GO-CHAT: An SOA-based chat app built in Go lang

Submitted by
Nadir Hussain
7171505

To
Dr. Letterio Galletta

For the course of
Distributed Programming for Web, IoT and Mobile Systems

Submitted On
Jan 8, 2024

Table of Contents

Abstract

Modern distributed systems face challenges in ensuring reliable, real-time communication. This project, **GO-CHAT: An SOA-based chat app built in Go lang**, addresses these issues by leveraging Service-Oriented Architecture (SOA) principles to create a modular, scalable, and fault-tolerant chat application. The app consists of decoupled services—Authentication, Messaging, Contacts, Consumer, and API Gateway—communicating seamlessly using **RabbitMQ** as the asynchronous message broker.

The backend is implemented in Go, with **SQLite** using a repository pattern for data management and WebSockets for real-time communication. The frontend employs Go templates for dynamic, server-rendered UIs. Key features include real-time one-to-one messaging, contact management, chunk-based file transfers, emoji reactions, and user management for secure login, registration, and password recovery.

Services are containerized with **Docker** and hosted on **Render** for ease of deployment and scalability. Testing was conducted using **Postman** for API and Socket validation. While, Test cases were written for the Authentication service as part of a learning task, ensuring reliability and compliance with defined functionality. By combining **WebSocket**, **HTTP**, and **AMQP** protocols, **GO-CHAT** delivers an efficient and user-friendly real-time communication solution, demonstrating the practical application of SOA principles.

Introduction

In the era of digital transformation, real-time communication has become a cornerstone of modern applications, facilitating seamless interaction and collaboration among users. Distributed systems play a pivotal role in achieving this by providing scalable and reliable infrastructures. However, these systems face challenges such as ensuring message consistency, managing cross-service communication, and maintaining low latency. Addressing these challenges requires robust architectural designs and implementation strategies.

This project, **GO-CHAT: An SOA-based chat app built in Go lang**, is a practical implementation of a distributed system built with Service-Oriented Architecture (SOA) principles. SOA enables the development of modular, decoupled services, allowing independent scaling and fault tolerance. The application focuses on real-time messaging and notification delivery while ensuring reliability and efficiency.

The system comprises multiple decoupled services, including Authentication, Messaging, Contacts, Consumer, and API Gateway, each fulfilling a specific role. Communication between these services is facilitated by RabbitMQ, an asynchronous message broker. The backend is implemented using Go, leveraging WebSockets for real-time communication and SQLite for data management through the repository pattern. The frontend employs Go templates to deliver a dynamic and server-rendered user interface.

The project incorporates key features such as secure user authentication with JWT and session management using cookies, contact management, real-time messaging, file sharing, emoji reactions, and password recovery. Additionally, Docker containerization ensures smooth deployment, while hosting on Render provides scalability. By combining WebSocket, HTTP, and AMQP protocols, **GO-CHAT** offers an efficient, real-time communication experience that highlights the practical application of SOA principles in addressing modern distributed system challenges.

Design and Architecture

The design of **GO-CHAT** is based on the principles of Software architectures, and System architecture principles. Specifically, it leverages Service-Oriented, Pub-Sub architectures - Software architecture, and Client-Server architecture - System architecture, ensuring modularity, scalability, and fault tolerance. The system is composed of independent services that communicate through RabbitMQ for asynchronous messaging. Each service is designed with a specific purpose, enabling independent scaling, easy debugging, and fault isolation.

System Architecture

The system architecture consists of the following components:

1. **Authentication Service**: Provides secure login, registration, and password recovery using JWT-based, session tokens authentication.
2. **Messaging Service**: Manages real-time messaging using WebSockets and RabbitMQ for asynchronous delivery. Handles chat specific notifications.
3. **Contacts Service**: Handles contact requests, approvals, and user search functionality. Handles contacts specific notifications.
4. **API Gateway**: Acts as the central entry point, routing client requests to the appropriate backend service.
5. **Consumer (Frontend)**: A Go templates-based UI for user interaction.

These components interact using WebSocket, HTTP, and AMQP protocols to deliver a seamless user experience. The complete system architecture can be roughly represented as

