Programmer's Guide

University of Maryland Global Campus
SWEN 670 – Team B
Spring Semester
Version 1.0

March 25, 2023

Version	Issue Date	Changes
0.1	3/22/2023	Initial Version
1.0	3/25/2023	Milestone 3 Submission

1. Introduction

1.1. Purpose

The ViroTour programmer's guide is a document that helps the audience walk through the process of setting up the tools, showing the architecture, and explaining all the decisions made in general.

1.2. Intended Audience

The main targets of this runbook are the development team and stakeholders. Because the document is both technical and informative, it will help them understand the work done in such an environment.

1.3. Technical Project Stakeholders

Stakeholder Name	Project Role
Dr. Mir Assadullah	Client/Professor
Roy Gordon	Project Mentor
Robert Wilson	DevSecOps Mentor
Ivelin Tchangalov	Project Manager (PM)
KC Harden	Product Owner (PO)
Alex Armel Wabo Tebu	Business Analyst (BA)
Hang Wang	Sr. Software Engineer (Dev Lead)
Nancy Lay	Test Manager (Test Lead)
Kelvin Huynh	Jr. Software Engineer (Dev)
Melika Shahani	Jr. Software Engineer (Dev)
Jean Pita Diomi Kazadi	Jr. Software Engineer (Dev)
Jah-wilson Teeba	Jr. Software Engineer (Dev)
Ronald Milligan	Jr. Software Engineer (Dev)
Ian Fischer	Jr. Software Engineer (Dev)

Table 1.3 - Technical Project Stakeholders.

1.4. Project Documents

Here is the list of documents in the software documentation package.

	Document	Version	Date
1	Project Management Plan (PMP)	3.0	03/25/2023
2	Software Requirements Specification (SRS)	2.0	3/25/2023

3	Technical Design Document (TDD)	1.0	2/12/2023
4	Software Test Plan (STP)	1.0	2/12/2023
5	Programmers Guide (PG)	1.0	3/25/2023
6	Deployment and Operations (DevOps/Runbook)	1.0	3/25/2023
7	User Guide (UG)	1.0	
8	Test Report (TR)	1.0	

Table 1.4 - Project Documents

1.5. References

Changing the contrast and brightness of an image! OpenCV. (n.d.). Retrieved March 23, 2023, from https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html

How to fast change image brightness with python + opencv? Stack Overflow. Retrieved March 23, 2023, from https://stackoverflow.com/questions/32609098/how-to-fast-change-image-brightness-with-python-opencv

Implement glow filter in CV2 python. Stack Overflow. Retrieved March 23, 2023, from https://stackoverflow.com/questions/68592934/implement-glow-filter-in-cv2-python

Rosebrock, Adrian. (2020). *Blur and anonymize faces with OpenCV and Python.* Retrieved March 24, 2023 from https://pyimagesearch.com/2020/04/06/blur-and-anonymize-faces-with-opencv-and-python/

SQLALCHEMY 2.0 documentation. SQLAlchemy Unified Tutorial - SQLAlchemy 2.0 Documentation. (n.d.). Retrieved March 23, 2023, from https://docs.sqlalchemy.org/en/20/tutorial/index.html

Quick start¶. Flask. (n.d.). Retrieved March 23, 2023, from https://flask-sqlalchemy.palletsprojects.com/en/3.0.x/quickstart/

1.6. Definitions, Acronyms, and Abbreviations

Product Name	Version
Anaconda	Open-source distribution of Python and R programming languages
Android	Mobile operating system by Google
API	Application Programming Interface
Azure	Cloud computing service by Microsoft

CRUD	Create Retrieve Update Delete: the four basic functions of persistent storage
EasyOCR	Optical Character Recognition (OCR) engine
Flask	Lightweight web application framework for Python
Flask-SQLAlchemy	Flask extension for SQLAlchemy
Flask-Testing	Flask extension for unit testing
Flask-WTF	Flask extension for integrating WTForms
GeoJSON	A format for encoding geographic data structures
Git	Distributed version control system for software development
GitHub	Web-based platform for version control and collaboration
IDE	Integrated Development Environment
Jinja2	Template engine for Python
JSON	JavaScript Object Notation: a lightweight data-interchange format
Keras	High-level neural networks API (Application Programming Interface) for Python
Marshmallow	A popular serialization/deserialization library for Python
Matplotlib	Plotting library for Python
NumPy	Library for numerical computing in Python
OpenCV	Open-source computer vision and machine learning software
ORM	Object-Relational Mapping: a programming technique for converting data between incompatible type systems using object-oriented programming languages
Pip	Package installer for Python
POST	HTTP method for sending data to a web server
PyCharm	JetBrains IDE for Python
PyClipper	Polygon clipping library for Python
Pygame	Set of Python modules designed for writing video games
Pygame Zero	Library for making video games
Pyrsistent	Immutable data structures for Python
Pytest	Testing framework for Python
Scikit-image	Image processing library for Python
SciPy	Library for scientific computing in Python
SQL	Structured Query Language for managing relational databases
SQLAlchemy	SQL toolkit and Object-Relational Mapping (ORM) library for Python
TensorFlow	Open-source machine learning library by Google
Torch	Open-source machine learning library
Torchvision	Open-source computer vision library for Torch
UI	User Interface
Werkzeug	WSGI (Web Server Gateway Interface) utility library for Python
WTForms	Python package for creating web forms

Table 1 - Document Acronyms and Definitions.

2. Operating Systems and Development Tool Versions

Product Name	Version
Android Studio	Electric Eel | 2022.1.1
Anaconda	22.9.0
Git	2.19.2
Pip Package	23.0
PyCharm IDE	Community Edition 2022.3.2

Table 2 - Project Software Tool Versions.

3. Development Environment

3.1. Integrated Development Environment

3.1.1 PyCharm

PyCharm is a JetBrains IDE for Python, similar to the Java IDE, IntelliJ. This will provide all the tools needed to build our backend application which uses the Python language.

PyCharm's documentation and downloads can be found here:
https://www.jetbrains.com/pycharm/

3.1.2 Python

Python is a general-purpose language that can use multiple design paradigms, including object-oriented, procedural, and structured. We will use this language for all development in the PyCharm IDE. We need version 3.9.X or earlier for the EasyOCR library used in the image text extraction functionality. The latest at time of writing (3.11.2) does not work with EasyOCR implementation.
Installing Anaconda can facilitate the use of multiple Python versions.
https://www.anaconda.com/products/distribution

Python's documentation can be found here:
https://www.python.org/

3.1.3 Flask

Flask is a lightweight web application framework that works well with Python web application frameworks. This project will use a flask server to handle

the communication between the front end and back end of the ViroTour Application.

When opening Flask in PyCharm, import the project from the spring2023/virotour_local/flask folder.

3.1.4 Project Setup

The project setup instructions for the back-end Flask server can be found in the repository README under the spring2023/virotour_local/flask directory.

The machine should be restarted after the Anaconda installation.

1. Open a command prompt and change the directory to the spring2023/virotour_local/flask folder.
    a. ```cd spring2023/virotour_local/flask```
2. Create the virtual environment.
    a. ```conda create -n py39 python=3.9.16```
    b. Type ‘y’ when prompted.
    c. Activate the virtual environment.
        i. ```conda activate py39```
3. Note that you may need to set your execution policies to activate.
    a. ```Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process```

Edit the PyCharm settings to have PyCharm use the conda environment that was just created.

