University Messaging System

Mentor: Yiqian Yang

Authors: Jordan Bettencourt , Charles Oxyer , William Suppiger , Peter Holmes , Robert Hung , Fei Ren

Copyright Statement

Table of Contents

5 Deployment

5.1 Backend Deployment

5.2 Frontend Deployment

General

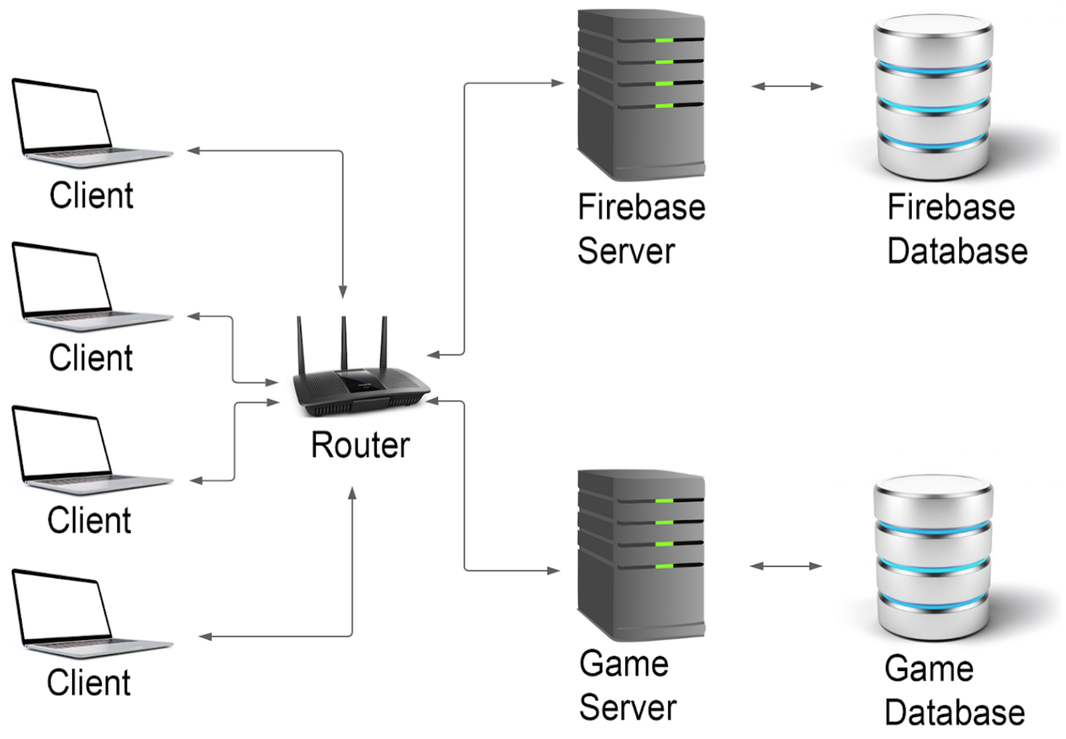
1. Project Overview

One of the biggest issues facing students during the Covid era is a lack of effective educational technology and communication. There are simply too many platforms being used that do an inefficient job handling individual students' needs. We plan to build a unified education-based messaging platform that will seamlessly connect students to their professors and other classmates. It can be hard to keep track of key dates and instructions during online learning, and providing students with the ability to message their professors in real-time will help keep them on track. Our web application will allow students to directly message professors as well as other students in the class, including image-sharing capabilities in each chat. Students will also be able to create their own group chats separately from the class itself, which can benefit them by providing a single location for all of their school-related messages to be found and stored. The platform includes an authentication system to ensure platform-wide safety and security. The platform will also include functionality such as an Office Hour queue system. The key feature of this queue is that it will allow students to play mini-games against each other to increase their position in the queue, essentially speeding up the time it takes for them to talk to the professor (if they win, of course). Waiting in a queue can often be mundane, so this will also increase student happiness while waiting to speak to a CP or Professor. The initial user authentication will be done using Google Firebase, using the student's .edu email address. From here, we will implement our own authentication system on top of Google's system for game processing.

2.1. Technical Overview

The software is implemented using Java for the back end and JavaScript for the front end. More specifically, the project uses the React web framework and Firebase website hosting to handle the front end of the web application. IntelliJ IDEA and Visual Studio Code are used for development. We will be using Firebase authentication services to handle the Google Account login data and our own servers to authenticate users for game processing.

2.2. Basic Architecture



2.3. Basic Hardware Requirements

In order to meet the requirements, we will be building a server to process all of the game data and ensure that the users have a smooth experience on our web app. This server is in charge of processing game data, sending updated data to the clients, and storing the data as needed on a local database. This local database will hold game-specific data and unique identifiers to validate our users for game purposes. The main authentication will be done by Firebase in order to support Google login, but this will then allow the client to send requests to the Java backend server.

