Homework 4: WebSockets

## Learning Objective 1: WebSocket Functions

Write the following 3 functions in util.websockets. These will be helper functions that you'll use throughout the rest of the assignment.

**Compute Accept**

Write a function named **compute\_accept** that takes a WebSocket key as a parameter (As a string) and returns the correct accept (As a string) according to the WebSocket handshake.

The output must be character for character exactly what is expected. Make sure you don't have any extra characters in your output (Not even white space).

You may use libraries to compute the SHA1 hash and Base64 encoding.

**Parsing Frames**

Write a function named **parse\_ws\_frame** that takes bytes as a parameter that represents the bytes of a WebSocket frame and parses all the values of the frame. The function returns an object containing the following fields (You have some freedom in how you design the class for this object as long as it has the required fields).

* fin\_bit
  + An int with the value of the fin bit (Either 1 or 0)
* opcode
  + An int with the value of the opcode (eg. if the op code is bx1000, this field stores 8 as an int)
* payload\_length
  + The payload length as an int. Your function must handle all 3 payload length modes
* payload
  + The unmasked bytes of the payload

**Creating Frames**

Write a function named **generate\_ws\_frame** that takes bytes as a parameter and returns a properly formatted WebSocket frame (As bytes) with the input bytes as its payload. Use a fin bit of 1, an op code of bx0001 for text, and no mask. You need to handle all 3 payload length modes.

### Testing Procedure

1. Running the code using Python (Not Docker), run tests on the functions defined above to verify that it returns the correct output on all inputs
2. This objective will be autograded in Autolab and you will receive feedback as soon as you submit. You are allowed as many submissions as you'd like in order to complete this objective (All other objectives are manually graded after the deadline)

## Learning Objective 2: Live Chat With WebSockets

You can make the following simplifying assumptions when working with WebSockets in this assignment:

* The 3 reserved bits will always be 0
* You can ignore any frames with an opcode that is not bx0001, or bx1000, or bx0000
* Additional WebSocket headers are compatible with what we discussed in class (ie. You don’t have to check the Sec-WebSocket-Version header)

In this objective, you will modify the chat feature to use WebSockets instead of HTTP, AJAX, and polling. This will make the chat "live" in that each user will receive new messages immediately [minus network delays] instead of waiting for the next poll request.

To start this objective, you should change the "ws" variable in functions.js to true. This will change the frontend to use WebSockets when sending and receiving chat messages. It will still use your "/chat-messages" path to retrieve old messages when the page loads, but will receive new messages over a WebSocket connection.

#### WebSocket Handshake

Implement the handshake of the WebSocket protocol at the path "/websocket".

| **const socket** = **new *WebSocket***(**'ws://'** + ***window***.**location**.**host** + **'/websocket'**); |
| --- |

This line, which is in the provided JavaScript, will make a GET request to the path "/websocket" and attempt to upgrade the TCP socket to a WebSocket connection.

During this connection process, you must authenticate the user based on their authentication token in their cookies. This is your only chance to authenticate the WebSocket connection so it must be done during this handshake. For the duration of the connection, you can assume any WS frame sent over the connection is authenticated as this user. If they cannot be authenticated, you should still proceed with the connection as a guest user.

#### WebSocket Frames

The provided JavaScript and HTML will send WebSocket frames containing chat messages when a user submits text to the chat. The payload of each frame will be a JSON string in the format:

{

'messageType': 'chatMessage',

'message': message\_submitted\_by\_user

}

By adding a messageType, you can handle more than just chat messages using your WebSockets. You should check the message type of any WebSocket frame to determine how to handle the payload.

For this objective, you will parse WebSocket frames that are received from any open WebSocket connection, parse the bits of the frame to read the payload, then send a WebSocket frame to all connected WebSocket clients, including the sender, containing the new message. The message sent by your server must be in the format:

{

'messageType': 'chatMessage',

'username': username\_of\_the\_sender,

'message': html\_escaped\_message\_submitted\_by\_user,

'id': id\_of\_the\_message

}

For the username, use the username that you authenticated during the WS handshake. If they are not authenticated, set their username to "Guest".

**Disconnections**: When a frame with an opcode of bx1000 (disconnect) is received, the connection should be severed and removed from any storage on your server. When new messages are received, any disconnected WebSocket connections should not be sent the new message (ie. Disconnects must be handled gracefully).

**~~Chat History~~**~~: A GET request is sent to /chat-history when the page loads to request this content and render it to the page. Once you build the /chat-history endpoint properly, you will see the message history appear when the page loads. The format is the same as your GET /chat-messages path, but the starter code uses the /chat-history. You can either add this endpoint, or modify the front end to use the /chat-messages endpoint. Either way, all users should be able to see the history of the chat when the page loads.~~

**Buffering**: Your server must be able to handle WebSocket frames of any size. Specifically, you must implement the following functionality:

