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 a second form next to the chat input that allows users to upload images to the chat. You should preserve your original chat input as we'll use it in the next HW.

As long as uploaded images appear in chat with the proper username, you are welcome to accomplish this objective with any approach you'd like (You still cannot use libraries to complete it, but you have flexibility in the design of your architecture). If you do not want to design your own architecture, this is the expected/recommended approach:

1. Modify the front end by adding a new HTML form with the multipart/form-data encoding type next to your current chat input
2. Parse requests created by your form, save the image to disk, and create a chat message in your database containing 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)
   1. Let the existing GET /chat-message 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.

It is ok if your site only handles .jpg images and assumes that every file upload is a .jpg file for this objective.

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.

**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)

**Router Note**: If you are using the router from HW2 and are hosting your images at the path "/public/image/<filename>", you can use "add\_route('GET', 'public/image/.', your\_function)". The '.' in the path is a regex symbol that means "anything except an empty string" so it will match anything that starts with this path and has any string where you expect the filename. This way, your function will never be called with an empty filename since that will reach your 404 (unless it matches a different path). You can also use '.' to differentiate between two similar paths when one takes a parameter (eg. GET "/chat-message/$" returns all messages while GET "/chat-message/." looks up a message by id. Recall '$' means match the end of the string. Alternatively, you can remove the '$' as long as you add this path after the one that expects an id).

### 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)
   1. Verify that you are redirected to, or remain on, 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)
   1. Verify that you are redirected to, or remain on, 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
   1. Verify that the images appear in chat as expected
9. Upload another image of arbitrary size (At least 1MB)
   1. Verify that you are redirected to, or remain on, the homepage and that the image appears in chat with the username "Guest"
10. Go back to the first browser
    1. 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
    1. If using curl, you can use this command:
       1. 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]

## Learning Objective 3: Movie Uploads

Update your app to support both jpg images and mp4 videos using the same form that you build in LO2. When a file is uploaded, you may check the file extension to determine if it is an image or video.

* If the filename ends with exactly ".jpg" you may assume the file is an image and handle it the same way you did in LO2
* If the filename ends with exactly ".mp4" you may assume the file is a video and handle it accordingly
* No other file extensions will be used while testing this objective. Not even ".jpeg"
* The file extensions will always match the file types while testing this objective

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

Note: Since we're using polling, your video will reload every few seconds. You're welcome to code a solution for this, but it's not required and if your video constantly reloads that is ok for grading.

### Testing Procedure

1. Follow the testing procedures from LO2, but while uploading mp4 videos instead of images (It can be assumed that images work if LO2 passed)
   1. Only use files with names that end with exactly ".jpg" or ".mp4" and with types that match their file extensions
2. Verify that the videos appear in chat as expected
3. **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
4. **Security**: Look through the code to verify that prepared statements are being used to protect against SQL injection attacks [If SQL is being used]

## Application Objective 1: File Types

-Multiple file types. Find the file type using the binary. We test with mismatched file extensions (eg. an mp4 with an extension of gif). Support jpg, png, gif, and mp4 (we'll test with all of these and bad extensions. Hint: Research "file signatures"

Expand the support of your file upload form to accept more types of files as well as checking the types of the files. Add the following features to your form:

1. Add support for all jpg, png, gif, and mp4 files
2. You cannot rely on the file extension for this objective. 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 let's the OS know what program it should use to open it by default
3. To check the type of the file, you must read some of the bytes of the file content itself. You will want to research "file signatures" to accomplish this

### 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 of each type [jpg, png, gif, mp4]. Each file should have a file extension that does not match its actual type
   1. Make at least 1 upload with a filename that has no extension
4. Verify that all uploads appear in chat
5. Check the MIME type of all images and videos and verify that they match the type of the files uploaded

## Application Objective 2: Adaptive Bit-Rate Streaming

Re-encode and host all mp4 uploads using MPEG-DASH [or HLS] with at least 2 different resolutions. To accomplish this, you are expected to:

1. Install ffmpeg in your docker image
2. When an mp4 is uploaded, use ffmpeg to re-encode the video, save all the resulting files to disk, store the filename of the video index file in your database with the chat message
   1. You are allowed to use client libraries for ffmpeg, or make sys calls and command line arguments directly
3. Import a video player that supports MPEG-DASH [or HLS] playback for your front end

### 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 an mp4 video
4. Verify that the video appears in chat
5. Verify that the video is hosted using MPEG-DASH [or HLS] with at least 2 different resolutions

## 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 |
| 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, for at least 1 step. 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 |