



**UNIVERSITATEA TEHNICĂ**  
DIN CLUJ-NAPOCA

# Tinder Clone

Name: MAXIM FRANCESCO  
Group: 30234

## Table of Contents

<b><i>Deliverable 1</i></b> .....	<b>3</b>
<b>Project Specification</b> .....	<b>3</b>
<b>Functional Requirements</b> .....	<b>3</b>
<b>Use Case Model</b> .....	<b>3</b>
Use Cases Identification: .....	3
UML Use Case Diagrams .....	3
<b>Supplementary Specification</b> .....	<b>4</b>
Non-functional Requirements.....	4
Design Constraints .....	5
<b>Glossary</b> .....	<b>5</b>
<b><i>Deliverable 2</i></b> .....	<b>5</b>
<b>Domain Model</b> .....	<b>6</b>
<b>Architectural Design</b> .....	<b>7</b>
Conceptual Architecture .....	7
Package Design.....	9
Component and Deployment Diagram .....	10
<b><i>Deliverable 3</i></b> .....	<b>11</b>
<b>Design Model</b> .....	<b>11</b>
Dynamic Behavior .....	11
Class Diagram .....	12
<b>Data Model</b> .....	<b>13</b>
<b><i>System Testing</i></b> .....	<b>14</b>
<b><i>Future Improvements</i></b> .....	<b>15</b>
<b><i>Conclusion</i></b> .....	<b>16</b>
<b><i>Bibliography</i></b> .....	<b>16</b>

# 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

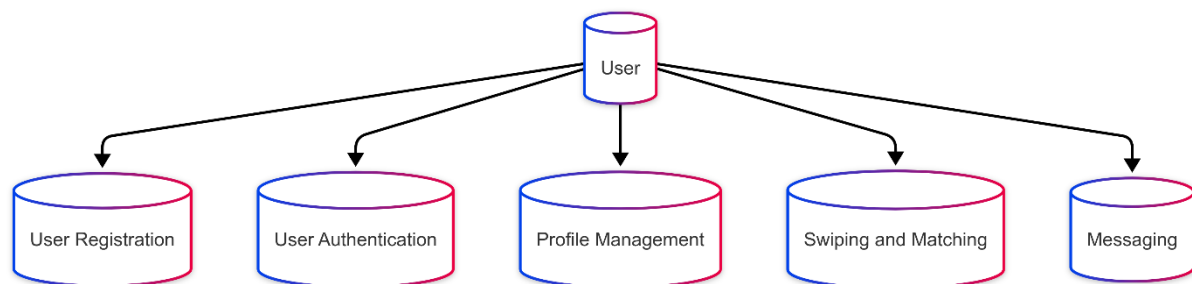
- **Main Success Scenario:**

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.

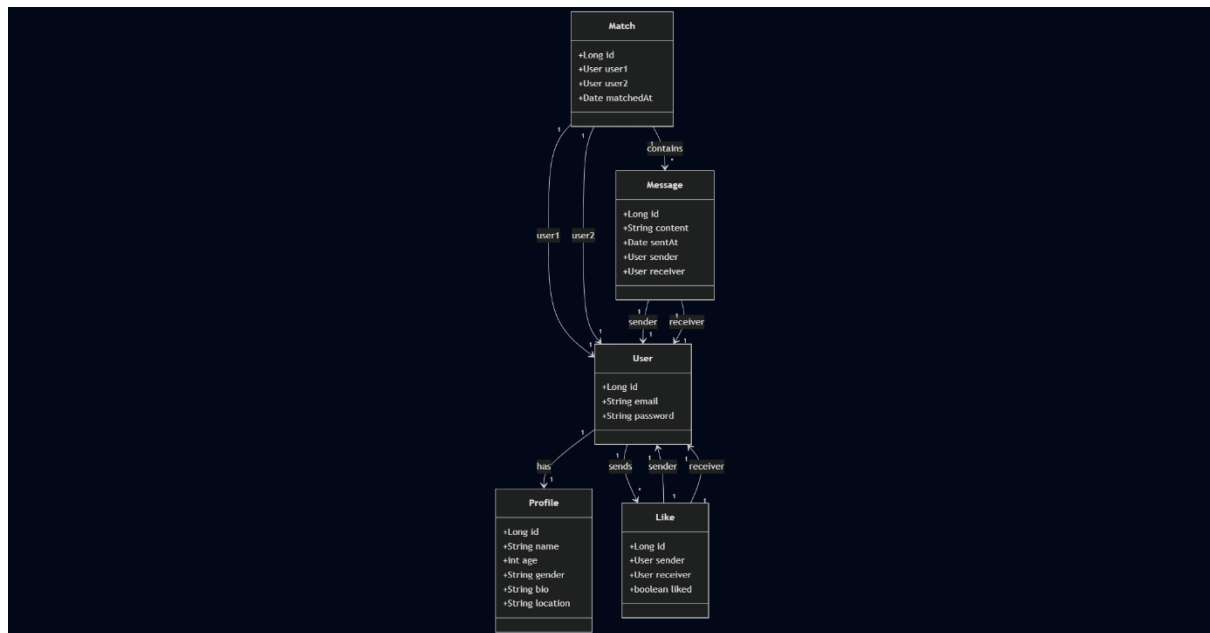
##### Entities:

