**Programming session #1:** Wednesday, December 6 @ 4:00PM – November 30 @ 2:00AM (with breaks in between)

- My component for this project is creating the notes section on the meeting dashboard. My plan for building out this feature is to first build out the UI for the chat, and then connect it with a database. While building the UI, I will also add in the functionality to convert items to markdown, edit, and delete comments.

- The chat in itself is a fancier list, so I will use Reactstrap's list component to build out my chat, where each new message is a list item.

- I will now add two buttons ("edit" and "delete) that will appear when the note is hovered over. Since each note item will need to have this, I will include these buttons directly in the NoteItem component I'm building.
    - I will need to import the Aphrodite library in order to use CSS, since I want to use the ":hover" selector.
    - When trying to edit a note, clicking "edit" will reveal a <textarea>, which is fairly simple logic.
    - However, when clicking "delete", I would like a modal to be toggled. However, I realized that it would be inefficient to create a new modal for each message.
    - So, what I did is that I created a <modal> component on the same level as the notes list. Each message would be passed a callback function that would toggle the <modal>'s state, allowing it to appear/disappear on click for each message.

- Because Reactstrap uses a row/column system as opposed a true grid system with Flex/Material, vertical positioning on the note display and the note input box within the note chat is made more difficult. I don't want the amount of space that each component to change with screen size, so I am thinking about either using "vh" or "%" to be able to have the chat resize in response to screen width.
    - I first tried to implement this using "%" units, since it better represented the problem, since it's not dependent on either screen width or height. I was having some issues with having the "%"s render. After looking online, I realized that other people were having issues with using "%".
    - I ended up using "vh". Even though "vh" isn't the perfect solution for this problem since it's dependent on screen height, the way that I proportioned the input box and the notes display made it such that it should work on almost all displays.

**Programming session #2:** Thursday, December 6 @ 2:00PM – November 30 @ 2:30AM (with breaks in between)

- Originally, each edit input would turn back into displayed HTML from a <textarea> when enter was pressed. Because of this, I quickly realized that I was unable to make multi-line notes.
    - To fix this, I modified the event listener for keypress events to listen for keypresses of "enter" and "shift" at the same time. At first, I thought I would have to keep track in my state which keys were pressed, since the keypress only tracks one key pressed at a time.
    - However, after doing some more research, I learned that there is an "shiftKey" attribute per event object that tracks whether the shift key is pressed alongside any key, so I ended up using that.

- Currently, the way that each note is implemented is that anyone can edit each note. However, I only people who created a note have the permissions to edit and delete it. Implementing this feature was fairly simple because all it required was taking in the currentUser prop and comparing as to whether the display name of the current user matched the user who sent the message.

- With all the basic functionality of my UI done, it was now time to connect to Firebase. I decided to not create my database connections at the same time I was developing my UI, since it would be hard to debug, since I wouldn't know whether issues were coming from the UI or from the database calls.
    - My thought is to create a new table called "messages", with keys mapping to each meeting ID hash. If there is a message sent in the meeting, then there will be new messages with unique IDs as an array under the meeting ID hash.

- I also implemented the feature so that only people who created a note have the permissions to edit and delete it. This was fairly simple because all it required was taking in the currentUser prop and comparing as to whether the display name of the current user matched the user who sent the message.

**Programming session #3:** Friday, December 9 @ 5:00PM – November 30 @ 10:00 PM (with breaks in between)

- My goal for this programming session is to be able to implement a new feature that Joel suggested, where each note will have a button that opens the task modal with some prepopulated data, if applicable. My first goal is to be able to have the task modal open on click of this button.
    - The difficulty in being able to understand how to structure this is that the task modal is not a direct ancestor to each note. That is, the task modal is a sibling to each note. I'm not sure how to approach this problem entirely, since I've resolved state updating from parent to child, and child to parent, but never child to a parent's sibling.
    - I realize that this problem is a combination of updating a parent's state and updating a child's state. So, I decided to pass back a callback function that would update a parent's showModal state, which would represent whether the task toggle should be displayed. Then, it would pass the state down to the <modal> component as a prop. If the parent's showModal state is true, then the modal would call its toggle() function, and if not, then it would not be shown.
    - While trying to develop this feature, I ran into problems getting the toggle() function to ever be shown, since toggle() was never being called. I originally put the logic in componentDidMount(), but toggle was never called because the component's props were never were defined. After debugging, I realized the same was the case for componentWillMount(). I even tried it in the constructor, but this didn't yield any results. I then did research on React lifecycle events, and tried to find an event in which a component was passed props. I found that this was the componentWillReceiveProps() function. After I moved my logic in to this life cycle event function, the functionality worked!

- I now will write an algorithm that will allow me to extract the first instance of a highest level header I can find and populate it as the task's title. This problem seemed quite complex, but I first broke it down to simpler tasks.
    - I think I will approach this by doing a string.indexOf() search for each header tag from 1 to 6, and store them correspondingly in an array from 0 to 5. I then thought that since indexOf() returns the first instance of a substring, I could leverage this fact to be able to find the earliest instance of a certain tag.
    - Then, the algorithm becomes much simpler, because it becomes searching through the array from the lowest index to the highest index for the first instance of indexOf()'s result is not -1.