Quickchat
API Specification

COMP 504 chat app design project

Team Quickchat

Lead: Bill Huang, Yang Zhou, Zhen Kao
Frontend Developer: Jeff Zhou, Jianing Lou
Backend Developer: Jerry Zhang, Zhengtong Liu

Table of Contents

Chat app use cases

ID	1	Name	Create account
Description		A user creates an account with his/her information.	
Actions		1. A user can create an account with a unique username and other information such as ages, school, and interests. 2. If the account name is new and there are no format errors in the input information, the user successfully creates an account and thus can login with the account. 3. If the account name already exists, the user will see a message of a failure registration.	

ID	2	Name	Check account information
Description		Users check their account information.	
Actions		1. Users can check their account information in the chat app main page. 2. The information block will present the user's information, including the account name, the age, the school name, and the interests.	

ID	3	Name	Create a chat room
Description		A user creates a chat room.	
Actions		1. A user creates a chat room with a chat room name, a size limit of this room, and the option to make this room public or private. 2. Once the room is successfully created, users can see the new chat room in the chat room list.	

ID	4	Name	Join a chat room
Description		A user joins a chat room.	
Actions		1. A user browses the chat room list and selects the chat room s/he wants to join. 2. If s/he is able to join the room, s/he enters the room and can start a conversation in that chat room. 3. If s/he has no right to join that room, the user will receive a message saying s/he cannot join the room.	

ID	5	Name	Invite a user to the chat room
Description		A user invites another user to the chatroom s/he is in.	
Actions		1. A user invites another user into his/her current chat room. 2. The invited user will join the chat room and the other user can see his/her presence in the room.	

ID	6	Name	Leave a chat room
Description		A user leaves the chat room.	
Actions		1. A user select the option to leave the chat room. 2. The user who leaves will not see the chat room in his/her chat room list. 3. The other users in the room will not see the user's presence in that chat room.	

ID	7	Name	Kick a user
Description		The admin kicks a user out of the chat room.	
Actions		1. The admin maintains the order of a chat room and kicks out users that disobey chat room rules. 2. The user who has been kicked out cannot chat in the room anymore. 3. The other users in the room will see a user has been kicked out.	

ID	8	Name	Get user list of a chat room
Description		A user gets the user list of a chat room.	
Actions		1. A user checks who is in the same room by the user list. 2. The information block presents all users' username in the list.	

ID	9	Name	Send a message
Description		A user sends a message to other users or an individual in the same room.	
Actions		1. A user selects who should receive the message. 2. A user types the message s/he wants to send and send the message to the target users. 3. A user sends a text, an emoji, or a link.	

ID	10	Name	Receive a message
Description		A user receives a message from other users in the same room.	
Actions		1. A user sees a new message in his/her chat room that is sent from others. 2. If users are not in the targets of this message, they cannot receive the message.	

ID	11	Name	Delete a message
Description		A user deletes a message	
Actions		1. A user deletes a message and the message will disappear in the chat room.	

ID	12	Name	Edit a message
Description		A user edits a message.	
Actions		1. A user selects a message to edit and sends the updated message again. 2. All users in the chat room will see the updated message.	

ID	13	Name	User login
Description		A registered user login to the chat app.	
Actions		1. A user can login with his/her username and password. 2. If the user submit correct information, then s/he can login and see the main page of the chat room. 3. If the user submit wrong information, s/he will receive a message to indicate the login is failed.	

ID	14	Name	See all chat rooms
Description		A user checks all available chat rooms.	
Actions		1. A user sees all the chat rooms s/he can join.	

ID	15	Name	Get warning
Description		A user gets a warning.	
Actions		2. A user gets a warning because s/he used hate speech. 3. A user checks how many times of hate speech s/he has used.	

Design decision

Observer design pattern

We utilize Java's property change mechanism to realize the observer design pattern. Specifically, the pattern is a one-to-many relationship between objects. Once an object changes its state, the other objects are signaled and updated automatically. In the Quickchat app case, a user can send a message to a group chat or direct message. Then the property change mechanism allows all the other users to either receive the message or receive nothing (not in the same chat room). In all, we leverage this pattern to implement real-time chat room functionality to allow users to chat together easily.

