VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

PROJECT REPORT

MangaHub

NET-CENTRIC PROGRAMMING (IT096IU)

Course by

Dr. Le Thanh Son & MSc. Nguyen Trung Nghia

Group members – Student ID:

Nguyễn Anh Quân - ITITIU22129

Đặng Huỳnh Minh Phúc - ITITIU22126

Table of contents

CHAPTER 1: INTRODUCTION

1. Overview:

MangaHub is a comprehensive manga and comic tracking system created as a team project for designed for teams of 2 students over a 10-11 week timeline, the project focuses on building a practical network application using the Go programming language. It demonstrates the implementation of five key communication protocols: TCP, UDP, HTTP, gRPC, and WebSocket, incorporating real-time synchronization, user authentication, and community features. This allows users to discover manga, manage personal libraries, track reading progress, and join in chatting room.

2. Objective:

The primary goals of MangaHub emphasis on network programming and distributed systems:

- Network Application: build a realistic, scoped application in Go, experience with protocol implementation.
- Protocol: Implement and integrate TCP, UDP, HTTP, gRPC, and WebSocket
- Networking Concepts: understand of concepts of client-server models, broadcasting, heartbeats (PING/PONG), and concurrent handling of connections.
- Concurrent and Distributed Programming: Develop skills in Go (e.g., goroutines, channels, mutexes) and basic distributed patterns, such as hubs for managing clients in TCP/UDP/WebSocket servers.

3. Key features:

- User authentication: register/login with JWT tokens
- Manga catalog with search by title, author or genre
- Personal library management
- Reading progress tracking with real-time sync
- Real-time global chat (websocket) with typing indicator
- Progress broadcasting via TCP (reliable) and UDP (fast)
- gRPC service ready for microservices

- Responsive web interface

4. Used tools:

- Programming Language: Go (Golang)
- Web Framework: Gin
- WebSocket Library: Gorilla WebSocket
- gRPC: Go gRPC library.
- Database: SQLite
- Authentication: JWT (JSON Web Tokens)
- Documentation Tools: Swagger (Swaggo) for API docs
- IDE: Visual Studio Code – For code editing and debugging
- Version Control: Git – For collaboration and code management
- Testing Tools: Telnet for TCP testing, custom UDP client, Postman for API testing
- UI Tools: HTML5, CSS3, JavaScript
- Communication Tools: Discord, Zalo, Messenger

CHAPTER 2: METHODOLOGY

1. **Goals:**

Gain experience in network application development using Go: We aimed to build an end-to-end system that mirrored real-world applications, focusing on integration

Implement and integrate all five required communication protocols (TCP, UDP, HTTP, gRPC, WebSocket): Each protocol had a role in the system architecture and demonstrate their practical differences and complementary use

Strengthen understanding of networking concepts through progressive, manageable implementation: We approached complex topics from simple to complex (e.g., connection management, broadcasting, heartbeats), from simple servers with no IU to full inter-server communication

Develop skills in concurrent programming and basic distributed system: Heavy emphasis was placed on Go's concurrency features to handle multiple clients safely across protocol servers

Produce a functional system demonstrating network programming: The final product had to support live demonstration of all protocols working together

2. **Rules:**

Our MangaHub project followed strict rules and constraints from the course assignment project:

- Team Composition: Teams consisted of exactly 2 students per group. No individual submissions or larger teams were allowed.
- Programming Language: Had to use Go (Golang) exclusively. No other languages were permitted
- Protocol Requirements: All five communication protocols (TCP, UDP, HTTP, gRPC, WebSocket) must be implemented and integrated.
- Scope Limitations: The project used the revised scope for feasibility. We avoided overly complex features and focused on basic manga tracking, progress sync, and chat.
- Delivery:
    - Source code and documentation submitted as a zip file named Group11_MangaHub.zip on Blackboard before the due date.

- All files included: code, README, database, and any necessary docs files.

- Collaboration: Code must be original work of the team, no external code beyond standard libraries and approved dependencies (like Gin, Gorilla WebSocket, gRPC).

3. Instruction:

Project Setup Steps:

a. Clone the repository:

git clone ./MangaHub-main.git

cd MangaHub-main

b. Install dependencies:

go mod tidy

required packages: (Gin, Gorilla WebSocket, gRPC, etc.)

c. Open the project in VS Code:

Open the folder MangaHub-main in VS Code

Let the Go extension install additional tools when prompted (like gopls, delve for debugging, etc.)

The project has five servers that run at the same time, used background processes in terminal:

go run cmd/api-server/main.go &

go run cmd/tcp-server/main.go &

go run cmd/udp-server/main.go &

go run cmd/websocket-server/main.go &

go run cmd/grpc-server/main.go &

- API server runs on port 8080 (Swagger docs at /swagger/index.html)
- TCP server on port 9090 (test with telnet localhost 9090)
- UDP server on port 9091
- WebSocket chat on port 9093
- gRPC server on port 9092

To stop all servers type Ctrl+C

Testing the Features:

- Open web/search_manga.html and web/broadcast_chatroom.html using Live Server extension or any browser
- Register/login via API (use REST Client or browser page)
- Test progress update – it will automatically broadcast to TCP and UDP
- Use telnet for TCP client, or provided test UDP client in cmd/udp-server/test

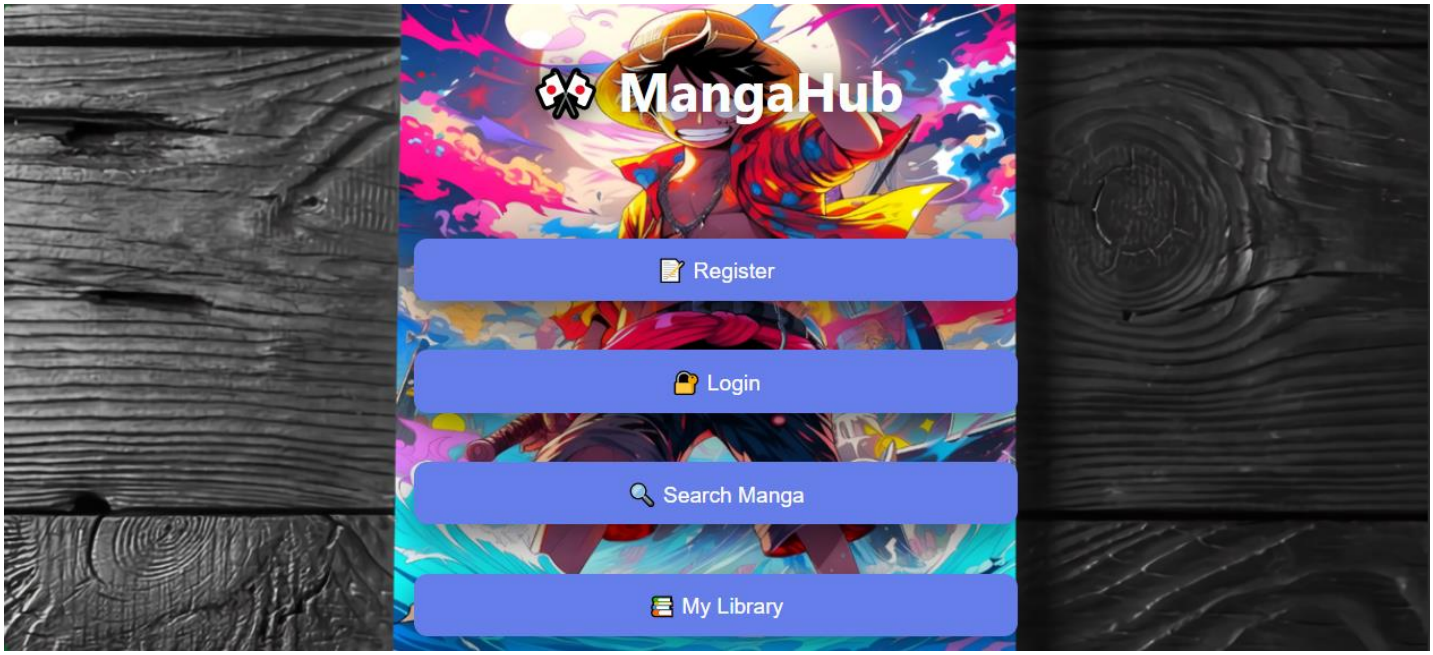
Database: The SQLite file is at data/mangahub.db. It creates automatically on first run and seeds with sample manga (One Piece, Jujutsu Kaisen, etc.)

