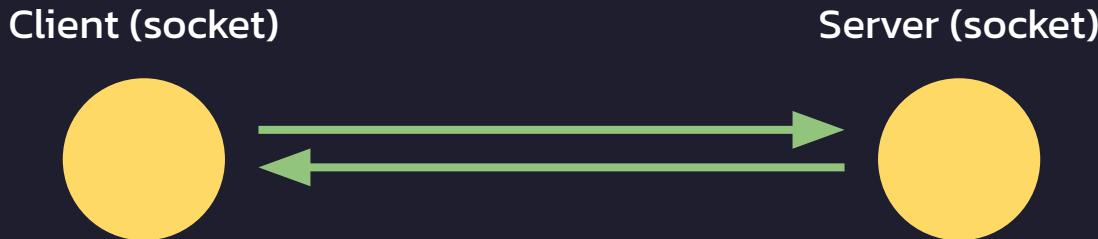


Socket Programming

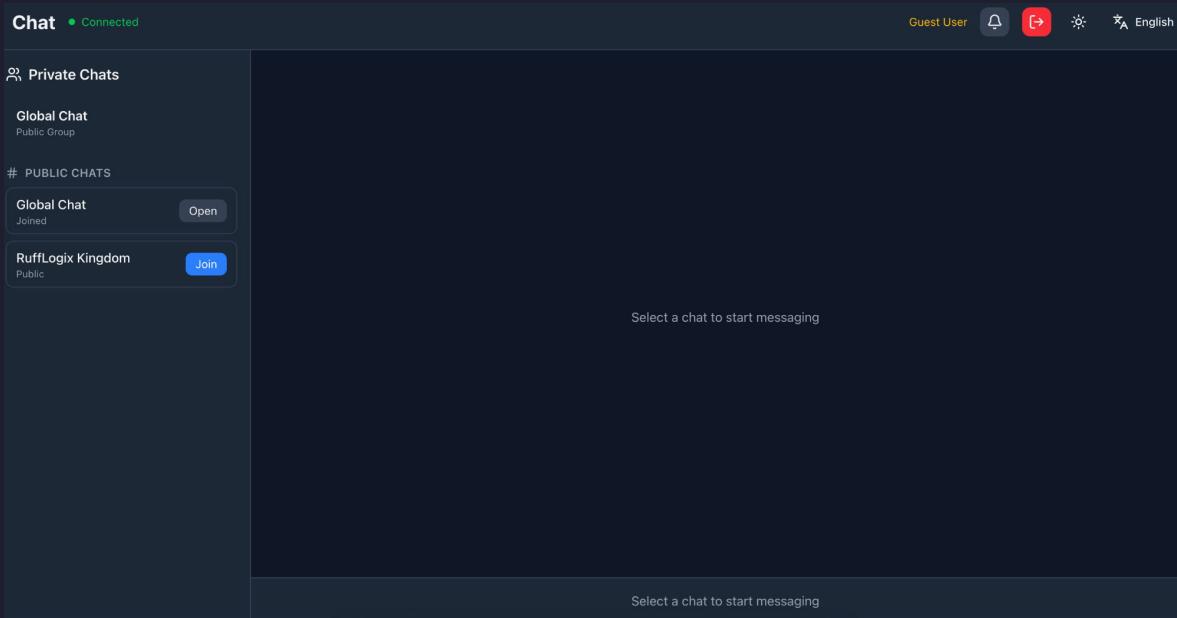
Socket programming is a way to connect 2 nodes on a network to communicate to each other.

One socket listen particular port at the IP address while the other socket while the client reaches out to the server.



Objective

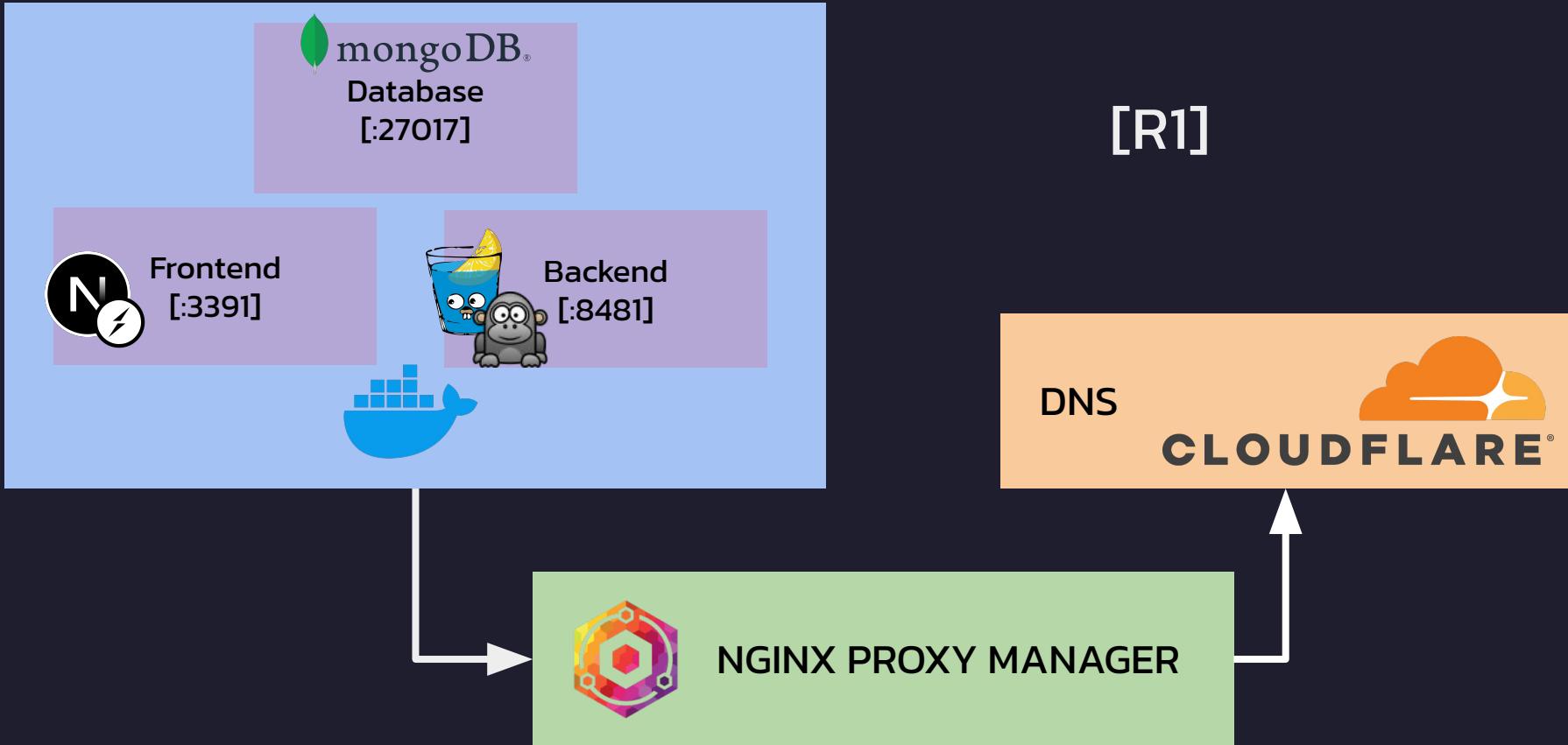
This project is chat web application using web socket for realtime communication.