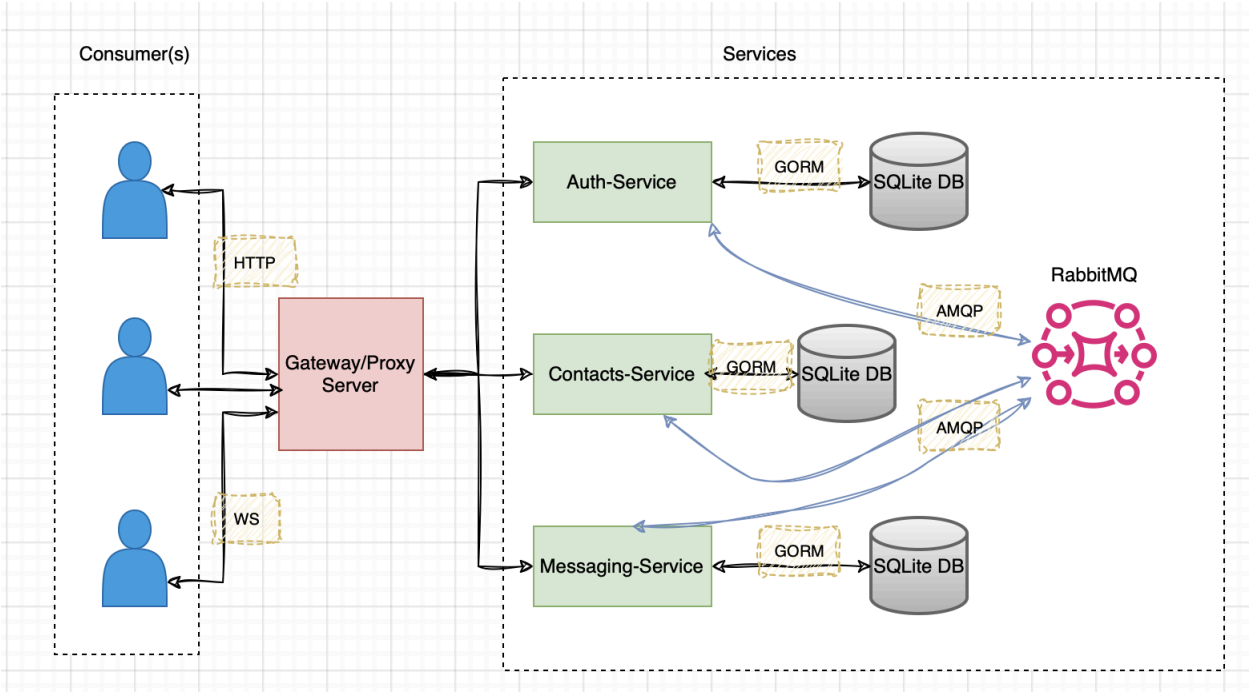


Figure 1.0 : GO CHAT architecture diagram with communication flow and protocols

System Components and Protocols

In this section, I explain each component of the system, the protocol it uses, the data flow to and from that service.

1. Consumer

- **Description**: This is a stand-alone frontend web app built with Go templates that consumes APIs, listens to sockets and makes requests. It leverages Javascript and CSS for smooth user experiences.
- **Protocols Used**:
  - **HTTP**: For REST API-based requests like login, registration, password recovery, contact search requests.
  - **WebSocket (WS)**: For real-time, low-latency communication, such as sending and receiving chat messages, sending and accepting/rejecting requests in real-time, sending and tracking chunk based files with progress.
- **Data Flow**:
  - Users send HTTP requests to perform actions like login or manage contacts.
  - Persistent WebSocket connections are established for real-time messaging, contacts and notification updates.

    - The data flows from the consumer to the **Gateway/Proxy Server**.
    - The Gateway then checks for the intended service, and the protocol request comes with.
    - Then it forwards the request along with headers to the respective service.
    - Upon resolution to request, the response is sent back to the gateway, and from there to the consumer.

2. Gateway/Proxy Server

- **Description**: This serves as a centralized access point, routing client requests to the appropriate backend services.
- **Protocols Used**:
    - **HTTP**: For forwarding client API requests to services like Auth-Service, Contacts-Service, and Messaging-Service.
    - **WebSocket**: For managing real-time communication with the Messaging-Service.
- **Data Flow**:
    - Receives HTTP requests from consumers and routes them to the respective services.
    - For WebSocket-based real-time communication, it establishes a persistent connection with the **Messaging and Contacts Service**.
    - Acts as a proxy, abstracting the complexity of backend services from the consumers.

3. Auth-Service

- **Description**: This service handles user authentication, registration, session management, password recovery and user search along with user details functionality.
- **Protocols Used**:
    - **HTTP**: For secure communication with the Gateway.
    - **AMQP** for listening and publishing to RabbitMQ queues
- **Data Flow**:
    - Receives login/registration or relevant requests from the Gateway.
    - Interacts with SQLite using GORM to SELECT, UPDATE, DELETE from the **User** and **Sessions** tables using repository pattern.
    - Returns a response to the Gateway, which forwards it to the consumer.

