UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Tinder Clone

Name: MAXIM FRANCESCO
Group: 30234

Table of Contents

Deliverable 1 ..... ***3***

Project Specification **3**

Functional Requirements **3**

Use Case Model **3**

Use Cases Identification: 3

UML Use Case Diagrams 3

Supplementary Specification **4**

Non-functional Requirements 4

Design Constraints 5

Glossary **5**

Deliverable 2 ..... ***5***

Domain Model **6**

Architectural Design **7**

Conceptual Architecture 7

Package Design 9

Component and Deployment Diagram 10

Deliverable 3 ..... ***11***

Design Model **11**

Dynamic Behavior 11

Class Diagram 11

Data Model **11**

System Testing ***11***

Future Improvements ***11***

Conclusion ***11***

Bibliography ***11***

Deliverable 1

Project Specification

This project is a clone of the Tinder application, built using Angular for the frontend and Java Spring Boot for the backend. It allows users to register, create and manage their profiles, browse other profiles (by swiping “like” or “dislike”), form mutual matches, and communicate via text messages. Currently, all modules are implemented except for configuration, DTOs, centralized exception handling, WebSocket communication, and security. This document outlines the specification and design for the project.

Functional Requirements

1. **User Registration and Authentication:**
   - Users must register with an email and password.
   - The system should authenticate users via a dedicated endpoint.
2. **Profile Management:**
   - Users can create, view, and update their profiles.
   - A profile includes details such as name, age, gender, bio, location, and photos.
3. **Swiping and Matching:**
   - Users can swipe through other profiles, choosing to “like” or “dislike.”
   - A match is created when two users mutually “like” each other.
4. **Messaging:**
   - Matched users can send text messages to each other in real time or asynchronously.
   - Users can view the conversation history with each match.

Use Case Model

Use Cases Identification:

Use Case 1: User Registration

- **Use Case:** User Registration
- **Level:** User Goal
- **Primary Actor:** New User
- **Main Success Scenario:**
  1. The user accesses the registration page.
  2. The user enters the required data (email, password, etc.).
  3. The system validates the information and creates a new account.
  4. The user receives a confirmation message.
- **Extensions:**
  - If the email is already in use, the system informs the user and requests an alternative email.

Use Case 2: Swiping and Matching

- **Use Case:** Swiping and Matching
- **Level:** User Goal
- **Primary Actor:** Authenticated User
- **Main Success Scenario:**
  1. The user browses through other users’ profiles.

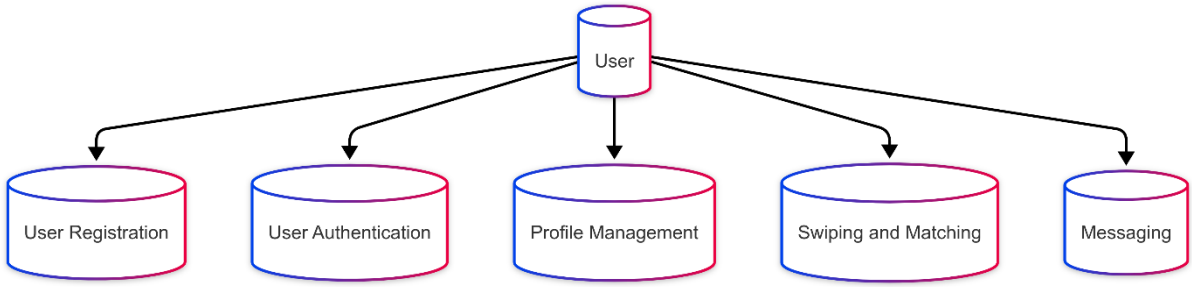
2. The user swipes “like” or “dislike” on each profile.
3. When both users “like” each other, a match is created.

- **Extensions:**
  - If a user swipes “dislike,” that profile is not shown again.

Use Case 3: Sending Messages

- **Use Case:** Sending Messages
- **Level:** User Goal
- **Primary Actor:** User with a Match
- **Main Success Scenario:**
  1. The user selects a match from the conversation list.
  2. The user types a message in the chat interface.
  3. The system saves the message and transmits it to the recipient.
  4. The conversation view is updated in real time (or upon refresh).
- **Extensions:**
  - If the message fails to send (e.g., due to a weak connection), the system notifies the user and allows a retry.

