LMMT-SecureDove

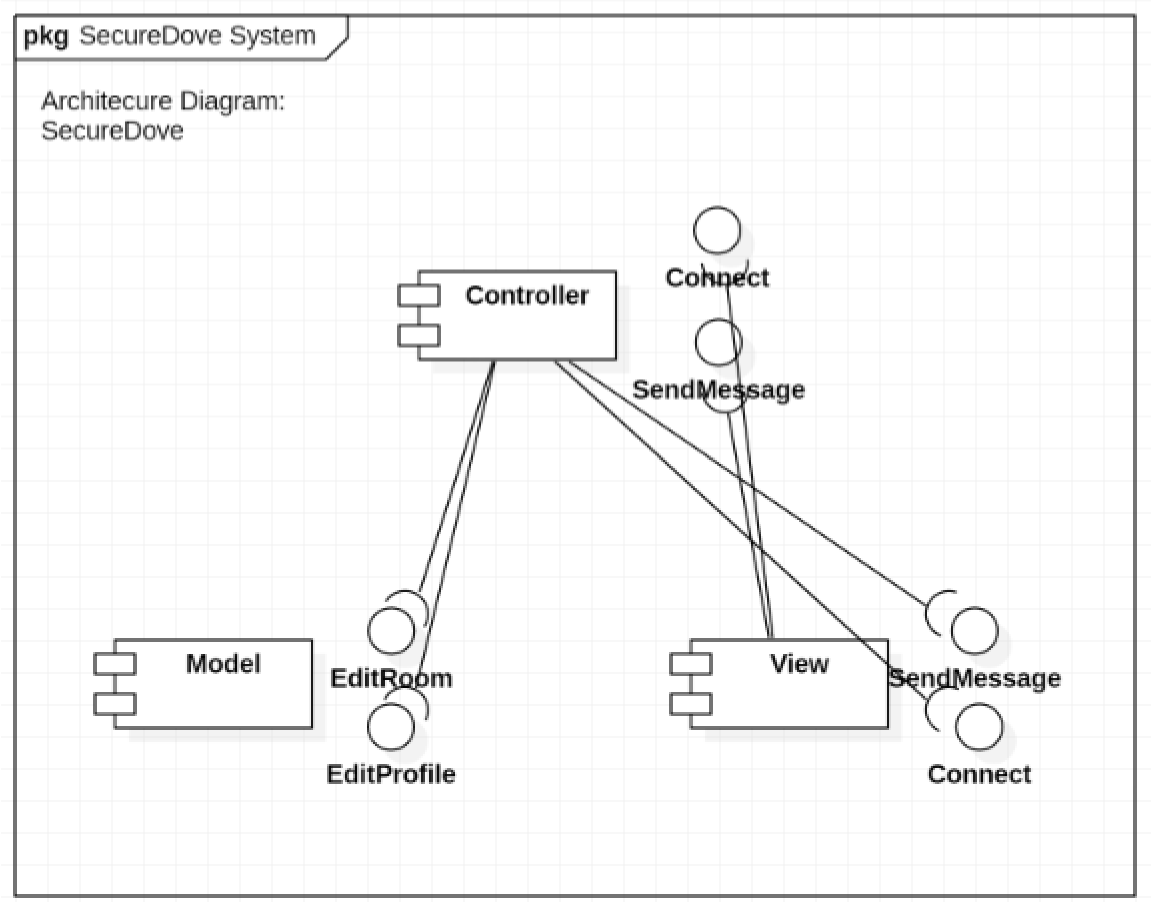
CPTS 428 Fall 2023
Professor Haipeng Cai

Logan Kloft
Manjesh Puram
Matthew Gerola
Taylor West

Table of Contents

Architectural Design

The best way to describe our architecture is through the MVC architecture. The following component diagram displays the high-level view of our architecture.



At a high level, the model provides objects to keep track of user and room information. User information is referred to as a ‘Profile’ and room information is called a ‘Room’. In the backend, these objects are implemented as dictionaries which provide fast lookup based on Room and Profile ids.

