Project Report: Distributed Messaging Web Application

Sami Taider Rayan Hanader
84401097 84401096

December 25, 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, Kafka, Flask, FastAPI, and MySQL, to achieve scalability and event-driven 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 Kafka, Zookeeper, Flask, FastAPI, and MySQL in isolated containers, ensuring consistent development and production environments.

Kafka

Kafka serves as the backbone for event streaming and message queuing. It enables asynchronous communication between services and facilitates multithreading, allowing multiple users to perform concurrent actions. The application defines a single Kafka topic, `groupchat-events`, to manage all event types.

Flask

Flask is employed for the frontend API. It handles HTTP requests, provides routes for user interactions, and acts as the producer of events to Kafka.

FastAPI

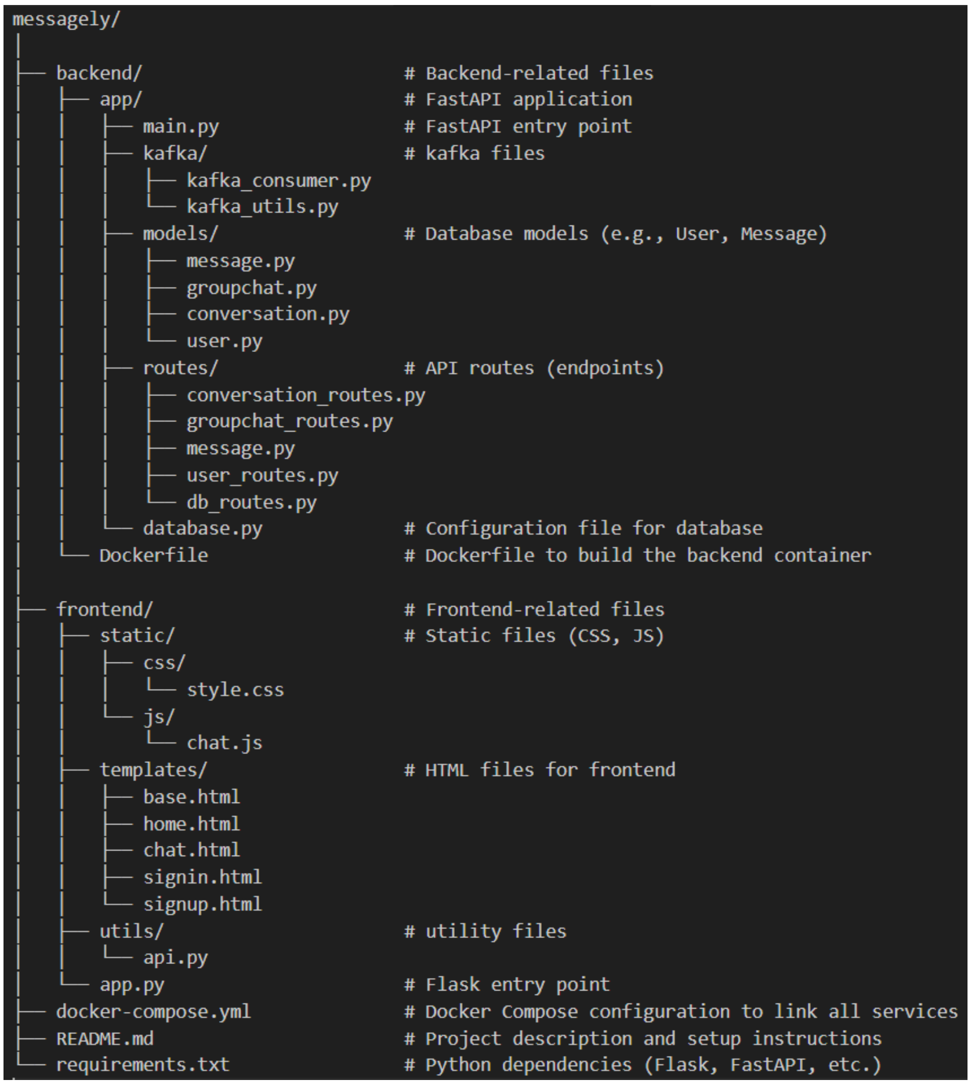
FastAPI is used as the backend service for processing events consumed from Kafka. It contains the core business logic, ensuring actions like sending messages, managing group chats, and updating user data are executed effectively.

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 fault tolerance. The following figure illustrates the code architecture:



```
messagely/
│
├── backend/                      # Backend-related files
│   ├── app/                      # FastAPI application
│   │   ├── main.py               # FastAPI entry point
│   │   ├── kafka/                # kafka files
│   │   │   ├── kafka_consumer.py
│   │   │   └── kafka_utils.py
│   │   ├── 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
│   └── 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
│   └── app.py                    # Flask entry point
├── docker-compose.yml            # Docker Compose configuration to link all services
├── README.md                     # Project description and setup instructions
└── requirements.txt              # Python dependencies (Flask, FastAPI, etc.)
```

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 produces an event to the Kafka topic using the Confluent Kafka Python client.
3. Kafka stores the event in its queue.
4. A Kafka consumer, implemented in FastAPI, polls the queue for new events.
5. The consumer processes the event and updates the database via MySQL.

Challenges Encountered

Developing this application involved several challenges:

Kafka Configuration

Setting up Kafka and Zookeeper locally posed difficulties due to compatibility issues and resource limitations. Errors such as `Access is denied (0x5)` were encountered and resolved by adjusting permissions and configuring environment variables appropriately.

Session Management

Transitioning from synchronous session updates to an asynchronous event-driven model required redesigning user authentication workflows. For instance, ensuring that session variables like `session["user_id"]` were updated correctly after a `signin` event was a significant hurdle.

Concurrency Bugs

Multithreading introduced race conditions and bugs, especially when multiple users attempted to perform actions simultaneously. These issues were mitigated by refining event processing logic and implementing robust error handling.

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, 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 Kafka, Flask, FastAPI, and MySQL resulted in a robust and responsive system capable of handling concurrent user actions.