HW9 – Final Chat App

Chat App API Specification Document

Team Tofu

Xiongfeng Song (xs16), Hao Lin (hl76), Neil Chen (ndc1)
Ian Fan (yf18), Siyang Zhang (sz55), Zhaokang Pan (zp7)

December 2nd, 2019

1. Summary

This project involves the implementation of a chat application (chatapp) using the design principles conveyed by COMP504: Graduate Object-Oriented Programming and Design. This chatapp allows for 1-to-1 messaging between users that are members of chat rooms. Some (but not all) of the features are:

1. Users have profiles (age, location, school)
2. Users are ephemeral (their information is deleted once they close their session)
3. A user can create a chatroom with profile restrictions
4. A user can join a chat room if they meet profile restrictions as set by the owner of a chat room
5. A user can view a list of chat rooms that they have joined and a list of chat rooms that they meet the profile restrictions for
6. A user can view the list of other users in the chat room
7. A user may be present in multiple rooms
8. Rooms may have multiple users
9. A user can choose to leave one or all of their chat rooms
10. A user will be forcibly removed from a chat room if they send a message with the word "hate"
11. A user can send a message to any other user in a chat room that they are both in
12. An owner of a room can send a message to all other users in their chat room
13. When a user sends a message, they will be notified when their message is received
14. When a user leaves a room, it is indicated to other users in the same chatroom why they left the room (voluntarily left, connection closed, forced to leave)

This project's version of the chatapp is hosted on Heroku as chatapp-team-tofu.

2. Use Cases

The use cases for the chatapp can be grouped into three main categories:

1. Creation of users and chatrooms
2. Selection of chatrooms by users
3. Message passing within chatrooms

2.1 Use Cases: Creation of Users and Chatrooms

1. User creation
   When a new session is opened, a user will be asked to create a profile with the following information:
   - Username (a non-empty string with letters and numbers)
   - Age (integer between 18 and 150)
   - Location: A continent
     - Asia, Africa, Europe, North America, South America, Oceania, Antarctica

- School

2. User deletion
   If a user closes their session, then their information is removed from the chatapp. They will be removed from all of their currently joined chatrooms. If they re-open a session, then they will have to create a new profile, which will be unconnected to their previous profile.

3. Chatroom creation
   A user can create up to a maximum number of rooms. They will be the owner of these rooms.
   An owner must set the following information for each chatroom:
   - Room name (a non-empty string with letters and numbers)
   - A list of restrictions, if desired:
     - A range of ages permitted to enter the chatroom
     - A list of locations permitted to enter the chatroom (continents)
     - A list of schools permitted to enter the chatroom

4. Chatroom deletion
   An owner can delete a chatroom. This will remove all users from the chatroom, including the user.
   If an owner's session is closed, then all of the chatrooms for which they are owners will also be deleted.

2.2 Selection of Chatrooms by Users

1. Users viewing chatrooms
   A user can view a list of joined chatrooms they are currently a member of. A user can also view a list of joinable chatrooms for which they are not a member of, but that they meet the profile restrictions for.

2. Users joining a chatroom
   A user can choose to join a chatroom for which they are not a member of, but that they meet the profile restrictions for. When they join the new chatroom, then this chatroom will be removed from the list of joinable rooms, and added to the list of joined rooms for this user.

3. Users leaving a chatroom
   A user can choose to leave a chatroom that they are a member of. When they leave the chatroom, then this chatroom will be removed from the list of joined rooms, and will be added to the list of joinable rooms.

2.2 Message Passing Within Chatrooms

1. Users sending private messages
   A user can select a joined chatroom to view all other users in. A user can then select one

another user to be the current receiver. A user will then see all messages that have passed between them and the receiver during the current session. A user can then send a message to the receiver. The message status should be returned to the sender indicating whether the message was successfully received or lost.

2. Owner broadcasting
   An owner can select a joined chatroom. If they are the owner of the selected chatroom, then they can view all messages that have been broadcasted to this chatroom during the current session. An owner can then send a message to all users within the chatroom except themselves. When all receivers have received the broadcast, the owner will get a confirmation that their message has been received by all receivers.

3. Auto-ban
   If any user sends a message containing the string "hate", then they will be removed from all of their current chatrooms.

3. API Design Decisions

The game design follows the Model-View-Controller software design pattern. The model manages the chatrooms, users, and messages. The view displays the chatapp for the user. The controller accepts user input and passes it to the model. Our design is based off of the GhostBuster Team's API.

A UML diagram of our model is provided at the end of this document in section 5.

The Model, View, and Controller are discussed in more detail in sections 3.1, 3.2, and 3.3, respectively. Details for the interfaces are also provided in section 3.1.

3.1 Model

We have an abstract class called AbstractCmd, and an interface called ResponseAdapter, to be discussed in more detail.

We have an abstract class called AbstractCmd which consists of a single field (a Logger, from GhostBuster Team's API, which we do not use in our code) and a single method, execute, which allows us to execute any command extending AbstractCmd on an instance of a user. Our AbstractCmd allows us to extend many concrete commands which allow us to perform a variety of actions on the users:

1. CloseConnectionCmd is a singleton command which closes a connection by removing a user's session from the userNameMap and by removing a user's username from the userNameToSession.

