Homework 2: Authentication

You will continue to add features to your app from homework 1. There is no new repo to clone.

Learning Objective 1: Password Strength

Write the following functionality in your util package:

- A file named util/auth.py containing the following functions (This file does not define a class)
  - A function named extract_credentials that takes a Request object and returns a list containing 2 elements, a username then a password, both as strings (str)
    - The request object will either be in the format of a registration or a login request as generated by the provided front end (The same front end that you used for HW1). You must use the front end as-is since this is the exact format expected by the grader in Autolab
    - These requests will contain a url encoded string (The same format as query strings) containing these two values. You must parse the username and password from this encoded string. This includes decoding the percent-encoded characters
    - You are not allowed to use Python's urllib, or any other library that parses url encoded strings for you, to write this method
    - The username will only ever contain alphanumeric characters (eg. the username will never contain percent-encoded characters
    - The password can contain alphanumeric characters as well as the following special characters {'!', '@', '#', '$', '%', '^', '&', '(', ')', '-', '_', '='} (Note: We are limiting passwords to these 12 special characters to limit the tedium of writing this method. In the real world, you should support every valid character and you would also use a library to make this support trivial)
  - A function named validate_password that takes a string (str) and returns a boolean specifying if the password meets all of the following 6 criteria. If the function returns True, the password meets all the criteria and should be considered acceptable
    - The length of the password is at least 8
    - The password contains at least 1 lowercase letter
    - The password contains at least 1 uppercase letter
    - The password contains at least 1 number
    - The password contains at least 1 of the 12 special characters listed above
    - The password does not contain any invalid characters (eg. any character that is not an alphanumeric or one of the 12 special characters)

Testing Procedure

1. Running the code using Python (Not Docker), run tests on the methods/functions defined above to verify that they return 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: Authenticated Chat

Add authentication to your app. The provided front end contains 2 forms:

- A registration form
    - Used when a user creates an account by entering a username and a password
- A login form
    - Used to login after a user creates an account by entering the same username and password that was used when they registered

You should study the provided code to understand the format of the HTTP requests created by these forms. Do not modify the way these forms behave. They are designed to work with the extract_credentials method from LO1 and you should call this method to parse the username and password from both registration and login requests.

When a user sends a **registration** request, store their username and a salted hash of their password in your database. Their password must pass the criteria tested by your validate_password method or the registration fails.

When a user sends a **login** request, authenticate the request based on the data stored in your database. If the [salted hash of the] password matches what you have stored in the database, the user is authenticated.

When either a registration or login request is received, you must respond with a 302 Found redirect that sends the user back to your homepage. There is no requirement for sending messages to the user regarding failed registration login attempts. If no feedback is given to the user, that is ok for this objective (Please give proper messages when developing apps in the real world).

When a user successfully logs in, set an **authentication token** as a cookie for that user with the HttpOnly directive set. These tokens should be random values that are associated with the user. You must store a **hash** (no salt) of each token in your database so you can verify them on subsequent requests.

Authenticated Chat: Whenever a chat message is sent by an authenticated user, the message should contain their actual username instead of "Guest". (This is the only way we'll know if a user was properly authenticated and logged in).

Log out: When a user is logged in, they must have a log out button available to them. This will require you to modify your HTML template to add this button (preferably, you would replace the registration and login forms with a log out button for logged in users). When this button is pressed, the user's auth token must be invalided by your server (eg. the auth token they were issued should no longer work even if your server receives it as a cookie value after log out). After logging out, the user should see the registration and login forms again if they were removed.

Delete Messages: Add functionality to the delete message buttons in chat such that if a user clicks the delete button on one of their own messages, the message is removed from chat, but clicking on someone else's message does not remove it. The HTTP requests sent will contain the id of the message that was clicked. Use this id and the user's auth token to decide whether or not the message should be removed from chat. When a message is removed, it should no longer be sent in the chat-history response.

