COSC 419
Lab 3: Making a Simple Login Application

In this lab, we'll be using the Flask framework to create a simple login system. Since we don't have a database yet, we'll hard-code the correct username and password into our application for now. Our application will prompt the user to log in, and use a session variable as a flag to signify a logged-in user. If a user tries to access our secure page without the session variable being set, we'll redirect them back to the login page. If the user does have the session variable set, we'll allow them to see the secure page. Lastly, we'll include a logout route that removes the user's session variable, and then redirects them to the login page.

Setting Up our Templates

Before you begin this lab, please take a copy of your **`myapp.py`** file, so that your previous lab can still be marked. You can easily make a copy using the following command, inside of your web root folder (/var/www/html):

```
cp myapp.py lab2_myapp.py
```

Verify that the lab2_myapp.py copy contains all your lab 2 code. Then, open up your original myapp.py file, and remove the lab 2 code from it. Your myapp file should only contain the basics required for a "Hello World" application before you begin (import, app object, single route for '/').

Once you've done that, we can get started setting up our templates. Download the **`baseTemplate.html`**, **`secureTemplate.html`**, and **`loginTemplate.html`** files from the lab Moodle or GitHub.

The **`baseTemplate.html`** file will serve as our "parent" template for the application – it contains the base of our DOM, including the <html> and <body> tags. There is a comment line in the HTML indicating where you need to add a template **`block`**. This will use the template logic format **`{% %}.`** You can name this block whatever you want, but you'll need to use that same name when we reference this block in our other two templates. Don't forget the **`endblock`** statement either:

```
{% block content %} {% endblock %}
```

This is where we will insert HTML from our other templates to create our two main pages. Once you've done that, either use SFTP to transfer the file or copy-paste the contents into a new file on your web server, inside the **`templates`** folder.

Next, open up the **loginTemplate.html** file. This file contains the form and styling for our login page. The first thing you'll need to do is add an **extends** directive to the HTML, extending the parent **baseTemplate.html** file:

```
{% extends "baseTemplate.html" %}
```

After the extend directive, you should wrap all the HTML in the file in a block directive, identical to the one you used in the **baseTemplate.html** file. This way, the whole login HTML chunk will be placed inside the base HTML to form our login page. Once you've done that, transfer the finished **loginTemplate.html** file to your **templates** folder as well.

Finally, we'll open up the **secureTemplate.html** file. This HTML represents the "secure" page that users will be able to access when they are logged in. To begin, extend the base template and add the content block around all the HTML, just as you did above for the **loginTemplate.html** file.

The secure page template should print the user's email address in place of the **_NAME_** placeholder. Do this by replacing **_NAME_** with an appropriate variable name in double braces:

```
{{ username }}
```

This will print out a **username** variable that we pass to the template when we call **render_template**.

Lastly, include some HTML of your own where indicated by a comment in the file. This will go below the top bar. This content can be anything you want (please keep it quasi-appropriate). There will be one bonus mark for students who do something interesting with this space.

Once you have done that, transfer your **secureTemplate.html** file to your server as well. You should now have all three template files in your **templates** folder, and both the login and secure templates should extend the base template, and fill it's content block.

Serving the Login Page

Now, open up your **myapp.py** file. Add a **route** directive for **'/'**, which accepts both GET and POST methods. Add a function for this route, and use the **request.method** object to check if the request being caught was a GET request:

```
if request.method == "GET":
```

You will need to import the **request** module from the **flask** library. If the request type is GET, we need to check two things:

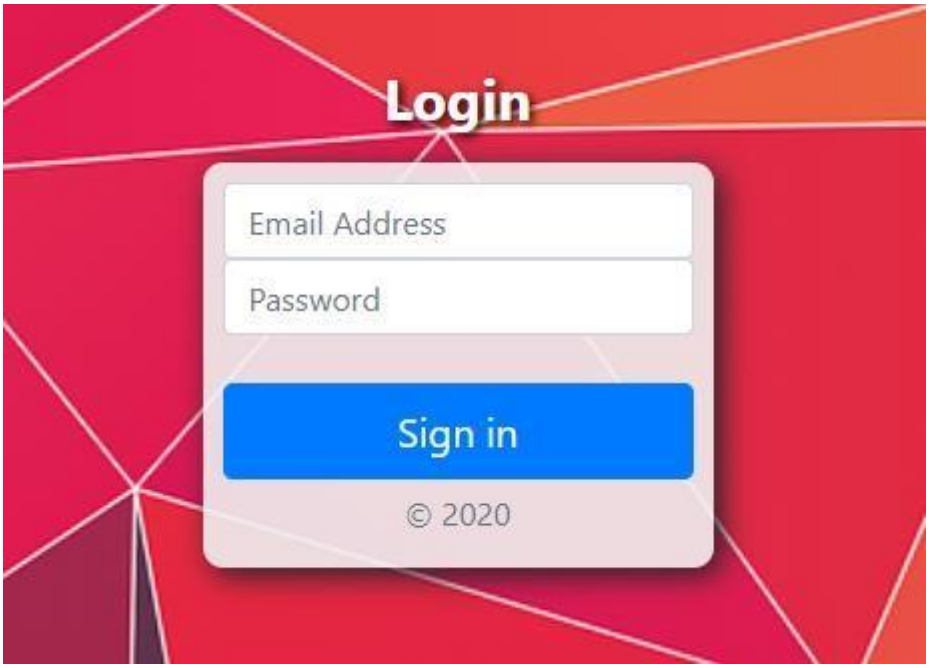
1. If the user has a session variable, redirect them to the secure page.

