SAPIENZA
UNIVERSITÀ DI ROMA

SwapIt

Laboratory of Advanced Programming

Facoltà di Ingegneria dell'informazione, Informatica e Statistica

Master's Degree in Engineering in Computer Science

Student – Maria Laura Lipari
Matr. 2135594

Student – Giulio Dal Bono
Matr. 2139493

Student – Salvatore Chiacchio
Matr. 2098209

Academic Year 2024/2025

Abstract

This project proposes the development of a distributed platform designed to facilitate peer-to-peer skill exchange, fostering a community centered around mutual learning, inclusivity, and collaboration without spending money.

Users register, sign in, publish the skills they offer and those they want to learn, discover compatible peers through matching and search, start a conversation via integrated chat, and coordinate exchanges directly in messaging. After each session, both sides can leave ratings and reviews to build trust and reputation.

Targeted at everyone interested in learning or sharing abilities at zero cost: students, professionals, hobbyists, newcomers in a city, or anyone curious. The platform helps people connect through shared interests and grow their skills together.

The user should be able to:

- Register as a user of the platform.
- Log in and access a personal area.
- Publish the skills they offer and the skills they want to learn during the registration.
- Search for and discover compatible partners.
- Send/receive a swap proposal, negotiate time slots, and schedule a session.
- See the history of what she/he learned.
- Contact other users through the platform.
- Read and write reviews and see user reputations.

Table of content

Introduction

A distributed software architecture involves designing and implementing software systems where the various functionalities and components are spread across multiple nodes or machines within a network. In this setup, different parts of the software can operate on separate physical machines, virtual servers, or containers, communicating with each other over a network using standard communication protocols such as HTTP, TCP/IP, or other specialized protocols.

The primary objective of this architecture is to improve the system's scalability, availability, and reliability. By distributing the system's components, the load on individual nodes is reduced, and the workload can be balanced across multiple processing resources. This distribution also enhances fault tolerance, ensuring that a failure on a single machine does not compromise the entire system.

A key aim of distributed software architecture is to make the physical distribution of software components invisible to end-users. This means users should not need to know where the different components are running or how they are interconnected. The architecture should offer a seamless interface for accessing the system, irrespective of its physical distribution.

Distributed systems offer several benefits over monolithic, single-server systems:

- Scalability and Flexibility: distributed systems facilitate the addition of computing resources as service demand increases.
- Fault Tolerance: by spreading components across multiple nodes, distributed systems reduce the risk of a single point of failure.
- Reliability: a well-architected distributed system can handle node failures without major performance issues.
- Speed: distributed systems, with their scalability and load balancing, maintain high performance even under heavy loads, ensuring an optimal user experience.
- Geographic Distribution: distributed systems reduce latency and enhance user experience by serving content closer to users, regardless of their location.

Microservices, or microservices architecture, is a method of designing and implementing enterprise applications by breaking down a large application into modular components or services. Each module focuses on a specific task or business objective and interacts with other modules and services through a well-defined communications interface, such as an application programming interface (API). These services are developed independently, each running its own process and typically managing its own database. A service can perform various functions, including generating alerts, logging data, supporting user interfaces (UIs), handling user identification and authentication, and executing other computing and processing tasks.

This architectural approach leverages virtual container and networking technologies extensively. It is

known for facilitating streamlined development, deployment, and scalability of modules, making it particularly well-suited for application development in modern public cloud environments.

Key characteristics of a microservices architecture include:

- Unique Components: services are designed and deployed as standalone components, each fulfilling a specific function or requirement.
- Decentralization: microservices operate with minimal dependencies, though their loose coupling necessitates frequent and extensive communication between components.
- Resilience: resilience requires robust software design, site reliability engineering (SRE) practices, redundant deployments, failover mechanisms, and high scalability techniques.
- API-Based: microservices architecture uses APIs and API gateways to enable communication between services and with other applications.
- Data Separation: each service manages its own database or storage, ensuring data separation.
- Automation: due to the potentially large number of components, microservices rely on automation and orchestration technologies for deployment and scaling, reducing the need for manual intervention.

Docker is a widely used platform that simplifies the development, deployment, and management of applications through containerization. Containers are lightweight, portable, and isolated environments that bundle an application with its dependencies, ensuring consistency across different environments. Docker allows developers to package applications into containers, complete with all necessary libraries and dependencies, making deployment on any Docker-supported infrastructure straightforward. This approach streamlines development, enhances deployment efficiency, and improves scalability and reliability.

Requirements

1. Registration and Authentication

Users must be able to access the platform's services easily, whether they are new or returning.

For new users, the platform provides a straightforward registration flow.

For returning users, the login process is equally simple and fast with their existing credentials.

The platform is designed with user experience in mind, ensuring that both the registration and login sections are easy to find and navigate, minimizing any potential confusion and making the overall experience seamless and efficient.

2. User Profile Management

Registered users can manage their profile settings.

It should be easy to modify personal information submitted during registration, allowing users to correct or update their details if they have changed over time and to curate the user's skills:

- Offered skills;
- Skills to learn.

3. Skill Discovery and Exchange

The core service is to connect people who want to learn with those who want to teach/share skills.

This is facilitated through:

- A search panel.
- Match suggestions that highlight compatible partners based on offered vs. desired skills.
    - You can also base your choice by viewing the profile of the user who offers the skill you are interested in.

A dedicated Activity section provides an overview of historical interactions: past/future swaps.

4. Communication (Integrated Chat)

To facilitate coordination between peers, the platform provides integrated real-time chat.

- Users can message each other in-app without sharing personal contact information.
- Chat history is preserved so users can review what was agreed and follow up after the

exchange.

This approach streamlines coordination and improves privacy and safety.

5. Reviews

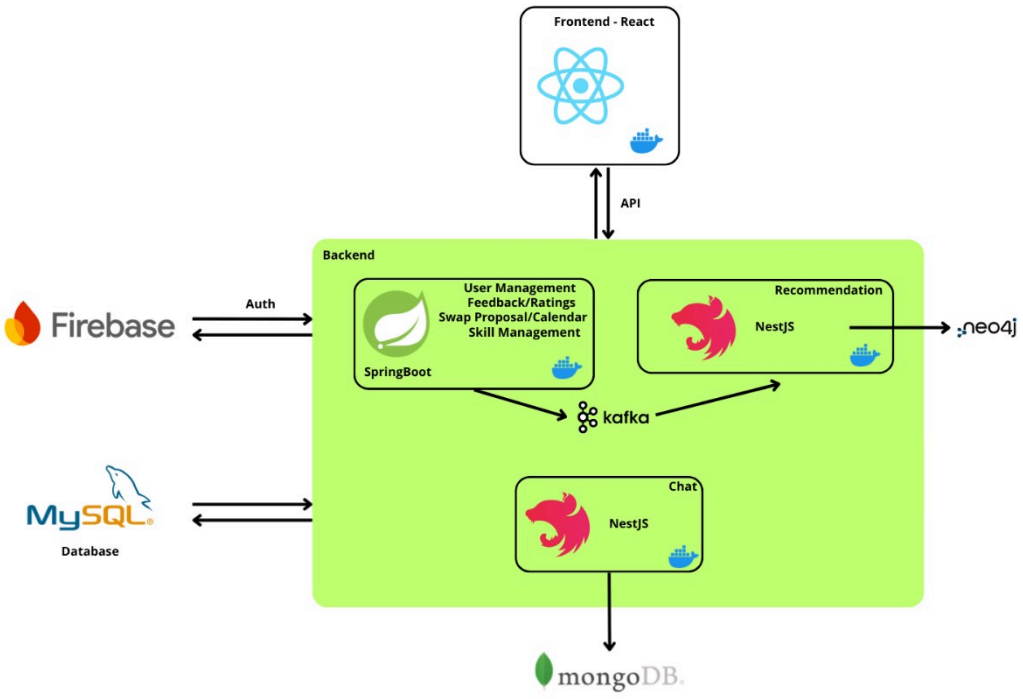
To build trust and provide useful signals about reliability and teaching/learning quality, the platform implements a feedback system.

Each user can post and view reviews about their counterparts after an exchange:

- Reviews include a star rating (1–5) and a short comment;
- Feedback focuses on aspects like clarity, preparedness, punctuality, responsiveness, and overall experience;
- Star rating is also shown in the match card profile;
- A dedicated Activity section provides an overview of historical interactions: submitted/received reviews.

These reviews help learners choose partners confidently and encourage positive, respectful interactions across the community.

Architecture and Components



The system architecture of SwapIt is designed to be a robust, scalable solution for storing skill/profile data and facilitating user interactions. The platform adopts a microservices approach, where each component owns a clear responsibility and communicates over the network using REST (HTTP) and WebSocket (for chat). Asynchronous events are handled via Apache Kafka to decouple services and improve resilience.

Below is an overview of the key architectural components and technologies used:

1. **Client (Web)**
    a. Users interact with the platform through a responsive **web client**.
2. **API Gateway & Interfaces**
    a. An **API Gateway** fronts all backend services for routing, CORS.
    b. **REST APIs (HTTP/JSON)** expose synchronous operations (profiles, skills, proposals, feedback).
    c. **WebSocket** channels enable **real-time chat** between users to coordinate exchanges directly in-app.
3. **Microservices**
    a. **Firebase Authentication:** dedicated for registration/login of a user.
    b. **User, Skill and Swap Management**: dedicated for managing user and skill data, and swap proposals for teaching lessons.
    c. **Recommendation Service**: dedicated for matching algorithms for skills.
    d. **Chat Service**: dedicated for handling users' chat interactions.
    e. **Frontend**: dedicated for ui's creation of the web app and routing between pages

Each microservice runs independently and exposes a well-defined API; cross-service communication uses **REST** and events (**Kafka**) to reduce coupling.