UML Use Case Diagrams



Supplementary Specification

Non-functional Requirements

1. **Performance:**
   - The system must respond to API requests within 2 seconds.
   - **Justification:** Fast response times are critical to ensure a smooth and engaging user experience in a social application.
2. **Scalability:**
   - The architecture must support significant growth in the number of users by using a modular structure with the possibility of transitioning to microservices.
   - **Justification:** As the user base grows, the system must handle increased traffic without performance degradation.
3. **Reliability:**
   - The system should maintain an uptime of 99.9% and handle errors gracefully.
   - **Justification:** Continuous availability is essential for a dating application where user engagement is time-sensitive.
4. **Security:**

- User data (e.g., passwords, personal information) must be encrypted and the system should implement proper authentication and authorization mechanisms (such as JWT).
- **Justification:** Protecting sensitive user data is critical, especially for social applications handling personal information.

Design Constraints

1. **Languages and Frameworks:**
   - The backend must be developed in Java using Spring Boot.
   - The frontend will be implemented using Angular.
2. **Development Process:**
   - Maven or Gradle will be used for dependency management and building the project.
   - Testing will be carried out using JUnit and Mockito on the backend.
3. **Component Usage:**
   - Spring Data JPA will be used for database interactions.
   - Libraries such as Lombok will be used to reduce boilerplate in the model classes.
4. **Other Constraints:**
   - Initial configuration for aspects like DTOs, exception handling, WebSocket, and security will be added in subsequent development phases.

Glossary

1. **User:**

A person using the application, identified by an email and password.

- **Validation:** Must be a valid email and a password of at least 8 characters.

2. **Profile:**

A set of personal information about a user (name, age, gender, bio, location, photos).

3. **Like:**

The action a user takes to express interest in another user's profile.

- **Values:** true (like) or false (dislike).

4. **Match:**

A connection created when two users mutually "like" each other.

- **Rule:** A match is only generated if both users have liked each other.

5. **Message:**

A text communication between two users who have a match.

- **Attributes:** Message content, sender, receiver, and timestamp.

6. **Swiping:**

The mechanism by which users navigate through profiles, choosing to like or dislike.

7. **WebSocket:**
A protocol enabling real-time, two-way communication between client and server (to be implemented in later stages).

Deliverable 2

Domain Model

The domain model defines the core entities and relationships within the Tinder Clone application. It captures the main business logic and represents the structure of the application using conceptual classes.

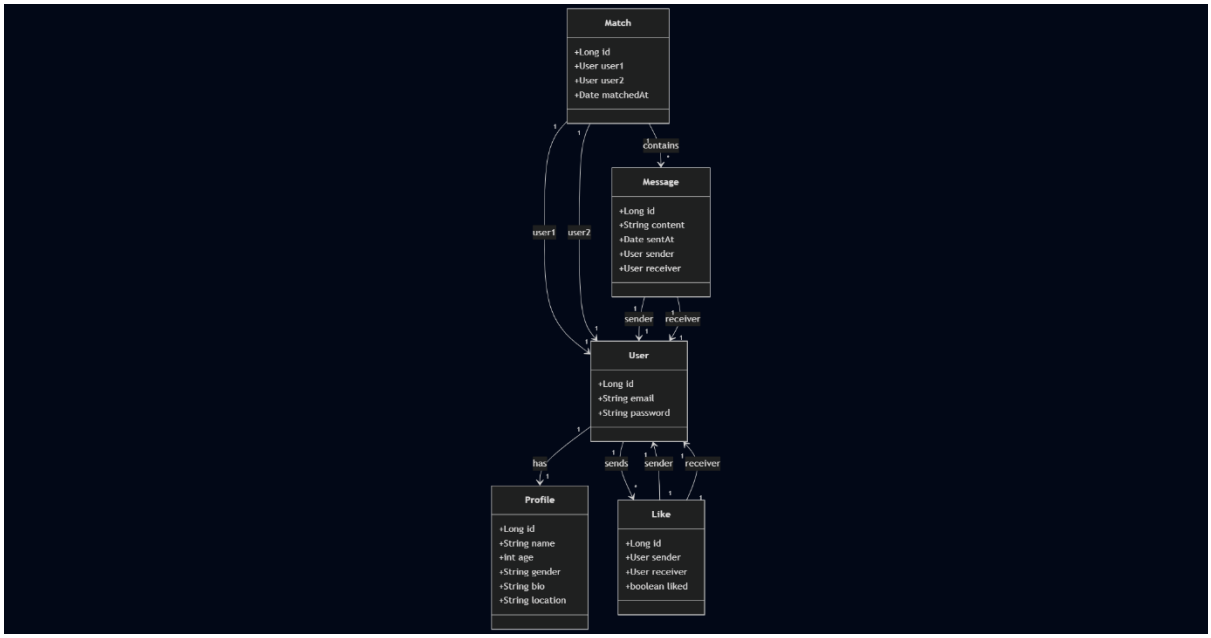
Entities:

