Homework 1: HTTP and Docker

In this assignment, you will lay the foundation for the web server you will build throughout this course. You will become familiar with the basics of HTTP and Docker.

Docker

Note: For grading, Docker and docker compose are required for most objectives in the HW

Setup your web server to deploy with Docker and docker compose. In the root directory of your submission, include a Dockerfile and a docker-compose.yml file with everything needed to create an image and run your server in a container. You must run 2 containers, one for your app and one for your database.

- Map local port 8080 to your app in your docker-compose.yml file

Testing Procedure

This testing procedure for each objective [except learning objective 1] will be followed by the course staff while grading your submission.

1. Download your submission and extract the zip file in a new directory
2. cd into to the directory containing your submission
3. Run the command "docker compose up"
4. Navigate to http://localhost:8080/ in a browser and verify that your web page loads

You should use these steps to ensure that your Dockerfile/docker-compose.yml are set up properly to run your server. Even if you complete all the objectives, any error with your Docker/docker-compose.yml setup may result in not earning credit for the objectives. **Many students have lost objectives by rushing right before the deadline and submitting with the wrong docker configuration**. Don't let this happen to you.

Note: Testing in the course will be done using the latest versions of both Chrome and Firefox. You should ensure that your app functions properly in both browsers.

Learning Objective 1: Parsing and Routing HTTP Requests

To start this assignment, clone this github repository containing the starter code. You will expand this same starter code for every homework in this course and it contains much code for features that you won't fully implement until future assignments: https://github.com/jessehartloff/WebAppProject

The Request Class

Complete the Request class in the util package of the starter code. The constructor has been set up and contains all the variables that should be populated by parsing the input which will be an HTTP request in bytes. Your task is to parse the bytes and set the 'body', 'method', 'path', 'http_version', and 'headers' instance variables. Note that the body must be set as bytes while the other values will be set as str's (headers will map str to str).

The headers variable is a dictionary that must be populated with all of the headers of the request where the keys in the dictionary are the names of each header and the values are the header values. You must parse all of the headers of the request and add them all to this dictionary. (Technical note: HTTP requests are allowed to contain multiple headers with the same name. This means it's not 100% correct to store headers in a dictionary, but this will serve our purposes in this course while avoiding an increase in complexity).

If there is a Cookie header, all cookies must be parsed and stored in the cookies dictionary as Strings (The dictionary will map str to str). Your parser must support multiple cookies. Be sure to remove any leading/trailing white space from the cookie names and values.

Note: The raw Cookies header should still be in your headers dictionary even after parsing the individual cookies in the cookies dictionary.

The Router Class

You'll set up a Router class that will help you organize the code of your server throughout the semester. It is recommended that you study the provided hello path from the starter code and write at least one path (eg. hosting the HTML file at "GET /") without using the router to gain a better understanding of the structure of a server before implementing your Router class.

After Homework 1 you are welcome to make changes to the design of your router, but you are required to use a router throughout all homework assignments. The organization this router provides will prove useful as your server code gains more complexity.

Write a class named Router in a file named util/router.py with the following methods and functionality:

- If you use a constructor, it must take no parameters
- A method named "add_route" that takes 4 parameters and is used to specify how to route a specific request based on its method and path
    - The 4 parameters are
        - The HTTP method (str) of the request
        - The path (str) of the request
        - A **function** that takes a Request object, and a TCPHandler as parameters and does not return anything. This will be the function that handles request matching the given method and path
        - A boolean indicating if the path must match exactly or begin with the provided path. This boolean should have a default value of False
    - The add_route method does not return anything

    - This method is called to add a route to the router. For example, if you have a function named "send_home_page" that takes a Request and a TCPHandler and send an HTTP response containing the HTML of your home page using the handler that should be called on a GET request for the path "/", you would call add_route("GET", "/", send_home_page, True)
- A method named "route_request" that takes a Request object and a TCPHandler and does not return anything
    - This method will check the method and path of the request, determine which added path should be used, and call the function associated with that path with arguments matching the parameters of this method (eg. forward the Request and TCPHandler to the matching function)
    - If there is no path for the request, return a 404 response
    - If multiple routes match the request, use the route that was added first with the add_route method
    - When determining if a route is a match, the method and path must both match. If the boolean (from the add_route method) was True, the path must match exactly. If the boolean was False, the path of the request must start with the path for the route for it to be considered a match (eg. add_route("GET", "/public/image/", serve_image) uses the default boolean value of False so a request of "GET /public/image/eagle.jpg" would be routed to the serve_image function

Testing Procedure

1. Running the code using Python (Not Docker), run tests on the Request and Router classes to ensure they have all the functionality described above
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: Hosting an App

Host Public Files: Host all of the files in the public directory under paths starting with /public/ (eg. When you receive a request for the path /public/style.css you should respond with the contents of the style.css file from the public directory). All files must be served with the correct MIME type.