4. **Data Storage**
    a. Per-service databases (**MySQL** schemas) to ensure isolation, independent scaling, and clear ownership.
    b. **Neo4j**: graph database for recommendation system data structure.
    c. **MongoDB**: NoSQL database for message storage
    d. **Redis**: Socket ID storage
5. **Messaging & Event Bus**
    a. **Apache Kafka** for asynchronous workflows and eventual consistency
6. **Ideas for Future Features**
    a. **Availability & Scheduling Helpers (optional)**: Lightweight availability windows and

reminders surfaced inside chat (no external calendars required).

b. **OAuth2 / Social Login**: Integration with Identity Providers (e.g., Google, Facebook) to streamline onboarding.

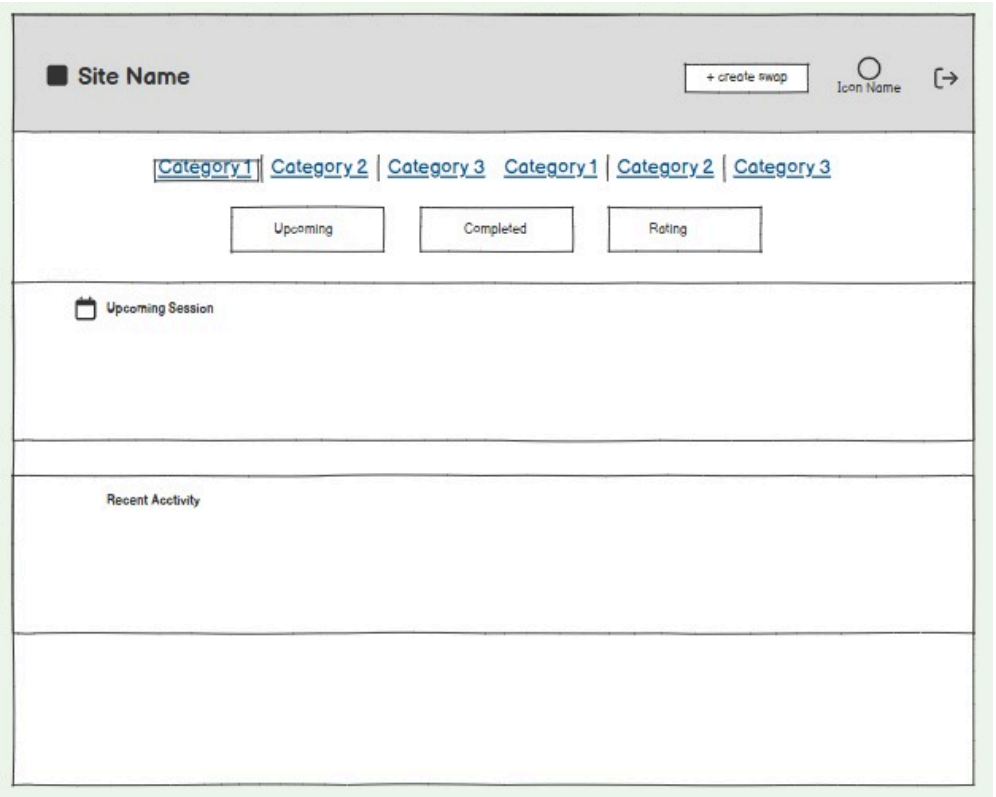
User Stories

1. Registration and Authentication

- **User Story 1.1: Registering a New User**
  - **As a**: New User
  - **I want to**: Create an account and complete a short onboarding to set my interests (skills to learn) and skills I can offer
  - **So that**: I can access the platform's services

- **User Story 1.2: Authenticating an Existing User**
    - **As a:** Registered User
    - **I want to:** Log in using my credentials (e-mail and password)
    - **So that:** I can access my personal area and platform features

- **User Story 1.3: View Home Page**
  - **As a:** Registered User
  - **I want to:** See the home page after login
  - **So that:** I can quickly access matches, skill discovery, active chats, review section, and access my personal area

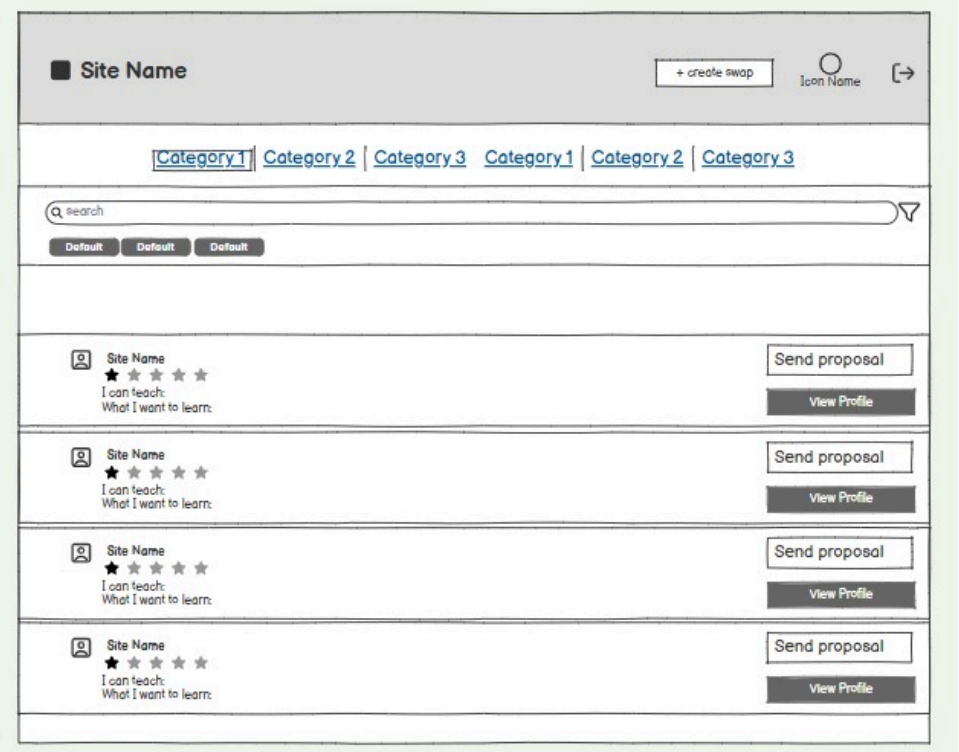


2. User Profile Management

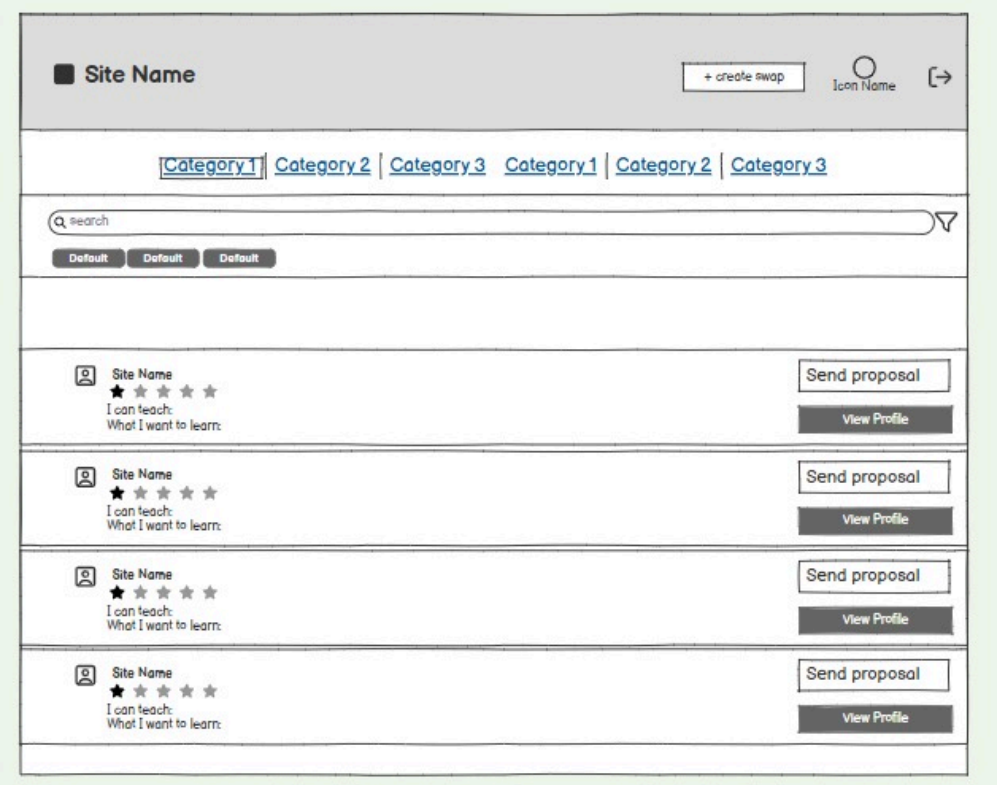
- **User Story 2.1: Updating the User Profile**
  - **As a:** Registered User
  - **I want to:** Update my personal information and manage my skills (offered / to-learn)
  - **So that:** My profile reflects the correct information

