⚡ Final Project ⚡

How To Guides

Due Dec 13 at 11:59pm **Points** 100 **Questions** 16 **Available** after Oct 25 at 2:15pm
Time Limit None **Allowed Attempts** Unlimited

Instructions

Overview

The final project in this course is for you to demonstrate your understanding of the course material by designing and implementing a scalable web system. You must come up with your own creative idea for the application domain and determine how to design a scalable system to implement you application idea. What you use and how you use it is entirely up to you with the requirements specified below. In short, you must use multiple back-end services that support the application including a front-end client using Svelte, databases, and an extended event-bus. Your application must incorporate the ideas and concepts covered in the class such as the sending and receiving of events in the back-end as well as using stores and props in the front-end. Your entire application must be wrapped in Docker and the entire system should run using Docker Compose. The goal is to have a completed product by the time you submit the project.

🔥🔥🔥🔥 Keep your project simple. A complicated project will only cause you to work too long and too hard. We are not evaluating every single line of code. We want you to focus and give us what we ask for (and only what we ask for). Spend time creating a bigger project on your own time. We simply want to assess that you learned the material in the course. 🔥🔥🔥🔥

Requirements

The following requirements are **non-negotiable**:

1. You must use Svelte as the front-end framework.
2. Your front-end must include at least 4-5 components.
3. Your front-end must use props.
4. Your front-end must use stores.
5. Your front-end must use CSS or Bootstrap or some other CSS framework or a combination of custom CSS and a framework.
6. You must design and implement an application that has data worth storing in a database (if your idea doesn't, pick another idea).
7. You must use HTTP communication between the front-end and back-end.
8. You must have at least 3-4 micro-services **not including** the front-end client.
9. You must have at least 1 database that is used.
10. Your services must be reasonably interesting, that is, they must do something interesting that supports your application. At the same time, we are not looking for you to implement an AI service.
11. Your services must use morgan to log incoming HTTP requests and Winston to log to the console and a file.
12. You must use PM2 to run your services.
13. At least 1 service must have at least 5 PM2 clones.
14. Each service must have a `package.json` file that includes scripts for running it in `dev` mode (using `nodemon`) and another to run it in production (PM2). You can reference the SnapBook `package.json` files for guidance.
15. You must use Docker and Docker Compose to boot the system in containers.

The following are items that you **are not required** to implement in your project:

1. Authentication and Login.

Video Demonstration

In addition to the application of your knowledge above, you are required to give a 3-5 minute demonstration of your application running and showing us how it works. Your presentation must show us how you boot up your system and how you interact with it in the browser.

Time Management

This assignment should take you approximately 16 hours (approximately the effort of 2 homework assignments). Please plan accordingly to ensure that you are able to take breaks between working on it. It is important that you give yourself some space to allow creativity and thinking to happen. Trying to rush this assignment at the last minute is not the best approach and often leads to a lesser submission and less understanding and knowledge gained by you.

Organization

This assignment is organized as a quiz. Each question asks you to complete a particular part of the problem. You are often asked to include either a written response, a screenshot, or both. We will also ask you to submit you final code that you implemented. The assignment is organized from top to bottom. You typically will need to complete the tasks that precede a task before you complete a particular task itself. The assignment will autosave to allow you to work on it incrementally. The reasoning behind using a quiz to structure the assignment is to make it more clear exactly what we are asking you to do and what we are looking for you to deliver.

Scoring

We will be using the following rubric to score your project submission:

1. **(Front-End/Back-End) HTTP Communication**
    - Excellent (5): Student demonstrated thorough use of HTTP communication between front-end and back-end.
    - Good (4): Student effectively utilized HTTP communication but with minor issues.
    - Satisfactory (3): Student used HTTP communication but with noticeable errors or incomplete implementation.
    - Needs Improvement (2): HTTP communication was attempted but not fully functional or missing key components.
    - Poor (1): No evidence of proper HTTP communication.
2. **(Front-End) Svelte Framework**
    - Excellent (5): Student extensively used Svelte for front-end development, showcasing a deep understanding.
    - Good (4): Student successfully employed Svelte, meeting most requirements.
    - Satisfactory (3): Student utilized Svelte with some issues or incomplete features.
    - Needs Improvement (2): Limited usage of Svelte with major gaps in implementation.
    - Poor (1): Negligible use of Svelte.
3. **(Front-End) Component Implementation**
    - Excellent (5): Student impressively designed and incorporated 4-5 well-constructed front-end components.
    - Good (4): Student successfully included 4-5 components but with minor shortcomings.
    - Satisfactory (3): Student included 4-5 components, however, design or functionality lacked finesse.
    - Needs Improvement (2): Less than 4-5 components were implemented or they were rudimentary in nature.
    - Poor (1): Few or no recognizable components were present.
4. **(Front-End) Props Usage**

- Excellent (5): Student effectively employed props to pass data between components, showi
- Good (4): Student used props adequately but with minor inconsistencies.
- Satisfactory (3): Student attempted to use props, but usage was not seamless or complete.
- Needs Improvement (2): Props usage was minimal or caused significant confusion in the application.
- Poor (1): No proper utilization of props.

5. **(Front-End) Store Implementation**

   - Excellent (5): Student expertly integrated stores within the front-end architecture.
   - Good (4): Student successfully utilized stores, but with minor issues.
   - Satisfactory (3): Student incorporated stores, though their usage was limited or problematic.
   - Needs Improvement (2): Store implementation was attempted but lacked significant features or proper function.
   - Poor (1): No clear evidence of store usage.

6. **(Front-End) CSS and Styling**

   - Excellent (5): Student creatively styled the front-end using CSS, Bootstrap, or other frameworks, resulting in a polished look.
   - Good (4): Student applied CSS/Bootstrap effectively, though some areas lacked refinement.
   - Satisfactory (3): Student used CSS/Bootstrap, but design was inconsistent or lacked appeal.
   - Needs Improvement (2): Styling was minimal, not consistent, or significantly detracted from the user experience.
   - Poor (1): Poor or absent styling.

7. **(Front-End/Back-End) Data Worth Storing**

   - Excellent (5): Student skillfully designed an application with meaningful data for database storage.
   - Good (4): Student successfully included valuable data for database, but with minor gaps.
   - Satisfactory (3): Student incorporated data that could be stored, though its relevance or volume was questionable.
   - Needs Improvement (2): Data worth storing was lacking in substance or coherence.
   - Poor (1): No substantial data for storage.

8. **(Back-End) Micro-Services Quantity and Complexity**

   - Excellent (5): Student impressively implemented 3-4 complex and interesting micro-services.
   - Good (4): Student integrated 3-4 micro-services with moderate complexity.
   - Satisfactory (3): Student incorporated 3-4 micro-services, but complexity or significance was limited.
   - Needs Improvement (2): Micro-services were simplistic or failed to support application's depth.
   - Poor (1): Inadequate or missing micro-services.

9. **(Back-End) Database Integration**

   - Excellent (5): Student expertly integrated at least 1 database into the application architecture.
   - Good (4): Student successfully incorporated a database, but with minor issues.
   - Satisfactory (3): Student included a database, though its integration or purpose was unclear.
   - Needs Improvement (2): Database integration was attempted but incomplete or ineffective.
   - Poor (1): No proper database integration.

10. **(Back-End) Use of PM2 and Clones**

    - Excellent (5): Student adeptly employed PM2 to run services, with at least 1 service having 5 clones.
    - Good (4): Student effectively utilized PM2, but had minor inconsistencies or issues.
    - Satisfactory (3): Student used PM2 but with significant errors or misunderstandings.
    - Needs Improvement (2): PM2 usage was minimal, and/or service clones were lacking.
    - Poor (1): No or incorrect usage of PM2 and clones.

11. **(Back-End) Package.json Scripts**

- Excellent (5): Student meticulously provided accurate package.json scripts for development and production.
- Good (4): Student included package.json scripts, but with minor inaccuracies or issues.
- Satisfactory (3): Student's package.json scripts had noticeable errors or significant gaps.
- Needs Improvement (2): Scripts were inadequate, incorrect, or missing key components.
- Poor (1): Absent or severely flawed package.json scripts.

12. **(Back-End) Docker and Docker Compose Integration**

- Excellent (5): Student skillfully integrated Docker and Docker Compose for containerization.
- Good (4): Student successfully used Docker and Docker Compose, but with minor gaps.
- Satisfactory (3): Student's Docker integration was flawed or incomplete, affecting containerization.
- Needs Improvement (2): Docker and Docker Compose implementation had significant issues or were incomplete.
- Poor (1): No evidence of proper Docker usage.

13. **(Back-End) Logging and Error Checking**

- Excellent (5): Thorough and structured logging across components and services. Clear error messages and robust error handling enhancing reliability.
- Good (4): Adequate logging, informative error messages, and effective error handling contribute to a reliable application.
- Satisfactory (3): Basic logging, error messages might lack detail, and error handling is satisfactory for stable operation.
- Needs Improvement (2): Limited logging, unclear error messages, and inconsistent error handling impact application understanding.
- Poor (1): Inadequate logging, missing/error-prone messages, and weak error handling lead to instability and confusion.

14. **Code Organization and Code Quality**

- **Excellent (5):** Highly organized code with clear structure (folders and files), consistent naming, and effective comments. Top-tier code quality with efficient algorithms and minimal redundancy.
- **Good (4):** Well-organized code with good separation, consistent naming, and decent comments. Code quality is generally solid, with minor areas for improvement.
- **Satisfactory (3):** Acceptably organized code with some structure, but naming and comments need enhancement. Code quality is satisfactory but could be refined.
- **Needs Improvement (2):** Code organization is unclear, naming and comments need improvement. Code quality indicates inefficiencies and room for enhancement.
- **Poor (1):** Disorganized code, inconsistent naming, and lack of comments. Low-quality code with redundancies and disregard for best practices.

15. **Application Idea**

- Excellent (5): Student proposed a highly interesting and creative application idea, demonstrating strong alignment with requirements.
- Good (4): Student proposed an engaging application concept, but some aspects were not fully aligned with requirements.
- Satisfactory (3): Student's application idea was moderately interesting and relevant to the project.
- Needs Improvement (2): Application idea lacked originality or clear connection to requirements.
- Poor (1): Uninteresting or irrelevant application concept.

Coding Expectations

How To Guides

In all assignments, we have high expectations for the JavaScript code you submit. The foremost requirement is that the code should be **readable and easily understandable**. This means adhering to consistent and clear coding conventions, using meaningful variable names, and structuring the code logically. By writing clean and well-organized code, you will demonstrate your ability to produce maintainable and scalable solutions.

Modularity is another crucial aspect of our assignments. We expect you to break down your code into smaller, self-contained functions or modules. This practice not only enhances code reusability but also promotes a more organized and manageable codebase. Moreover, modular code enables easier debugging and encourages collaborative development.

Error-proofing is of paramount importance in any coding task. We anticipate you to incorporate robust error handling mechanisms to prevent unexpected crashes or undesirable outcomes. This includes validating inputs, catching and handling exceptions, and providing informative error messages to aid in troubleshooting.

Additionally, we place a significant emphasis on **well-commented** code. You should provide clear and concise comments throughout your code, explaining the purpose of functions, describing complex operations, and offering insights into your thought process. This not only assists instructors in understanding your approach but also prepares you for the collaborative nature of professional programming.

It is essential that you understand that points will be deducted for poorly written code. By setting these expectations, we aim to instill good coding practices and prepare you to become proficient developers capable of delivering high-quality, scalable, and maintainable code in real-world scenarios.

Academic Honesty Reminder

It is absolutely critical that you complete this work independently. This means that you shall not share code with others, receive code from others, search the internet for the solution, use an AI assistant to solve the problem for you, or do anything besides using your own brain. You are allowed to lookup documentation for JavaScript functions and other aspects of the language and/or APIs that we have covered up to this point. If you want to really learn, then you need to bring it on your own. Any evidence of academic dishonesty will not be tolerated and will be met with severe consequences. Please see the full details of our **academic honesty policy (https://umamherst.instructure.com/courses/6709/pages/syllabus-academic-honesty)** and the use of **AI assistants (https://umamherst.instructure.com/courses/6709/pages/syllabus-ai-assistants)** for further understanding.

Take the Quiz Again

Attempt History

	Attempt	Time	Score
LATEST	Attempt 1	14,414 minutes	100 out of 100

Score for this attempt: **100** out of 100
Submitted Dec 13 at 10:17pm
This attempt took 14,414 minutes.

Question 1	**5 / 5 pts**

Application Idea

Please provide us with a thoughtful 600-word essay of the description of your project.

- What did you design and implement?
- Why did you choose the domain you did?
- Why is your application interesting and why would someone use it?
- Why would your application need to scale?

