**Technologies Used:**

**Requirements:**

**File Structure:**

* **‘/src’:** Source code files. Includes *‘server.py’* and *‘user.py’* files.
* **‘/src/static’:** Base folder for css, javascript, and images folders
* **‘/src/templates’:** Folder for html files. Also includes base-templates folder.
* **‘/src/static/css':** Folder for all css files.
* **‘/src/static/javascript’:** Folder that includes *‘chat.js’* file.
* **‘/src/static/images’:** Folder to store image assets used.
* **‘/src/templates/base-templates’:** Folder which stores html files used as a base for other pages. Benefists from *‘Jinja Templates’*.

**Source Code:**

* *Figure numbers should be added for all the images. We can do it after every image is placed.*

**user.py file:**

metin, ekran görüntüsü, yazı tipi, iş kartı içeren bir resim

Açıklama otomatik olarak oluşturuldu

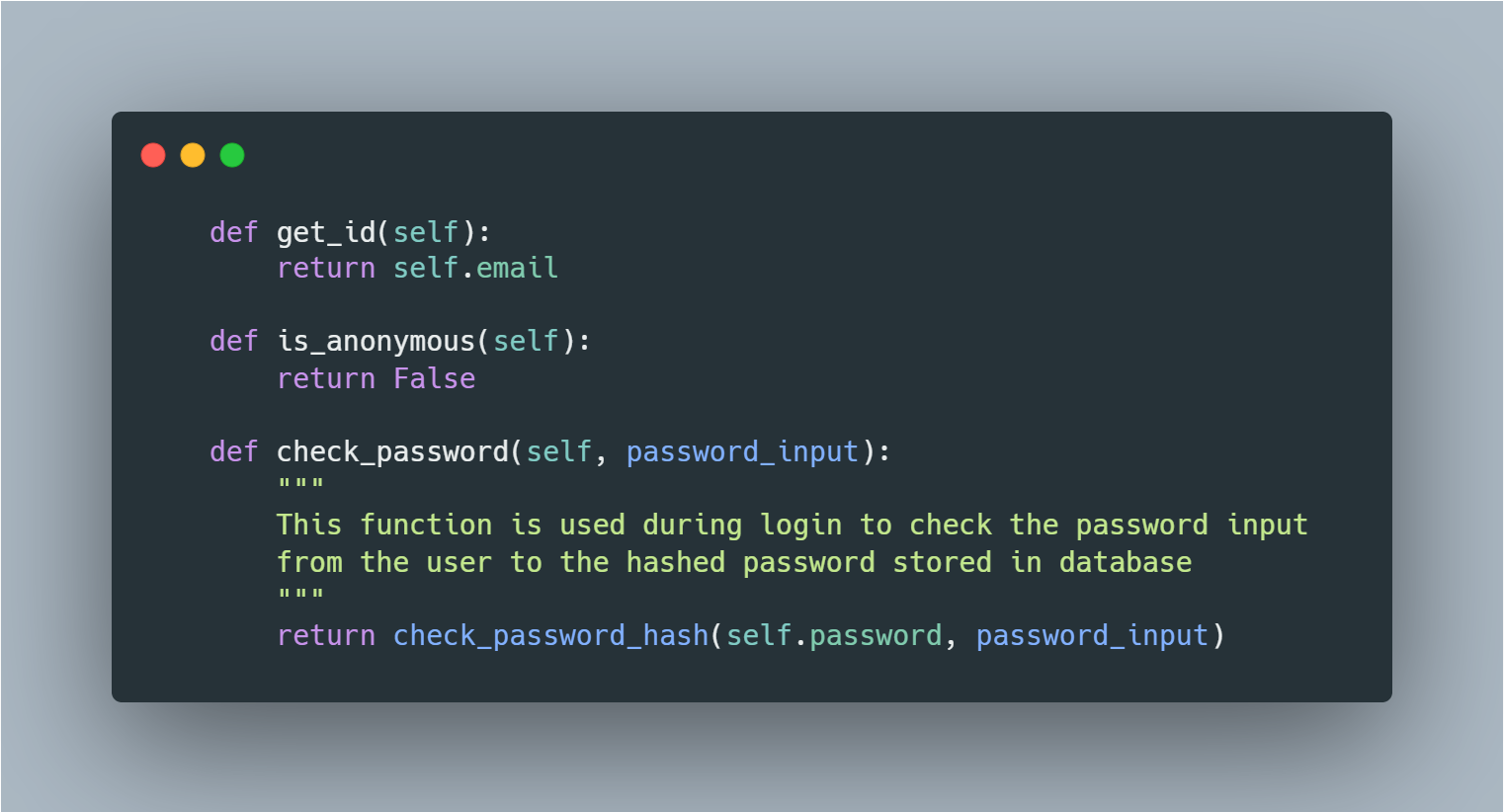
* *Imports will be explained here.*



Class *‘User’*. It inherites from class *‘UserMixin’* which is a class from *‘flask\_login’* module. *‘UserMixin’* provides default implementations for methods like *‘is\_authenticated()’* and *‘is\_active()’*, which are commonly used in user authentication systems.

*‘\_\_init\_\_()’* method initializes the following attributes of the *‘User’* class:

* **name:** The name of the user.
* **email:** The email address of the user.
* **password:** The password of the user.
* **department:** The department to which the user belongs.