3. Swap system

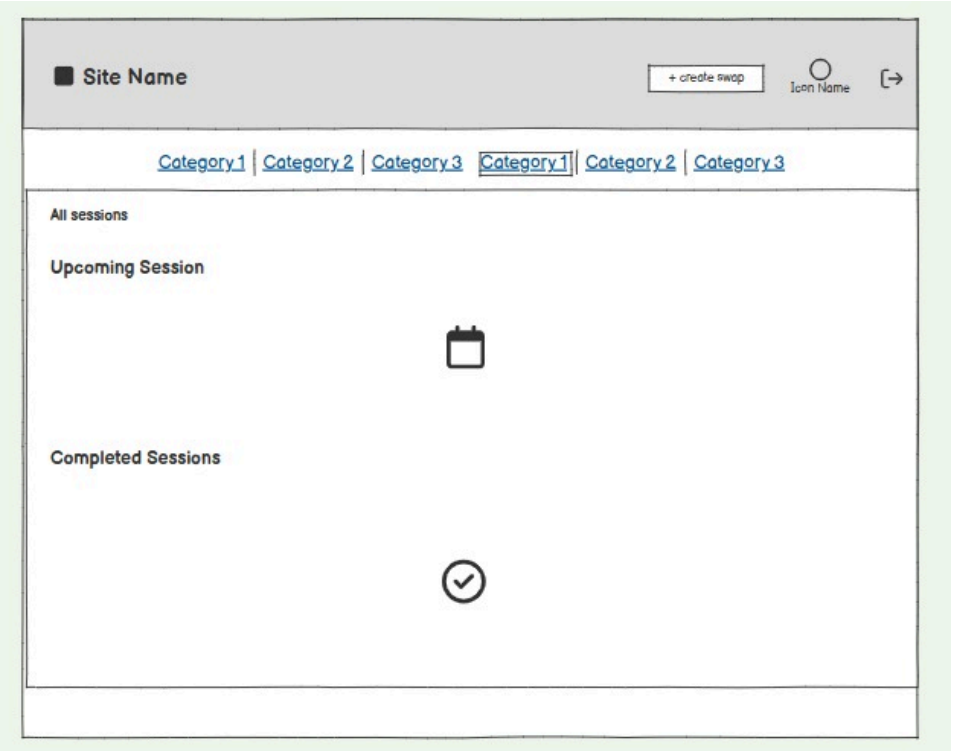
- **User Story 3.1: Searching for a Skill/Partner**
  - **As a:** Registered User
  - **I want to:** Search for skills or compatible partners
  - **So that:** I can find skills of my interest to learn**.**

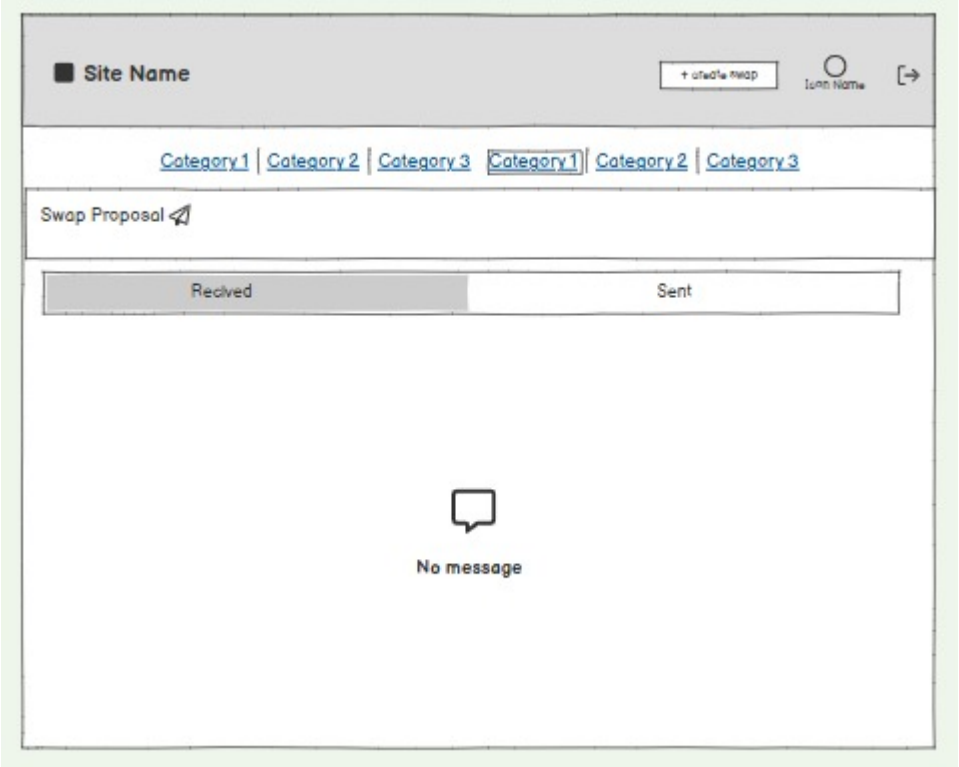


- **User Story 3.2: Match Management**
  - **As a:** Registered User
  - **I want to:** be able to match with a partner who offers the skills that i want to learn
  - **So that:** I can set up a meeting for the first lesson

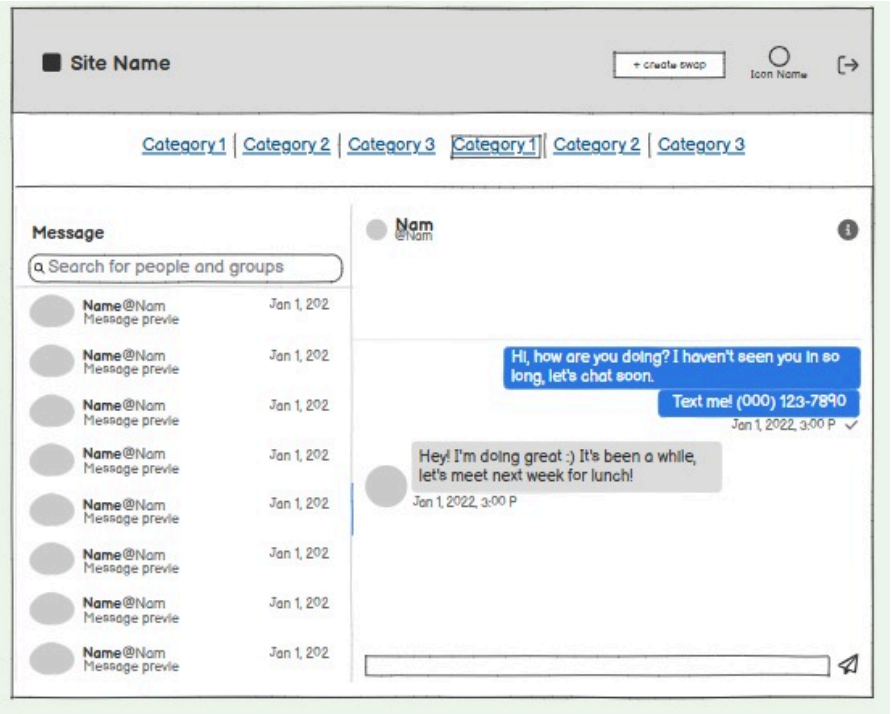


- **User Story 3.3: Viewing the history of exchanges**
  - **As a:** Registered User
  - **I want to:** View the history of my passed match on the platform
  - **So that:** I can keep track of my activities

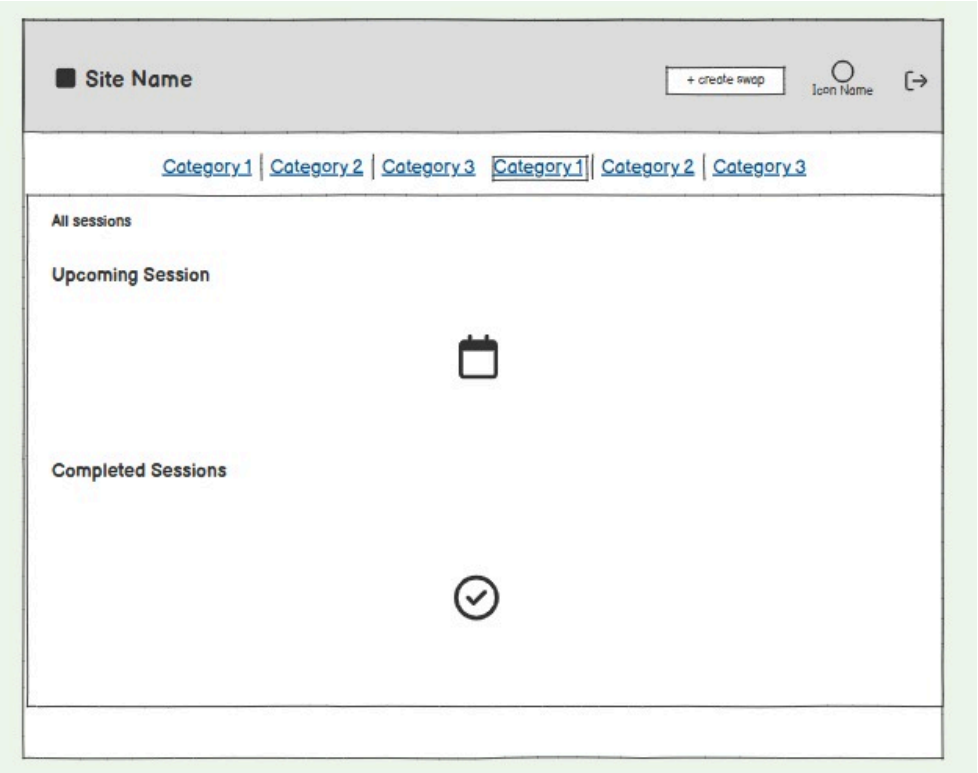




- **User Story 3.4: Chat Management**
  - **As a:** Registered User
  - **I want to:** interact with other user via chat
  - **So that:** I can understand better what they are offering and set up future meetings.