Singleton design pattern

Some classes should be initialized once and be reused for the next time, such as UserDB, ChatroomStore, MessageStore, multiple different factories, etc. We utilize the singleton design pattern to ensure that only one such class instance exists and implement the getInstance() method to share the public singleton class object.

Command design pattern

We leverage the command design pattern to execute message sending, editing, deleting, etc., to separate message update operations with the ChatAppStore implementation because the ChatAppStore class should only send commands to update messages via property change instead of dealing with message update logic.

Strategy design pattern

The Quickchat allows users to send, edit, delete, and recall messages. We implement these message update functionality with the strategy design pattern to maintain code robustness.

Factory design pattern

To deal with different message update commands, message update strategies, we utilize the singleton factory design pattern to unify the creation of these class instances. With this design pattern, we can extend our app functionality more effortlessly in the future.

Websocket timeout handling

To handle websocket timeout issues, we utilize a tip called ‘heart-beat’ to maintain the connection of the client and server session. That is, we set a time interval in our client sides to continuously send information through the websocket in order to reset the timeout countdown. Since the purpose of the information is to keep the session alive, the server side leaves that information without further processing.

Edge case handling

1. If the admin leaves the chat room, we dismiss the chat room because there is no admin to manage the room and may cause chaos.

2. If there is only one user remaining in a chat room, we automatically let the last user leave the chat room because s/he can not chat with others.

The specification of Interfaces, abstract classes

IChatroom

Description: the interface of a Chatroom class.

Methods:

getChatroomId: Get this chat room id.

return type	int	**return**	this chat room id

getUsersInChatroom: Get all the users' username in this chat room.

return type	boolean	**return**	all the users' username in this chat room

isPrivateChatroom: Check if this chat room is private.

return type	boolean	**return**	if this chat room is private

getAdmin: Get all the admins’ usernames in the chatroom.

return type	String[]	**return**	all the admins’ usernames in the chatroom

isUserInChatroom: Check if this user exists in the chat room.

return type	boolean	**return**	if this user exists
parameters			
type	**name**	**desc**	
String	username	the username of the user to be checked	

isUserAdmin: Check if this user is the admin in the chat room.

return type	boolean	**return**	if the user is the admin
parameters			
type	**name**	**desc**	
String	username	the username of the user to be checked	

isUserInBanList: Check if this user is in the ban list.

return type	boolean	**return**	if user is in the ban list
parameters			
type	**name**	**desc**	
String	username	the username of the user to be checked	

addUserToBanList: Add this user to the ban list.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
String	username	the username of the user to be banned	

addUserToChatroom: Add this user to the chatroom.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
String	username	the username of the user to be banned	

removeUserFromChatroom: Remove this user from the chatroom.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
String	username	the username of the user to be added	

removeUserFromBanList: Remove this user from the ban list.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
String	username	the username of the user to be freed	

addAdmin: Add this user as admins.

return type	void	**return**	None
parameters			

type	name	desc
String	username	username of the user to be added as an admin

removeAdmin: Remove this user from the admin list.

return type	void	return	None
parameters			
type	name	desc	
String	username	username of the user to be removed from the admin list	

getSize: Get the size of the chatroom.

return type	int	return	chatroom size

Chatroom (implements *IChatroom*)

Fields:

type	name	description
String	chatroomName	the name of this chatroom
int	chatroomId	this chatroom id
int	chatroomSize	this chatroom size
boolean	isPrivateChatroom	if the chatroom is private
HashSet <String>	bannedUser	the banned user set
HashSet <String>	users	all users set in this chatroom
HashSet <String>	admins	all admins set in this chatroom

IChatroomStore

Description: the interface of a Chatroom store for holding chat rooms.