4. Contacts-Service

- **Description**: Manages user contacts, including sending, accepting, or rejecting contact requests functionality.
- **Protocols Used**:
    - **HTTP**: For contact-related API interactions such as GetContacts, routed through the Gateway.
    - **AMQP**: For publishing and subscribing to RabbitMQ queues, ensuring asynchronous communication with auth-service.
    - **Websockets**: For real-time contact handling such as sending request, accepting/rejecting requests while sending notification to the target user(s).
- **Data Flow**:
    - Processes HTTP requests from the Gateway to perform contact management operation such as GetContacts.
    - Publishes events; such as DecodeJWT to get logged in user details, GetUsersDetails to fetch details of multiple users; to RabbitMQ, which are consumed by auth-service and responded back with details on queue.
    - Interacts with SQLite using GORM to SELECT, UPDATE, DELETE from the **Contact** and **ContactRequests** tables using repository pattern.

5. Messaging-Service

- **Description**: Manages real-time one-to-one messaging for text and file types between users, and stores message history.
- **Protocols Used**:
    - **HTTP**: To get messages of a user, or download files routed through the Gateway.
    - **WebSocket**: For real-time communication with the Gateway, enabling instantaneous text message delivery, chunk based large file uploads on server, with progress updates to the sender.
    - **AMQP**: For publishing and subscribing to RabbitMQ queues, ensuring asynchronous communication with auth-service.
- **Data Flow**:
    - Real-time chat messages flow through WebSocket connections to the Gateway, which routes them to the Messaging-Service.
    - The service stores messages in SQLite and sends them in real-time if the recipient is online, otherwise queues them.
    - The files are stored on server's file system under /uploads directory, while the meta is still stored in same **Message** table as it is for text

- Acknowledgments or notifications are sent back through the Gateway to the consumer.

6. SQLite Databases

- **Description**: Each service (which needs to) uses its own SQLite database for localized data storage.
- **Protocols Used**:
  - **GORM**: Used to interact with the SQLite databases.
- **Data Flow**:
  - **Auth-Service DB**: Stores **User** credentials and **Session** information.
  - **Contacts-Service DB**: Manages **Contact** lists and **ContactRequests**.
  - **Messaging-Service DB**: Stores **Message**s history and message metadata.

7. RabbitMQ

- **Description**: A message broker used to facilitate asynchronous communication between services.
- **Protocols Used**:
  - **AMQP (Advanced Message Queuing Protocol)**: For publishing and subscribing to queues.
- **Data Flow**:
  - Services like Contacts-Service and Messaging-Service publish events (e.g., GetDetails, JWTDecode) to Auth-service via RabbitMQ.
  - RabbitMQ ensures reliable delivery of these events to subscribed services or consumers.

Technologies and tools Used

The development of **GO-CHAT: An SOA-based chat application** involves a range of technologies, tools, and frameworks that ensure efficiency, scalability, and reliability. Below is a detailed list:

1. Programming Language

- **Go (Golang)**:
    - Used for implementing all backend services and the frontend (server-rendered templates).
    - It was the best choice for this type of application, due to its performance, simplicity, and support for concurrent programming.

2. Backend Communication

- **HTTP (HyperText Transfer Protocol)**:
    - Used for RESTful API communication between the client (via the Gateway) and the backend services. Also, it was used to serve templates from server to client.
- **WebSocket**:
    - Enabled real-time, bidirectional communication for low-latency features like messaging, contact updates and notifications.
- **AMQP (Advanced Message Queuing Protocol)**:
    - Facilitated asynchronous communication between services using RabbitMQ.

3. Message Broker

- **RabbitMQ**:
    - Used for inter-service communication, ensuring asynchronous, reliable, and scalable message delivery.

4. Database

- **SQLite**:
    - Used SQLite database for simplicity, as it is lightweight, serverless database used for local data storage in each service.
    - Chosen for simplicity and ease of integration with GORM ORM.
- **GORM**:
    - Used GORM for database interactions and operations, as it is a powerful Object-Relational Mapping (ORM) library for Go.
    - Simplifies database operations like querying, insertion, and updates in SQLite.

5. Frontend

- **Go Templates**:
    - Used to create server-rendered HTML pages for the user interface.

    - Enabled dynamic and secure UI generation directly from the server.
- **CSS:**
    - Used for styling the html/templates
- **Javascript:**
    - Used for consuming websockets on client side, and to make UI interactive such as snackbars, dropdowns were controlled by JS.

6. Deployment and Containerization

- **Docker**:
    - Used to containerize each service for consistent environments across development and production.
    - Simplified the project delivery, scaling and execution
- **Render**:
    - Hosted the service on render.com as web-services, to access the GO-CHAT anywhere
- **CloudMQ**:
    - Leveraged the free service of CloudMQ to get RabbitMQ broker instance

7. Testing Tools

- **Postman**:
    - Used for testing and validating REST API endpoints during development.
    - Simplifies debugging and ensures API compliance.
- **Functional Testing Scripts**:
    - Written in Go to test individual functionalities of the auth-service.