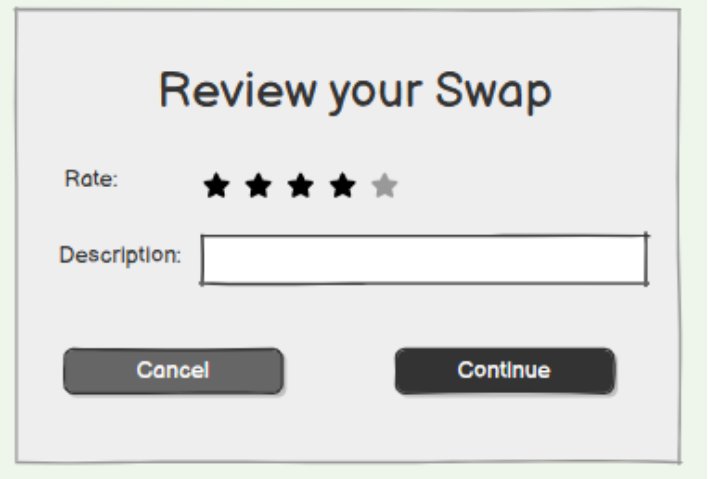


- **User Story 3.5: Upcoming Sessions**
  - **As a:** Registered User
  - **I want to:** see the upcoming sessions in the dedicated page
  - **So that:** I can prepare for the lessons

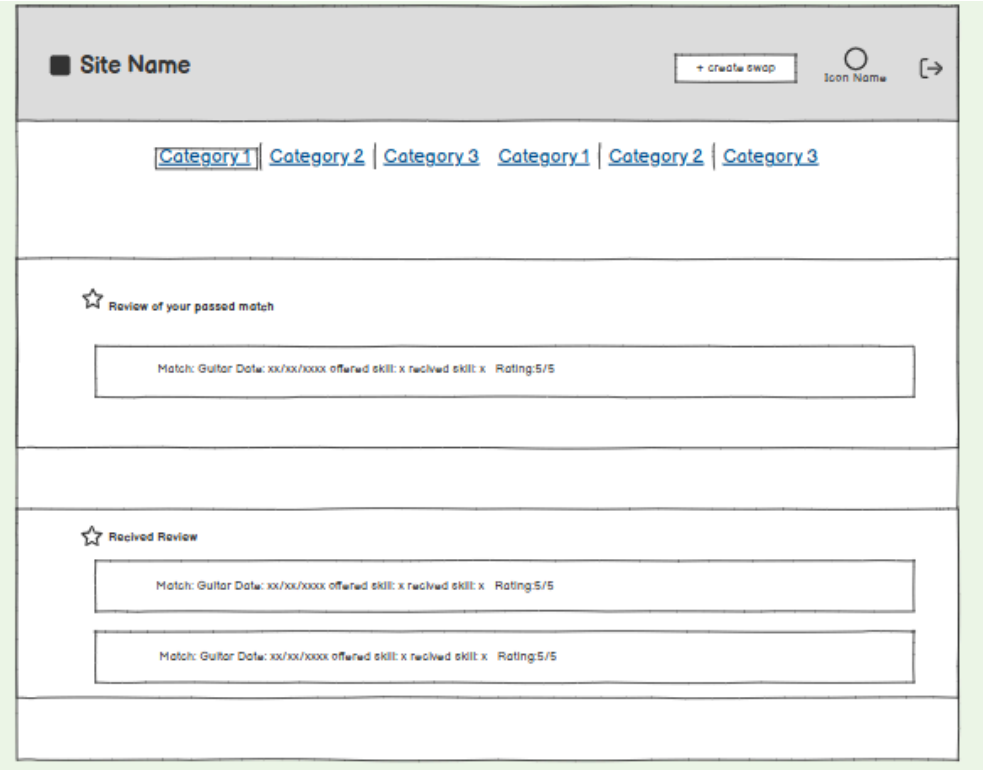


4. Manage Reviews

- **User Story 4.1: Leave Reviews**
  - **As a:** Registered User
  - **I want to:** rating another user's skill and leaving a comment
  - **So that:** other users can visualize my reviews and base their match better.



- **User Story 4.2: Viewing the History of Reviews**
  - **As a:** Registered User
  - **I want to:** View the history of passed review that I received and placed on the platform
  - **So that:** I can easily access and manage my review history.



Schemes of the Microservices' Databases

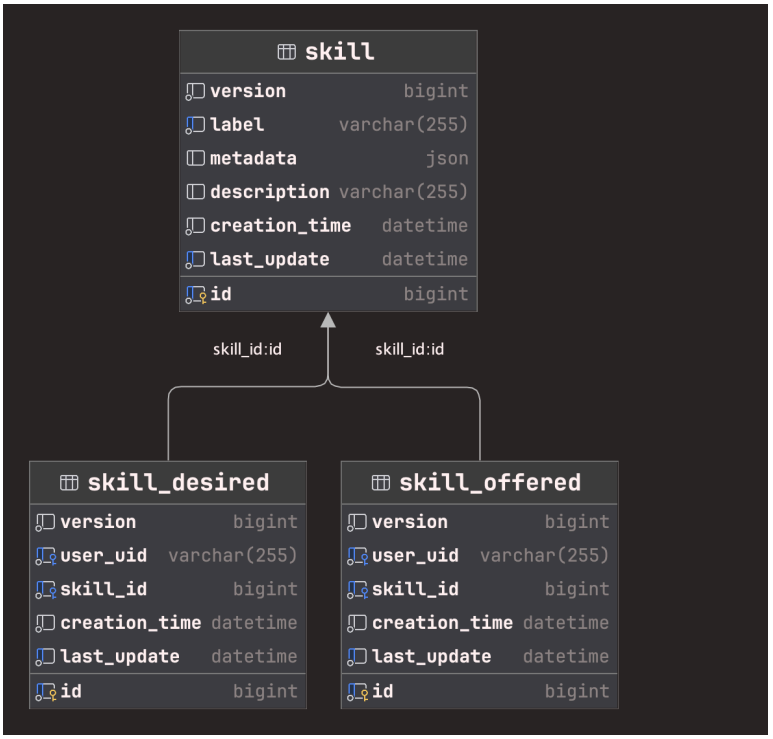
User Microservice

The database for this microservice consists of two tables: **User** and **Feedback**.

The **User** table holds user-related information, including a unique uid (the primary key), email password and a profile picture. This structure allows for well-defined user profiles within the database.

The **Feedback** table is designed to store user reviews about another specific user. It features a unique identifier for each review (primary key), references to the user who authored the review through **reviewer Id**, and a score to quantify the review.

Skill Microservice



The database schema for this microservice consists of three main entities: **Skill**, **Skill Desired** and **Skill Offered**.

The **Skill** class represents a skill sharable by a user, with attributes including a unique identifier (**id**), **label**, **metadata** and a brief **description**, and the **username** of the user that shared it on the platform.

The **Skill Desired** represents the link between the users and the skill, in particular the skills a user is willing to learn.

As for Skill Desired, **Skill Offered** represents the skills a user is willing to teach.

Swap Proposal Microservice

The **SwapProposal** class represents a proposal for a match of teaching lesson about the skills indicated. As attributes we include a unique identifier (**id**), **date**, **start_time**, **end_time** of the lesson, **presentation_letter** and **status**.

Microservices

1. User, Skill and Swap Management

This microservice is implemented in Kotlin with Spring Boot and provides a skill-swap platform featuring users, skills, swap proposals, feedback, and Google Calendar integration. It uses Spring Web for REST APIs, Spring Data JPA with MySQL for persistence, and Liquibase for database migrations. OpenAPI/Swagger documents the APIs, and Actuator is enabled for application insights. Spring Cloud Stream Kafka is included for eventing (e.g., feedback producer). Tests use JUnit 5, Mockito-Kotlin, and H2.

Primary entities include **User** (uid, email, username, profile picture), **Skill** (label, description, JSON metadata), **SkillDesired**, **SkillOffered**, **SwapProposal** (date/time window, presentation letter, status, requester/offerer, skills), and **Feedback** (rating, review, reviewer/reviewed). Controllers expose CRUD and lookup operations:

- **User**: list, get by uid/email, create, update, delete.
- **Skill**: manage skills and desired/offered associations.
- **SwapProposal**: create, update, list, get by id, manage proposal lifecycle.
- **Feedback**: create and retrieve feedback for users.
- **Google Calendar**: list events and create events using OAuth2 with reminders.

Requests and responses use DTOs with explicit mappers, avoiding direct entity exposure. Configuration sets context path `/SwapItBe`, Swagger at `/swagger-ui.html`, and MySQL connection details. JSON is the response format across endpoints.

2. Recommendation service

This microservice is implemented in NestJS/TypeScript and exposes a recommendation API on port 3002 that leverages Neo4j graph analytics and Kafka event ingestion to propose swap partners for SwapIt users.
It runs as a hybrid HTTP + Kafka consumer application: REST endpoints are served through Nest's HTTP server with Swagger UI at /api, while a Kafka microservice listens to SwapIt backend topics (SkillEvent, FeedbackEvent, SwapProposalEvent) to keep the recommendation graph consistent.