[PART I]

System Architecture Design

Part I: System Architecture Design



Part I: System Architecture Design

[R2] Socket connections between frontend and backend

```
Socket Programming

const connect = () => {
  // Note: WebSocket doesn't support custom headers in browser
  // Token needs to be passed via query parameter or sent after
  connection
  const token =
    typeof window !== "undefined"
      ? localStorage.getItem("auth_token")
      : null;
  const wsUrl = token
    ? `${WS_BASE_URL}/ws?token=${encodeURIComponent(token)}`
    : `${WS_BASE_URL}/ws`;

  console.log("WebSocket: Connecting to", wsUrl);
  ws.current = new WebSocket(wsUrl);
  // ...
```

```
Socket Programming

func (h *implWSHandler) HandleWS(w http.ResponseWriter, r *http.Request) {
  conn, err := upgrader.Upgrade(w, r, nil)
  if err != nil {
    log.Printf("Failed to upgrade connection: %v", err)
    http.Error(w, "Could not open websocket connection",
    http.StatusBadRequest)
    return
  }
  defer conn.Close()

  var userID int64
  log.Printf("WebSocket connection established from %s", r.RemoteAddr)

  // Set read deadline to prevent hanging connections
  conn.SetReadDeadline(time.Time{})
  // ...
```

[PART II]

Fundamental Implementation

Implementation Details

Server Side:

Connection Handling

- REST: HTTP Routes (/login, /register, etc.)
- SOCKET: websocket support (/ws route)

Room Management

- Map (userId, chatId, rooms)

Event Handling

- When client connects to socket, server will wait for messages
- The transferred data will be in JSON

Implementation Details

Client Side:

Connection

- Use custom React hook
- When user tried to connect, will create new WebSocket client

Message Event

- Will call sendMessage function in React hook to send data via websocket
- The React hook will create JSON payload and send to the server socket

Receiving Event

- The React hook will listen to the event from the server
- When event occur, there is function to handle the event, render web page again

Implementation Details

User Management

- Username must be unique
- Client List
 - There is a list of online users

Chat Room

- Chat Rooms
 - Every chat room are isolated
 - Chat interface

Implementation Details

[R3] Each client must have unique name



Socket Programming

```
func (s *implAuthService) Register(username, password, name, email string)  
(*entity.User, string, error) {  
    // Check if user already exists  
    existingUser, _ := s.userRepo.GetUserByUsername(username)  
    if existingUser != nil {  
        return nil, "", errors.New("username already exists")  
    }  
    // ...
```

Implementation Details

[R4] Each client can see a list of names of all clients that are currently connected to the server including its own name

```
Socket Programming

func (h *implWSHandler) broadcastOnlineUsersList() {
    onlineUserIDs := h.roomService.GetOnlineUsers()

    // Send personalized online user lists to each
    for _, userID := range onlineUserIDs {
        h.sendOnlineUsersList(userID)
    }
}
```

```
Socket Programming

func (s *implRoomService) GetOnlineUsers() []int64 {
    s.mutex.RLock()
    defer s.mutex.RUnlock()

    userIDs := make([]int64, 0, len(s.clients))
    for userID := range s.clients {
        userIDs = append(userIDs, userID)
    }
    return userIDs
}
```