Methods:

getChatroom: Get this chatroom with the given chatroomId.

return type	IChatRoom	**return**	this chatroom object
parameters			
type	**name**	**desc**	
int	chatroomId	the id of this chatroom	

createChatroom: Create a chat room with this given chatroom name, chatroom size, and if the chatroom is set as a private room.

return type	IChatRoom	**return**	this chatroom object
parameters			
type	**name**	**desc**	
String	chatroomName	the name of this chatroom	
int	chatroomSize	the size of the chat room	
boolean	isPrivateChatroom	if the chatroom is private	

getAllChatroom: Get all chatrooms.

return type	IChatRoom[]	**return**	the chatroom list of all chatrooms

removeChatroom: Remove a chatroom from the chatroom store with the chatroom id.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
int	chatroomId	the id of this chatroom	

ChatroomStore (implement *IChatroomStore*)

Fields:

type	**name**	**description**
ArrayList <IChatRoom>	chatRooms	the list of all existing chat rooms.
ChatroomStore	instance	the singleton instance of the chatroom store.

Methods:

public static **getInstance**: Get the singleton instance of the chatroom store.

return type	ChatroomStore	**return**	the singleton instance of the chatroom store.

public static **checkPrivilegeChatroom**: Check if this user is an admin in this chatroom.

return type	boolean	**return**	if this user is an admin in this chatroom
parameters			
type	**name**	**desc**	
String	user	the initiator of the action	
int	chatroomId	the chatroom id to be examined	

IUpdateMessageStrategy

Description: UpdateMessageStrategy to send update info to users with a given message.

Methods:

update: Strategy for updating user data with given user and message.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	user	the user data to be updated	
IMessage	message	the message related to the user	

default **toReceiveMessage**: check if a user can receive the message

return type	boolean	**return**	boolean value if the user can receive the message
parameters			
type	**name**	**desc**	
String	username	this user to check if s/he can receive the message	

String	target	this message target (i.e. all or the specific username)
int	chatroomId	the message is sent in this chatroom id

DeleteMessageStrategy (implements *IUpdateMessageStrategy*)

Fields:

type	**name**	**description**
IUpdateMessage Strategy	instance	the instance of the strategy

Methods:

public **update**: Strategy for updating user data with given user and message.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	user	the user data to be updated	
IMessage	message	the message related to the user	

public static **getInstance**: get an instance for this strategy.

return type	IUpdateMessage Strategy	**return**	an instance of the strategy

EditMessageStrategy (implements *IUpdateMessageStrategy*)

Fields:

type	**name**	**description**
IUpdateMessage Strategy	instance	the instance of the strategy

Methods:

public **update**: Strategy for updating message history with given user and message.

return type	void	**return**	None

parameters		
type	**name**	**desc**
IUser	user	the user data to be updated
IMessage	message	the message related to the user

public static **getInstance**: get an instance for this strategy.

return type	IUpdateMessage Strategy	**return**	an instance of the strategy

NullMessageStrategy (implements *IUpdateMessageStrategy*)

Fields:

type	**name**	**description**
IUpdateMessage Strategy	instance	the instance of the strategy

Methods:

public static **getInstance**: get an instance for this strategy.

return type	IUpdateMessage Strategy	**return**	an instance of the strategy

SendMessageStrategy (implements *IUpdateMessageStrategy*)

Fields:

type	**name**	**description**
IUpdateMessage Strategy	instance	the instance of the strategy

Methods:

public **update**: Strategy for sending messages.

return type	void	**return**	None
parameters			
type	**name**	**desc**	

IUser	user	the user to be sent
IMessage	message	the message tobe sent

public static **getInstance**: get an instance for this strategy.

return type	IUpdateMessage Strategy	**return**	an instance of the strategy

---

IUpdateMessageStrategyFactory

Description: UpdateMessageStrategyFactory to generate UpdateMessageStrategy.