* **user\_follower\_count:** The count of followers for the user (default is 0).
* **followed\_id:** A list of user IDs that the current user is following (default is an empty list).
* **followed\_community:** A list of community IDs that the current user is following (default is an empty list).
* **user\_community\_count:** The count of communities the user is a part of (default is 0).



These methods belong to*‘User’* class.

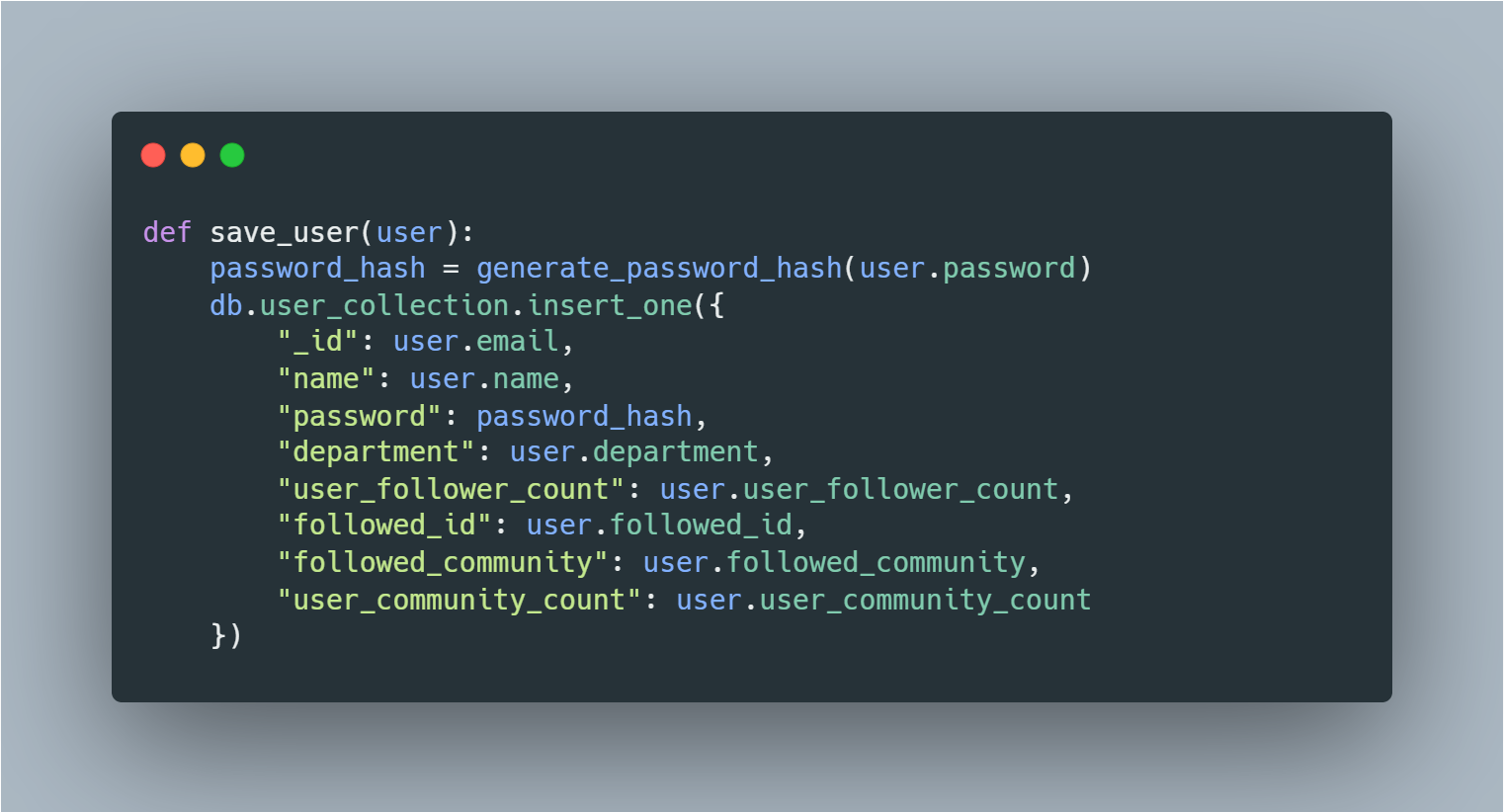
*‘get\_id()’* method is supposed to return a **unique identifier** for the user**.** In this implementation,*‘email’* attribute of *‘User’* class is used as a **unique identifier.** Hence, this method returns the relevant email address of the user.

