Project 3 Document:

Team Number:

Team 1

Team members:

Ibrahim – Lead Frontend Developer

Maren – Lead Backend Developer

Zack – Scrum Master / Full Stack Developer

Abdullah – Frontend Developer

Nick – Backend Developer

Saurav – Frontend Developer

Project Name:

JayMatch

Project Synposis:

A KU-verified dating web app. Students sign in with @ku.edu, create profiles, get daily match suggestions, and chat when there's a mutual like.

Project Architecture:

Overview

JayMatch is a KU-exclusive dating application designed to create a safe and engaging platform for students to connect. The app verifies KU identities through @ku.edu email addresses, preventing fake profiles and fostering a trustworthy environment. Users can create personal profiles, receive daily match suggestions, and chat with other verified KU students after a mutual like.

JayMatch is structured as a modern full-stack web application consisting of a frontend, backend, and database layer, deployed using cloud-based services for reliability and scalability. Security, performance, and usability are central to the architecture.

1. Frontend (Angular)

The frontend is built using Angular, chosen for its modular design and efficient component system. The Angular app manages all user interactions, dynamic UI updates, and routing between pages such as:

- Login and verification
- Profile creation and editing
- Daily matches and likes
- Chat interface

It communicates exclusively with the backend via RESTful APIs over HTTPS, ensuring no sensitive data is handled client-side. The frontend is hosted on Vercel, which supports static Angular builds and provides continuous deployment through Git integration.

Key Angular modules:

- AuthModule for login, MFA, and JWT handling
- ProfileModule for managing user data
- MatchModule for browsing and liking
- ChatModule for real-time messaging (WebSocket integration)

2. Backend (Rust)

The backend serves as the core of the system, implemented in Rust using the Axum or Actix-web framework. Rust provides performance, safety, and memory reliability, critical for handling sensitive user data securely.

The backend has several responsibilities:

- Authentication & MFA: Validates KU email addresses and issues JWTs after multi-factor verification via email.
- Profile Management: Handles CRUD operations for user data.
- Matchmaking: Runs the “Daily 10” recommendation algorithm that matches users based on profile attributes and preferences.
- Chat Service: Manages one-to-one chat sessions between matched users through WebSocket connections.
- Security Enforcement: Uses Argon2 for password hashing, HTTPS for all traffic, and role-based access for internal routes.

The backend is deployed via Fly.io or Railway, with automatic redeployments on code updates. It acts as a middleman between the frontend and the SQLite database.

3. Database Layer (SQLite)

SQLite serves as the lightweight relational database for JayMatch. It stores user profiles, messages, and authentication data. SQLite was chosen for its simplicity and ease of integration with Rust (via SQLx or Diesel).

Schema outline:

- users(id, email, password_hash, name, year, major, bio, interests, photo_url)
- matches(user_id, match_id, created_at)
- messages(id, sender_id, receiver_id, content, timestamp)
- reports(id, reporter_id, reported_id, reason, timestamp)

Data integrity and privacy are ensured through input validation and restricted query permissions. In future sprints, the database may migrate to PostgreSQL if scalability demands increase.

4. Communication Flow

All frontend actions route through the backend API. The Angular client never communicates directly with the database.

Example flow: User Login

1. User enters KU email.
2. Frontend sends request → /auth/send_code.
3. Backend verifies domain and emails MFA code.
4. User submits code → /auth/verify_code.
5. Backend validates and issues JWT.
6. Frontend stores token and redirects to Profile page.

5. Deployment & Infrastructure

- Frontend: Deployed on Vercel.

- Backend: Deployed on Fly.io, Railway, or Render (Rust build).
- Database: SQLite file persisted on backend server storage.
- Domain/TLS: Configured through a custom domain provider, enforcing HTTPS.

Continuous integration pipelines will automatically build, test, and deploy updates. Secrets (API keys, JWT signing keys) are stored securely in environment variables.

6. Security & Privacy

Security is a major design principle for JayMatch.

- Password Hashing: Argon2 used to protect user passwords.
- JWT Authentication: Stateless session management ensures fast, secure access control.
- HTTPS: All frontend-backend communication encrypted.
- Access Control: Only KU emails are permitted during signup.
- Privacy Controls: Users can block or report others; reported content is logged for moderation.

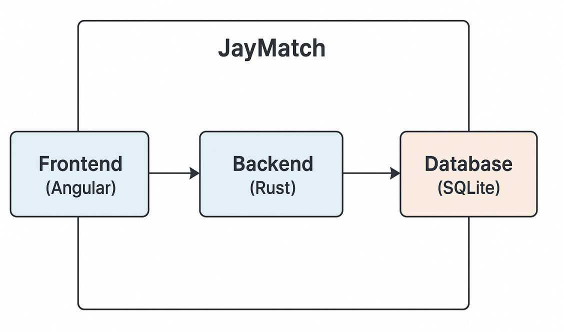

Diagram 1: This diagram outlines the three main parts of JayMatch and how they interact. The Angular frontend communicates with the Rust backend through secure RESTful APIs, while the

backend connects to the SQLite database for all data storage and retrieval. WebSocket connections between the frontend and backend enable real-time chat and match updates.