Methods:

make: Make a new MessageStrategy.

return type	IUpdateMessage Strategy	**return**	the IUpdateMessageStrategy instance
parameters			
type	**name**	**desc**	
String	type	type of the strategy, "edit", "send", "delete"	

UpdateMessageStrategyFactory (implements *IUpdateMessageStrategyFactory*)

Fields:

type	**name**	**description**
UpdateMessageStrategyFactory	instance	the instance of the factory

Methods:

public static **getInstance**: get an instance for this factory.

return type	UpdateMessage StrategyFactory	**return**	an instance of the factory

---

IMessageCommand

Description: MessageCommand interface to update user based on a message.

Methods:

execute: Execute a Command on user.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	user	the user to be executed the command on.	

UpdateMessageCommand (implements *IMessageCommand*)

Fields:

type	**name**	**description**
IMessage	message	the message to be updated
IUpdateMessage Strategy	strategy	the strategy we use to update

---

IMessageCommandFactory

Description: MessageCommandFactory to generate IMessageCommands.

Methods:

make: Make a IMessageCommand with a given method.

return type	IMessageCommand	**return**	object
parameters			

type	name	desc
IMessage	message	message au
String	method	method of the command, delete, send, edit

MessageCommandFactory (implements *IMessageCommandFactory*)

Fields:

type	name	description
MessageCommandFactory	instance	the instance of the factory

Methods:

public static **getInstance**: get an instance for this factory.

return type	MessageCommandFactory	return	an instance of the factory

---

IAuthenticator

Description: Authenticator class for authenticating and generating jwt for users.

Methods:

getJwtForUser: Generate JSON Web Token for a user.

return type	String	return	the jwt for the user
parameters			
type	**name**	**desc**	
IUser	user	user object	

authenticateJwt: Authenticate a user with a given jwt.

return type	IUser	return	the user object. if failed to authenticate, return null

parameters		
type	**name**	**desc**
String	jwt	JSON Web Token provided by the user

Authenticator (implements *IAuthenticator*)

Fields:

type	**name**	**description**
Authenticator	instance	the instance of the authenticator
String	issuer	who is the issuer
String	secret	the secret code
Algorithm	algorithm	the algorithm we use to encrypt
JWTVerifier	verifier	to verify the ciphertext

Methods:

public static **getInstance**: get an instance for this factory.

return type	Authenticator	**return**	an instance of the authenticator

---

IUser extends *PropertyChangeListener*

Description: User interface defines the basic behavior of a user.

Methods:

getUsername: Get username.

return type	String	**return**	username

getSchool: Get user school.

return type	String	**return**	user school

getInterests: Get user interests.

return type	ArrayList<String>	**return**	user's interests

getHateSpeechCount: Get how many times a user said 'hate speech'.

return type	int	**return**	times of hate speech

countHateSpeech: Accumulate user count of hate speech by 1.

return type	int	**return**	the new hate speech count

isUserBanedBySystem: check if a user is banned because of using 'hate speech'.

return type	boolean	**return**	true if the user is banned.

AUser (implements *IUser*)

Description: AUser abstract class defining constructor for the User.

Fields:

type	**name**	**description**
String	username	the user's name
String	password	the user's password
String	school	the user's school
int	age	the user's age
ArrayList<String>	interests	the user's interests

Methods:

public **AUser**: User constructor.

parameters		
type	**name**	**desc**
String	username	username
String	password	password
String	school	school
int	age	age
String[]	interests	interests

public **authentication**: if the password matches with the user's password.

return type	boolean	**return**	true if password correct

parameters		
type	**name**	**desc**
String	password	password

User extends *AUser*

Methods:

public **propertyChange**: This method gets called when a bound property is changed.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
PropertyChangeEvent	evt	A PropertyChangeEvent object describing the event source and the property that has changed.	

IUserDB

Description: User DB responsible for holding and maintaining all user info.