Core responsibilities include:

- Consuming CloudEvent-compatible messages to mirror SwapIt domain changes into a Neo4j graph, creating User and Skill nodes plus OWNS, DESIRES, RATES, and SWAPPED_WITH relationships.
- Executing multi-level recommendation queries: a level-2 traversal prioritizes users who successfully swapped with the requester's prior partners; a fallback pipeline covers skill matching, popularity, and recency.
- Enriching graph hits with live user and skill detail through the SwapIt backend (SWAPIT_BE_URL), ensuring DTO-based responses without exposing raw Neo4j records.
- Serving REST operations: GET /recommendations/swaps/:userUid (with optional limit) returns

enriched recommendations and rationales, while GET /recommendations/graph/stats exposes graph totals.

- Operating against configurable infrastructure via .env (Neo4j host/credentials, Kafka brokers, backend URL, service port) with Docker recipes that join the existing environment network and health checks for both Neo4j and the service itself.

Important modules:

Neo4jModule wraps driver lifecycle, running parametrized Cypher via GraphRepository.

KafkaModule wires Nest microservices transport with a CloudEventInterceptor to normalize JSON/CloudEvent payloads.

GraphModule centralizes node/relationship primitives plus Cypher queries, including the scoring cascade.

Tooling & ops:

Tests rely on Jest (npm run test, test:e2e, test:cov); npm run build compiles TS for production.

Docker packaging (Dockerfile + docker-compose.yml) binds port 3002, injects environment, and depends on a healthy Neo4j container; Kafka connectivity is configured for multi-broker strings.

Logging announces broker targets and subscription topics at startup, aiding deployment diagnostics.

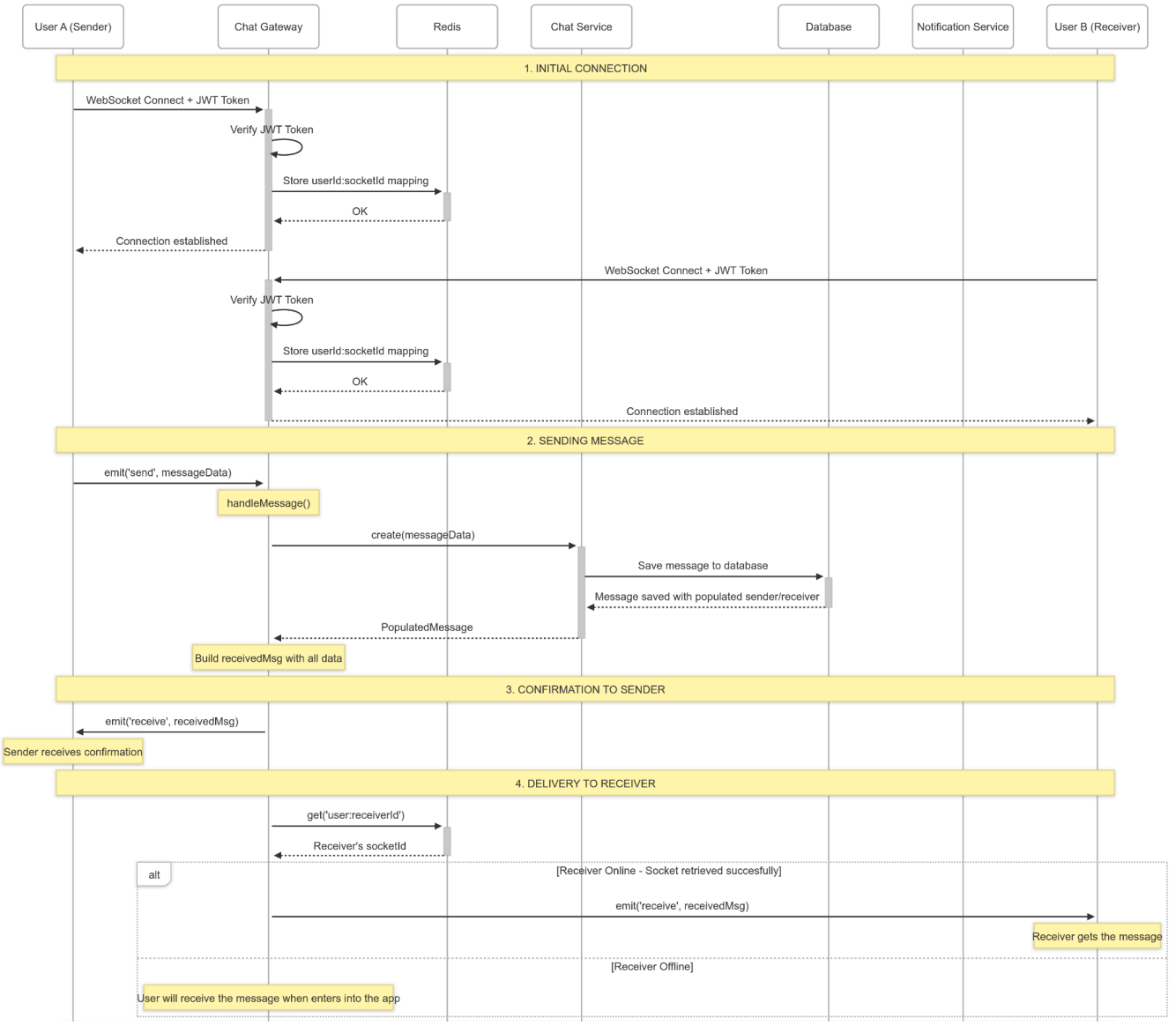
Overall, the service keeps recommendations reactive to user activity, combining graph traversal with backend enrichment to deliver actionable swap suggestions and graph insights. Next steps typically involve running npm run start:dev locally (after copying env.example to .env) or npm run docker:up alongside the SwapIt infrastructure.

Elasticsearch for advanced search capabilities, the current implementation employs SQLite's built-in querying capabilities, which efficiently searches titles and authors.

3. Chat service

This microservice implements a realtime chat backend and accompanying REST APIs using NestJS (TypeScript). A Socket.IO gateway handles live events (connect/handshake with Firebase ID token, sendMessage, typing, markRead, presence), while HTTP controllers provide persistent operations for conversations, message history, uploads and read receipts.

Messages and conversations are persisted in MongoDB via Mongoose schemas (e.g., message.schema.ts), and Redis is used both as the Socket.IO adapter for cross-instance delivery and as a transient store for presence and typing state. Authentication is handled by verifying Firebase tokens (server-side guard such as **firebase-auth.guard.ts**), and the verified Firebase UID is the trusted identity for all actions. Core domain objects are User (Firebase uid, email, display name, avatar, status/lastSeen), Conversation (id, type direct/group, participants, title, lastMessage) and Message (id, conversationId, senderUid, content, type, attachments, createdAt, editedAt, readBy, metadata).



Typical flows: a client connects with a token, joins a conversation room, emits sendMessage with an optional client tempId, the server validates participant rights, persists the message, publishes it via Redis so all instances broadcast to their sockets, and returns an ACK with the definitive id and timestamps. REST endpoints support listing and retrieving conversations, paginated message history, message CRUD, uploads (multipart, storing to local uploads or delegating to external object storage), and marking messages as read. Security includes Firebase auth, participant authorization checks, content sanitization, and suggested rate limiting to mitigate spam. For deployments the service is containerized with Docker and can be composed with MongoDB and Redis; configuration is driven by environment variables (MONGO_URI, REDIS_*, Firebase credentials, UPLOAD_DIR, PORT). Testing is done with Jest: unit tests for services and guards, Supertest for REST, and Socket.IO client tests for gateway flows. Operational considerations include offline delivery and push notifications (FCM), idempotency for duplicate sends (tempId + server dedupe), message ordering strategies, file size limits and direct-to-cloud uploads for large files, and retention/archival policies for compliance.

4. Frontend

SwapIt's frontend is a React & Next.js written in TypeScript and styled with shadcn/ui plus Tailwind utilities.

The root layout wraps all routes in an AuthProvider, letting the app flip between the marketing landing page and the authenticated dashboard. That provider manages Firebase email/password auth, fetches user profiles from the REST backend at http://localhost:8080/SwapItBe/api, normalizes avatars, assembles offered and desired skill lists, computes rating and completed swaps, and starts a Socket.IO session with the chat service once sign-in succeeds. Registration happens through AuthModal, which collects account data, optional profile photo, and initial skills; the provider creates the Firebase user, mirrors it in the backend, spawns missing skills, links them to the user, and caches the resulting state. Profile edits run a differential sync so only changed skills get added or removed.

Networking is centralized, whose ApiClient wraps fetch with safe handling of empty responses and typed DTOs for users, skills, swap proposals, feedback, and calendar events.

The companion useApiCall hook yields data/loading/error so components stay reactive. Chat support is split: lib/chatApi.ts covers REST calls, while lib/chatClient.ts maintains a singleton socket.io-client bound to the current Firebase ID token. ChatSection consumes both layers,loading historical threads, mapping user data, attaching websocket listeners, rendering the conversation UI, and swapping optimistic messages with server acknowledgements to remove duplicates or temp IDs.

The main Dashboard arranges content across tabs: "Home" surfaces stats, "Matches" renders SkillMatcher, "Proposals" and "Sessions" reuse backend data enriched with user/skill info, "Chat" mounts ChatSection, "Profile" reuses ProfileSetup, and "Review History" collates completed swaps with feedback given/received. Tab selection persists in localStorage, modal flows support new swap creation and ratings, and data fetches are reused to limit backend load. Endpoint constants are hard-coded and should migrate to environment variables for production; Firebase config in lib/firebase.ts also needs secure provisioning.