Your Answer:

I made a simple chatroom where multiple users can connect and communicate with each other. To make the project's scale appropriate for the scope of this assignment, I set the following key restrictions that differ from a typical chat application: (1) users can only send text (no image or video), (2) there is only one central chatroom where all users are in, and (3) users can simply access the URL and start chatting. Therefore, an exciting and natural extension of this project would be to allow users to (1) send other forms of data, (2) make new chatrooms and invite specific users, and (3) create an account and log in.

I chose to implement a chatroom because it relates well with the theme of this course, scaleability. The application should support many users simultaneously accessing the chatroom to send, receive, and display text. Moreover, chatting or messaging can be a good feature in many web applications. For instance, any informational or advertisement website can benefit from a "chat with our agent" feature to make it more dynamic and user-friendly. A delivery app could use one, too, to ensure a communication line between the delivery person and the customer. A task management app will significantly benefit from having a communication method between its users. Chatroom is the most basic communication tool that applications that want users or customers to communicate with each other (or with an agent) can add to their product. This versatility and wide range of applicability motivated me to implement a chatroom.

Although this "feature" stems from a restriction on this project, I think the idea of a central chatroom where anyone can join freely is interesting. It is similar to random chat apps but with everyone all at once. Someone may use it when they are bored and want to chat with a fellow random anonymous internet user. Because every user is already invited to the chatroom (assuming a scenario where this app is deployed and live on the web), I would expect the response time to be reasonably short. It is also easy to access and use since it is a web-based application. Users do not need to download an app on their device other than their web browser app.

This app follows the microservices and event-bus design on top of Socket.io for communication between browsers. There are three microservices: The chatroom service handles the communication between browsers and a Socket.io server. This microservice is where the Socket.io server is created. The moderator service checks the chat message content and replaces it with "The message has been moderated." if it contains profanity. Lastly, the vote service manages the upvote feature of the app. All data–user, chat history, and vote history–are stored in MongoDB. So when a new user joins, they can see all the past chat history. The communication between microservices is handled

by the event bus. The client directly fetches data from a microservice or uses soc
emit different events and data. All services generate logs in the logs/combined.log file and send them to the console. Meeting the event bus architecture requirement while using Socket.io was challenging and interesting. This app could have been easily converted to a monolith design, but I immensely enjoyed the learning experience.

As for why my application needs to scale, as mentioned above, it needs to scale to support an increasing number of users. Many clients will request data from the chat server simultaneously to display data on the browser, and without scaleability, the server may fail.

Question 2 **5 / 5 pts**

(Front-End/Back-End) HTTP Communication

Provide a screenshot of 3 places in your project code that demonstrate the use of HTTP communication. This should include:

- 1 HTTP request in the front-end.
- 1 HTTP request in the back-end.
- 1 HTTP response in the back-end.

For each screenshot provide a short description of what the screenshot is of.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

1. 1 HTTP request in the front-end. (client/src/routes/ChatRoom.svelte)

```
onMount(async () => {
    // get chat history and user list from backend by contacting chatroom service
    const res1 = await fetch('http://localhost:5000/init');
    const { chats, users } = await res1.json();

    console.log("chatroom chat, users init", chats, users)

    chatStore.set(chats)
    activeUsersStore.set(users);

    // get votes data from votes service
    const res2 = await fetch('http://localhost:5002/init');
    const votes = await res2.json();

    console.log('Chatroom votes init: ', votes)

    voteStore.set(votes);
```

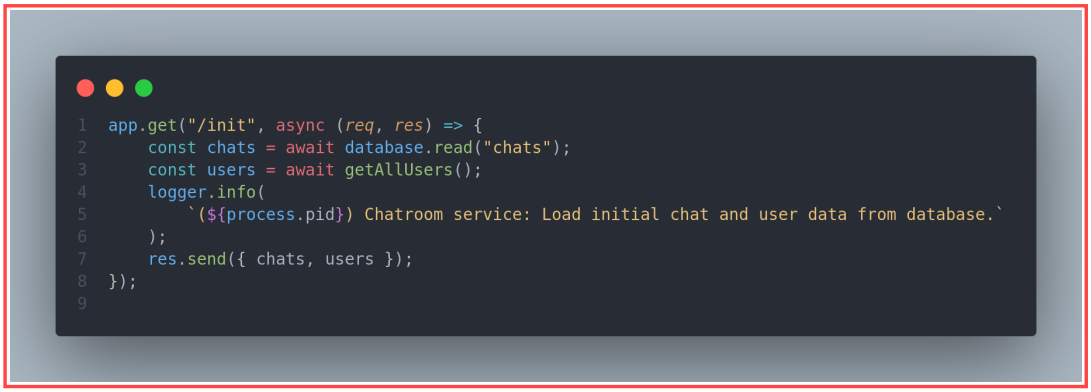
This code snippet is from the ChatRoom component of the Svelte front-end. When the chatroom is loaded (onMount), two GET requests are sent to the backend (localhost:5000/init is chatroom service and localhost:5002/init is vote service) to retrieve the chat, user, and vote data stored in the database. The chatroom and vote services access the MongoDB database and send the stored information to the client, which is then rendered into chat bubbles, active user lists, and vote counts.

2. 1 HTTP request in the back-end. (moderator/index.js)

```
// send event to event bus
try {
    await fetch("http://event-bus:4000/events", {
        // await fetch("http://localhost:4000/events", {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({
            type: "ChatModerated",
            data: moderated,
        }),
    });
    logger.info(
        `(${process.pid}) Moderator Service: Emitted ChatModerated event.`
    );
} catch (error) {
    logger.info(`(${process.pid}) Moderator Service: ${error}`);
    res.status(500).send({
        status: "ERROR",
        message: error,
    });
}
```

This code is from the moderator service, where it responds to a "ChatCreated" event. After moderating the message, the service makes a POST request to the event bus with an object of type "ChatModerated" and moderated chat content. Once the event bus receives this request, it relays the new event object to other connected microservices.

3. 1 HTTP response in the back-end. (chatroom/index.js)



```
app.get("/init", async (req, res) => {
    const chats = await database.read("chats");
    const users = await getAllUsers();
    logger.info(
        `(${process.pid}) Chatroom service: Load initial chat and user data from database.`
    );
    res.send({ chats, users });
});
```

When the client makes an initial request to load data upon mounting (as in the first screenshot), the chatroom service reads chats and user data from the database and responds with an object with two fields: chats and users. The client uses this data to render chatroom components.

Question 3 **5 / 5 pts**

(Front-End) Svelte Framework

Provide a screenshot of a part of your front-end code that demonstrates your deep understanding of the Svelte framework.

Provide a short description of the code and why you think it demonstrates deep understanding.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

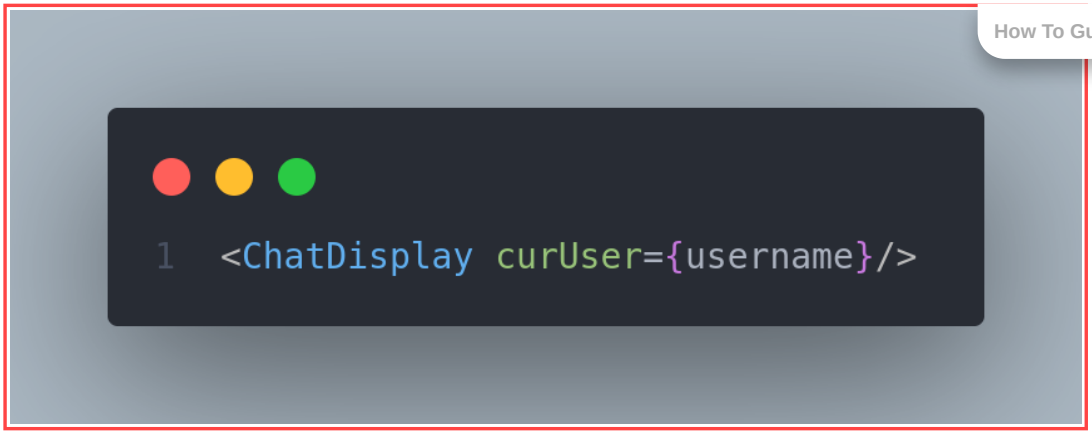
Your Answer:

[client/src/routes/ChatDisplay.svelte]

```svelte
<script>
    import BubbleReceived from "./BubbleReceived.svelte";
    import BubbleSent from "./BubbleSent.svelte";
    import AdminMessage from "./AdminMessage.svelte";

    import { onMount, afterUpdate } from "svelte";
    import { chatStore, socket } from "./stores.js";

    export let curUser;
    let chatContainer;

    onMount(() => {
        chatContainer = document.querySelector('.chat-container');

        // received message
        socket.on("message", (data) => {
            chatStore.update(chats => [...chats, data])
        })
    })

    afterUpdate(() => {
        if (chatContainer) {
            chatContainer.scrollTop = chatContainer.scrollHeight;
        }
    });

</script>

<div class="chat-container m-2">
    <ul class="chat-display px-1">
        {#if $chatStore.length > 0}
            {#each $chatStore as chat (chat.id)}
                {#if chat.username === 'admin'}
                    <AdminMessage message={chat} />
                {:else if chat.userid === socket.id}
                    <BubbleSent sent={chat} curUser={curUser}/>
                {:else if chat.username !== 'admin'}
                    <BubbleReceived received={chat}  curUser={curUser}/>
                {/if}
            {/each}
        {/if}
    </ul>
</div>

```

This is the complete code of ChatDisplay component. It utilizes various functionalities of the Svelte framework well, including props, stores, onMount, afterUpdate, if-else, and each block.

Below is how this component is created in its parent component, ChatRoom.svelte:



```
<ChatDisplay curUser={username}/>
```

Because username is created by the user at the parent level, it is delivered to ChatRoom via a prop called curUser.

chatStore is a writable store, and socket is a socket.io-client object. Both are defined in store.js as below:

```
import { writable } from 'svelte/store';
import io from 'socket.io-client';

// Initial data retrieved from MongoDB or set to empty arrays
export const chatStore = writable([]);
export const activeUsersStore = writable([]);
export const voteStore = writable({})

export const socket = io('ws://localhost:5000');
```

Once the component is mounted, it listens for new incoming messages using socket.on(). When the "message" event is received (sent by Socket.io server in the chatroom service), it updates chatStore with update() method to include the new chat.

In the HTML portion of this component, it reads each chat object stored in chatStore using {each} block. Because chatStore is marked as a reactive statement within the block ($chatStore), this component is re-rendered whenever chatStore content is changed.

Each chat message can be one of the three types based on its ID and the current user's socket.id information. ChatDisplay component makes this distinction using {if-else if} block and creates different child components (AdminMessage, BubbleSent, or BubbleReceived). The chat information is passed on to the child component using props, like message, sent, curUser, and received.

Question 4

5 / 5 pts

(Front-End) Component Implementation

Provide a screenshot of 1 component in your front-end code that you believe was well constructed.

Provide a short description of what the screenshot is of and why you think this component is a good demonstration of a well constructed component.

🔥 Important 🔥

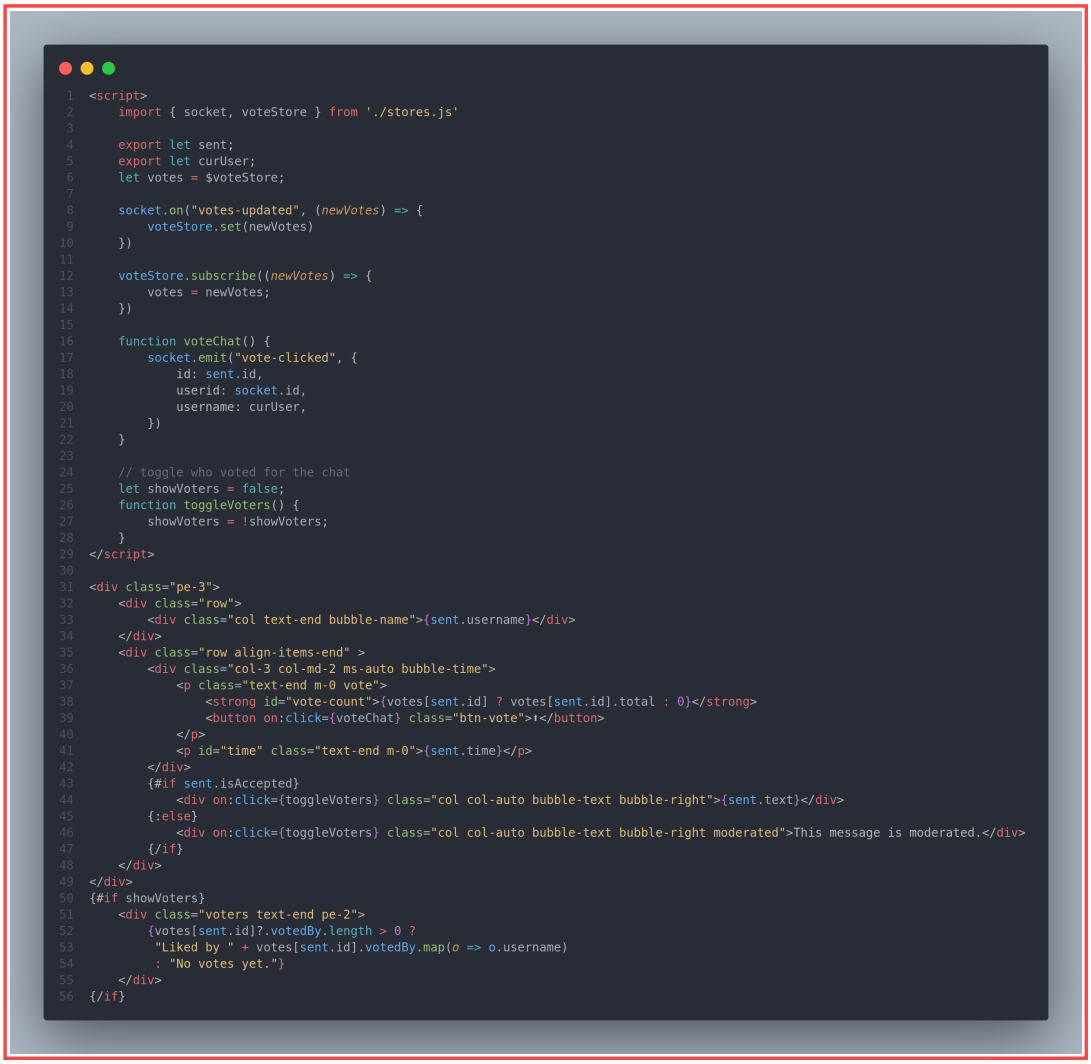
⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