Implementation Details

[R5] Each chat between clients (both Private and Group message) must have its own chat room.

```
Socket Programming

const joinChat = useCallback(
  (chatId: number) => {
    console.log(`Joining chat ${chatId} for user ${userId}`);
    sendEvent({
      type: EVENT_TYPES.JOIN,
      data: {
        chat_id: chatId,
      },
      created_by: userId,
    });
  },
  [sendEvent, userId]
);
```

```
Socket Programming

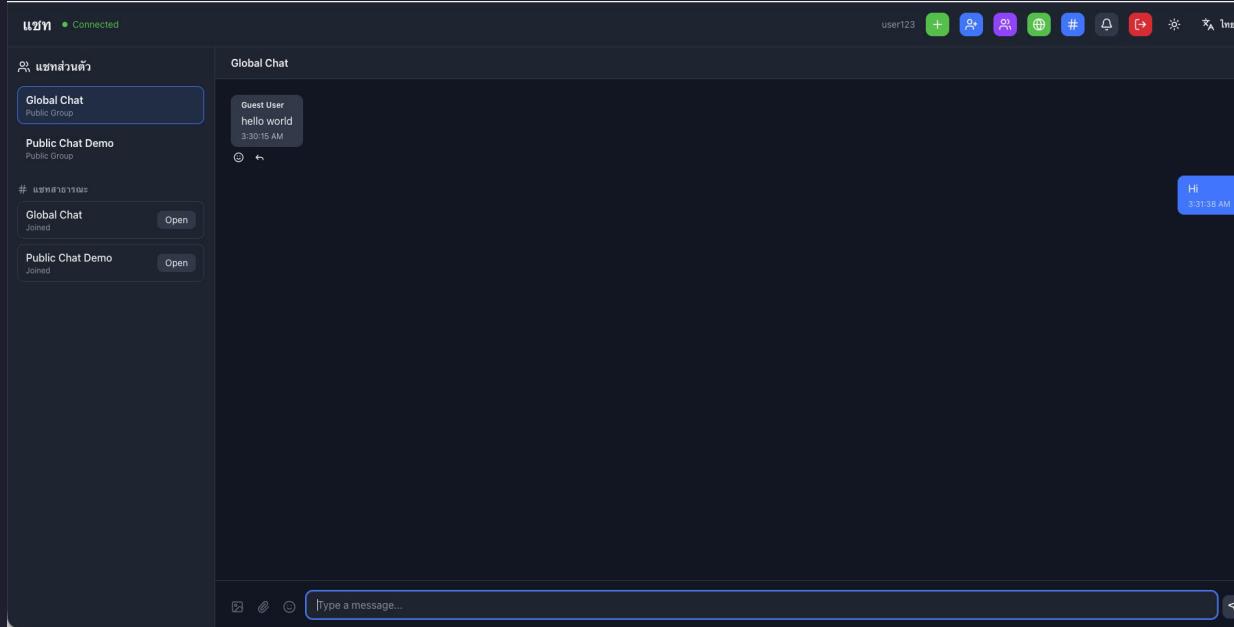
func (h *implHTTPHandler) sendMessage(c *gin.Context) {
  chatID, _ := strconv.ParseInt(c.Param("id"), 10, 64)

  var req struct {
    Content   string `json:"content"`
    Type      string `json:"type" binding:"required"`
    MediaURL string `json:"media_url"`
    FileName  string `json:"file_name"`
    FileSize  int64  `json:"file_size"`
    ReplyToID *int64 `json:"reply_to_id"`
  }

  if err := c.ShouldBindJSON(&req); err != nil {
    c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    return
  }
  // ...
}
```

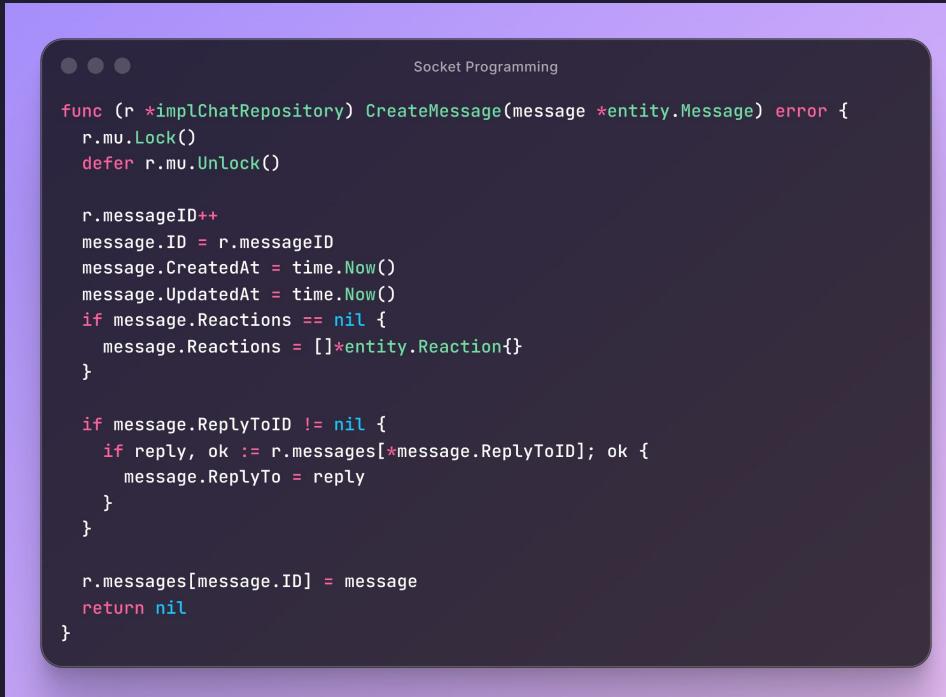
Implementation Details

[R6] Each chat room (both Private and Group message) must include a chat box for sending text messages and chat window for displaying the sent messages and new incoming messages.



Implementation Details

[R7] Each client can send a direct text message to any clients in the list. Only the sender and receiver can see the messages.



```
Socket Programming

func (r *implChatRepository) CreateMessage(message *entity.Message) error {
    r.mu.Lock()
    defer r.mu.Unlock()

    r.messageID++
    message.ID = r.messageID
    message.CreatedAt = time.Now()
    message.UpdatedAt = time.Now()
    if message.Reactions == nil {
        message.Reactions = []*entity.Reaction{}
    }

    if message.ReplyToID != nil {
        if reply, ok := r.messages[*message.ReplyToID]; ok {
            message.ReplyTo = reply
        }
    }

    r.messages[message.ID] = message
    return nil
}
```

Implementation Details

```
...                               Socket Programming

func (h *implHTTPHandler) createChat(c *gin.Context) {
    var req struct {
        Name      string `json:"name" binding:"required"`
        Description string `json:"description"`
        IsPublic   bool   `json:"is_public"`
        Type      string `json:"type" binding:"required"`
    }

    if err := c.ShouldBindJSON(&req); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    // Get user ID from context (assumes auth middleware sets this)
    userID := c.GetInt64("user_id")

    chat := &entity.Chat{
        Type:      entity.ChatType(req.Type),
        Name:      req.Name,
        Description: req.Description,
        IsPublic:  req.IsPublic,
        CreatedBy: userID,
    }

    if err := h.chatService.CreateChat(chat); err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    // Add creator as admin member
    h.chatService.AddMember(chat.ID, userID, "admin")

    c.JSON(http.StatusCreated, chat)
}
```

[R8] Each client can create a chat group(s), which initially includes only themselves as a member.