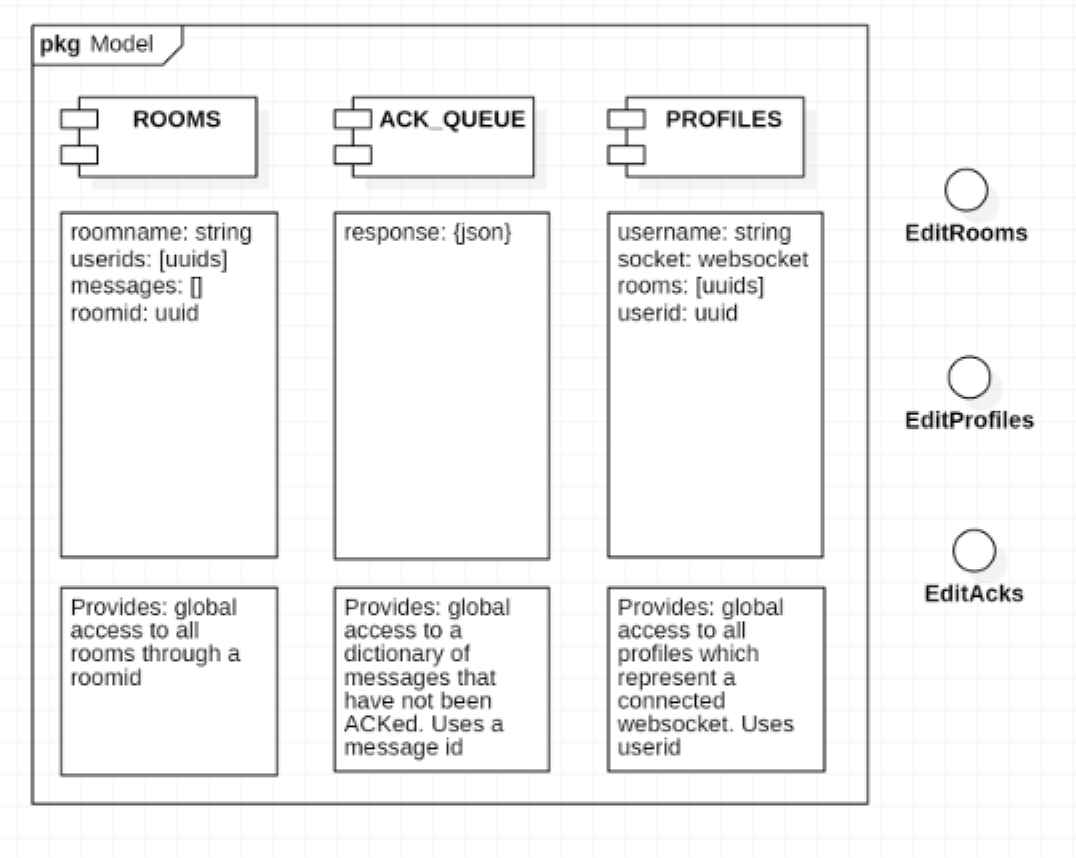
The controller is the actual server that listens for connections. Upon receiving a connection, the Controller will handle messages from the View which might include updating model information or broadcasting messages to Profiles. In the backend, the server is a websocket server from Python’s standard library of modules.

The view is an interface used for connecting to the server, sending requests to the server, and responding to responses from the server. On the frontend, the view uses React.

Component Design

In this part of the document, we will take a deep dive into each component of the architecture diagram.