2. CreateGroupCmd is a command which creates a ChatRoom by adding a Session owner and a ChatRoom to the chatRoomMap and userNameMap, respectively. It also sends a

system message that the User has created the new ChatRoom.

3. JoinGroupCmd is a command which adds a User to a ChatRoom by adding the ChatRoom to the User's list of rooms, and by adding the User to the ChatRoom's list of users. It also sends a system message that the User has joined the ChatRoom. It contains a helper method which determines whether a User is qualified to join a ChatRoom based on its restrictions.

4. LeaveAllGroupsCmd is a command which removes a User from all ChatRooms; it does this by removing the User from all ChatRooms, and then removing all ChatRooms from the User. It also sends a system message, providing a reason as to why the User left the ChatRoom.

5. LeaveGroupCmd is a command which removes a User from a single ChatRoom; it does this by removing the User from the ChatRoom, and then removing the ChatRoom from the User. If the User leaving is the only last User in the ChatRoom, then it also removes the ChatRoom from the list of ChatRooms. If other Users remain in the ChatRoom, then it also sends a system message, providing a reason as to why the User left the ChatRoom, and transfers ownership of the ChatRoom to the next-most-recently-joined User in the ChatRoom.

6. SendMessageCmd is a command which sends Message to a User by obtaining the ChatRoom from the Message class, then using the ChatRoom's notifyUser method to send the message to the User.

We have a class called Constant which stores various useful constants, mostly Strings, which improve the clarity of our code.

We have a class called Message which contains a type, sender, receiver, message text, and a groupName. This class allows us to pass messages using our design, and it acts mostly as a class to store data, and has no internal logic. It contains getters and setters for each of its fields, and an additional method to convert its fields into a String.

We have a class called GetListWrapper that allows us to more easily display the list of ChatRooms in the front-end. It consists of three lists of ChatRooms: ones which a User is the owner of, ones which a User is a member of, and all other existing ChatRooms that the User is not an owner or member of.

We have a class called RequestPayload that allows the front-end to send a user and a group to the leave and join commands in the Model (http requests).

We have an interface called ResponseAdapter which contains a single method, getJsonRepresentation, which is intended to generate a json representation of response to clients. It allows us to implement many Adapters which generate json representations of messages:

1. CreateGroupResponseAdaptor creates a json representation of the message sent when a new ChatRoom is created.

2. GetListResponseAdaptor creates a json representation of GetListWrapper that allows us to more easily display the list of ChatRooms in the front-end.

3. JoinGroupResponseAdapter creates a json representation of the message sent when a User joins a ChatRoom.

4. LeaveAllGroupsResponseAdaptor creates a json representation of the message, containing a reason, sent, when a User leaves all ChatRooms.

5. LeaveGroupResponseAdaptor creates a json representation of the message, containing the reason and groupName, sent, when a User leaves a ChatRoom.

6. MessageSendAdaptor creates a json representation of a normal message sent.

We have a class called ChatRoom which contains a groupName, owner, and various fields for determining the restrictions on the Users which can enter the ChatRooms, as well as getters and setters for these fields. It also contains a method, notifyUser, which either sends a message to all joined Users that are not the sender, or sends a message to the User that is the receiver of the message. It also contains an additional method that can convert its fields to a String.

We have a class called DataStorage that contains no logic and is only used to store three Maps: userNameMap (Session to User), userNameToSession (String username to Session), and chatRoomMap (String to ChatRoom).

We have a singleton DispatchAdapter which communicates with the model and view. It contains methods to create or close connections, update a user's profile, create ChatRooms, have users join or leave ChatRooms, send messages, and get the list of ChatRooms for each user.

We have a class called PayloadParser which creates instances of various classes based on json payloads; these classes are User, ChatRoom, Message, and RequestPayload.

We have a class called Status which is used to wrap an integer status code using json.

We have a class called User which contains a username, profile information for the User (age, location school), the Session of the User, and the list of joined ChatRooms for the User, as well as getters and setters for these fields. User implements propertyChangeListener.

3.2 View

The view displays the various elements of the chatapp, including:

- A list of rooms, with some rooms marked as **joinable** and others marked as **joined**, with options for a user to join the joinable rooms. Rooms are only displayed if the user's profile meets the requirements set by the room owner.
- The ability to expand a room's display to show a list of current users within that room.
- The ability to select another current user within a joined room, and show all message history sent between the two users during the current session. Messages change how they are displayed when they have been received by the receiving user. When users leave a chatroom (voluntarily, via session close, or via banning) this will also be broadcasted to all users within their joined rooms as a message.
- Buttons to leave chatrooms.
- Menus for chatroom owners to broadcast messages to all users.
- Menus to create profiles and new chatrooms.

The view supports user interactions with all of these displayed elements.