2.4. Basic Software Requirements

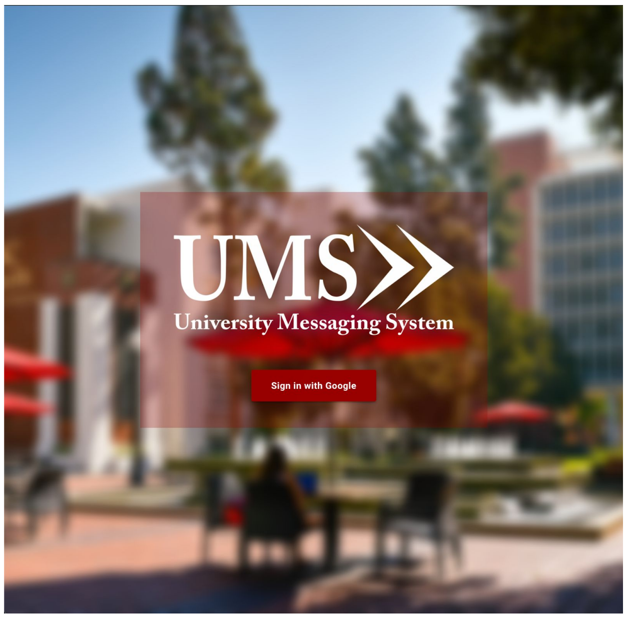
The front end of the app will be running as a website. Users will need to navigate to the site on a web browser with Javascript enabled. More specifically, Chrome or Firefox work best. Safari does not support Firebase APIs. A PC or Mac computer will be hosting the server, and this server will be running in Java. Therefore, this computer will need to have Java installed. We will be storing a local database on the server computer, so this computer will need to have MySQL installed.

Linked Resources

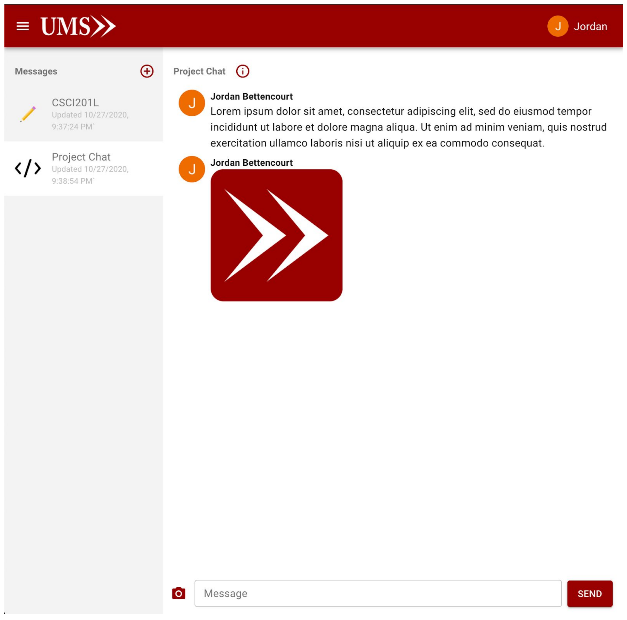
- https://medium.com/bb-tutorials-and-thoughts/how-to-develop-and-build-java-rest-api-65f708c22fb3
- https://www.boxuk.com/insight/creating-a-rest-api-quickly-using-pure-java/
- https://spring.io/guides/tutorials/react-and-spring-data-rest/
- https://medium.com/bb-tutorials-and-thoughts/how-to-develop-and-build-react-app-with-java-backend-c1e6c5c93ae
- https://scrimba.com/scrim/cBLr6Df6?pl=pgGEGtW

3.1. Frontend Screenshots

LandingPage



ChatPage with a chat open

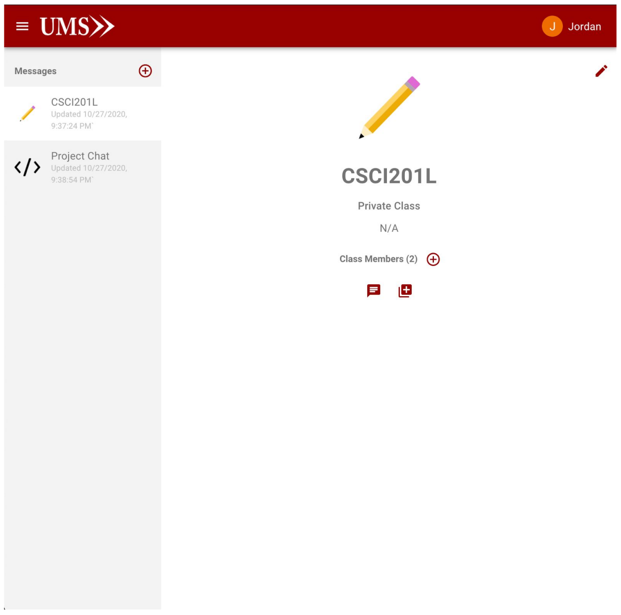


UMS
Jordan
Messages
CSCI201L
Updated 10/27/2020, 9:37:24 PM
Project Chat
Updated 10/27/2020, 9:38:54 PM
Project Chat
Message
SEND

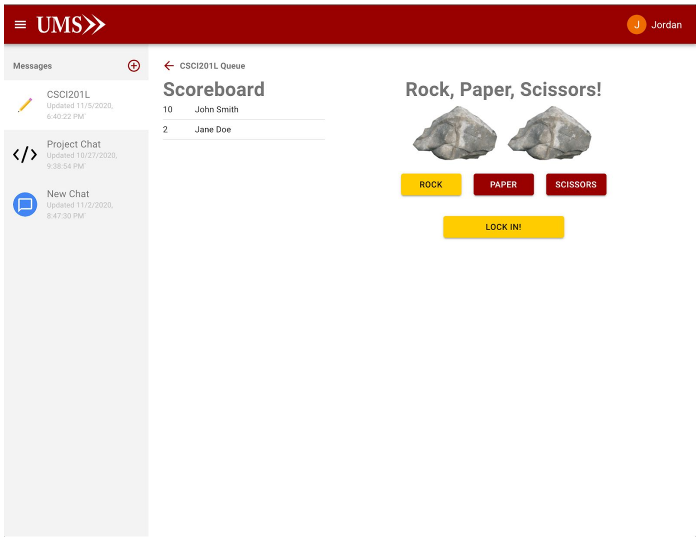
Jordan Bettencourt
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Jordan Bettencourt