[client/src/routes/BubbleSent.svelte]



```
<script>
    import { socket, voteStore } from './stores.js'

    export let sent;
    export let curUser;
    let votes = $voteStore;

    socket.on("votes-updated", (newVotes) => {
        voteStore.set(newVotes)
    })

    voteStore.subscribe((newVotes) => {
        votes = newVotes;
    })

    function voteChat() {
        socket.emit("vote-clicked", {
            id: sent.id,
            userid: socket.id,
            username: curUser,
        })
    }

    // toggle who voted for the chat
    let showVoters = false;
    function toggleVoters() {
        showVoters = !showVoters;
    }
</script>

<div class="pe-3">
    <div class="row">
        <div class="col text-end bubble-name">{sent.username}</div>
    </div>
    <div class="row align-items-end" >
        <div class="col-3 col-md-2 ms-auto bubble-time">
            <p class="text-end m-0 vote">
                <strong id="vote-count">{votes[sent.id] ? votes[sent.id].total : 0}</strong>
                <button on:click={voteChat} class="btn-vote">+</button>
            </p>
            <p id="time" class="text-end m-0">{sent.time}</p>
        </div>
        {#if sent.isAccepted}
            <div on:click={toggleVoters} class="col col-auto bubble-text bubble-right">{sent.text}</div>
        {:else}
            <div on:click={toggleVoters} class="col col-auto bubble-text bubble-right moderated">This message is moderated.</div>
        {/if}
    </div>
</div>
{#if showVoters}
    <div class="voters text-end pe-2">
        {votes[sent.id]?.votedBy.length > 0 ?
         "Liked by " + votes[sent.id].votedBy.map(o => o.username)
         : "No votes yet."}
    </div>
{/if}
```

This is the component for chat bubbles sent from a user. I think this is well-constr because it separates the concern of rendering the chat bubble from its parent component. It also actively utilizes BootStrap's grid system as well as custom CSS classes for styling. It's also a reactive component, which shows different/updated information based on several conditions. If the message is moderated (i.e. it contains profanity), the message is replaced by "This message is moderated." The vote count information is read from voteStore and is updated whenever a user upvotes the chat. The memo below that shows who upvoted the chat can be toggled by clicking on the bubble. The memo is also conditionally rendered. If no one liked the chat, it says "No votes yet," but if there's at least one upvote, it lists the usernames of all those who liked that chat.

Overall, I think it is a small but cute component that takes advantage of many Svelte features.

Question 5 **5 / 5 pts**

(Front-End) Props Usage

Provide a screenshot of props that are defined and being used to pass data between components.

Provide a short description of what the screenshot is of and why you think this shows proficiency in their usage.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** **(https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

[ChatRoom.svelte]

```
<ActiveUsers curUser={username}/>
<hr/>

<form on:submit={enterRoom}>
    <div class="row w-100">
        <div class="form-group col">
            <input bind:value={username} class="form-control col" type="text" maxlength="20" placeholder="Your name" size="11" required>
        </div>
        <button class="btn btn-primary col-1">Join</button>
    </div>
</form>

<ChatDisplay curUser={username}/>

<SendChat curUser={username}/>
<div class="activity">{activity}</div>
```

[ChatDisplay.svelte]

```
<div class="chat-container m-2">
    <ul class="chat-display px-1">
        {#if $chatStore.length > 0}
            {#each $chatStore as chat (chat.id)}
                {#if chat.username === 'admin'}
                    <AdminMessage message={chat} />
                {:else if chat.userid === socket.id}
                    <BubbleSent sent={chat} curUser={curUser}/>
                {:else if chat.username !== 'admin'}
                    <BubbleReceived received={chat}  curUser={curUser}/>
                {/if}
            {/each}
        {/if}
    </ul>
</div>
```

Among the seven components I implemented for the project, ChatRoom and ChatDisplay components use props to send data to their child components.

Props are used to deliver information defined or retrieved at the parent component's level but (also) used at the child component level.

For example, username is entered by the user in the ChatRoom component but used by ActiveUsers, ChatDisplay, and SendChat components. So this is communicated via "curUser" prop.

In ChatDisplay component, a full chat history list is updated and retrieved from chatStore. Because each chat bubble only needs one chat bubble's information, the array of chat bubble objects is iterated with {each} block, and each object is passed on to AdminMessage, BubbleSent, or BubbleReceived component as props.

Question 6 **5 / 5 pts**

(Front-End) Store Implementation

Provide a screenshot of stores that are defined and being used to manage data b components.

Provide a short description of what the screenshot is of and why you think this shows expert usage.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap) to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

[stores.js]

```
import { writable } from 'svelte/store';
import io from 'socket.io-client';

// Initial data retrieved from MongoDB or set to empty arrays
export const chatStore = writable([]);
export const activeUsersStore = writable([]);
export const voteStore = writable({})

export const socket = io('ws://localhost:5000');
```

chatStore stores a list of all chat bubble objects (contains chat ID, username, user ID, text content, whether it passed moderation, etc.). activeUsersStore is a list of user objects (username, user ID) currently accessing the chatroom. voteStore is an object whose key is the chat ID, and value is an upvote information object (a list of users who upvoted the chat, count of upvotes it received).

Chat history, users list, and vote information are stored in separate databases (collections) in MongoDB, so separating them into different stores is reasonable.

To reduce the number of times the application needs to access and read the database, when a new chat/user/upvote is made, the changes are (1) reflected on the frontend by updating the respective store and (2) the backend is notified to update the information stored in the database. A full reload from the backend only happens when the user initially joins or refreshes the page.

During a single continuous session (where the user does not leave the app by closing/refreshing the browser), utilizing stores allows us to support faster rendering

because information doesn't need to be loaded from the backend or database for single update.

Below are screenshots of components that use these stores:

[ActiveUsers] - activeUsersStore

```
<script>
    import { activeUsersStore } from './stores.js'
    export let curUser;
</script>

<div class="row row-cols-auto">
    {#if $activeUsersStore.length > 0}
        <div class="col"><strong>Active users: </strong></div>
        {#each $activeUsersStore as user}
            <div class="col col-left user me-2">
                {user}
            </div>
        {/each}
    {:else}
        <div class="col"><strong>No active users.</strong></div>
    {/if}
</div>

{#if $activeUsersStore.length > 0 && curUser}
    <strong class="welcome">Welcome {curUser}!</strong>
{/if}
```

[BubbleSent, BubbleReceived] - voteStore

```svelte
import { socket, voteStore } from './stores.js'

export let sent;
export let curUser;
let votes = $voteStore;

socket.on("votes-updated", (newVotes) => {
    voteStore.set(newVotes)
})

voteStore.subscribe((newVotes) => {
    votes = newVotes;
})

function voteChat() {
    socket.emit("vote-clicked", {
        id: sent.id,
        userid: socket.id,
        username: curUser,
    })
}
```

[ChatDisplay] - chatStore

```svelte
import { onMount, afterUpdate } from "svelte";
import { chatStore, socket } from "./stores.js";

export let curUser;
let chatContainer;

onMount(() => {
    chatContainer = document.querySelector('.chat-container');

    // received message
    socket.on("message", (data) => {
        chatStore.update(chats => [...chats, data])
    })
})
```

(Front-End) CSS and Styling

Provide a screenshot of where CSS or a CSS framework is being used in the front-end.

Provide a short description of what the screenshot is of and how it gives your application a polished look.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** **(https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

I used Bootstrap 5 and my custom classes to style the application. All CSS stylings are written in global.css, which is imported to the topmost component, +page.svelte.

[global.css]

```css
* {
    font-family: 'Inter', sans-serif;
    --main-color: rgb(53, 136, 239);
    --secondary-color: rgb(106, 167, 242);
    --gray-0: #e6e6e6;
    --gray-1: #bababa;
    --gray-2: #a0a0a0;
    --gray-3: #767676;
}

.welcome {
    color: var(--main-color);
}

/* ACTIVE USERS PILL */
.user {
    background-color: var(--gray-3);
    border-radius: 10rem;
    padding-left: 0.5rem;
    padding-right: 0.5rem;
    color: white;
}

/* "XXX is typing..." */
.activity {
    font-style: italic;
    color: var(--gray-1);
}

/* CHAT DISPLAY SECTION */
.chat-container {
    height: 500px;
    border: var(--gray-1) solid 2px;
    border-radius: 1rem;
    overflow-y: auto;
}

.chat-display {
    list-style-type: none;
    padding: 0;
    overflow-x: hidden;
}

.bubble-name {
    font-size: small;
```

```
    font-size: small;
    color: var(--gray-3);
}
.bubble-text {
    padding: 0.5rem;
    padding-left: 1rem;
    padding-right: 1rem;
    max-width: 60%;
    word-wrap: normal;
}

.bubble-left {
    border-radius: 1rem 1rem 1rem 0rem;
    background-color: var(--gray-0);
}

.bubble-right {
    border-radius: 1rem 1rem 0rem 1rem;
    background-color: var(--main-color);
    color: white;
}

.bubble-time {
    color: var(--gray-2);
    font-size: small;
}

.admin-message {
    color: var(--gray-3);
}

.moderated {
    color: var(--gray-2);
    font-style: italic;
}

/* VOTING FOR CHAT BUBBLE */
.vote {
    color: var(--main-color);
}

.btn-vote {
    color: var(--main-color);
    background-color: transparent;
    border-color: transparent;
    padding: 0;
}

.voters {
    font-size: small;
```

```
95      color: var(--secondary-color);
96  }
97
```

[+page.svelte]

```
1  <script>
2      import './global.css';
3      import ChatRoom from './ChatRoom.svelte';
4
5  </script>
6
7
8  <div class="p-4">
9      <h1 class="text-center"><strong>Chat426</strong></h1>
10     <p>Set your nickname and have fun :)</p>
11     <hr/>
12     <ChatRoom />
13 </div>
14
```

Although it is possible, using pure CSS to align DOM elements can be very confusing. BootStrap's pre-defined classes, especially the grid system (col/row) and spacing (p-2, ms-3, etc.) made the development process much easier. My custom styling gave the application a more polished and consistent look by adjusting the details, such as size, color, and additional positioning and alignment rules.

Question 8 **5 / 5 pts**

(Front-End/Back-End) Data Worth Storing

Provide a screenshot of where you are using meaningful data in your application code.

Provide a short description of what the screenshot is of and why the data is meaningful.

