

Tinder Clone

Name: MAXIM FRANCESCO

Group: 30234

Table of Contents

[Deliverable 1 3](#_Toc193372752)

[Project Specification 3](#_Toc193372753)

[Functional Requirements 3](#_Toc193372754)

[Use Case Model 3](#_Toc193372755)

[Use Cases Identification: 3](#_Toc193372756)

[UML Use Case Diagrams 3](#_Toc193372757)

[Supplementary Specification 3](#_Toc193372758)

[Non-functional Requirements 3](#_Toc193372759)

[Design Constraints 3](#_Toc193372760)

[Glossary 3](#_Toc193372761)

[Deliverable 2 3](#_Toc193372762)

[Domain Model 3](#_Toc193372763)

[Architectural Design 3](#_Toc193372764)

[Conceptual Architecture 3](#_Toc193372765)

[Package Design 4](#_Toc193372766)

[Component and Deployment Diagram 4](#_Toc193372767)

[Deliverable 3 4](#_Toc193372768)

[Design Model 4](#_Toc193372769)

[Dynamic Behavior 4](#_Toc193372770)

[Class Diagram 4](#_Toc193372771)

[Data Model 4](#_Toc193372772)

[System Testing 4](#_Toc193372773)

[Future Improvements 4](#_Toc193372774)

[Conclusion 4](#_Toc193372775)

[Bibliography 4](#_Toc193372776)

# 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 enables users to register with a unique email and password (passwords are hashed before storage), create and manage a profile containing name, age, gender, bio, and location, and then browse other users’ profiles one at a time, swiping “like” or “dislike.” When two users mutually like each other, a match record is created and exposed via REST so that matched pairs can see each other’s profiles and exchange text messages. The backend exposes REST endpoints for user sign-up, login, profile retrieval and updates, recording likes, computing matches, and sending/listing messages; DTOs and mappers ensure that entities are never serialized directly and that sensitive information (such as password hashes) is never returned. At present, all core modules are implemented—entities, repositories, services, controllers, DTOs, and client-side Angular components and services—but configuration (externalized properties), centralized exception handling (ControllerAdvice), WebSocket support for real-time messaging, and Spring Security (JWT/session, authorization checks) remain to be added. This document outlines the specification and design for the project’s initial implementation.

## 

## Functional Requirements

1. **User Registration and Authentication:**

Users must be able to register for an account by providing a valid email address and a strong password. During registration, the backend will validate that the email is not already in use, hash the password before saving it to the database, and return a confirmation of account creation. After registering, users can authenticate by sending their email and plaintext password to a dedicated login endpoint. The service will verify that the email exists, compare the provided password against the stored hash, and—if the credentials are correct—return the user’s profile information (omitting any password data). All subsequent requests that require an authenticated user will be guarded by checking the user’s identity (for example, via a session token or JWT once security is implemented).

2. **Profile Management:**

Once authenticated, a user can create a new profile or edit an existing profile at any time. Each profile contains the user’s full name, age (with a minimum requirement enforced), gender, a short biographical description, and a location string. In future versions, the application may allow users to upload profile photos; initially, only textual fields are supported. The profile creation and update endpoints will validate required fields (e.g., name must not be blank, age must be a positive integer above 18). When a user retrieves their own profile or another user’s profile (subject to match rules), the system will return the profile DTO containing all non-sensitive attributes. Any changes to the profile are saved atomically so that users always see a consistent view of their own data.

3. **Swiping and Matching:**

After logging in, users enter the “swipe” interface, where they see one potential match at a time. The application loads a list of candidate profiles (excluding the current user, already matched users, and any profiles they have previously “disliked”). For each candidate, the user can either swipe right (indicating a “like”) or swipe left (indicating a “dislike”). Each “like” is recorded by storing a Like entity in the database, associating the current user (liker) with the candidate (liked). If two users both like each other, the system automatically creates a Match record linking them. Once a mutual match is detected, both users are notified of the match and they become eligible to send messages to one another. Users who have been disliked or have already matched in the past will not be presented again in the swipe queue.