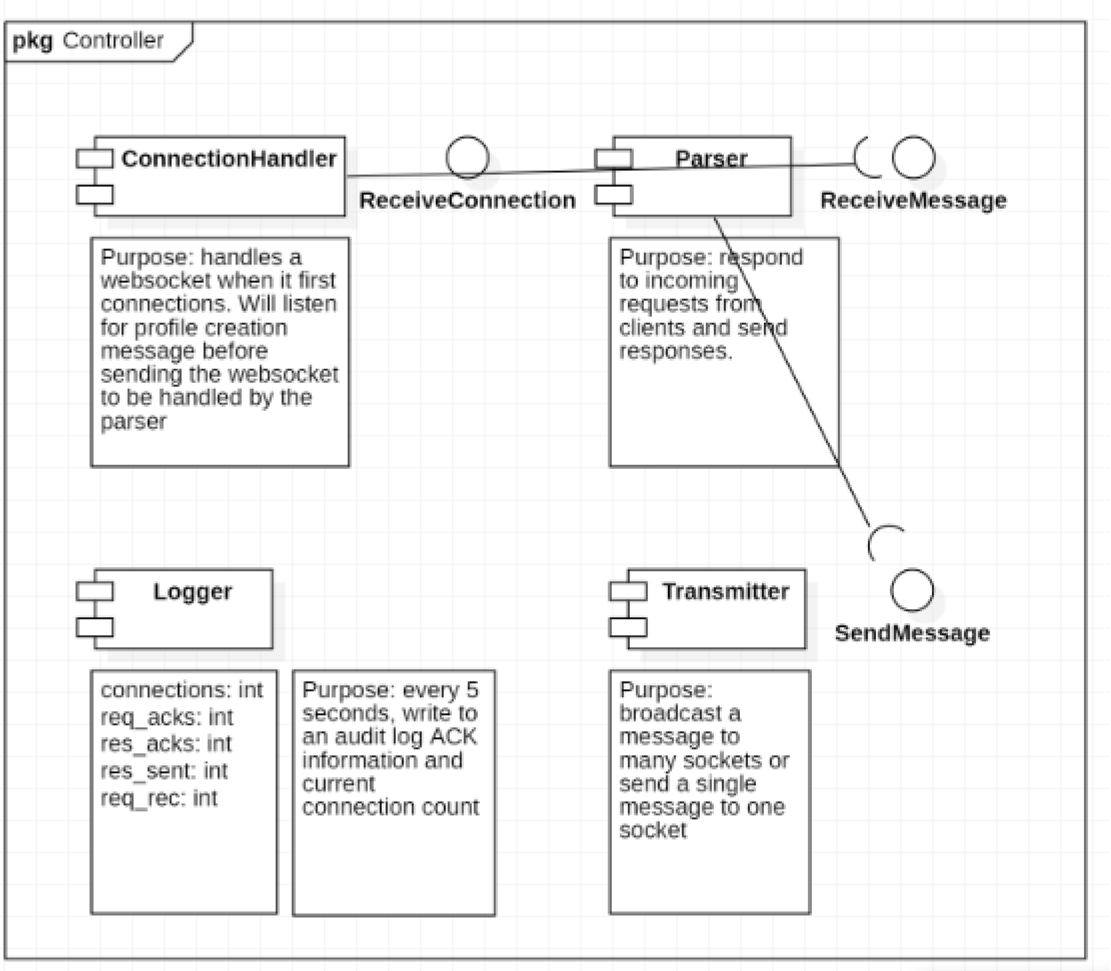
Model


pkg Model
ROOMS
ACK_QUEUE
PROFILES
roomname: string
userids: [uuids]
messages: []
roomid: uuid
response: {json}
username: string
socket: websocket
rooms: [uuids]
userid: uuid
Provides: global access to all rooms through a roomid
Provides: global access to a dictionary of messages that have not been ACKed. Uses a message id
Provides: global access to all profiles which represent a connected websocket. Uses userid
EditRooms
EditProfiles
EditAcks

The above shows the three objects that Model makes available to other components. The purpose of the model is to store and provide access to state information during runtime. A room contains profiles that are in the room. It also contains all messages that were sent to this room. The controller forwards ROOM information to the View and edits a ROOM or creates a PROFILE based on messages from the View.

The ACK_QUEUE object has no meaning to an outside viewer, but on the inside it allows the server to keep track of how many of the messages that it has sent to the client were received. How this interaction works is explained in the controller component.