Implementation Details

[R9] Each client can see a list of all existing chat groups (with the member list) created by any clients.

>> More in Next Slide <<

```
Socket Programming

func (h *implHTTPHandler) getAllChats(c *gin.Context) {
    userID, exists := c.Get("user_id")
    if !exists {
        c.JSON(http.StatusUnauthorized, gin.H{"error": "User not authenticated"})
        return
    }

    userIDInt := userID.(int64)

    // Get all chats from the service
    allChats, err := h.chatService.GetAllChats()
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    // Get online users for member status
    onlineUserIDs := h.roomService.GetOnlineUsers()
    onlineUsersMap := make(map[int64]bool)
    for _, id := range onlineUserIDs {
        onlineUsersMap[id] = true
    }

    // Define response types
    // ...

    var response []ChatWithMembersResponse
    for _, chat := range allChats {
        // Filter chats: show public chats or private chats where user is a member
        // ...
    }

    c.JSON(http.StatusOK, response)
}
```

Implementation Details

[R9] Each client can see a list of all existing chat groups (with the member list) created by any clients.

```
Socket Programming

members, err := h.chatService.GetMembers(chat.ID)
if err != nil {
    continue // Skip chats with member fetch errors
}

var membersWithStatus []ChatMemberWithStatus
for _, member := range members {
    user, err := h.userRepository.GetUserByNumericID(member.UserID)
    if err != nil {
        user = nil // Skip user details if we can't fetch
    }
    membersWithStatus = append(membersWithStatus, ChatMemberWithStatus{
        ChatMember: *member,
        User:       user,
        IsOnline:   onlineUsersMap[member.UserID],
    })
}

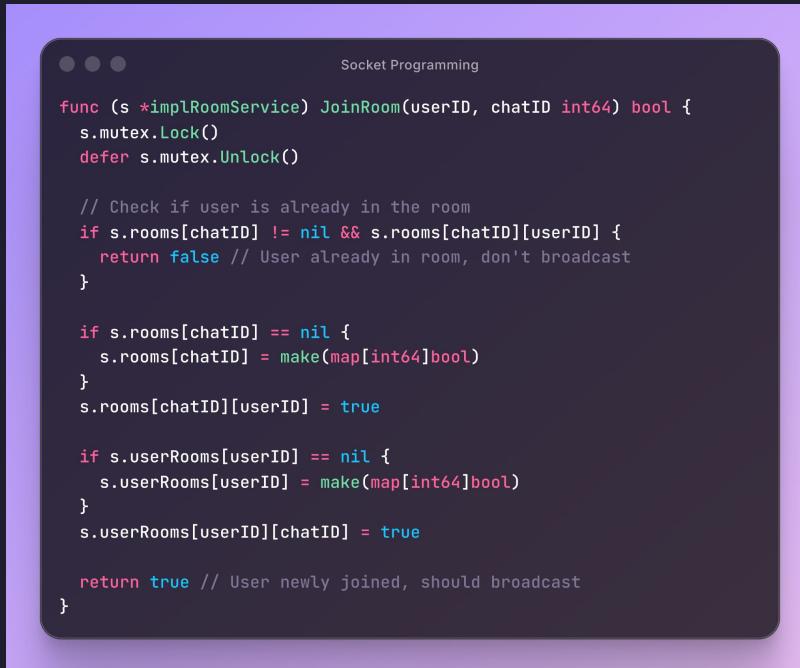
response = append(response, ChatWithMembersResponse{
    Chat:  *chat,
    Members: membersWithStatus,
})
```

```
Socket Programming

if !chat.IsPublic {
    // Check if user is a member of this private chat
    isMember := false
    members, err := h.chatService.GetMembers(chat.ID)
    if err != nil {
        continue // Skip chats with member fetch errors
    }
    for _, member := range members {
        if member.UserID == userIDInt {
            isMember = true
            break
        }
    }
    if !isMember {
        continue // Skip private chats where user is not a member
    }
}
```

Implementation Details

[R10] Clients can choose to join a group chat only by themselves; they are not added automatically by the group creator.



Socket Programming

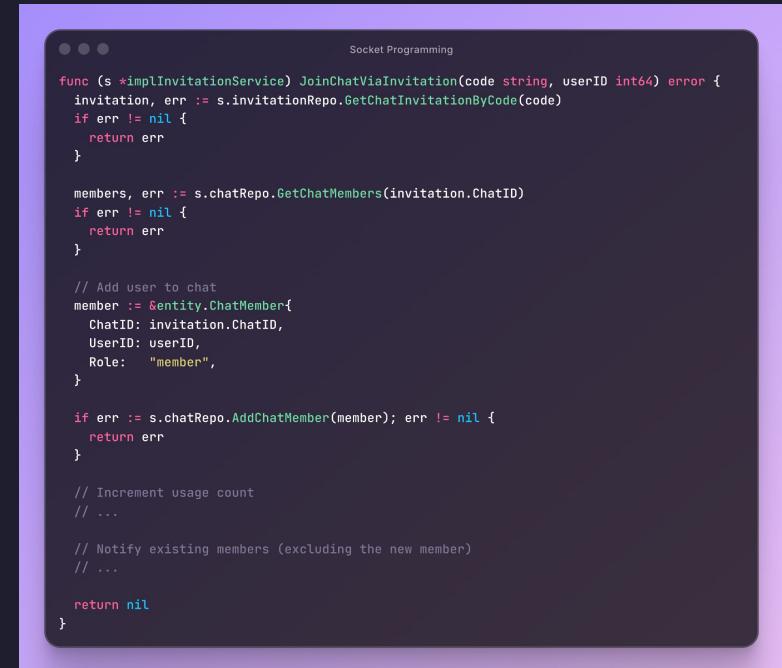
```
func (s *implRoomService) JoinRoom(userID, chatID int64) bool {
    s.mutex.Lock()
    defer s.mutex.Unlock()

    // Check if user is already in the room
    if s.rooms[chatID] != nil && s.rooms[chatID][userID] {
        return false // User already in room, don't broadcast
    }

    if s.rooms[chatID] == nil {
        s.rooms[chatID] = make(map[int64]bool)
    }
    s.rooms[chatID][userID] = true

    if s.userRooms(userID) == nil {
        s.userRooms(userID) = make(map[int64]bool)
    }
    s.userRooms(userID)[chatID] = true

    return true // User newly joined, should broadcast
}
```



