvements to enhance its functionality, user experience, and overall value proposition. One potential avenue for improvement is the expansion of social features, such as enhancing the user profile system to allow for more customization and personalization options. Additionally, implementing advanced recommendation algorithms could help surface relevant content to users based on their interests and past interactions, improving content discovery and engagement.

Furthermore, incorporating real-time messaging and collaboration features could foster more dynamic and interactive community interactions among Breddit users. Another area for improvement is scalability and performance optimization, ensuring that Breddit can seamlessly handle increased user traffic and content volume as its user base continues to grow.

Additionally, continuous refinement of the platform's security measures and privacy controls will be essential to safeguard user data and maintain trust and confidence among Breddit's community members.

Overall, by prioritizing these future improvements, Breddit can continue to evolve and innovate, providing a compelling and engaging platform for users to discover, share, and connect around their interests.

4.5 Conclusion

In conclusion, Breddit represents a significant step forward in the realm of social content aggregation and community engagement. Through meticulous design, robust development, and rigorous testing, Breddit has evolved into a platform that empowers users to explore, share, and connect with like-minded individuals across a wide array of topics and interests. The journey from conceptualization to realization has been marked by dedication, collaboration, and a relentless pursuit of excellence. As I reflect on the achievements and challenges encountered throughout the development process, it becomes evident that Breddit is more than just a product — it is a testament to the collective efforts and aspirations of a passionate developer striving to make a positive impact in the digital landscape.

Looking ahead, the future holds endless possibilities for Breddit. With a solid foundation in place, the platform is poised for continuous growth, innovation, and refinement. By listening to user feedback, staying abreast of emerging trends, and embracing technological advancements, Breddit can evolve into a dynamic and indispensable hub for online discourse, collaboration, and community building.

5 Bibliography

- Spring Websockets
- Redux Toolkit
- Spring boot Async
- Mockito for Spring