Project Report: Distributed Messaging Web Application

Sami Taider Rayan Hanader
84401097 84401096

December 26, 2024

Introduction

This project is a distributed messaging web application designed to support real-time user interactions such as authentication, private conversations, and group chat messaging. It leverages a modern tech stack, including Docker-Compose, Flask, FastAPI, and MySQL, to achieve scalability and real-time communication. This report provides a detailed explanation of the tech stack, architecture, and challenges encountered during the development process.

Tech Stack Overview

The application uses the following technologies:

Docker-Compose

Docker-Compose is used to containerize and manage the deployment of the various services in the application. It simplifies running multiple components like Flask, FastAPI, and MySQL in isolated containers, ensuring consistent development and production environments.

Flask

Flask is employed for the frontend API. It handles HTTP requests, provides routes for user interactions, and manages WebSocket connections for real-time message delivery.

FastAPI

FastAPI is used as the backend service for processing user actions and managing data. It contains the core business logic, ensuring actions like sending messages, managing group chats, and updating user data are executed effectively.

MySQL

MySQL is the relational database used to persist user data, messages, and group chat information. It provides a structured and reliable storage solution for the application.

Application Architecture

The architecture of the application is designed to ensure modularity, scalability, and real-time communication. The following figure illustrates the code architecture:

```
messagely/
│
├── backend/                      # Backend-related files
│   ├── app/                      # FastAPI application
│   │   ├── models/               # Database models (e.g., User, Message)
│   │   │   ├── message.py
│   │   │   ├── groupchat.py
│   │   │   ├── conversation.py
│   │   │   └── user.py
│   │   ├── routes/               # API routes (endpoints)
│   │   │   ├── conversation_routes.py
│   │   │   ├── groupchat_routes.py
│   │   │   ├── message.py
│   │   │   ├── user_routes.py
│   │   │   └── db_routes.py
│   │   └── database.py           # Configuration file for database
│   ├── main.py                   # FastAPI entry point
│   ├── requirements.txt          # Python dependencies
│   └── Dockerfile                # Dockerfile to build the backend container
│
├── frontend/                     # Frontend-related files
│   ├── static/                   # Static files (CSS, JS)
│   │   ├── css/
│   │   │   └── style.css
│   │   └── js/
│   │       └── chat.js
│   ├── templates/                # HTML files for frontend
│   │   ├── base.html
│   │   ├── home.html
│   │   ├── chat.html
│   │   ├── signin.html
│   │   └── signup.html
│   ├── utils/                    # utility files
│   │   └── api.py
│   ├── Dockerfile                # Dockerfile to build the frontend container
│   ├── requirements.txt          # Python dependencies
│   └── app.py                    # Flask entry point
├── docker-compose.yml            # Docker Compose configuration to link all services
└── README.md                     # Project description and setup instructions
```

Figure 1: Code Architecture

Workflow

1. A user action, such as signing in or sending a message, triggers an HTTP request to the Flask app.
2. The Flask app processes the request, updates the database via MySQL, and broadcasts the relevant message or event to all connected clients via WebSockets.
3. Clients receive the message or event in real-time and update their user interface accordingly.

Challenges Encountered

Developing this application involved several challenges:

WebSocket Implementation

Implementing reliable and scalable WebSocket connections within the Flask application required careful consideration of message handling, and error handling.

Real-time Synchronization

Ensuring that all clients received updates in real-time and maintained consistent application state presented a significant challenge. This was addressed by carefully designing the message broadcasting mechanism and implementing robust error handling for WebSocket connections.

Integration of Flask and FastAPI

Balancing the roles of Flask and FastAPI in the system architecture required careful planning. Flask was chosen for handling frontend routes and WebSocket connections, while FastAPI was designated for backend logic, ensuring a clear separation of concerns.

Conclusion

This project demonstrates the effective use of a modern tech stack to build a scalable and distributed messaging application. Despite the challenges encountered, the integration of Flask, FastAPI, MySQL, and WebSockets resulted in a robust and responsive system capable of handling concurrent user actions and delivering real-time updates.