Methods:

userExists: Check for if username exists in user db.

return type	boolean	**return**	true if user exists, false otherwise
parameters			
type	**name**	**desc**	
String	username	username string	

authenticateUser: Authenticate a user based on given username and password.

return type	IUser	**return**	if the user exists, return the IUser instance, otherwise, return null.
parameters			
type	**name**	**desc**	

String	username	username username for the user
String	password	password password for the user

addNewUser: Add a user to the user db. Username should be unique.

return type	IUser	**return**	the IUser instance if success, null if failure (username already exists)
parameters			
type	**name**	**desc**	
IUser	user	the IUser instance	

getUserInfo: Get user info based on its username.

return type	IUser	**return**	user obj
parameters			
type	**name**	**desc**	
String	username	username user's username	

getSessionByUser: Get the session of the user. If user is logged in, return null

return type	Session	**return**	websocket session corresponding to the user. return null if user is not logged in
parameters			
type	**name**	**desc**	
IUser	user	user instance	

registerUserSession: Register logged in user session.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	user	user instance	
Session	session	session of the user	

removeUserSession: Remove a session.

return type	void	**return**	None

parameters		
type	**name**	**desc**
Session	session	session instance

isUserLoggedIn: Check if a user is logged in by checking if we have its session.

return type	boolean	**return**	true if we have the user's session
parameters			
type	**name**	**desc**	
String	username	username of the user	

UserDB (implements *IUserDB*)

Fields:

type	**name**	**description**
ArrayList<IUser>	users	all users in the database
Map<Session, IUser>	sessionUserMap	a hash map stores each session and its users
UserDB	instance	the instance of the database

Methods:

public static **getInstance**: get an instance for this database.

return type	UserDB	**return**	an instance of the database

IChatAppStore

Description: IChatAppStore is responsible for storing all the objects in the ChatApp. Objects include user, chat room and messages.

Methods:

loginUser: Login user returns the jwt for the user.

return type	String	**return**	"" if login fails, jwt if login success

parameters		
type	**name**	**desc**
String	username	username
String	password	password

registerNewUser: register for a user with an new account.

return type	JsonResponse	**return**	json response
parameters			
type	**name**	**desc**	
String	username	username for the user	
String	password	password for the user	
String	password	school info of the user	
int	school	interests of the user	
String[]	age	the age of the user	

getUser: Get the user info of the specific user.

return type	JsonResponse	**return**	json response
parameters			
type	**name**	**desc**	
String	username	username for the user whose user info to be retrieved	
IUser	initiator	initiator the user who called this method	

createChatroom: create a chat room.

return type	JsonResponse	**return**	json response
parameters			
type	**name**	**desc**	
String	chatroomName	name for the chatroom	
IUser	initiator	the user who called this method	
boolean	privateChatroom	is this chatroom private.	
int	chatroomSize	size of the chatroom	

joinUserIntoChatroom: join a user into a chatroom.

return type	JsonResponse	**return**	json response
parameters			
type	**name**	**desc**	
String	username	username for the user to join the chat room	
IUser	initiator	the user who called this method	
int	chatroomId	chat room id	

removeUserFromChatroom: Remove a user from the chatroom.

return type	JsonResponse	**return**	json response
parameters			
type	**name**	**desc**	
String	username	username for the user to join the chat room	
IUser	initiator	the user who called this method	
int	chatroomId	chat room id	

getUserInChatroom: Get all users in a chat room.

return type	JsonResponse	**return**	json response
parameters			
type	**name**	**desc**	
IUser	initiator	the user who called this method	
int	chatroomId	chat room id	

sendMessageInChatRoom: Send a message in the chat room.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	initiator	the user who called this method	
int	chatroomId	chat room id	
String	message	message content	

String	target	target user's username, 'all' for all user in the chatroom

deleteMessage: Delete a message given the message id.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	initiator	the user who called this method	
int	messageId	message id of the message	

editMessage: Edit an existing message.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
IUser	initiator	the user who called this method	
int	messageId	message id of the message	
String	newContent	new content for the message	

ChatAppStore (implements *IChatAppStore*)

Fields:

type	**name**	**description**
IChatroomStore	chatroomStore	an instance of chatroomStore
UserDB	userDB	an instance of userDB
ChatAppStore	instance	the instance of the store
MessageStore	messageStore	an instance of messageStore
Gson	gson	Gson

Methods:

public static **getInstance**: get an instance for this store.

return type	ChatAppStore	**return**	an instance of the store

public **loginUser**: Login user returns the jwt for the user

return type	String	**return**	"" if login fails, jwt for the user if login success
parameters			
type	**name**	**desc**	
String	username	User's username	
String	password	password	

IMessage

Description: Message interface defining the message structure.