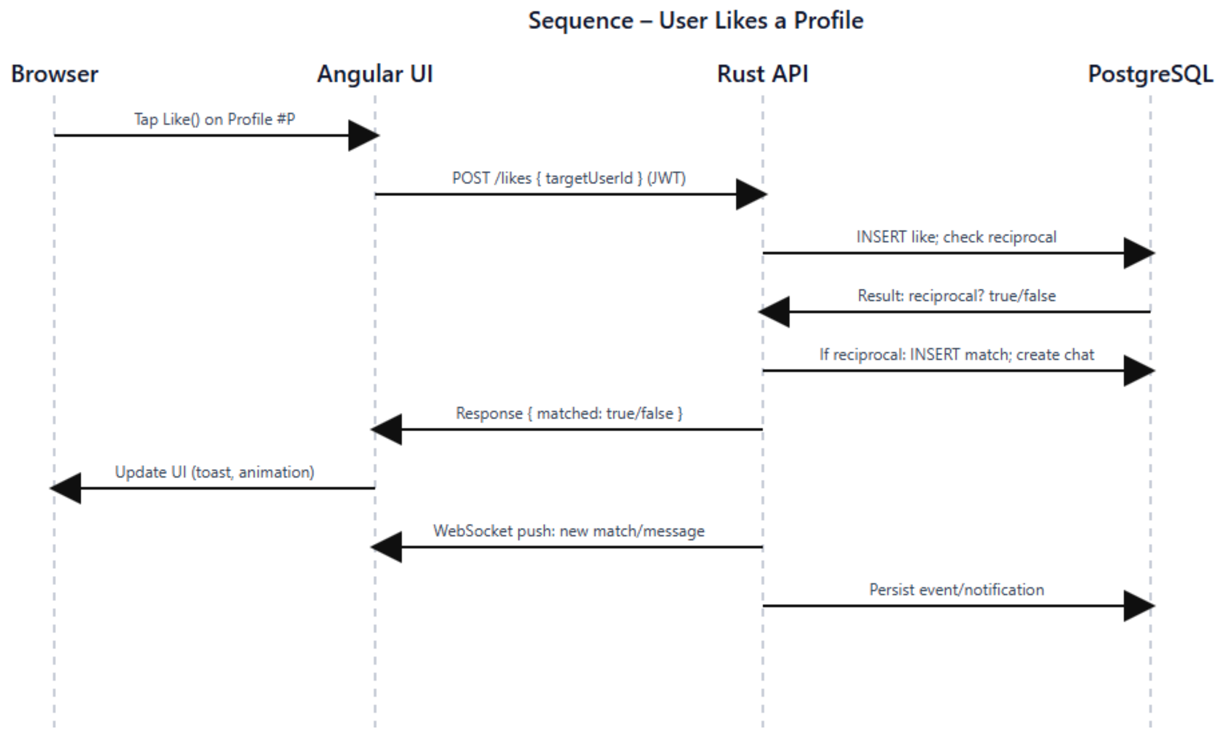
Sequence – User Likes a Profile
Browser
Angular UI
Rust API
PostgreSQL
Tap Like() on Profile #P
POST /likes { targetUserId } (JWT)
INSERT like; check reciprocal
Result: reciprocal? true/false
If reciprocal: INSERT match; create chat
Response { matched: true/false }
Update UI (toast, animation)
WebSocket push: new match/message
Persist event/notification

Diagram 2:This sequence diagram shows how a user “like” action flows through the system. When the user taps the Like button in the browser, the Angular UI sends a POST /likes request with the target user ID and JWT to the Rust API. The API inserts the like into PostgreSQL and checks if it’s reciprocal. If both users have liked each other, the API creates a match and chat entry in the database. The API then returns a response to the UI indicating whether a match occurred. The UI updates the browser with a toast or animation, and if a new match is created, a WebSocket event notifies both users while persisting a match event or notification in PostgreSQL.