Security: The X-Content-Type-Options: nosniff header must be set on all responses (Please double/triple check this header for the exact spelling and syntax. If you are off by 1 character, the browser will not disable MIME type sniffing which renders your header useless and can be a security issue)

Root Path / Home Page: The index.html file must be hosted at the root path "/". Visiting the root path should load the entire page. (eg. When you receive a request for the path "/", you will serve the file public/index.html).

Images: The images must also be hosted. For example, a request for the path "/public/image/kitten.jpg" should display this adorable image:

UTF-8: Some files contain emojis that will be displayed when the page loads. These characters must display properly.

Content-Length: You must properly compute the Content-Length of each file. Recall that this length is the number of bytes, not the length of the string. For full credit, this length must be computed programmatically. Points will be lost if the length is hard-coded for each response.

Visit Counter: You must include a visit counter on your page that is tracked using a cookie. When a user first requests this page, set a cookie to 1 to track the number of times they visited this page. If the cookie is already set (Subsequent visits), read the cookie to check the number of times the user visited, increment this value by 1, then set the cookie again with the incremented value. This cookie should only be incremented when you receive a request for the home page (Path "/").

The visit counting cookie must have an expiration time of 1 hour or longer. **It cannot be a session cookie**.

The provided HTML has a placeholder with the string "{{visits}}" that should be replaced with your number of visits. To do this, you are expected to read the contents of the HTML file as a string, replace the placeholder with the correct value for the number of visits based on the value of the cookie, then serve the result after the replacement.

Do not set this cookie using JavaScript on the front end. Your Python server must set and read this cookie.

404: If a request is received for any path that should not serve content, return a 404 response with a message saying that the content was not found. This can be a plain text message.

Testing Procedure

1. Start your server using "docker compose up"
2. Open a browser and navigate to http://localhost:8080/
3. Verify that the browser displays the website
    a. Verify that an image of an eagle displays on the page
    b. Verify that the 2 emoji (non-ASCII characters) display properly
4. Verify that a UB icon is displayed in the tab for the page
5. In the network tab of the browser console:
    a. Verify that the HTML, CSS, JavaScript, and image were all served through separate HTTP requests
    b. Check each HTTP response for the correct MIME type
    c. Check each HTTP response for the correct Content-Length
    d. Verify in your code that the Content-Length was computed and not hardcoded
    e. **Security**: Verify that the "X-Content-Type-Options: nosniff" header is set on each response with this exact spelling and syntax
        i. To verify that the header is working, change the MIME type of your JS response to anything that is incorrect and verify that the JS did not run. If the browser is allowed to MIME sniff, your JS will still run even if the wrong MIME type on your response
6. Open a browser and navigate to http://localhost:8080/public/image/<image_name>
    a. <image_name> can be any image provided in the starter repo
    b. Verify that the browser displays <image_name>
7. Check the visit counter functionality:
    a. In the application tab of the browser console, find and clear all cookies for localhost
    b. Refresh the page
    c. Verify that the page displays a "1" for the visit counter
    d. Refresh the page
    e. Verify that the page displays a "2" for the visit counter
    f. In the application tab of the browser console, verify that there is a cookie set and that it has an expiration time (Make sure it is not a session cookie)
    g. Open a new window of the same browser and navigate to http://localhost:8080/
    h. Verify that the page displays a "3" for the visit counter
    i. Open a different browser (eg. If Chrome was used in the previous steps, use Firefox here and vica-versa) and navigate to http://localhost:8080/
    j. Verify that the page displays a "1" for the visit counter
    k. In the application tab of the browser console, find the cookie storing the number of visits and change the value to 100
    l. Refresh the page
    m. Verify that the page displays a "101" for the visit counter
8. Test 404 functionality:
    a. Open a browser, with the browser console open on the network tab, and navigate to http://localhost:8080/<any_string> where <any_string> is chosen by the tester and is anything that does not match a path used in this assignment

        i. Do not choose an <any_string> that starts with "public" (This makes it simpler to setup a server that serves every file in the public directory dynamically if a student chooses) or "chat-messages"
    b. Verify that the browser displays a message indicating that the requested content was not found
    c. In the browser console, verify that the response has a response code of 404, the Content-Type matches the type of content served, and the Content-Length contains the correct value
9. Review the server code to ensure that the Router class has been used to route each request

