Homework 3: Media Uploads

At this point, you have built a dynamic web application where users can login and chat with each other using only a TCP socket, your understanding of web protocols, and your programming skills. This is a great accomplishment. Now, let's take this further and allow users to upload and share images and movies.

Learning Objective 1: Multipart Parsing

Write a function in util.multipart named parse_multipart that takes a Request object as a parameter. This function will assume that the input Request is a multipart request and will extract all the relevant values of the request. This function returns an object containing the following fields (You have some freedom in how you design the classes for this object as long as they have the required fields).

- boundary
    - The value of the boundary from the Content-Type header as a string
- parts
    - A list of all the parts of the request in the order in which they appear in the request
    - Each part must be an object with the following fields
        - headers
            - A dictionary of all the headers for the part in the same format as a Request object
        - name
            - The name from the Content-Disposition header that matches the name of that part in the HTML form as a string
        - content
            - The content of the part in bytes
            - Note: The content may be binary and must never be decoded as a string. In LO2, this content will be binary images
            - Note: The content may contain the sequence b"\r\n\r\n" and there will be a test case for this. Be sure that you are not corrupting your data when this sequence is in the content of a part

Testing Procedure

1. Running the code using Python (Not Docker), run tests on the function 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: Image Uploads

Add functionality to your server that will allow users to upload images to chat using the "upload" button. When receiving requests sent using this button, your server should:

1. Parse the requests using your parse_multipart function
2. S0ave the uploaded image to disk, and create a chat message in your database containing a path to this filename as the src of an img (You will save and serve HTML as the message. This is safe since it will be your server generating the HTML)
    a. Let the existing GET /chat-messages path render these images on the front end via polling
3. Respond to the request with a 302 redirect to your homepage

When an image is uploaded, your server will save the image as a file. It is recommended that you devise a naming convention for the image files instead of using the names submitted by your users. Naming the images "image1.jpg", "image2.jpg", "image3.jpg", etc is fine.

For this objective, it is ok if your server only handles .jpg images and assumes that every file upload is a .jpg file.

Your uploads must persist through a server restart. You should store your images in files (It's generally bad practice to store large files in a database), and store the filenames in your database. Since your images are stored in files, they will already persist through a restart.

Buffering: Your app must allow for large images to be uploaded. You'll accomplish this by buffering your HTTP requests. Read the content length of the request and buffer until you read the whole body. Your buffering should be able to handle arbitrarily large files. You must use proper buffering for this. Do not increase the TCP buffer size by passing a large int to the recv method (That wouldn't work anyway for very large files).

Security: Don't allow the user to request arbitrary files on your server. Starting with this objective, you will be hosting content at paths that cannot be hardcoded since you don't know which images will be uploaded to your site. Even if you replace the file names with your own naming convention (eg. "image1.jpg" "image2.jpg") you still don't know how many images will be uploaded. This means that you must accept some variable from the user that you will use to read, and send, a file from your server. You must ensure that an attacker cannot use this variable to access files that you don't want them to access. (In this course, it is sufficient to not allow any '/' characters in the filename. Eg. remove any "/" characters from the requested filename after extracting it from the path)

Testing Procedure

1. Start your server using docker compose up
2. *Use only jpg images while testing this objective
3. Open a browser (Firefox and/or Chrome) and navigate to http://localhost:8080/
4. Use the image upload form to upload an image of arbitrary size (At least 1MB) (Without logging in)
    a. Verify that you are redirected to the homepage and that the image appears in chat with the username "Guest"
5. Register an account and login
6. Use the image upload form to upload an image of arbitrary size (At least 1MB)
    a. Verify that you are redirected to the homepage and that the image appears in chat with your username

7. Open a second browser in incognito mode
8. Navigate to http://localhost:8080/ in the second browser
    a. Verify that the images appear in chat as expected
9. Upload another image of arbitrary size (At least 1MB)
    a. Verify that you are redirected to the homepage and that the image appears in chat with the username "Guest"
10. Go back to the first browser
    a. Without refreshing, verify that all 3 images appear in chat as expected
11. Restart the server using docker compose restart
12. Refresh both browsers and verify that all images appear as expected for each user
13. Check the submitted code to ensure a very large TCP buffer was not used
14. **Security**: Verify that '/' characters are not allowed in the requested filename (Using Postman or curl), or look through the code to ensure that this attack is addressed
    a. If using curl, you can use this command:
        i. `curl --path-as-is http://localhost:8080/public/image/../../server.py`
15. **Security**: Look through the code to verify that prepared statements are being used to protect against SQL injection attacks [If SQL is being used]
16. Review the server code to ensure that the Router class has been used to route each request (If not explicitly state, this is assumed to be part of the testing procedures for every HW objective)

Learning Objective 3: File Types

Expand the functionality of your upload feature to include multiple different image types as well as video files. Specifically, your server must accept uploads of jpg, png, gif, and mp4 files. When serving these files, you must send them with the appropriate MIME type.

If the upload is an mp4, create a chat message using the HTML video element to display the video on the front end.

Never trust your users: You cannot rely on the file extension to determine the type of the file. You must read the file signature from the bytes of the files themselves (As opposed to the filename) to determine the type. Note that file extensions are merely a substring of the filename and can be changed to whatever you'd like when you name a file. It's not even required that a file have an extension. They are only meant to be a convenience as it makes it clear to us what the type is and lets the OS know what program it should use to open it by default

Testing Procedure

1. Start your server using docker compose up
2. Open a browser (Firefox and/or Chrome) and navigate to http://localhost:8080/
3. Upload at least one file of each type [jpg, png, gif, mp4]. Each file should have a file extension that does not match its actual type
    a. Make at least 1 upload with a filename that has no extension
4. Verify that all uploads appear in chat
5. When a video is uploaded, ensure that the video plays, or can be played, in all browser windows
6. Check the MIME type of all images and videos and verify that they match the type of the files uploaded

Application Objective: Size 240

Resize all images and videos to have a max width and height of 240 pixels. The original aspect ratio of the uploaded media must be preserved.

The intention is that you resize these files when they are uploaded and save the resized files to disk. Part of the purpose is to protect your server, and users, from people uploading very large files that take a lot of disk space and network traffic. If the chat is spammed with high resolution images and videos, your page loads will get slower. By storing and hosting smaller files, this risk is mitigated.

Note that the media itself must be resized. The height and width applies to the files that are saved on your server. This objective is **not** about the way the files are displayed on the front end, though they will naturally be displayed in their new sizes to the user without changing the front end at all (eg. changing the size via CSS or HTML attributes is not acceptable).

Testing Procedure

1. Start your server using docker compose up
2. Open a browser (Firefox and/or Chrome) and navigate to http://localhost:8080/
3. Upload at least 4 files including
   a. A portrait oriented image (height > width)
   b. A landscape oriented image (width > height)
   c. A portrait oriented video (height > width)
   d. A landscape oriented video (width > height)
4. Verify that all media appears in chat as expected. For each, check that:
   a. The files display without corruption
   b. No height or width exceeds 240 pixels in the files that are hosted (Use the network tab to check the resolutions)
   c. The original aspect ratios are maintained (The media should not be distorted, just resized)
   d. The actual files that are served

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)