ChatPage with a class open



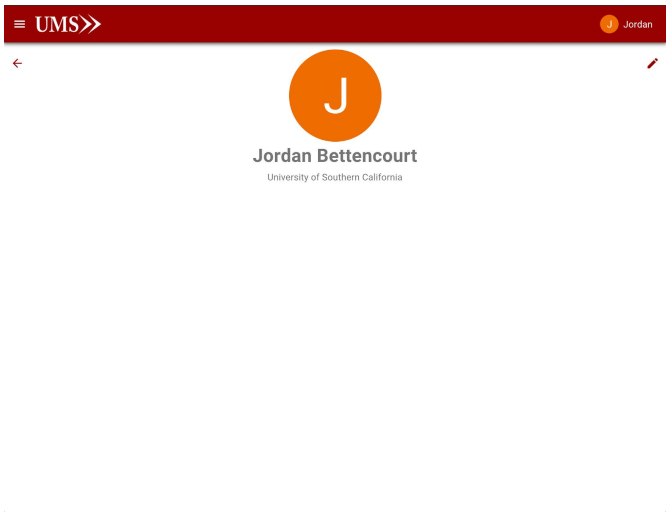
QueuePage for a classroom, showing RPS and Scoreboard components



Scoreboard

10	John Smith
2	Jane Doe

ProfilePage



3.2. Frontend Architecture Details

auth.js

Type	Name	Description
const GoogleAuthProvider	provider	Variable storing the authentication provider to connect to the Firebase server for authentication

		purposes.
export function	login()	Signs the user into our site using Firebase authentication.
export function	logout()	Signs the user out of our site.
export function	getProfilePicUrl()	Gets the user’s Google profile picture or a default placeholder if one doesn’t exist.
export function	getDisplayName()	Returns the signed-in user’s display name.
export function	getUserUuid()	Returns the signed-in user’s uuid.
export function	getUserEmail()	Returns the signed-in user’s email address.
export function	isUserSignedIn()	Returns true if a user is signed-in to our website.

chatrooms.js

Type	**Name**	**Description**
const String	LOADING_IMAGE_URL	Loading image URL for when a user uploads a file.
export function	getChatroomsListener(uid, callback)	Gets a listener for a particular user's chatrooms. The callback function should expect to be passed a snapshot when a user's chatrooms change. Returns the built listener so that the caller can detach later if they want to.
export function	getMessagesListener(chatroomId, callback)	Gets a listener for a particular chatroom's messages. The callback should expect to be passed a Firestore snapshot whenever the messages collection changes. Returns the built listener so that the

		caller can detach later if they want to.
export function	getChatroomInfoListener(chatroomId, callback)	Gets a listener for a particular chatroom. The callback should expect to be passed a Firestore snapshot whenever the data in the chatroom document is modified. Returns the listener so the caller can detach later if they want to.
export function	getMembersListHandler(chatroomId, callback)	Gets a listener for a particular chatroom. The callback should expect to be passed a Firestore snapshot whenever the data in the chatroom document is modified. Returns the listener so the caller can detach later if they want to.
export function	saveChatroomImage(chatroomId, file)	Sends a request to the Firebase server to update the given chatroom's icon image after uploading the given file.
export function	saveChatroomSettings(chatroomId, chatName, isPrivate, institution)	Updates a given chatroom's settings in the Firebase server to match the given settings.
export function	createChatroom(ownerUid, isChatPrivate, members)	Creates a new chatroom, assigning the ownerUid admin permissions for the chatroom. Also sets default settings for the chatroom in addition to allowing the members access.
export function	createClassroom(ownerUid, isChatPrivate, members)	Creates a new classroom, assigning the ownerUid admin permissions for the classroom. Also sets the default settings for the classroom in addition to allowing the members access.
export function	sendMessage(chatroomId,	Sends a message to the

	messageText)	chatroom with the given text.
export function	sendImageMessage(chatroomId, file)	Sends an image message to the chatroom with the given file's url after it has been uploaded to the storage server.
export function	loadUserProfile(uuid, callback)	Loads a given user's profile given their ID. Sends the response back to the callback.
export function	checkUserData()	Checks if the signed-in user is in our Database. If not, we add them because they are a new user. Otherwise, we look for information to update and update accordingly.
export function	saveNewUserProfile()	Saves our currently signed-in user's profile to our Database.
export function	saveMessagingDeviceToken()	Saves the FCMToken to our server so that we can send the user push notifications when new messages are sent to them.
export function	requestNotificationPermission()	Requests permission to send the user push notifications.
export function	onInviteUserSubmit(chatroomId, userEmail)	Handler for when the Invite User button is clicked on a chatroom/classroom.
export function	inviteUserToChat(chatroomId, uid)	Invite a user to the current chat based on their email.

service.js

Type	Name	Description
const Object	config	Stores the Firebase API key information.
export const Object	auth	Object for storing a reference to firebase authentication.

export const Object	firestore	Object for storing a reference to firebase Firestore.
export const Object	storage	Object for storing a reference to firebase Storage.
export const Object	messaging	Object for storing a reference to firebase Messaging.

App.js

Type	**Name**	**Description**
function	PublicRoute()	This component allows us to define a 'public route,' which will redirect to the login page if the user is not authenticated.
function	PrivateRoute()	This component allows us to define a 'public route,' which redirects to the chat page if the user is authenticated.
function	render()	Renders the App component.
function	componentDidMount()	Adds an authentication callback for when the App mounts on the page.

