INFO3180 - Lab 4 (25 Marks)

Flask Login and File Uploads

Due: **March 6, 2024 at 11:59pm**

The goal of this lab is to implement a working login system using Flask-Login. The authentication will be against a PostgreSQL database and all routes should be protected to prevent viewing unless a user is logged in. You will also create database migrations using Flask-Migrate and create a File Upload form to handle file uploads.

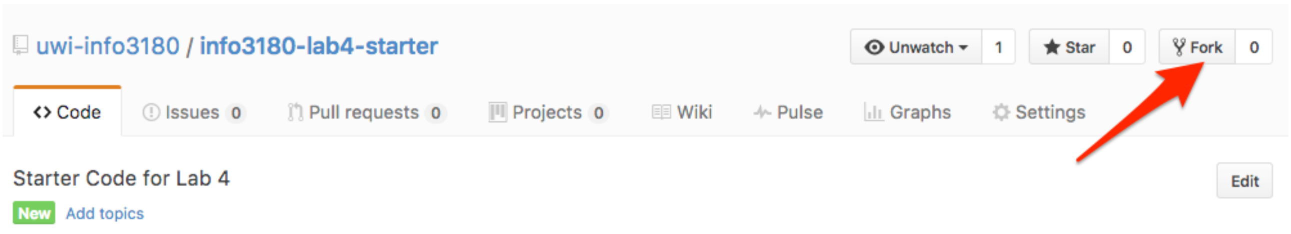
It is recommended that you read the documentation for the following Flask extensions:

- Flask-Login documentation at https://flask-login.readthedocs.org/
- Flask-Migrate documentation at https://flask-migrate.readthedocs.io .
- Flask-WTF File Uploads - https://flask-wtf.readthedocs.io/en/1.0.x/form/#file-uploads

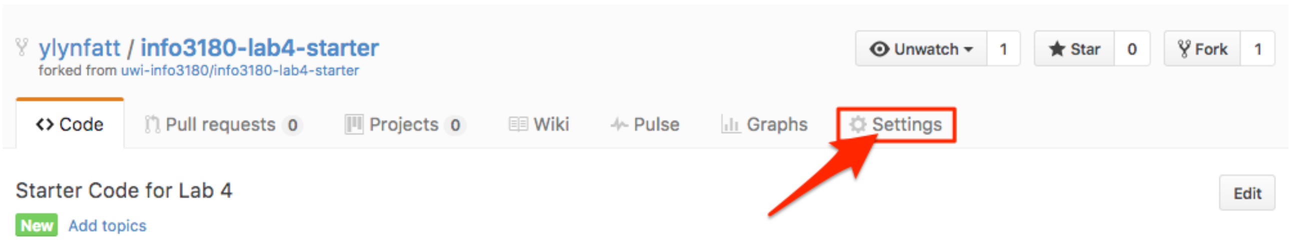
Step 1 - Fork and Clone the starter code repository

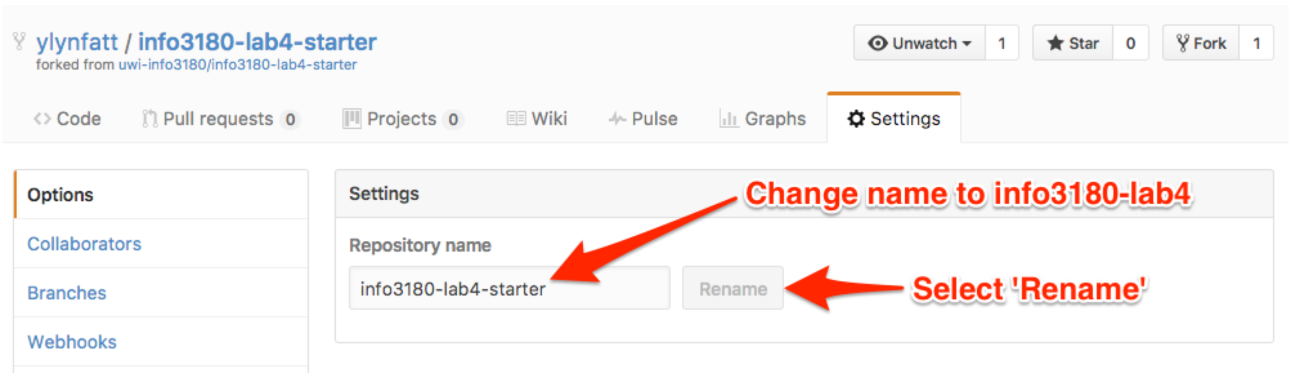
Start with the starter code at: https://github.com/uwi-info3180/info3180-lab4-starter