When a user attempts to remove a message that is not their own, respond to the request with a 403 response code. If it is their own, you may choose the response you send (200 with a short message is fine).

While building authentication, you must implement the following additional features:

- The auth token cookie must have an expiration time of 1 hour or longer. It cannot be a session cookie

Note: We will still check your visit counting cookie from homework 1 during this objective. Adding authentication must not break your previous code. This means that you must be able to handle multiple cookies simultaneously.

Security: Never store plain text passwords. You must only store salted hashes of your users' passwords. It is strongly recommended that you use the bcrypt library to handle salting and hashing.

Security: Only hashes of your auth tokens should be stored in your database (Do not store the tokens as plain text). Salting is not expected.

Security: Set the HttpOnly directive on your cookie storing the authentication token.

Testing Procedure

1. Start your server with "docker compose up"
2. Open a browser and navigate to http://localhost:8080/
    a. Clear all cookies stored for localhost
3. Find the registration form:
    a. Attempt to register with a password that does not meet all the required criteria
    b. Attempt to login with the same username/password
    c. Enter a message in chat and verify that the displayed username is "Guest"
    d. Register with a valid username/password
4. Find the login form and enter the same username, but an incorrect password

    a. Enter a message in chat and verify that the displayed username is "Guest"
5. Submit the login form again with the correct username and password from the registration step
    a. Send a message in chat and verify that it contains the chosen username
    b. Verify that the visit counter still updates properly on refresh
6. Restart the server with "docker compose restart"
7. Navigate to http://localhost:8080/
8. Send another message and verify that it contains the chosen username
9. Open a second browser (Use Chrome and Firefox. Switch to the one you didn't use in the previous steps)
    a. Register and login with a different username
    b. Send a message in chat and verify that the new username is used
    c. Verify that the message also appears in the first browser without taking any action (It should update via polling)
10. In the first browser:
    a. Find your auth cookie and copy your auth token. Store this token for later
    b. Modify your auth token and refresh the page
        i. Enter a message in chat and verify that the displayed username is "Guest"
    c. Paste your auth token back into your cookie and refresh the page
        i. Send a message in chat and verify that it contains the chosen username
    d. Find a way to logout through the app's interface and logout
        i. Refresh the page
        ii. Send a message in chat and verify that the displayed username is "Guest"
    e. Modify your cookies to set an auth token with the value saved earlier to simulate the same state you had when you were logged in, then refresh the page
        i. Send a message in chat and verify that the displayed username is "Guest"
    f. Log in again by entering your username/password
        i. Send a message in chat and verify that it contains the chosen username
11. Refresh the second browser
    a. Send a message in chat and verify that it contains the chosen username of your second account
12. In the first browser, check the delete feature by:
    a. Attempt to delete a message that is either a Guest message or a message from the other account and verify in the browser console that you receive a 403 response and the message is still in chat
    b. Delete one of your own message and verify that it was removed from chat in both browsers
13. **Security:** Check the server code to ensure passwords are being salted and hashed before being stored in the database
14. **Security:** Look through the code and verify that the tokens are not stored as plain text **and** that the cookie values are not the same as what's stored in the DB (ie. Whatever value you use in cookies, the DB must store a hash of that value. Setting

the cookies to the hashed values defeats the purpose of hashing and is a security risk)
15. **Security:** Verify that the cookies HttpOnly directive is set
16. **Security:** Verify that HTML is escaped in **chat messages**
17. ~~**Security:** Verify that HTML is escaped in **usernames**~~ (This is implied since usernames are said to only contain alphanumeric characters, and special characters would be percent-encoded)
18. **Security:** Look through the code to verify that prepared statements are being used to protect against SQL injection attacks [If SQL is being used]
19. Review the server code to ensure that the Router class has been used to route each request (This is assumed to be part of the testing procedures for every HW objective for the rest of the semester)

Learning Objective 3: XSRF Tokens

Add randomly generated XSRF tokens to your chat feature that are associated with the user that made the request for your home page. If a chat submission is received that does not contain a valid token for the user, do not save the submitted data and respond with a 403 Forbidden response notifying the user that their submission was rejected.