4. **Website Design – Step-by-step run:**

a. Frontend (HTML Files):

The frontend consists of two standalone HTML files in the web/ folder. Both are written in plain HTML5, CSS3, and JavaScript.

i. search_manga.html



This file handles user authentication, manga search, library management, and progress updates.

Key sections and how the JavaScript works:

Global Variables and Constants:

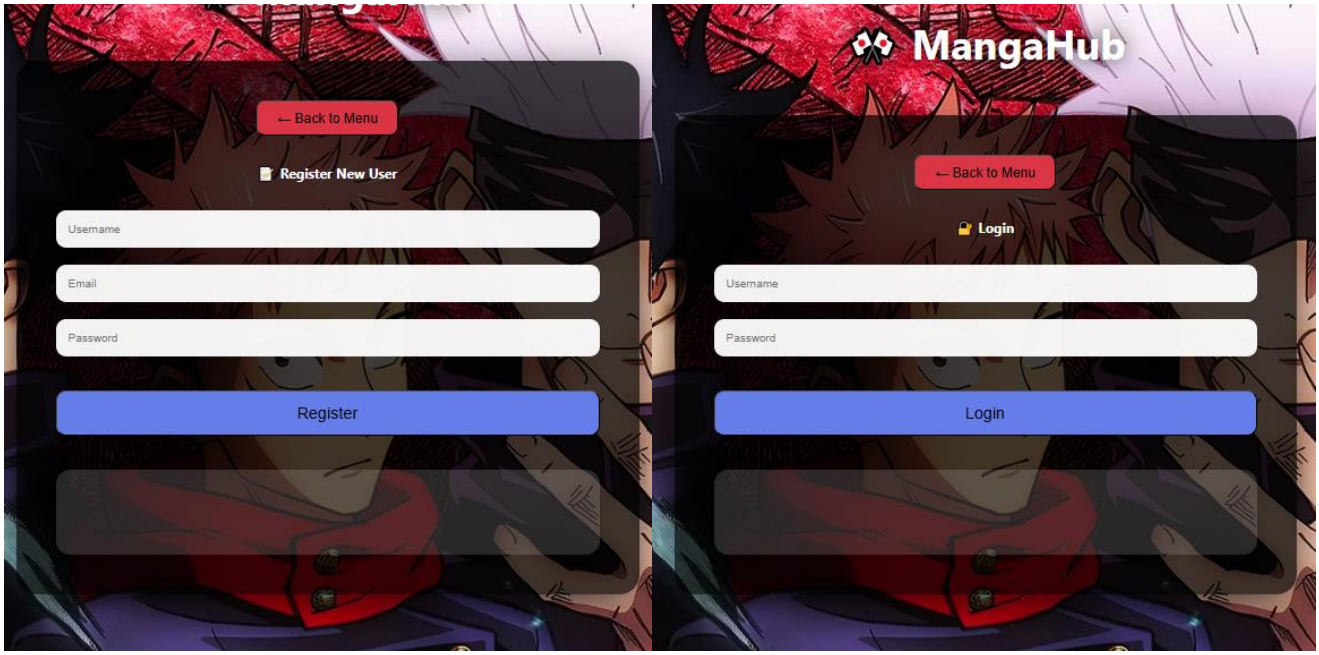
const API_URL = '/';

let token = localStorage.getItem('token') || '';

Stores the backend API address and JWT token from localStorage for authenticated requests.

showMain(), showLogin(), showRegister() switch between main menu, login form, and register form by changing display styles.

- Page loads → window.onload = **showMain()**
- User clicks "Login" → **showLogin()**
- User fills and submits → **login()** → if success → **showMain()**
- User clicks "Register" → **showRegister()** → if success → **showMain()**
- **logout()** → clears token + **showLogin()** or back to main-menu



login() and register() Collect form data, send POST to /auth/login or /auth/register using fetch, save token to localStorage, then call **getLibrary()** and **searchManga()** to refresh the page

register() function:

- Sends data to POST /auth/register
- Backend creates user, hashes password with bcrypt, saves to database, returns success or error (e.g., duplicate username/email)
- On success: shows message and switches to login screen
- On failure: shows error from server (like "username already exists")

login() function:

- Sends data to POST /auth/login
- Backend checks password with bcrypt.CompareHashAndPassword, generates JWT token (valid 24 hours) if correct, run:
    - Saves token to global variable token and localStorage (so it survives page refresh)
    - Saves username and user_id too
    - Calls getLibrary() to load personal manga list

  - Calls searchManga() to show the catalog
  - Switches UI to main page with showMain()
- If fail: shows error (usually "invalid credentials")

After successful login:

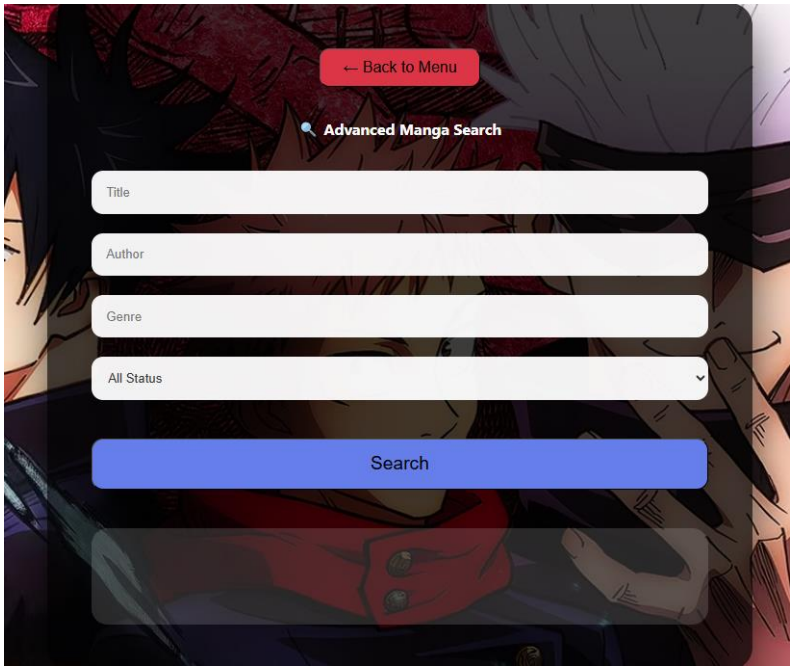
- All protected API calls (like getLibrary, addToLibrary, updateChapter)
- The backend JWT middleware checks this token on every protected route.
- If token is missing or invalid → 401 Unauthorized.

searchManga() Called when user types in search input or selects genre. Sends GET request to /manga/search with query/genre parameters. Displays results as cards with title, author, genres, description, and "Add to Library" button.