Fork the repository to your own account



In github.com rename it by going to settings and changing the name to info3180-lab4





To start working on your code clone it from your newly forked repository for example:

```
git clone https://github.com/{yourusername}/info3180-lab4
```

Step 2 - Setting up your PostgreSQL database

Installing PostgreSQL Ubuntu Linux

On Ubuntu Linux you can install Postgres using the following command:

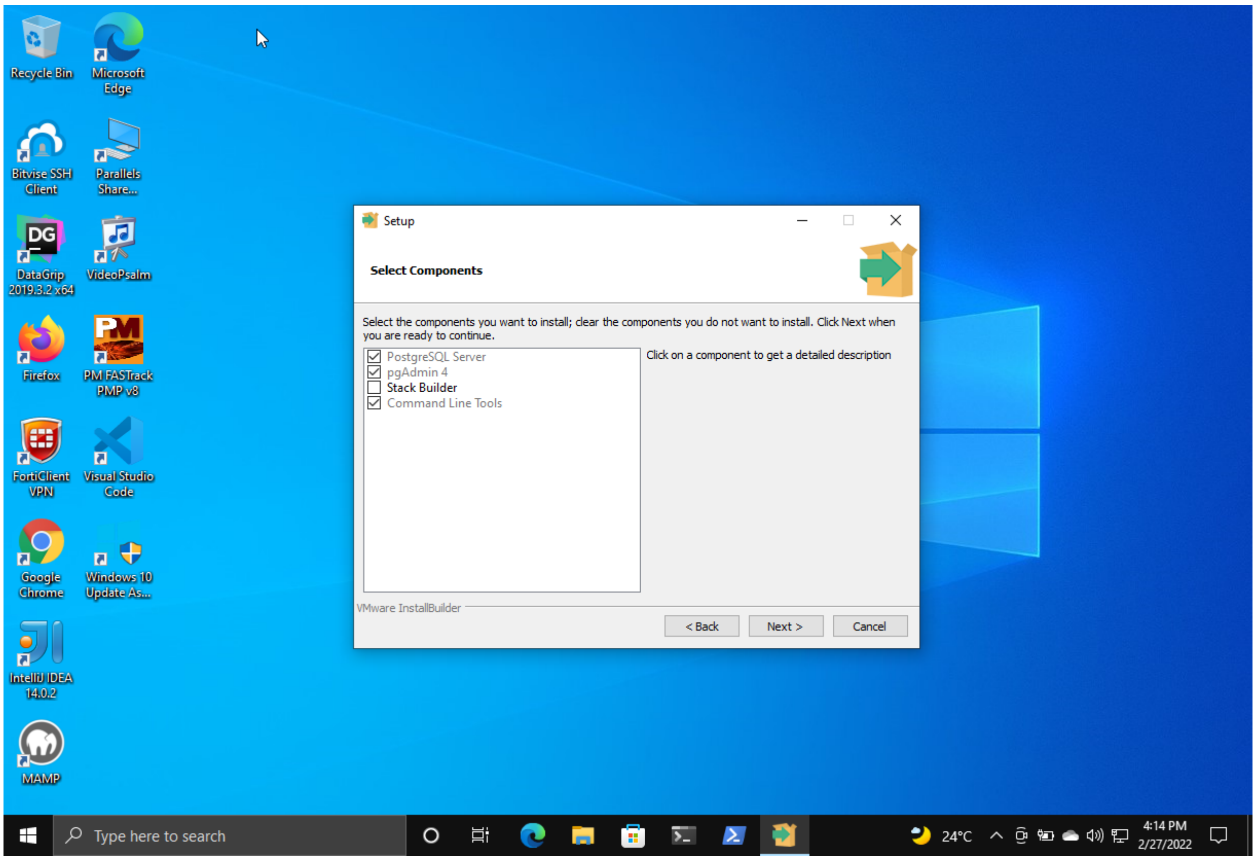
```
sudo apt install postgresql
```

Then connect to PostgreSQL command line interface by running in your Terminal:

```
sudo -u postgres psql
```