Controller

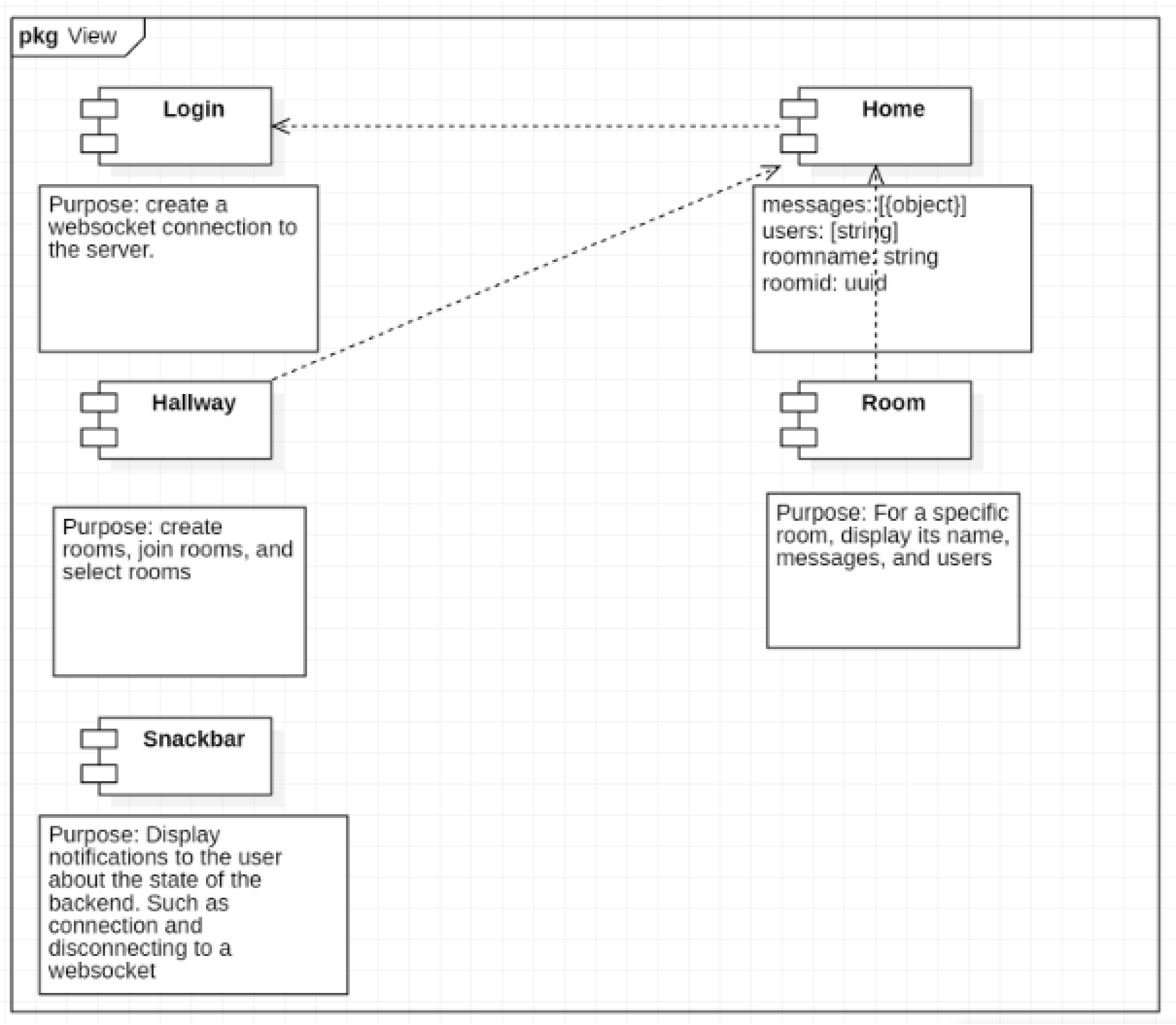


pkg Controller
ConnectionHandler
ReceiveConnection
Parser
ReceiveMessage
Purpose: handles a websocket when it first connections. Will listen for profile creation message before sending the websocket to be handled by the parser
Purpose: respond to incoming requests from clients and send responses.
Logger
Transmitter
SendMessage
connections: int
req_acks: int
res_acks: int
res_sent: int
req_rec: int
Purpose: every 5 seconds, write to an audit log ACK information and current connection count
Purpose: broadcast a message to many sockets or send a single message to one socket

The controller waits for incoming websocket connections. Then passes the connection to the ConnectionHandler component. The ConnectionHandler will listen for one message which it assumes is the Profile creation message. The ConnectionHandler will create a profile and bind the socket to that profile before passing the profile to the parser. The parser listens for all incoming requests on the websocket and will handle the requests appropriately. When the parser needs to send a message, it will use the Transmitter which is just used to group together the functions for sending server responses. The transmitter provides an interface for either broadcasting a single message to multiple clients or a single message to a single client.

The logger runs on its own thread and simply prints the listed global variables to a file called audit.txt. After some time, we might decide to analyze the audit file which will display the total requests received and responses sent and how many of those were ACKed or not. ACKing or acknowledging is the process we use to verify that a message was successfully received by the server or client.

