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 still sent to /chat-messages when the page loads to request all the chat history. This end point must still host all of the messages sent to chat even after switching to WebSockets. Users will get new messages via WebSocket frames, but when the page first loads, they can get all the past messages with this end point.

Buffering: Your server must be able to handle WebSocket frames of any 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.

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.

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
    a. Verify that each user can see both their own messages, and messages sent by other users in real time
    b. Verify that the correct usernames show up with each message
    c. 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
    a. 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*
    a. 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
    a. Verify that it appears in all tabs/browsers with the authenticated username (ie. Auth tokens persist through a server restart)
15. Buffering:
    a. 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
    b. Verify that each user can see both their own messages and messages sent by other users in real time
    c. Verify that the messages have been sent/received using a WebSocket connection
16. **Security**: Verify that submitted HTML displays as text and is not rendered as HTML

Learning Objective 3: Multiple Frames

Expand your WebSocket code to support the processing of multiple frames under the following two conditions:

Continuation Frames: Check the FIN bit and handle messages that are sent over multiple frames. We will test with payloads > 131000 using chrome which sends messages over this size in multiple frames

Back-to-back Frames: Your server must also handle multiple 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

Testing Procedure

1. Start your server using docker compose up
2. Open two browsers and navigate to http://localhost:8080/ on both
3. Register accounts and login in both browsers
4. From Chrome, send multiple messages with a payload length > 131000
    a. Verify that each user can see these messages
5. 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)
    a. Send several messages from the modified tab, with < 50 characters, and verify that all users see all the sent messages 5 times
6. **Security**: Verify that submitted HTML displays as text and is not rendered as HTML

Application Objective: Video Chat Over WebRTC

Add features to your server so it can function as a WebRTC signaling server for your users. This signaling will occur over your WebSocket connection. 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. To avoid overcomplicating this objective, you may assume that there are exactly 2 WebSocket connections when receiving WebRTC messages. This means that when a message is received on one WebSocket connection, you will send the message to the only other WebSocket connection.

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 exactly 2 browser tabs (The same browser will be used for both clients) and navigate to http://localhost:8080/ on each
3. In each tab:
    a. Allow the camera/mic to be accessed
    b. If the local video has not started on the page load, click the "Start My Video" button
4. In one of the 2 tabs, click the "Start Video Chat" button
5. Verify that 2 videos are show in both of the tabs
    a. 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 tabs. You'll also hear some horrible sounding audio with feedback. Don't worry, that means it's working :)
6. Shut down the server by pressing ctrl+c in the terminal running docker-compose (or stop the containers using Docker desktop
7. 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 docker-compose file in the root directory that exposes your app on port 8080
- 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, but all features are functional
1 (Incomplete)	Not all features outlined in this document are functional, but an honest attempt was made to complete the objective. Following the testing procedure gives an incorrect result, or no results at all, during any step. This includes issues running Docker or docker-compose even if the code for the objective is correct
0.3 (Incomplete)	The objective would earn a 3, but a **security** risk was found while testing
0.2 (Incomplete)	The objective would earn a 2, but a **security** risk was found while testing
0.1 (Incomplete)	The objective would earn a 1, but a **security** risk was found while testing
0 (Incomplete)	No attempt to complete the objective or violation of the assignment (Ex. Using an HTTP library)

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

Autograded objectives are graded on a pass/fail basis with grades of 3.0 or 0.0.

Security Essay

For each objective for which you earned a 0.3 or 0.2, you will still have an opportunity to earn credit for the objective by submitting an essay about the security issue you exposed. These essays must:

- Be at least 1000 words in length
- Explain the security issue from your submission with specific details about your code
- Describe how you fixed the issue in your submission with specific details about the code you changed
- Explain why this security issue is a concern and the damage that could be done if you exposed this issue in production code with live users

Any submission that does not meet all these criteria will be rejected and your objective will remain incomplete.

Due Date: Security essays are due 1-week after grades are released.

Any essay may be subject to an interview with the course staff to verify that you understand the importance of the security issue that you exposed. If an interview is required, you will be

contacted by the course staff for scheduling. Decisions of whether or not an interview is required will be made at the discretion of the course staff.

When you don't have to write an essay:

- If you never submit a security violation, you never have to write an essay for this course. Be safe. Be secure

- If you earn a 0.1, there's no need to write an essay since you would not complete the objective anyway
- If you earn a 0.3 or 0.2 for a learning objective after the expected deadline, you may fix the issue and resubmit for the final deadline instead of writing an essay (Or you can write the essay so you don't have to sweat the final deadline)