Installing PostgreSQL on Windows and MacOS

1. Download the installer from https://www.postgresql.org/download/windows and then open the installation file you downloaded. Follow the various steps to complete the installation. On the "Select Components" screen, ensure you only select "PostgreSQL Server", " pgAdmin4" and "Command Line Tools". Do NOT select "Stack Builder" as it is not necessary.

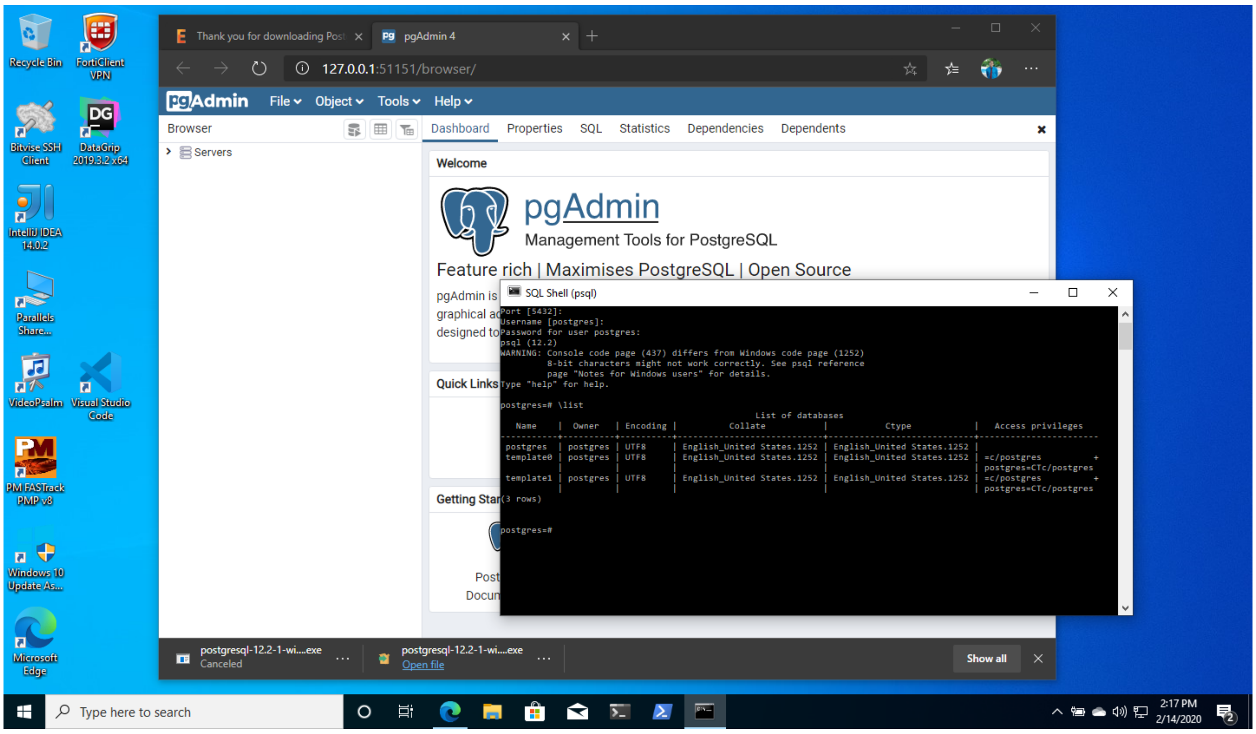


Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- [✓] PostgreSQL Server
- [✓] pgAdmin 4
- [ ] Stack Builder
- [✓] Command Line Tools