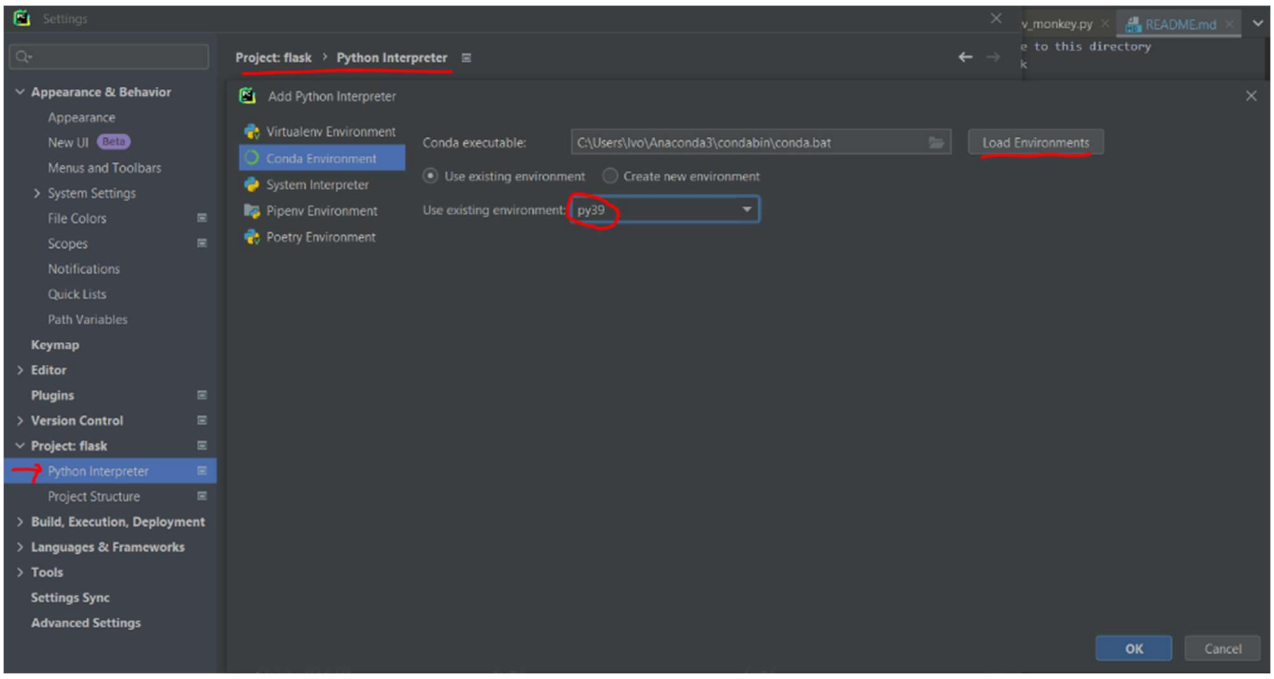

Figure 3.1.4 - Assigning New Conda Environment Under Python Interpreter Setting

4. Install predefined python libraries.

```pip install –r requirements.txt --user```

5. Run the tests.

```pytest app/tests/```

6. Run the Server.

```python run.py```

7. The following libraries should be installed (as a result of step 4):

Library Installations	**Version**
attrs	22.2.0
certifi	2022.12.7
charset-normalizer	3.0.1
click	8.1.3
colorama	0.4.6
contourpy	1.0.7
cycler	0.11.0
easyocr	1.6.2
exceptiongroup	1.1.0
flasgger	0.9.5
Flask	2.2.3
Flask-SQLAlchemy	3.0.3
Flask-Testing	0.8.1
Flask-WTF	1.1.1
fonttools	4.38.0
greenlet	2.0.2
idna	3.4
imageio	2.26.0
importlib-metadata	6.0.0
importlib-resources	5.12.0
iniconfig	2.0.0
itsdangerous	2.1.2
Jinja2	3.1.2
jsonschema	4.17.3
kiwisolver	1.4.4
lazy_loader	0.1
MarkupSafe	2.1.2
matplotlib	3.7.0
mistune	2.0.5
```

networkx	3
ninja	1.11.1
numpy	1.24.2
opencv-contrib-python	4.5.5.64
opencv-python	4.5.5.64
opencv-python-headless	4.5.4.60
packaging	23
Pillow	9.4.0
pluggy	1.0.0
pyclipper	1.3.0.post4
pyparsing	3.0.9
pyrsistent	0.19.3
pytest	7.2.1
python-bidi	0.4.2
python-dateutil	2.8.2
PyWavelets	1.4.1
PyYAML	6
requests	2.28.2
scikit-image	0.20.0
scipy	1.9.1
shapely	2.0.1
six	1.16.0
SQLAlchemy	2.0.4
tifffile	2023.2.28
tomli	2.0.1
torch	1.13.1
torchvision	0.14.1
typing_extensions	4.5.0
urllib3	1.26.14
Werkzeug	2.2.3
wincertstore	0.2
WTForms	3.0.1
zipp	3.14.0

Table 3.1.4 - Library and Version Numbers

3.1.5 - Troubleshooting the Environment

If you are having trouble with the environment, here is how to wipe everything in Anaconda and start over:

```git pull ```
```conda deactivate py39```
```conda remove env -n py39 --all```
```conda install --revision 0```
```conda create -n py39 python=3.9.16```
```conda activate py39```
```pip install -r .\requirements.txt --no-cache-dir --user --force-reinstall ```

4. Frameworks and Languages

Our software application is built using a combination of frameworks and languages to achieve the desired functionality and performance.

For the back end, we use Python which is a high-level, interpreted programming language with dynamic semantics. Python is a popular language for web development because it is easy to learn, has a vast library of modules and packages, and has a simple syntax that makes it easy to write readable and maintainable code. We use Python for tasks such as server-side logic, database management, and API development.

By using these frameworks and languages, we are able to build a high-performance and scalable application that provides a great user experience. We continually evaluate new frameworks and languages to ensure that we are using the best tools for the job and delivering the best results to our users.

5. Development Process

The team used GitHub for version control during the development process and used the following steps:

1. **Create a GitHub branch:** Branching lets the team have different versions of the main project repository at one time. This allows the team to implement and test different business logic.

2. **Making and committing changes:** Team members create and test business logic. After testing, the changes are made to the branch for others to review.

3. **Open a Pull Request:** PR is used to propose new changes and request that someone from the team review and pull in your contribution and merge them into their branch.

4. **Merge Pull Request:** In this final step, you will merge your branch into the main branch. After the merge, the changes you made to your local branch will be incorporated into the main branch. If the PR conflicts with the existing code, the team member must fix the conflicts before merging.

6. Code Structure

6.1. Core: Upload Images to Server
```

The initial step in ViroTour's workflow is generating a tour by providing the name and description of the tour. The tour is stored in an SQLite database through the use of SQLAlchelmy as an object relational mapper. To initiate this request the parameters are passed into api_add_tour(). This function is decorated with the following end point:

```
@app.route('/api/tour/add', methods=['POST'])
```

Figure 6.1.1 – POST Route to Create Tours.

api_add_tour_images()

When a tour is initialized, the images can be uploaded. During the image upload process, a set of related sequential images are uploaded to the server and are stored in a designated folder. The images within the folder are then associated to a tour of a specific location. In other words, the set of images represents a single location within the tour. The images within the location folder will be stitched together as a single panoramic image. A tour can have multiple locations.

To initiate this request, the name of the tour and a set of image paths are passed into the function. This function is decorated with the following endpoint:

```
@app.route('/api/tour/add/images/<string:tour_name>', methods=['POST'])
```

Figure 6.1.2 – POST Route to Add Images to a Tour.

The tour_name value passed into the function is what is used to query the database for an existing tour. The tour_id returned from the query is then used to generate a new Location record associated to the tour.

```
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
location = Location(tour.id)
db.session.add(location)
db.session.commit()
```

Figure 6.1.3 – Adding New Location to Current Tour.

When the Location record is created, the image upload process iterates through a dictionary of image files and uploads each image into uploads/raw_images/ folder and the new raw image filename is prefixed with the following:

T[tour_id]_L[location_id]_original filename.extension

For example, a file renamed to T1_L1_3456789.jpg would represent an image file that belonged to Tour 1 and Location 1. As a new image is uploaded, the image is automatically associated to a tour and a location within the database.

```
image = Image(tour.id, location.location_id, result[filename])
db.session.add(image)
db.session.commit()
```

Figure 6.1.4 – Image Object Associated to Specific Location and Tour.

When the images are initially uploaded to a location, the state of the images is set to "original". The function returns the tour_id and all the files.

```
location.state = "original"
db.session.commit()
app.logger.info(f'State of location_id {location.location_id} is {location.state}')
```

Figure 6.1.5 – State of Location Set to Original.

api_get_tour_images()

The api_get_tour_images() function returns all raw images in their original form prior to being processed or stitched together to create a panorama. To initiate this request, the name of the tour and location_id is passed to the following function.

This function is decorated with the following endpoint:

```
@app.route('/api/tour/images/raw-images/<string:tour_name>/<int:location_id>', methods=['GET'])
```

Figure 6.1.6 – GET API Route to Retrieve a Tour's Location's Images.

The tour_name and location_id is used to query the database to retrieve all image file paths associated to the tour at the given location.

```
# Get Tour
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
# Get Location
location = db.session.query(Location).filter((Location.tour_id == tour.id) &
                                             (Location.location_id == location_id)).first()
# Get Images
images = db.session.query(Image).filter(Image.location_id == location.location_id).all()
for image in images:
    result.append(image.file_path)
```

Figure 6.1.7 – Retrieving File Paths for a Location Within a Tour.

This function returns all the image file paths.

api_upload_resolve_path()

This is an internal function used to retrieve the absolute path of images stored within the /uploads/ folder and accepts the relative path of an image as a parameter. The "UPLOAD_FOLDER" app.config variable is defined in _init_.py file.

api_set_panoramic_image()

This is an internal function used to save the path of the panoramic image in the database after images are stitched together. This function accepts a tour_name, location_id, and path to the panoramic file as parameters. The tour_name and location_id values are used to query the database to retrieve the location record. The file path of the panoramic is then set.

```
# Get Tour
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
# Get Location
location = db.session.query(Location).filter((Location.tour_id == tour.id) &
                                             (Location.location_id == location_id)).first()
location.pano_file_path = path
db.session.commit()
```

Figure 6.1.8 – Assigning Panoramic Image Path to Location Object.

api_get_panoramic_image()

After api_set_panoramic_image() has been called to store the path to the panoramic file, this function can be used to retrieve the path to the panorama. To initiate this request, the tour_name and location_id values are passed to the function. This function is decorated with the following endpoint:

```
@app.route('/api/tour/images/panoramic-image/<string:tour_name>/<int:location_id>', methods=['GET'])
```

Figure 6.1.9 – GET API Route to Retrieve a Tour's Location's Panoramic Image.

