Memento design patterns

Applied on: Chat feature after leaving room and tapping chat

```
import ChatScreenContent from "../../components/organisms/ChatScreenContent";
import { useSelector } from "react-redux";

const ChatScreen = (props) => {
    const socket = useSelector((state) => state.socket).socket

    console.log(socket.connected)

    useFocusEffect(
        useCallback(() => {
            if (!socket.connected) {
                socket.connect()
            }

            socket.emit('join-room', 'Location 1')

            const leaveRoom = (room) => {
                socket.emit('leave-room', room)
            }

            return () => leaveRoom('Location 1');
        })
    )

    return (
        <ChatScreenContent></ChatScreenContent>
    )
}
```

How is it applied: Once user leaves the room, the latest chat is copied over to the “Chat” screen located on the home page. This style of implementation allows users to view their chat history with a store with minimal loading time as the application does not need to search for the last chat room that the user was in and retrieve the information from when the user joins to when he/she leaves.

Chain of Responsibility Pattern

Applied on: Registration feature

```
async function validateLoginInput(data) {
  let errors = {};
  // Check the password and confirmed password are equal
  if (data.password != data.confirmationPassword) {
    errors.password = "Password Mismatch";
  }
  // Check if the email already exists
  if (
    await signUpTemplate.findOne({
      email: data.email,
    })
  ) {
    errors.email = "Email already exists";
  }
  // Check if the username already exists
  if (
    await signUpTemplate.findOne({
      userName: data.userName,
      googleRegistered: false,
    })
  ) {
    errors.userName = "Username already exists";
  }

  return { errors, isValid: isEmpty(errors) };
}
```

```
async function validateLoginInput(data) {
  let errors = {};
  // Check if the email already exists
  if (
    await signUpTemplate.findOne({
      email: data.email,
    })
  ) {
    errors.email = "Email already exists";
  }
  else{
    // Check if the username already exists
    if(
      await signUpTemplate.findOne({
        userName: data.userName,
        googleRegistered: false,
      })
    ) {
      errors.userName = "Username already exists";
    }
    else{
      // Check the password and confirmed password are equal
      if (data.password != data.confirmationPassword) {
        errors.password = "Password Mismatch";
      }
    }
  }

  return { errors, isValid: isEmpty(errors) };
}
```

How is it applied (Suggested): During the registration function, the checking of the process will stop if an email exists first, then a user name, followed by a password matching check. This makes the checking step efficient as the application will not need to process any field pass the invalid field and will only proceed if everything is in order. This also allows other checks to be implemented easily for the future.

Observer Pattern

Applied on: Shops to be displayed

How is it applied: Details regarding store that is displayed will be constantly changing and or updating based on different users using the application. When a change in the user's device is detected, the app will automatically update the relevant information about the shops. One example is as such, a user will be confused if he/she lives in Pioneer but gets shown a random shop in Changi. However, a user can also search for a shop in Changi via the search feature. Hence in this situation the user is the observer, and the shops are the subject