ChatPage.js

Type	**Name**	**Description**
const function	useStyles(theme)	Styles the chat page
function	checkUserData()	Calls the checkUserData() function from the chatrooms firebase component
function	setChatroom(chatroomID)	Sets the state of the chatroom ID
function	showProfile(show, uuid)	Sets the state for showProfile (true or false) and uuid
function	render()	Renders the ChatPage

ChatSettings.js

Type	Name	Description
const function	useStyles(theme)	Styles the chat settings page.
function	componentWillReceiveProps(nextProps)	Function to handle when the parent component passes props to this component.
function	updateSettings()	Saves the chatroom settings when the user clicks save.
function	handleCheckedChange(name, event)	Updates the state of the checkboxes when the user clicks them.
function	handleTextInputChange(name, event)	Updates the state of the text input boxes when the user types something.
function	toggleEdit()	Toggles the edit setting for chatroom settings.
function	triggerChatIconUpload()	Opens an image file chooser.
function	uploadChatIcon(e)	Receives the attached file reference and processes it.
function	render()	Renders the ChatSettings component.

ChatView.js

Type	Name	Description
const function	useStyles(theme)	Styles the ChatView.
function	handleMessages(snapshot)	This function handles firestore snapshots of changes to the current chatroom's messages collection. Assume for now that all additions come in chronological order, so all added messages are appended to the end of the messages array.

function	handleChatroom(snapshot)	We have access to all of the chatroom metadata here. We can load and await updates for settings, etc here.
function	componentDidMount()	Sets the chatroom listeners for the ChatView.
function	showSettings()	Changes the showSettings state for the ChatView.
function	hideSettings()	Changes the showSettings state for the ChatView.
function	showClassroom()	Changes the showClassroom state for the ChatView.
function	hideClassroom()	Changes the showClassroom state for the ChatView.
function	showQueue()	Changes the showQueue state for the ChatView.
function	hideQueue()	Changes the showQueue state for the ChatView.
function	componentDidUpdate(prevProps)	Component will update when the chatroomId changes. We clean up our old listeners and create new ones.
function	render()	Renders the ChatView component.

ClassroomPage.js

Type	**Name**	**Description**
const function	useStyles(theme)	Styles the ChatView.
function	componentWillReceiveProps(nextProps)	When new props are sent to the component, this will update the component's state accordingly.
function	updateSettings()	Responsible for updating the classroom's settings when the user clicks save.

function	handleCheckedChange(name, event)	Updates the state of the checkboxes when the user clicks them.
function	handleTextInputChange(name, event)	Updates the state of the text input boxes when the user types something.
function	toggleEdit()	Toggles the edit setting for chatroom settings.
function	triggerChatIconUpload()	Opens an image file chooser.
function	uploadChatIcon(e)	Receives the attached file reference and processes it.
function	render()	Renders the ClassroomPage component.

ComposeBar.js

Type	**Name**	**Description**
const function	useStyles()	Styles the CompseBar.
function	handleKeypress(event)	If the enter key was pressed, we will send the message instead of adding a new line to the message.
function	componentDidMount()	Register the keypress handler.
function	componentWillUnmount()	Deregister the keypress handler.
function	updateMsg(event)	Sets the state of the component to update the InputText element.
function	sendMsg()	Validates and then sends the message to the server.
function	focusMessageTextInput()	Focuses the message input so the user can keep typing a new message.
function	onImageChange(event)	Handles when an image gets attached so we can send it to

		the chatroom.
function	render()	Renders the ComposeBar component.

LandingPage.js

Type	Name	Description
function	render	Renders a landing page for the user to log into the web app with their Google account.
const function	useStyles	Styles the login page.

LoginButton.js

Type	Name	Description
async function	handleLogin()	async/await multithreaded function to handle login asynchronously.
function	render()	Renders the Sign in with Google login button.

Member.js

Type	Name	Description
const function	useStyles(theme)	Styles the Member component.
function	render()	Renders the Member component.

MemberView.js

Type	Name	Description
const function	useStyles(theme)	Styles the Member

		component.
function	handleMembers(doc)	Receives a user’s document and sets the component’s state accordingly.
function	componentDidMount()	Creates the members listener if a chatroomId is passed in.
function	componentWillUnmount()	Clears the listeners.
function	componentDidUpdate(prevProps)	Updates the component if the chatroomId has changed.
function	showForm()	Toggles the showForm state of the component.
function	addNewMember(event)	Calls onInviteUserSubmit() to add a new member to the chatroom and clears the form.
function	updateMember(event)	Updates the newMember state.
function	render()	Renders the MemberView component.

Message.js

Type	**Name**	**Description**
function	useStyles(theme)	Styles the message component.
function	render()	Renders the message component.

NavBar.js

Type	**Name**	**Description**
function	useStyles(theme)	Styles the NavBar component.
function	showMenu(event)	Shows the navigation menu to the user.

function	hideMenu(event)	Hides the navigation menu from the user.
function	render()	Renders the NavBar component.

ProfilePage.js

Type	Name	Description
function	useStyles(theme)	Styles the ProfilePage component.
function	render()	Renders the ProfilePage component.

ProfileView.js

Type	Name	Description
function	useStyles(theme)	Styles the ProfileView component.
function	componentDidMount()	Loads the user's profile when the component is rendered on the page.
function	handleCheckedChange(name, event)	Updates the state of the checkboxes when the user clicks them.
function	handleTextInputChange(name, event)	Updates the state of the text input boxes when the user types something.
function	loadUserProfileCallback(snapshot)	Parses and sets the component's state to the incoming user's data from the snapshot.
function	render()	Renders the ProfileView component.