8. Development and Collaboration Tools

- **VS Code (Visual Studio Code)**:
    - Used for writing, testing, and debugging the codebase.
- **Git & GitHub**:
    - Used for version control and easy deployment to render
- **GitHub codespaces**:
    - Before delivering this project, I tested the dockerized app using github codespaces
- **Draw.io**:
    - Used to create architecture, data flow, and deployment diagrams.
- **DB Browser for SQLite**:
    - Used to browse, perform operations on databases with this tool. For debugging purposes only.

Topics of course Covered

As of my knowledge about the course, I have at least implemented these topics, while there may be others I have implemented but missed to mention here.

1. Service-Oriented Architecture (SOA)
2. Client-Server Architecture
3. Publish-subscribe Architecture
4. HTTP Protocol along with RESTful API Design in Go
5. AMQP Protocol
6. RabbitMQ for Message Brokering
7. Database Management with SQLite and Repository pattern
8. Object-Relational Mapping (ORM) using GORM
9. Authentication and Authorization (JWT)
10. Session Tokens and cookies
11. Concurrency in Go / goroutines
12. Go Templates
13. Environment Variables (.env) Management
14. Asynchronous Communication
15. Middlewares

While these are some topics I explored on my own and implemented while building this app

1. Real-Time Communication with WebSocket Protocol
2. Containerization with Docker
3. Cloud Deployment using Render
4. Testing APIs with Postman

Features and Functionalities

User Authentication:
Secure login, registration, and password recovery using JWT.

Real-Time Messaging:
One-to-one real-time chat using WebSocket.

Contact Management:
Add, accept, and reject contact requests. Search for users to add as contacts.

File Sharing:
Chunk-based file uploads and downloads to and from server within chat.

Emoji Reactions:
Send and display emoji reactions in chat messages.

Notifications:
Real-time notifications for contact requests and chat events.

Dynamic User Interface:
Server-rendered HTML pages using Go templates.

Persistent Data Storage:User data, messages, and contacts stored in SQLite databases.

Implementation Details

In this section, I have explained the folder structure and code details for each service and app.

Auth-Service

- **AMQP**: Handles RabbitMQ configuration and inter-service communication. amqp_config.go - Configures RabbitMQ for message brokering and ensures reliable asynchronous messaging.
- **Handlers**: Manages HTTP endpoints for authentication operations like login, registration, and password recovery.
- **Middleware**: Implements JWT-based, session_token authentication for securing API endpoints.
- **Models**: Defines data structures for users and sessions
- **Repository**: Contains logic for CRUD operations on user and session data
- **Tests**: Functional and unit tests for all authentication workflows (e.g., login, registration, password recovery).
- **Utils**: Utility functions for environment variable management (load_envs.go), input validation, and application constants.
- **Dockerfile**: Containerised the Auth-Service for deployment, ensuring consistent environments across development and production.
- **.env**: Stores sensitive configuration such as RabbitMQ credentials, database settings, and JWT secrets.
- **main.go**: Initializes the service, connects to RabbitMQ and SQLite, and sets up HTTP routes and middleware.

Consumer

- **AMQP**: Handles RabbitMQ configuration and message consumption.
- **Handlers**: Manages HTTP requests for interacting with backend services, and rendering templates
- **Middleware**: Implements authentication middleware to secure routes.
- **Static**: Contains static assets such as CSS (chat.css, dashboard.css) and JavaScript (helpers.js) for UI design and smooth experience.
- **Templates**: Includes server-rendered HTML templates for pages like chat, contacts, and dashboard.
- **Utils**: Provides reusable utilities for constants, environment variables, template rendering, and validation.
- **Dockerfile**: Containerised the service for consistent deployment.
- **.env**: Manages environment-specific configurations.
- **main.go**: Initializes routes, Http Handler, WebSocket handler, and serves static files.

Contacts-Service

- **AMQP**: Manages RabbitMQ configuration for asynchronous messaging.
- **Handlers**: Contains logic for handling HTTP requests related to contact management.
- **Middleware**: Implements authentication to secure routes.
- **Models**: Defines the structure for contact-related data.
- **Repository**: Handles database operations for storing and retrieving contact information.
- **Utils**: Includes reusable utilities like constants, environment loading, WebSocket handling, and user details.
- **Dockerfile**: Containerizes the Contacts-Service for deployment.
- **.env**: Manages environment configurations for RabbitMQ, database, etc.
- **main.go**: Initializes the service, sets up routes, and integrates with RabbitMQ.

Gateway

- **Middleware**: Contains cross-origin resource sharing (CORS) handling logic to manage cross-domain requests.
- **Utils**: Includes utility files for HTTP and WebSocket proxying, enabling routing of requests to the appropriate backend services.
- **Dockerfile**: Containerised the Gateway for consistent deployment across environments.
- **.env**: Configures service endpoints and other environment-specific settings.
- **main.go**: Initializes the proxy server and sets up routing for HTTP and WebSocket connections.