* Handle messages of arbitrary size. This includes both messages where the 7-bit payload is set to 126 and 127. We will test with frames larger than 65536 bytes to ensure you handle all three cases. You will need to buffer your WebSocket frames. You should read from the TCP connection once, parse only the headers of the frame, parse the payload length, then check if you've read that many bytes of payload. If not, you must buffer before de-masking and reading the payload.
* You must check the FIN bit and handle messages that are sent over multiple frames. We will test with payloads > 131000, as chrome sends messages over this size in multiple frames
* Your server must also handle multiple large messages sent back-to-back. If another frame is sent while you are buffering, your last read from the TCP socket may contain the beginning of the next frame. Ensure that you are properly parsing these frames even when one read from the socket contains parts of multiple frames

**Router Note**: If you are using the router from HW2, you may want to add a second parameter to route\_request that is a reference to your handler and pass this handler to your WebSocket function (The one you added to the route). This way, your function can enter a loop after establishing a WebSocket connection while also interacting with the handler directly (Which controls access to the TCP socket including the "recv", and "sendall" methods).

**Database**: Use your database to store all of the chat history for your app. This will allow the chat history to persist after a server restart.

**Security**: Don't forget to escape the HTML in your users' comments.

**Grading Note**: It is acceptable if your image/video upload form no longer works properly after switching to WebSockets since it was designed to work with the polling endpoint. If this feature is broken, it will not affect your HW4 grade. However, if you do ensure this feature works, it will no longer have the issue of reloading the videos on every poll response.

### Testing Procedure

1. Start your server using docker compose up
2. Open a browser and navigate to http://localhost:8080/
3. Open the network tab of the browser console (refresh the page if necessary)
4. Verify that there is a successful WebSocket connection with a response code of 101
5. Open a second browser (Use Chrome and Firefox) and repeat steps 2-4 to verify that the server supports multiple simultaneous WebSocket connections
6. Register accounts and login in both browsers
7. Enter several chat messages with < 50 characters in each browser
   1. Verify that each user can see both their own messages, and messages sent by other users in real time
   2. Verify that the correct usernames show up with each message
   3. Verify that the messages were sent using WebSockets by checking the messages tab of the 101 request
8. Close one browser, then send a message with < 50 characters in the other
   1. Verify that the app is still functional and that no errors appeared in the docker compose output
9. Restart the server using docker compose restart
10. Open a new *incognito window* and navigate to http://localhost:8080/
11. Verify that all messages are visible on the page
12. Refresh the tabs/browsers and verify the chat history appears as expected
13. Send a message with < 50 characters in the new *incognito window*
    1. Verify that it appears in all tabs/browsers with a username of "Guest"
14. Send a message with < 50 characters from the first browser/tab
    1. Verify that it appears in all tabs/browsers with the authenticated username (ie. Auth tokens persist through a server restart)
15. Buffering:
    1. Enter chat messages of varying size. Ensure that at least one message has a payload length >=126 but <=65536, at least one has payload length >65536 but <131000, and another [from Chrome] with payload length > 131000
    2. Verify that each user can see both their own messages and messages sent by other users in real time
    3. Verify that the messages have been sent/received using a WebSocket connection
    4. Modify the JavaScript on at least 1 tab to send messages **5 times** when the user sends a message (These messages will be sent back-to-back such that the messages may be partially read when reading the previous message from the TCP socket)
    5. Send several messages from the modified tab and verify that all users see all the sent messages 5 times
16. **Security**: Verify that submitted HTML displays as text and is not rendered as HTML

## Learning Objective 3: User List

In the area of the page with the login/registration forms, add a live list of all users who are both logged in and connected via WebSocket. To accomplish this, you should design a new messageType for your WebSocket messages that will update this list for all users whenever an authenticated user connects or disconnects.

You have a fair amount of flexibility in how you approach this objective on both the back end and front end, as well as the communication between the two.

Note: It is intended that you use WebSockets to update the user list, but you are allowed to use polling and AJAX to send the latest list to each user.

### Testing Procedure

1. Start your server using docker compose up
2. Open a browser and navigate to http://localhost:8080/
3. Verify that no users appear in the user list, or there is no user list at all
4. Open a second browser (Use Chrome and Firefox), register an account, and login
   1. Verify that this user appears in the user list on both browsers
5. In the first browser, register and login with a different account
   1. Verify that this username appears in the user list on both browsers (It is ok if their own username doesn't show in a users browser and it only shows the other user)
6. Close one of the browsers
   1. Verify that the username of the closed browser was removed from the user list of the remaining browser

## Application Objective 1: Direct Messages

Add a DM feature to your app. In the user list, add a clear way to send a DM to each user. You also need to display all DMs that were sent to a user as well as making it clear who sent the DM. DM history must persist throughout a restart.

Note: You will want to add at least one more messageType and design the structure of this message type. You'll also need an endpoint to retrieve the DM history for each logged in user.

Note: It's ok if past DMs don't show up for users who are not currently in the user list (ie. You only need to show the DMs for users who are currently logged in to the app).

**Security**: DM messages must be kept private. Never send the content of a DM to any user except the sender and recipient of the DM.