QueuePage.js

Type	Name	Description
function	useStyles(theme)	Styles the QueuePage component.
function	componentWillReceiveProps(nextProps)	When new props are sent to the component, this will update the component’s state accordingly.
function	updateQueue()	Updates the queue if the user is admin for the classroom.
function	handleCheckedChange(name, event)	Updates the state of the checkboxes when the user clicks them.
function	handleTextInputChange(name, event)	Updates the state of the text input boxes when the user types something.
function	render()	Renders the QueuePage component.
function	componentDidMount()	Adds the user to the queue.
function	toggleEdit()	Toggles whether the user is in edit mode or not.
function	addUserToQueue()	Adds the current user to the queue on the Java backend server.
function	removeUserFromQueue()	Removes the current user from the queue on the Java backend server.

Scoreboard.js

Type	Name	Description
function	useStyles(theme)	Styles the Selector component.
function	componentDidMount()	Set the state of the component to keep updating the scoreboard.

function	componentWillUnmount()	Stops the component from repeatedly fetching updates since the scoreboard is no longer displaying.
function	fetchScoreboardUpdate()	Fetches data from our Java server to update the scoreboard.
function	lockTopMember()	Locks the top member and removes them from the queue in the Java backend.
function	removeLockedMember()	Removes the locked member from the scoreboard.

Selector.js

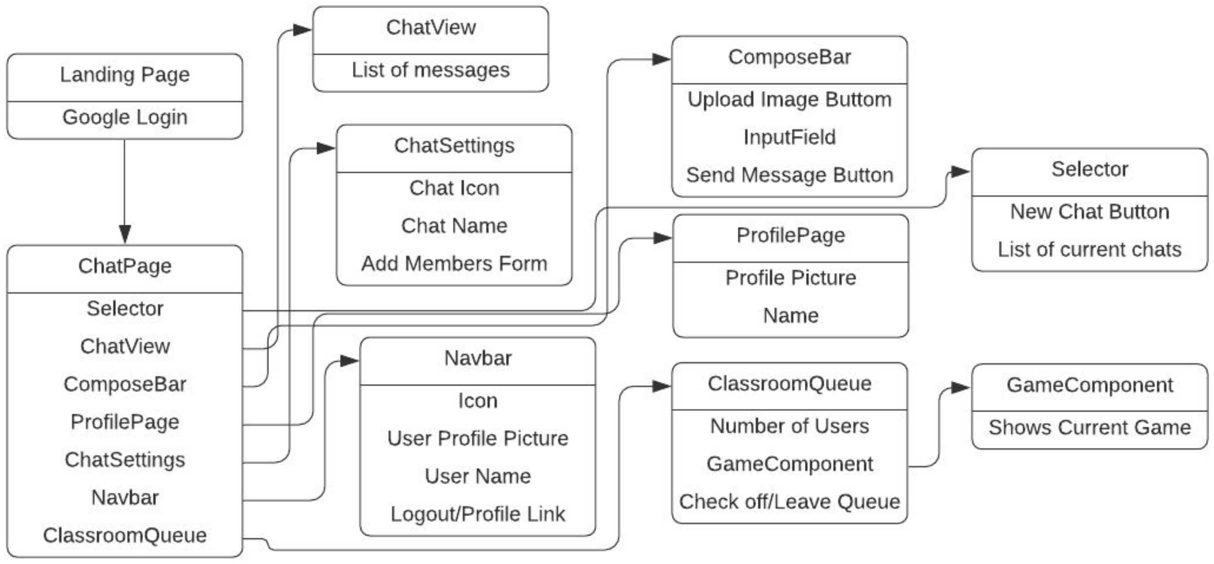
Type	Name	Description
function	useStyles(theme)	Styles the Selector component.
function	handleChatrooms(snapshot)	
function	createNewChatroom()	Creates a new chatroom environment for users.
function	componentDidMount()	
function	formatTimestamp(timestamp)	Formats the timestamp.
function	render()	Renders the Selector component.

RPS.js

Type	Name	Description
function	useStyles(theme)	Styles the RPS component.
function	componentDidMount()	Sends the user email to back end so they know who is playing.
function	componentWillUnmount()	Clears the interval of the timed loop to prevent infinite

		loop in case user leaves game.
function	fetchRPSUpdate()	Fetches the opponent data from API. Updates component states accordingly
function	start()	Makes a call to fetchRPSUpdate() in a timed loop to constantly check if opponent move has been made. Updates the state accordingly.
function	selectWinner()	Shows the appropriate ending screen (win, lose, tie) based on the winner variable returned by API call
function	selectChoice()	Makes a PUT API Call sending the user selection after he hits the lock in button
function	render()	Renders the three options of rock, paper, and scissors as buttons that the user can click to make their selection

3.3. Client Architecture (Est. 40hr)



The entire web app is launched in App.js. This takes care of the basic routing which will launch the user into the Landing Page or the Chat Page if they are logged out or in, respectively. The above image shows the route as if the user was initially logged out. Once the user is on the Chat Page, they will see their current chatrooms and classes, in addition to several different components: the Navbar takes care of top-level navigation; the ChatSettings component shows information about the current chatroom/class and allows the user to update the information if they have admin rights; the Chat View shows all of the current messages in the current room/class; the Profile Page allows the user to look at their own profile in addition to the profile of others; the Compose Bar allows the user to send a message or upload an image into the current chat; The Selector shows all of the user's current classes and chatrooms, sorted by newest timestamp; the ClassroomQueue allows professors to create an office hour queue system with games to keep students entertained; the Game Component will be the wrapper for the games that are hosted in the ClassroomQueue system, which for the sake of time, only supports RPS (Rock Paper Scissors).