This function accepts a tour_name, location_id as parameters. The tour_name and location_id values are used to query the database to retrieve the location record. The pano_file_path is returned.

```
# Get Tour
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
# Get Location
location = db.session.query(Location).filter((Location.tour_id == tour.id) &
                                             (Location.location_id == location_id)).first()
pano_image = location.pano_file_path
```

Figure 6.1.10 – Assigning Panoramic Image File Path.

api_get_panoramic_images()

This function is used to return all file paths to panoramic images for a given tour. To initiate this request, the tour_name is passed to the function. This function is decorated with the following endpoint:

```
@app.route('/api/tour/images/panoramic-images/<string:tour_name>', methods=['GET'])
```

Figure 6.1.11 – GET API Route to Retrieve All Panoramic Image File Paths in a Tour.

This function accepts a tour_name as a parameter to query the Tour table for the tour_id. The tour_id is then used to filter the Location table for all locations/panoramas associated to the tour. A list of panoramic images is returned.

```
# Get Tour
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
# Get Location
locations = db.session.query(Location).filter((Location.tour_id == tour.id)).all()

pano_images = [{
    'pano_file_path': location.pano_file_path,
    'location_id': location.location_id
} for location in locations]
```

Figure 6.1.12 – Variable to Hold Each Location's Panoramic Image File Path.

api_get_panoramic_image_file()

This function will send the file, as well as contents of the file, over the server using Flask's send_file(). To initiate the request, the tour_name and location_id values are passed to the function. This function is decorated with the following end point:

```
@app.route('/api/tour/images/panoramic-image-file/<string:tour_name>/<int:location_id>', methods=['GET'])
```

Figure 6.1.13 – GET API Route to Retrieve a Specific Location's Panoramic Image.

This function accepts a tour_name, location_id as parameters. The tour_name and location_id is used to query the database to retrieve the location record. The pano_file_path is retrieved and passed into flask's send_file():

```
# Get Tour
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
# Get Location
location = db.session.query(Location).filter((Location.tour_id == tour.id) &
                                             (Location.location_id == location_id)).first()
pano_image_file = api_upload_resolve_path(location.pano_file_path)

return send_file(pano_image_file)
```

Figure 6.1.14 – Passing Panoramic File Path to Server.

6.2. Core: Create Panoramic Image

This code is a function that automatically processes tourist tours. This function extracts the information of a tourist tour from the database and after processing the information, stores the new information in the database. First, the name of the tour is given as input to the function and using the input, we extract the information about the tour from the database. Then, we find all the places that are in this tour. For each location, the compute_panoramic function is used to create a panoramic image. After the image is processed, the panorama output path is saved in the database and the location status is changed to "panorama". Then, this image is stored in the database. Next, we find neighbors for all locations. Here, a function named compute_neighbors is used, which uses an algorithm to find places that are close to each other. These results are stored in the database. In the next step, the images of each place are darkened using the image_blur_faces function, and the pixel images of the faces are removed. After the image is processed, the image is stored in the database. Finally, we check the images related to each location using the image_extract_text function. In the next line, using the query db.session.query(Tour), the desired tour is extracted from the database and filtered with the condition Tour.name == tour_name to find the tour with the desired name in the URL. Then, using first(), the first tour according to the conditions is taken and placed in the tour variable.

```
# Get Tour
tour = db.session.query(Tour).filter(Tour.name == tour_name).first()
```

Figure 6.2.1 – Retrieve Tour That Matches Provided Tour Name in URL.

In the next line, using the query Location.query, all the locations related to the tour are taken and stored in locations. This search is performed using the filter Location.tour_id == tour.id, where tour.id is the tour ID obtained in the previous step. In the next for location in locations: loop, for each location, the compute_panoramic() function is called to calculate the panoramic image. The result of this function is stored in the output_path variable. Then the location status is changed to panoramic and registered in the database using db.session.commit(). Also, the api_set_panoramic_image() function is called to send the panoramic image to the server. In the next for location in locations: loop, for each location, the image_blur_faces() function is called to process the image and hide the face. The result of this function is stored in the output_path variable and the processed image is sent to the server using the api_set_panoramic_image() function.

```
# For all locations, compute panoramic_images image
locations = Location.query.filter(Location.tour_id == tour.id).all()
for location in locations:
    app.logger.info(f"Computing Panoramic for location_id:{location.location_id}")
    output_path = compute_panoramic(tour_name, location.location_id)
    location.state = 'panoramic'
    db.session.commit()
    app.logger.info(f'State of location_id {location.location_id} updated to panoramic')
    api_set_panoramic_image(tour_name, location.location_id, output_path)
```

Figure 6.2.2 – Retrieve Tour That Matches Provided Tour Name in URL.

In the next part of the code, a loop is created over all the touristic places of the tour, in order to calculate the panoramic images for each place. First, using the tour address, all of its tourist locations are retrieved and stored in a list called locations. Then, for each location, the panoramic image is calculated and stored in the output path labeled output_path. Then the location status is changed to panoramic and saved in the database. Then the panoramic image is sent to another service that communicates with this function using the api_set_panoramic_image function. In fact, this function can be used for other services because it completes computational operations related to panoramic images. In the next section, that is, in the "for all locations, compute neighbors" section, neighbors can be calculated using all locations. This operation is done using the compute_neighbors function. The output of this function, hotspot_results, contains information such as what neighbors each location has. In the next section, "for all images, blur faces", all images for panoramic tourist sites are processed using the image_blur_faces function to mask the faces in those images.

```
# For all images, blur faces
for location in locations:
    app.logger.info(f"Computing Blur Faces for location_id:{location.location_id}")
    output_path = image_blur_faces(tour_name, location.location_id)
    api_set_panoramic_image(tour_name, location.location_id, output_path)
```

Figure 6.2.3 – Blur Faces Functionality Applied to Locations' Images.

After calculating the panorama images for all the places, four more stages of information processing are done in different locations where the panorama has been calculated:

Calculating hotspots: In this step, using different tour locations, the places that are most likely to be seen on the tour are identified. Here the compute_neighbors function is used to calculate these values. *For all images, identifying and erasing faces*: In this step, the images of each place are identified and erases the faces in it. For this, the image_blur_faces function is used. *To extract text from images*: In this step, for all images of places, the text contents (the text itself and their respective box coordinates) in them are identified and extracted. The image_extract_text function is used for this. Finally, an empty JSON is returned with a response code of 200.

```
# For all images, extract text
for location in locations:
    app.logger.info(f"Computing Text Extraction for location_id:{location.location_id}")
    hotspot_results = image_extract_text(tour_name, location.location_id)
    api_set_text_search_results(tour.id, location.location_id, hotspot_results)

return jsonify({}), 200
```

Figure 6.2.4 – Apply Text Extraction and Set Search Results for Each Location in a Tour.

Now we have to talk a little about the details of panoramic image production.

In order to create panoramic images, the compute_panoramic function is used for each of the tour locations. In this function, first, using the api_get_tour_images function, a list of images of the desired location for the tour is received and stored in the image_list variable. Then, using the api_get_tour_by_name function, the id of the tour with the name tour_name is received and stored in the tour_id variable. Then the image file paths that are in the image_list list are placed in file_paths. Also, the initial file extension is stored in the file_extension variable. Then the path and name of the output file for the panoramic image is stored in the target_file variable.

```
def compute_panoramic(tour_name, location_id):
    image_list = api_get_tour_images(tour_name, location_id)
    tour_id = api_get_tour_by_name(tour_name)[0].json['id']
    file_paths = [x for x in image_list[0].json['server_file_paths']]
    file_extension = os.path.splitext(file_paths[0])[1]
    target_file = f"panoramic_images/T_{tour_id}_L_{location_id}_pano{file_extension}"
    args = ["--img_names", file_paths, "--output_path", f"{target_file}"]
```

Figure 6.2.5 – Beginning of Creating Panoramic Image Function, Assigning Values.

Next, a list of arguments required for the stitch function is created as a flatlist. In this part, all the values of the args list are placed in the flat list. Finally, the stitch function is called to create the panoramic image using flatlist and the resulting image is returned as output.

```
    flatlist = []
    for element in args:
        if type(element) == str:
            flatlist.append(element)
        else:
            for inner_element in element:
                flatlist.append(inner_element)
    app.logger.info(f'Input to panoramic compute: {flatlist}')

    return stitch.main(flatlist)
```

Figure 6.2.6 – Computing Panoramic Image with Stitching Functionality.

Now let's examine the stitch.