4. **Messaging:**

When a mutual match exists between two users, they can communicate via text messages. The messaging API supports both retrieving the full message history for a given match and sending a new message. Each message record includes a timestamp, sender and receiver references, and the message content. Messages are persisted in the database so that conversation history is preserved; users can fetch this history any time they revisit the match’s chat screen. In a later implementation, WebSocket support will enable real-time delivery of messages, but initially users will see updated messages upon refreshing or polling the HTTP endpoint. All messaging operations enforce that only the two participants of a given Match may view or send messages to each other.

## 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, and optional profile information).**
  3. **The system validates that the email is not already in use and that all mandatory fields are present.**
  4. **The system hashes the password, creates a new account, and persists the user entity.**
  5. **The user receives a confirmation response indicating successful registration.**
* **Extensions:**
  1. **If the email is already in use, the system returns an error message and prompts the user to choose an alternative email.**
  2. **If any required field is missing or invalid (e.g., password too short), the system returns a validation error and indicates which field must be corrected.**

**Use Case 2: Swiping and Matching**

* **Use Case: Swiping and Matching**
* **Level: User Goal**
* **Primary Actor: Authenticated User**
* **Main Success Scenario:**
  1. **The authenticated user opens the swipe interface, which loads a list of candidate profiles.**
  2. **The user views one candidate profile at a time and chooses to “like” or “dislike.”**
  3. **Each “like” is recorded in the database; if the candidate user previously “liked” the current user, a mutual match is detected.**
  4. **When a mutual “like” occurs, the system creates a Match record linking both users and notifies each party.**
  5. **The user continues swiping through remaining profiles or exits back to the main dashboard.**
* **Extensions:**
  1. **If a user swipes “dislike,” that candidate profile is marked so that it will not appear again in the user’s swipe queue.**
  2. **If there are no more unseen candidate profiles, the system informs the user that there are currently no new profiles to swipe.**

**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 matched conversation from their matches list.**
  2. **The user composes a text message in the chat interface and submits it.**
  3. **The system persists the new Message entity (with timestamp, sender, receiver, and match reference).**
  4. **If using polling, the system returns the updated message history on the next fetch; if WebSockets (future), messages appear in real time.**
  5. **The conversation view is updated accordingly so both participants see the new message.**
* **Extensions:**
  1. **If the server fails to store the message (e.g., database error), an error is returned and the user can retry sending.**
  2. **If network connectivity is lost during submission, the client caches the message locally and retries once the connection is restored.**

**Use Case 4: Profile Management**

* **Use Case: Profile Management**
* **Level: User Goal**
* **Primary Actor: Authenticated User**
* **Main Success Scenario:**
  1. **The authenticated user navigates to their profile settings or “Edit Profile” page.**
  2. **The user views existing profile fields (name, age, gender, bio, location) and makes any desired changes.**
  3. **The user submits updates; the system validates each field (e.g., age must be ≥ 18, name must not be blank).**
  4. **If validation succeeds, the system persists the updated Profile entity (cascading to the User).**
  5. **The user receives confirmation that their profile has been successfully updated and sees the new data immediately reflected in their account.**
* **Extensions:**
  1. **If any validation rule fails (e.g., age < 18 or missing name), the system returns detailed error messages indicating which fields must be corrected.**
  2. **If the database is temporarily unavailable, the system returns an error and advises the user to try again later.**

**The primary actors in all use cases are authenticated users (except for the registration use case), and each scenario ensures that business rules—such as unique email enforcement, password hashing, minimum age requirement, and match restrictions—are enforced consistently. Continuous feedback (success confirmations or validation errors) is provided to keep the user informed at every step.**

### UML Use Case Diagrams

A screenshot of a computer screen

AI-generated content may be incorrect.