3.3 Controllers

The chatapp controller processes GET and POST requests and returns JSON representations of information returned by the Dispatch Adapter. It communicates with all of the clients on the web socket. The chatapp controller supports the following endpoints:

- /create is a POST request that allows users to create chat rooms.
- /join is a POST request that allows users to join chat rooms.
- /leave is a POST request that allows users to leave chat rooms.
- /list is a POST request that retrieves a list of possible chatrooms for a user.

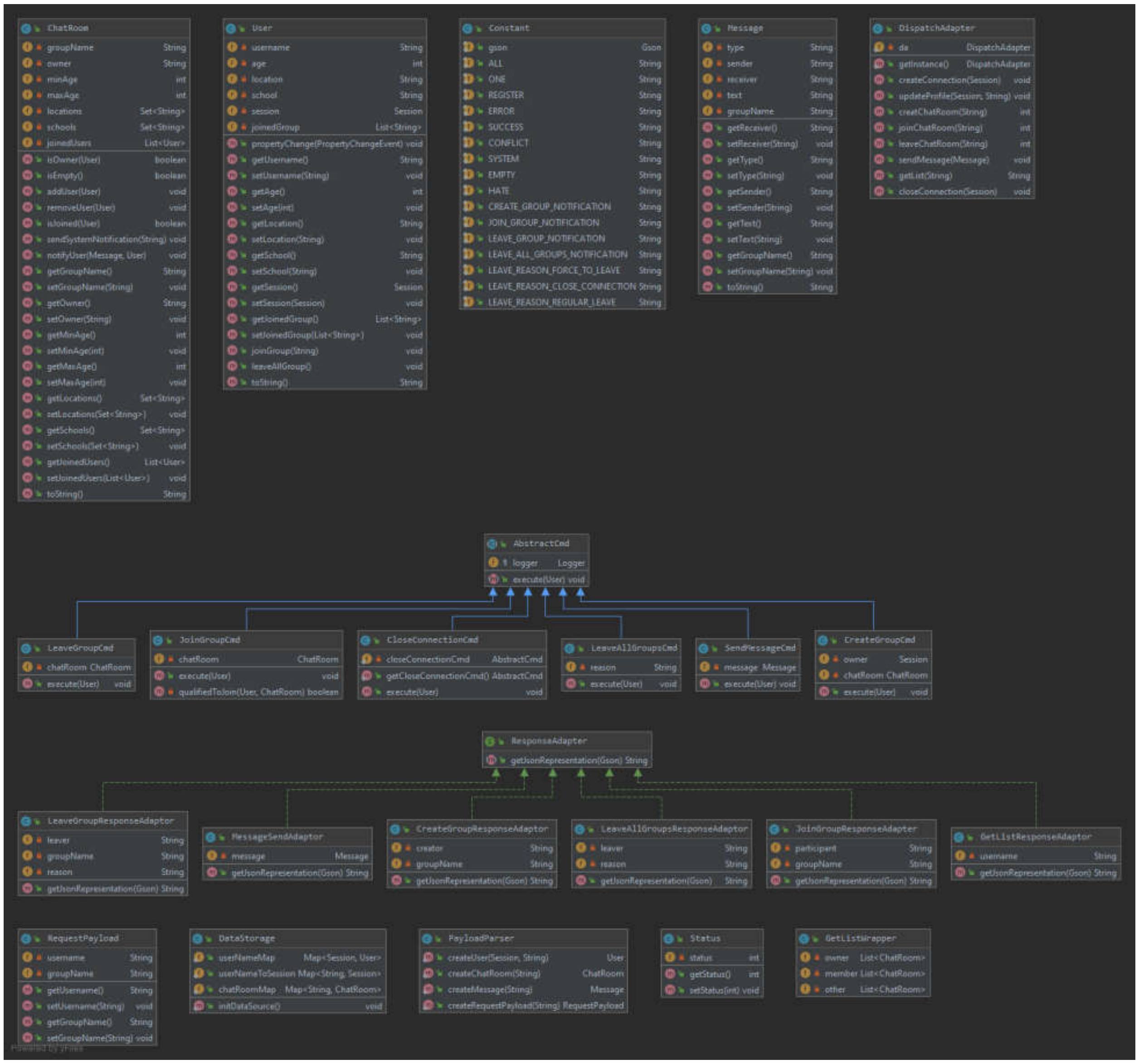
The chatapp controller also has methods to connect to Heroku and interact with the web socket controller.

The web socket controller creates a web socket for the server, can open and close user sessions, and enables message sending.

4. Additional Information

The chatapp is hosted on Heroku. There is a sufficient number of unit tests to ensure 85% code coverage for all lines of code in the model.

5. UML Diagram



6. Project Team Members

Six people contributed to this project:

xs16	Xiongfeng Song	Project Lead
hl76	Hao Lin	Tech Lead
ndc1	Neil Chen	Doc Lead
yf18	Ian Fan	Developer
sz55	Siyang Zhang	Developer
zp7	Zhaokang Pan	Developer

The guidelines for this project were provided by course COMP504: Graduate Object-Oriented Programming and Design, taught during the Fall 2019 semester by Mack Joyner at Rice University.