- **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**.

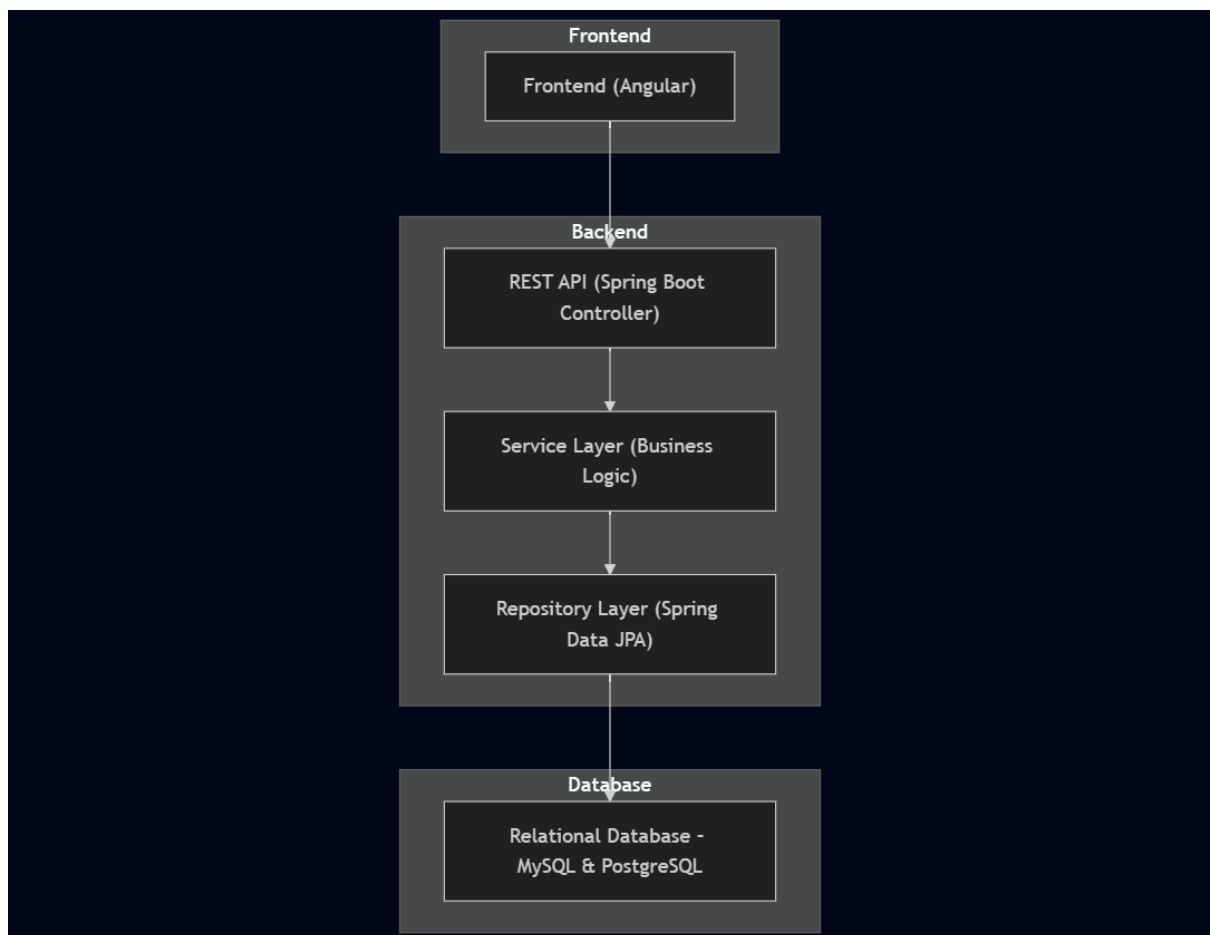
- **Database Layer:**

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:

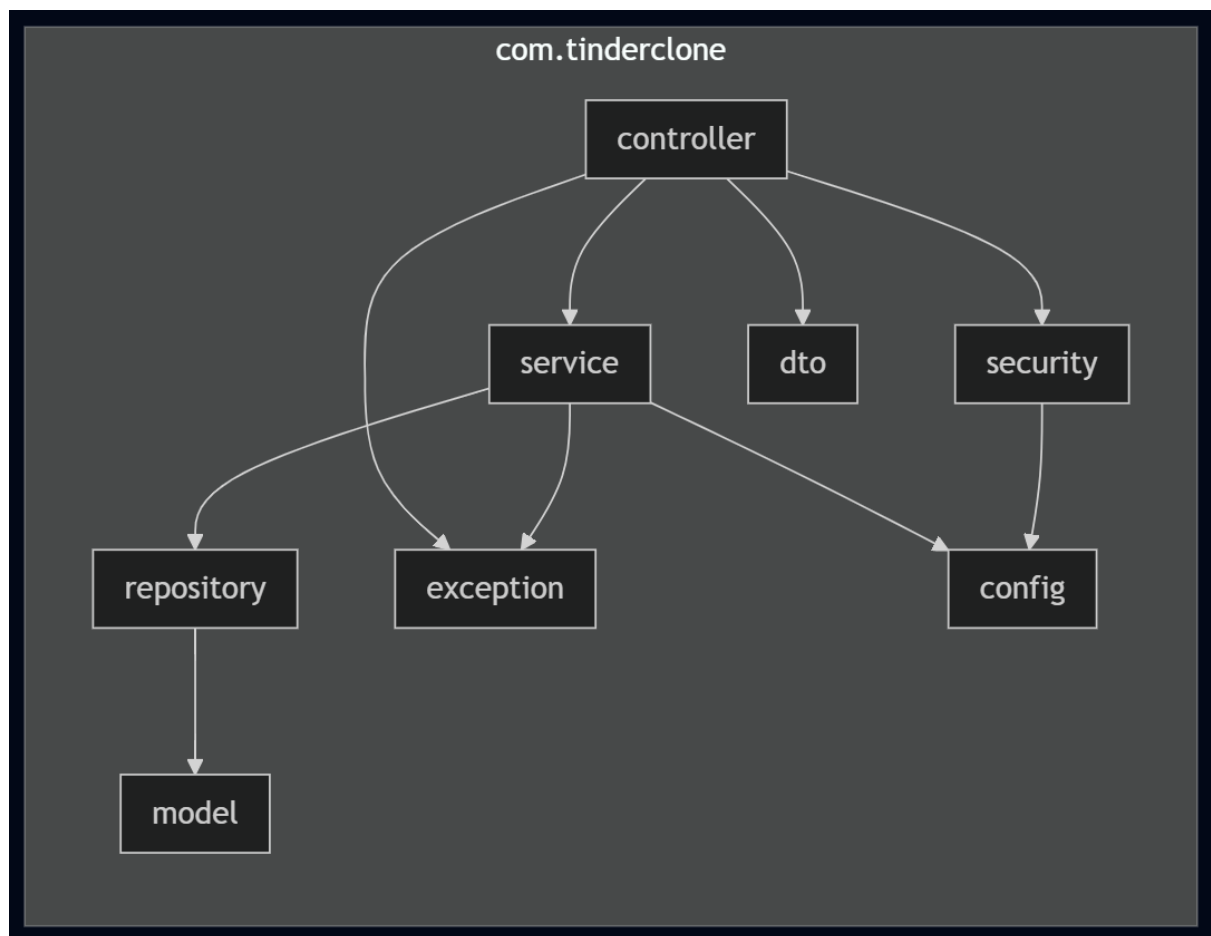
Layered architecture was chosen for its **simplicity, clear separation of concerns, and maintainability**. Each layer is loosely coupled and can be tested, updated, or scaled independently. This structure also aligns well with Spring Boot's component-based architecture and Angular's module system, allowing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

