Caliber Data Labs Assignment

Osman Baalbaki – osman@alumni.ubc.ca

1. Approach:

The first step in approaching this problem is deciding which libraries to use. The three main options for the design of the REST API are Flask (Vanilla), Flask-RESTFul, and Django. Django is the oldest and most established of the three. While, Flask-RESTFul is more geared towards designing REST APIs. However, Flask is pretty light compared to Django and does not require extra dependencies unlike Flask-RESTFul. After some more research and consideration of my personal preferences, I decided to use Vanilla Flask.

The next step is to divide my learning/coding work into smaller tasks and tackle each individually before integrating everything together. The three parts I divided my work to are:

- **Glare-Check/Logic**: This part involves finding the altitude and azimuthal difference angles and subsequently checking if there is possibility of glare.
- **Error Handling:** This part involves creating custom error classes, as well as handling default http errors and unexpected errors.
- **Authentication:** This is not necessary for the functionality and was not directly requested by the problem statement. However, for the sake of learning and to create a more realistic REST API, I decided to look into different authentication techniques and test them.

2. Findings/Work:

For the main logic of my API, I decided to make use of the pysolar Python library. This library has get_altitude and get_azimuth functions that take in a latitude, longitude, and a time object and give the altitude and azimuth angles of the sun. Since the time given is in UNIX time, I need to convert it into a date and time format. This requires me finding the timezone, as UNIX time is timezone independent. To find the timezone, I used a timezonefind function from the timezonefinder Python library. This function takes in latitude and longitude and finds the timezone at the specified location. This timezone is then used when converting my UNIX time to date and time format. Lastly, the outputs from my sun_position() function (the sun altitude and azimuth angles) are used in the view function to determine if their is glare, and the result is displayed to the user as JSON.

For handling errors, I decided to create my own exception classes. This allows me to customize the output to the user of the API and throw errors specific to my API. Additionally, I am going to have a @app.errorhandler(Exception) to handle all unexpected errors not handled by our custom error classes. For those unexpected errors, I will always return the same generic message. Lastly, I added error handlers for default http errors such as 404 (if user enters wrong endpoint) or 405 (if user inputs a method that is not allowed).

For my error output format, I decided to use a modified version of IETF's RCF7807 standard.[1] This will tell the user the title of the error, a status (code) and details of the error/how to proceed.

Even though authentication is not necessary for the functionality of this API, I decided to test different authentication techniques for the sake of learning, and just as a good habit to get used to when creating APIs. The three authentication techniques that I considered are: Basic, Bearer/Token

[1] https://tools.ietf.org/html/rfc7807

Authentication, and OAuth2. I tested Basic authentication where the user sends a username and password that are matched with a user-database. I also tried Bearer authentication, which is an extension of Basic auth, where the user gets a limited-time token upon sending their login information. This token can be used when sending methods through the REST API for authentication. I did not get the chance to test OAuth2.0. Nonetheless, for the final REST API I decided to not include any authentication as to not stray too far from the assignment's requirements.

Lastly, some unit testing is done. If any authentication is added, it is important to perform unit testing for all the authentication functionalities.

3. Limitation/Assumptions:

- Weather is not accounted for in the glare results. In reality, even if the sun is within the correct altitude and azimuth angles relative to the camera, clouds can still block the sun's glare.
- The API only allows one image metadata to be entered at a time.
- You cannot use this API for 'predicting' glare in the future, as I made it throw an exception if the provided epoch time is in the future.

4. Recommendations/Future Work:

- One possible extension to this project based on one of the limitations is to feed the latitude, longitude and time data to an API that can give us the weather at the specified time and place. Thereafter, we can factor the effect of clouds on blocking the sub. Obviously, there are caveats to this approach, as a cloudy weather does not necessarily mean that the clouds are eliminating the sun's glare.
- One possible improvement is to use get_altitude_fast() and get_azimuth_fast. These are faster but less accurate alternatives to get_altitude() and get_azimuth. Depending on the importance of the speed for our API, using these functions instead could be beneficial.
- Allow the input of nested JSON objects so the user can input multiple sets of image metadata and get the results corresponding to each.
- Implement Authentication. If this is to be only used internally to communicate between a server and clients/engineers, then a Bearer/Token Authentication is likely sufficient. If this is to be used as a public API with consumers having access, then OAuth2.0 Authentication is necessary.
- Add more custom errors. My code already captures several custom errors and sends tailored messages to the users of the API. However, there are likely more custom errors/exceptions that can be added to make the API more robust and easier to use.