Socket Programming

```
func (s *implInvitationService) JoinChatViaInvitation(code string, userID int64) error {
    invitation, err := s.invitationRepo.GetChatInvitationByCode(code)
    if err != nil {
        return err
    }

    members, err := s.chatRepo.GetChatMembers(invitation.ChatID)
    if err != nil {
        return err
    }

    // Add user to chat
    member := &entity.ChatMember{
        ChatID: invitation.ChatID,
        UserID: userID,
        Role:   "member",
    }

    if err := s.chatRepo.AddChatMember(member); err != nil {
        return err
    }

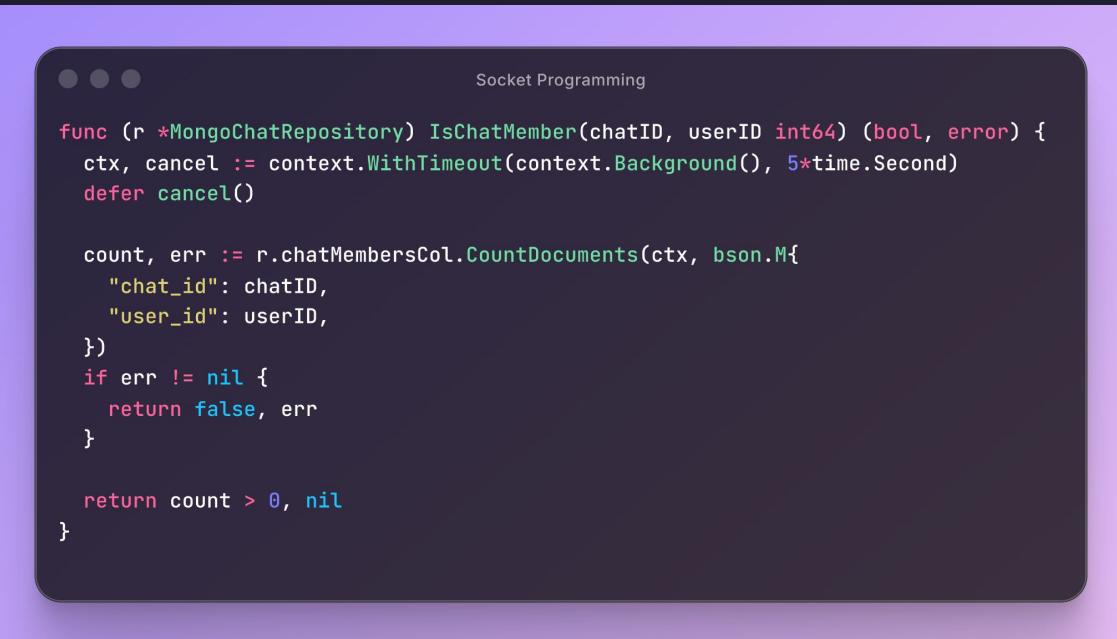
    // Increment usage count
    // ...

    // Notify existing members (excluding the new member)
    // ...

    return nil
}
```

Implementation Details

[R11] Each client can send a text message to the group that they joined. Only the members of the chat group can see the messages.



The screenshot shows a mobile application window titled "Socket Programming". At the top, there are three gray dots. Below the title, the code for the `IsChatMember` function is displayed in a dark-themed code editor:

```
func (r *MongoChatRepository) IsChatMember(chatID, userID int64) (bool, error) {
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    count, err := r.chatMembersCol.CountDocuments(ctx, bson.M{
        "chat_id": chatID,
        "user_id": userID,
    })
    if err != nil {
        return false, err
    }

    return count > 0, nil
}
```