These tokens must be associated with a specific user. For example, when an authenticated user (check the auth token) requests your home page, you will generate (or retrieve) an XSRF token for this user. Whenever the user submits a chat message, you must check if their XSRF token matches a token that has been issued to this user. If the token was not issued to this user, you should respond with a 403.

It is ok to reuse a token multiple times for a single user. It is also ok to only have one token per user.

If a request is made from an unauthenticated user, you are not required to send/check a XSRF token. Guests can still chat without these tokens.

Note: This objective will require further modification of the HTML template and the JavaScript so you can inject the XSRF token to the page and the AJAX requests that send chat messages.

Testing Procedure

1. Start your server using docker compose up
2. Open a browser and navigate to http://localhost:8080/
3. Register an account and login
4. Refresh the page (To ensure that a request is made that includes an auth token)
5. Inspect the HTML of the chat form and verify that there is a XSRF token included somewhere as part of the form
6. Open a second browser, navigate to http://localhost:8080/, register and login with a different username than the one used in step 3
7. Find the XSRF in the second browser and copy it into the browser in the first browser

8. Send a chat message in the first browser and verify that the response is a 403 and the chat message does not appear in the chat (This XSRF token was not issued to this user)

Application Objective: OAuth - Login with Spotify

Build a login with Spotify feature. To do this, use the button labeled Login with Spotify. Then, use Spotify's OAuth 2.0 API to allow your users to login using their Spotify account. After authenticating their account through Spotify, use the User Profile endpoint to get the user's email address. Use this email address as their username for any messages they send.

You will use the Authorization Code Flow to build this feature and you should follow the documentation to guide your development. You will need a Spotify account to access the dev dashboard and this can be a free account. Note: You do not need to use the optional state.

Use "http://localhost:8080/spotify" as your redirect_uri. The course staff will use this exact string in their Spotify app used to test your app.

Don't forget to add the Spotify account you'll use for testing under User Management.

You may use a library (eg. requests) to send HTTP requests to the Spotify API. You are also allowed to use the base64 library.

Note: You are not required to handle refresh tokens for either of these AOs. You should code this for extra practice and experience, but since Spotify tokens expire in 1 hour it would be too tedious to test/grade such a requirement.

Security: Never share your app's client secret or client id. Your submission must not contain your private information and should instead have placeholder text where it should go.

Instead of submitting your code with this secret and id, set these values as environment variables in your docker-compose.yml file. Your submission should have placeholder values for these variables (ie. "changeMe" or "Put your client secret here"). For testing, we will paste our own secret/id into your docker-compose.yml before running docker compose up. You may also set your redirect_uri this way if you prefer not to hardcode it.

Security: You must properly use the Authorization Code Flow. Using a different flow, or improperly using the Authorization Code Flow, that exposes your app to security issues discussed in lecture is a security concern.

Testing Procedure

1. Create an app through Spotify's Dev Dashboard
    a. Set your redirect_uri to "http://localhost:8080/spotify"
    b. Under User Management, add the Spotify account you will use for testing
    c. Copy your client secret and client id into the corresponding environment variables in the submissions docker-compose.yml file

    d. If there is an environment variable for the redirect_uri, set it to the string above. Otherwise, you may assume that it is hardcoded to this string

2. Start the server using docker compose up
3. Open a browser and navigate to http://localhost:8080/
4. Click the login with Spotify button
5. When you are redirected to Spotify, accept the authorization
6. You should be redirected back to the home page
7. Send a message in chat and verify that the email used for your Spotify account is displayed as your username
8. **Security:** Look through the code and verify that no private information was submitted. This includes: client secret, client id, access tokens, refresh tokens, or authorization codes
9. **Security:** Look through the code and verify that the Authorization Code Flow was properly used

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)