





Capstone Project: Scalable Real-Time Chat System with Redis Pub/Sub

Objective

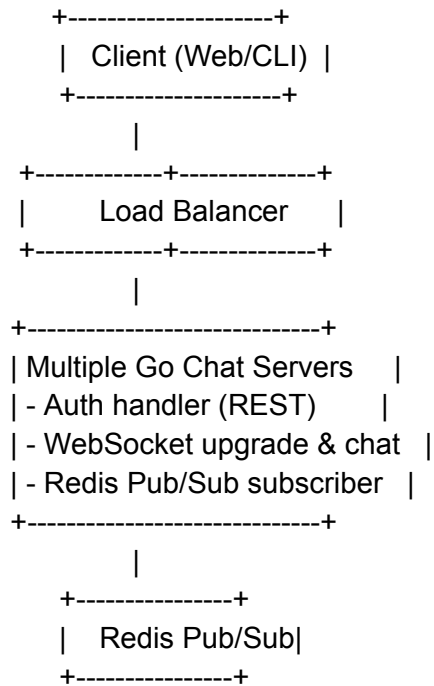
Build a real-time chat system using Go that supports:

- Multiple users chatting concurrently
 - Channel-based messaging (e.g., general, support, tech)
 - Persistent chat history
 - Redis for message distribution across instances
-

Core Features

Feature	Description
 User Login	Simple auth system with session/token-based login
 Real-Time Chat	WebSocket-based communication between users
 Channels/Rooms	Users can join channels to chat in groups
 Chat History	Persist chat messages in Redis or PostgreSQL
 Distributed Pub/Sub	Use Redis to scale chat across multiple instances

🧱 Architecture



🧩 System Components

1. Auth API (REST)

- Register, Login
- Generate JWT or session tokens
- Protect WebSocket endpoint

2. Chat Gateway (WebSocket)

- Handle WebSocket upgrades

- Read/send messages to Redis pub/sub
- Keep alive connections

3. Redis Pub/Sub

- Publish messages per channel
- All instances subscribe to the same channel
- Enables horizontal scaling

4. Message Persistence (Optional)

- Store messages in PostgreSQL or Redis
- Expose a REST endpoint to fetch past messages



Demo Scenarios

- User logs in, joins “general” channel
- Sends message → all users in “general” receive it in real time
- Server restart does not lose chat history
- Multiple server instances still sync via Redis



Suggested Folder Layout

```
go-chat-system/  
├── api/  
│   ├── auth.go  
│   └── chat.go  
├── ws/  
└── client.go
```

```
| |— hub.go
|— redis/
|   |— pubsub.go
|— models/
|   |— message.go
|— db/
|   |— store.go
|— main.go
|— go.mod
|— README.md
```

Tech Stack

- **Go** for the backend
 - **Redis** for Pub/Sub and optional message store
 - **PostgreSQL** for persistence (optional)
 - **WebSocket** for real-time communication
 - **JWT** for authentication
-

Stretch Goals

- Typing indicator support
 - Private 1-on-1 chats
 - Message delivery acknowledgment
 - React frontend or CLI client
 - Docker-compose orchestration
-



Success Criteria

- WebSocket support for 1000+ concurrent users
 - Message delivery under 100ms latency
 - Proper synchronization across multiple servers using Redis
 - Clean, modular Go code with clear docs
-