- Token Check, if no token (user not logged in), shows a message asking to login. Prevents unnecessary API calls.
- API Request
  - Sends GET to /users/library
  - Includes Authorization: Bearer ${token} header
  - This endpoint is protected by JWT middleware – backend verifies token and gets user_id from claims
- Returns an array of objects, each containing:
  - manga: full manga details (title, author, total_chapters, etc.)
  - progress: user's current_chapter and status
- Build HTML Display
- Title "My Library"
  - If empty → friendly message encouraging to add manga
  - Otherwise, creates a grid of cards
  - Each card shows:
  - Title and author
  - Current status (reading/completed/plan_to_read)
  - Input field pre-filled with current chapter
  - Max limited to total_chapters

- Update button that calls updateChapter(manga.id) when clicked

- Update Display: Uses the same setResult('library_result', html) helper to inject the HTML into the #library_result div



My Library button:

- Searches manga → sees "Add" buttons in results
- Clicks "Add to Reading" / "Plan to Read" / "Mark Completed" → **addTo**Library() runs
- Library refreshes automatically → **getLibrary()** shows the new entry
- User changes chapter number in the input field → clicks "Update" → **updateChapter()** runs
- Progress saved + broadcasted → library refreshes again with new chapter

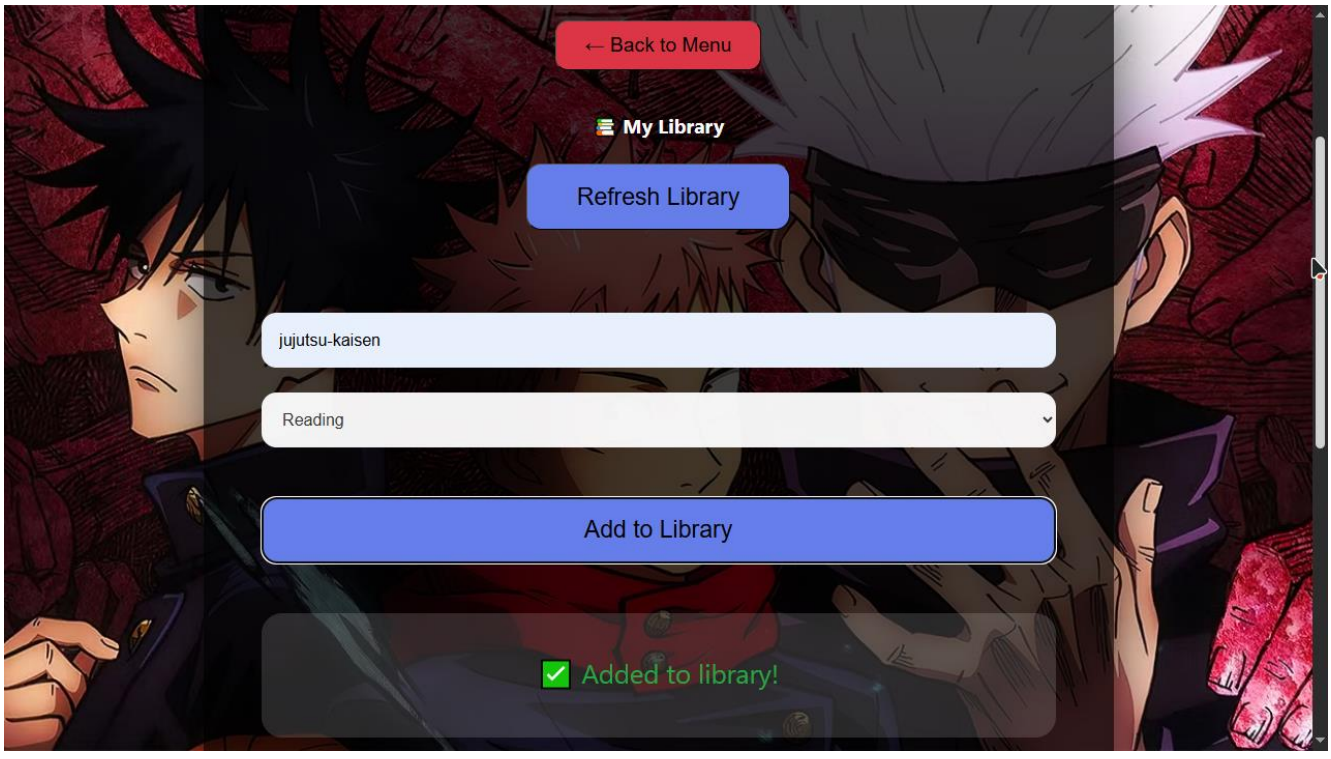
addToLibrary(manga_id, status) Sends POST to /users/library with manga_id and status (reading/completed/plan_to_read). Shows success/error message and refreshes library.

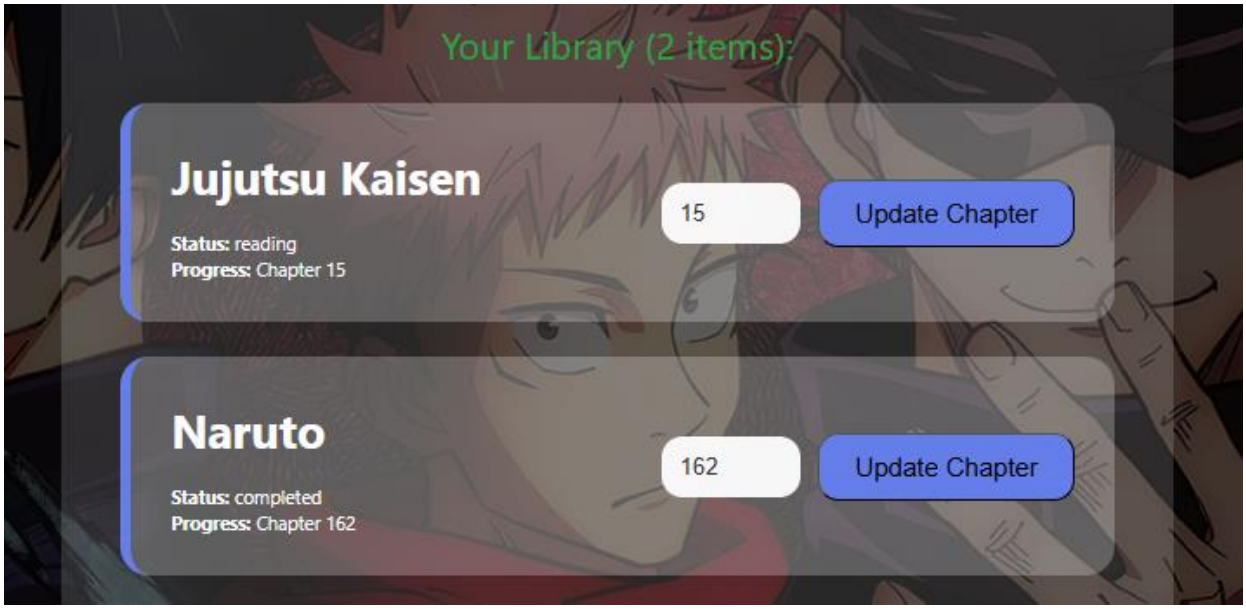
- Collects the manga_id (passed from button onclick) and status ("reading", "plan_to_read", or "completed")
- Checks if user is logged in (token exists)
- Sends POST to /users/library with JSON body { manga_id, status }
- Includes Authorization: Bearer token header