3.4. Server Architecture (Est. 40hr)

Overview

The plan for the backend is still broad in our approach (depending on how many games we will be able to develop) and will be more fine-tuned by the end of the next week. We will be using Springboard as a REST API service for the backend which will interact with multiple threads and a database. Sidenote: UID might be swapped out for email authentication if the frontend would prefer.

3.4.1. Springboard API

Springboard is a REST API architecture which will make it easy for React to make simple calls to the backend. We will use a RESTful architecture to design this application, likely with only a dozen or so endpoints. Here is a preliminary list of API endpoints with the priority of creation/importance for the application:

API Call	Parameters/Results	Backend Action
GET /game/join/{type}/{email} • **201** Created • **202** Accepted • **404** Not Found	Parameters: • type: "rps", "ttt" • email: "example@gmail.com" Return: HTTP Status/Result • **201** Created ○ Created new game • **202** Accepted ○ join request works ○ return game ID (String) • **404** Not Found ○ game type does not exist Sample Return: gameID `{"556dc391"}`	Logic: • INPUT person **A** • IF queue is empty ○ Create new game with **A** ○ Add game to queue • ELSE IF queue is not empty ○ Pop game **G** from queue ○ Add **A** to game **G** • ENDIF • Return gameID of **G** Controller attempts to match the player with the existing game. If a game does not exist, a new game is created. Every time a game is created it is added to a map with gameID as key and the game as the value. gameID's are generated internally in Game objects, gameID's are returned to the frontend for future access to game data.

GET /game/{gameID}
• **200** OK
• **404** Not Found
	Parameters:
	• gameID: "12345"
	
	Return: HTTP Status
	• **200** OK
	  ○ valid game ID
	  ○ return JSON of game
	• **404** Not Found
	  ○ game ID does not exist
	
	Sample Return: game JSON
	`{"score": 0,`
	` "gameID": "556dc391",`
	` "gameType": "rps",`
	` "gamePoints": 1,`
	` "user1": "u1@gmail.com",`
	` "user2": "u2@gmail.com",`
	` "state": 5,`
		A Game object is stored in a ConcurrentHashMap in the back end Game Controller. Through Spring, the frontend can request the game JSON to view the current state of the game and general game information.

	`"user1move": "1",`
	`"user2move": "2",`
	`"winner": "1"}`	

PUT /game/{gameID}/{selection}/{email} • 200 OK • 401 Unauthorized • 404 Not Found • 406 Not Acceptable • 409 Conflict	Return: HTTP Status • 200 OK a. Move was made • 401 Unauthorized a. Email does not match email of users in game b. Game has ended • 404 Not Found a. game ID does not exist • 406 Not Acceptable a. invalid game move requested • 409 Conflict a. Not client's turn, waiting for opponent b. Game has not started, waiting for an opponent	When a POST request is made, the backend finds the correct Game object by the provided gameID and makes a move using the selection and user information provided. This is done by calling the move method that each Game object has.

GET /scoreboard	Return:
	• Current scoreboard standings
	
	Sample:
	`{"User2":2,`
	`"User1":1,`
	`"User3":3}`
		HashMap of users and their current score is stored in the scoreboard controller

GET /scoreboard/user/{email}	Parameters: • User email Return: • Specific user that matches the given email	Returns user that matches the given email
DELETE /scoreboard/user/{email}	Parameters: • User email Return: • Users current value	Removes email and user from the scoreboard using the given email

For authentication, we will add email and a game ID as a required parameter for all game related calls. Also, based on the return code, the client side will be able to interpret what to display. These statuses can follow standard HTTP protocol as found here https://httpstatuses.com/.

3.4.2. Game Object

This is the main logic for the games that are available such as Rock Paper Scissors. Each specific game will extend directly from the abstract class Game. When a request is made from the front end, the request is routed through the Game Controller and into individual Game objects. Such requests include joining a new game, making a move in a game, or getting the game data. The Game objects will then handle all game logic internally using the data members and methods below.

Game.java - abstract class for all games

Data Members

Modifier and Type	Name	Description
`protected UUID`	**`gameID`**	Unique game ID for each lobby
`protected String`	**`gameType`**	String identifier for game type
`protected int`	**`gamePoints`**	Number of points the game is worth
`protected String`	**`user1`**	ID of player 1
`protected String`	**`user2`**	ID of player 2
`protected int`	**`state`**	Stores state of the game (which players turn, which player won, etc)
`protected String`	**`user1move`**	Most recent move for player 1
`protected String`	**`user2move`**	Most recent move for player 2
`protected String`	**`winner`**	Winner of most recent round or game

Methods

Modifier and Type	Name	Description
`protected boolean`	**`addUser(`**`String u2`**`)`**	Adds player 2 to game
`protected abstract String`	**`move(`**`String user,String move`**`)`**	Makes move for user
`protected abstract void`	**`updateGame()`**	Updates game state or winner

RPS extends Game - rock paper scissors, example of Game child class

Data Members

Modifier and Type	Name	Description
`int`	`score`	Keeps track of score, best of 3 rounds
`*includes all data members from Game.java`		

Methods

Modifier and Type	Name	Description
`protected boolean`	`addUser(String user2)`	Starts game, calls super.addUser(user2)
`public String`	`move(String user,String move)`	Updates move for user, calls updateGame()
`public void`	`updateGame()`	Updates game state, winner, score