Messaging-Service

- **AMQP**:Configures RabbitMQ for managing message queues, ensuring reliable delivery of chat messages.
- **Handlers**: Contains HTTP and WebSocket handlers for managing messaging functionality, including real-time chat.
- **Middleware**: Provides authentication middleware for securing chat-related endpoints.
- **Models**: Defines data structures for messages and related entities.
- **Repository**:Manages database operations for storing and retrieving chat messages.
- **Utils**:Includes helper utilities for environment management, WebSocket handling, file saving, and user data management.
- **Dockerfile**: Containerizes the service for scalable deployment.
- **.env**: Configures RabbitMQ, database, and other environment variables.
- **main.go**: Sets up the service, integrates RabbitMQ, and initializes WebSocket and HTTP routes.

Docker-compose.yml file

- **docker-compose.yml**: Defines the multi-container setup for local testing and development. It makes sure each service runs within bridged network to avoid CORS issue

Testing and Deployment

In this section, I explain how testing was carried out for this app. I mainly wrote test cases for auth-service, while for system as a whole was tested using Postman API and Websocket testing

Testing

1. **Auth-Service Unit Testing**:
    - Utilized **httptest** and **testing** packages for testing HTTP handlers and middleware.
    - Incorporated `github.com/stretchr/testify/assert` for making assertions and validating test outcomes.
    - Focused on core functionalities like login, registration, and password recovery.
2. **API and WebSocket Testing**:
    - Used **Postman** for end-to-end testing of the entire application.
    - Tested REST APIs for authentication, contact management, and messaging.
    - Verified WebSocket functionality for real-time messaging and notifications.

Deployment

1. **Containerization**:
    - Dockerized all services, ensuring consistent development and production environments.
    - **Dockerfile** included in each service for container building.
    - **docker-compose.yml** used for multi-container orchestration in local testing.
2. **Cloud Hosting**:
    - Deployed all services on FREE **Render** instance, leveraging its support for containerized applications.
    - Leveraged CloudMQ's RabbitMQ instance to use it on deployment.

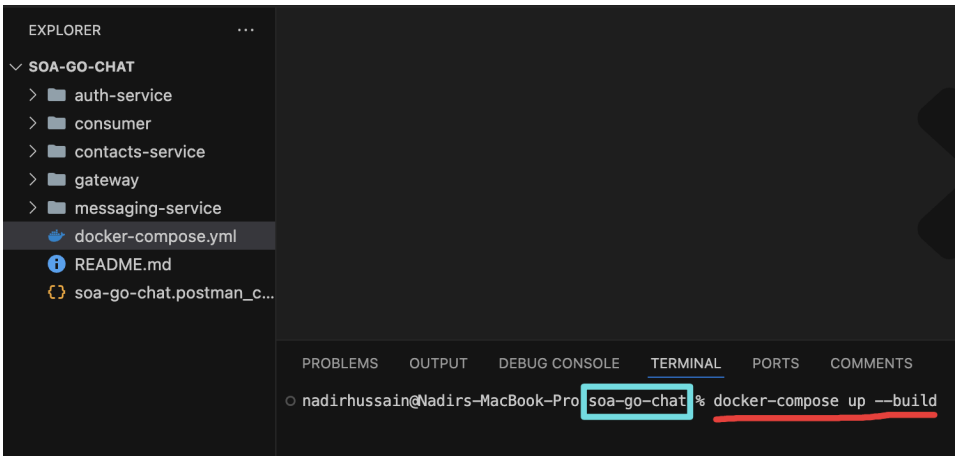
I did deployment testing on Github codespaces to make sure the app runs on your end with minimal configuration and few commands.

Accessing the app

There are 2 ways to access the app

1. **Run the Code on Your System**
    a. Clone the repository or download and extract the code to your local machine.
    b. Ensure Docker is installed and running.
    c. Navigate to the project directory (at root where docker-compose.yml file is available), and open a terminal
    d. Run this command in terminal
    docker-compose up --build
    e. Access the app on ***http://localhost:8085*** (replace PORT with the configured port in .env & in docker-compose.yml if you get error for PORT, and repeat step d and e)

    This is all you need for this method, after docker is installed and running



2. **Access the deployed URL provided by render**
    https://soa-go-chat-3.onrender.com/

    Note: As this app is deployed on a **FREE** instance of the render server, it may happen that the app takes some time to load for the first time, because the instance might be sleeping at first attempt. You may even have to refresh and try again. In worst cases, it may not let you login/register and server may not respond and we need to restart the server manually and it will work fine.