- Backend inserts or updates the user_progress row (initial chapter = 0 if new)
- If success: shows "Added to library!" message, calls **getLibrary()** to refresh and show the new manga in "My Library"
- If error (e.g., manga not found): shows error message

getLibrary() Sends GET to /users/library (with Bearer token). Displays user's current manga list with current chapter input and update button.

- Called after login, after **addToLibrary()**, and after updateChapter
- Fetches full list from /users/library (protected endpoint)
- For each entry, builds a card showing:
  - Manga title and author
  - Current status
  - Number input pre-filled with current_chapter (min=0, max=total_chapters)
  - "Update" button that calls **updateChapter(manga.id)**
- If library empty → shows friendly message
- Updates the #library_result div instantly





updateChapter(manga_id) Reads the chapter input value next to that manga. Sends PUT to /users/progress with manga_id and current_chapter. Shows alert on success and refreshes library.

- Gets the manga_id from the button
- Reads the value from the specific input field (id = "chapter-${manga_id}")
- Validates it's a valid non-negative number
- Sends PUT to /users/progress with JSON { manga_id, current_chapter }
- Includes Bearer token
- Backend updates the current_chapter in user_progress table
- Also triggers broadcast: fetches username and manga title, sends JSON progress update to TCP and UDP internal endpoints
- If success: shows alert "Chapter updated successfully!", calls getLibrary() to refresh the display with new chapter
- If error: shows alert with error message

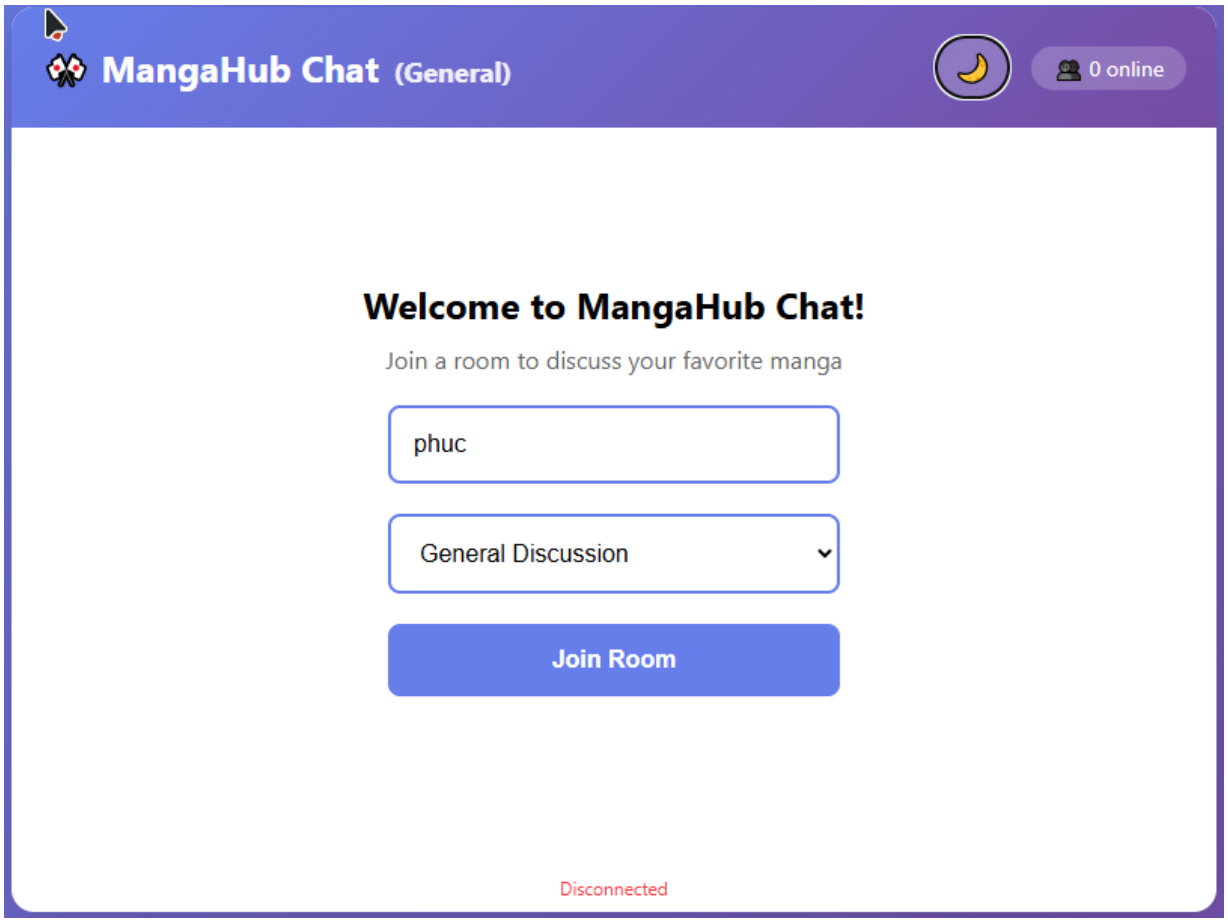
logout() Clears token from localStorage and goes back to login screen.

All API calls use Authorization: Bearer ${token} header for protected routes. Progress broadcast to TCP/UDP.