This function uses the OpenCV library. This script combines the rotation model images together into a panorama image. First, an argparse.ArgumentParser() object is created to receive the input arguments in the script execution. The modes variable contains two modes: cv.Stitcher_PANORAMA and cv.Stitcher_SCANS. In the main function, the input arguments are specified, the image files are saved to an img_names list, and the output is saved to the path specified by output_path. If the number of input images is equal to one, this image is considered as a panorama image and is copied to the specified path as output. Otherwise, images are read using OpenCV and rotation is applied to a panoramic image using stitcher. Finally, the combined panorama image is saved in path and the output is returned as panorama image path.

At the beginning of the code, the modules needed to run the code are imported with the help of import. Also, a logger is created. Then, variables named modes and parser have been created. The modes variable is a tuple of two values named cv.Stitcher_PANORAMA and cv.Stitcher_SCANS. The parser variable is an instance of the argparse. The ArgumentParser class is used to process and validate command line arguments. Multiple arguments are defined for the stitch.py command using parser.add_argument. The --img_names and --output_path argument elements take as input the names of the files to merge and the output path of the panorama file, respectively. Then, the main function is defined with the in_args parameter. Said

parameter is actually the command line input used to execute the code. Using parser.parse_args(in_args), the command line input is checked and validated. Then, the names of the files to be merged and the output path of the panorama file are placed in the img_names and output_path variables using args.img_names and args.output_path respectively.

```
args = parser.parse_args(in_args)
img_names = args.img_names
output_path = args.output_path
output_path_resolved = api_upload_resolve_path(output_path)
```

Figure 6.2.7 – Initial Setup for Stitching Functionality.

In this section, if only one image is given as input, instead of merging the images, the input image is copied and output. If the number of images is greater than one, the images are read using the cv.imread function. In the next section, an instance of the cv.Stitcher class is created using the cv.Stitcher.create function and the modes defined in modes.

After reading each image, we add it to the imgs list. Then, a Stitcher of the cv.Stitcher class is created using the mode *mode*. This mode is taken as an input from the user and used to determine stitcher settings. Available modes are PANORAMA and SCANS. In line 31, the images are stitched together using stitcher and in mode. In addition, the pasting status of the images is checked using the status variable. If this value is not equal to cv.Stitcher_OK, an error message is displayed.

```
# If only one image input, then assume it is panoramic already
if len(img_names) == 1:
    shutil.copyfile(api_upload_resolve_path(img_names[0]), output_path_resolved)
    return output_path_resolved
imgs = []
for name in img_names:
    full_path_of_file = api_upload_resolve_path(name)

    full_img = cv.imread(cv.samples.findFile(full_path_of_file))

    if full_img is None:
        app.logger.info("Cannot read image ", name)
        exit()

    imgs.append(full_img)
stitcher = cv.Stitcher.create(args.mode)
```

Figure 6.2.8 – Stitching Images Together Functionality.

The result of pasting images as a panoramic image is placed in the pano variable. The created panorama image is saved in the folder specified using output_path_resolved, with the path of the created panorama sent to be returned.

```
status, pano = stitcher.stitch(imgs)

if status != cv.Stitcher_OK:
    app.logger.info("Can't stitch images, error code = %d" % status)
    return None

app.logger.info("Success! Stitched {}".format(img_names))
cv.imwrite(output_path_resolved, pano)
return output_path_resolved
```

Figure 6.2.9 – Saving Panoramic Image and Returning Its File Path.

6.3. Core: Determining Transition Hotspots

Transition hotspots are indeed an important element in connecting two panoramic images in a virtual tour. A transition hotspot is a clickable area within a panoramic image that triggers a transition to another panoramic image when clicked.

The direction of the hotspot refers to the direction in which the user can navigate to reach the next panoramic image. For example, if the hotspot is located on the right side of the panoramic image, clicking on it would take the user to the next panoramic image located to the right. By detecting transition hotspots through the algorithms described in this section, it is possible to create a virtual tour in which the user can seamlessly navigate from one panoramic image to another, creating a sense of continuity and immersion.

The main transaction hotspots recognition algorithm function is called:

find_hotspot(img_list)

The input parameter for hotspot calculation is an array containing the names of all the image files that will be used in the virtual tour. The array must contain at least two image files for the hotspot calculation to be performed. If the length of the array is less than two, the function will return a null value.

This requirement is necessary because the hotspot calculation involves connecting two or more panoramic images to create a virtual tour. If there are not enough images, it is not possible to create a tour with multiple locations for the user to navigate through.

When this function receives a list of panoramic images, OpenCV will read two of them in order and adjust the image resolution according to the given scale to speed up the processing. (fig. 6.3.1 below). The process of adjusting the image resolution is an important step in the function's performance optimization. By scaling down the image resolution, the function can reduce the computational complexity of subsequent image processing operations, resulting in faster processing times and improved overall performance. In the image queue, all images will be compared with the next image until the last and second-to-last images are compared.

```
hotspot_list = []
spot_img = None
scale = 3

for i in range(0, len(img_list) - 1):

    img1 = cv.imread(img_list[i], cv.IMREAD_GRAYSCALE)
    img2 = cv.imread(img_list[i + 1], cv.IMREAD_GRAYSCALE)

    w1 = img1.shape[0] // scale
    h1 = img1.shape[1] // scale

    w2 = img2.shape[0] // scale
    h2 = img2.shape[1] // scale

    img1_small = cv.resize(img1, (h1, w1))
    img2_small = cv.resize(img2, (h2, w2))

    cv.imwrite('img1_small.jpg', img1_small)
```

Figure 6.3.1 - Read and Optimize Image Resolution.

The feature detection algorithm used in this function, SIFT (Scale-Invariant Feature Transform), is a popular technique for detecting and describing distinctive features in images. SIFT works by identifying scale-invariant keypoints in an image and computing a descriptor for each keypoint based on the image gradients.

The FLANN-based matching algorithm used in this function is a fast approximate nearest neighbor algorithm that can efficiently match feature points between pairs of images. FLANN (Fast Library for Approximate Nearest Neighbors) is a library that provides several algorithms for performing nearest neighbor searches, including KD-trees and hierarchical k-means trees (fig. 6.3.2) (fig. 6.3.3).

```
w = img1_small.shape[0]
h = img1_small.shape[1]

# draw blk image as background
img_bk = np.zeros((w, h, 3), np.uint8)

# Initiate SIFT detector
sift = cv.SIFT_create()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1_small, None)
kp2, des2 = sift.detectAndCompute(img2_small, None)

# FLANN parameters
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)  # or pass empty dictionary
flann = cv.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des1, des2, k=2)
```

Figure 6.3.2 - SIFT / FLANN Matching.

Figure 6.3.3 shows the set of matching points obtained by applying the SIFT algorithm to two images, with lines connecting the points that share the same features.

Figure 6.3.3 - SIFT Matching Results.

Once the matching is complete, the function can use the matched feature points to calculate the region of maximum density within the same range. The center point of this region can represent the hotspot between two images.

```
for k in range(0, len(list_kp1)):
    list_int_kp1.append((int(list_kp1[k][0]), int(list_kp1[k][1])))


for center in list_int_kp1:
    spot_img = cv.circle(img_bk, center, radius=min(h1, w1) // 8, color=(255, 255, 255), thickness=-1)

if spot_img is not None:
    cv.imwrite('hotspot.jpg', spot_img)
    list = caculate_points('hotspot.jpg', scale)
    hotspot_list.append(list)

if i + 1 == len(img_list) - 1:
    list_int_kp2 = []
    for i in range(0, len(list_kp2)):
        list_int_kp2.append((int(list_kp2[i][0]), int(list_kp2[i][1])))

    for center in list_int_kp2:
        spot_img = cv.circle(img_bk, center, radius=min(h1, w1) // 8, color=(255, 255, 255), thickness=-1)

    if spot_img is not None:
        cv.imwrite('hotspot.jpg', spot_img)
        list = caculate_points('hotspot.jpg', scale)
        hotspot_list.append(list)
```

Figure 6.3.4 - Heatmap Algorithm.

Figure 6.3.4 shows the code snippet demonstrating how to calculate the feature distribution image using the matched feature points. This algorithm plots a heatmap based on the density of the feature points and uses the convex hull algorithm to find clusters of high-density feature points. The convex hull is a mathematical technique that finds the smallest convex polygon that contains all of the points in a given set. In this case, the convex hull is used to identify the clusters of feature points that have the highest density. (fig. 6.3.4) (fig. 6.3.5) (fig. 6.3.6).