2. If the user does not have a session variable, call **render_template** on the login template and return it to the user.

Since we don't have a session variable set up yet, we'll just return the login page for now. You'll need to import **render_template** from **flask** as well, if you haven't already. You can then return the rendered template. Remember, the call should be made to the child template (the one which extends the parent):

```
return render_template("loginTemplate.html")
```

Save your file, and restart your web server. If you visit your IP address, you should see a page like this:



Handling User Input

Now that we have a login page, we can begin the other part of our '/' route: catching POST requests. Add a second if statement that checks to see if the request type was a POST request:

```
if request.method == "POST":
```

If the request was a POST request, this means the user submitted the login form, and we need to check the credentials. Use the **request.form.get()** function to fetch the user's inputs. The HTML form has two named inputs: **email** and **password**. Check to see if the email matches **test@test.net**, and check if the password is equal to **myAwesomePassword**. If either the email or the password are incorrect, perform a **redirect** back to the login page ('/') using the redirect function. You will need to import this from the flask module as before:

return redirect("/")

If the email and password are correct, we need to set a session variable to keep track of the user's logged-in status. Import **`session`** and set your application's secret key. Then, assign a new session variable to the user. The key and value may be anything you want, but you'll need use the same key throughout your code.

Once you've set a session variable, redirect the logged-in user to **`'/secure'`**.

Accessing the Secure Page

Create a new route in your application for **`/secure`**. This route should only accept GET requests, not POST requests.

In the function for this new route, we need to check to see if the session variable we set when the user logged in exists. We can do this easily (replace **`key`** and **`value`** with variable name you used):

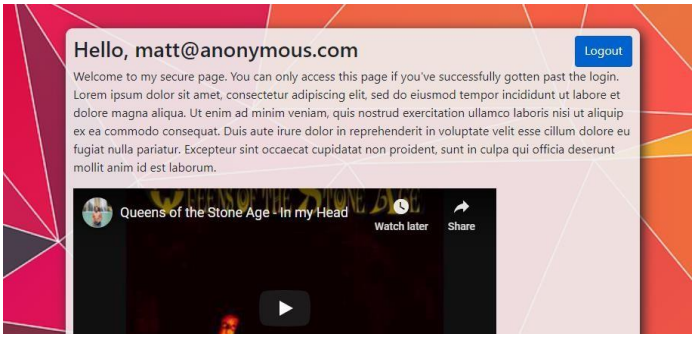
```
if session.get("key") == value:
```

If the session key exists and is equal to the expected value that we set earlier, then use **`render_template`** to render **`secureTemplate.html`** and return the page to the user. Don't forget to pass the user's email to the template to fill in the **`{{ username }}`** field. You will need to find a way to pass the username between requests, as the redirect will not preserve the original request (**Hint: use a session variable**):

```
return render_template("secureTemplate.html", username=email)
```

Otherwise, if the session key doesn't exist (suggesting the user isn't logged in), redirect the user back to '**/**' so that they can log in. Make sure that only a user with the correctly set session key can access the page, even if the key isn't set, or is set to the incorrect value.

Save your file and restart your server. Try visiting your website and logging in with the correct email and password. You should be redirected to the /secure page, which should look something like this (your content may be different):



Letting the User Log Out

The secure template page includes a logout button, which uses the **`/logout`** as a link. Create a new route for this, and a function which uses the **`session.pop()`** function to remove the session variable from the user. Pass it a second default value to make sure it doesn't throw an error, even if the user visits the login page after the having already logged out:

```
session.pop("key", None)
```

Then, have it redirect the user back to the home page '/'.

Finishing the Login

Now that you've got a working session variable, go back to the original '**/**' route function. Remember, if the user is already logged in, we want to redirect them to the **`/secure`** page. If the request type is a GET request, check if the session variable is set: if it is, redirect the user to the secure page. Otherwise, just render the login template like usual and return it to the user.

Submission

There is no required submission for this lab. You will be marked directly on the state of your server, using the following schema:

Marks	Item
3 Marks	GET/POST request handling for login page
2 Marks	Secure page with session variable checks
1 Mark	Logout page correctly removes session variable
1 Bonus Mark	Custom secure page content is interesting and/or funny