Click on a component to get a detailed description

2. Once the installation is complete, look for and open "**SQL Shell (psql)**" in the Start menu or via the Search box on Windows (or using the Spotlight Search on MacOS) to access the command line interface for Postgres. Optionally you can try to use the **pgAdmin GUI tool**.



```
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (12.2)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \list
                                                    List of databases
   Name    |  Owner   | Encoding |          Collate           |           Ctype            |   Access privileges
-----------+----------+----------+----------------------------+----------------------------+-----------------------
 postgres  | postgres | UTF8     | English_United States.1252 | English_United States.1252 |
 template0 | postgres | UTF8     | English_United States.1252 | English_United States.1252 | =c/postgres          +
           |          |          |                            |                            | postgres=CTc/postgres
 template1 | postgres | UTF8     | English_United States.1252 | English_United States.1252 | =c/postgres          +
           |          |          |                            |                            | postgres=CTc/postgres
(3 rows)


postgres=#
```

Setting up a database user and creating a database

If using the SQL Shell (psql) option

Once you have installed PostgreSQL and are connected to the PostgreSQL command line interface, create a user and a database and set a password for the user and then make them the owner of the lab4 database by doing the following steps:

```
create user "lab4_user";
create database "lab4";
\password lab4_user
alter database lab4 owner to lab4_user;
```

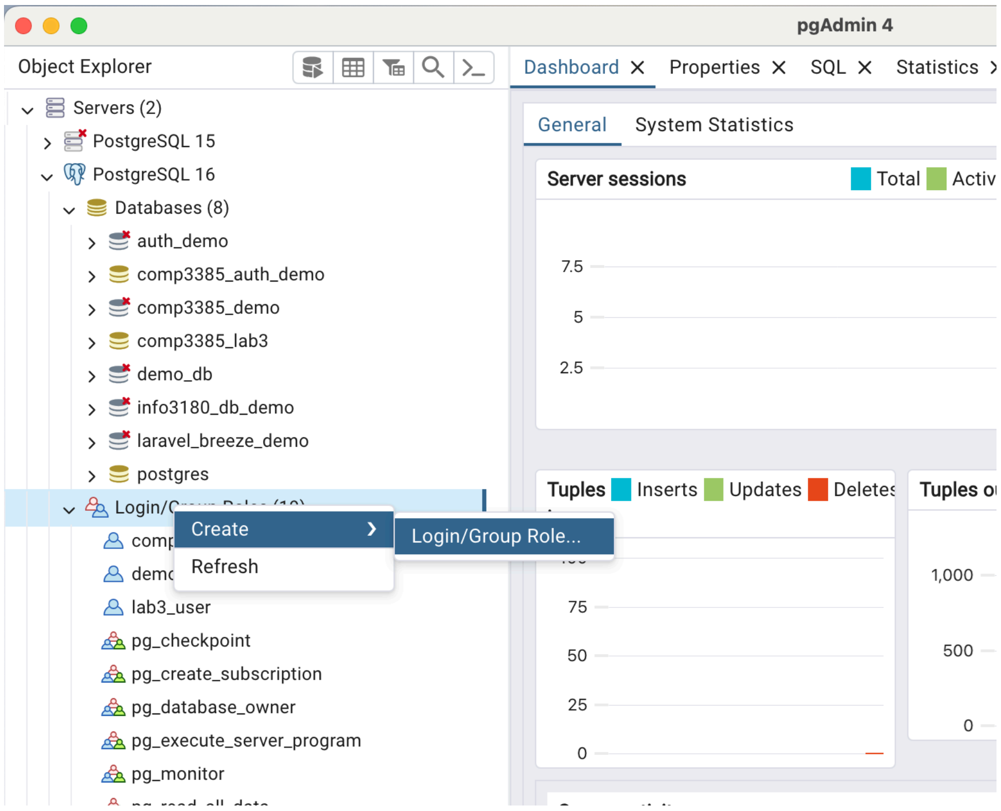
Note: The **`\password lab4_user`** command will prompt you to enter a password for the **lab4_user**. You can use whatever password you would like. Just ensure you remember it as you will need it for the next step.