TTT extends Game - tic tac toe, another example of Game child class

Data Members

Modifier and Type	Name	Description
`private int[]`	`board`	Tic Tac Toe game board
`private String`	`playerX`	Player that goes first
`private String`	`playerO`	Player that goes second
`*includes all data members from Game.java`		

Methods

Modifier and Type	Name	Description
`protected boolean`	`addUser(String user2)`	Determine which user goes first, start game, calls super.addUser(user2)
`public String`	`move(String user,String move)`	Updates move for user, updates board, calls updateGame()
`public void`	`updateGame()`	Updates game state, winner

UID.java - uses java.util.UUID to generate unique game IDs

Methods

Modifier and Type	Name	Description

`public static UUID`	`getID()`	Returns a unique UUID

3.5. Firebase Database Architecture (Est. 5hr)

Firebase schema:

- users
    - <user uid>
        - name
        - profile picture url
        - email
        - institution
            - majors (list of major uuids)
                - name of major
            - interests (list of interest uuids)
                - name of interests
- chatrooms
    - <uuid>
        - name
        - private type (true, false)
        - chatIconUrl
        - institution
        - membersArr (array of user uuids)
        - adminsArr (array of admin user uuids)
        - timestamp
        - messages
            - <uuid>
                - user’s name
                - user’s uuid
                - text OR imageUrl
                - timestamp
                - profilePicUrl (of user)
- fcmTokens
    - <token id>
        - user uid

3.6. Flows

User Sign Up

1. User hits the landing page.

2. User clicks "Sign in with Google".
3. System creates a document for the user with form data.
4. System adds the user to relevant chat rooms.

User Creates Chat Room/Group Message

User clicks the “New Message” button. This will show a small popup menu, which will let the user choose between a Chat Room or a Classroom. Clicking either of these will generate a new chat room where the user is by themself, with presets for either a classroom or a chat room. They will have to invite new people to the chat and change the chat’s settings in the ChatSettings page.

User Sends a Message

1. User composes a message in the message box.
2. User presses 'send'.

User Sends an Image Message

1. User clicks on the camera button and a file picker pops up. The maximum file size is currently set to 5MB.
2. When the user selects the image, it uploads to the database. The loading spinner is then updated once the file is successfully processed to display the image.

User Receives a Message

1. The message automatically loads in the respective chatroom and displays as its own message. This shows the sender’s profile picture, name, and the message itself. If it is an image message, an image shows. If it’s a text message, text shows.
2. We also update the timestamp of the chatroom whenever a new message appears. This allows us to update the order of the chatrooms on the sidebar.

3.7. Screens

Navbar

The Navbar will have a logo on the left. On the right, we will allow the user to sign out and view their profile in addition to showing their display name and image.

Landing Page

The landing page is where users that aren't signed in go. This shows an image in the background and a logo. The users can then click a “Sign in with Google” button. Acts as the Login Page.

Chat Page

The messages page is where users can see their active chat rooms and private messages. We probably want to use a master-detail view, with a list of rooms on the left side of the screen, and then the contents of the messages themselves taking up the rest of the available space. On small displays, the master-detail view should probably collapse into a hamburger menu.

Chat Settings Page

Allows the user to change the settings of the current class/chatroom if they have administrative rights on the class/chatroom. If they don’t it simply displays the general information of the room to the user.

Profile Page

Shows a user’s basic information. This includes profile picture, name, and potentially other details such as their major if they have it updated on the web app. If it is the current user’s profile, we will give them the option to edit the fields.

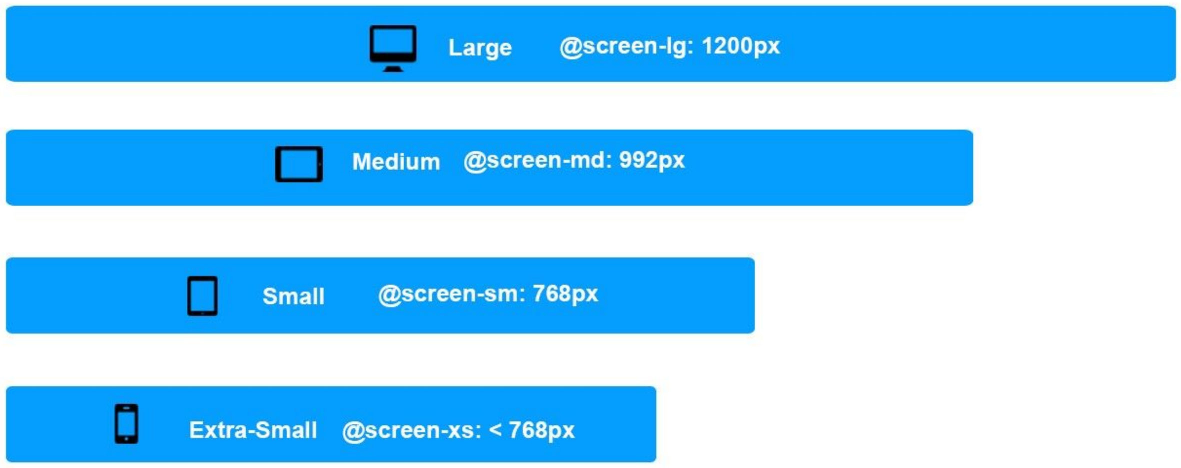
Queue Page

Shows the users spot in queue, the other students waiting in queue, and how many points each student has. Point totals will be updated after students play the minigames. Users waiting in queue can open this games page which will display the games they can challenge other players to. The queue will use a points system to see who gets selected next in the queue. By winning a game, you will win a certain amount of points, and if you lose, you will lose those points.