Deployment uses a multistage Dockerfile: install dependencies with npm ci, build Next.js, prune dev packages, copy the standalone output, run as a non-root user, and start via node server.js. docker-compose.yml exposes the app on port 3000 with.

DOCKER

Docker is a platform that allows for the automation of application deployment within containers. Containers are lightweight and isolated environments that can run applications consistently, regardless of the underlying operating system. This approach simplifies dependency management, portability, and scalability of applications. Docker uses an image-based model: an image is a complete package that contains everything needed to run an application, including code, libraries, and settings. To create an image Docker uses a Dockerfile: is a text file that contains a set of instructions for building a Docker image. It defines the environment for your application, including the base image to use, the software packages to install, configuration settings, and the commands to run when the container starts. Here's our Chat service Dockerfile description :

```
# --- Build stage ---
FROM node:18.20.8-alpine AS builder
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# --- Production stage ---
FROM node:18.20.8-alpine AS production
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install --omit=dev
COPY --from=builder /usr/src/app/dist ./dist
COPY --from=builder /usr/src/app/firebase ./firebase
COPY --from=builder /usr/src/app/serviceAccountKey.json ./serviceAccountKey.json
EXPOSE 3000
CMD ["node", "dist/main.js"]
```

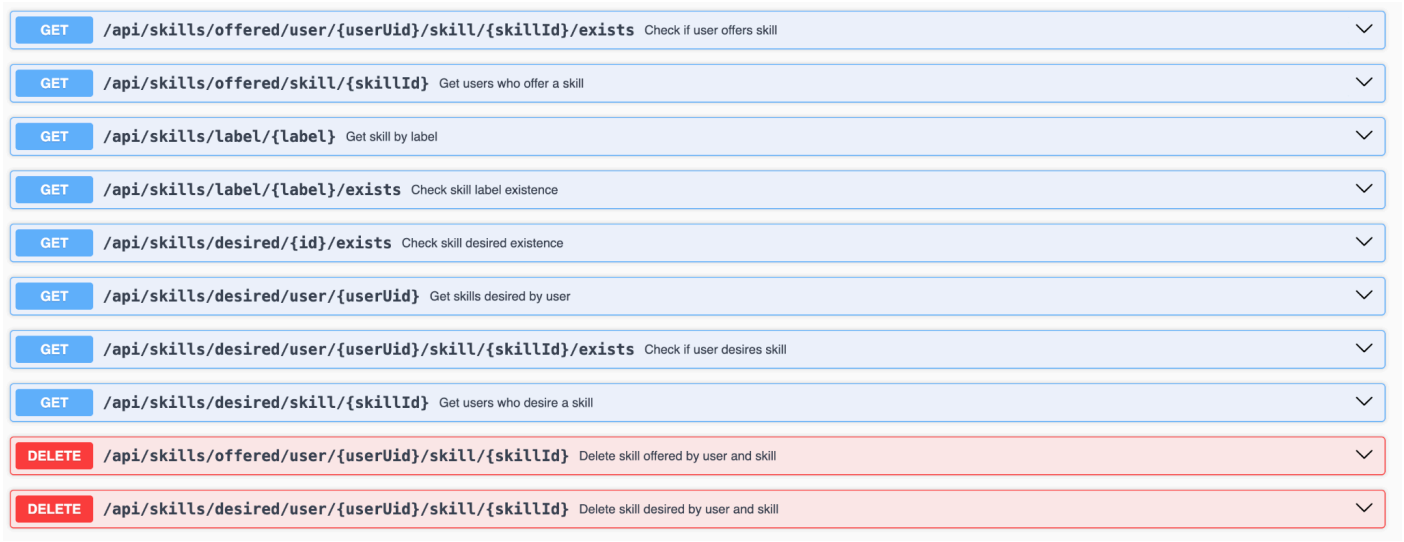
REST Interface Endpoints:

User Management

Endpoints

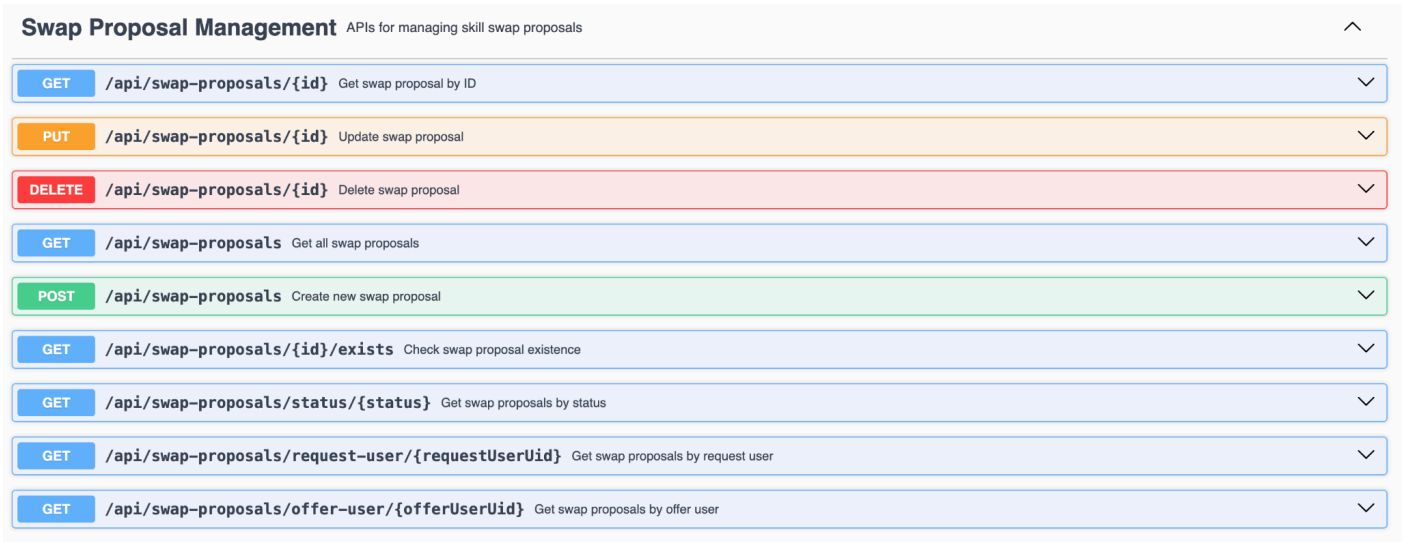
Skill Management

Endpoints



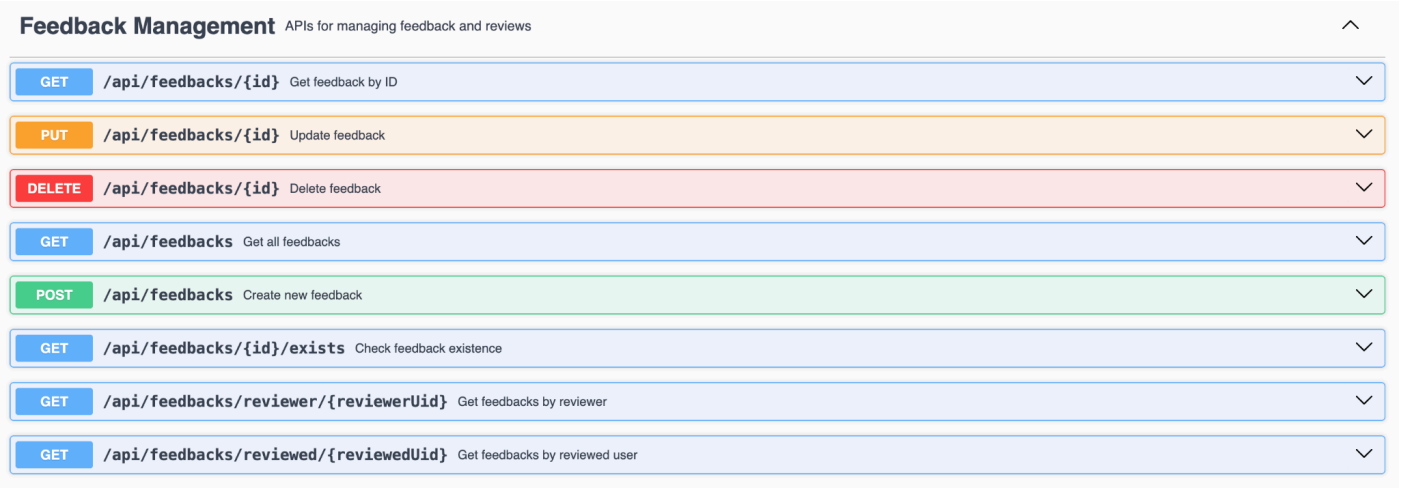
Swap Proposal Management

Endpoints

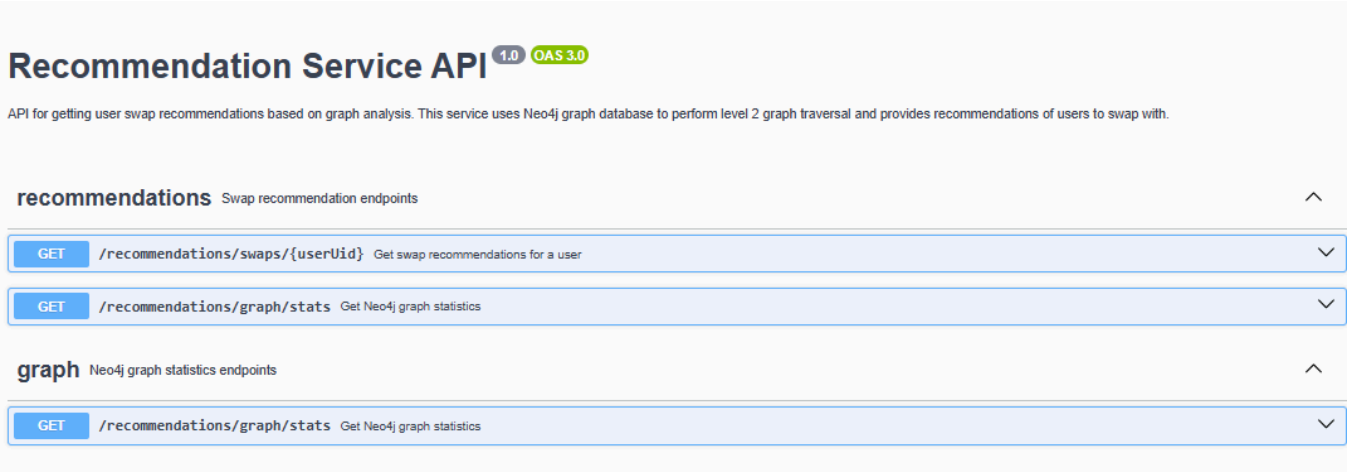


Feedback Management

Endpoints



Recommendation Service



Authentication

The authentication service of our chat system ensures that only verified users can access and exchange messages securely. We implemented user authentication using **Firebase Authentication** combined with **JSON Web Tokens (JWT)** to manage identity verification and session handling. When a user logs in or registers, Firebase generates a unique JWT containing the user's credentials and identification data, which is then sent to the client. This token is attached to subsequent requests to the chat server, allowing the system to verify the user's authenticity without repeatedly asking for login details. On the server side, we have a microservice called NestJs which is used to validate these JWTs and protect our chat endpoints and of course to create the socket system to allow users to communicate.

Technologies

To build Swapit, we selected a modern and flexible technology stack that covers frontend, backend, data persistence, messaging, and deployment. At the frontend we use **React** with **Next.js**, which enables highly interactive user interfaces and a smooth user experience. The backend is powered by a combination of **Spring** (for the **Kotlin** ecosystem) and **NestJS** (for the **Node.js** ecosystem) , this dual-backend approach gives us flexibility in choosing the best tool for different services and promotes a micro-services or modular architecture.

On the data layer we opted for multiple datastores, each chosen for its specific strengths: **MySQL** serves as our relational database for structured transactional data, **MongoDB** handles unstructured or semi-structured document-style data with high scalability, and **Neo4j** is used when graph relationships matter , for example in matching and recommendation features. For real-time event streaming and messaging we rely on the publish-subscribe system of **Apache Kafka**, which allows decoupled services to communicate reliably at scale.

To streamline deployment, environment consistency, and scalability we containerize all components using **Docker**. This ensures that each service (frontend, backends, datastores, Kafka brokers) can run in its own isolated environment, simplifying development, testing and production operations. Finally, we integrate **Firebase** to manage authentication , leveraging its ready-made infrastructure to accelerate development of those cross-platform features.

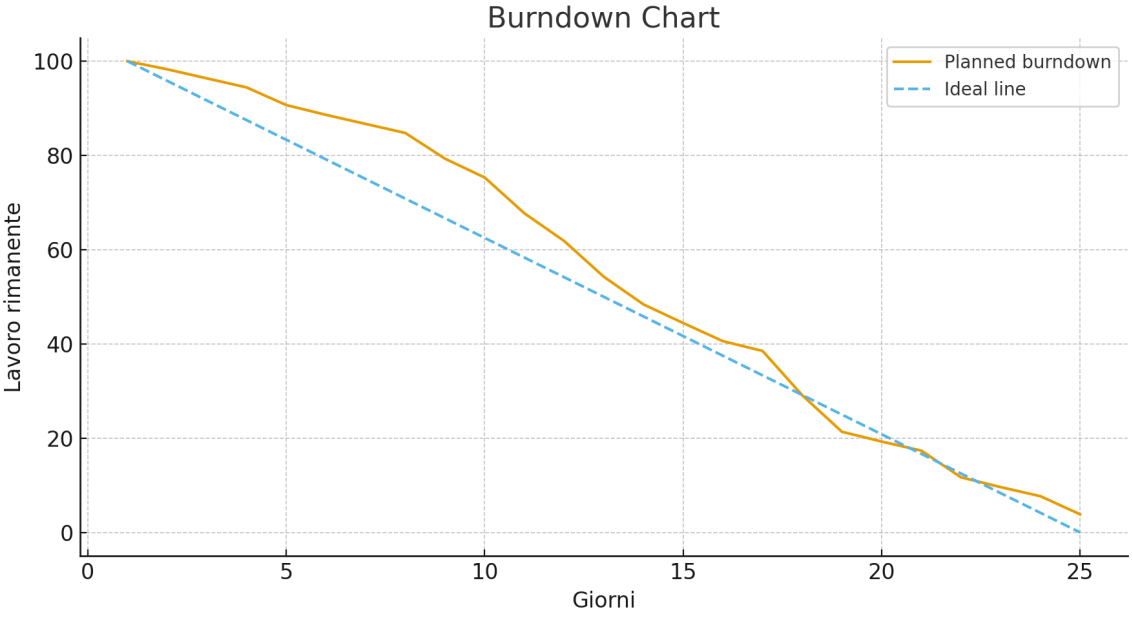
Together, these technologies form a cohesive and scalable foundation for Swapit. They allow us to move quickly, to support different types of data and interactions, and to evolve the system over time. By combining React, Spring, NestJS, MySQL, MongoDB, Neo4j, Kafka, Docker and Firebase, we ensure that the architecture is robust, future-proof and capable of adapting to changing requirements.

Scrum

SCRUM PROJECT MANAGEMENT GANTT CHART
GANT CHART AND BURNDOWN

SPRINTS
SPRINT 1
SPRINT 2
SPRINT 3
SPRINT 4
SPRINT 5

VBS	Task Title	Task Owner	stimate (hr	Start	Due Date	uration (days	Sprint	Status	Compl
1	**Project definition**		8						
1.1	Define user stories	Entire team	4	2025-06-09	2025-06-10	2	1	Completed	100%
1.3	Define architecture	Entire team	2	2025-06-11	2025-06-12	2	1	Completed	100%
1.4	Define technology stack	Entire team	2	2025-06-12	2025-06-13	2	1	Completed	100%
2	**User registration and authentication**		9						
2.1	Implement user login (frontend)	Salvatore Chiacchio	2	2025-06-15	2025-06-15	1	2	Completed	100%
2.2	Implement user registration (fro	Salvatore Chiacchio	2	2025-06-16	2025-06-16	1	2	Completed	100%
2.3	Develop user registration and	Giulio Dal Bono	2	2025-06-17	2025-06-17	1	2	Completed	100%
2.4	Connect DBMS to handle user	Giulio Dal Bono	3	2025-06-17	2025-06-17	1	2	Completed	100%
3	**Swapit dashboard**		14						
3.1	Implement page for match (fron	Salvatore Chiacchio	4	2025-06-21	2025-06-22	2	2	Completed	100%
3.2	Search & filters endpoint (backe	Giulio Dal Bono	2	2025-06-22	2025-06-23	2	2	Completed	100%
3.3	Develop match management s	Giulio Dal Bono	4	2025-06-23	2025-06-25	3	2	Completed	100%
3.4	Implement SQL queries for skill	Giulio Dal Bono	2	2025-06-23	2025-06-25	3	2	Completed	100%
3.5	Manage skills data in the datab	Giulio Dal Bono	2	2025-06-23	2025-06-25	3	2	Completed	100%
4	**Account management**		8						
4.1	Implement account manageme	Salvatore Chiacchio	3	2025-06-25	2025-06-26	2	2	Completed	100%
4.2	Implement account manageme	Giulio Dal Bono	3	2025-06-30	2025-07-02	3	2	Completed	100%
4.3	Develop API calls for informati	Giulio Dal Bono	2	2025-06-30	2025-07-02	3	2	Completed	100%
5	**User Reviews and Rating**		8						
5.1	Implement reviews page for a sp	Salvatore Chiacchio	3	2025-08-17	2025-08-18	2	3	Completed	100%
5.2	Implement sql query for reviews	Giulio Dal Bono	2	2025-08-19	2025-08-20	2	3	Completed	100%
5.3	Develop API calls for contact d	Giulio Dal Bono	3	2025-08-19	2025-08-20	2	3	Completed	100%
6	**Chat development**		4						
6.1	Implementchat page (frontend)	Salvatore Chiacchio	1	2025-08-19	2025-08-20	2	4	Completed	100%
6.2	Implementchat page (backend)	Giulio Dal Bono	2	2025-08-19	2025-08-22	3	4	Completed	100%
7	**API testing**		2						
7.1	Test API calls to backend	Giulio Dal Bono	2	2025-09-20	2025-09-20	1	5	Completed	100%
8	**Docker Compose Implementation**		3						
8.1	Docker Compoze integration	Giulio Dal Bono	3	2025-09-20	2025-09-20	1	5	Completed	100%
9	**Documentation**		7						
9.1	word with user stories, estimate	Maria Laura Lipari	5	2025-10-28	2025-11-01	5	5	Completed	100%
9.2	Slides	Maria Laura Lipari	2	2025-10-31	2025-11-01	2	5	Completed	100%