You can now quit the PostgreSQL command prompt by using:

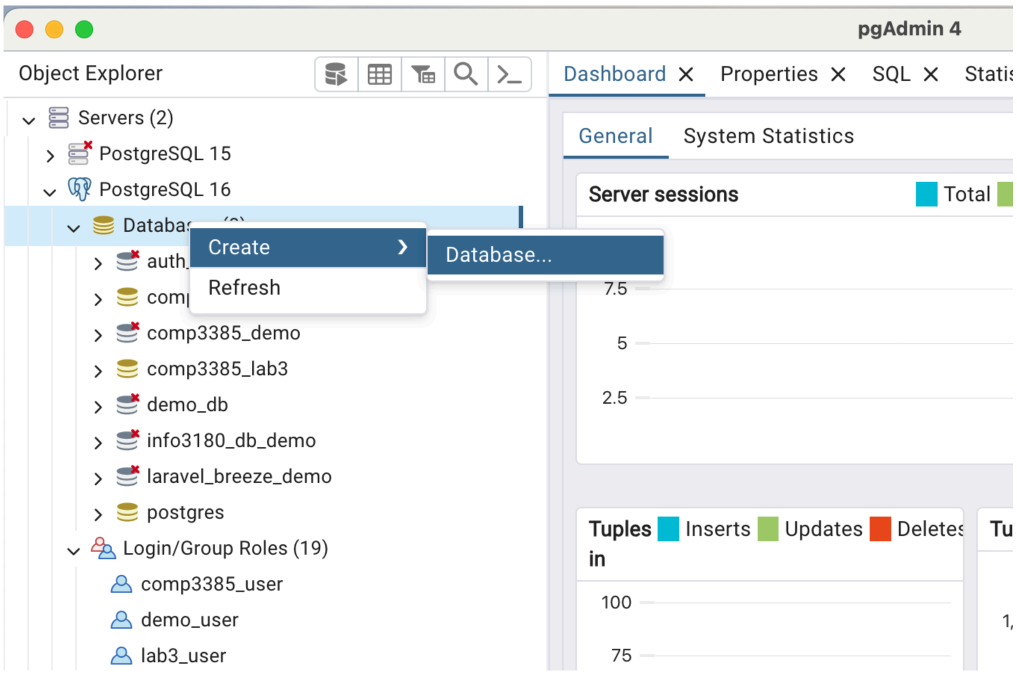
```
\q
```

If using the pgAdmin (GUI) option

1. Right click on PostgreSQL 16 in the list of servers and select **Connect Server**. You will be asked to enter the password for the **`postgres`** admin user. Enter your the password you used during the Postgres installation and click Ok.
2. Right click on **Login/Group Roles** and select **Create > Login/Group Role**.



3. Under the **General** tab an in the Name field enter **`lab4_user`** . Then under the Definition tab enter a password for the user in the **Password** field. Lastly, under the **Privileges** tab, ensure the **Can login?** option is enabled.
4. On the list of Databases right click and select **Create > Database**.



5. Give your database the name **`lab4`** . Then in the field for Owner, change it to use **`lab4_user`** instead.

Step 3 - Setup your upload folder and PostgreSQL database connection string

Look in the **`app/config.py`** file. You'll notice that the **`SQLALCHEMY_DATABASE_URI`** configuration (which is used to tell SQLAlchemy how to connect to the database) references an environmental variable called **`DATABASE_URL`** . We will need to set this environmental variable to tell it the correct connection string to properly connect to your **`lab4`** database with the **`lab4_user`** and password you created.

We'll do this by copying the **`.env.sample`** file and renaming it to **`.env`**. Once that is done update the **`DATABASE_URL`** environmental variable with the correct connection string. You will also need to set your **`SECRET_KEY`**.

The **`UPLOAD_FOLDER`** value should point to the "**`./uploads`**" folder so that Flask can know what folder to store your uploaded files. **Note:** The folder has already been created for you in the root of your application.

Note: Ensure you create your virtual environment as you have done in previous labs and activate it. Also ensure you take a look at the dependencies in your **`requirements.txt`** file and install these libraries for the application from your **`requirements.txt`** file by using the **`pip install -r requirements.txt`** command.

Exercise 1 - Define a Flask Login User model and create your first migration

Take a look at your **`models.py`** file. You will notice that you have been provided an initial **`UserProfile`** class. Pay attention to the properties that have been defined. These represent the columns that will be mapped to a table in your database. Also pay attention to the **`is_authenticated()`**, **`is_active()`**, **`is_anonymous()`**, and **`get_id()`** methods. These are needed by Flask-Login.