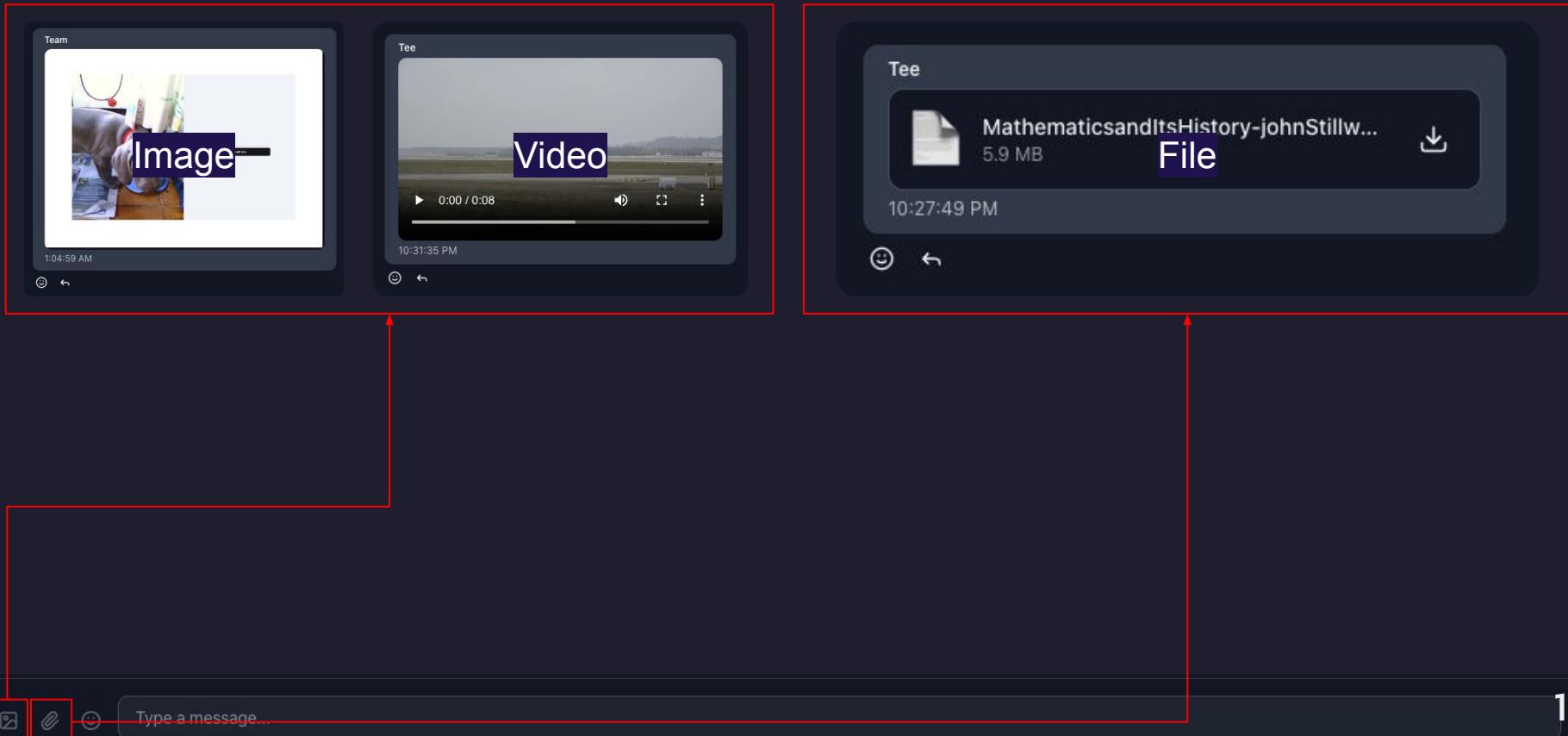
[PART III]
Special Point

Special Point 1: Friend System

The screenshot displays several components of the friend system:

- Add Friend (เพิ่มเพื่อน) Pop-up:** Shows tabs for "เพิ่มด้วยชื่อผู้ใช้" (Add by username) and "ลิงก์เชิญ" (Invite link). A text input field contains "ไส้กรอกใช้ชื่อ ID" (Filler use ID), and a button says "ส่งคำขอเป็นเพื่อน" (Send friend request).
- Friend Invitation Link Pop-up:** Shows the invitation link: <https://network.ruffblitz.com/invite/friend/8MU>. It includes a copy icon and a note that it expires on 11/24/2025, 2:17:39 AM.
- Friends List (เพื่อน (1)):** Shows 1 offline friend: John Doe (@tokyo683). An "Offline" status indicator is present.
- Invitation Failed Alert:** A red circular icon with an 'X' is shown above the text "Invitation Failed" and the message "you are already friends with this user". A blue "Go to Chat" button is at the bottom.
- Invitation Accepted Alert:** A green circular icon with a checkmark is shown above the text "Invitation Accepted!" and the message "Friend invitation accepted successfully!". Below it, a note says "Redirecting to chat...".
- Friend Requests (แจ้งเตือนเพื่อนใหม่) Panel:** Shows two notifications:
 - New Friend Request:** Someone accepted your friend invitation on 11/17/2025, 1:35:29 AM. Status: Accepted.
 - New Friend Request:** Someone accepted your friend invitation on 11/17/2025, 2:18:53 AM. Buttons: Accept (green) and Reject (red).

Special Point 2: Sending Photo, File, Video



Special Point 3: Reply Message

John Doe
สวัสดีครับ
2:22:55 AM

↳ Replying to: สวัสดีครับ...

Hello World
2:23:01 AM

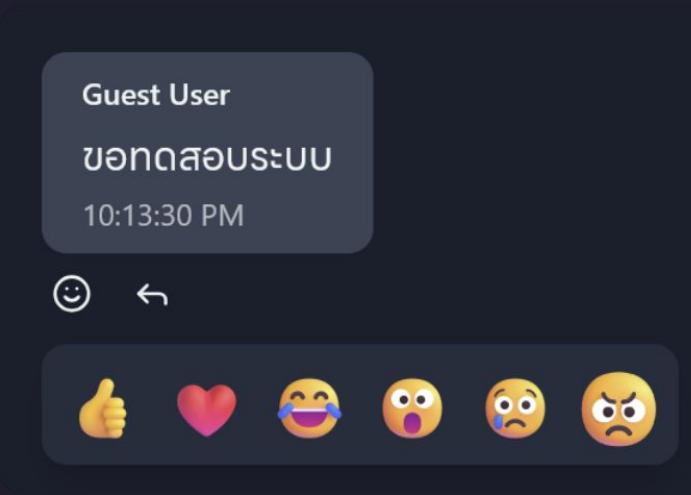
↳ Replying to: Hello World...

Hello World 2
2:23:16 AM



Type a message...

Special Point 4: React Message with Emojis



Special Point 5: Translate UI

The screenshot displays a user interface for adding friends, comparing the English version on the left with the Thai version on the right. Both versions are enclosed in a red rectangular box.