View


pkg View
Login
Purpose: create a websocket connection to the server.
Home
messages: [{object}]
users: [string]
roomname: string
roomid: uuid
Hallway
Purpose: create rooms, join rooms, and select rooms
Room
Purpose: For a specific room, display its name, messages, and users
Snackbar
Purpose: Display notifications to the user about the state of the backend. Such as connection and disconnecting to a websocket

The view is the frontend of the application which is written in React. The component names are actually the same as the React Components used in the frontend. Notice that only the Home component has a state that we care about. The rest of the components manage the behavior (visual qualities) of the frontend. The Login component establishes connection to the websocket server and then passes off the baton to the Home component. The Home component contains both the Hallway and Room component. The Home component also contains a way to store the current room’s id, users, and messages. The Hallway component writes this content while the Room component reads and displays changes to this content. More specifically, the Hallway listens for responses from the Server and will update state accordingly. When these updates occur, the Room component will change what it displays. The snackbar component is an additional quality of life feature that displays when the websocket is first connected and when the websocket disconnects.

Data Models

From the deep dive of the architecture, you will remember multiple references to Requests and Responses. We use requests to refer to communication that originates in the client and responses for communication that originates in the server (which may or may not be triggered by a request from the client).

You will notice that these request and response objects share very loose characteristics with a request and response from the HTTP protocol. In order to better understand the data in our program (which is the bloodline of all applications) we list the request and response objects and their meaning inside of the program:

```
# to create a profile
# request:
{
        "type": "profile",
        "verb": "post",
        "id": "mid",
        "content": {"username": "MyName"}
}
# response:
{
  "id": "mid",
  "type": "profile",
  "verb": "post",
  "content": {"username": "MyName", "userid": "a valid uuid"},
}

# to create a room
# request:
{
        "type": "room",
        "verb": "post",
        "id": "mid",
        "content": {"roomname": "MyRoom"}
}
# response:
{
  "id": "mid",
    "type": "room",
    "verb": "post",
    "content": {"roomname": "MyRoom", "roomid": "a valid uuid"},
}

# to get a room using uuid
# request:
{
             "type": "room",
             "verb": "get",
             "id": "mid",
             "content": {"roomid": "a valid uuid"}
}
# response: Messages are in order of time sent. The most recent is last.
{
    "type": "room",
    "verb": "get",
    "id": "mid",
    "content": {
        "roomid": "a valid uuid",
        "roomname": "RoomName",
        "profiles": ["Logan", "Taylor", "Matthew", "Manjesh"],
        "messages": [{"header": "Logan @ 9/24/2023 11:25", "message": "A message"}],
    },
}

# to join a room
# request:
{
    "type": "room",
    "verb": "put",
    "id": "mid",
    "content": {"roomid": "a valid uuid"},
}
# response:
{
    "type": "room",
    "verb": "put",
    "id": "mid",
    "content": {"roomname": "MyRoom", "roomid": "a valid uuid"},
}

# to send a message
# request
{
    "type": "message",
    "verb": "post",
    "id": "mid",
    "content": {"roomid": "a valid uuid", "message": "A message"},
}

# client listen for a new message: - will broadcast message to relevant room
{
    "type": "message",
    "verb": "get",
    "id": "mid",
    "content": {
        "roomid": "A valid uuid",
        "header": "Logan @ 9/24/2023 11:25",
        "message": "A message",
    },
}

# client listen for user to leave a room - will update user list
{
    "type": "room",
    "verb": "delete",
    "id": "mid",
    "content": {
        "roomid": "A valid uuid",
        "profiles": ["Logan", "Taylor", "Matthew", "Manjesh"],
    },
}
```

```
# client listen for user to join a room - will update user list
{
    "type": "room",
    "verb": "patch",
    "id": "mid",
    "content": {"roomid": "A valid uuid", "username": "New user"},
}

# server sends ACK message for client requests
{"type": "ACK", "verb": "post", "content": {"id": "mid"}}

# server receives ACK message from client
{"type": "ACK", "verb": "post", "content": {"id": "mid"}}
```

Software Documentation

The following describes how to install, run, and use the application.
Code Link: https://github.com/LoganKloft/LMMT-SecureDove

Requirements:

- Python
- Pip
- Node
- NPM

Installation:

- Clone the github repository
- Navigate to the ‘client’ directory
- Run ‘npm install’ to install node packages required for the client
- Navigate to the ‘Server’ directory
- Run ‘pip install -r requirements.txt’ to install python modules required for the Server