Now let us create our first migration by adding the Flask-migrate commands to our application and make them accessible to the Flask CLI. Open your **`app/__init__.py`** file and add the following in the appropriate places:

```
from flask_migrate import Migrate
migrate = Migrate(app, db)
```

You can then run the following commands to create and run your migrations:

```
flask db init
flask db migrate
flask db upgrade
```

You should now see a **`migrations`** folder in the root directory of your application. And if you check your database (by connecting back to the PostgreSQL Shell/command prompt or using the pgAdmin GUI), you should also see two tables **`alembic_version`** and **`user_profiles`**. Do this by running:

`\c lab4` `(this connects to the database)`
`\dt` (You should see the tables listed.)
`select * from user_profiles;` (should show your empty database table)

Commit your code and push to your Github repository.

Exercise 2 - Add a password field, Create a new migration and update your database

In order to manage local logins, you'll need to add a **password** field to the **`UserProfile`** class in your **`models.py`** file. Ensure the column is of the **String** type that can take up to **128** characters. Also you will want to ensure that you hash this password before it is stored in the database. To do this ensure you import the **`generate_password_hash()`** function from the **`werkzeug.security`** library and create an **`__init__()`** method within your **`UserProfile`** class. You can view this example as a reference: https://github.com/uwi-info3180/flask-demo-user/blob/master/app/models.py

You have now added a new password field to your model, however, you need to create a new migration to represent that change and also upgrade your database to reflect that change.

```
flask db migrate
flask db upgrade
```

Check your database again to see if the table now has the password column.

Commit your code and push to your Github repository.

Exercise 3 - Add a user to your database so you can login.

Now we will need to add a user so that we can test our login. Launch a python prompt from the command line and then you can use the following commands to quickly add a user.

```
$ flask shell
>>> from app import db
>>> from app.models import UserProfile
>>> user = UserProfile(first_name="Your name",
last_name="Your last name", username="someusername",
password="somepassword")
>>> db.session.add(user)
>>> db.session.commit()
>>> quit()
```

Note: You could also have connected to your PostgreSQL database and written the appropriate SQL insert statement. Or created a form within your web application and have it add the user when the form is successfully submitted (you'll do this in your project).

Use the PostgreSQL Shell/Command Prompt or the pgAdmin GUI and query the user_profiles table again to see if the user was added.

Exercise 4 - Update the /login route

Now let us take a look at your **`views.py`** file. You will see that a **`login`** route, view function and **`login.html`** template has been defined, however it is incomplete. It needs to be able to do the following:

- Validate that a username and password was entered on your login form. Change the if statement to check **`if form.validate_on_submit()`**.
- Check that the username and password entered matches that of the user you created in your database in Exercise 3. You'll need to actually make a query to the database using the **`UserProfile`** model and utilize the **`filter_by()`** method.
- You will also need to ensure that when checking that the password matches, that you utilize the **`check_password_hash()`** function from the **`werkzeug.security`** library (this will need to be imported).
- Redirect the user to the **`/upload`** route and display a **`flash`** message to the user letting them know that they have successful logged in. You will create this route in the next exercise.

Commit your code and push to your Github repository.

Exercise 5 - File Upload Form

1. Open the **`forms.py`** file and create a Flask-WTF form class named "**`UploadForm`**" with a file upload field only. Ensure that you create the necessary form validation to make the file upload field **required** and allow **only image files** (e.g. jpg and png files) to be uploaded.
2. Import that form in your **`views.py`** file.
3. Alter the **`upload`** view function so that:
    a. You ensure that you add the **@login_required** decorator to the route so that it is only accessible when a user is logged in.

   b. You instantiate the form class you created in Step 1 and pass it to your **`upload.html`** template file via the **`render_template`** function when a **`GET`** request is made.
   c. You validate the form data on submit (using the appropriate Flask-WTF function) when a POST request is made.
   d. If the validation passes, save the file to your upload folder using the configuration option you created earlier in the setup of the lab. **Note:** Also ensure you use the **`secure_filename()`** function before saving the file. (Look at the top of your ***`views.py`*** file to ensure that it has already been imported.)