ii. **broadcast_chatroom.html**

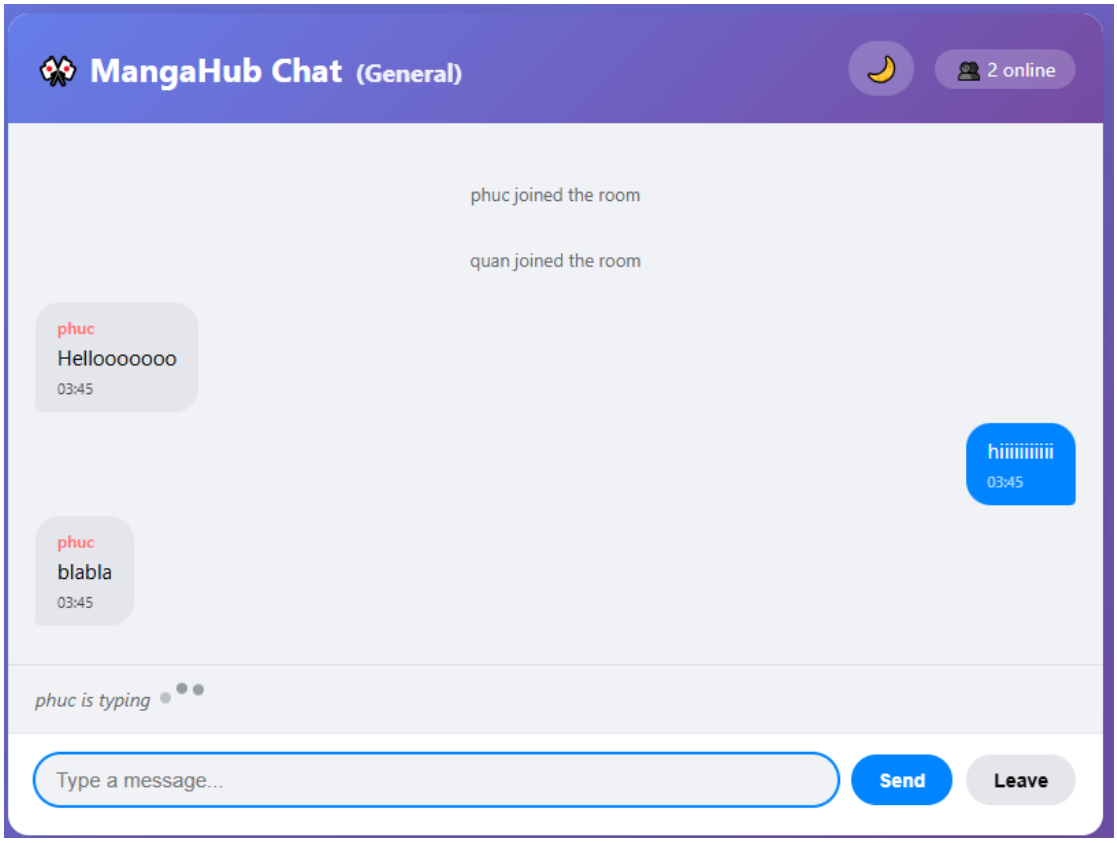
Real-time chat system using WebSocket. All chat logic is handled through several interconnected JavaScript methods that manage connection, messaging, typing indicators, online count, and UI updates.

- User opens page → sees join screen
- Enters username/room → **joinChat()** → WebSocket connects
- Server sends history + join system message
- User types → typing indicator sent and shown to others
- User sends message → **sendMessage()** → appears locally + sent to server → broadcast to room
- Others receive via onmessage → see message instantly
- Online count updates periodically
- User leaves → clean disconnection with system message



joinChat() function:

- Collects the username from the input field and the selected room (default: "general")
- Validates that username is not empty
- Creates a new WebSocket connection
- Sets up four important WebSocket event handlers:
  - onopen: sends a system message that user joined, loads message history
  - onmessage: processes incoming messages
  - onclose: shows disconnection message and hides typing indicator
  - onerror: logs error
- Hides the join screen and shows the full chat interface



sendMessage() function:

Called when user presses Enter or clicks send button.

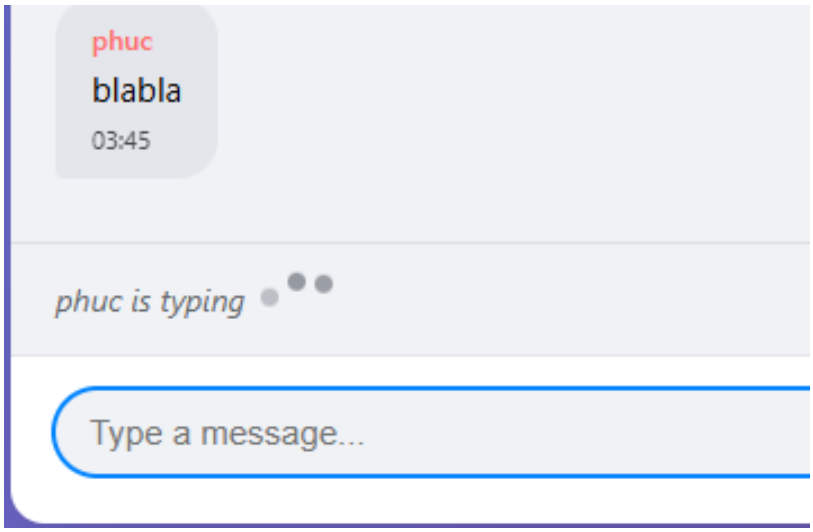
- Gets the text from the message input field

- Trims whitespace and checks if message is not empty
- Creates a JSON object { type: 'chat', text: message }
- Sends it directly through the WebSocket (ws.send())
- Clears the input field and resets typing state
- Adds the sent message immediately to the local chat window (optimistic UI) with current time

onmessage handler (inside joinChat) function:

processes all incoming data from the WebSocket server, use JSON and handles different message types:

- "chat": regular user message → appends to chat window with username, time, and escaped text
- "system": join/leave notifications → shows in gray/italic style
- "typing": someone is typing → shows "username is typing..." indicator for 3 seconds
- Also handles initial message history: when connecting, server sends last 50 messages → all are added at once



Typing Indicator System:

Two parts work together for real-time typing feedback:

- On input event in message field:
  - If user starts typing and isTyping flag is false → sends { type: 'typing', username, room }
  - Sets isTyping = true
  - When field becomes empty → resets isTyping = false
- Server broadcasts typing event to all in room → other clients show indicator

- Indicator auto-hides after 3 seconds if no new typing event arrives

updateOnlineCount() function:

Runs every 5 seconds using setInterval.

- Sends GET request to localhost:9093/stas
- Server returns JSON { online_users: number }
- Updates the online counter display in header
- Gives users a sense of active community

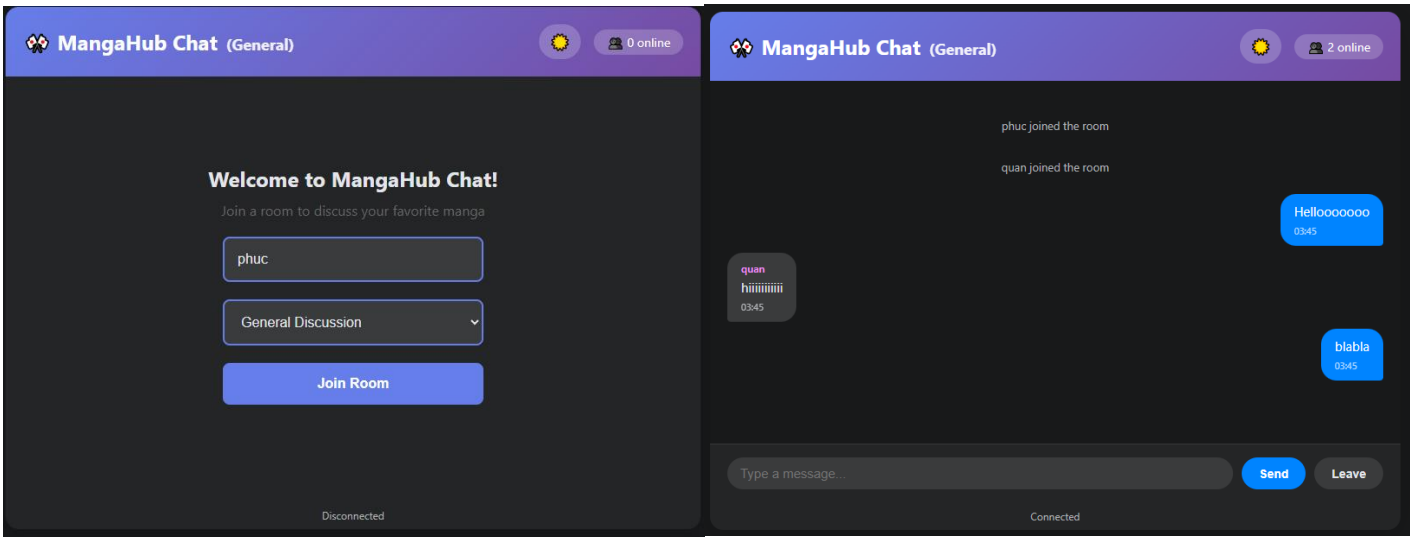
leaveChat() function and **Modal Handling:**