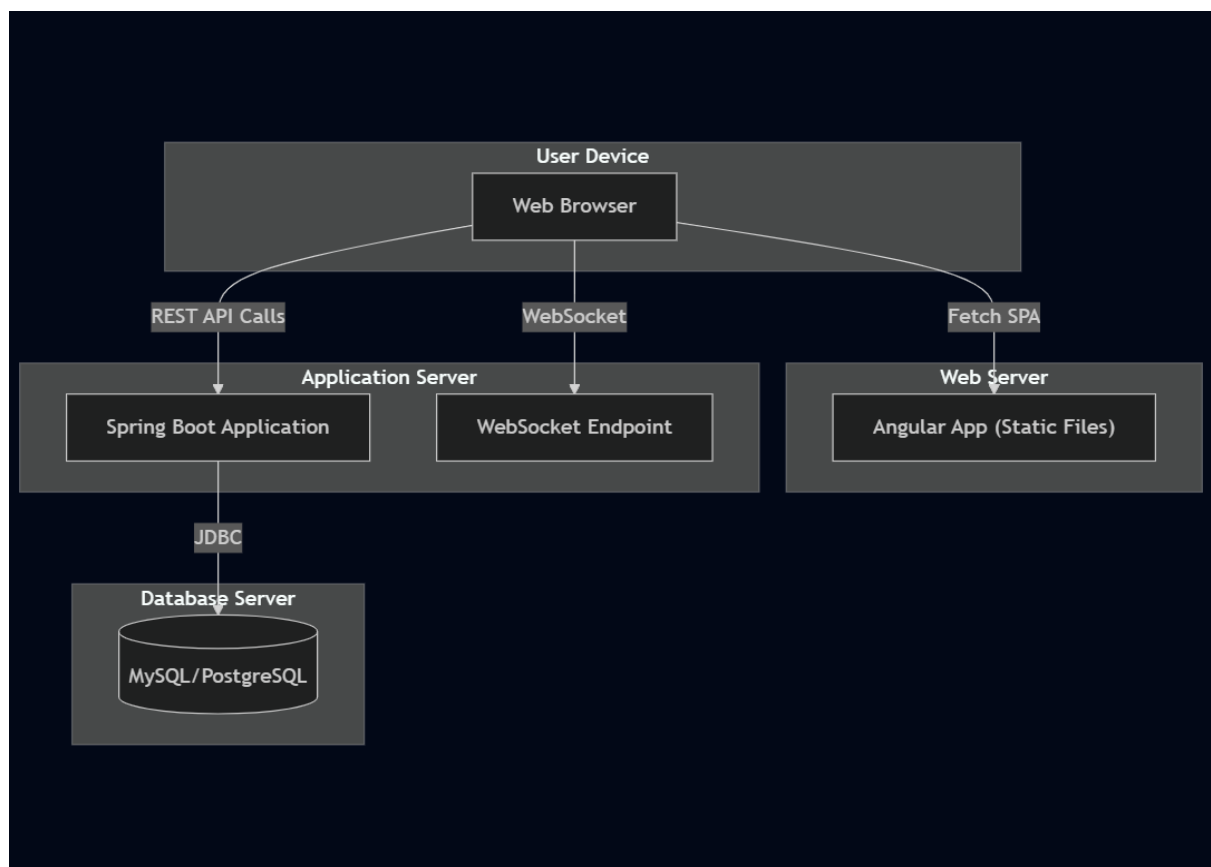
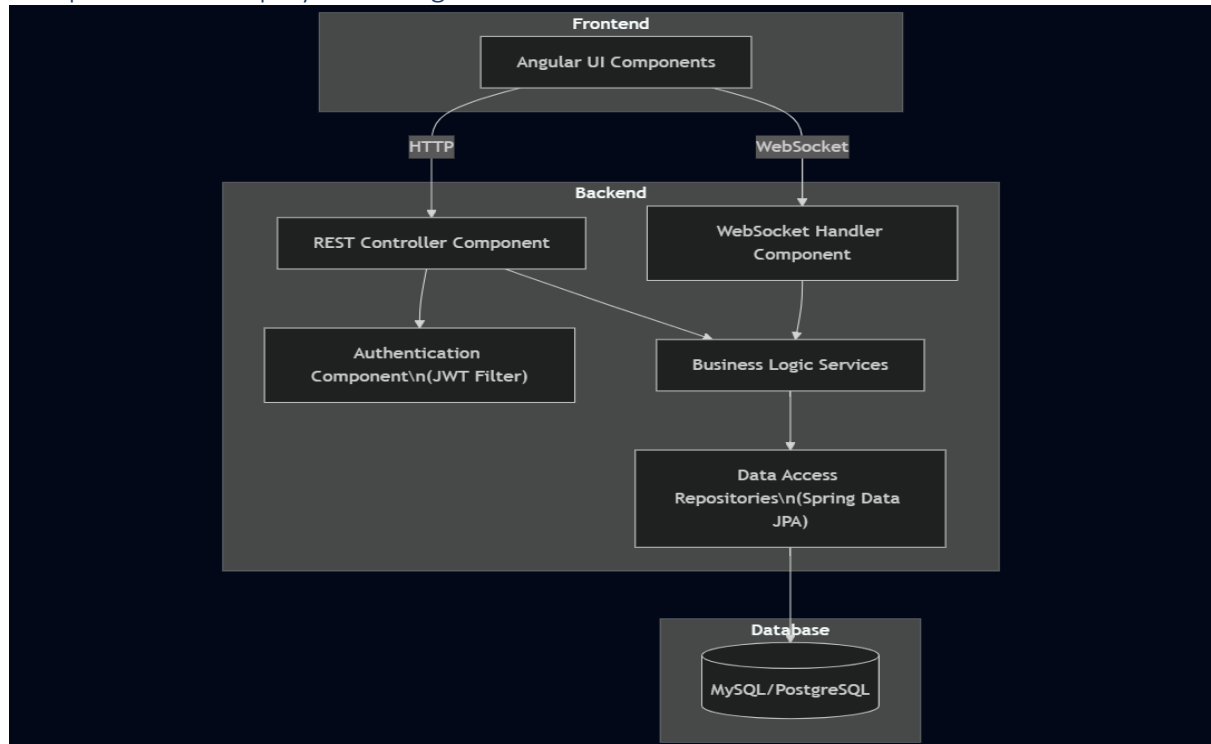