SCRUM PROJECT MANAGEMENT GANTT CHART
RELEASE BACKLOG

Priority	Task Title	Task Description	Task Owner	uration (day:	Sprint	Status
	Project definition					
Middle	Define user stories	Capture requirements as user stories with clear accept	Entire team	1	1	Completed
Middle	Define architecture	Specify distributed architecture, service boundaries, c	Entire team	1	1	Completed
Middle	Define technology stack	Choose FE/BE/DB/tools (versions & libraries) and justifi	Entire team	1	1	Completed
	User registration and authentication					
High	Implement user login (frontend)	Build sign-in UI with validation, error states, and auth A	Salvatore Chiacchio	3	2	Completed
High	Implement user registration (frontend)	Build sign-up flow with password strength, consent, ar	Salvatore Chiacchio	1	2	Completed
High	Develop user registratyion and auth (backend)	Implement sign-up/login endpoints, password hashing	Giulio Dal Bono	4	2	Completed
High	Connect DBMS to handle user information (backend)	Model user tables, migrations, repositories, indexes, ar	Giulio Dal Bono	2	2	Completed
	Swaplt dashboard					
High	Implement page for match (frontend)	UI for suggested matches (cards, quick actions: open	Salvatore Chiacchio	2	2	Completed
High	Search & filters endpoint (backend)	REST endpoint for skill search with filters, paging, and	Giulio Dal Bono	2	2	Completed
High	Develop match management system (backend)	Matching service computing compatibility and recomn	Giulio Dal Bono	2	2	Completed
High	Implement SQL queries for skill search (backend)	Optimized, parameterized queries (indexes/FTS where	Giulio Dal Bono	2	2	Completed
High	Manage skills data in the database (backend)	Schema & CRUD for offered/desired skills, levels, tags	Giulio Dal Bono	2	2	Completed
	Account management					
High	Implement account management page (frontend)	Profile edit UI (bio, avatar upload, languages, timezone	Salvatore Chiacchio	1	2	Completed
High	Implement account management page (backend)	Read/update profile endpoints with validation and perr	Giulio Dal Bono	4	2	Completed
High	Develop API calls for informations update (backend)	PATCH endpoints for partial updates; auditing/consist	Giulio Dal Bono	2	2	Completed
	User Reviews and Rating					
High	Implement reviews page for a spiecifc account (fronter	Read/write reviews UI with average score and paginati	Salvatore Chiacchio	3	3	Completed
High	Implement sql query for reviews display (backend)	Aggregates and listing queries for user reviews.	Giulio Dal Bono	2	3	Completed
High	Develop API calls for contact details (backend)	Expose limited contact info to matched users with priva	Giulio Dal Bono	2	3	Completed
	Chat development					
High	Implementchat page (frontend)	Real-time chat UI (threads, typing, unread), integrated	Salvatore Chiacchio	2	4	Completed
High	Implementchat page (backend)	WebSocket handlers, message persistence, delivery/r	Giulio Dal Bono	2	4	Completed
	API testing					
High	Test API calls to backend	Postman/automated tests for happy paths, edge case	Giulio Dal Bono	1	5	Completed
	Docker Compose Implementation					
High	Docker Compose integration	Compose file + env templates to run FE, BE, DB (and b	Giulio Dal Bono	1	5	Completed
	Documentation					
Middle	word with user stories, estimation FP/COCOMO + scrur	Compile the submission booklet (US, estimates, proce	Maria Laura Lipari	2	5	Completed
Middle	Slides	Prepare the 15-minute presentation deck and demo s	Maria Laura Lipari	2	5	Completed

STATUS KEY
Not Started
In Progress
Completed
Completed

WORK ESTIMATE IN HOURS
1
2
4
4
8
16
24
40
80

SCRUM PROJECT MANAGEMENT GANTT CHART
USER STORIES

As a	I want to	So that	Added by	Date addes
New user	create an account and complete a short onboarding to set my interests and skills I can offer	access the platform's services	Entire team	10/06/2025
Registered user	log in using my credentials	access my personal area and platform features	Entire team	10/06/2025
Registered user	see the home page after login	quickly access matches, skill discovery, active chats, review section, and my personal area	Entire team	10/06/2025
Registered user	update my personal information and manage my skills (offered /to-learn)	my profile reflects the correct information	Entire team	10/06/2025
Registered user	search for skills or compatible partners	find skills of my interest to learn	Entire team	10/06/2025
Registered user	be able to match with a partner who offers the skills that I want to learn	set up a meeting for the first lesson	Entire team	10/06/2025
Registered user	view the history of my past matches on the platform	keep track of my activities	Entire team	06/08/2025
Registered user	interact with other users via chat	understand better what they are offering and set up future meetings	Entire team	10/06/2025
Registered user	see the upcoming sessions in the dedicated page	can prepare for the lessons	Entire team	23/07/2025
Registered user	rating another user's skill and leaving a comment	other users can visualize my reviews and base their match better	Entire team	10/06/2025
Registered user	View the history of passed review that I received and placed on the platform	I can easily access and manage my review history.	Entire team	10/06/2025

Estimation (Function Points & COCOMO II)

Firebase Authentication

External Inputs (EI): These are processes where the system receives data from outside (like user actions).

Registration - 1 low input 3 fp
Login - 1 low input 3 fp
Logout - 1 low input 3 fp

tot 9 fp

Total External Inputs: 3

Final Count of Function Points

External Inputs (EI): 3 External Outputs (EO): 0 User Inquiries (UI): 0
Internal Logical Files (ILF): 0 External Interface Files (EIF): 0

tot 9 fp

User, Skill and Swap Proposal Management

External Inputs (EI): These are processes where the system receives data from outside (like user actions).

Create/Update User - 2 average input 8 fp
Create/Update Skill - 2 average input 8 fp
Create/Update Swap Proposal - 2 high input 12 fp

tot 28 fp

Total External Inputs: 6

External Outputs (EO): This is where the system sends data out to the user.

Match details - 1 low output 4 fp
Proposal details - 1 low output 4 fp
Review details - 1 low output 4 fp

tot 12 fp

Total External Output: 3

External Queries:

Read operations User - 1 low input 3 fp
Read operations Skill - 1 low input 3 fp
Read operations Swap Proposal - 1 low input 3 fp

Tot 9 fp

Total External queries: 3

Internal Logical Files (ILF): These are files that are maintained within the system.

User Table - Tracking all users and their information - 3 det
Skill Table - Tracking all skills present in the system - 3 det
Skill Desired Table - Tracking all skills desired by users in the system - 2 det
Skill Offered Table - Tracking all skills offered by users - 2 det
Swap Proposal Table - Tracking all swap proposals planned between users, with relative status - 9 det
Feedback Table - Tracking all reviews made by users - 4 det

Total Internal Logical Files: 6
Tot 23 fp

External Interface Files (EIF): These are files that are used by the system but maintained by another system.

Firebase Auth reference

Final Count of Function Points

External Inputs (EI): 6
External Outputs (EO): 3
External Queries (EQ): 3
Internal Logical Files (ILF): 6
External Interface Files (EIF): 1

tot 28 + 12 + 23 + 9 = 72 fp

Recommendation service

External Inputs (EI): These are processes where the system receives data from outside (like user actions).

Request match generation - 1 average input 4 fp
Refresh recommendation - 1 average input 4 fp

tot 8 fp

Total External Inputs: 2

External Outputs (EO): This is where the system sends data out to the user.

Ranked match list - 1 average output 5 fp

tot 5 fp

Total External Output: 1

External Queries:
Retrieve match results - 1 average input 4 fp

Tot 4 fp

Total External queries: 1

Internal Logical Files (ILF): These are files that are maintained within the system.
Graph skill record - for skills and user recommendation system

Total Internal Logical Files: 1

External Interface Files (EIF): These are files that are used by the system but maintained by another system.
Access to User, Skill and Swap Proposal Management service

Final Count of Function Points

External Inputs (EI): 2
External Outputs (EO): 1
External Queries (EQ): 1
Internal Logical Files (ILF): 1
External Interface Files (EIF): 1

tot 8 + 5 + 4 = 17 fp

Chat Management service

External Inputs (EI): These are processes where the system receives data from outside (like user actions).
Send message - 1 average input 4 fp
Create chat session - 1 average input 4 fp

tot 8 fp

Total External Inputs: 2

External Outputs (EO): This is where the system sends data out to the user.

Deliver message - 1 average output 5 fp

tot 5 fp

Total External Output: 1

External Queries:

Retrieve chat history - 1 average input 4 fp
Retrieve live message - 1 average input 4 fp

Tot 4 + 4 = 8 fp

Total External queries: 2

Internal Logical Files (ILF): These are files that are maintained within the system.

Chat message store - for recording messages
Chat sessions - for saving sessions' keys with redis

Total Internal Logical Files: 2

External Interface Files (EIF): These are files that are used by the system but maintained by another system.

Firebase for authentication

Final Count of Function Points

External Inputs (EI): 2
External Outputs (EO): 1
External Queries (EQ): 2
Internal Logical Files (ILF): 2
External Interface Files (EIF): 1

tot 8 + 5 + 8 = 21 fp

Total Function Points Summary

Microservice	Total FP
Firebase Authentication	9
User, Skill, Swap Proposal Management	72
Recommendation Service	17
Chat Management	21
Grand Total	119

References

https://github.com/giuliodalbono/2139493_swapit