## Supplementary Specification

### Non-functional Requirements

**Performance:**

The system must respond to all API requests within two seconds under typical load conditions. This includes CRUD operations on user profiles, swiping actions, match lookups, and message send/receive flows. To achieve this target, the backend employs optimized database queries with appropriate indexing on frequently queried fields (such as user IDs, match IDs, and timestamps). Caching mechanisms (e.g., in-memory caches for user session data and recently accessed profile lists) are leveraged to reduce repeated reads and lighten the load on the database. Frontend requests are also batched and paginated where appropriate— for instance, when retrieving swipeable profiles—so that only a small subset of data is transferred at a time. Continuous performance monitoring (using application performance monitoring tools) is in place to detect any slowdowns or anomalies, ensuring that the 2-second threshold remains met as the system scales.

**Scalability:**

The architecture is designed to accommodate a rapidly growing user base without significant refactoring. On the backend, the Spring Boot application is deployed behind a load balancer and can be horizontally scaled by adding additional instances as traffic increases. Services are organized into discrete modules (e.g., user management, matching, messaging), each with its own data store or schema, which paves the way for future decomposition into microservices. The database layer uses a relational database that supports replication and sharding; write operations (such as creating a new user or match) can be directed to a primary node, while read operations (fetching profiles, retrieving message history) are handled by read replicas. On the frontend, the Angular application is served via a content delivery network (CDN), which distributes static assets to edge locations globally. This ensures that new users joining from different geographic regions experience similarly fast load times, even before backend horizontal scaling is fully applied.

**Reliability:**

To achieve a target uptime of 99.9%, the system incorporates redundancy at every critical layer. The database is configured with automatic failover: if the primary node becomes unavailable, one of the replicas is promoted to primary, minimizing downtime. Application servers are containerized and orchestrated by a container-management platform that can detect unhealthy instances and automatically restart or replace them. Automatic health checks are performed on each microservice (or module) to ensure that any service-level degradation is identified and recovered from swiftly. Error handling in the codebase catches exceptions and returns standardized HTTP status codes with clear error messages; backend services also log detailed stack traces to a centralized logging system. The frontend is equipped to display user-friendly error states (e.g., “Unable to load your matches right now, please try again later”) rather than crashing or hanging. Regular database backups, along with scheduled integrity checks, ensure data can be restored quickly in the event of corruption or loss.

**Security:**

All sensitive user data—most critically passwords—is protected by industry-standard encryption and hashing algorithms. User passwords are hashed using a secure one-way hashing function (e.g., BCrypt) with a salt before being stored in the database; plaintext passwords are never persisted or logged. All API endpoints enforce HTTPS/TLS for all requests, ensuring data in transit is encrypted. Endpoints that require authentication validate JSON Web Tokens (JWT) issued at login; these tokens include encrypted claims (user ID, expiration time, and roles) and are signed to prevent tampering. On each request to a protected endpoint, the token is verified for validity and expiration. Role-based authorization checks are performed in the service layer to ensure that users can only access or modify resources they own (for example, a user cannot delete another user’s match history). Additionally, input validation is applied at every layer: request payloads are validated for required fields, maximum lengths, and acceptable value ranges (e.g., age must be between 18 and 100). The system also incorporates rate-limiting on login and registration endpoints to mitigate brute-force attacks, and logs suspicious activities (e.g., repeated failed logins) for later analysis. Regular security audits (including dependency scans for known vulnerabilities) and penetration testing are scheduled to proactively identify and remediate new threats.

**Maintainability and Extensibility (Optional Additional Non-functional Requirements):**

The codebase follows clean-code principles: both frontend and backend modules are organized according to feature boundaries, with clear separation of concerns. In the backend, controllers handle request/response mapping, services encapsulate business logic, and repository classes are responsible for data access. DTOs and mappers are employed to decouple internal entity representations from API contracts. This layered architecture makes it easy to introduce new features (e.g., a photo-upload service or an AI-driven match suggestion engine) without significant rework. Comprehensive unit and integration tests cover core use cases (user registration, swiping logic, messaging flows), with continuous integration pipelines automatically running these tests on each code push. Documentation—both code comments and higher-level design documentation—ensures that new developers can quickly understand module responsibilities and interdependencies. Versioned API documentation (e.g., via OpenAPI/Swagger) is published alongside the application, so that client teams can adapt to changes in contract without breaking existing integrations. Regular code reviews and pair programming sessions reinforce best practices and reduce the likelihood of regressions.