- **User**: Represents a system account. Each user has login credentials (email, password) and is linked to one profile.
- **Profile**: Contains personal and public information about the user, such as name, age, gender, location, bio, and profile photos.
- **Like**: Represents an action performed by a user to express interest in another user's profile. Each like is directional (from sender to receiver) and can be either positive (like) or negative (dislike).
- **Match**: Represents a successful mutual like between two users. A match is created only when both users have liked each other.
- **Message**: Represents a chat message exchanged between two matched users. Messages are stored per match and track sender, receiver, content, and timestamp.

Relationships:

- A User has exactly one Profile.
- A User can send many Likes, and receive many.
- A Like is linked to two users: sender and receiver.
- A Match connects two users who liked each other.
- A Match can contain many Messages.
- A Message is sent by one user to another within a specific match.

This conceptual model serves as the foundation for the application's database structure and object-oriented implementation in the backend.



Architectural Design

Conceptual Architecture

The Tinder Clone application follows a **Layered Architecture** (also known as N-Tier Architecture), a well-established architectural pattern for building scalable and maintainable enterprise applications. The system is divided into separate layers, each with a distinct responsibility:

Layers:

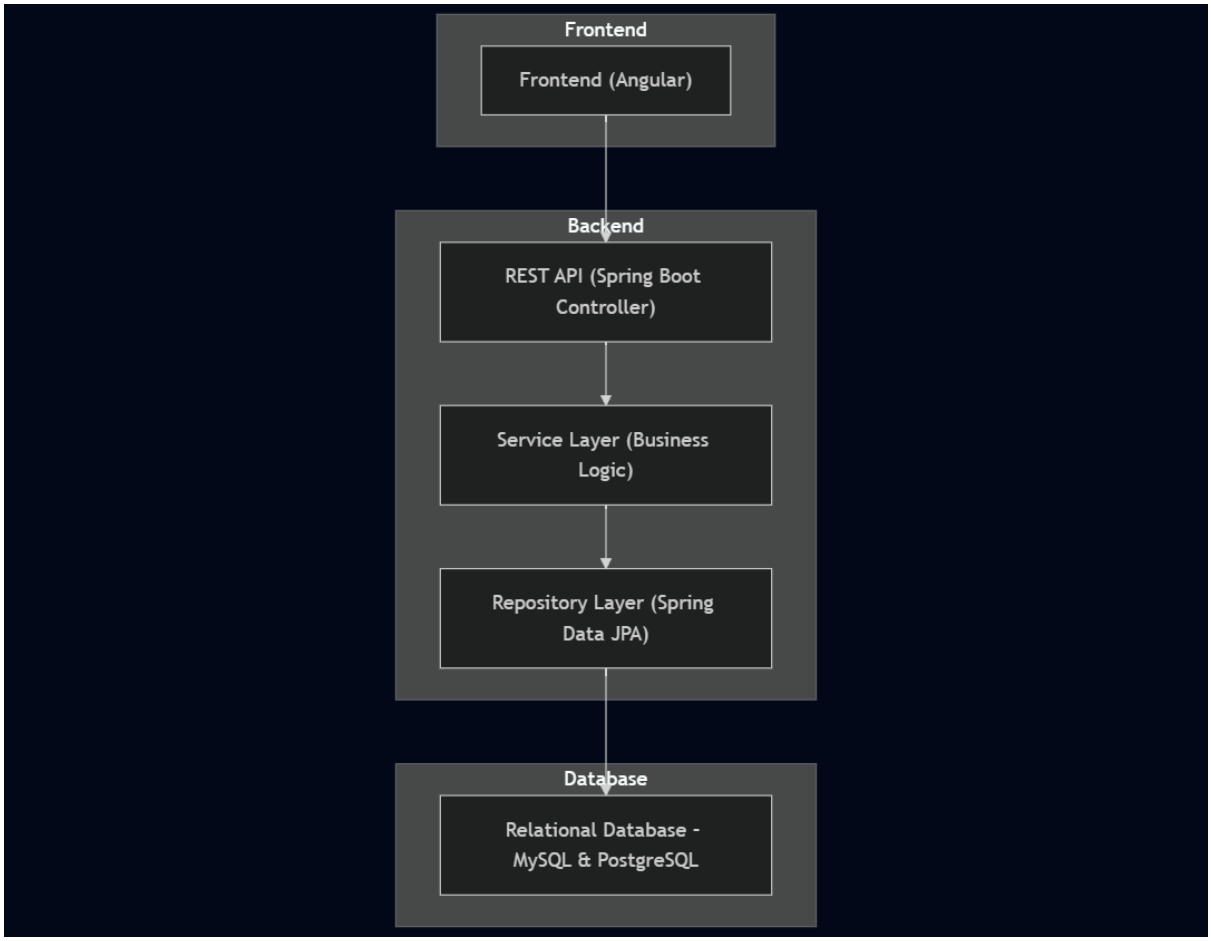
- **Presentation Layer (Frontend):**
  Built with **Angular**, this layer handles the user interface and user experience. It communicates with the backend via HTTP requests to RESTful APIs and is responsible for displaying data, managing routing, and collecting user input.
- **Application Layer (Controller):**
  Implemented in **Spring Boot**, this layer exposes REST endpoints to the frontend. It processes client requests, delegates tasks to the appropriate services, and returns structured responses.
- **Business Logic Layer (Service):**
  Contains the core application logic. Each service encapsulates the business rules and orchestrates interactions between different components. This separation ensures that logic is testable and reusable.
- **Persistence Layer (Repository):**
  Uses **Spring Data JPA** to abstract access to the relational database. Repositories handle all CRUD operations and interact with domain entities such as User, Profile, Like, Match, and Message.