*‘is\_anonymous()’* explanation goes here…

*‘check\_password()’* method is used to check if a provided password matches the hashed password stored for the user. The check\_password\_hash function is part of a password hashing library, werkzeug.security. This method is used during the login process to verify the correctness of the entered password.

* **self.password:** The hashed password stored in the user object.
* **password\_input:** The plain text password input provided during the login attempt.

The method returns True if the password matches, and False otherwise.



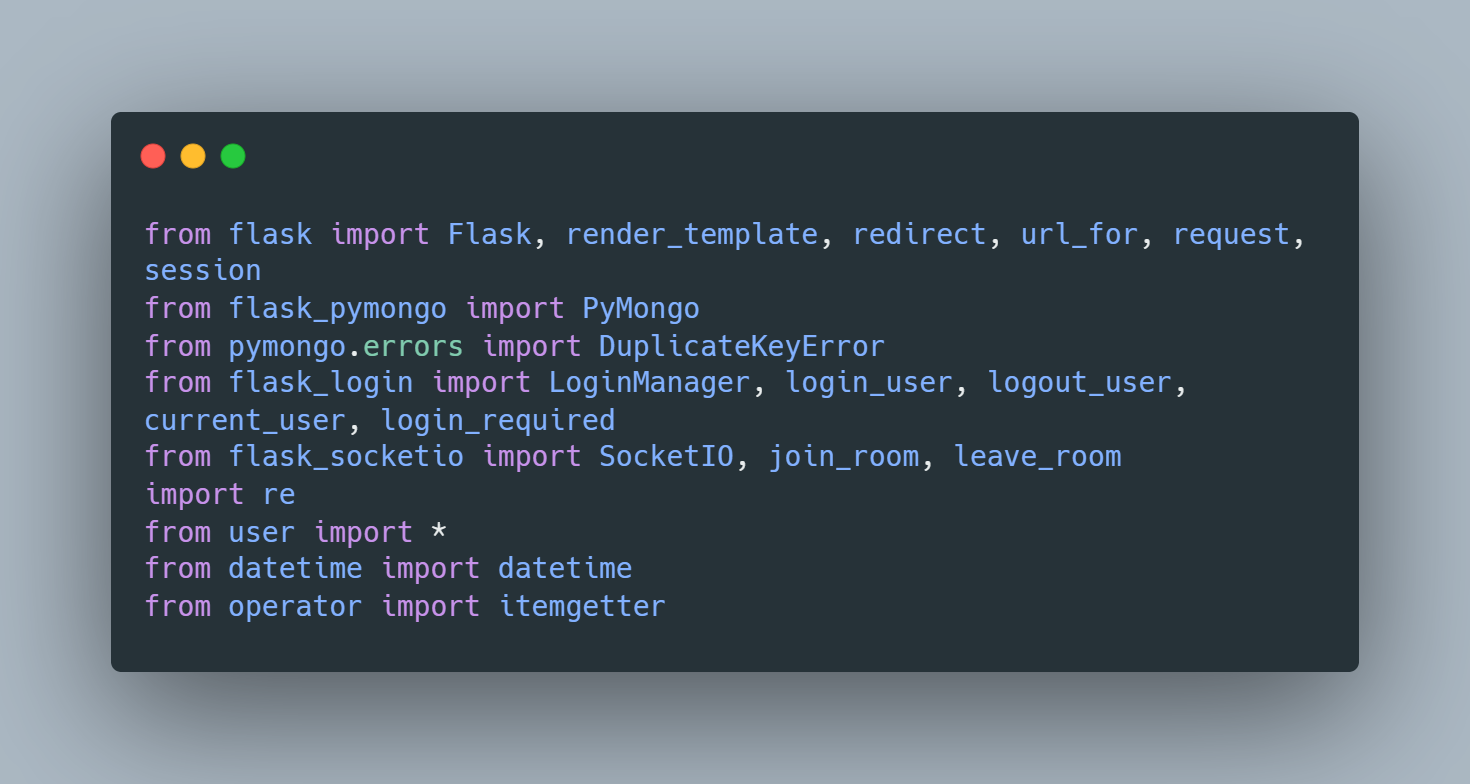
*‘save\_user()’* function is responsible for saving user information to a MongoDB database, specifically during the signup process. It takes a *‘User’* object, hashes the password, and then inserts the user's information into a MongoDB database collection named *‘user\_collection’.* User email is serving as a **unique identifier** for the user in the collection.