English Version (Left):

- Add Friend** button
- Invite Link** tab (highlighted)
- Friend Invitation Link** button
- https://network.ruffblitz.com/invite/friend/ft9Xl** URL
- Copy** icon
- Expires: 11/24/2025, 2:24:54 AM**

Thai Version (Right):

- เพิ่มเพื่อน** button
- ลิงก์เชิญ** tab (highlighted)
- Friend Invitation Link** button
- https://network.ruffblitz.com/invite/friend/qOb'** URL
- Copy** icon
- .expires: 11/24/2025, 2:24:42 AM**

At the top of the page, there is a navigation bar with the following elements:

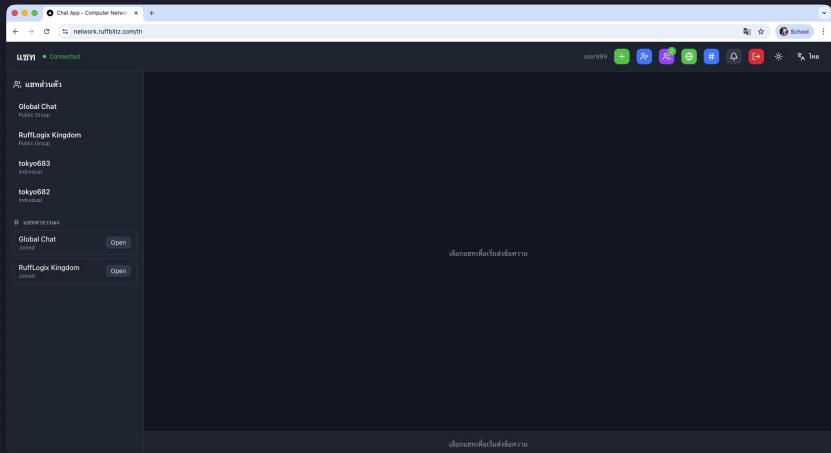
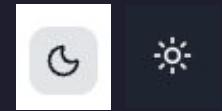
- User icon: user999
- Profile picture icon
- Friends icon: 2 notifications
- Globe icon: Internationalization
- # icon: Hashtag
- Notification bell icon
- Share icon: Red background
- Sun icon: Light mode
- Flag icon: Language selection (English)

To the right of the language selection, there is a dropdown menu with "English" and "ไทย" options.

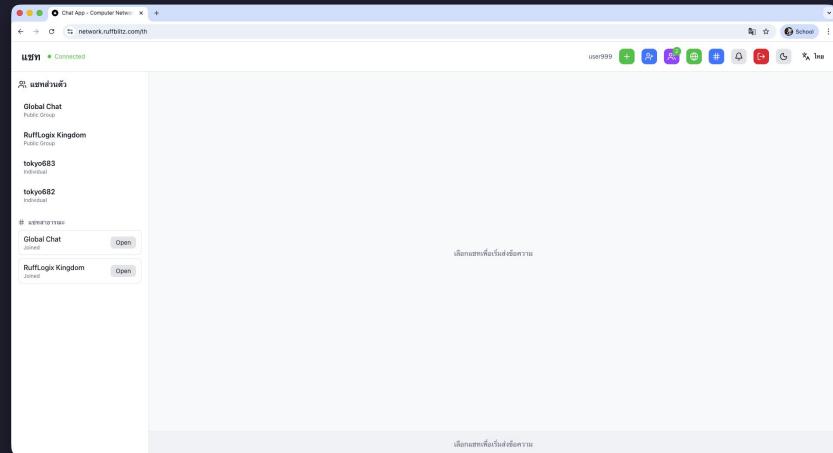
EN

TH

Special Point 6: Light/Dark Theme



Dark Theme



Light Theme

Special Point 7: Authentication

เข้าสู่ระบบ

ชื่อผู้ใช้

รหัสผ่าน

เข้าสู่ระบบ

Need an account? Register

Continue as Guest

Guests can only access public chats

Login

This image shows the login screen of a mobile application. It features a dark-themed interface with light-colored input fields. At the top is a large blue button labeled "เข้าสู่ระบบ" (Login). Below it are two input fields: one for "ชื่อผู้ใช้" (Username) and one for "รหัสผ่าน" (Password). Further down are two more input fields, also for "ชื่อผู้ใช้" and "รหัสผ่าน". At the bottom left is a blue button labeled "Continue as Guest", and at the bottom right is a link "Need an account? Register". A note at the very bottom states that guests can only access public chats.

สมัครสมาชิก

Name

ชื่อผู้ใช้

Email

รหัสผ่าน

สร้างบัญชี

Have an account? Login

Continue as Guest

Guests can only access public chats

Register

This image shows the registration screen of a mobile application. It has a similar dark-themed design to the login screen. At the top is a large blue button labeled "สมัครสมาชิก" (Register). Below it are four input fields: "Name", "ชื่อผู้ใช้" (Username), "Email", and "รหัสผ่าน" (Password). At the bottom is a blue button labeled "สร้างบัญชี" (Create Account). Below this button are links for "Have an account? Login" and "Continue as Guest". A note at the very bottom states that guests can only access public chats.