🔥 Important 🔥

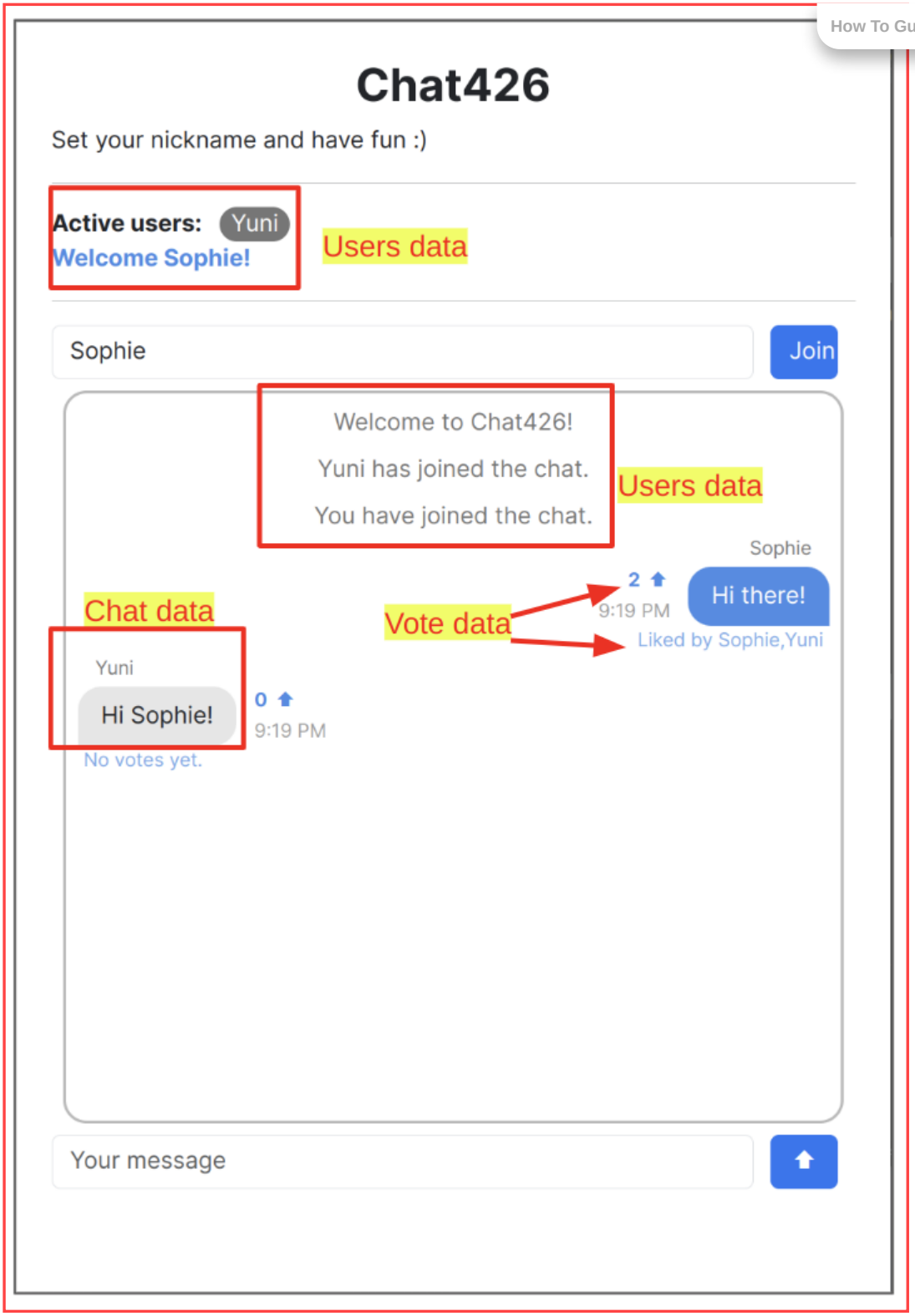
⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap) to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

As a simple chatting application, my application has three main types of "useful information": chat, user, and vote. Chat and user information are essential for any messaging application as they constitute the core details required for communication. Vote data is a nice addition to improve user experience. How each piece of information is used in my application is straightforward. Chat data allows users to communicate by displaying sent and received messages. User data shows users who are participating in the conversation now. Vote data provides an easy and fun way for additional communication to show agreement. Here's a screenshot of the browser and how each data appears on the screen:

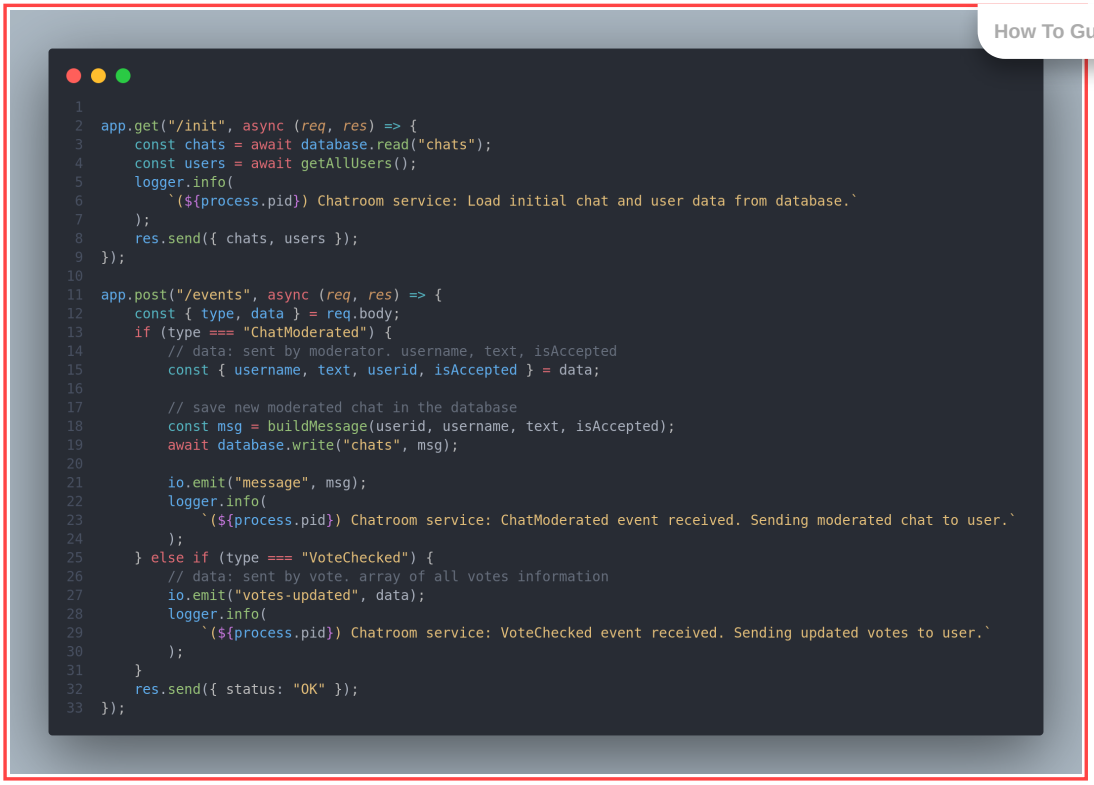


As described in previous questions, each type of information is stored in a separate collection in MongoDB. The {chatroom} microservice deals with chat and user data, and the {vote} microservice deals with vote data. Below is the code in each microservice that shows how they connect to the database:

[chatroom/database.js] - MongoDB client for Chat and User data

```
import { MongoClient } from "mongodb";
import logger from "./logger.js";

const url = "mongodb://admin:secret@db:27017";
const client = new MongoClient(url);
const chatsDB = "chats";
const usersDB = "users";

let users;
let chats;

const userCollection = async () => {
    if (!users) {
        await client.connect();
        const db = client.db(usersDB);
        users = db.collection(usersDB);
    }
    return users;
};

const chatCollection = async () => {
    if (!chats) {
        await client.connect();
        const db = client.db(chatsDB);
        chats = db.collection(chatsDB);
    }
    return chats;
};

// dataType: 'users' | 'chats'
const read = async (dataType) => {
    try {
        const collection =
            dataType === "users"
                ? await userCollection()
                : await chatCollection();
        const docs = await collection.find({}).toArray(); // {}: find all documents in the collection
        logger.info(
            `(${process.pid}) Chatroom service: Reading ${dataType} collection.`
        );
        return dataType === "users" ? docs[0]?.usersArray || [] : docs || []; // default if docs is undefined: []
    } catch (err) {
        logger.error(err);
    }
};

// dataType: 'users' | 'chats'
// 'users' case: data is an object with schema of {"usersArray": [Array of user objects]}
// 'chats' case: data is one chat object created with buildMessage(...)
const write = async (dataType, data) => {
    try {
        const collection =
            dataType === "users"
                ? await userCollection()
                : await chatCollection();
        if (dataType === "users") {
            await collection.deleteMany({});
        }
        await collection.insertOne(data);
        logger.info(
            `(${process.pid}) Chatroom service: Writing to ${dataType} collection.`
        );
    } catch (err) {
        logger.error(err);
    }
};

// TESTING;
// const demoChats = [];
// let demoUsers = [{ usersArray: [] }];
// const read = (dataType) => {
//  return dataType === "users" ? demoUsers[0].usersArray : demoChats;
// };
// const write = (dataType, data) => {
//  dataType === "users" ? (demoUsers = [data]) : demoChats.push(data);
// };

export default { read, write };

```

[chatroom/index.js] - initial data load (/init) and adding new chat message to database (/event). Chat, User Data



```
app.get("/init", async (req, res) => {
    const chats = await database.read("chats");
    const users = await getAllUsers();
    logger.info(
        `(${process.pid}) Chatroom service: Load initial chat and user data from database.`
    );
    res.send({ chats, users });
});

app.post("/events", async (req, res) => {
    const { type, data } = req.body;
    if (type === "ChatModerated") {
        // data: sent by moderator. username, text, isAccepted
        const { username, text, userid, isAccepted } = data;

        // save new moderated chat in the database
        const msg = buildMessage(userid, username, text, isAccepted);
        await database.write("chats", msg);

        io.emit("message", msg);
        logger.info(
            `(${process.pid}) Chatroom service: ChatModerated event received. Sending moderated chat to user.`
        );
    } else if (type === "VoteChecked") {
        // data: sent by vote. array of all votes information
        io.emit("votes-updated", data);
        logger.info(
            `(${process.pid}) Chatroom service: VoteChecked event received. Sending updated votes to user.`
        );
    }
    res.send({ status: "OK" });
});
```

[chatroom/index.js] - User functions. User Data

```
// user functions
const UsersState = {
    users: async () => await database.read("users"),
    setUsers: async function (newUsersArray) {
        // update database
        await database.write("users", { usersArray: newUsersArray });
    },
};

async function activateUser(userid, username) {
    const user = { userid, username };

    // filter: making sure there's no duplicate
    const activeUsers = await UsersState.users();
    UsersState.setUsers([
        ...activeUsers.filter((user) => user.userid !== userid),
        user,
    ]);
    return user;
}

async function userLeavesApp(userid) {
    const activeUsers = await UsersState.users();
    UsersState.setUsers(activeUsers.filter((user) => user.userid !== userid));
}

async function getUser(userid) {
    const activeUsers = await UsersState.users();
    return activeUsers.find((user) => user.userid === userid);
}

async function getAllUsers() {
    const activeUsers = await UsersState.users();
    return activeUsers.map((user) => user.username);
}

```

[vote/database.js] - MongoDB client for Vote Data

```
import { MongoClient } from "mongodb";
import logger from "./logger.js";

const url = "mongodb://admin:secret@db:27017";
const client = new MongoClient(url);
const votesDB = "votes";

let votes;
const voteCollection = async () => {
    if (!votes) {
        await client.connect();
        const db = client.db(votesDB);
        votes = db.collection(votesDB);
    }
    return votes;
};

const read = async () => {
    try {
        const collection = await voteCollection();
        const docs = await collection.find({}).toArray(); // {}: find all documents in the collection
        logger.info(`(${process.pid}) Vote Service: Reading votes collection.`);
        return docs[0] || {}; // default if docs is undefined: {}
    } catch (err) {
        logger.error(err);
    }
};

const write = async (data) => {
    try {
        const collection = await voteCollection();
        await collection.deleteMany({}); // delete everything
        await collection.insertOne(data);
        logger.info(
            `(${process.pid}) Vote Service: Writing to votes collection.`
        );
    } catch (err) {
        logger.error(err);
    }
};

// TESTING
// let demoVotes = {};

// const read = () => {
//  return demoVotes;
// };
// const write = (data) => {
//  demoVotes = data;
// };

export default { read, write };

```

[vote/index.js] - initial data load (/init) and adding new vote information to database (/event; also handles whether the click should upvote or unvote the chat depending on whether the user has already liked the chat).

```
app.get("/init", async (req, res) => {
    const votes = await database.read();
    logger.info(`(${process.pid}) Load initial votes: ${votes}`);
    res.send(votes);
});

app.post("/events", async (req, res) => {
    // data: id, userid, username
    const { type, data } = req.body;
    const { id, userid, username } = data;

    // listen for new comments
    if (type !== "VoteClicked") {
        res.send({ message: "Nothing to check." });
        return;
    }

    const votes = await database.read();

    // chat was never liked before (e.g. does not exist in the database yet)
    if (votes[id] === undefined) {
        votes[id] = {
            id,
            total: 1,
            votedBy: [{ username, userid }],
        };
    }
    // chat was voted before. check if this user already liked the chat
    else {
        const foundIndex = votes[id].votedBy.findIndex(
            (userObj) => userObj.userid === userid
        );

        // hasn't vote for this chat before. Increase vote and add user info to votedBy array
        if (foundIndex === -1) {
            votes[id].total += 1;
            votes[id].votedBy.push({ username, userid });
        }
        // already voted for this chat. Decrease vote and remove user info from votedBy array
        else {
            votes[id].total -= 1;
            votes[id].votedBy.splice(foundIndex, 1);
        }
    }

    // update DB
    await database.write(votes);

    // send event to event bus
    try {
        await fetch("http://event-bus:4000/events", {
            // await fetch("http://localhost:4000/events", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({
                type: "VoteChecked",
                data: votes,
            }),
        });
        logger.info(
            `(${process.pid}) Vote Service: Emitted VoteChecked event.`
        );
    } catch (error) {
        logger.error(`(${process.pid}) Vote Service: ${error}`);
        res.status(500).send({
            status: "ERROR",
            message: error,
        });
    }
    res.send(votes);
});
```