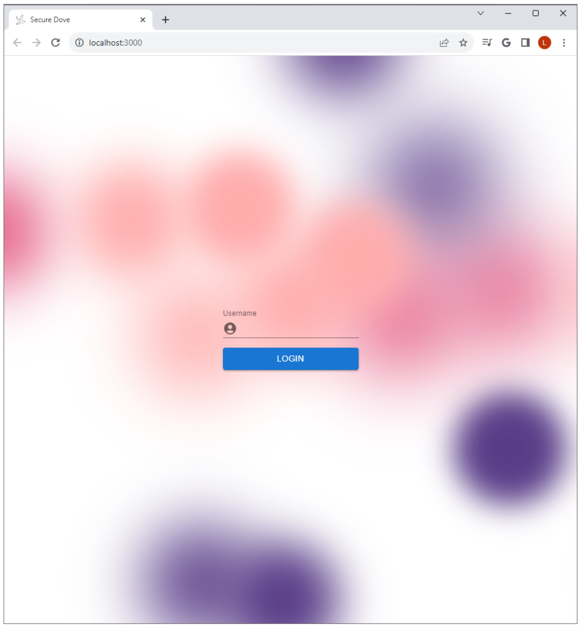
Run:

- First start the server:
    - Open a terminal in the project directory
    - Navigate to the ‘Server’ directory
    - Run ‘python app.py’
- Now start client(s)
    - You may wish to use multiple clients to simulate users communicating
    - Open a terminal in the project directory
    - Navigate to the ‘client’ directory
    - Run ‘npm start’
    - If not open already, open your web client and connect to ‘localhost:3000’
        - Note that if you already have an application running on port 3000, then you will need to replace the port with the one specified when you ran ‘npm start’

Usage:

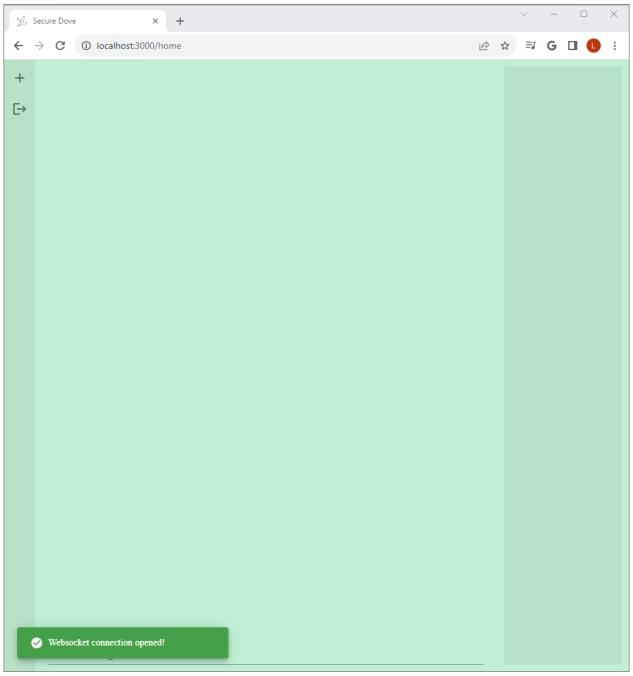
The following will walk you through how to use the application. The prerequisite is that you have two clients open on the browser with the server running in the background. Note that if functionality randomly stops working, restart the process at ‘localhost:3000’. This occurs when an error on the backend goes unhandled which causes the server side of the websocket to disconnect from the client.