## Package Design



## Component and Deployment Diagram

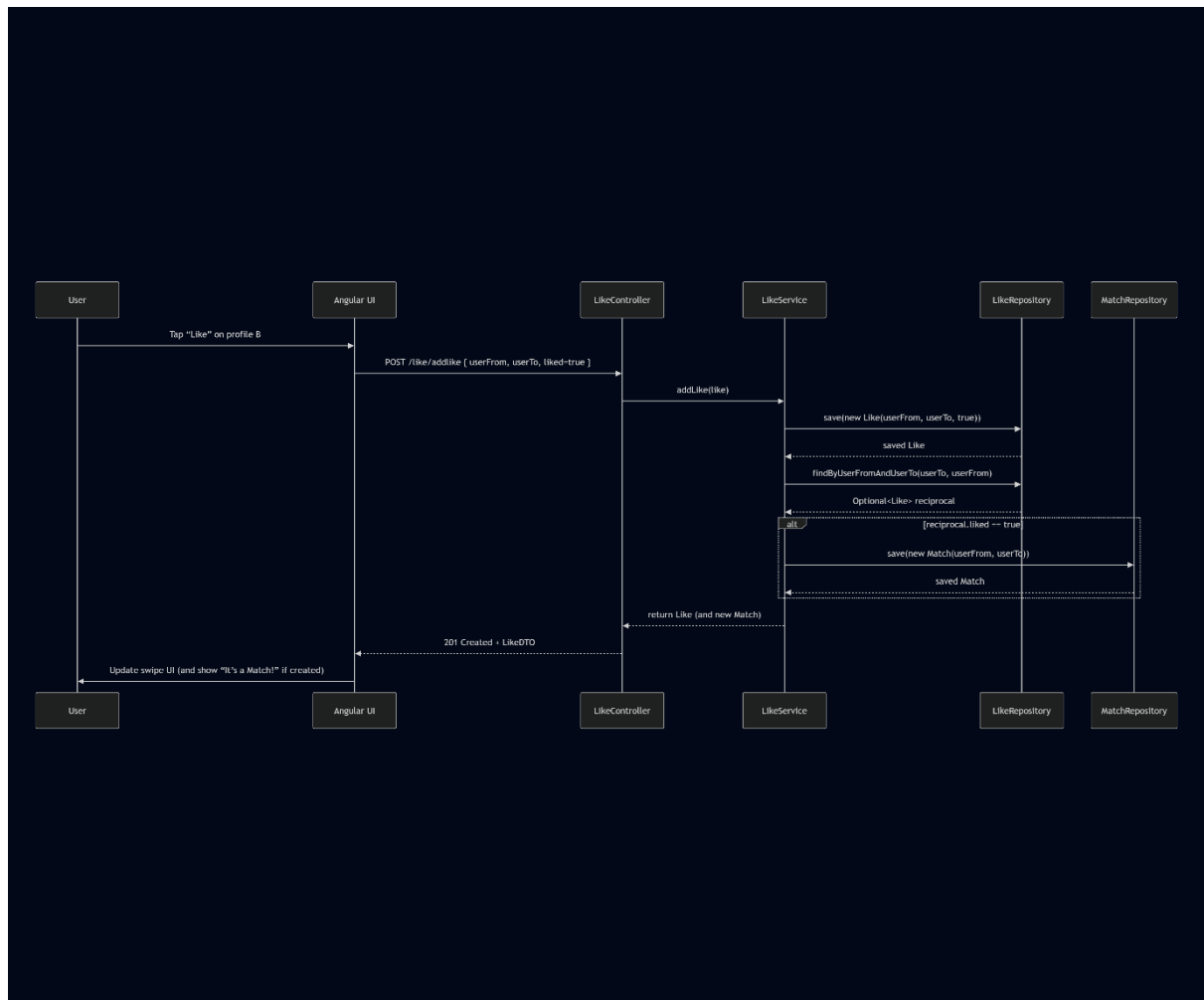


## Deliverable 3

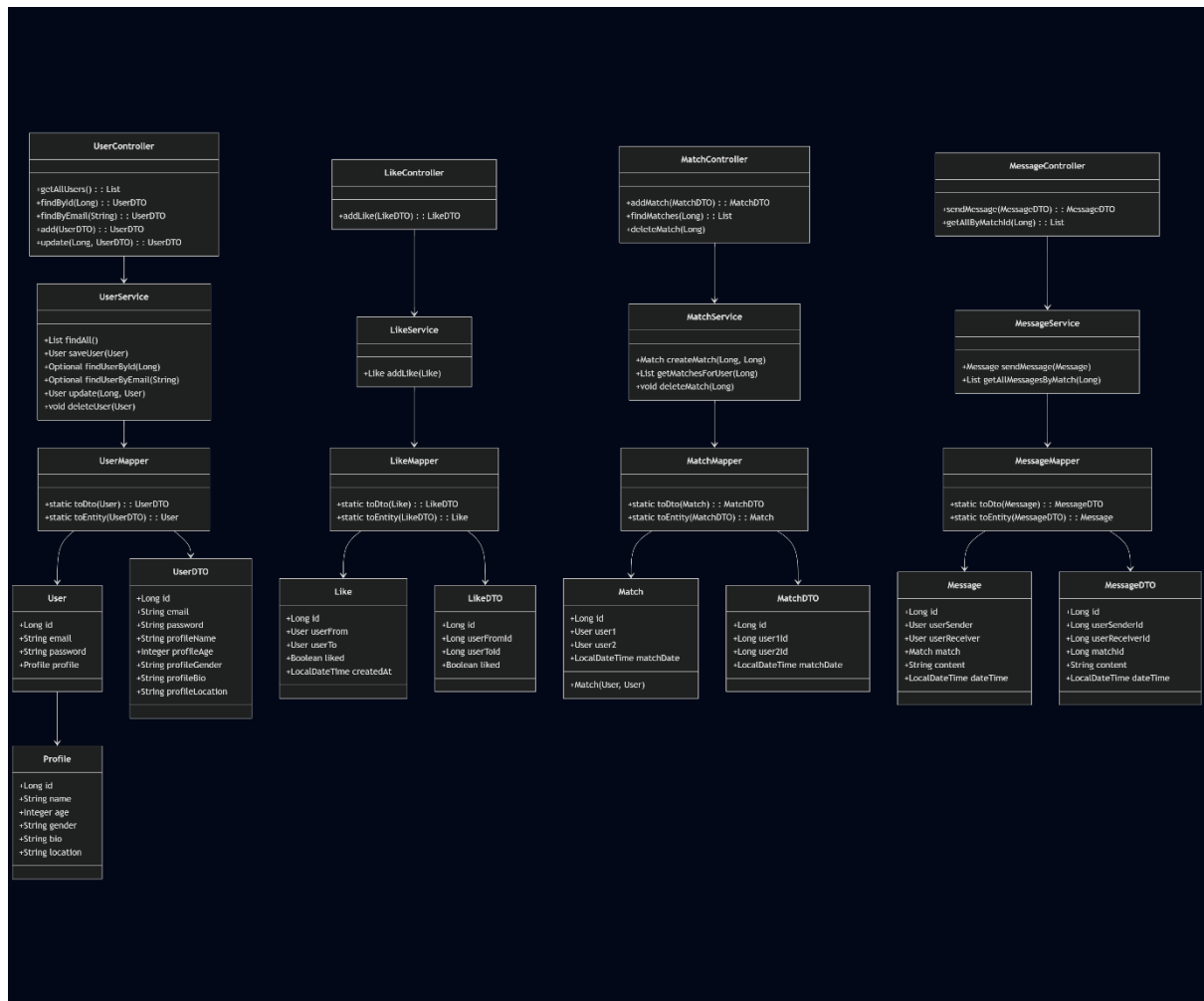
### Design Model

### Dynamic Behavior

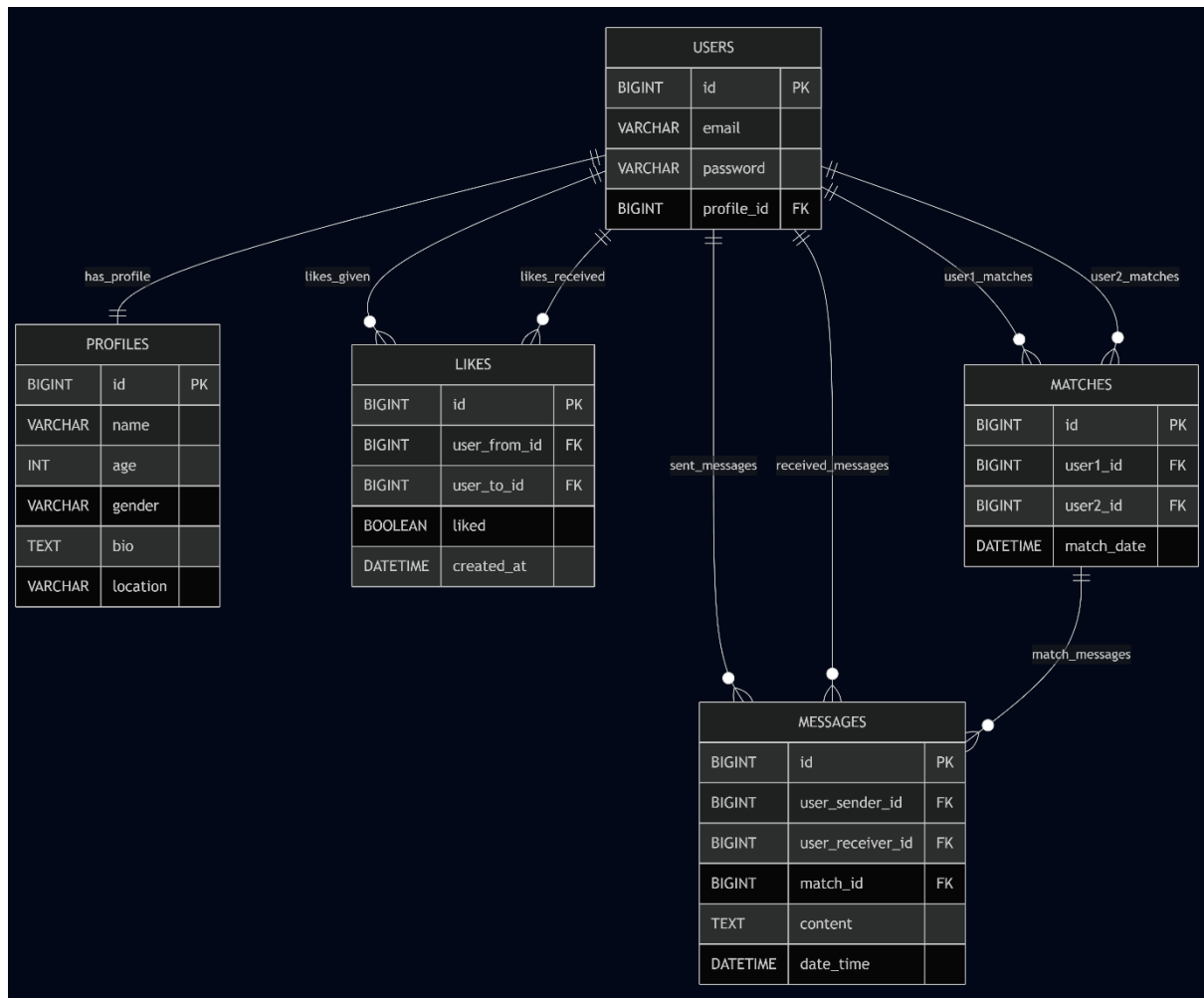




## Class Diagram



## Data Model



## System Testing

### Unit Testing:

On the backend, we write JUnit 5 tests with Mockito to isolate and verify each service method, mapper, and repository interaction. For example, the UserService's saveUser, update, and lookup methods are covered by unit tests that mock all dependencies. On the frontend, we use Jasmine/Karma to test each Angular component and service in isolation, mocking HTTP calls to confirm that form validation, state updates, and error handling behave as expected.

### Integration Testing:

We leverage Spring's @WebMvcTest for controller-service integration, and @DataJpaTest against an in-memory H2 database to verify JPA repository queries. These tests confirm that REST endpoints correctly translate HTTP requests into service calls and that entities are persisted and retrieved as intended. In parallel, our Angular test suite uses the real HttpClientTestingModule to exercise service-backend communication (with mocked backend responses).