- **Database Layer:**
  A relational database (such as MySQL or PostgreSQL) is used to store persistent data. The schema reflects the domain model, supporting relationships like one-to-one (User–Profile), one-to-many (Match–Messages), and many-to-one (Likes between users).

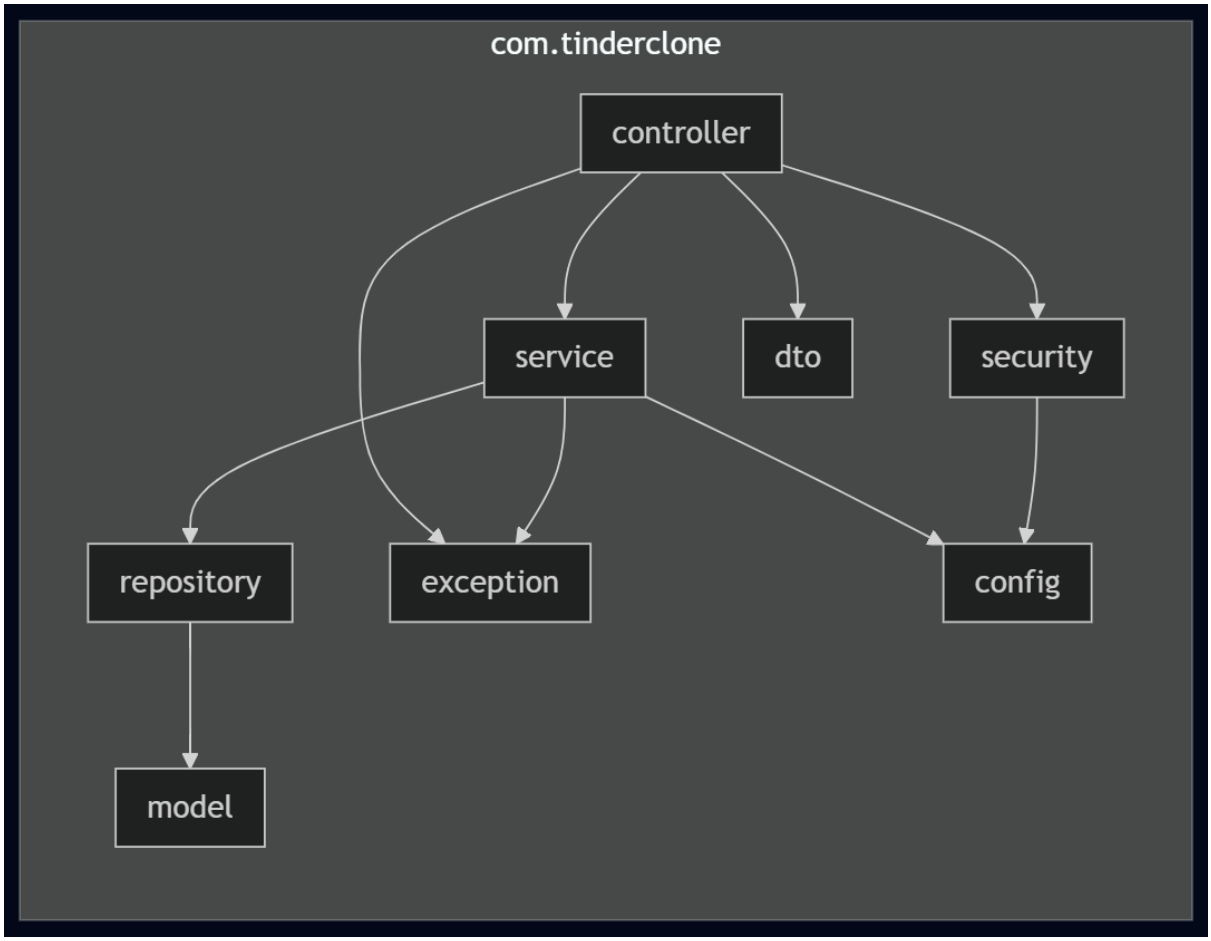
Motivation for Choosing Layered Architecture:

Layered architecture was chosen for its **simplicity, clear separation of concerns, and maintainability**. Each layer is loosely coupled and can be tested, updated, or scaled independently. This structure also aligns well with Spring Boot's component-based architecture and Angular's module system, allowing a clean fullstack implementation.

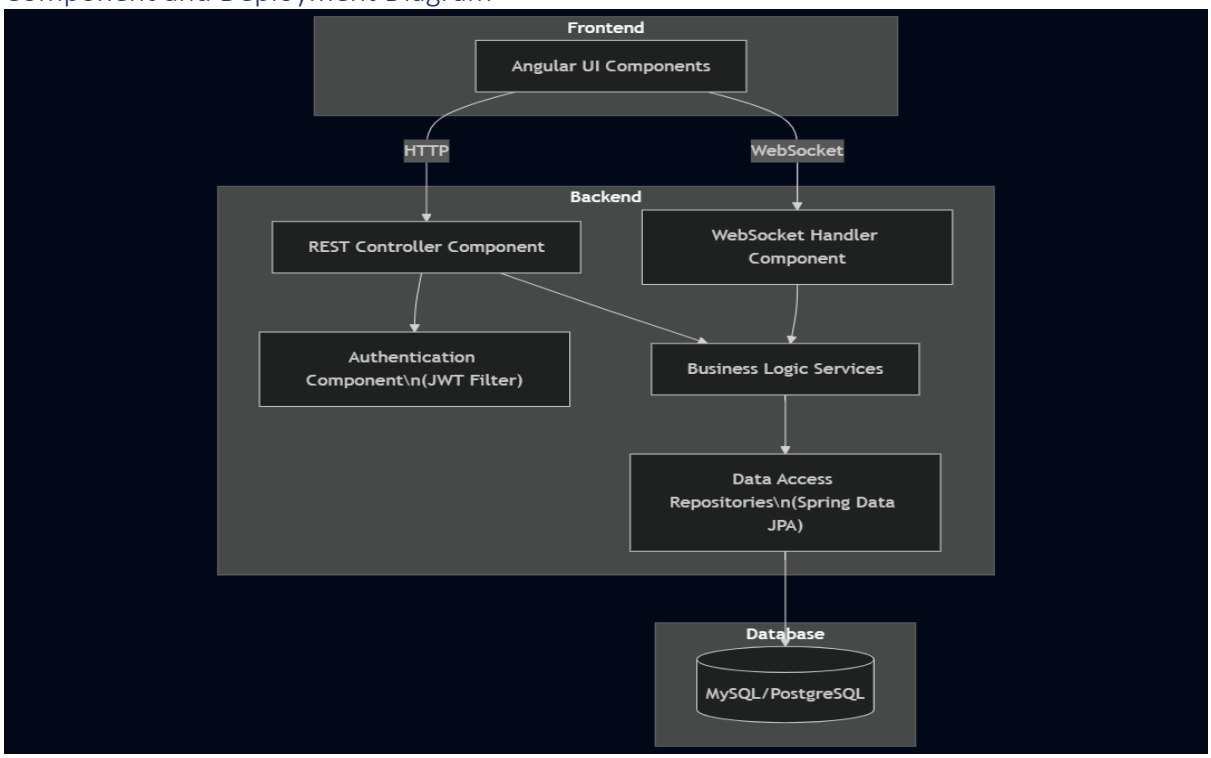
Additionally, the architecture supports **future extensibility**—such as adding security (Spring Security), real-time communication (WebSockets), or breaking down into microservices as the project evolves.