4. Open the **`upload.html`** file and ensure you add the appropriate **`method`** and **`enctype`** attributes to the **`<form>`** tag to allow for file uploads. Also display the file upload field and it's associated label using the correct variables from the Flask-WTF form you created.

   Note: You also need to ensure you print out the Flask-WTF **`csrf_token`** hidden field or else your validation may fail.

5. Test that your file uploads work by starting the development server for your Flask app, go to the **`/upload`** route and log in. Upload a file and then check your uploads directory to see if it is there.

6. Commit your code to your repository.

Exercise 6: Listing your uploaded files

Here's an example python script for iterating over some files in a specific directory. **Note:** You will need to make modifications to get this to work in your lab, as it will not work as you see it here.

```
import os
rootdir = os.getcwd()
print rootdir
for subdir, dirs, files in os.walk(rootdir + '/some/
folder'):
    for file in files:
        print os.path.join(subdir, file)
```

1. Use the code from the example above to help you create a helper function called **`get_uploaded_images()`** in your app which iterates over the contents of the **`uploads`** folder and stores the filenames in a python list which the function will return. **Hint:** Yes, you will need to make some modifications to the code above, it won't work as is.
2. Create a route called "**`/uploads/<filename>`**" and a view function called **`get_image()`**, it should take an argument called "**`filename`**". You will use this view function to *return* a specific image from your upload folder using the **`send_from_directory()`** function. e.g. **`send_from_directory(os.path.join(os.getcwd(), app.config['UPLOAD_FOLDER']), filename)`**

3. Next, create another new route called "**`/files`**" and its respective **`files()`** view function. You should also render a template named "**`files.html`**" which lists the image files uploaded in the **`uploads`**

folder as an HTML list (ie. using an unordered or ordered list) of images.

Note: You should use the function you created in Step 1 to get the image and pass the list of image filenames to your template. You will then need to use the **`url_for()`** function to help print the proper url in your **`<img />`** tags that uses the route you created in Step 2.

4. Open **`header.html`** and add a new menu item to your navigation called "**View Images**" and link it to the "**files**" route that you created in Step 2.
5. Commit your code to your Github repository.
6. Ensure that the **`/files`** route is only accessible to users who are logged in by adding the **@login_required** decorator to this route as well.
7. Now see if you can modify your **`files.html`** template file to display and style your list of images to look like the screenshot (see Figure 1).