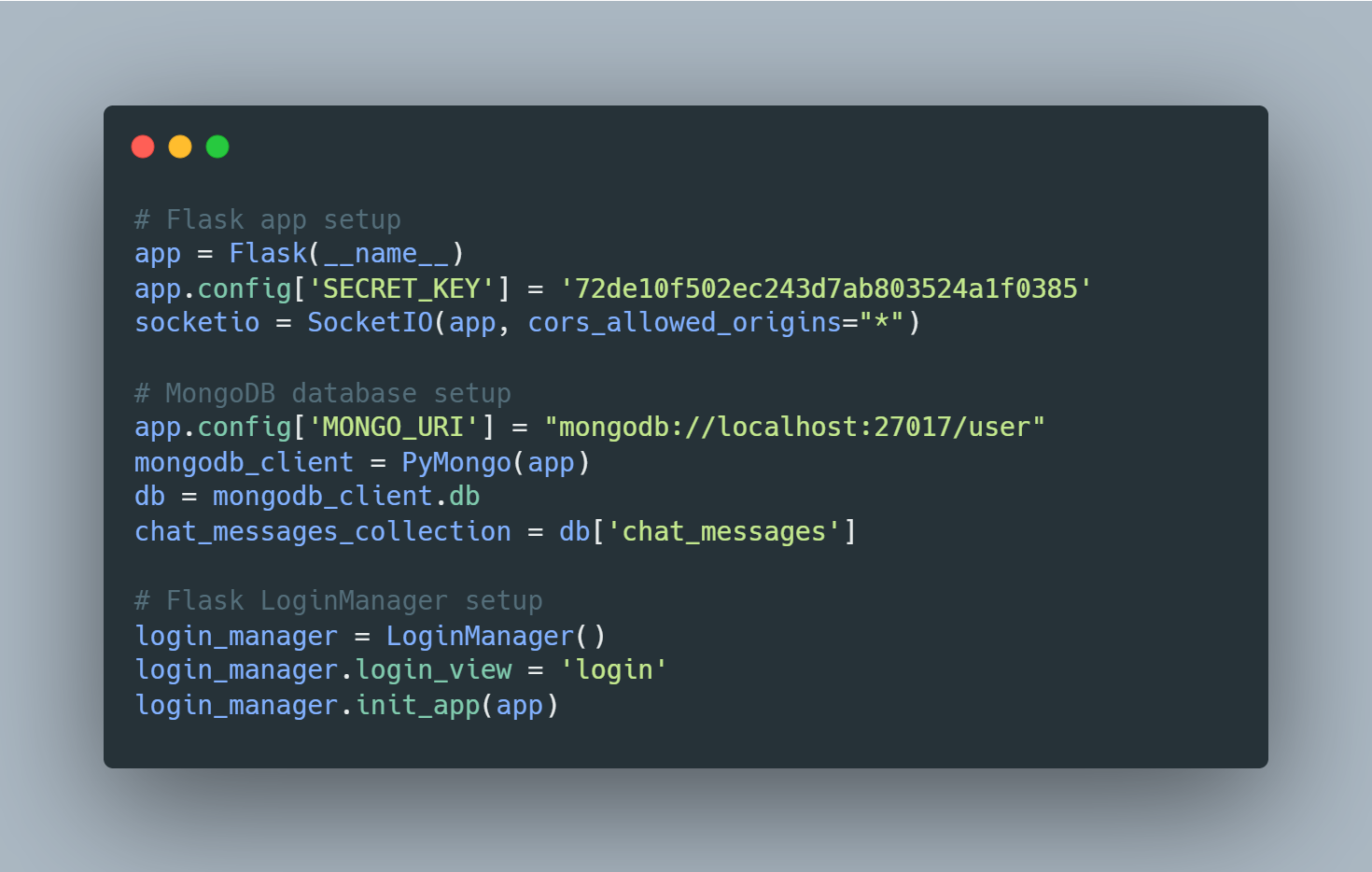
These two functions, *‘get\_user’* and *‘get\_username’*, are responsible for retrieving user information from a MongoDB database based on the provided email address.

*‘get\_user()’* function uses *‘find\_one()’* to query the *‘user\_collection’* in the MongoDB database. The query is based on the email address ('\_id': email). If a user with the specified email is found, the function creates a *‘User’* object using the retrieved data and returns it. If no user is found (*‘find\_one()’* returns None), the function returns None.

*‘get\_username()’* uses *‘find\_one()’* to query the *‘user\_collection’*. It ensures that only the *'name'* field is retrieved. If a user with the specified email is found and has a *'name'* field, it returns the username. If not found or if the user doesn't have a *'name'* field, it returns None.



* *Imports will be explained here.*



The first part creates an instance of the *‘Flask’* application and sets a secret key, which is used for session management and other security releated features. An instance of the *‘SocketIO’* class is also configured.

The second part configures the MongoDB database, *‘MONGO\_URI’* and sets it to *‘localhost’*.

The last part of the code snippet… explained here.



The decorator *‘@app.route()’* defines a route for the URL path *‘/login’*. The route accepts both *‘POST’* and *‘GET’* HTTP methods.

*‘login()’* function handles both rendering the login page and the login process.