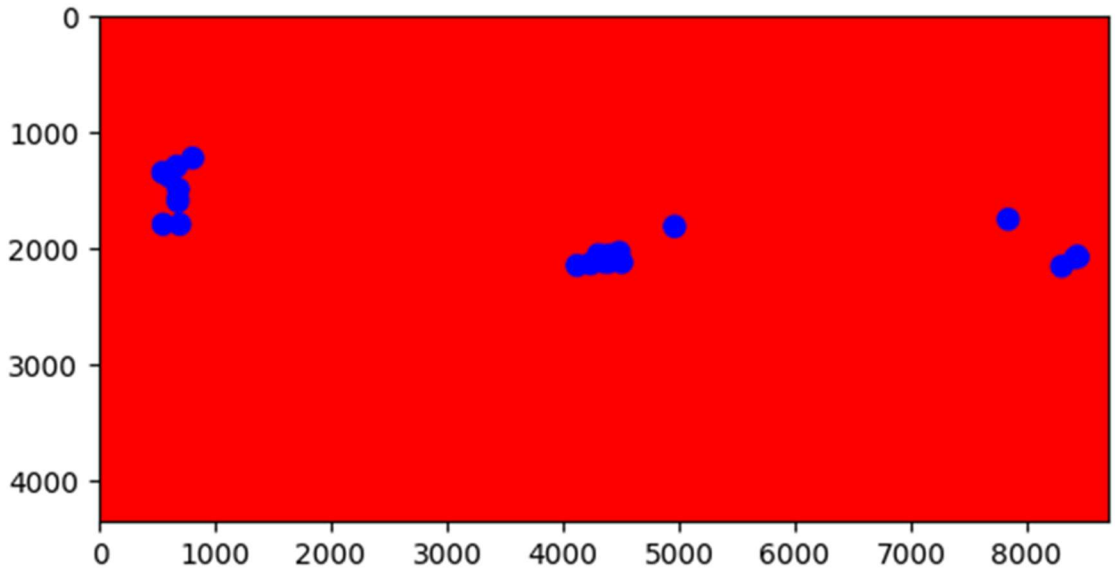

Figure 6.3.5 - Matching Points Results.

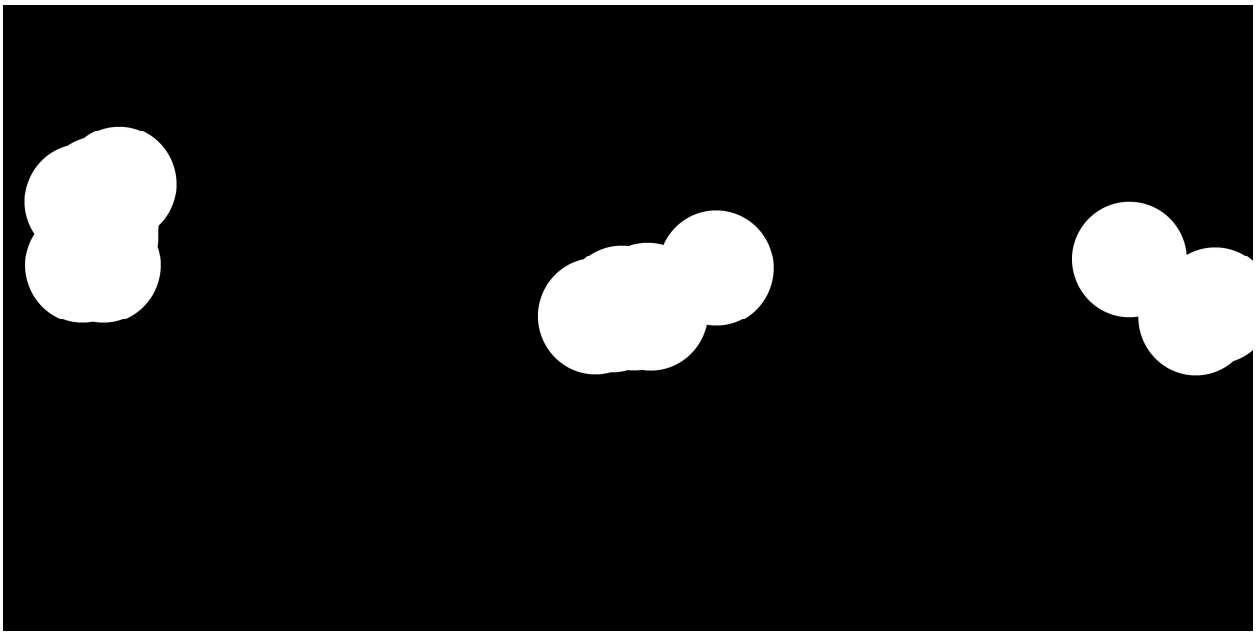

Figure 6.3.6 - Generated Heatmap.

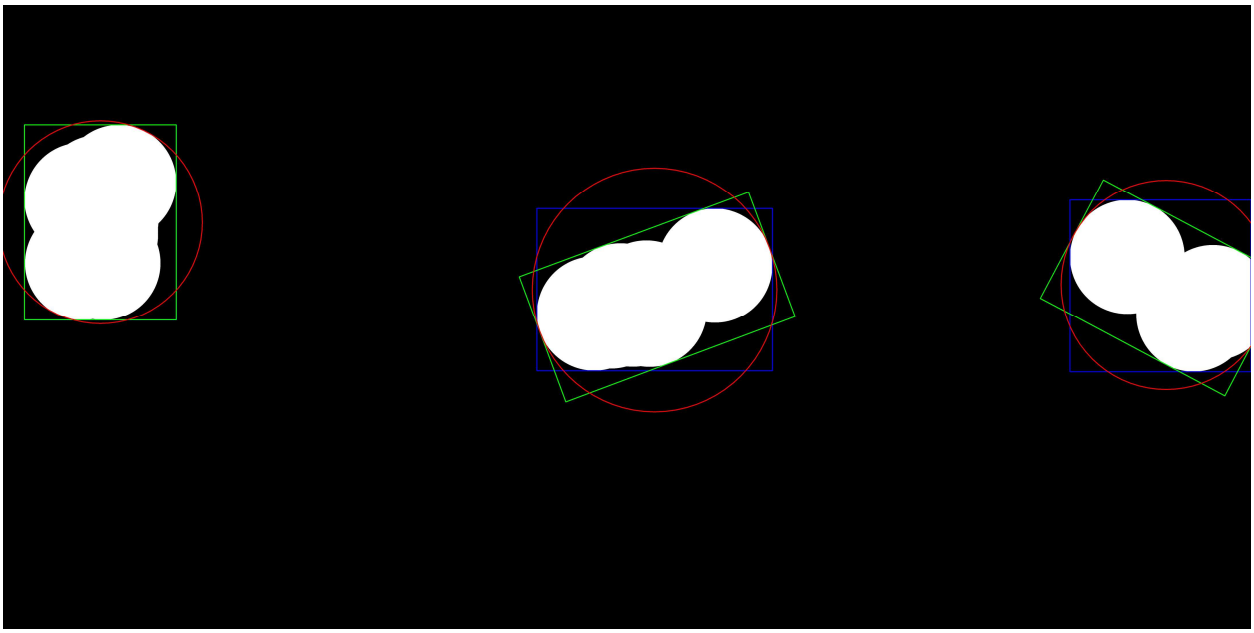

Figure 6.3.7 - Using the Convex Hull Algorithm to Determine the Center Point of the Heatmap.

The algorithm then extracts the center point of each high-density cluster as the hotspot location (fig. 6.3.7). These hotspots represent the areas of the image that are likely to be of interest to the user and can be used to create clickable areas for navigating between images in a virtual tour.

6.4. Editing: Extract Text from Images

When the user creates a panoramic image (as mentioned above) from a given input set of images, the image is analyzed for visible text, storing the text in a table within a managed server. The metadata in question (which pertain to one extracted text object out of many, within one panoramic image) are the following:

- Text_id (integer): unique identifier within the database table
- Location_id (integer): unique identifier to represent a specific location within a tour
- Text_content (string): the extracted text value
- Position_x (integer): the x coordinate of the extracted text object
- Position_y (integer): the y coordinate of the extracted text object
- Position_z (integer): was to-be z coordinate; defaulted to 0 for all text objects for now

```
class Text(db.Model):
    __tablename__ = 'extracted_text_table_v1'

    text_id = db.Column(db.Integer, primary_key=True)
    location_id = db.Column(db.Integer)
    text_content = db.Column(db.String(255))
    position_x = db.Column(db.Integer)
    position_y = db.Column(db.Integer)
    position_z = db.Column(db.Integer)
```

Figure 6.4.1 – Portion of the Text Database Model in the 'models.py' File.

The ViroTour application uses an open-source library called EasyOCR to handle its text extraction. The extraction of text using this library calls for a language of interest and a file path to the image that will be processed for text content. The below snippet of code calls for English language recognition ("en", just one of many language codes the library provides) and utilizes an image file path (the recently processed panoramic image) that will undergo text extraction.

```
def compute_extracted_text_list(image_url):

    listOfExtractedTexts = []
    reader = easyocr.Reader(['en'])
    result = reader.readtext(image_url)
```

Figure 6.4.2 – Initial EasyOCR Implementation in Text Extraction Function.

EasyOCR's response post-extraction yields a list of objects, which includes the four box coordinates that surround the extracted object, the actual text value, and a corresponding percentage value that resembles how accurate the extracted text is. Below is an example of one object from the list of objects that EasyOCR produces:

```
([[5801, 1850], [6112, 1850], [6112, 1923], [5801, 1923]], 'OME CENTER', 0.9998408723923645)
```

Figure 6.4.3 – One Object from EasyOCR Response.