### API Contract Testing:

Using a collection of Postman scripts (or REST-Assured suites), we automate requests against the running API to verify status codes, response payload schemas, and error

conditions. We check successful registration (201 Created), duplicate registration (400 Bad Request), login credential checks (200 OK vs. 401 Unauthorized), and proper DTO shapes for user, match, and message endpoints.

#### **End-to-End Testing:**

A Cypress test suite simulates user journeys in a real browser: signing up, logging in, swiping to like profiles, creating matches, and sending messages. These tests verify that the entire stack—Angular frontend, WebSocket messaging, and Spring Boot backend—works seamlessly, and that the UI reflects backend state correctly.

#### **Performance and Reliability Testing:**

We run JMeter load tests against critical endpoints (e.g. GET /user/all, POST /messages/send) under concurrent load to ensure 95% of requests respond within our 2 second SLA. We also configure synthetic health-check pings and monitor uptime to maintain 99.9% availability.

#### **Security Testing:**

Automated scans using OWASP ZAP validate that common vulnerabilities (e.g. injection, XSS) are not present. We additionally write integration tests to attempt unauthorized access to protected resources (profile update, match deletion) and confirm that the application returns 401 or 403 as appropriate.

By combining these methods—unit, integration, API contract, E2E, performance, and security testing—we achieve thorough coverage, catch regressions early in CI, and ensure that both functional and non-functional requirements are met.

## Future Improvements

As the Tinder Clone matures, several enhancements could extend its functionality, improve user engagement, and strengthen its competitive position:

1. **Real-Time Presence and Typing Indicators**

Augment the messaging experience by showing when a matched user is online, typing, or has read a message. Implementing WebSocket “presence” events will make conversations feel more immediate and encourage responsiveness.

2. **Rich Media Support**

Allow users to send photos, voice notes, or short video clips within the chat interface. This would involve extending the Message model to carry multimedia payloads and integrating a storage service (e.g., AWS S3 or Firebase Storage) with secure, expiring download URLs.

3. **AI-Driven Match Suggestions**

Leverage machine learning to suggest potential matches based on profile similarities, past swiping behavior, and conversational chemistry. A recommendation engine could run offline—analyzing user interactions—and surface high-quality recommendations each time a user opens the app.

4. **In-App Video and Voice Calling**

Integrate a WebRTC-based voice and video calling feature so matched users can connect face-to-face without leaving the application. This would strengthen user engagement, reduce drop-off between matching and meeting, and differentiate the platform.

5. **Advanced Safety and Verification**

Add selfie-based profile verification to reduce fake accounts, plus in-chat safety prompts (e.g., “Report” or “Block” buttons) and ephemeral “panic” alerts. A centralized moderation dashboard could track reports and enforce community guidelines at scale.

6. **Location-Based Features**

Introduce geospatial matchmaking—showing nearby users in real time—and add “event” features for groups of friends to meet in common locations or attend local gatherings.

7. **Subscription & Monetization**

Implement a tiered subscription model offering “super likes,” profile boosts, and invisible browsing. Integrate payment processing (Stripe or PayPal) and an admin dashboard to manage promotions, refunds, and analytics.

8. **Cross-Platform Mobile Apps**

Build native iOS and Android clients using React Native or Flutter, sharing core business logic with the web via a common REST/WebSocket API, to reach a broader audience and offer push notifications.

## Conclusion

This Tinder Clone demonstrates how a modern, full-stack application can be architected, designed, and implemented using Angular on the front end and Spring Boot on the back end. By adhering to a clear domain model, layered architecture, and robust DTO-based APIs, we have ensured separation of concerns, maintainability, and testability. The dynamic behavior captured in our sequence diagrams highlights seamless user flows—from registration and profile editing to swiping, matching, and messaging—while our class and data models show how entities relate and how GoF patterns (e.g., Mapper for DTO conversion) promote clean code.

Comprehensive unit and integration tests verify each controller, service, and repository, giving confidence in core functionality. Non-functional requirements around performance, scalability, reliability, and security have guided our technology choices—Angular’s standalone components and Spring Data JPA ensure rapid development and easy cloud deployment.

Looking ahead, the roadmap of future improvements—such as real-time presence, rich-media messaging, AI-driven match suggestions, and mobile clients—provides a clear path to evolve this prototype into a production-grade platform. Overall, this project not only fulfills the specified functional requirements but also lays a solid foundation for continuous enhancement, demonstrating best practices in modern web application development.

## Bibliography