Methods:

getContent: Get the content of a message.

return type	String	**return**	the message content

setContent: Edit content of message.

return type	void	**return**	Nonde
parameters			
type	**name**	**desc**	
String	content	content new content of message	

getId: Get the message id.

return type	int	**return**	message id

getTarget: Get the target receiver for the message.

return type	String	**return**	return "all" for sending the message to all the users in the chatroom, otherwise, return a username.

getChatroomId: Get the chat room id which the message is in.

return type	int	**return**	the chatroom id

getSender: Get the sender of the message.

return type	String	**return**	the sender username of this message

Message (implements *IMessage*)

Fields:

type	**name**	**description**
String	sender	the message's sender
String	target	the target to send the message
int	id	the message's id
int	chatroomId	the chat room which contains the message

public **Message**: Message constructor.

parameters		
type	**name**	**desc**
String	sender	sender of message
String	target	target of message
int	id	the message's id
int	chatroomId	the chat room which contains the message
String	content	content of message

HttpRequestAdapter

Description: Adapter for REST request handling.

Fields:

type	**name**	**description**
HttpRequestAdapter	instance	the instance of the adapter
IChatAppStore	store	an instance of ChatAppStore

Methods:

public static **getInstance**: get an instance for this adapter.

return type	HttpRequestAdapter	**return**	an instance of the adapter

public **registerUser**: create a new user.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
JsonObject	request	http request	

public **getUserInfo**: check if initiator is authenticated and then fetch the info for him.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
String	userToken	the token of the user who want to look up a user	
String	username	the user's name of which we want to look for	

public **createChatRoom**: create a chat room.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
String	userCookie	cookie for the user	
JsonObject	request	client side request	

public **addUserToChatRoom**: Add user to a chat room.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
String	userCookie	cookie for the user	
JsonObject	request	client side request	

public **removeUserFromChatroom**: Remove this user from this chat room.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
String	userCookie	cookie for the user	
int	chatroomId	chatroom id to remove from	
String	username	the username to be removed	

public **getAllChatRooms**: Get all chat room list of this user.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
String	userCookie	cookie for the user	

public **getUserInChatroom**: Get all users in this chat room.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
String	userCookie	cookie for the user	
String	chatroomIdStr	string representation of the chatroom id	

public **logInRegisteredUser**: **""** if login fails; jwt for the user if login success.

return type	JsonResponse	**return**	JsonResponse
parameters			
type	**name**	**desc**	
JsonObject	request	httprequest	

WebSocketAdapter

Description: WebSocket Adapter to handle communication through web socket.

Fields:

type	**name**	**description**
IChatAppStore	store	the instance of the chat app store
JsonParser	jsonParser	the json parser
Gson	gson	Gson

Methods:

public **onClose**: Close the user's session and remove that session from the store.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
Session	session	The user whose session is closed.	

public **onMessage**: Processing the client side message request, including send, delete, etc.

return type	void	**return**	None
parameters			
type	**name**	**desc**	
Session	session	The session user sends the message.	
String	message	The message to be processed.	