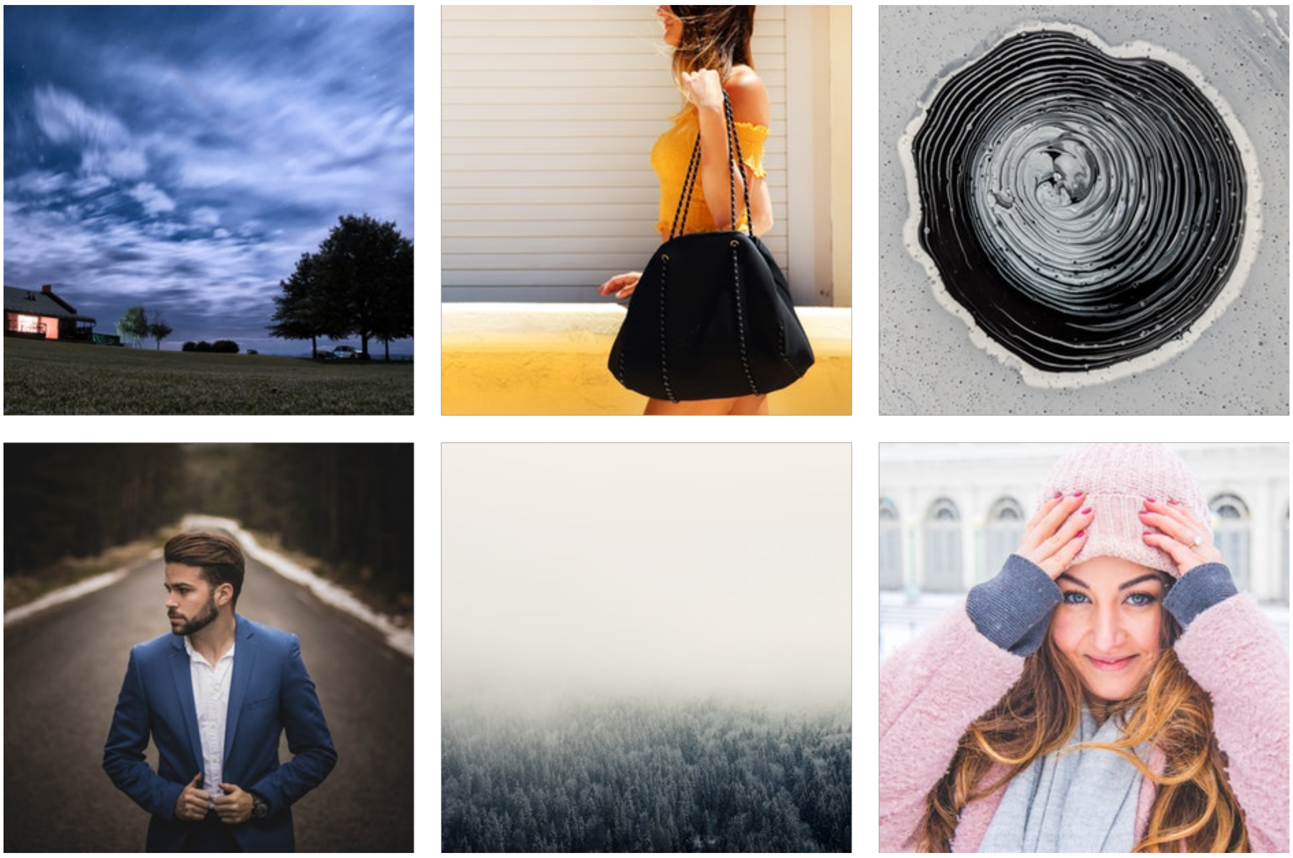


Figure 1: Images should be displayed in a Grid

Hint: You may want to try using CSS Grid or Flexbox to help with this layout. 😉 **DO NOT use an HTML table**.

Commit your code and push to your Github repository.

Exercise 7 - Create a logout route

1. Create a **`logout`** route and ensure you use the Flask-Login, **`logout_user()`** method to logout a user.
2. Ensure you also **`flash`** a message to the user and **`redirect`** them to the **home** route.
3. Also, update your **`header.html`** template file and ensure that you switch the login link to display logout link only when the **`current_user is_authenticated.`**

`Commit your code and push to your Github repository.`

Submission

Submit your code via the "Lab 4" link on the VLE. You should submit the following link:

1. Your Github repository URL for your Flask app e.g. https://github.com/{yourusername}/info3180-lab4

Grading

1. You should have 2 migration files and when the migrate command is run it should recreate the database table with all the relevant columns for this lab. (1 marks)
2. Your models.py file the `UserProfile` class should have a password field. (1 mark)
3. Your UserProfile model should have a constructor (ie __init__() method) defined with instance variables and ensure your password is hashed before storing it in the database. (2 marks)
4. You should query the database for the username and password and ensure the login_user() method is used to login that user. (2 marks)
5. Create UploadForm class and file upload field defined with appropriate validation rules. (2 marks)
6. Use form.validate_on_submit() to check form validation (1 mark)
7. Use app.config['UPLOAD_FOLDER'] in code to save file, also ensure you use secure_filename() (2 marks)
8. The form should have the correct 'method' and 'enctype' attribute set. Also the form should have a file field and CSRF Token. (2 marks)
9. Your Image File should successfully upload (1 mark)

10. The get_uploaded_images() function should be defined and it should return a Python list of image filenames. (2 marks)
11. A Files route, associated view function and template file should be created (2 marks)
12. A "/upload/<filename>" route with associated get_image() view function that uses the send_from_directory() function to return a specific image from the uploads folder. (2 mark)
13. The file upload form and files routes should only be accessible when logged in. (1 mark)
14. The Files route should look like screenshot (see Figure 1) and use CSS Grid or Flexbox to define the grid layout. (1 marks)
15. A link should be added to the navigation bar for the page that displays all the uploaded images. (1 mark)
16. Create a logout route and logout() view function which uses the logout_user() method. (1 marks)
17. Add a logout and login link in your header.html template which should change when user logged in/out. (1 marks)