1) Login on the clients



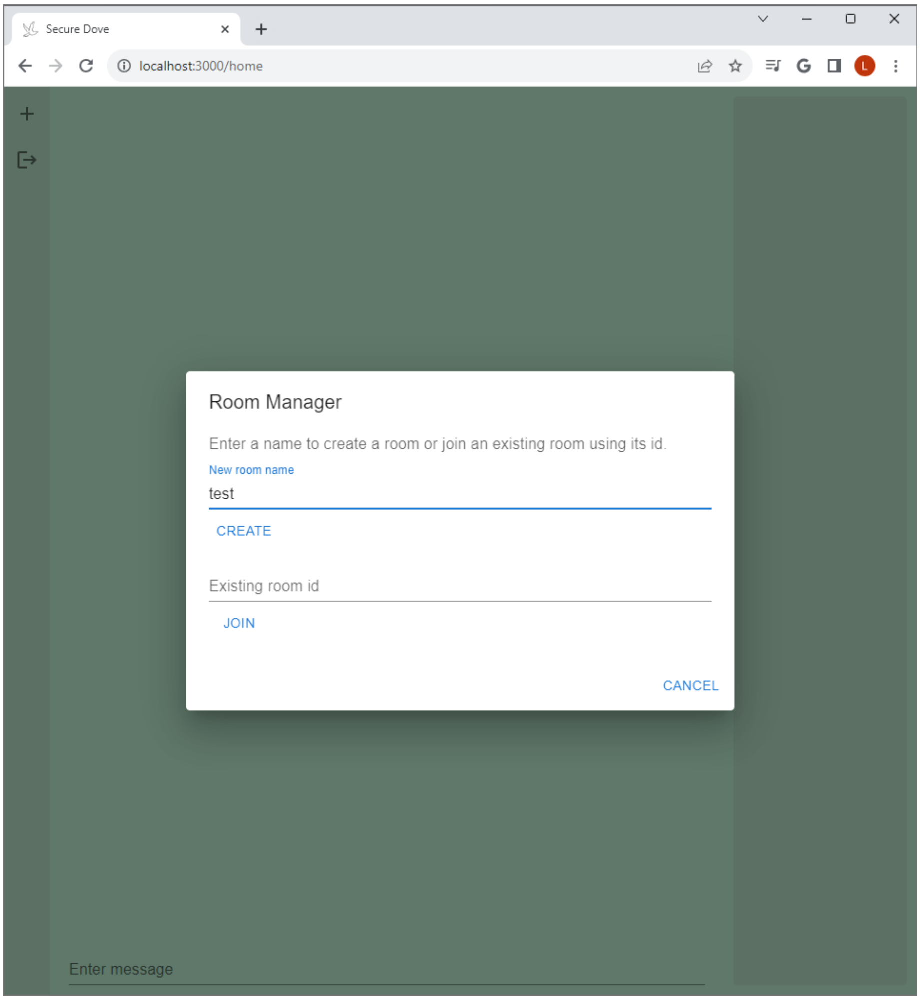
You will be greeted with the following page. Enter a name you wish to go by in the chat application. Then click the ‘login’ button. Make sure to do this for both clients.

2) Create a room



Websocket connection opened!

Next choose one of the clients and click the ‘+’ button at the top left of the screen. This will cause a popup dialog to appear.



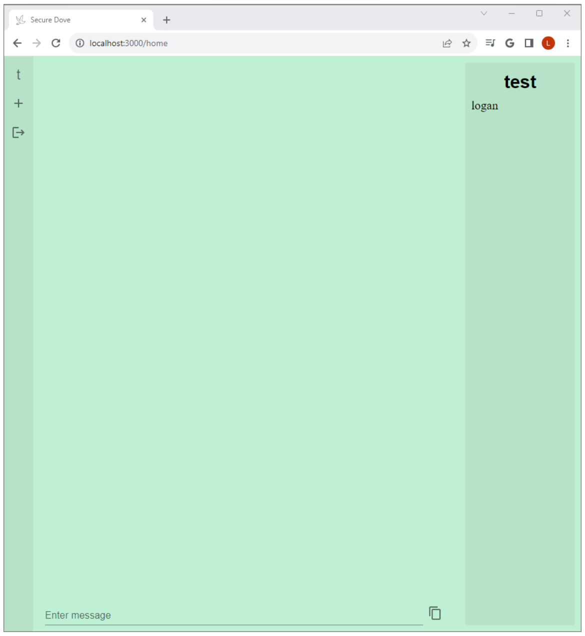
Room Manager

Enter a name to create a room or join an existing room using its id.

