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

| The 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 significant points 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 HTTP Requests

To start this assignment, clone this github repository containing the starter code: <https://github.com/jessehartloff/WebAppProject>

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

### Testing Procedure

1. Running the code using Python (Not Docker), run tests on the Request constructor to verify that the instance variables are properly set
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 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

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.

**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.

**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
   1. Verify that an image of an eagle displays on the page
   2. 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:
   1. Verify that the HTML, CSS, JavaScript, and image were all served through separate HTTP requests
   2. Check each HTTP response for the correct MIME type
   3. Check each HTTP response for the correct Content-Length
   4. Verify in your code that the Content-Length was computed and not hardcoded
   5. **Security**: Verify that the "X-Content-Type-Options: nosniff" header is set on each response
6. Open a browser and navigate to http://localhost:8080/public/image/<image\_name>
   1. <image\_name> can be any image provided in the sample site
   2. Verify that the browser displays <image\_name>
7. Check the visit counter functionality:
   1. In the application tab of the browser console, find and clear all cookies for localhost
   2. Refresh the page
   3. Verify that the page displays a "1" for the visit counter
   4. Refresh the page
   5. Verify that the page displays a "2" for the visit counter
   6. 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)
   7. Open a new window of the same browser and navigate to http://localhost:8080/
   8. Verify that the page displays a "3" for the visit counter
   9. 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/
   10. Verify that the page displays a "1" for the visit counter
   11. In the application tab of the browser console, find the cookie storing the number of visits and change the value to 100
   12. Refresh the page
   13. Verify that the page displays a "101" for the visit counter
8. Test 404 functionality:
   1. 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
      1. 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"
   2. Verify that the browser displays a message indicating that the requested content was not found
   3. 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

## 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 (yes, the path is the same for both endpoints)
  + 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 5 seconds. You can change this value to fit your testing preferences, but make sure it's <= 5 seconds before you submit (eg. If you set it to 30 seconds, we might assume you app is broken while grading)

**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. Refresh the page
7. Verify that all sent messages appear when the page loads
8. Restart the server using docker compose restart
9. Refresh the page and verify that all the messages still appear on the page
10. Look through the code and verify that a database running via docker compose is being used for the persistent storage
11. **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: RESTful API 1

Update your chat message end points to be a [mostly] RESTful API providing CRUD (Technically CRUDL) operations.

You may assume that all requests are properly formatted for this objective (ie. You don’t have to validate the requests).

* **Create**
  + POST /chat-messages
  + Update this end point to respond with a "201 Created" response code
    - The body of your response code will be a JSON object of the record that was created. This will be the same as what the user sent, with the addition of the username and id.
    - {"message": "Hi there", "username": "Guest", "id": "1"}
* **Retrieve - Single**
  + GET /chat-messages/{id}
    - Where {id} is the id of the record to retrieve
    - Eg. /chat-messages/1
    - Eg. /chat-messages/2
  + If the record exists, respond with a 200 OK response code
    - The body of your response will be a JSON object containing the requested record
    - If "/chat-messages/2" is requested, return {"message": "hello", "username": "Guest", "id": "2"}
  + If there is no record for the requested id, or the record has been deleted, return a 404 Not Found
    - You may choose the message of the 404
  + Note that if a GET request for /chat-messages is received, you must still return a list of all chat messages. In your code, you can check for a second "/" in the path to distinguish between the two

### 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 in the browser console:
   1. Send several chat messages and verify that the response code after each has a 201 Created response code
   2. Check the response and verify that it is a JSON string in the correct format
   3. Note the <id> from one of the responses
   4. Navigate to http://localhost:8080/chat-message/<id>
   5. Verify that the JSON string of the message is sent from the server
   6. Navigate to http://localhost:8080/chat-message/<bad\_id> where <bad\_id> is any string that is not one of the ids of the messages created
   7. Verify that the server returns a 404 response
4. Restart the server using "docker compose restart"
5. Access GET /chat-messages and verify that the records persist after a server restart

## Application Objective 2: RESTful API 2

Add the following endpoints to your chat messages API.

You may assume that all requests are properly formatted for this objective (ie. You don’t have to validate the requests)

* **Delete**
  + DELETE /chat-messages/{id}
    - There is no body to this request
  + If the record exists, respond with a 204 No Content response code
    - There is no body to your response (We don’t want to send deleted records)
  + If there is no record for the requested id, or the record has already been deleted, return a 404 Not Found
    - You may choose the message of the 404
  + 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
* **Update**
  + PUT /chat-messages/{id}
    - The body of the request will be a JSON object with message and username fields
    - {"message": "Welcome to CSE312!", "username": "Jesse"}
    - When this request is received, update the record with id of {id} using the data from the body of the request
  + If the record exists, respond with a 200 OK response code
    - The body of your response will be a JSON object containing the updated record
    - {"message": "Welcome to CSE312!", "username": "Jesse", "id": "2"}
  + If there is no record for the requested id, or the record has been deleted, return a 404 Not Found
    - You may choose the message of the 404
  + Note: There is no provided way to test this end point in the starter code. It is up to you to develop a way to test the update functionality. You do not have to provide this method on the front end of your app as we will have a separate way to test it during grading. Your testing method is for your own benefit.

### Testing Procedure

This procedure tests both application objectives.

1. Start your server using "docker-compose up"
2. Open a browser and navigate to http://localhost:8080/
3. Open the network tab in the browser console:
   1. Send several chat messages
   2. Note the <id>s from several responses
   3. Delete one of the records with DELETE /chat-messages/<id>
      1. Verify the response is a 204 No Content with no body
      2. Make a GET /chat-messages/<id> request to verify that the record is no longer accessible and returns a 404
   4. By any means necessary, not through the front end, update one of the records with PUT /chat-messages/<id> request
      1. Verify that the response contains the updated record
   5. Make a GET /chat-messages request to verify the change
      1. Verify that the deleted message is not in the result
      2. Verify that the updated record is properly updated
4. Restart the server using "docker compose restart"
5. Access GET /chat-messages and verify that the records persist after a server restart

## 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 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 |
| 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 |

Autograded objectives are graded on a pass/fail basis with grades of 1.0 or 0.0.