For *‘POST’* requests (form submission):

* Retrieves the user-entered email and password from the form.
* Converts the email to lowercase for case-insensitive matching.
* Calls the *‘get\_user()’* function to retrieve user information from the database.
* Checks if the user exists and the entered password is correct using *‘usr.check\_password(\_password)’*.
* If the login is successful, the user is logged in using *‘login\_user(usr)’*, and the user's email is stored in the session. The user is then redirected to the *‘user’* endpoint.
* If the login fails, an error message is set, and the user is redirected back to the login page with the error message displayed.

For *‘GET’* requests (loading the login page):

* Checks if a user is already logged in by looking for *‘user’* in the session.
* If a user is logged in, they are redirected to the *‘user’* endpoint.
* If not logged in, the login page is rendered, and any error message (if present) is displayed.



*‘@login\_required’*  decorator ensures that only authenticated users can access this route. If a user is not authenticated, they will be redirected to the login page.

*‘user()’* function renders the user homepage if the user is logged in, otherwise it redirects the user back to the login page.



*‘logout()’* function logs out the user and redirects them back to the login page.



*‘@socketio.on(“start\_chat”)’* decorator indicates that this function will be called when the client emits a *‘start\_chat’* event.

*‘get\_username()’* function is used to get the username associated with the user's email stored in the session.

The target user's username is received from the data received in the *‘start\_chat’* event.

*‘user\_list’* is a sorted list of usernames to create a unique and consistent chat room identifier.

*‘room’* is a unique chat room identifier based on the sorted list of usernames. This ensures that the chat room identifier is consistent regardless of the order of usernames.

*‘join\_room’* function is used to make the current Socket.IO client join the specified chat room. In this case, it's the room with the unique identifier created based on the usernames.



*‘@socketio.on(“leave\_chat”)’* decorator indicates that this function will be called when the client emits a *‘leave\_chat’* event.

Client username and target username is received.

A unique room identifier is formed (It is the same one when the user starts a chat with some other client)

*‘leave\_room’* function is used to make the current Socket.IO client leave the current room.



*‘@socketio.on(“send\_message”)’* decorator indicates that this function will be called when the client emits a *‘send\_message’* event.

*‘get\_username’* function is used to get the username associated with the user's email stored in the session.

The target user's username as well as the message is received from the data received in the *‘start\_chat’* event.

Again, a sorted user list is used to form a unique room identifier.

