ECE1779 Assignment 1 Documentation

General Architecture

How to Use the Web Application

- Start the EC2 instance on AWS Educate (under angela.ye@mail.utoronto.ca account).
- SSH into EC2 and run ./start.sh under home directory. Nginx and Gunicorn will automatically serve the app on port 80 and 5000.

```
./start.sh
```

- Use the following command to run gen.py

```
python3.7 gen.py [parameters]
```

Specifications

There are some assumptions we made in our web application. Firstly, all usernames are case insensitive. For example, `SpIdErMaN` and `spiderman` refer to the same username and can be used to log into the same account. Next, we assume that the username and the password both cap at 20 characters long. However, the password has to be at least 4 characters while the username can be as short as 1 character. We also limited the type of images a user can upload to JPG, JPEG and PNGs only. The maximum file size is set at 10 MB.

Frontend

```
opencv-app
├── README.md
├── babel.config.js
├── dist
├── node_modules
├── package-lock.json
```

```
    ├── package.json
    ├── public
    |   └── index.html
    └── src
        ├── App.vue
        ├── assets
        |   └── global.css
        ├── components
        |   ├── ImageUpload.vue
        |   ├── LoginStatus.vue
        |   ├── NavBar.vue
        |   └── Welcome.vue
        ├── main.js
        ├── router
        |   └── index.js
        └── views
            ├── Home.vue
            ├── Login.vue
            ├── Register.vue
            ├── Thumbnails.vue
            └── Upload.vue
```

The table below summarizes the main files and directories.

File	Description
babel.config.js	Babel configuration file.
dist/	Compiled frontend webpages and resources.
node_modules/	Directory used to store all npm dependency modules.
package.json	JSON file used to specify npm dependencies.
public/	Contains Vue render template.
src/	Vue source files.
src/main.js	Vue render entrance.
src/App.vue	Vue render mounting component in main.js.
src/assets	Global assests including .css files.
src/components	Vue components.

src/router	Front end vue router config file.
src/views	Vue view pages.

Our frontend was developed with Vue and Element UI. We implemented five responsive pages in total, including welcome page, log in page, register page, upload page, and thumbnail list page. Axios was used to send GET/POST requests and several validators are implemented to perform multiple input checks in the front end.

After the coding was done, we built a release version using webpack, and then used Flask to send frontend pages to clients.

Backend

The backend of the web application is implemented using the FLASK framework. The general file structure is outlined as follows:

```
├── Pipfile
├── Pipfile.lock
├── README.md
├── api
|   ├── endpoints
|   |   ├── photo.py
|   |   ├── ta.py
|   |   └── user.py
|   ├── helpers.py
|   ├── parsers.py
|   ├── restplus.py
|   ├── serializers.py
|   ├── yolo-coco
|   |   ├── coco.names
|   |   ├── yolov3.cfg
|   |   └── yolov3.weights
|   └── yolo.py
├── app.py
├── config.py
├── database
|   ├── __init__.py
|   └── models.py
├── migrations
```

```
|   ├── README
|   ├── alembic.ini
|   ├── env.py
|   ├── script.py.mako
|   └── versions
|       └── 9a71a8450641_.py
└── uploads
```

The table below summarizes the main files and directories.

File	Description
Pipfile	The file used by Pipenv virtual environment to manage project dependencies.
app.py	The entry point file that initializes flask server and database.
config.py	Loads .env file with variables based on ENV and populates the config object that can be imported in other files
api/	Folder containing API logic.
endpoints/	Folder for files for API endpoints.
user.py	File containing logic for registering and logging in a user.
photo.py	File containing logic for uploading, processing and viewing the photos.
ta.py	File containing the two endpoints for the TA to perform automatic testing.
helpers.py	File that contains helper functions and logic called by the endpoints to communicate with the DB.
parsers.py	Parsers are the structure templates for body data used in POST requests.
restplus.py	File storing the definition of the RESTPlus API.
serializers.py	Similar to parsers, serializers define the (JSON) structure of outputted data returned from endpoints.
yolo-coco/	Folder containing YOLO weights and configs and names of classes in coco dataset.
yolo.py	File that contains code for performing object detection using YOLOv3.
database/	Folder for database settings and models.
__init__.py	Creates a database instance that can be imported from other files.
models.py	SQLAlchemy Representations of database tables

migrations/	Database migration versions generated by flask db init and manually edited to reflect the database structure
uploads/	Folder for storing photos on the local file system.

Our backend framework uses Flasks's Flask-RESTPlus library to make REST APIs. It comes with a feature called Swagger UI to generate a webpage at `relative_path_to_url:port/api` that displays documentation for the APIs and allows for test queries. `restplus.py` constains the definition of a Flask-RESTPlus API instance in `api/`. The RESTPlus API is split into three namespaces: `user`, `photo` and `ta`, each stored in a separate file in `api/endpoints/`. These namespaces are added to the API using the `api.add_namespace()` function in `app.py` with each referring to a different URL prefix.

`app.py` in the root directory contains the logic for configuring and starting the Flask application. We have a function `create_app()` which sets up a Flask blueprint to host the API under the `/api` URL prefix. Meanwhile, the frontend logic is hosted in the same Flask application but with the `/` URL prefix.

Using Flask-RESTPlus, we define each endpoint by a separate class, which can have different HTTP methods issued to it. For each HTTP method, such as `GET` or `POST`, there may be decorators validating the method parameters. For example, `api.expect()` takes in a parser which includes a specific set of argument values that the HTTP method is allowed to handle, while `api.marshal_with()` returns a JSON object with the same fields as specified in the parameter.

Lastly, for authentication, we used the `flask_jwt_extended` library. We configured the application to store JSON Web Tokens (JWTs) in cookies. Each time a user is logged in, we create new access and refresh JWTs using the user's ID and set them in the response cookies. These cookies are set to expire after 24 hours. When a user logs out, we send the frontend a response to delete the cookies. For uploading and viewing the photos, we used the decorator `@jwt_required` in the HTTP methods so that the endpoints are protected and only users with a JWT can access it. Furthermore, we use `get_jwt_identity()` to validate a user's ID. This ID is used to perform DB operations so that a user cannot access or modify another user's photos.

Database Schema

Our database consists of two tables: `users` and `photos`.

The table `users` consists of three columns: `id`, `username`, and `password`. The `id` is the primary key of the table and is auto-incremented with each new record. The `username` is a string and the `password` is generated using the PBKDF2 algorithm from Python's `passlib.hash` library, whereby a 16-byte salt is autogenerated.

The table `photos` consists of five columns: `id`, `user_id`, `thumbnail_link`, `photo_link`, and `processed_link`. The `id` is the primary key of the table and is auto-incremented with each new record. `user_id` is the foreign key to the `users` table's `id`. `thumbnail_link`, `photo_link`, and `processed_link` contains the filename (with extensions) to the thumbnail photo, original photo, and the processed photo, respectively. To save or access the photos, the web application will navigate to the directory stored in `UPLOAD_FOLDER` in `config.py`.