New room name: test

Existing room id

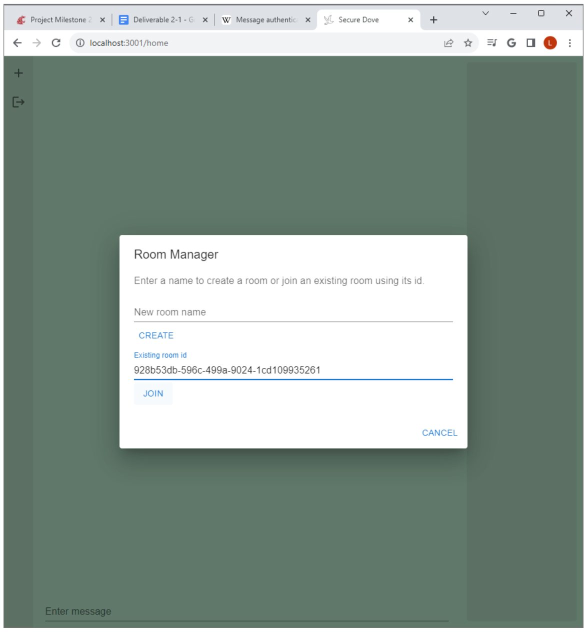
The image above displays the popup dialog mentioned before. Since we want to create a room, we will enter a name in the “New room name” text field. Then click create.



Important: Make sure to select the room you just created in the top left corner. You will see a screen similar to above. Notice that the room name and user list now appears on the right. Less noticeable is the appearance of a copy button directly to the right of the “Enter message” text field. You will use this in the next step.

3) Join the room

Before joining the newly created room on the other client, make sure to click the ‘copy’ icon shown in the last image for ‘Create a room’. This will copy a unique code to your clipboard that allows users to join a room.

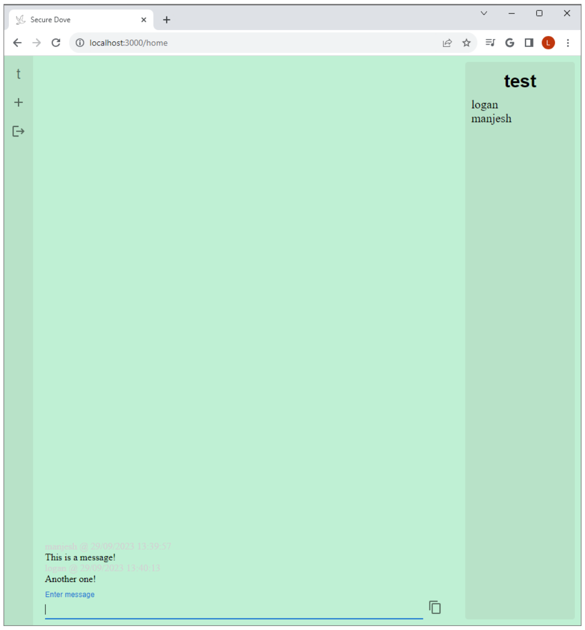


Enter a name to create a room or join an existing room using its id.

928b53db-596c-499a-9024-1cd109935261

When you are ready to join a room on the other client, click the '+' icon just like we did before. However, this time we will copy our room code into the text field labeled 'Existing room id'. Now click the 'Join' button. **Important:** Make sure to click the room that shows up in the top left to join it!

4) Send a message



manjesh @ 29/09/2023 13:39:57
This is a message!
logan @ 29/09/2023 13:40:13
Another one!

Sending a message is simple. Simply type your message into the "Enter message" field and press 'enter' to broadcast the message to everyone in the room. You may send messages from multiple clients and they will appear on all other clients connected to that room. This concludes the messaging features of 'Secure Dove'.

Features & Requirements Revisions

The following are proposed features or requirements from the previous milestone that did not make the cut for this milestone. We briefly describe the purpose of the feature or requirement and then provide rationale as to why they did not appear in this milestone. The previous milestone document (Deliverable 2-1) has been updated to reflect these changes in its use case tables and quality plan.

Removal of MAC requirements:

The team has decided to not to go forward with adding MAC to messages. The team figures that testing our end-to-end encryption is sufficient for the project. Incorporating a MAC adds additional layers of complexity that are not necessary. Additionally, the goal is to secure messages such that they aren't able to be read and end-to-end encryption addresses this (confidentiality), whereas MAC addresses integrity issues (has the message been changed).

Limit Chat Room Mechanics

The proposed chat room functionality was that a user could create, join, leave, and delete a chat room. For this project, the necessary functions are that users are able to create and join a chat room in addition to sending messages to each other. Thus, it is not important that a user necessarily be able to selectively decide to leave (or delete) a chat room. Instead, a user may join as many chat rooms as they wish (or the server can handle) and then upon disconnecting from the web app, the server will handle deletion of chat rooms and removal of users from chat rooms.