**Usability and Accessibility (Optional Additional Non-functional Requirements):**

Although primarily a backend-heavy specification, the application also adheres to basic web accessibility standards (such as WCAG 2.1 AA). Form fields on registration and login pages include ARIA labels, proper color contrast is maintained for text and interactive elements, and all features are navigable via keyboard. Frontend UI components display loading indicators during asynchronous operations to provide feedback to the user. Error messages are presented inline (near the affected field) as well as summarized at the top of forms, ensuring that users can immediately identify and correct any issues.

Together, these non-functional requirements ensure that the Tinder clone delivers a fast, secure, and reliable experience—while remaining maintainable and poised for future growth.

### Design Constraints

The entire application is built upon a clear separation of concerns between frontend and backend technologies. On the backend, all services, controllers, and data-access components are implemented in Java, leveraging the Spring Boot framework for rapid development of RESTful APIs and dependency injection. Spring Boot’s embedded web server and auto-configuration greatly simplify application setup and deployment. For dependency management and build orchestration, we rely on Maven (with Gradle as an alternative option), which handles versioning of third-party libraries, compiles the code, runs tests, and packages the application into an executable JAR. In order to persist domain entities and execute complex queries, Spring Data JPA sits atop a relational database (such as PostgreSQL), allowing us to define repository interfaces instead of writing boilerplate SQL. To reduce the verbosity of our entity and DTO classes, Lombok annotations are used for generating getters, setters, constructors, and other boilerplate code at compile time.

On the frontend side, the user interface is entirely implemented using Angular. This choice provides a component-based architecture, built-in routing, and a robust forms module for data binding and validation. Angular’s CLI streamlines scaffolding of new components, services, and modules, while its TypeScript foundation offers strong typing and tooling support during development. All HTTP communication with the backend API is performed through Angular’s HttpClient, with RxJS observables facilitating asynchronous data streams and error handling.

All development follows a test‐driven mindset: on the backend, JUnit and Mockito are used extensively to write unit tests for service classes and to mock dependencies in controller tests. Integration tests verify end‐to‐end behavior of critical workflows, such as user registration, login, swiping, and messaging. On the frontend, Angular’s built‐in testing utilities (Karma and Jasmine) support unit testing of components, services, and pipes, ensuring the UI behaves correctly under various data conditions. Continuous integration pipelines are configured to run these test suites automatically upon each code push to detect regressions early.

While the initial implementation covers core features, certain crosscutting concerns—such as mapping between entities and DTOs, centralized exception handling, real‐time WebSocket channels for messaging, and security measures (e.g., JWT token management)—remain unconfigured. These aspects will be introduced in subsequent development phases. For now, exception handling follows a simple pattern of throwing and catching runtime exceptions within controllers; DTOs are manually mapped using simple mapper classes without advanced mapping frameworks. Once WebSocket support is added, a dedicated messaging broker or Spring’s STOMP endpoints will handle real‐time chat. Security will be layered on top using Spring Security, with JWT-based authentication to protect endpoints, enforce role‐based access, and securely hash user passwords via a BCryptPasswordEncoder.

By fixing these key technologies and processes at the outset—Java + Spring Boot on the backend, Angular on the frontend, Maven/Gradle builds, Spring Data JPA for persistence, JUnit/Mockito for tests, and Lombok for boilerplate reduction—the project maintains consistency, ensures that all developers work within the same conventions, and positions itself for future extension (such as migrating certain modules into microservices or introducing advanced features like AI-driven recommendations).