Below is how the client contacts the '/init' endpoints to retrieve data (also shown in Question 2 above:

[client/src/routes/ChatRoom.svelte]

```
// get chat history and user list from backend by contacting chatroom service
const res1 = await fetch('http://localhost:5000/init');
const { chats, users } = await res1.json();

console.log("chatroom chat, users init", chats, users)

chatStore.set(chats)
activeUsersStore.set(users);

// get votes data from votes service
const res2 = await fetch('http://localhost:5002/init');
const votes = await res2.json();

console.log('Chatroom votes init: ', votes)

voteStore.set(votes);

```

Once loaded, the data is processed by microservices and rendered by different components of the application. All components, other than the welcome message ("Chat426; Set your nickname and have fun :)") depend on these data. Hence, to show where I am using meaningful data in my application, I'd need to provide screenshots of the entire codebase. Please refer to my source code here if needed: **final-project.zip (https://umamherst.instructure.com/users/34292/files/3288233?wrap=1&verifier=rW6KuRq28ljBsl8DAn1MqjE0IBCIlkmTiT3vlifX)** **(https://umamherst.instructure.com/users/34292/files/3288233/download?verifier=rW6KuRq28ljBsl8DAn1MqjE0IBCIlkmTiT3vlifX&download_frd=1)**

Question 9 **5 / 5 pts**

(Back-End) Micro-Services Quantity and Complexity

Provide a screenshot of your folder structure of your project that explicitly shows your services as folders. (there should be 3-4 not including the front-end service).

Provide a short description of why you believe each of your services are complex enough to submit for this project.

🔥 Important 🔥

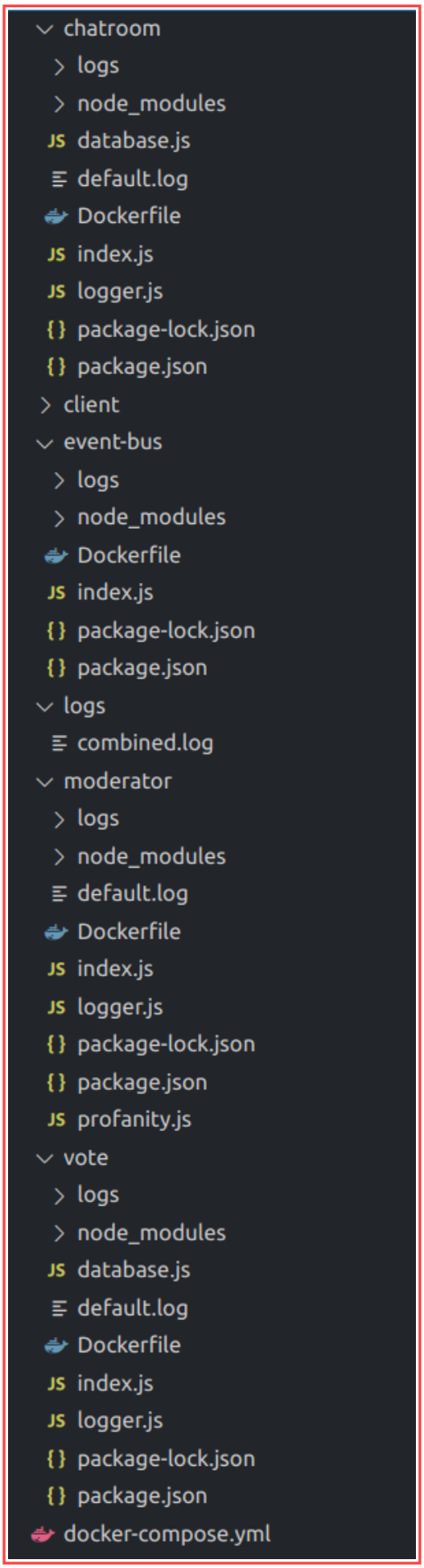
⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** **(https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

Here is a list of folders created for the project: chatroom, client, event-bus, logs, moderator, and vote. According to the project description, Chatroom, moderator, and vote are the microservices.


chatroom
logs
node_modules
database.js
default.log
Dockerfile
index.js
logger.js
package-lock.json
package.json
client
event-bus
logs
node_modules
Dockerfile
index.js
package-lock.json
package.json
logs
combined.log
moderator
logs
node_modules
default.log
Dockerfile
index.js
logger.js
package-lock.json
package.json
profanity.js
vote
logs
node_modules
database.js
default.log
Dockerfile
index.js
logger.js
package-lock.json
package.json
docker-compose.yml

[chatroom]

Chatroom is the most essential and complex microservice among the three. It creates the Socket.io server with an express server and handles various events emitted to the socket server ("message", "user-list", "vote-updated", "enter-room", "connection", "disconnect"), as well as direct fetch calls from the client and the event bus ('/init', '/event' endpoints). It also creates chat objects with the buildMessage function and updates the active user list with user functions. Lastly, the MongoDB client for Chat and User database resides in this service. In fact, because this microservice is quite large, I think the chat and user functions could have been decoupled into two microservices (potential future improvement).

Its responsibility is crucial to run the app and is complex enough to be in this project. Below is its index.js file:

```
import express from "express";
import cors from "cors";
import morgan from "morgan";
import { Server } from "socket.io";
import { randomBytes } from "crypto";
import database from "./database.js";
import logger from "./logger.js";

const PORT = process.env.PORT || 5000;
const ADMIN = "admin";

const app = express();
app.use(cors());
app.use(express.json());
app.use(morgan("dev"));

const expressServer = app.listen(PORT, () => {
    logger.info(`(${process.pid}) Chatroom Server: Listening on port ${PORT}`);
});

const io = new Server(expressServer, {
    // Needs cors because our frontend is hosted on different server
    cors: {
        origin: "*",
    },
});

app.get("/init", async (req, res) => {
    const chats = await database.read("chats");
    const users = await getAllUsers();
    logger.info(
        `(${process.pid}) Chatroom service: Load initial chat and user data from database.`
    );
    res.send({ chats, users });
});

app.post("/events", async (req, res) => {
    const { type, data } = req.body;
    if (type === "ChatModerated") {
        // data: sent by moderator. username, text, isAccepted
        const { username, text, userid, isAccepted } = data;

        // save new moderated chat in the database
        const msg = buildMessage(userid, username, text, isAccepted);
        await database.write("chats", msg);

        io.emit("message", msg);
        logger.info(
            `(${process.pid}) Chatroom service: ChatModerated event received. Sending moderated chat to user.`
        );
    } else if (type === "VoteChecked") {
        // data: sent by vote. array of all votes information
        io.emit("votes-updated", data);
        logger.info(
            `(${process.pid}) Chatroom service: VoteChecked event received. Sending updated votes to user.`
        );
    }
    res.send({ status: "OK" });
});

// socket.io: listen for events submitted from frontend
io.on("connection", (socket) => {
    const ID = socket.id; // userid

    // upon connection, emit only to user
    socket.emit("message", buildMessage(ID, ADMIN, "Welcome to Chat426!"));

    // upon connection, emit to all others (except for self)
    socket.on("enter-room", async (username) => {
        logger.info(
            `(${process.pid}) Chatroom service: User ${ID} entered the chat.`
        );
        // check if user already exists
        const activeUsers = await getAllUsers();
        if (activeUsers.includes(username)) {
            socket.emit("invalid-username");
            return;
        }

        const user = await activateUser(socket.id, username);

        // message to user who joined
        socket.emit(
            "message",
            buildMessage(ID, ADMIN, `You have joined the chat.`)
        );

        // message to everyone else
        socket.broadcast.emit(
            "message",
            buildMessage(ID, ADMIN, `${user.username} has joined the chat.`)
        );

        // update user list for the room user now joined
        io.emit("user-list", {
            users: await getAllUsers(),
        });
    });

    // when user disconnects, emit to all others
```

```
    socket.on("disconnect", async () => {
        const user = await getUser(socket.id);

        if (!user) return;

        // remove user from UsersState
        await userLeavesApp(user.userid);

        // send message
        io.emit(
            "message",
            buildMessage(ID, ADMIN, `${user.username} has left the chat.`)
        );

        // send updated user list
        io.emit("user-list", {
            users: await getAllUsers(),
        });

        logger.info(`(${process.pid}) User ${socket.id} disconnected.`);
    });

    // listen for a message event
    socket.on("message", async ({ username, text, userid }) => {
        // contact moderator service via event bus
        try {
            await fetch("http://event-bus:4000/events", {
            // await fetch("http://localhost:4000/events", {
                // event but url
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({
                    type: "ChatCreated",
                    data: { username, text, userid },
                }),
            });
            logger.info(
                `(${process.pid}) Chatroom Service: Emitted ChatCreated event.`
            );
        } catch (err) {
            logger.error(`(${process.pid}) Chatroom Service: ${err}`);
        }
    });

    // Listen for typing activity
    socket.on("activity", (username) => {
        socket.broadcast.emit("activity", username);
    });

    // Listen for clicking like button
    socket.on("vote-clicked", async ({ id, userid, username }) => {
        // contact vote service via event bus
        try {
            await fetch("http://event-bus:4000/events", {
            // await fetch("http://localhost:4000/events", {
                // event but url
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({
                    type: "VoteClicked",
                    data: { id, userid, username },
                }),
            });
            logger.info(
                `(${process.pid}) Chatroom Service: Emitted VoteClicked event.`
            );
        } catch (err) {
            logger.error(`(${process.pid}) Chatroom Service: ${err}`);
        }
    });
});

/**
 * NOTE
 * - io.emit(): to everyone connected to the server
 * - socket.emit(): to the connected user
 */

// does not impact user state
function buildMessage(socketid, username, text, accepted = true) {
    return {
        id: randomBytes(4).toString("hex"),
        userid: socketid,
        username,
        text,
        time: new Intl.DateTimeFormat("default", {
            hour: "numeric",
            minute: "numeric",
        }).format(new Date()),
        vote: 0,
        isAccepted: accepted,
    };
}

// user functions
const UsersState = {
    users: async () => await database.read("users"),
    setUsers: async function (newUsersArray) {
        // update database
        await database.write("users", { usersArray: newUsersArray });
    },
};

async function activateUser(userid, username) {
    const user = { userid, username };
```

```
    // filter: making sure there's no duplicate
    const activeUsers = await UsersState.users();
    UsersState.setUsers([
        ...activeUsers.filter((user) => user.userid !== userid),
        user,
    ]);
    return user;
}

async function userLeavesApp(userid) {
    const activeUsers = await UsersState.users();
    UsersState.setUsers(activeUsers.filter((user) => user.userid !== userid));
}

async function getUser(userid) {
    const activeUsers = await UsersState.users();
    return activeUsers.find((user) => user.userid === userid);
}

async function getAllUsers() {
    const activeUsers = await UsersState.users();
    return activeUsers.map((user) => user.username);
}
```

[moderator]

Inspired by the moderator microservice I implemented for homework 3, this microservice checks whether a newly created chat contains profanity. The list of profanity is imported from copied from **https://github.com/coffee-and-fun/google-profanity-words/blob/main/data/en.txt (https://github.com/coffee-and-fun/google-profanity-words/blob/main/data/en.txt)** . Given that this app provides an open-for-all chatting room, a moderator service would be fitting. This service listens for "ChatCreated" event delivered by the event bus. After checking the chat content, it updates the "isAccepted" field of the chat object and emits "ChatModerated" event to the event bus. Given that it has HTTP calls that communicate with the event bus and it manipulates/updates the input data (chat object), I think it is complex enough for this project. Below is a screenshot of its index.js file.

```
import express from "express";
import morgan from "morgan";
import cors from "cors";
import { profanity } from "./profanity.js"; // Array<string>
import logger from "./logger.js";

const app = express();

app.use(morgan("dev"));
app.use(cors());
app.use(express.json());

app.post("/events", async (req, res) => {
    const { type, data } = req.body;

    // listen for new comments
    if (type !== "ChatCreated") {
        res.send({ message: "Nothing to check." });
        return;
    }

    const moderated = { ...data, isAccepted: true };

    // scan comment for profanity words
    const textWords = data.text.split(/\s+/);

    for (const word of textWords) {
        const lowerCaseWord = word.toLowerCase();
        if (profanity.includes(lowerCaseWord)) {
            moderated.isAccepted = false;
            logger.warn(
                `(${process.pid}) Moderator Service: Chat ${data.id} used banned word(s).`
            );
            break;
        }
    }

    // send event to event bus
    try {
        await fetch("http://event-bus:4000/events", {
            // await fetch("http://localhost:4000/events", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({
                type: "ChatModerated",
                data: moderated,
            }),
        });
        logger.info(
            `(${process.pid}) Moderator Service: Emitted ChatModerated event.`
        );
    } catch (error) {
        logger.info(`(${process.pid}) Moderator Service: ${error}`);
        res.status(500).send({
            status: "ERROR",
            message: error,
        });
    }

    res.send(moderated);
});

app.listen(5001, () => {
    logger.info(`(${process.pid}) Moderator Listening on 5001`);
});