The percentage value for each object represents the validity of the extracted text. In the example above, the found text of “OME CENTER” has a percentage of 0.9998 or 99.98% accuracy which is high. Examples were analyzed and found that filtering the extracted text objects for those only with a 0.05% or higher yielded the most extracted text objects with correct text values.

With the coordinates being used to create an average X and Y position (to simulate a generalized location of the given text object), taking note of the text value, and filtering via validity percentage, the text extraction function receives a list of text objects (the output of the EasyOCR library), and we return a list of objects formatted as below:

```
{'position': {'x': 5956, 'y': 1886, 'z': 0}, 'content': 'OME CENTER'}
{'position': {'x': 7959, 'y': 1907, 'z': 0}, 'content': 'WELCOME'}
{'position': {'x': 2934, 'y': 2043, 'z': 0}, 'content': 'Wekcome Center'}
{'position': {'x': 7959, 'y': 2077, 'z': 0}, 'content': '01 Arbus IMAX Theater'}
{'position': {'x': 7959, 'y': 2152, 'z': 0}, 'content': '~LOWER LEVEL'}
{'position': {'x': 2940, 'y': 2162, 'z': 0}, 'content': 'Airbus IMAX? Theater @1'}
{'position': {'x': 7932, 'y': 2264, 'z': 0}, 'content': ':4 Space Hangar'}
```

Figure 6.4.4 – The Return Format of Text Extraction Function Result (List of Objects).

What we return is a list of objects, where each object has a generalized X and Y position (that corresponds and lies within the computed panoramic image) and the text value/content. Below is EasyOCR in action, with another library called OpenCV that visualizes EasyOCR’s algorithm in action:

Figure 6.4.5 – Visual Display of EasyOCR's Text Extraction using OpenCV CV2's rectangle() and putText() Function.

For a closer look, below is a close-up view of the righthand side of figure 6.4.5:

Figure 6.4.6 – Close-up View of Figure 6.5, Revealing Extracted Text Outline and Text.

The green visualization with the CV2 function is only implemented in a testing branch and not implemented in the actual application. Only the text extraction logic and code (found in figures 6.4.1 to 6.4.4) are implemented, and the extracted text object list is later stored in the database on the server for further reference.

6.5. Editing: Detect and Blur Human Faces

The development process for editing the applied filters of a panoramic image includes the following steps:

1. You will first import the following into your .py file:
    a. os
    b. cv2
    c. api_panoramic_image
    d. api_upload_resolve_path
    e. api_get_tour_by_name

```
import os

import cv2

from virotour import db, app
from virotour.api.image_upload import api_get_panoramic_image, api_upload_resolve_path
from virotour.api.tour import api_get_tour_by_name
from virotour.models import Location

```

Figure 6.5.1 – File Imports.

The uploaded images for creating the panoramic view need to be edited to blur out faces for confidentiality.

2. This can be accomplished using the "image_blur_faces" method, which is called with the tour_name and location_id parameters. The tour_name refers to the name of the tour while the location_id is the identifier of the panoramic location. These parameters are used to obtain the image_path of the panoramic image using the "api_get_panoramic_image" method.

```

def image_blur_faces(tour_name, location_id):
    image_path = api_get_panoramic_image(tour_name, location_id)[0].json['server_file_path']
    if image_path is None:
        return None
    file_extension = os.path.splitext(image_path)[1]
    tour_id = api_get_tour_by_name(tour_name)[0].json['id']
    target_file = f"panoramic_images/T_{tour_id}_L_{location_id}_pano_blurred{file_extension}"
    image_blur_faces_main(image_path, target_file)
    # Get Location
    location = db.session.query(Location).filter(Location.location_id == location_id).first()
    # update state of panoramic to blurred
    location.state = "blurred"
    db.session.commit()
    app.logger.info(f'State of location_id {location_id} updated to blurred')
    return target_file
```

Figure 6.5.2 – Obtaining the Panoramic File Path to Later Blur Faces From.

3. To get the file extension of the panoramic image file, you can use the "os.path.splitext" method. The tour_id can be retrieved using the "api_get_tour_by_name" method. The target_file is then created by combining the tour_id, location_id, and file_extension.

4. The "image_blur_faces_main" method is called with the image_path and target_file variables as parameters. This method detects human faces and applies the blurring effect to the panoramic image and returns the target_file.

```
28  def image_blur_faces_main(image_path, target_file):
29      output_path_resolved = api_upload_resolve_path(target_file)
30      full_path_of_file = api_upload_resolve_path(image_path)
31      full_img = cv2.imread(cv2.samples.findFile(full_path_of_file))
32      detectorPath = getPathToDetectorFile()
33      detector = loadFaceDetector(detectorPath)
34      grayImage = grayscaleImage(full_img)
35      faces = detectFaces(grayImage, detector)
36      result = blurFaces(full_img, faces)
37      cv2.imwrite(output_path_resolved, result)
38      return target_file
```

Figure 6.5.3 – Blurring Faces Functionality.

5. The getPathToDetectorFile() retrieves the path to the haarcascade_frontalface_default.xml file. This is a generated file that is used to detect frontal human faces. It is not meant to be edited manually. The loadFaceDetector(pathToDetector) method loads this detector file to later be used to detect faces.

6. The grayscaleImage(image) turns the image to grayscale and returns it. A grayscale file is needed to detect the faces.

7. The detectFaces(grayImage, detector) is the method that is using the detector to find faces.

```
def detectFaces(grayImage, detector):
    faces = detector.detectMultiScale(grayImage,
                                      scaleFactor=1.05,
                                      minNeighbors=7,
                                      minSize=(30, 30),
                                      flags=cv2.CASCADE_SCALE_IMAGE)
    return faces
```

Figure 6.5.4 – Detecting Faces to Blur.

The values that we may edit to better find faces is scaleFactor, which is the layers of the image zoomed in and out that should be scanned at different scales. Making this value larger causes the processing time to increase and can cause more false positives. The minNeighbors value sets the number of positive face detections that are needed for a face to be detected. Making this value smaller causes the number of false positives to increase. The minSize value sets the minimum size in pixels that a face can be. Smaller faces are ignored. Editing these values is necessary to reduce the number of false positives and find the actual faces that should be blurred.

8. The blurFaces(image, faces) method loops through the detected faces and applies a Gaussian Blur to them.

Figure 6.5.5 – Facial Recognition and Blurring.

6.6. Editing: Apply Filters

When the api_get_panoramic_image_file() function is called, it will retrieve the filter setting stored from the database for a particular location. This filter setting will then be used to adjust the brightness of a panorama by calling the adjust_contrast_brightness() function. The api_get_panoramic_image_file() expects tour_name and location_id as parameters. The function verifies the value of the filter setting. There is control logic to return the original unfiltered image if the filter setting is 0; otherwise adjust_saturation_brightness() is called to modify the image. The modified image is returned.

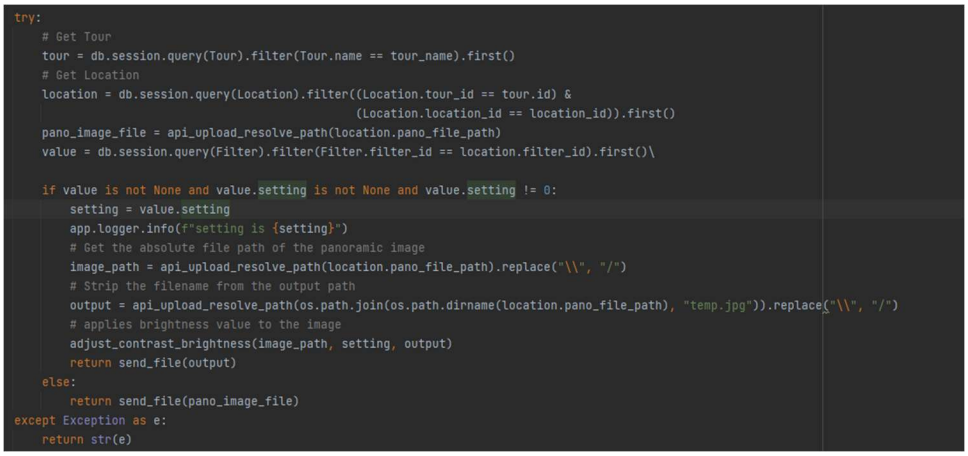

Figure 6.6.1 – api_get_panoramic_image_file.

This function is decorated with this endpoint.

Figure 6.6.2 – Endpoint for api_get_panormaic_image_file).

The tour_name and location_id values are what is used to query the database for location data by specific location id.

The apply_glow_effect() function saves filter settings within the database. When the apply_glow_effect() function is called, the required input parameters it expects are the location_id and brightness value. The location_id is the unique id of the panoramic image. The brightness value is an integer value ranging from 1 to 255. A brightness value of 0 tells the system to reset to the original panoramic prior to applying the filter.

The function will query the Location table for a Location record, matching the location_id that was passed in.

Figure 6.6.3 – Retrieving Location That Matches Provided Location Id (Only One Value).