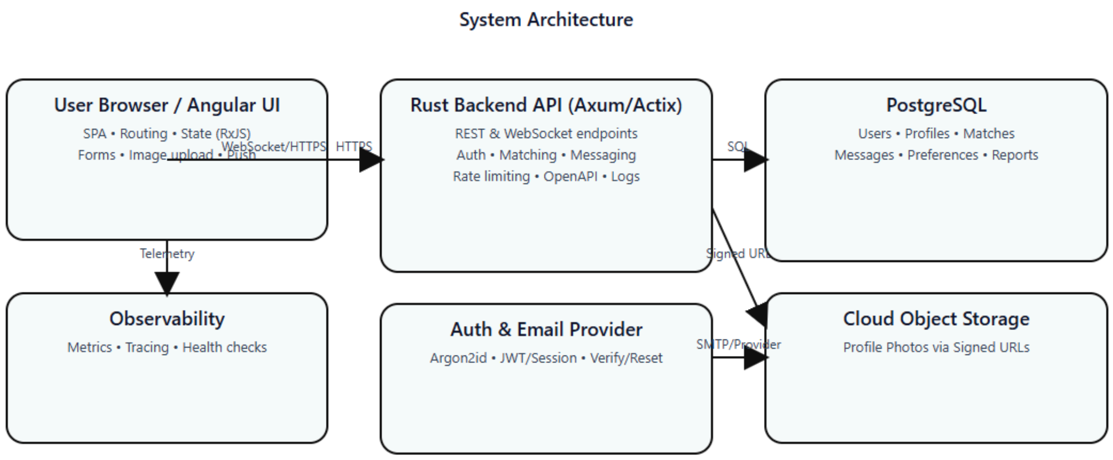
System Architecture
User Browser / Angular UI
SPA • Routing • State (RxJS)
Forms • Image upload • Push
WebSocket/HTTPS
HTTPS
Rust Backend API (Axum/Actix)
REST & WebSocket endpoints
Auth • Matching • Messaging
Rate limiting • OpenAPI • Logs
SQL
PostgreSQL
Users • Profiles • Matches
Messages • Preferences • Reports
Telemetry
Signed URL
Observability
Metrics • Tracing • Health checks
Auth & Email Provider
Argon2id • JWT/Session • Verify/Reset
SMTP/Provider
Cloud Object Storage
Profile Photos via Signed URLs

Diagram 3:This system architecture diagram outlines the core components and data flow of the application. The User Browser/Angular UI serves as the front end, handling routing, state management, forms, image uploads, and push notifications through HTTPS and WebSocket connections. The Rust Backend API (built with Axum or Actix) manages authentication, matching, messaging, and rate limiting, while exposing REST and WebSocket endpoints. It interacts with PostgreSQL for persistent data such as user profiles, matches, messages, and reports, and with Cloud Object Storage for handling profile photos via signed URLs. The backend also communicates with an Auth & Email Provider for user verification, session management, and password resets using Argon2id and JWTs. Finally, Observability components capture metrics, tracing, and health checks to monitor system performance and reliability.

Sprint 1 Requirements List

Requirement ID	Description	Story Points	Priority	Assigned To
R1	Create Angular skeleton with routing and layout structure.	1	1	Ibrahim
R2	Implement KU email login and MFA verification system.	1	1	Maren
R3	Set up backend in Rust with initial API routes and endpoints.	1	1	Suarav
R4	Build SQLite database with user, match, message, and report tables.	3	2	Abdullah
R5	Create backend endpoints for profile creation, update, and fetch.	2	2	Maren

R6	Add like/pass API to support user interactions for matching.	2	3	Saurav
R7	Implement Angular routing guards and page navigation.	2	6	Zack
R8	Test backend endpoints and verify database integration.	5	4	Nick

Total story points for Sprint 1: **17**

Sprint 1 Requirements Artifacts

Each artifact demonstrates the implementation and verification of a Sprint 1 requirement. Artifacts will include screenshots, terminal outputs, and test logs verifying completion.

- **R1:** Screenshot of Angular project structure and initial running build.
- **R2:** Screenshot showing the login form and working MFA verification process.
  R3: Terminal output from `GET /api/health` confirming backend setup.
- **R4:** Screenshot of SQLite database showing `users`, `matches`, and `messages` tables.
- **R5:** Postman results of `/api/profile` CRUD operations.

- **R6:** Screenshot or console log showing like/pass functionality and match trigger.
- **R7:** Screenshot of Angular route configuration file and navigation flow.
- **R8:** Screenshot or PDF summary of API test results verifying database integration.

All artifacts will be placed in a folder.

Non-Functional Requirements

- **NFR1 – Security:** Argon2 password hashing, JWT authentication, and HTTPS encryption.

  NFR2 – Performance: Daily match suggestions generated in under 300 ms.
- **NFR3 – Privacy:** Only verified @ku.edu accounts can register and access the system.
- **NFR4 – Usability:** Fully responsive interface with mobile and desktop compatibility.

Deployment Plan

- **API:** Fly.io or Railway (Docker deployment for Rust backend).
- **Database:** SQLite file stored on backend server with persistence.
- **Frontend:** Angular static build deployed on Vercel with HTTPS enforced.
- **TLS:** Configured through Cloudflare or Vercel’s certificate provider.
- **CI/CD:** Automated build, test, and deployment pipelines via GitHub Actions.