- Shows a confirmation modal when user tries to leave
- On confirm: closes WebSocket, shows join screen again
- Modal can be closed by clicking outside

Dark Mode Toggle

Simple button that adds/removes "dark-mode" class on body.

Changes background, container colors, and text for better night viewing

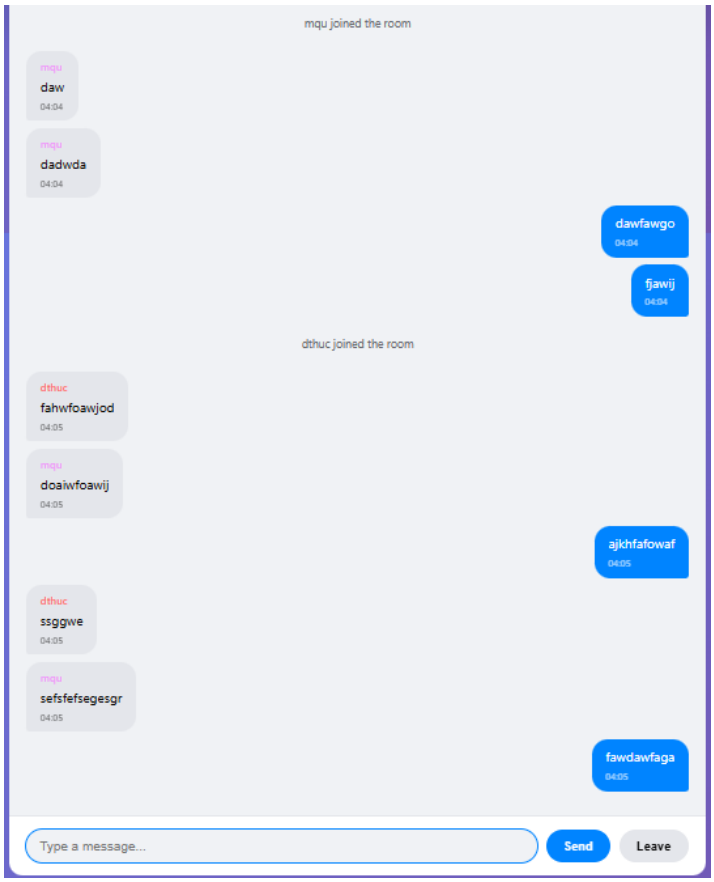


Message Display and Scrolling

Every time a message is added:

- Creates a div with proper classes (user/system)

- Appends username, time, and text
- Inserts into chat messages container
- Auto-scrolls to bottom so newest message is always visible



CHAPTER 3:
PROTOCOLS USED

1. HTTP

HTTP server is implemented in **cmd/api-server/main.go** using the Gin web framework.

Key characteristics in MangaHub:

- Runs on port 8080
- Stateless request-response model
- Uses JSON for request and response bodies
- Supports standard HTTP methods: GET, POST, PUT
- Includes JWT-based authentication for protected routes
- Provides interactive API documentation via Swagger

Main HTTP Endpoints and Their Roles

Authentication

- POST /auth/register → creates new user (username, email, password → hashed with bcrypt)
- POST /auth/login → verifies credentials → returns JWT token (valid 24 hours)

Manga Catalog (Public – no auth needed)

- GET /manga → returns all manga
- GET /manga/search?query=...&genre=... → searches by title/author or filters by genre

Library and Progress (Protected – requires Bearer JWT token)

- GET /users/library → returns user's added manga with current progress and status
- POST /users/library → adds manga to library with specified status (reading/completed/plan_to_read)
- PUT /users/progress → updates current_chapter (and optionally status)

Corresponding Code **(from cmd/api-server/main.go)**

```
router := gin.Default()

// Public routes
```

```
router.POST("/auth/register", handlers.Register)
router.POST("/auth/login", handlers.Login)
router.GET("/manga", handlers.GetAllManga)
router.GET("/manga/search", handlers.SearchManga)

// Protected routes group
authorized := router.Group("/users")
authorized.Use(auth.Middleware())  // JWT validation
{
    authorized.GET("/library", handlers.GetUserLibrary)
    authorized.POST("/library", handlers.AddToLibrary)
    authorized.PUT("/progress", handlers.UpdateProgress)
}
```

Triggers Broadcasting:

The most important part: when user updates chapter via PUT /users/progress

In handlers.UpdateProgress (main.go):

- Validates input and token
- Updates user_progress table in SQLite
- Retrieves username and manga title
- Constructs shared.ProgressUpdate struct
- Marshals to JSON
- Sends HTTP POST to internal endpoints:

This shows perfect integration: HTTP receives user action → processes → uses internal HTTP calls to trigger other protocols.

Swagger Integration

```
router.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
```

2. TCP

TCP server runs independently on port 9090 (client connections) and exposes an internal HTTP endpoint on port 9091 to receive progress triggers from the main API server.

- Connection-oriented: clients must establish and maintain a persistent TCP connection
- Reliable delivery with acknowledgment and retransmission
- Built-in flow control and ordering
- Heartbeat mechanism to detect disconnected clients
- Clients authenticate by sending their UserID after connecting
- Broadcasts JSON-formatted progress updates to all registered clients

Main files:

- **cmd/tcp-server/main.go** - entry point, starts listener and internal HTTP receiver
- **internal/tcp/hub.go** - central hub managing all clients (register/unregister/broadcast)
- **internal/tcp/tcp_client.go** - readPump (heartbeat handling) and writePump (sending updates + PING)

Starting the TCP Listener (main.go)

```
listener, err := net.Listen("tcp", ":9090")
if err != nil {
    log.Fatal("Error starting TCP listener:", err)
}

for {
    conn, err := listener.Accept()
    if err != nil {
        log.Println("Error accepting connection:", err)
        continue
    }
    go handleTCPConnection(conn)  // Handle each client in a goroutine
}
```