4. Testing

4.1. Frontend Testing:

- Test screen sizes



-
- Verify firebase works correctly and uploads data to our 'users' collection when users log in. This works when a user's information is properly displayed in the top right corner when logged in and when sending messages.
- Test that correct rock paper scissors graphics get chosen based on user selections using Jest.
- Test that the correct rock paper scissors winner is received from REST API.
- Test the scoreboard displays sample data properly.
- Test that messages are properly sent to a chatroom. This will need to be done via manual testing.
- Test that new classrooms/chatrooms are created properly. For security reasons, this will need to be done manually.
- Test that changing the name/chat icon/chat settings/class settings changes the data in the database properly. For security reasons, this will need to be done manually.
- NOTE: Due to security reasons, Firebase Messaging will report that the service worker could not be created when running on localhost or when attempting to test functionality via jest or other unit tests. This is normal behavior and is not something we can work around. For this reason, any test cases that we try to make for messaging capabilities on the frontend will fail. These must be manually verified for the time being.
- NOTE 2: All automated tests can be run by navigating to /ums/web/ in a terminal and running "npm test" in the frontend files.

4.2. Backend Testing:

- Test API Endpoints
    - Test Input validation for all POST requests
    - Verify GET requests return the correct result

- Test Game Object
- Test each game calculate result function
- Controller tests
    - Http code returns
    - Serialization correctness
- Test POST requests
    - Correctly stores email and username
    - Verify return behavior (201 or 409)
- Test DELETE requests
    - Correctly deletes email + username record
    - Doesn’t corrupt the database if not found
    - Verify return behavior

Command line test program for backend game logic

```
public class GameTest{
   public static void main(String[] args) throws IOException {
       Scanner in=new Scanner(System.in);
       Gson gson=new GsonBuilder().setPrettyPrinting().create();
       //front end request certain game
       boolean valid=false;
       String gameType="";
       Game g;
       while(!valid){
           System.out.print("Game Type (rps,ttt): ");
           gameType=in.nextLine();
           if(gameType.equals("rps") || gameType.equals("ttt")){
               valid=true;
               System.out.println("Accepted!");
           }
           else System.out.println("Invalid game type.");
       }
       //user identification to be obtained from frontend
       System.out.print("User 1: ");
       String user1=in.nextLine();
       //user 1 joins game, display JSON
       if(gameType.equals("rps")) g=new RPS(user1);
       else g=new TTT(user1);
       System.out.println("\nJSON returned:\n"+gson.toJson(g)+"\n");
       //add user 2, display JSON
       System.out.print("User 2: ");
       String user2=in.nextLine();
       g.addUser(user2);
       System.out.println("\nJSON returned:\n"+gson.toJson(g)+"\n");
       String u;
       while(g.state!=5){ //loop while game is ongoing
           //frontend sends move request
```

```
            System.out.print("Enter user requesting a selection: ");
            u=in.nextLine();
            if(gameType.equals("rps")) System.out.print("Enter selection for
             "+u+" (0,1,2): ");
            else if(gameType.equals("ttt")) System.out.print("Enter move for
             "+u+" (rowcol): ");
            String move=in.nextLine();
            //attempt move request
            System.out.println(g.move(u,move));
            System.out.println("\nJSON returned:\n"+gson.toJson(g)+"\n");
        }
        //Display final JSON
        System.out.println("\nGAMEOVER\nJSON returned:\n"+gson.toJson(g)+"\n");
    }
}
```

5. Deployment Documentation

5.1. Backend

Download the zipped file containing the eclipse project. This .zip file will include the GSON jar file and other Spring related jar files, all of which will be linked with maven. To import this project into eclipse:

1. Unzip the project.
2. Open Eclipse.
3. Click File -> Import.
4. Navigate to Maven -> Existing Maven Projects.
5. Navigate to the /FinalProject/ directory. Select the /dev directory in the project folder and click “Open”.
6. Eclipse will show the pom.xml file. Make sure the checkbox next to pom.xml is checked and click “Finish”.
7. Open /src/com/ums/backend/UMSBackend.java. Click “Run”. A Spring Application instance will open in eclipse’s terminal.
8. The backend server is running! Time to start up the web application’s frontend! Note: The web application is currently looking for a backend server running on localhost:8080. We have also set up a server running on http://206.189.75.50:8080. If you want to use this server, comment and uncomment a line in a file located in the frontend’s project files at /ums/web/src/components/game-logic/BackendServerLocation.js. All documentation is in the file.
9. If, for any reason, you would prefer to use our prepackaged backend jar file, navigate to /FinalProject/dev/target/ in your terminal and type “java -jar umsbackend-0.0.1-SNAPSHOT.jar”. This will start the backend server which we have already compiled from the sources of the backend project.

5.2. Frontend

Download the zipped file containing the web app source files. Also download and install NPM if you don't have it already. The .zip includes all assets needed to compile the ReactJS web application, except for npm libraries. To run this website locally:

1. Unzip the project.
2. Navigate to /ums/web/ in your terminal or command line interface.
3. Run "npm install". This installs all dependencies for the ReactJS web application.
4. Once this completes, run "npm start".
5. Congratulations! The web application is being served locally on port 3000. Please note, Safari currently does not support the needed web interfaces that Firebase relies on. As such, please use Firefox or Chrome to ensure the site will display properly.
6. Alternatively, the web app is being hosted at https://university-messaging-system.web.app/chat. This website is looking for a backend server on your localhost network, but editing the file located at /ums/web/src/components/game-logic/BackendServerLocation.js will allow for integration with the backend server we have running remotely. Please ensure that your server is running prior to using the web app if you plan to use game and/or queue features within a classroom.