Learning Objective 3: Guest Chat and Database Storage

The provided front end contains a chat area that will send requests to your server. Add the following paths to your server to enable the chat feature:

- A path "/chat-messages" that accepts POST requests
    - This path is used when a user types a chat message and clicks send
    - The front end will send these chat messages in a JSON string with the format:
        - {"message": "The message being sent"}
        - You are allowed to use Python's json module to parse and format your JSON strings
    - When you receive a chat message, you will store it in your database with:
        - The message that was sent, the username of the user that sent the message (You can use "Guest" for now), and a unique id for the message
    - You may choose what you send in response to these requests (The front end will ignore the response)
- A path "/chat-messages" that accepts GET requests
    - This path is requested using polling to get the all the chat messages that have been sent to the app
    - Respond with all of the chat history in a JSON string representing an array of objects where each object has the keys "message", "username", and "id"
    - Eg. If there were two chat messages submitted saying "Hi there" and "hello", a valid response would be [{"message": "Hi there", "username": "Guest", "id": "1"}, {"message": "hello", "username": "Guest", "id": "2"}]
    - The default polling in the started code is 3 seconds. You can change this value to fit your testing preferences, but make sure it's <= 3 seconds before you submit (eg. If you set it to 30 seconds, we might assume you app is broken while grading)
- A path "/chat-messages/{id}" that accepts DELETE requests
    - The {id} in the path is the id of the record to delete
        - Eg. DELETE /chat-messages/1 to delete the message with id 1
        - Eg. DELETE /chat-messages/2 to delete the message with id 2
    - Respond with a 204 No Content response code
        - There is no body to your response (We don’t want to leak information about deleted records)

        - You will give the same response even if there is no message with that id or the message has already been deleted
    - After this request is sent for a message, that message should never be served again (It's deleted)
    - Note that the "X" next to each chat message will send a DELETE request for that message

Database: This is the first objective that will require a database. Your database must run in a separate container using docker compose. No credit will be given if you store your data in the same container as your app. This is intended to give you experience working with multi-container apps.

Security: You must escape any HTML in the users' messages. Since your users can submit any text they want, a malicious user could submit HTML tags that attack other users. **You cannot allow this.** You must escape any submitted HTML so it displays as plain text instead of being rendered by the browser.

Security: If you are using a SQL database, you must protect against SQL injection attacks.

Testing Procedure

1. Start your server using docker compose up
2. Open a browser and navigate to http://localhost:8080/
3. Find the chat box and submit several times with text including at least once with text including HTML
4. Verify that each message appears on the page within several seconds
5. **Security:** Verify that the submitted HTML displays as text and is not rendered as HTML
6. Delete several chat messages by clicking the X next to those message
    a. Verify the response is a 204 No Content with no body
    b. Verify that the messages are removed from chat and never reappear throughout the rest of this procedure
7. Refresh the page
8. Verify that all sent messages appear when the page loads
9. Restart the server using docker compose restart (Or restart using Docker Desktop)
10. Refresh the page and verify that all the messages still appear on the page
11. Look through the code and verify that a database running via docker compose is being used for the persistent storage
12. **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: Highlight Your Own Messages

In a way of your choosing, each user's own messages should be highlighted in their browser (eg. Their name can be in a different color even if all names are still "Guest").

Note: This objective, as with most application objectives, is much more open-ended than the learning objectives. You are expected to come up with creative solutions using the tools discussed in lecture to complete this objective. You might find it useful to modify the front end and create/track more cookies.

Testing Procedure

1. Start your server using docker compose up
2. Open a FireFox and Chrome window and navigate to http://localhost:8080/ in both
3. Send chat messages from both browsers
4. Verify that in each browser, messages sent from that browser and marked differently than messages sent from the other browser
5. Refresh both browsers and verify that the messages are still marked properly

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 the static files you need to serve in the public directory (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.

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)