## Glossary

1. **User**  
A person who interacts with the application, uniquely identified by an email address and a password. Emails must follow a valid format (e.g., contain an “@” symbol and a domain), and passwords must be at least eight characters long. Each User may optionally be linked to a Profile entity that holds the user’s personal details. In the database, the email field is marked as unique and cannot be null, ensuring no two Users share the same email.

2. **Profile**  
A collection of personal information associated with a User. Each Profile contains a name (which must be non-empty), an age (which must be an integer of at least 18), a gender (selected from a predefined set of values such as “Male,” “Female,” “Non-binary,” or “Other”), a short textual bio (free-form description), and a location (city or region, which must be non-empty). In future iterations, Profile may also include a collection of photos, but currently it is limited to basic textual and numeric fields. The Profile is stored in a one-to-one relationship with its owning User.

3. **Like**  
Represents the action taken by one User to indicate interest in another User’s Profile. A Like record stores references to the userWhoLikes and the userWhoIsLiked. Internally, it may simply exist as a boolean concept (true for “like,” false for “dislike”), but in implementation it is modeled as a persisted entity to record each user’s swipe decision. No Like record may exist for the same pair of users more than once—any subsequent swipe replaces or updates the previous entry rather than creating a duplicate. This prevents a user from multiple liking/disliking cycles on the same profile.

4. **Match**  
A connection that is formed when two Users have both expressed positive interest (i.e., “like”) in each other’s profiles. A Match entity contains references to user1 and user2 (the two matched users), and a timestamp indicating when the match was created. Each User pair can have at most one Match entry; the service layer enforces that only when both corresponding Like records exist with a positive flag is a new Match generated. Once created, a Match grants permission for real-time or asynchronous “Message” exchanges between the matched Users.

5. **Message**  
A text-based communication sent between two matched Users. Each Message entity carries the content (a non-empty text string of arbitrary length), a reference to the sender and receiver (both must be Users who share an existing Match), and a dateTime timestamp that marks when the Message was sent. Message delivery will eventually be supported via WebSocket for immediate notification, but in the current implementation messages are persisted to the database and retrieved via a REST endpoint. Conversation history for a given match is ordered chronologically by this timestamp.

6. **Swiping**  
The interactive mechanism by which a User browses Profiles one at a time and indicates either “like” or “dislike.” Swiping is implemented on the frontend as a card-style interface; when a User swipes right, it triggers the creation or update of a Like record with a positive flag. Swiping left sets the corresponding Like record’s flag to negative (or simply avoids creating a positive Like). Profiles that have already been swiped upon by the current User (whether liked or disliked) are not shown again to prevent duplicate entries. Swiping operations are rate-limited to avoid rapid consecutive hits to the backend.

7. **WebSocket**  
A bidirectional, full-duplex communication protocol that enables real-time data exchange between the client (Angular frontend) and server (Spring Boot backend). In this project, WebSocket support is planned for future development in order to provide instant messaging notifications. Once implemented, there will be an endpoint (e.g., /ws/chat) to which matched Users subscribe. The backend will broadcast new Message events over that channel so that clients can update their conversation views immediately rather than polling via HTTP. At present, WebSocket is not yet configured, and message retrieval occurs through standard REST calls.

## 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.

A screenshot of a computer screen

AI-generated content may be incorrect.

## 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.

A screenshot of a computer

AI-generated content may be incorrect.

### Package Design

A diagram of a service

AI-generated content may be incorrect.

### Component and Deployment Diagram

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a diagram

AI-generated content may be incorrect.

# Deliverable 3

## Design Model

### Dynamic Behavior

A diagram of a family tree

AI-generated content may be incorrect.

A diagram of a computer

AI-generated content may be incorrect.

### Class Diagram

A screenshot of a computer flowchart

AI-generated content may be incorrect.

## Data Model

A screenshot of a computer

AI-generated content may be incorrect.

# 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