**Security**: Don't forget to escape the HTML in your users' comments.

### Testing Procedure

1. Start your server using docker compose up
2. Open 3 browsers, navigate to http://localhost:8080/, register 3 accounts, and login with a different account on each browser
3. In the user list, find a way to send a DM to other users
4. Send and receive DMs for all users (Send at least 1 message for all 6 combinations of sender/recipient)
   1. Verify that all DMs appear only to the sender and recipient of that message, that the DMs appear in real-time, and that the front end makes it clear who sent/received each DM
5. Open the network tab in one of the browsers and view the messages of the WebSocket connection
   1. Send a DM to this user and verify that it was sent over the WS connection
   2. **Security**: Send a DM between the other two users and verify that it was not sent to this user over the WS connection
6. Refresh the page with the network tab still open
   1. Verify that all DMs still appear as expected
   2. **Security**: Find the /chat-history or /chat-messages response and verify that only DMs to/from this user are sent to them
7. **Security**: Verify that submitted HTML displays as text and is not rendered as HTML

## 

## Application Objective 2: Video Chat DMs Over WebRTC

Add features to your server so it can function as a WebRTC signaling server for your users to start 1-on-1 chats with each other. This signaling will occur over your WebSocket connection. To accomplish this, add a button next to each user in the user list that will start a video chat with that user. When this button is pressed, it should start a WebRTC video chat between those two users. You may assume that both users have started their own video before this button is pressed.

Example functionality for WebRTC is provided on the front end of the starter code. You can use this as an example of how to establish video chat over WebRTC, but it will not work exactly for this objective. Specifically, you'll need to add a way to specify the username of the user with whom you'd like to chat.

In the starter code, there are three different types of messages used to establish WebRTC connections:

{'messageType': 'webRTC-offer', 'offer': offer}

{'messageType': 'webRTC-answer', 'answer': answer}

{'messageType': 'webRTC-candidate', 'candidate': candidate}

Where offer, answer, and candidate are all generated by the WebRTC code built into your browser. Your task is to forward these messages to the other user via their WebSocket connection. You must keep track of the WebSocket connections to route these messages to the appropriate WS connection based on the username.

Your server must be able to handle multiple simultaneous DM chats, but each user can only be in one DM chat at a given time.

Remember, your server is merely acting as a means for the 2 clients to communicate while they establish a peer-to-peer connection. The content of the messages you handle do not need to be parsed or processed. Your task is to extract the payload of these WebSocket messages, verify that they are not chat messages (and are WebRTC messages), then send the payload to the other WebSocket connection. The clients will do the rest through the front end.

### Testing Procedure

1. Start your server using docker compose up
2. Open 4 browsers, navigate to http://localhost:8080/, register 4 accounts, and login with a different account on each browser
3. In each browser:
   1. Allow the camera/mic to be accessed
   2. If the local video has not started on the page load, click the "Start My Video" button
4. In the user list, find a way to start a DM video chat with other users
5. In one of the 4 browsers, start a DM video chat with one of the other 3 users
   1. Verify that 2 videos are show in both windows, but only 1 video in the remaining 2 browsers
   2. Both videos will be showing the same feed, but the remote video should be slightly behind the local video which is evidence that there is a video streaming connection between the two browsers. You'll also hear some horrible sounding audio with feedback if you didn't mute it. Don't worry, that means it's working :)
6. In one of the remaining 2 browsers, start a DM video chat with the last browser
   1. Verify that all 4 browsers see 2 videos
7. Shut down the server by pressing ctrl+c in the terminal running docker compose (or stop the containers using Docker desktop)
8. Verify that the video streams are not interrupted. After the signaling server is used to establish the connection, this is a true peer-to-peer stream

## Submission

Submit all files for your server to AutoLab in a .**zip** file (A .rar or .tar file is not a .zip file!). Be sure to include:

* A Dockerfile in the root directory
* A docker-compose file in the root directory that exposes your app on ports 80 and 443
* All of the static files you need to serve (HTML/CSS/JavaScript/images)

| It is **strongly** recommended that you download and test your submission after submitting. To do this, download your zip file into a new directory, unzip your zip file, enter the directory where the files were unzipped, run docker compose up, then navigate to localhost:8080 in your browser. This simulates exactly what the TAs will do during grading.  If you have any Docker or docker compose issues during grading, your grade for each objective may be limited to a 1/3. |
| --- |

## Grading

Each objective will be scored on a 0-3 scale as follows:

| 3 (Complete) | Clearly correct. Following the testing procedure results in all expected behavior |
| --- | --- |
| 2 (Complete) | Mostly correct, but with some minor issues. Following the testing procedure does not give the exact expected results |
| 1 (Incomplete) | Clearly incorrect, but an honest attempt was made to complete the objective. Following the testing procedure gives completely incorrect results or no results at all. This includes issues running Docker or docker compose even if the code for the objective is correct |
| 0 (Incomplete) | No attempt to complete the objective or violation of the assignment (Ex. Using an HTTP library) -or- a security risk was found while testing the objective |

Note that for your final grade there is no difference between a 2 and 3, or a 0 and a 1. The numeric score is meant to give you more feedback on your work.

| 3 | Objective Complete |
| --- | --- |
| 2 | Objective Complete |
| 1 | Objective Not Complete |
| 0 | Objective Not Complete |