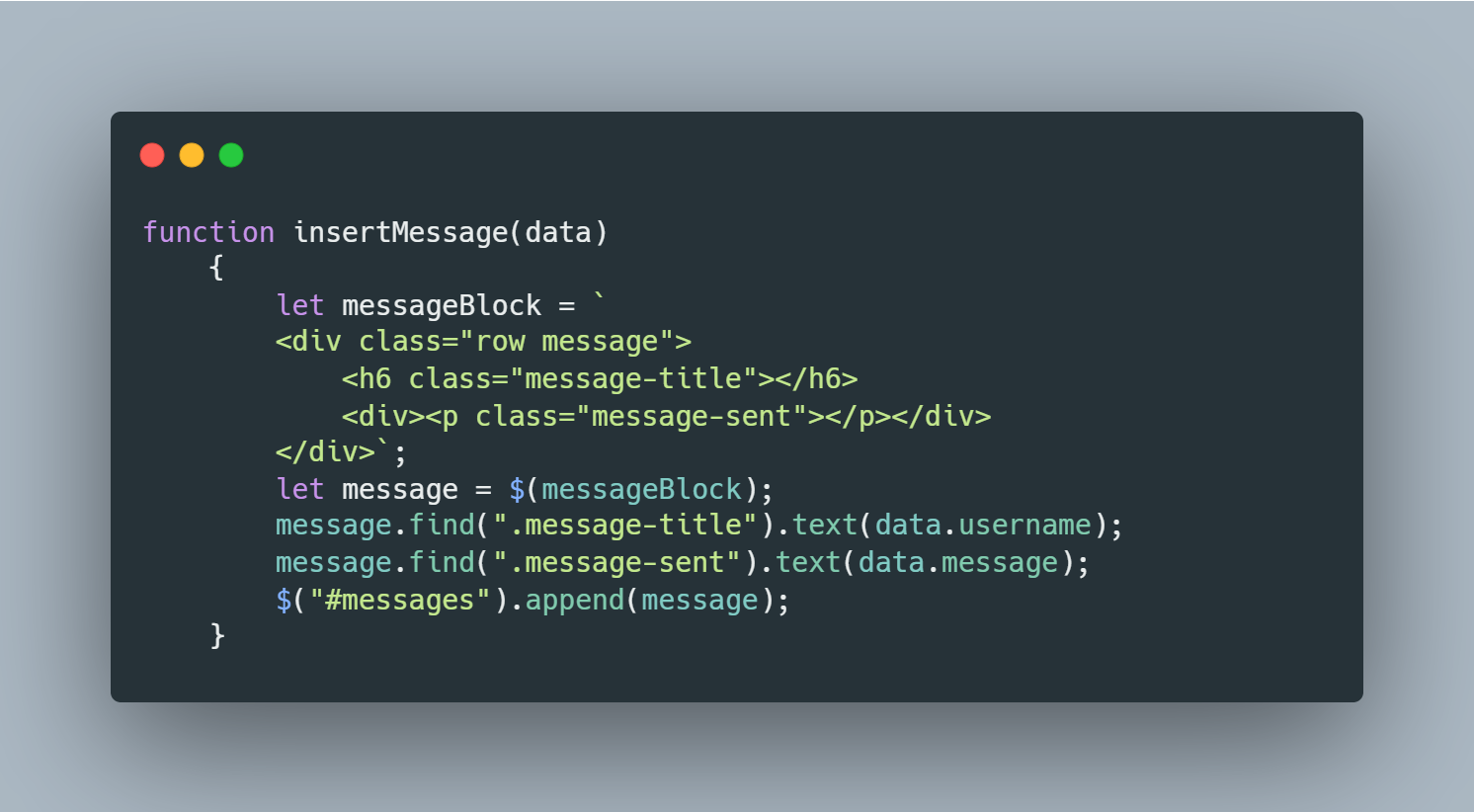
*‘socketio.emit()’* function broadcasts the message to everyone in the room with the event name being *‘chat\_message’*.

**chat.js file:**



Socket is connected to the *‘localhost:5000’*.

*‘#inputMessage’* is the id of the text input field of the chat page. It is disabled as default. When a chat is started, only then it will be enabled.



*‘insertMessage()’* function is used to insert a message received by the socketio. It formats the message and appends it to the relevant html tag.



*‘startChat()’* function enables the text input field of the chat page and emits the signal *‘start\_chat’* with the target username to indicate the chat is started.

*‘sendMessage()’* function emits the signal *‘send\_message’* with the target username and the message. After the message is emitted the text input field of the chat page is cleaned.

*‘leaveChat()’* function cleans the chat page, disables the input text field of the chat page and emits the signal *‘leave\_chat’* with the target username to the back-end to process it.



The first part of the code adds and event listener to the html elements whose ids are *‘#btnSend’* and *‘#leaveChatBtn’*.

When the send button is clicked and the text input field is not empty, *‘sendMessage()’* function is called with the target and the content of input message field.

When the leave chat button is clicked *‘leaveChat()’* function is called with the target.

The last part of the code is triggered when the *‘chat\_message’* signal is emitted. It calls the *‘insertMessage()’* method with the data received from the emitted signal.



These code snippets were used for testing purposes, Each of them symbolizes a predefined user. Each user is given a predefined html id and an event listener is attached to listen the mouse click event.

If the event is triggered, it first checks whether the user is engaged in another chat. If this is the case, the user leaves the current chat to attend to the other one. If not, user directly attends to the started chat.

References part:

For the images of the code snippets <https://carbon.now.sh/> is used. With material theme and ‘#ABB8C3’ for the backgroud color.