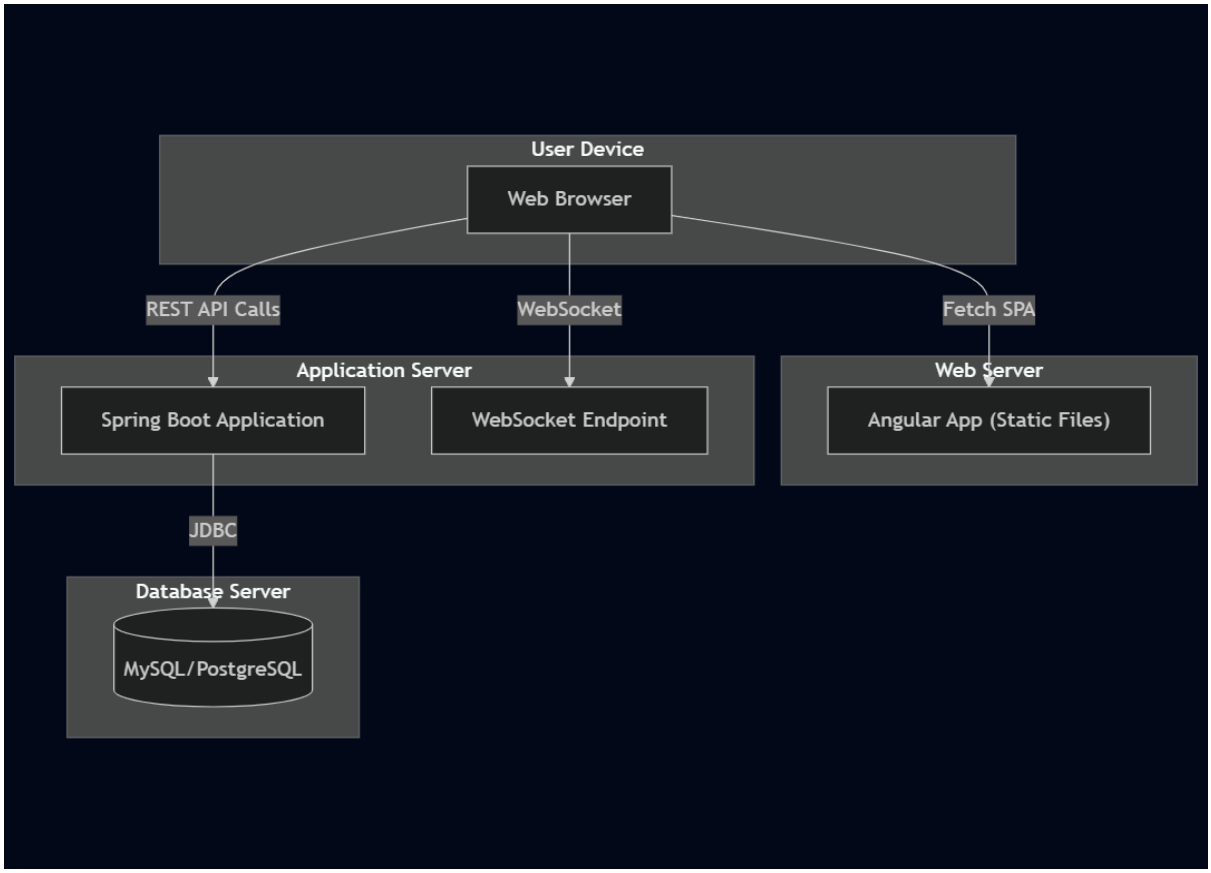


Package Design



Component and Deployment Diagram





Deliverable 3

Design Model

Dynamic Behavior

[Create the interaction diagrams (2 sequence) for 2 relevant scenarios]

Class Diagram

[Create the UML class diagram; apply GoF patterns and motivate your choice]

Data Model

[Create the data model for the system.]

System Testing

[Describe the testing methides and some test cases.]

Future Improvements

[Present some features that apply to the application scope.]

Conclusion

Bibliography