API SPEC

Template

RESTful API template

```
{“Path”: all path should follow the standard where a noun is followed by a verb
to describe user case (except for GET where parameter will be passed in URL)

details:{
    Method (get, post):

        “description”:,
        {"Request":
            {“Payload”:{
                {parameters (all parameters are REQUIRED):}
            }}
        },
        {“Response”:
            {“code”:,
            “success”:
            “description”:
            “payload”:{
            }},
            “cookie”:String,
            {“code”:,
            “success”:
            “description”:
            “payload”:{
            “cookie”:String
            }}
        }
    }
}
```

WebSocket data template:

```
{
    “type”: String,

    “payload”: {

    }
}
```

USE CASE & API

1. Create Account

```
{“Path”: “/user/create”,
    details:{
        {"Request":{
            POST:
                “description”:”Create new user”
                “Payload”:{
                    “username”: String,
                    “password”: String,
                    “age”: int,
                    “school”: String,
                    “interest”: []String
                }
        }},
        {Response:
                    {“code”: 201,
                    “success”: true,
                    “description”: “Successfully created new user”,
                    “payload”: {
                        “cookie”: String
                    }
                    },
                    {“code”: 409 ,
                    “success”:false,
                    “description”: ”Invalid username. (Already exists or Does not meet the requirement)”,
                    “payload”: {
                    }
                    }
        }
    }
}
```

For requests in 2-11, and 12, all requests should contain a cookie.

2. Present user’s info:

```
{“Path”: “/user/{username}”
    details:{
        GET:
            “description”:”get user info”,
            {"Request":{
                “Payload”:{
                }
                “cookie”: String,
            }},
            {“Response”:
                {“code”:200,
                “success”: true
                “description”:”Successfully retrieve user info.”
                “payload”:{
                    “username”: String,
                    “age”: int,
                    “school”: String,
                    “interest”: []String,
                    “chatrooms”:[]json(Chat room json)
                }},
                {“code”:403,
                “success”:false
                “description”:”Not authorized”
                “payload”:{
                }},
                {“code”:404,
                “success”:false
                “description”:”User not found”
                “payload”:{
                }}
            }
    }
}
```

3. Create Chat Room:

```
{“Path”: “/chatroom/create”
    details:{
        post:
            “description”:”Create a new chat room”,
            {"Request":{
                “Payload”:{
                    "size": int,
                    "chatroom_name": String,
                    "is_private": bool
                }
                “cookie”: String,
            }},
            {“Response”:
                {“code”:200,
                “success”:true
                “description”:”Create chat room success”
                “payload”:{
                    “chatroom_id”: int
                }},
                {“code”:403,
                “success”:false
                “description”:”Unauthorized, user is not authorized to create chat room”
                “payload”:{
                }},
                {“code”:400,
                “success”:false
                “description”:”Chat room size is not valid”
                “payload”:{
                }},
            }
        }
    }
}
```

4. Join chat room

5. Invite user to chat room:

The two use case (4&5) may be merged into one as "add user into chatroom," where initiater and added user can be the same

```
{“Path”: “/chatroom/join”
    details:{
        POST:
            “description”: “User join a chat room”,
            {"Request":{
                “Payload”:{
                    “Invitee(username)”: String,
                    “chatroom_id”:int,
                },
                “cookie”:String,
            }},
            {“Response”:
                {“code”: 200,
                “success”: true
                “description”: “successfully joined chatroom”
                “payload”:{
                }},
                {“code”: 401,
                “success”: false
                “description”: “user is not logged in / Unrecognized user”
                “payload”:{
                }},
                {“code”: 403,
                “success”: false
                “description”: “chatroom has private access”
                “payload”:{
                }},
                {“code”: 404,
                “success”: false
                “description”: “chatroom not found”
                “payload”:{
                }},
                {“code”: 409,
                “success”: false
                “description”: “chatroom had reached maximum size”
                “payload”:{
                }}
            }
    }
}
```