```

[vote]

This microservice ensures that each user can only vote for a chat once by checking the voteBy property of each vote object (loaded from the database). This check is activated when it receives a "VoteClicked" event from the event bus. When the check is completed

and vote objects are updated to be sent back to the client, it emits a "VoteChecke event to the event bus, which is then received and processed by {chatroom} service. It also communicates with the client directly to deliver a full list of vote data upon users' connection. This service maintains the Vote database and connects to it as a MongoDB client. Given the list of {vote} services responsibilities, I think it is complex enough for this project. Below is its index.js file:

```
import express from "express";
import cors from "cors";
import morgan from "morgan";
import database from "./database.js";
import logger from "./logger.js";

// create express app
const app = express();

app.use(morgan("dev"));
app.use(cors());
app.use(express.json());

app.get("/init", async (req, res) => {
    const votes = await database.read();
    logger.info(`(${process.pid}) Load initial votes: ${votes}`);
    res.send(votes);
});

app.post("/events", async (req, res) => {
    // data: id, userid, username
    const { type, data } = req.body;
    const { id, userid, username } = data;

    // listen for new comments
    if (type !== "VoteClicked") {
        res.send({ message: "Nothing to check." });
        return;
    }

    const votes = await database.read();

    // chat was never liked before (e.g. does not exist in the database yet)
    if (votes[id] === undefined) {
        votes[id] = {
            id,
            total: 1,
            votedBy: [{ username, userid }],
        };
    }
    // chat was voted before. check if this user already liked the chat
    else {
        const foundIndex = votes[id].votedBy.findIndex(
            (userObj) => userObj.userid === userid
        );

        // hasn't vote for this chat before. Increase vote and add user info to votedBy array
        if (foundIndex === -1) {
            votes[id].total += 1;
            votes[id].votedBy.push({ username, userid });
        }
        // already voted for this chat. Decrease vote and remove user info from votedBy array
        else {
            votes[id].total -= 1;
            votes[id].votedBy.splice(foundIndex, 1);
        }
    }

    // update DB
    await database.write(votes);

    // send event to event bus
    try {
        await fetch("http://event-bus:4000/events", {
            // await fetch("http://localhost:4000/events", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({
                type: "VoteChecked",
                data: votes,
            }),
        });
        logger.info(
            `(${process.pid}) Vote Service: Emitted VoteChecked event.`
        );
    } catch (error) {
        logger.error(`(${process.pid}) Vote Service: ${error}`);
        res.status(500).send({
            status: "ERROR",
            message: error,
        });
    }
    res.send(votes);
});
```

```
app.listen(5002, () => {
    logger.info(`(${process.pid}) Vote service: Listening on port 5002`);
});
```

Question 10

5 / 5 pts

(Back-End) Database Integration

Provide a screenshot of your folder structure of your project that explicitly shows your database service folder. (there should be at least 1).

Provide a short description identifying your database service folder (this should contain at least a Dockerfile) and tells us which database you are using, why you chose that database, and what data does it store.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

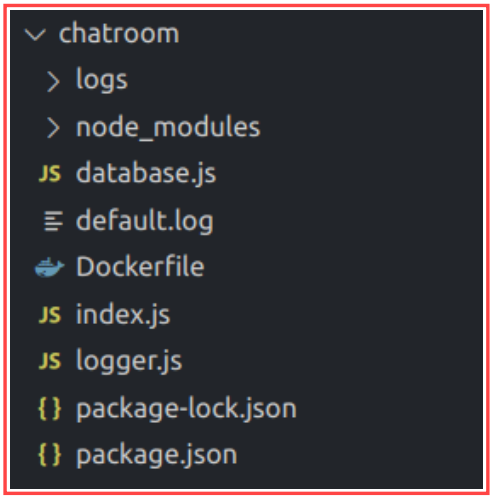
⚡ Use **CodeSnap** (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap) to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

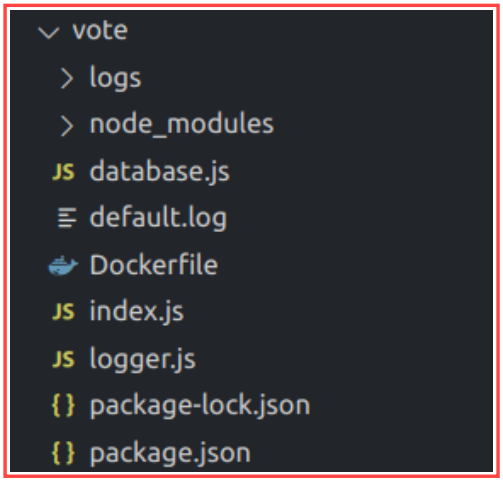
Your Answer:

Two microservices serve as MongoDB clients: chatroom and vote.

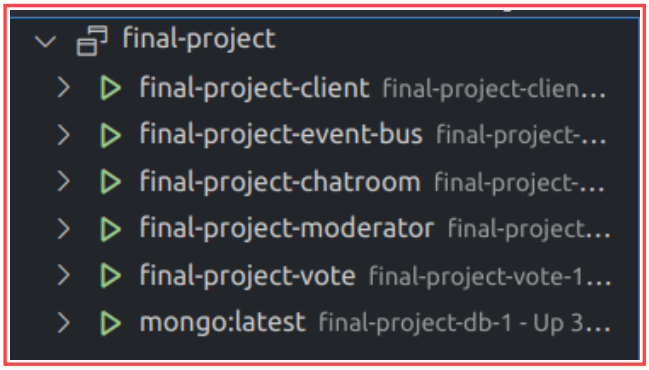
[Chatroom]



[vote]



[MongoDB in docker container]



I didn't make a separate service for the database. Each database.js file connects to the collection that is needed for each microservice.

I used MongoDB to avoid using an SQL database for a chat app. NoSQL does not require a tabular schema, so it saves space if the data is unstructured.

The Chat collection stores an array of chat objects, each of which looks like this:

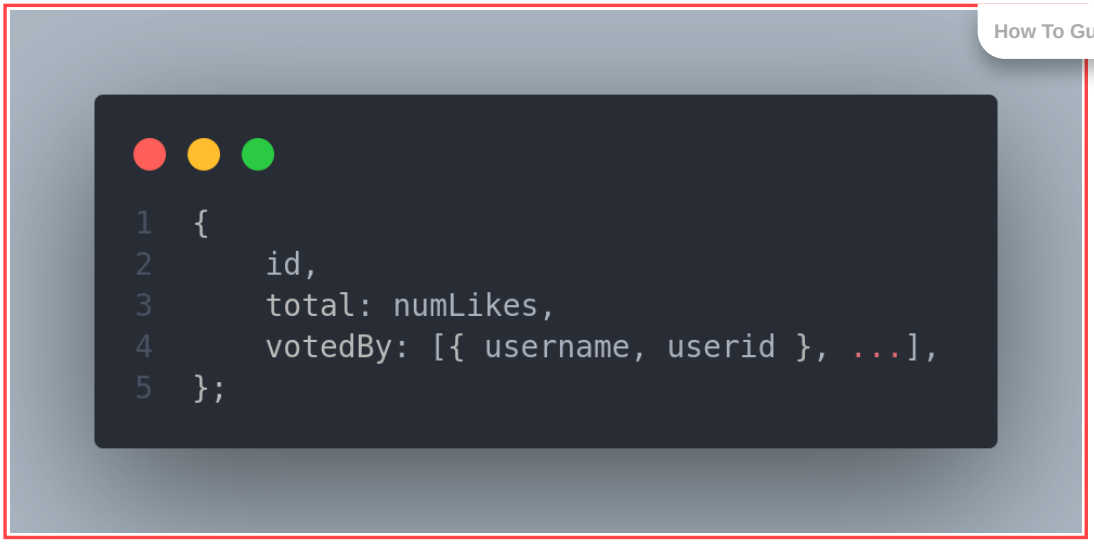
```
{
    id: randomBytes(4).toString("hex"),
    userid: socketid,
    username,
    text,
    time: new Intl.DateTimeFormat("default", {
        hour: "numeric",
        minute: "numeric",
    }).format(new Date()),
    vote: 0,
    isAccepted: accepted,
};
```

A new chat object is created and inserted into the collection when a user sends a new chat.

User collection is an array of user objects, each of which looks like this:

```
{ userid, username }
```

Vote collection is an object where the key is the chat ID and value is an object like this:



Question 11 5 / 5 pts

(Back-End) Use of PM2 and Clones

Provide a screenshot of the screen when you run `pm2 monit`.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

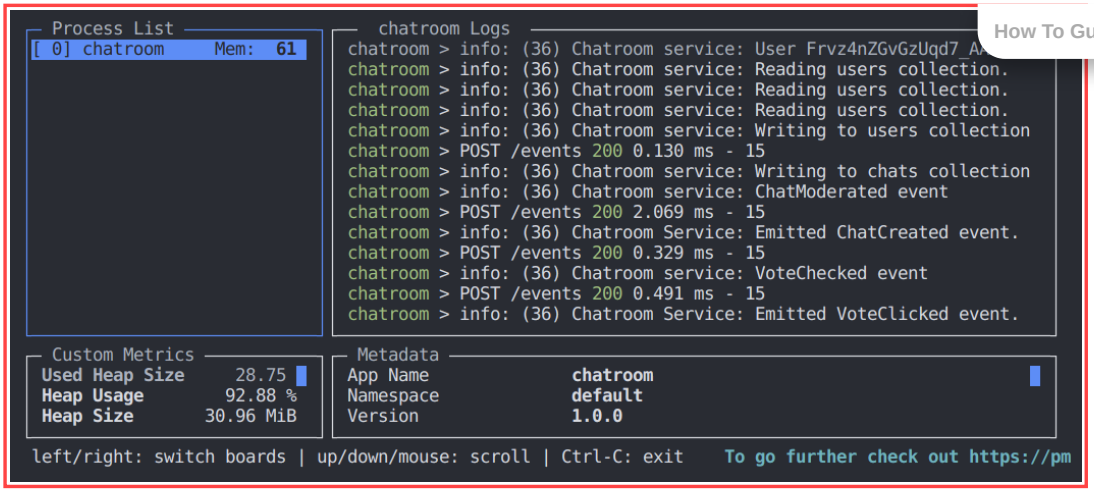
⚡ Use **CodeSnap** (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap) to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

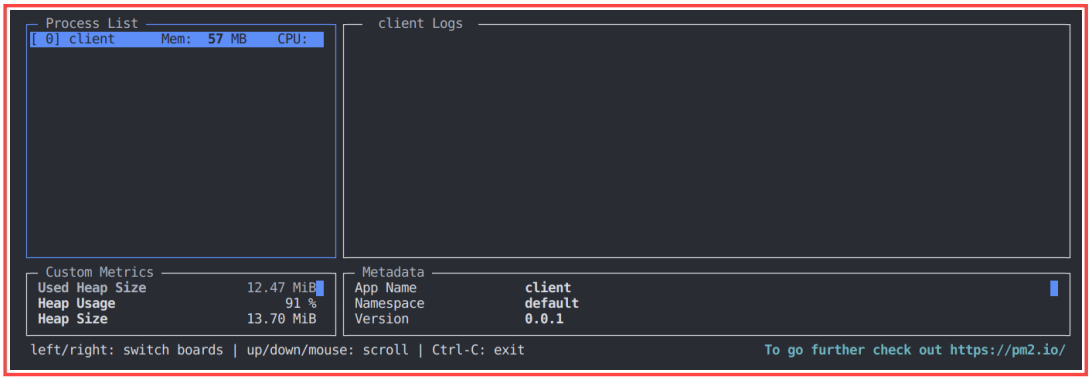
Your Answer:

Terminal command: `docker exec -it <container ID> sh` then `pm2 monit`

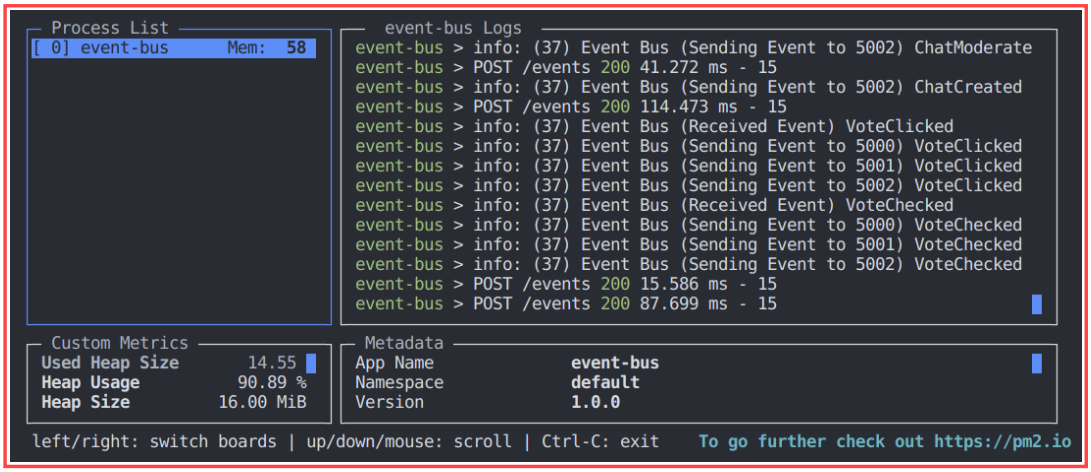
1. chatroom (1 instance)



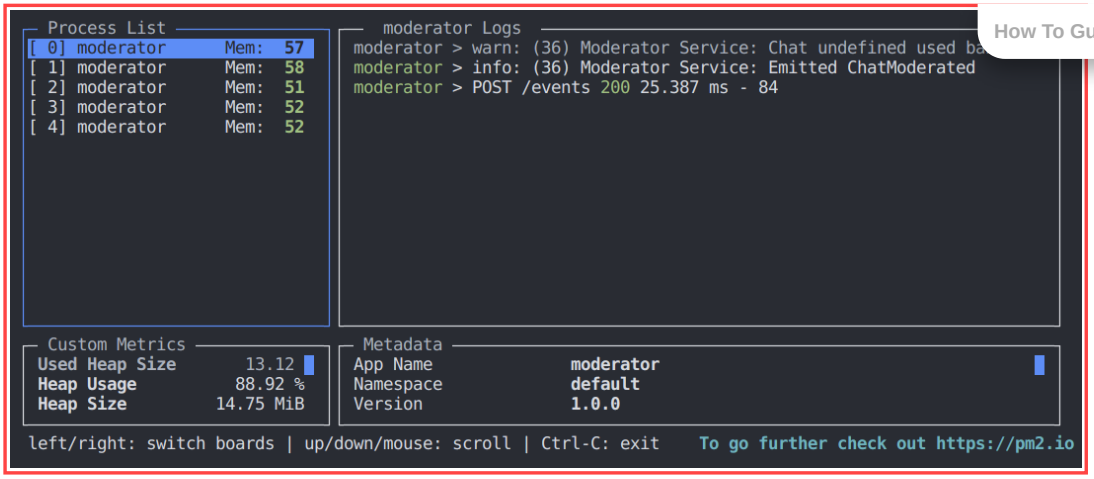
2. client (1 instance)



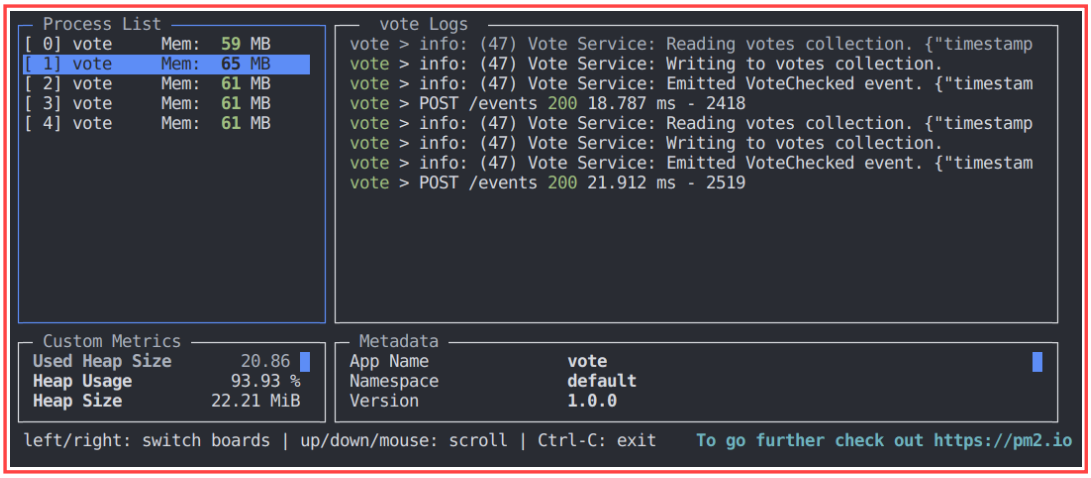
3. event-bus (1 instance)



4. moderator (5 instances)



5. vote (5 instances)



Question 12

5 / 5 pts

(Back-End) Package.json Scripts

Provide a screenshot of one of your `package.json` files with the required scripts.

🔥 Important 🔥

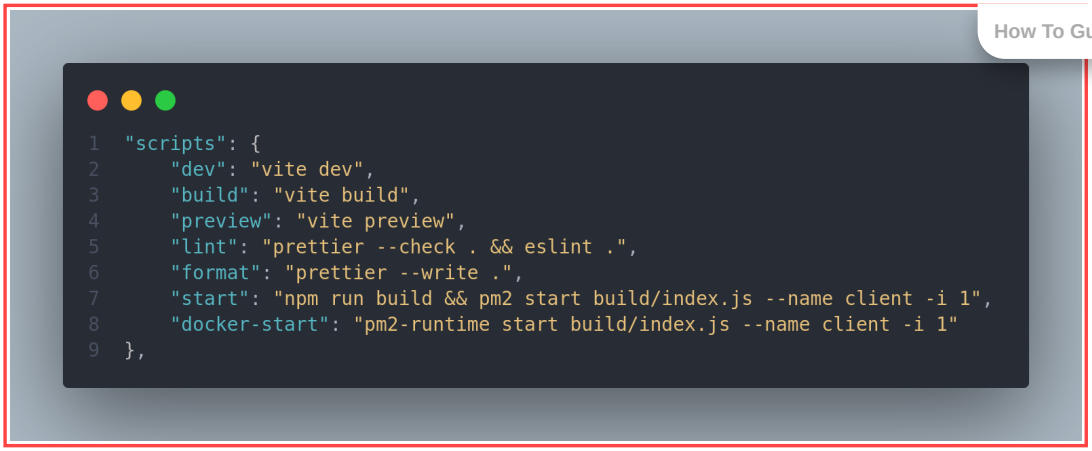
⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap) to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

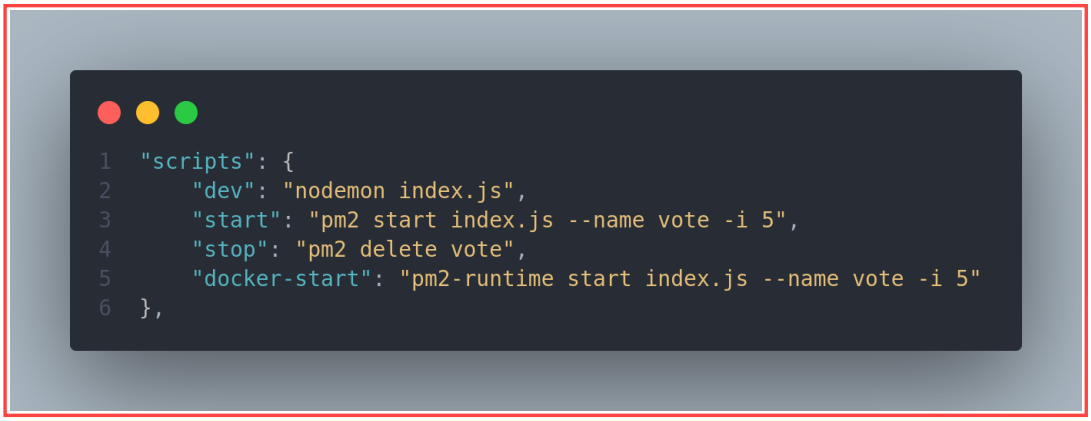
Your Answer:

1. client/package.json > scripts



```
"scripts": {
    "dev": "vite dev",
    "build": "vite build",
    "preview": "vite preview",
    "lint": "prettier --check . && eslint .",
    "format": "prettier --write .",
    "start": "npm run build && pm2 start build/index.js --name client -i 1",
    "docker-start": "pm2-runtime start build/index.js --name client -i 1"
},
```

2. vote/package.json > scripts. All other services' scripts are like this, but with different names.



```
"scripts": {
    "dev": "nodemon index.js",
    "start": "pm2 start index.js --name vote -i 5",
    "stop": "pm2 delete vote",
    "docker-start": "pm2-runtime start index.js --name vote -i 5"
},
```

Question 13 **5 / 5 pts**

(Back-End) Docker and Docker Compose Integration

Provide a screenshot of your `docker-compose.yml` file.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap** (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap) to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

```yaml
# verison of docker compose
version: "3.9"

services:
    client: # name of container
        build: client # path to Dockerfile
        ports:
            - "3000:3000" # maps container's port to a port on host machine
        volumes:
            - ./client:/usr/app # connect client folder to usr/app in the container
            # any modification made in client will be reflected in usr/app
            - /usr/app/build # keep the build directory built in the image
            - /usr/app/node_modules # also keep the node modules inside the built image
            - ./logs:/usr/app/logs # connect log volume
        environment:
            LOG_FILE: /usr/app/logs/combined.log

    event-bus:
        build: event-bus
        # we're not exposing port for event bus
        # only need to expose the ones used by client (browser)
        volumes:
            - ./event-bus:/usr/app
            - /usr/app/node_modules
            - ./logs:/usr/app/logs # connect log volume
        environment:
            LOG_FILE: /usr/app/logs/combined.log
    chatroom:
        build: chatroom
        ports:
            - "5000:5000"
        volumes:
            - ./chatroom:/usr/app
            - /usr/app/node_modules
            - ./logs:/usr/app/logs # connect log volume
        environment:
            LOG_FILE: /usr/app/logs/combined.log
    moderator:
        build: moderator
        ports:
            - "5001:5001"
        volumes:
            - ./moderator:/usr/app
            - /usr/app/node_modules
            - ./logs:/usr/app/logs # connect log volume
        environment:
            LOG_FILE: /usr/app/logs/combined.log
    vote:
        build: vote
        ports:
            - "5002:5002"
        volumes:
            - ./vote:/usr/app
            - /usr/app/node_modules
            - ./logs:/usr/app/logs # connect log volume
        environment:
            LOG_FILE: /usr/app/logs/combined.log

    db:
        image: mongo:latest
        volumes:
            - db-data:/data/db
        ports:
            - 27017:27017 # default MongoDB port
        environment:
            MONGO_INITDB_ROOT_USERNAME: admin
            MONGO_INITDB_ROOT_PASSWORD: secret

volumes:
    db-data:
    logs:
# run with docker compose up --build

```

(Back-End) Logging and Error Checking

Provide a screenshot of a part of the logs generated by your application.

Provide a screenshot of a part of your application where you believe you did an excellent job performing error checking and handling.

There is no written part required for this question.