Client Authentication and Registration (handleTCPConnection in main.go)

```
// Prompt and read UserID
fmt.Fprintf(conn, "Welcome to MangaHub Progress!\nEnter your UserID: ")
scanner := bufio.NewScanner(conn)
if !scanner.Scan() {
    return
```

```
}
userID := scanner.Text()

client := &tcp.Client{
    Conn:   conn,
    UserID: userID,
    Send:   make(chan []byte, 256),
}

tcp.GlobalHub.Register <- client
go client.WritePump()  // Sends messages and PING heartbeats
client.ReadPump()      // Listens for PONG and disconnection
```

Hub Broadcasting (internal/tcp/hub.go)

```
func (h *Hub) Run() {
    for {
        select {
        case client := <-h.Register:
            h.clients[client] = true
        case client := <-h.Unregister:
            if _, ok := h.clients[client]; ok {
                delete(h.clients, client)
                close(client.Send)
            }
        case message := <-h.broadcast:
            for client := range h.clients {
                select {
                case client.Send <- message:
                default:
                    close(client.Send)
                    delete(h.clients, client)
                }
            }
        }
    }
}
```

Heartbeat Mechanism (tcp_client.go)

- writePump: sends "PING\n" every 30 seconds
- readPump: responds with "PONG\n" when receiving PING, otherwise unregisters on disconnect

Receiving Progress from API Server

The API server triggers broadcast by sending HTTP POST to **/internal/progress** on localhost:9091.

In tcp-server main.go:

```
router.POST("/internal/progress", receiveProgress)

func receiveProgress(c *gin.Context) {
    var update shared.ProgressUpdate
    if err := c.ShouldBindJSON(&update); err != nil {
        return
    }
    tcp.GlobalHub.BroadcastProgress(models.UserProgress{...}, update.Username,
update.MangaTitle)
}
```

BroadcastProgress (hub.go)

```
func (h *Hub) BroadcastProgress(update models.UserProgress, username, mangaTitle string)
{
    msg := shared.ProgressUpdate{ /* fill fields */ }
    data, _ := json.Marshal(msg)
    h.broadcast <- data  // Sent to all clients via writePump
}
```

3. UDP

UDP server runs independently on **port 9091** and shares the same internal HTTP endpoint (on the same port) to receive progress triggers from the main API server.

- Connectionless: no handshake or persistent connection required
- Minimal overhead: smaller headers, faster transmission
- No guaranteed delivery, ordering, or retransmission

- Subscription-based: clients must periodically "ping" to stay registered
- Broadcasts JSON-formatted progress updates to all active subscribers
- Automatic cleanup of inactive clients

Main files:

- **cmd/udp-server/main-udp**.go - entry point, starts UDP listener and internal HTTP receiver
- **internal/udp/hub.go** - global hub managing subscriber addresses and broadcasting
- **internal/udp/udp_listener.go** - readPump (handles PING subscription) and writePump (sends broadcasts)

Starting the UDP Listener (udp_listener.go)

```
udpAddr, _ := net.ResolveUDPAddr("udp", ":9091")
udpConn, _ = net.ListenUDP("udp", udpAddr)

go readPump()  // Listens for incoming PINGs
go writePump() // Sends broadcasts from hub
```

Client Subscription via PING/PONG (readPump in udp_listener.go)

```
for {
    n, clientAddr, _ := udpConn.ReadFromUDP(buffer)
    message := strings.TrimSpace(string(buffer[:n]))

    if message == "PING" {
        client := &udp.ClientAddr{
            Addr:     clientAddr,
            LastSeen: time.Now(),
        }
        udp.GlobalHub.Register <- client     // Add or refresh subscriber
        udpConn.WriteToUDP([]byte("PONG\n"), clientAddr)
    }
}
```

Hub Management and Cleanup (hub.go)

```
func (h *Hub) Run() {
    ticker := time.NewTicker(10 * time.Second)
    for {
        select {
        case client := <-h.Register:
            h.clients[client.Addr.String()] = client
        case message := <-h.broadcast:
            for _, client := range h.clients {
                udpConn.WriteToUDP(message, client.Addr)
                client.LastSeen = time.Now()
            }
        case <-ticker.C:
            // Remove clients inactive > 30 seconds
            for key, client := range h.clients {
                if time.Since(client.LastSeen) > 30*time.Second {
                    delete(h.clients, key)
                }
            }
        }
    }
}
```

Receiving Progress from API Server

In udp-server main-udp.go:

```
router.POST("/internal/progress", receiveProgress)

func receiveProgress(c *gin.Context) {
    var update shared.ProgressUpdate
    if err := c.ShouldBindJSON(&update); err != nil {
        return
    }
    udp.GlobalHub.BroadcastProgress(models.UserProgress{...}, update.Username,
update.MangaTitle)
}
```

BroadcastProgress (hub.go)

```
func (h *Hub) BroadcastProgress(update models.UserProgress, username, mangaTitle string)
{
    msg := shared.ProgressUpdate{ /* fill fields */ }
    data, _ := json.Marshal(msg)
    h.broadcast <- append(data, '\n')  // Sent to all subscribers via writePump
}
```

4. WebSocket

WebSocket server runs independently on port 9093 and serves both the chat HTML page and the WebSocket endpoint.

- Persistent, full-duplex connection over a single TCP socket
- Low latency server-push capability
- Supports binary and text frames (uses text/JSON here)
- Built-in PING/PONG heartbeats for connection health
- Room-based broadcasting (messages only go to users in the same room)
- Maintains last 50 messages per room for late joiners

Main Files

- **cmd/websocket-server/main.go** - entry point, Gin router, serves chat HTML, upgrades to WebSocket
- **internal/websocket/hub.go** - central hub managing clients, rooms, registration, and broadcasting
- **internal/websocket/ws_client.go** - readPump (receives messages + typing) and writePump (sends messages + PING)

WebSocket Upgrade and Client Creation (main.go)

```
router.GET("/ws", handleWebSocket)

func handleWebSocket(c *gin.Context) {
    username := c.Query("username")
    room := c.Query("room")
    if room == "" {
        room = "general"
```

```
    }
    if username == "" {
        c.JSON(http.StatusBadRequest, gin.H{"error": "username required"})
        return
    }

    conn, err := upgrader.Upgrade(c.Writer, c.Request, nil)
    if err != nil {
        return
    }

    client := &websocket.Client{
        Hub:      hub,
        Conn:     conn,
        Send:     make(chan []byte, 256),
        Username: username,
        Room:     room,
    }

    hub.Register <- client

    // Send message history
    history := hub.GetMessageHistory(room)
    for _, msg := range history {
        data, _ := json.Marshal(msg)
        client.Send <- data
    }

    // Broadcast join message
    joinMsg := websocket.Message{Type: "system", Text: username + " joined the room",
...}
    data, _ := json.Marshal(joinMsg)
    hub.Broadcast <- data

    go client.WritePump()
    go client.ReadPump()
}
```