6. Leave Chat room (SAME AS 7):

7. Delete/Ban User in chatroom:

The two use case (6&7) may be merged into one as "user leave chatroom," where initiater and removed user may or may not be the same

```
{“Path”: “/chatroom/{chatroom_id}/member/{username}”
details:{
    DELETE:
    “description”: “Remove a user from a chat room”,
    {"Request":{
        “Payload”:{
        },
        “cookie”:String
    }},
    {“Response”:

      {“code”: 200,
        “success”: true
        “description”: “successfully removed user”
        “payload”:{
        }},

      {“code”: 401,
        “success”: false
        “description”: “user is not logged in / Unrecognized user”
        “payload”:{
        }},

      {“code”: 403,
        “success”: false
        “description”: “user does not have enough privilege”
        “payload”:{
        }},

      {“code”: 404,
        “success”: false
        “description”: “chatroom not found / user is not in given room”
        “payload”:{
        }}
      }
    }
  }
}
```

8. View exist user in a chatroom:

```
{“Path”:/chatroom/{chatroom_id}/members
  details:{
    GET:
    “description”:”Get all members in a chat room”,
    {"Request":{
        “Payload”:{}
        “cookie”: String,
    }},
    {“Response”:
      {“code”:200,
      “success”true:
      “description”:”Successfully find all members in a chat room”
      “payload”:{
        []User{
          “username”: String,
          “school”: String,
          “age”: int,
          “interests”:[]String
        }
      }},

      {“code”: 403,
        “success”:false
        “description”:”Unauthorized: user is not authorized to view all members in a chat room.”
        “payload”:{}},

      {“code”: 404,
        “success”:false
        “description”:”Chat room not found”
        “payload”:{}}
      }
  }
}
```

9. Send Message

Client -> Server:

```
{
  “type”:”send_message”,
  “cookie”: String,
  “payload”: {
    chatroom_id: Int,
    target: “all” || String( username for direct message)
    content: String
  }
}
```

10. Receive Message:

Server -> Client:

```
{
“type”:”new_message”,
“payload”: {
  chatroom_id: Int,
  target: “all” || String( username for direct message)
  content: String,
  sender: String,
  message_id: Int
}
}
```

11. Delete Message

Client -> Server:

```
{
  “type”:”delete_message”,
  “cookie”: String,
  “payload”: {
    message_id: Int
  }
}
```

Server -> Client:

```
{
  “type”:”delete_message”,
  “payload”: {
    message_id: Int
  }
}
```

12.Login Registered User:

```
    {“Path”: “/user/login”,
        details:{
            POST:
                “description”:”Login Registered User”
                {"Request":{
                    “Payload”:{
                        “username”: String,
                        “password”: String,
                    }
                }},
                {Response:
                        {“code”: 200,
                        “success”: true,
                        “description”: “Successfully Logged in”,
                        “payload”: {
                            “cookie”: String
                        }
                        },
                        {“code”: 403 ,
                        “success”:false,
                        “description”: ”Invalid username or password”,
                        “payload”: {
                        }
                        }
                }
        }
    }
```

13.Edit Message:

Client -> Server:

```
{
  “type”:”edit_message”,
  “cookie”: String,
  “payload”: {
    content: String,
    message_id: Int
  }
}
```

Server -> Client:

```
{
  “type”:”edit_message”,
  “payload”: {
    content: String,
    message_id: Int
  }
}
```

14. System Warning and Notification

Server -> Client

```
{
    "type":"notification",
    "payload": {
        "level": "warning" | "info",
        "message": String
    }
}
```

15. Show all chat rooms

```
{“Path”: “/chatroom”
    details:{
        GET:
            “description”:”get all chatrooms”,
            {"Request":{
                “Payload”:{
                }
                “cookie”: String,
            }},
            {“Response”:
                {“code”:200,
                “success”: true
                “description”:”Successfully retrieve user info.”
                “payload”:{
                    []Chatroom: {
                        chatroom_id: Int,
                        chatroom_name: string,
                        chatroom_size: Int,
                        is_private: bool,
                        admins: []String,
                        users: []String

                    }
                }},
                {“code”:401,
                “success”:false
                “description”:”user is not logged in / Unrecognized user”
                “payload”:{
                }}
            }
    }
}
```

16. Register the current session when user login

Client -> Server

```
{
    "type":"register_session",
    "cookie": String,
}
```