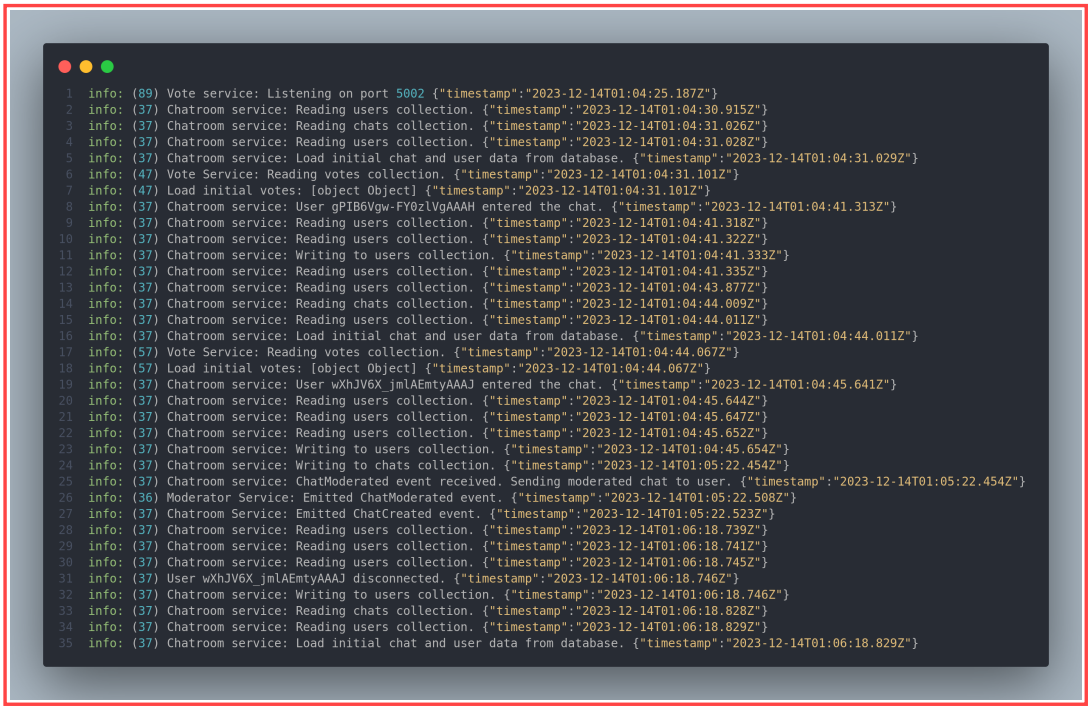
🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:



```
info: (89) Vote service: Listening on port 5002 {"timestamp":"2023-12-14T01:04:25.187Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:30.915Z"}
info: (37) Chatroom service: Reading chats collection. {"timestamp":"2023-12-14T01:04:31.026Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:31.028Z"}
info: (37) Chatroom service: Load initial chat and user data from database. {"timestamp":"2023-12-14T01:04:31.029Z"}
info: (47) Vote Service: Reading votes collection. {"timestamp":"2023-12-14T01:04:31.101Z"}
info: (47) Load initial votes: [object Object] {"timestamp":"2023-12-14T01:04:31.101Z"}
info: (37) Chatroom service: User gPIB6Vgw-FY0zlVgAAAH entered the chat. {"timestamp":"2023-12-14T01:04:41.313Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:41.318Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:41.322Z"}
info: (37) Chatroom service: Writing to users collection. {"timestamp":"2023-12-14T01:04:41.333Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:41.335Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:43.877Z"}
info: (37) Chatroom service: Reading chats collection. {"timestamp":"2023-12-14T01:04:44.009Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:44.011Z"}
info: (37) Chatroom service: Load initial chat and user data from database. {"timestamp":"2023-12-14T01:04:44.011Z"}
info: (57) Vote Service: Reading votes collection. {"timestamp":"2023-12-14T01:04:44.067Z"}
info: (57) Load initial votes: [object Object] {"timestamp":"2023-12-14T01:04:44.067Z"}
info: (37) Chatroom service: User wXhJV6X_jmlAEmtyAAAJ entered the chat. {"timestamp":"2023-12-14T01:04:45.641Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:45.644Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:45.647Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:04:45.652Z"}
info: (37) Chatroom service: Writing to users collection. {"timestamp":"2023-12-14T01:04:45.654Z"}
info: (37) Chatroom service: Writing to chats collection. {"timestamp":"2023-12-14T01:05:22.454Z"}
info: (37) Chatroom service: ChatModerated event received. Sending moderated chat to user. {"timestamp":"2023-12-14T01:05:22.454Z"}
info: (36) Moderator Service: Emitted ChatModerated event. {"timestamp":"2023-12-14T01:05:22.508Z"}
info: (37) Chatroom Service: Emitted ChatCreated event. {"timestamp":"2023-12-14T01:05:22.523Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:06:18.739Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:06:18.741Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:06:18.745Z"}
info: (37) User wXhJV6X_jmlAEmtyAAAJ disconnected. {"timestamp":"2023-12-14T01:06:18.746Z"}
info: (37) Chatroom service: Writing to users collection. {"timestamp":"2023-12-14T01:06:18.746Z"}
info: (37) Chatroom service: Reading chats collection. {"timestamp":"2023-12-14T01:06:18.828Z"}
info: (37) Chatroom service: Reading users collection. {"timestamp":"2023-12-14T01:06:18.829Z"}
info: (37) Chatroom service: Load initial chat and user data from database. {"timestamp":"2023-12-14T01:06:18.829Z"}
```

```
// Listen for clicking like button
socket.on("vote-clicked", async ({ id, userid, username }) => {
    // contact vote service via event bus
    try {
        await fetch("http://event-bus:4000/events", {
            // await fetch("http://localhost:4000/events", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({
                type: "VoteClicked",
                data: { id, userid, username },
            }),
        });
        logger.info(
            `(${process.pid}) Chatroom Service: Emitted VoteClicked event.`
        );
    } catch (err) {
        logger.error(`(${process.pid}) Chatroom Service: ${err}`);
    }
});
```

Question 15 5 / 5 pts

Code Organization and Code Quality

Provide a screenshot of a part of your application that demonstrates excellent code organization and quality.

Provide a short description of what the screenshot is of and why you think it demonstrates excellent code organization and quality.

🔥 Important 🔥

⚡ Make sure the images you paste in to the answer box are easily visible and clear. ⚡

⚡ Use **CodeSnap (https://marketplace.visualstudio.com/items?itemName=adpyke.codesnap)** to capture screenshots of your code. ⚡

⚡ Write clear and concise English sentences with proper grammar to receive full credit. ⚡

Your Answer:

```
import database from "./database.js";
```

```
// does not impact user state
function buildMessage(socketid, username, text, accepted = true) {
    return {
        id: randomBytes(4).toString("hex"),
        userid: socketid,
        username,
        text,
        time: new Intl.DateTimeFormat("default", {
            hour: "numeric",
            minute: "numeric",
        }).format(new Date()),
        vote: 0,
        isAccepted: accepted,
    };
}

// user functions
const UsersState = {
    users: async () => await database.read("users"),
    setUsers: async function (newUsersArray) {
        // update database
        await database.write("users", { usersArray: newUsersArray });
    },
};

async function activateUser(userid, username) {
    const user = { userid, username };

    // filter: making sure there's no duplicate
    const activeUsers = await UsersState.users();
    UsersState.setUsers([
        ...activeUsers.filter((user) => user.userid !== userid),
        user,
    ]);
    return user;
}

async function userLeavesApp(userid) {
    const activeUsers = await UsersState.users();
    UsersState.setUsers(activeUsers.filter((user) => user.userid !== userid));
}

async function getUser(userid) {
    const activeUsers = await UsersState.users();
    return activeUsers.find((user) => user.userid === userid);
}

async function getAllUsers() {
    const activeUsers = await UsersState.users();
    return activeUsers.map((user) => user.username);
}

```

The screenshots are from index.js of the chatroom service. I included this portion of the project because the user functions shown above reduced code duplication and provided a very convenient abstraction to work with databases.

buildMessage() is used throughout the file to make a new chat object. By hiding the user-independent fields (id, time, initial vote count) inside this function and receiving user-dependent properties as the argument (socket ID, username, chat text, and whether the moderator passed the text), this function helps to write cleaner and more reusable code.

The UserState object is the only piece of the code that directly contacts the User database. Using the output provided users() and setUsers() method, other user functions (activateUser, userLeavesApp, getUser, getallUsers) return the desired output or update the active user list in the database. It made debugging much easier, as I only had to edit fewer places when errors occurred. For example, I was getting the following error (copied from combined.log file): "error: BSON field 'insert.documents.0' is the wrong type 'array', expected type 'object' {"code":14,"codeName":"TypeMismatch","ok":0,"timestamp":"2023-12-13T02:20:21.036Z"}."

I soon realized I was calling collection.insertOne() with a new user array, not a JS object. I only had to change the setUser method inside UserState, from database.write('users', newUsersArray) to database.write('users', {usersArray: newUsersArray}), because the other user functions were not directly calling MongoDB-specific functions.

Question 16 **25 / 25 pts**

Project Demonstration

Submit a link to your project video demonstration. There are many ways to do a screen recording, please find an application that works best for your operating system. If you are not sure, please ask on Piazza for suggestions.

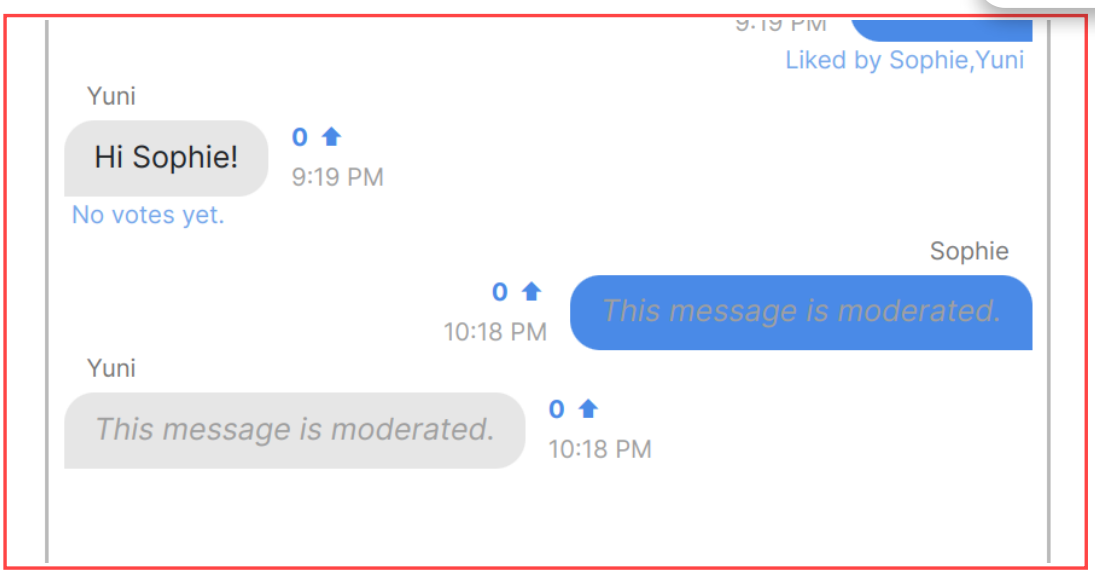
Your video demonstration must be between 3-5 minutes. We will not watch your video after the 5 minute mark. So, be quick and to the point in the video. We are not looking for you to show us code here. Just your application booting up using docker compose and the usage of the application.

Your Answer:

https://drive.google.com/file/d/1ctQ6r_gmwLX-GZZczxwsQquwekczIHx9/view?usp=sharing (https://drive.google.com/file/d/1ctQ6r_gmwLX-GZZczxwsQquwekczIHx9/view?usp=sharing)

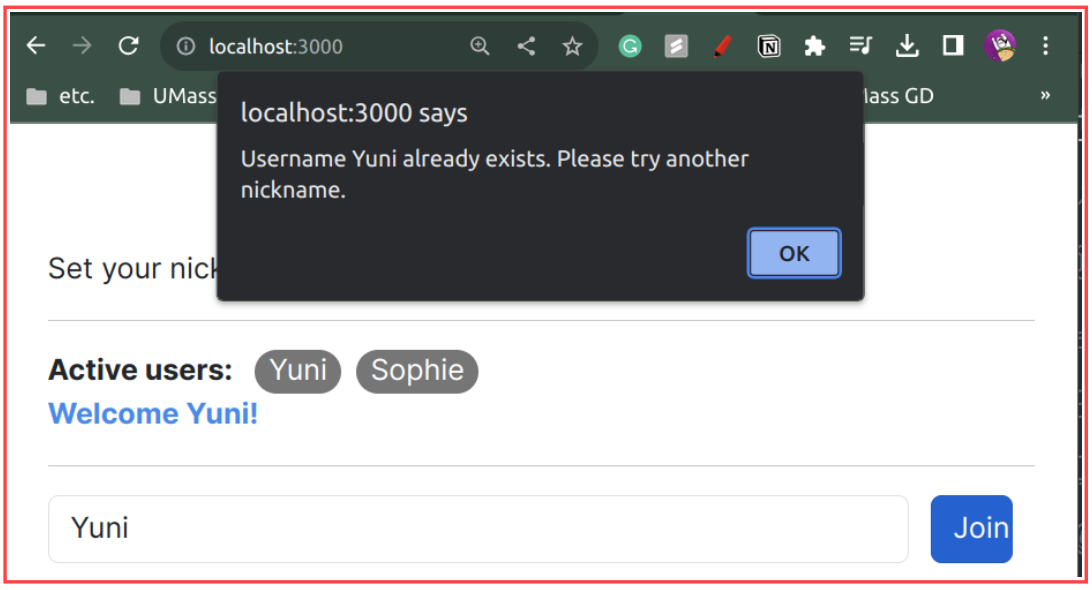
I just remembered that I should have included the moderator service demo and invalid username warning in the video. Below are the screenshots of these functionalities. I apologize for the additional writing:

[moderated chat bubble]



The messages above contained the f-word, but any word included in the profanity.js file will trigger the message to be moderated.

[Invalid username]



Because the name "Yuni" is already in the active user list, the browser sends an alert that the user should choose a different username. Usernames are not user IDs (user IDs are socket.id, which is why refreshing is equivalent to "logging off"), but I thought having identical names would be confusing.

Quiz Score: **100** out of 100