Homework 2: Dynamic Web App

After HW1, your server can host a static front end and a dynamic API. Now, you'll write a dynamic front end where users can share data, and images, with each other. As you do this, there are many security concerns that will arise since you cannot trust that every user will act in good faith.

For this HW, you'll modify your index.html by adding the provided HTML form and hosting it at the root path "/". If you created a custom page, you can add the form to your page. You may also modify the inputs as much as you'd like as long as it still contains the required functionality (eg. The form must upload an image and at least one other value).

```html
<form action="/image-upload" id="image-form" method="post" enctype="multipart/form-data">
  <label for="image-form-comment">Comment: </label>
  <textarea id="image-form-comment" name="comment" cols="40" rows="6"></textarea>
  <br/>
  <label for="form-file">Image: </label>
  <input id="form-file" type="file" name="upload">
  <input type="submit" value="Post">
</form>
```

Objective 1: Text Form and Your First HTML Template

Handle requests sent via the form by storing and displaying the comments your users submit. For this objective, you may submit the form without choosing a file to upload. When a submission is made, parse the multipart request to find the comment, store the comment on your server, and display all submitted comments on your home page.

All comments submitted via this form must be displayed on your home page. You should convert the HTML of your home page to a template to accomplish this. Use your template to add all of the stored form submissions to the page in any format. This will allow you to generate dynamic content for your users by programmatically writing HTML.

When the form is submitted, respond with a redirect to your home page.

Security: You must escape any HTML in the users' submissions. 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.

Testing Procedure

1. Start your server using docker-compose up
2. Open a browser and navigate to http://localhost:8080/
3. Find the form and submit it several times with text, but without an image, including at least once with text including HTML

4. Verify that with each submission, the browser is redirected to the home page (This must happen with a 300-level response code. Do not respond to the form with your HTML)
5. Verify that all of the submitted data is displayed somewhere on the home page
6. **Security**: Verify that the submitted HTML displays as text and is not rendered as HTML

Objective 2: Image Uploads

Enable users to upload images, in addition to their comments, using the form. After this objective, your app will effectively be an imageboard site.

You must display all images and captions uploaded through this form on your home page. Use your template to add all of the stored images and captions to the page in any format of your choosing, as long as the captions are clearly associated with their corresponding images. To display the images, add their paths to <img> elements in your HTML. You don't need to embed the images themselves in your HTML.

When an image is uploaded, your server will save the image as a file, and also store the corresponding caption in a way that associates it with the image. 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.

Note: You may need to set up your buffer to complete this objective depending on which browser/version used during testing. Some browsers will send the headers of an HTTP request before sending the body so you will only read the headers the first time you read from the TCP socket. You need to read again to receive the bytes of the image.

Security: Don't allow the user to request arbitrary files on your server machine. Starting with this objective, you will be hosting content at paths that cannot be hardcoded since you don’t know what 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 file path.)

Security: Set the "X-Content-Type-Options: nosniff" header on each response containing a user uploaded image. This will prevent attackers from uploading content that is not an image using this form.

1. Start your server using docker-compose up
2. Open a browser and navigate to http://localhost:8080/
3. Upload one of the images from the HW1 sample site along with a caption

4. Verify that the browser is redirected to the home page and the image and caption are displayed
5. Upload a second image from the HW1 sample site with a different caption
6. Verify that:
    a. The browser is redirected to the home page
    b. Both images and captions are displayed
    c. Each caption is clearly associated with the correct image
7. **Security**: Verify that '/' characters are not allowed in the requested filename
8. **Security**: Verify that submitted HTML displays as text and is not rendered as HTML
9. **Security**: Verify that the "X-Content-Type-Options: nosniff" header is set on each image response

Objective 3: Large Image Uploads

Use proper buffering for the image uploads in Objective 2 to ensure that your server can handle very large files.

Testing Procedure

1. Follow the testing procedures of Objective 2 with .jpg images large enough to overflow your buffer

Objective 4: Persistent Posts

Store user posts in your database. All posts must persist even when your server restarts. Since your images are stored in file, they will already persist through a restart. In your database, you should store the comments along with the file paths to their associated images.

You must use a database that runs as a separate service started by docker-compose. Running a database in the same container as your app (eg. using SQLite, MongoDB), or using files to store persistent data (Other than the images themselves) is not allowed. Using any database that runs outside of your Docker containers (eg. Using MongoDB Atlas) is not allowed. The goal is for you to gain experience working with multiple containers through docker-compose.

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

Testing Procedure

1. Follow the testing procedures of Objective 2
2. Restart the server using docker-compose restart
3. Open a browser and navigate to http://localhost:8080/
4. Verify that all post still appear on the home page
5. Look through the code and verify that a database running via docker-compose is being used for the persistent storage
6. **Security**: Look through the code to verify that prepared statements are being used to protect against SQL injection attacks [If SQL is being used]

Objective 5: XSRF Tokens

Add randomly generated XSRF tokens to your form. If a form submission is received that does not contain one of your tokens, do not save the submitted data and respond with a 403 Forbidden response notifying the user that their submission was rejected.

Each time you receive a request [for the path "/"], you should generate a new token and add it to the HTML of the form in a hidden input. Whenever a form submission is received, it is ok to accept it if it contains *any* of your tokens. You don't have to check if it was the token you issued to that user (We don't have a reliable way to check the user until we have authentication. IP addresses can change).

Testing Procedure

1. Start your server using docker-compose up
2. Open a browser and navigate to http://localhost:8080/
3. Inspect the HTML of one of the forms and verify that there is a XSRF token included somewhere as part of the form
4. Refresh the page and verify that the token is different
5. Delete the token
    a. Submit the form
    b. Verify that the response from the server has a code of 403 and the browser displays a message that the request was rejected
    c. Navigate to the home page and verify that the submitted data does not appear on the page
6. Modify the token again, but instead of deleting it, append extra characters to the end of the token
    a. Submit the form
    b. Verify that the response from the server has a code of 403 and the browser displays a message that the request was rejected
    c. Navigate to the home page and verify that the submitted data does not appear on the page

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.yml file in the root directory that exposes your app on port 8080 and runs a separate container for your database
- A Dockerfile that is used by your docker-compose configuration
- 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, navigate to localhost:8080 in your browser, and go through the testing procedures for all 5 objectives. 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 is equally weighted and will scored on a 0-3 scale as follows:

3	Clearly correct. Following the testing procedure results in all expected behavior
2	Mostly correct, but with some minor issues. Following the testing procedure does not give the exact expected results
1	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	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

The following conversions will be used when translating grades on this 0-3 scale to percentage grades:

3	100%
2	80%
1	60%
0	0%