1. Registered User
2. Guest User

Special Point 8: Public/Private Group Chat

#1 Private Group

คำเชิญ

ออกจากรหัส

สร้างแชท

ชื่อแชท *

คำอธิบาย

สาธารณะ

แชทส่วนตัว

ยกเลิก สร้างแชท

สร้างแชท

ชื่อแชท *

คำอธิบาย

สาธารณะ

แชทสาธารณะ

ยกเลิก สร้างแชท

คำเชิญแชท

ชื่อแชท

#1 Private Group

สร้างลิงก์เชิญ

Group Invitation Link

<https://network.ruffblitz.com/invite/chat/yKKM>

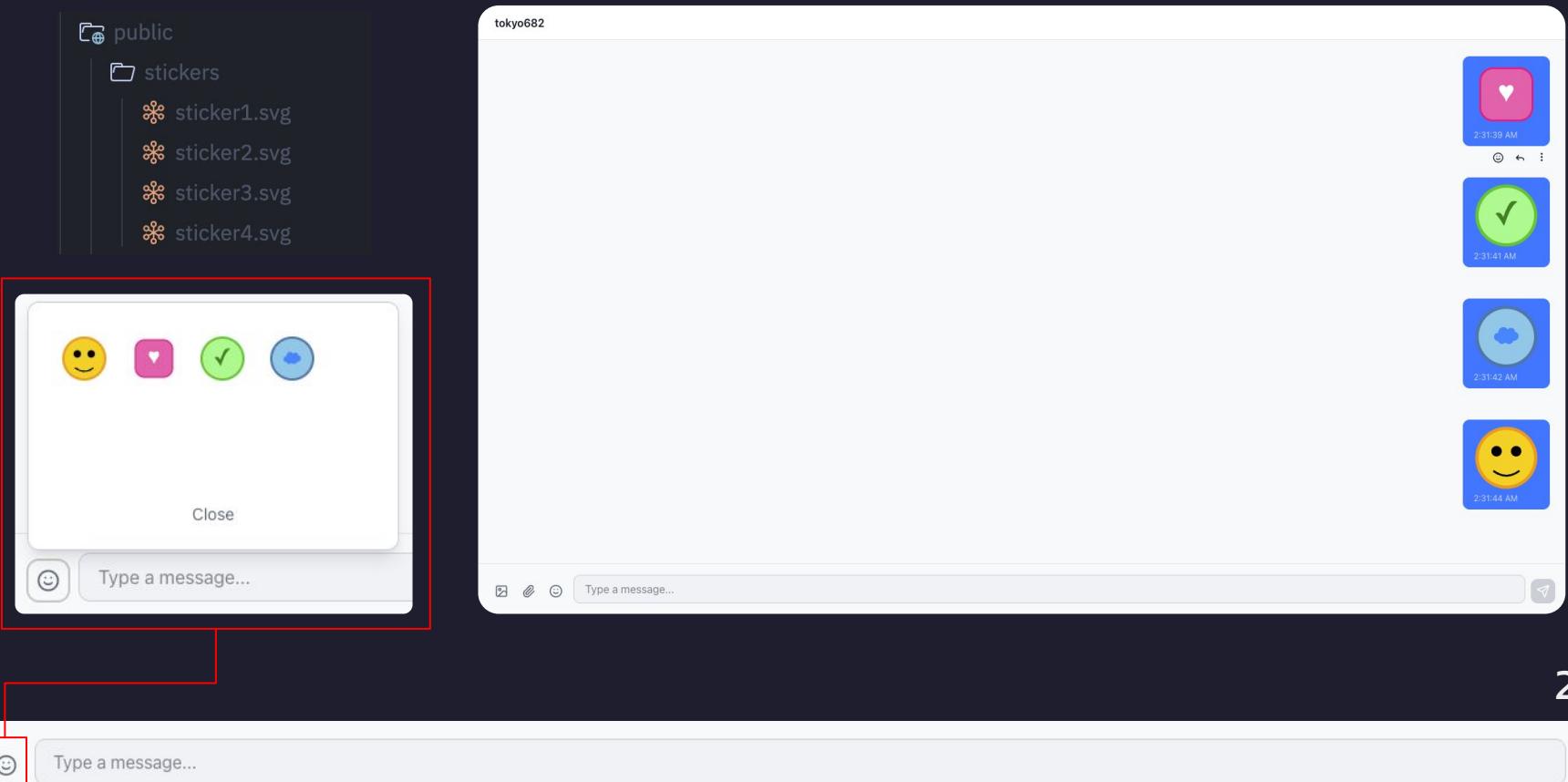
Expires: 11/24/2025, 2:31:08 AM

Create Invitation

Create Private Chat

Create Public Chat

Special Point 9: Stickers



Special Point 10: Show Users in Chat

The screenshot displays three separate chat lists, each with a title, a description, a member count, and a list of members.

- Global Chat**: A public chat room for everyone. Members (23): ONLINE (4) - Tee, John Doe, deammy, user999; OFFLINE (19) - Guest User, Guest User, Guest User, Guest User, someone, +14 more.
- RuffLogix Kingdom**: Hode Hode. Members (9): ONLINE (3) - Tee, user999, John Doe; OFFLINE (6) - someone, Guest User, Guest User, Nattarin, Guest User, +1 more.
- No description**: Individual. Members (2): ONLINE (2) - user999, John Doe.

Show a list of chats that includes chat members for only

1. public chats
2. chats where the user is a member

Special Point 11: Show Current Online Users

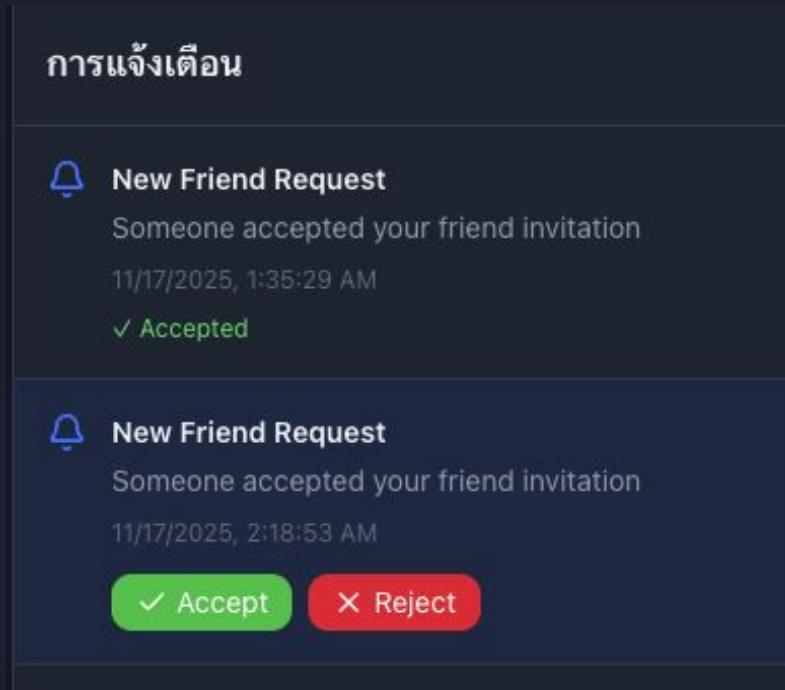
Show all users that are currently online

kto ผู้ใช้ออนไลน์ (4) X

User Icon	User Name	Handle	Status
J	John Doe	@tokyo683	Online
U	user999	@user999	Online
D	deammy	@Deammy	Online
T	Tee	@tokyo682	Online

Special Point 12: Notification

Show a notification when the user receives an invitation to join or add a friend



Source Code

<https://github.com/RuffLogix/computer-network-project>

Live Demo

<https://network.ruffblitz.com>

Thanks for Your Attention