The Location record will return information regarding the associated tour, panoramic file path, its neighbors, the current state the panoramic image is in (e.g., original, panoramic, blurred), and filter. The Location class model is detailed below.

```python
class Location(db.Model):
    __tablename__ = 'locations_table_v1'

    location_id = db.Column(db.Integer, primary_key=True)
    tour_id = db.Column(db.Integer)
    pano_file_path = db.Column(db.String(255))
    neighbors = db.Column(db.PickleType)
    state = db.Column(db.String(255))
    filter_id = db.Column(db.Integer)

    Ivelin Tchangalov
    def __init__(self, tour_id=None, pano_file_path=None, neighbors=None, state=None, filter_id=None):
        if neighbors is None:
            neighbors = []
        self.tour_id = tour_id
        self.pano_file_path = pano_file_path
        self.neighbors = neighbors
        self.state = state
        self.filter_id = filter_id

    Ivelin Tchangalov
    def __repr__(self):
        return '<Locations %r>' % self.location_id
```

Figure 6.6.4 – Location Object Model.

The apply_glow_effect() function will then proceed to store the brightness value as a filter value within the database and associate the filter to the location.

```python
# store filter value
filter = Filter(brightness)
db.session.add(filter)
db.session.commit()
# Save filter settings of panoramic image
location.filter_id = filter.filter_id
filter.filter_name = 'glow'
db.session.commit()
```

Figure 6.6.5 – Save Filter.

The adjust_contrast_brightness() is the primary method used in applying a glow effect to an image. The input parameters for the adjust_contrast_brightness accepts absolute path to the panoramic image file, brightness value ranging from 1 to 255 and absolute path to panoramic

output. The original panoramic image is not overwritten; instead, a new image is saved in a temporary location. To summarize the functionality of the adjust_contrast_brightness() function it corrects the image according to the input brightness value and saves it in the output path.

The adjust_contrast_brightness() function checks if the image file exists at the given path using the os.path.exists() function. The adjust_contrast_brightness function will then proceed and validate if the brightness input value is positive, the shadow value of the image is equal to the brightness variable value, and the light value is equal to 255. Otherwise, the shadow value is set to 0 and the light value is set to +255 brightness. Using the shadow and light values, the alpha_b and gamma_b parameters are calculated for the cv2.convertScaleAbs() function. The image is modified using these parameters and stored in the adjusted_image variable. Finally, the modified image is saved to the output path using the cv2.imwrite() function, and the output path is returned.

```
def adjust_contrast_brightness(image_path, brightness, output_file):
    if brightness == 0:
        return image_path

    # Check if file exists
    if os.path.exists(image_path):
        img = cv2.imread(image_path)
        dst = img.copy()
        if brightness != 0:
            if brightness > 0:
                shadow = brightness
                highlight = 255
            else:
                shadow = 0
                highlight = 255 + brightness

        alpha_b = (highlight - shadow) / 255
        gamma_b = shadow
        adjusted_image = cv2.convertScaleAbs(img, dst, alpha_b, gamma_b)
        # Write the image to the output path
        cv2.imwrite(output_file, adjusted_image)
        return output_file
```

Figure 6.6.6 – Adjusting Image Quality Based on Brightness Classification.

Below are before and after images of the adjust_contrast_brightness() function executed with a brightness value of 100.

Before	After

Figure 6.6.7 – Glow Effect on an Image.

During the engineering and discovery process for the glow feature, there were other ways to illuminate an image. These methods produced slightly different results and currently are not used but remain in the code base for future reference.

adjust_hue_saturation_value()

When the adjust_hue_saturation_value() function is called, it receives an image with jpg/png image format and three other parameters as input. The input parameters are:

- image_path: the absolute path of the input image file
- brightness: the amount of change of image brightness in numbers
- output: the absolute output path where processed images will be physically stored

By calling this function, it checks whether the input image file exists or not. Using the detect_brightness function, the image brightness is calculated and if it is less than 50%, the image is processed and saved in the output folder.

To process the image, the input image is read using the cv2.imread function. Using the cv2.cvtColor function, the image is converted into HSV color space and the values of three channels (H, S and V) are extracted. To change the brightness of the image, the values of the V channel are changed. For this, the values that are greater than the maximum allowed in terms of brightness (brightness - 255) are converted to 255, and the values that are smaller or equal to this value are increased by the brightness. Using the cv2.merge function, three channels are connected to each other and the image is converted to RGB color space.

Finally, the processed image is saved in the output folder with the same name as the input image, and the output path is returned as the output of the function.

```python
def adjust_hue_saturation_value(image_path, brightness, output):
    # Check if file exists
    if os.path.exists(image_path):
        # Apply filter if mean brightness was below 50%
        filename = os.path.basename(image_path)
        if detect_brightness(image_path) == "dark":
            img = cv2.imread(image_path)
            hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
            h, s, v = cv2.split(hsv)

            lim = 255 - brightness
            v[v > lim] = 255
            v[v <= lim] += brightness

            final_hsv = cv2.merge((h, s, v))
            adjusted_image = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
            # Check if output folder exists
            if not os.path.exists(output):
                os.makedirs(output)
            # Write the image to the output path
            cv2.imwrite(os.path.join(output, f'{filename}'), adjusted_image)
            return os.path.join(output, filename)
```

Figure 6.6.8 – Adjusting Hue and Saturation of an Image.

Below are before and after images when the apply_hue_saturation_value() function was executed with a brightness value of 100.

Before	After
	

Figure 6.6.9 – Hue and Saturation Adjustment on an Image.

apply_gaussian_filter()

The apply_gaussian_filter receives four inputs.

- image_path: the absolute path of the input image file
- radius and strength: the effective parameters of this effect
- output: the address where the result of the new image with the glow effect is placed

First, this function checks if the desired image file exists or not using the os module. If available, it loads the image in RGB color format using the cv2 module and applies an enhancement filter to the image using the GaussianBlur function. With the addWeighted function, it combines the unfiltered image with the filtered image and applies the effective parameters. Next, the function checks whether the output folder exists or not; if it doesn't exist, it creates it.

Finally, the new image with the glow effect is saved in the output path and the output file path of the processed image is returned.

```
def apply_gaussian_filter(image_path, radius, strength, output):
    # Check if file exists
    if os.path.exists(image_path):
        filename = os.path.basename(image_path)
        img = cv2.imread(image_path, cv2.IMREAD_COLOR)
        img_blurred = cv2.GaussianBlur(img, (radius, radius), 1)
        adjusted_image = cv2.addWeighted(img, 1, img_blurred, strength, 0)
        # Check if output folder exists
        if not os.path.exists(output):
            os.makedirs(output)
        # Write the image to the output path
        cv2.imwrite(os.path.join(output, f'{filename}'), adjusted_image)
        return os.path.join(output, filename)
```

Figure 6.6.10 – Gaussian Filter Function.

Below are before and after images when the apply_gaussian_filter() function was executed with radius = 1 and strength = 1.

Before	After

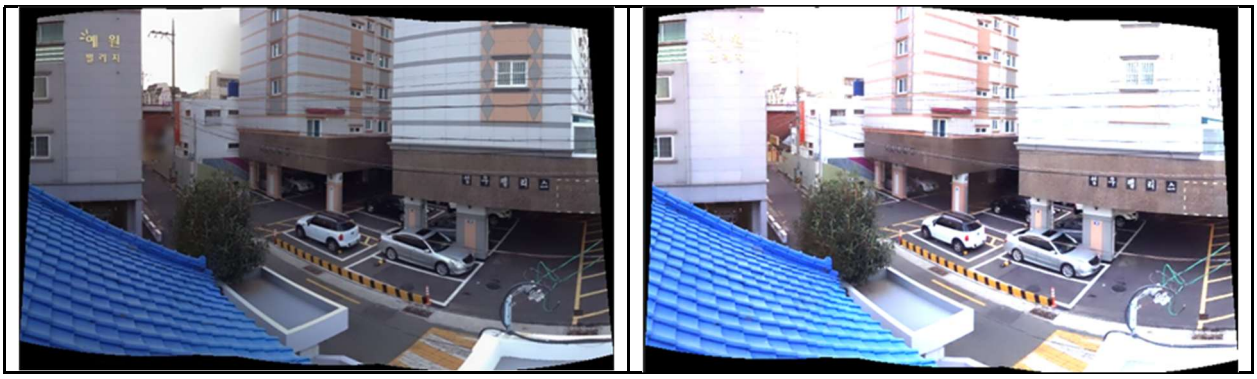

Figure 6.6.9 – Gaussian Filter Effect on an Image.

detect_brightness()

The detect_brightness function receives the absolute path of an image as input, calculates the average brightness of the image, and categorizes the image as dark or light. Within the detect_brightness() function, in the try-except block, the os.path.exists() function checks whether the image file exists at the given path. If the file exists, the image is loaded using the cv2.imread() function and in grayscale format. The average brightness of the image is calculated using the np.mean() function, and its percentage is obtained by dividing by 255 (the maximum value of one pixel). According to the average value, the image is categorized as being either ‘dark’ or ‘light’.