Hub Broadcasting and Room Management (hub.go)

```
func (h *Hub) Run() {
    for {
        select {
        case client := <-h.Register:
            h.clients[client] = true
            if _, ok := h.rooms[client.Room]; !ok {
                h.rooms[client.Room] = make(map[*Client]bool)
            }
            h.rooms[client.Room][client] = true

        case client := <-h.Unregister:
            // Remove from clients and room
            delete(h.rooms[client.Room], client)

        case data := <-h.Broadcast:
            var msg Message
            json.Unmarshal(data, &msg)

            // Save to history if chat/system
            if msg.Type == "chat" || msg.Type == "system" {
                h.messageHistory = append(h.messageHistory, msg)
                if len(h.messageHistory) > 50 {
                    h.messageHistory = h.messageHistory[1:]
                }
            }

            // Send only to clients in the same room
            for client := range h.rooms[msg.Room] {
                select {
                case client.Send <- data:
                default:
                    // Buffer full → disconnect
                    close(client.Send)
                    delete(h.clients, client)
                    delete(h.rooms[msg.Room], client)
```

```
                    }
                }
            }
        }
}
```

Client Pumps and Heartbeat (ws_client.go)

- **writePump**: sends queued messages + PING every 54 seconds (pongWait = 60s timeout)
- **readPump**: reads incoming messages, unmarshals JSON, sets server fields (time, username, room), re-marshals, broadcasts
- Handles typing events separately

Online Count Endpoint

```
router.GET("/stats", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{"online_users": hub.GetClientCount()})
})
```

5. gRPC

gRPC server runs independently on port 9092 and connects to the same SQLite database as other servers. It is designed as a separate service to demonstrate microservice patterns, even though it currently shares the database.

- Uses HTTP/2 for transport (multiplexing, header compression)
- Strongly typed messages and services via Protocol Buffers (.proto file)
- Unary RPC calls (request-response)
- Code generation for server and client stubs
- Built-in support for error codes and metadata
- Ready for future expansion (e.g., separate manga service)

Main files:

- cmd/grpc-server/main.go – entry point, starts gRPC server
- proto/manga.proto – service and message definitions

- proto/manga.pb.go and manga_grpc.pb.go – generated Go code
- internal/grpc/service.go – implementation of the gRPC service methods

Protocol Buffer Definition (proto/manga.proto)

```
service MangaService {
  rpc GetManga(GetMangaRequest) returns (GetMangaResponse);
  rpc SearchManga(SearchMangaRequest) returns (SearchMangaResponse);
  rpc UpdateProgress(UpdateProgressRequest) returns (UpdateProgressResponse);
}

message GetMangaRequest {
  string id = 1;
}

message GetMangaResponse {
  Manga manga = 1;
}

message Manga {
  string id = 1;
  string title = 2;
  string author = 3;
  repeated string genres = 4;
  string status = 5;
  int32 total_chapters = 6;
  string description = 7;
}
// ... other messages
```

Starting the gRPC Server (main.go)

```
lis, err := net.Listen("tcp", ":9092")
if err != nil {
    log.Fatalf("Failed to listen: %v", err)
}
```

```
grpcSrv := grpc.NewServer()
pb.RegisterMangaServiceServer(grpcSrv, grpc.NewMangaServiceServer(db))

log.Println("gRPC server listening on :9092")
if err := grpcSrv.Serve(lis); err != nil {
    log.Fatalf("Failed to serve: %v", err)
}
```

Service Implementation (internal/grpc/service.go)

```
func (s *MangaServiceServer) GetManga(ctx context.Context, req *pb.GetMangaRequest)
(*pb.GetMangaResponse, error) {
    var manga pb.Manga
    var genresStr string

    err := s.db.QueryRowContext(ctx,
        "SELECT id, title, author, genres, status, total_chapters, description FROM manga
WHERE id = ?",
        req.Id,
    ).Scan(&manga.Id, &manga.Title, &manga.Author, &genresStr, &manga.Status,
&manga.TotalChapters, &manga.Description)

    if err == sql.ErrNoRows {
        return nil, status.Errorf(codes.NotFound, "manga not found: id=%s", req.Id)
    }
    if err != nil {
        return nil, status.Errorf(codes.Internal, "database error: %v", err)
    }

    if genresStr != "" {
        manga.Genres = strings.Split(genresStr, ",")
    }

    return &pb.GetMangaResponse{Manga: &manga}, nil
}
```

Error Handling

Uses gRPC status codes:

- codes.NotFound → manga doesn't exist
- codes.InvalidArgument → bad input
- codes.Internal → database error

CHAPTER 4: CONCLUSION

1. Conclusion:

Throughout the development of MangaHub, our group has successfully built a multi-protocol manga tracking system that demonstrates the core principles of net-centric programming. The project integrates five network protocols: HTTP REST API, TCP, UDP, WebSocket, and gRPC - into an application that allows users to manage their manga library, track reading progress, receive real-time updates, and join in a manga community discussions. Key features include:

- A secure and documented REST API with JWT authentication, manga management, and progress tracking
- Reliable progress synchronization via TCP and low-latency notifications via UDP
- Real-time chat functionality with multiple rooms, typing indicators, and message history
- A gRPC service for potential future microservices expansion
- Interactive Swagger API documentation and comprehensive GoDoc code documentation
- A responsive web interface
- Bonus features such as multiple chat rooms, interactive API docs, and chat history

2. Future works:

- Mobile applications using the existing REST API and gRPC for cross-platform support.
- User profiles with avatars, reading statistics, and recommendations.
- Manga cover images and detailed metadata integration from external APIs
- Push notifications for mobile devices when progress is updated by other users.
- Advanced chat features: private messaging, message editing/deletion, emojis, and file sharing.
- Scalable deployment with Docker and Kubernetes for production use.
- Enhanced security: rate limiting, refresh tokens, and password reset functionality.
- Integration with external manga sources for automatic chapter updates.

3. Acknowledgement:

Our appreciation goes out to our instructor and everyone else who has helped us achieve the project's goals with their tremendous assistance:

- Dr. Le Thanh Son
- MSc. Nguyen Trung Nghia
- @TiagoTaquelim on youtube for the complete tutorials and instructions
- The open-source community - developers of Go, Gin, Gorilla WebSocket, and Swaggo

List of reference

Documentation & Tutorial:

Go Programming Language Documentation: https://go.dev/doc/
Gin Web Framework: https://gin-gonic.com/docs/
Gorilla WebSocket: https://github.com/gorilla/websocket
Tiago's WebSocket tutorial: https://github.com/sikozonpc/realtime-chat-go
https://www.youtube.com/watch?v=NPq3d2HkxWU
Swaggo (Swagger for Go): https://github.com/swaggo/swag
gRPC Go Quick Start: https://grpc.io/docs/languages/go/quickstart/
Tiago's gRPC tutorial: https://github.com/sikozonpc/oms
https://www.youtube.com/watch?v=ea_4Ug5WWYE
SQLite Documentation: https://www.sqlite.org/docs.html
mangahub_cli_manual(reference).pdf
mangahub_usercase(reference).pdf
mangahub_project_spec.pdf

Books & Articles:

"Network Programming with Go" by Jan Newmarch
"Building Microservices with Go" by Nic Jackson
REST API Design Guidelines (various sources from Google, Microsoft)

Tools & Libraries:

Visual Studio Code: https://code.visualstudio.com/
Postman API Client: https://www.postman.com/
Telnet (for TCP testing):

Inspiration & Assets:

Various manga community platforms (MyAnimeList, AniList) for feature ideas
Free icons and UI inspiration from open-source projects