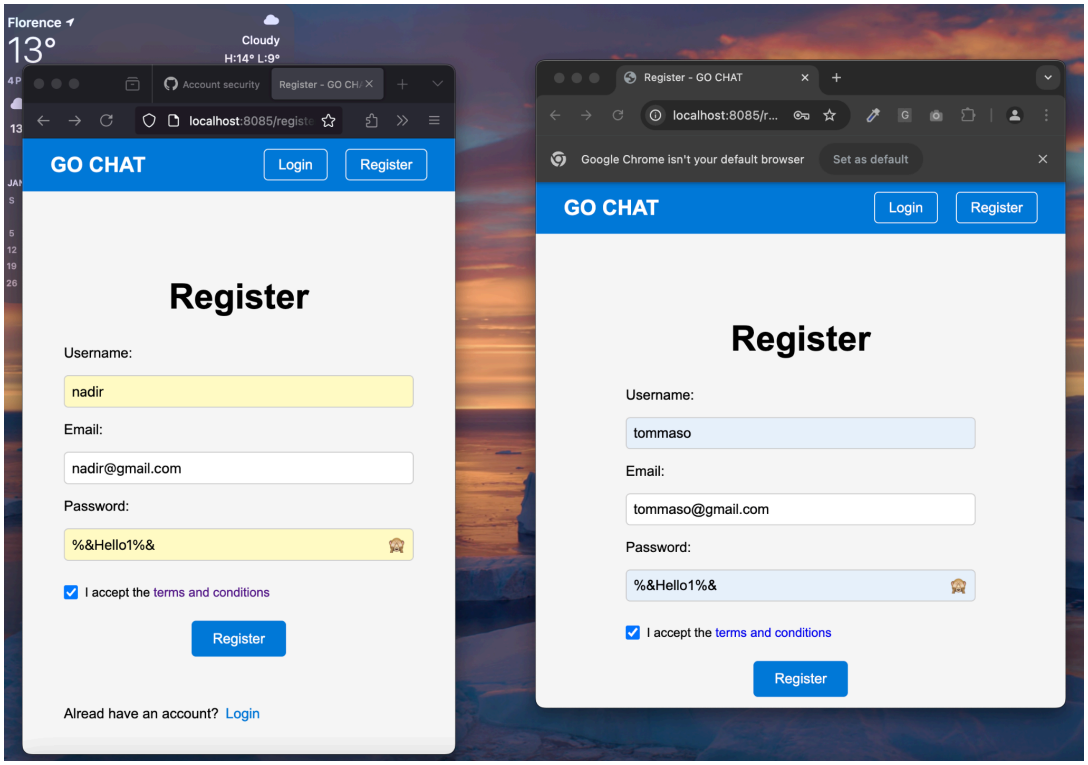
How does the app work?

- You can open the above same URL in 2 different browsers. Create 2 separate users. Then login, and Go on **Contacts** page. Type the the username of other user, and send contact request.
- On other user side, go into Contacts page, accept the request and get back to Dashboard
- Now when both users are on the dashboard, you can click their chat and send messages, files and emojis.

Visually, it should be like this.

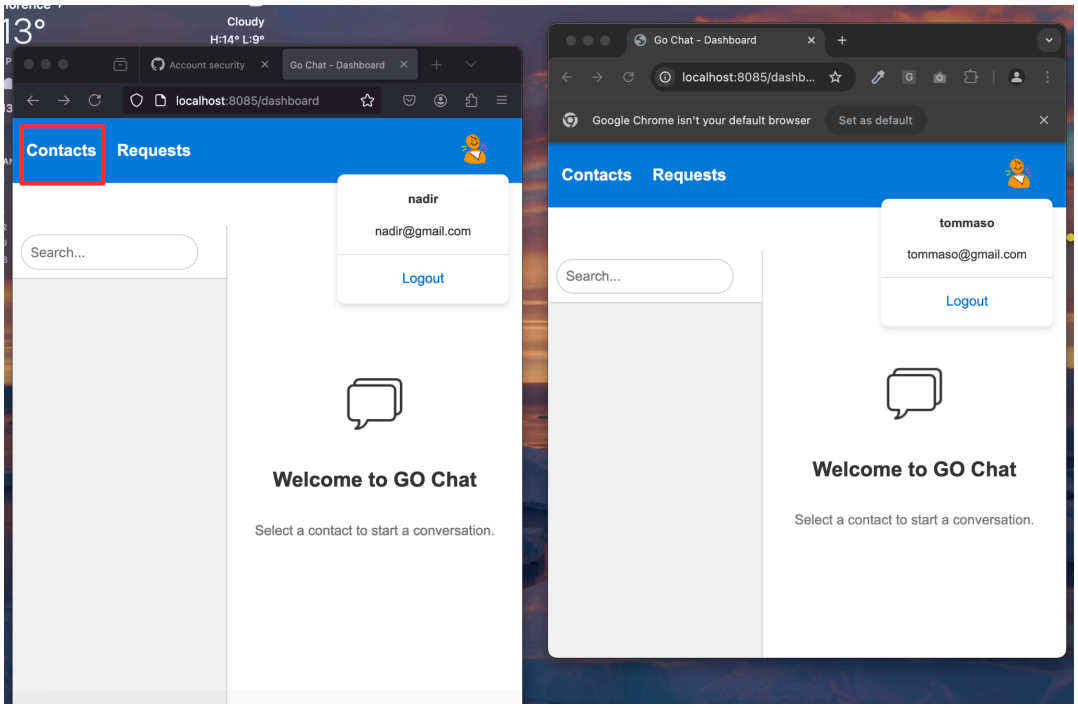
Step1:

2 Users should be registered. For exp: valid username, valid email and valid password may look like this



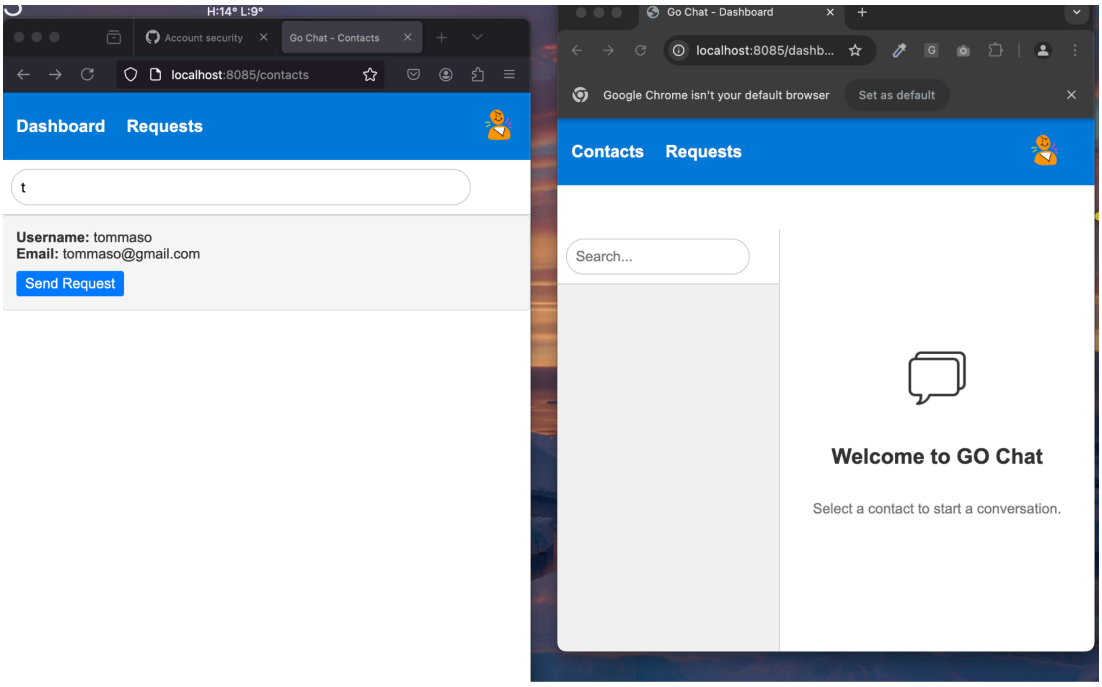
Step2:

After login, I will see something like this. Then i need to hit Contacts btn in top menu for any user.

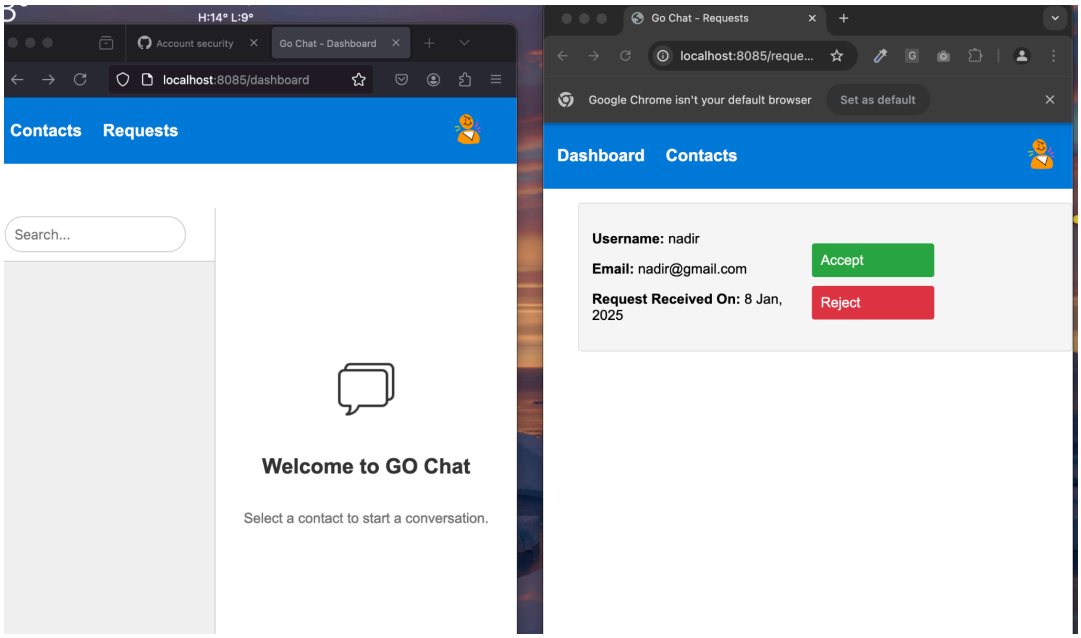


Step3:

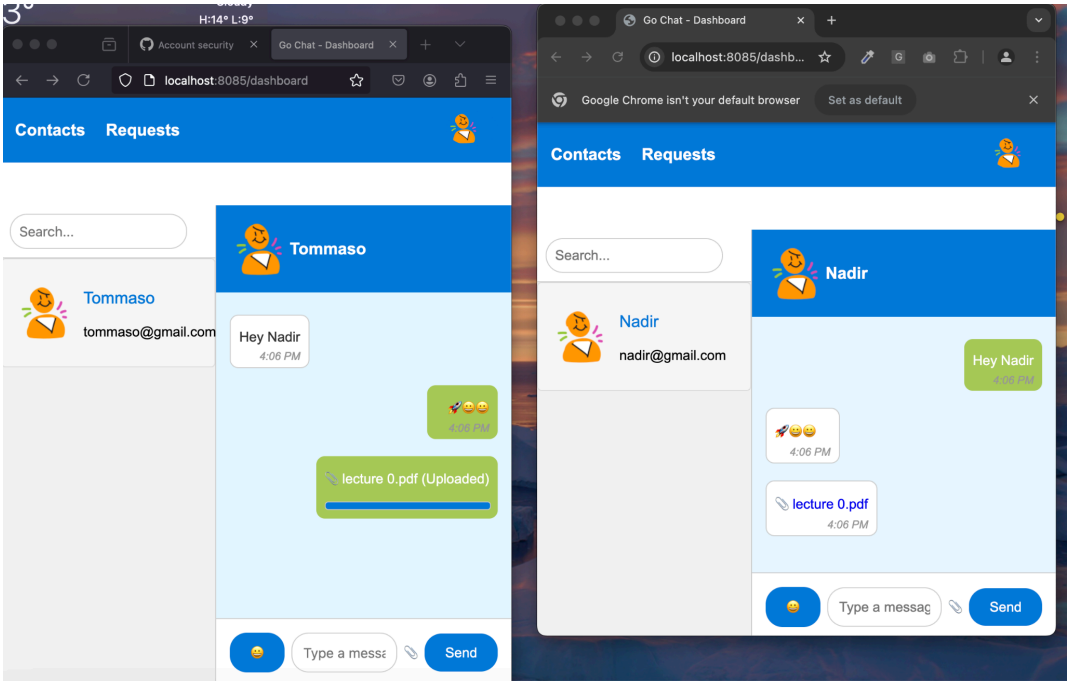
It will show like this, when i type t in the search field. The hit Send Request btn



Step4:
On other user side, in this case Tommaso, head to Requests tab and Accept the pending requests.



Step5:
Once accepted, both users are connected, and they can do chat now. Head to dashboards for both and click the contact in contact list and simply do chat.



Conclusion and Future Work

The **GO-CHAT** application successfully implements a Service-Oriented Architecture (SOA) for a modular, scalable, and fault-tolerant chat system. Key features like real-time messaging, secure authentication, contact management, and asynchronous communication were achieved using a combination of modern technologies, including Go, RabbitMQ, SQLite, and Docker. The app demonstrates the effective use of distributed programming principles, ensuring reliability, scalability, and seamless user interaction through WebSocket and REST APIs. Deployment on Render further validates the app's readiness for production environments.

In the future, this app can be extended to support Group Chat, video and voice calling, upgrading databases from SQLite to cloud-based databases. E2E encryption for text and file uploads can be implemented to keep the users' data private.

References

Learning and Tutorials from

- Moodle lectures
- https://go.dev/tour/welcome/1
- https://medium.com/@parvjn616/building-a-websocket-chat-application-in-go-388fff758575
- ChatGPT - https://chatgpt.com

Dockerizing go app

https://blog.logrocket.com/dockerizing-go-application/
https://www.geeksforgeeks.org/how-to-dockerize-a-golang-application/

Links to tools

- Cloud MQ
  https://www.cloudamqp.com/

  Render
  https://render.com/