If the average is below 49%, the image is classified as dark; otherwise as light. Information such as the filename, its category, and the percentage of average brightness is recorded in a log. The category of the image is returned as the output of the function. In the case of an error that the image file does not exist, or the image cannot be read successfully for any reason, the function will throw a FileNotFoundError exception, and the execution of the program will terminate.

```
def detect_brightness(image_path):
    try:
        # Check if file exists
        os.path.exists(image_path)
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        filename = os.path.basename(image_path)
        # Calculate mean brightness as percentage
        mean_percent = np.mean(img) * 100 / 255
        classification = "dark" if mean_percent < 49 else "light"
        app.logger.info(f'{filename}: {classification} ({mean_percent:.1f}%)')
        return classification
    except:
        FileNotFoundError
```

Figure 6.6.10 – Determining Brightness Value (Dark or Light) of an Image.

6.7. Search: Search Text

api_set_text_search(tour_name, search_input): return location information for a given search criteria.

The screenshots below demonstrate how to search for text and get the location id of the matches.

```
Ivelin Tchangalov
def get_search_results(client, tour_name, search_input):
    return parse_http_response(get_search_results_with_resp(client, tour_name, search_input))


Ivelin Tchangalov
def get_search_results_with_resp(client, tour_name, search_input):
    return client.get(f'/api/tour/search/{tour_name}/{search_input}')
```

Figure 6.7.1 – API Calls to Get Search Results.

```
# Test that we can search (partial match)
data = get_search_results(client, tour_name, 'level')
assert len(data['results']) == 2
assert data['results'][0]['location_id'] == 2
assert data['results'][1]['location_id'] == 2
assert data['results'][0]['text_content'] == 'LOWER LEVEL'
assert data['results'][1]['text_content'] == 'UPPER LEVEL'
```

Figure 6.7.2 – Test Case: Search for "level" and Get Results Back.

6.8. Editing: Edit Informational Hotspot

One feature of the virtual tours is to allow the tour creator to edit informational hotspots (whether it be to add, remove, or adjust location). These hotspots would be interactable points of interest that provide textual context at a certain location in the tour, manually set by the tour creator(s). Upon creating a new virtual tour, which is composed of several panoramic images (stitched together from 'n' number of regular images of the location in a circular fashion), there is to be zero informational hotspots by default. The vision of this feature would look like the draft below, created at the beginning of the technical discovery:

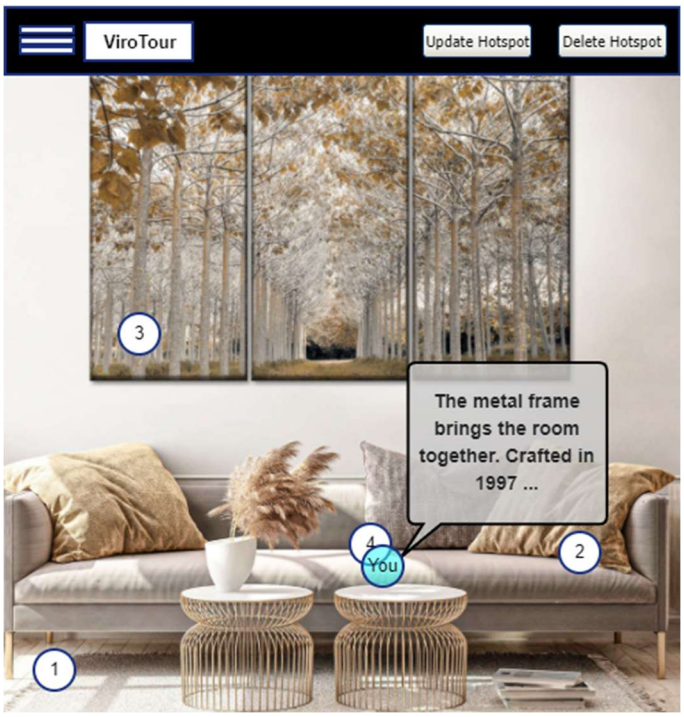

Figure 6.8.1 - UI Draft of Informational Hotspot (Not Actual Application).

Note: This informational hotspot feature (adding, removing, and updating – adjusting location and/or text content) has not yet been implemented and would be items to develop in future iterations of the application. No code mentioned in this section will be official and will be for assumptions on how it would be implemented in future development.

Informational hotspots are to be one of the interactive components for tour viewers, with its data being reliant at one of the computed transitional hotspots. In ViroTour, these transitional hotspots are modeled after the Location object, with each location object to have a reference to its respective (e.g., visible) informational hotspots when the user is at said location:

```python
class Location(db.Model):
    __tablename__ = 'locations_table_v1'

    location_id = db.Column(db.Integer, primary_key=True)
    tour_id = db.Column(db.Integer)
    pano_file_path = db.Column(db.String(255))
    neighbors = db.Column(db.PickleType)
    infoHotspot = db.Column(db.PickleType)
    state = db.Column(db.String(255))
    filter_id = db.Column(db.Integer)

    def __init__(self, tour_id=None, pano_file_path=None, neighbors=None, infoHotspot=None, state=None, filter_id=None):
        if neighbors is None:
            neighbors = []
        self.tour_id = tour_id
        self.pano_file_path = pano_file_path
        self.neighbors = neighbors
        self.infoHotspot = infoHotspot
        self.state = state
        self.filter_id = filter_id
```

Figure 6.8.2 - Location Model and How It Would Incorporate Informational Hotspot (in Red) - Not Officially Implemented.

At any given transitional hotspot / location within a tour view, a reference to a list of informational hotspots would be utilized to display all said informational hotspots (e.g., a clickable marker and textual content to view, like in fig. 6.8.1 above). To implement in future development, the model of the informational hotspot within the database would resemble the following:

- InfoHotSpot_id (integer): unique identifier within the database table
- Tour_id (integer): the id of the tour the informational hotspots was created in
- Location_id (integer): the id of the location (e.g., transitional hotspot) the informational hotspot was created from
- Text_content (string) the text the informational hotspot is presenting to the user at that marker
- Position_x (integer): the x coordinate of the hotspot object
- Position_y (integer): the y coordinate of the hotspot object
- Position_z (integer): was to-be z coordinate; defaulted to 0 for all hotspot objects for now

```
class InfoHotSpot(db.Model):
    __tablename__ = 'infoHotSpot_table_v1'

    infoHotSpot_id = db.Column(db.Integer, primary_key=True)
    tour_id = db.Column(db.Integer)
    location_id = db.Column(db.Integer)
    text_content = db.Column(db.String(8000))
    position_x = db.Column(db.Integer)
    position_y = db.Column(db.Integer)
    position_z = db.Column(db.Integer)
```

Figure 6.8.3 - Example Implementation of Informational Hotspot Model.

As touched upon briefly, the three functions of the feature would be as described below:

- **Adding an informational hotspot (POST route):** being able to add a hotspot at any clickable location, within a given tour, at a computed transitional hotspot/location
- **Deleting an informational hotspot (DELETE route):** should a tour creator want to remove a previously added hotspot, they would be able to click on it and delete it from view and the database
- **Editing an informational hotspot (UPDATE route):** this functionality would include two sub-functions:
    - **Moving the location of the informational hotspot**
    - **Changing the text content associated with the informational hotspot**

The following functions and routes would be implemented as shown below:

- A function that provides (at time of location selection) the informational hotspots that have been generated at said location. If none were created at the selected location, an empty array would be returned and no informational hotspots would be visible at said location.

```python
def set_informational_hotspots(tour_id, location_id):
    """This is an internal call, so there is not a user-facing route."""
    infoHotspots = db.session.query(InfoHotSpot).filter((InfoHotSpot.tour_id == tour_id)
                                                        & (InfoHotSpot.location_id == location_id)).all()

    result = [{
        'position_x': infoHotspot.position_x,
        'position_y': infoHotspot.position_y,
        'position_y': infoHotspot.position_z,
        'text_content': infoHotspot.text_content
    } for infoHotspot in infoHotspots]

    db.session.commit()
    return result
```

Figure 6.8.4 - Function That Sets Up Hotspots When New Location is Navigated to in a Given Tour.

- An API route that provides an array of informational hotspots' metadata for a specified location within a specified tour. The metadata would include the x-y-z coordinates of the hotspot within the panoramic image/location, as well as the manually set text the tour creator added to said hotspot.

```python
@app.route('/api/tour/<int:tour_id>/location/<int:location_id>/infohotspots', methods=['GET'])
def get_informational_hotspots(tour_id, location_id):
    """
        Retrieve informational hotspots at current location inside current tour.
    """
    infoHotspots = db.session.query(InfoHotSpot).filter((InfoHotSpot.tour_id == tour_id)
                                                        & (InfoHotSpot.location_id == location_id)).all()

    result = [{
        'position_x': infoHotspot.position_x,
        'position_y': infoHotspot.position_y,
        'position_y': infoHotspot.position_z,
        'text_content': infoHotspot.text_content
    } for infoHotspot in infoHotspots]
    payload = {
        'informational_hotspots': result
    }
    return jsonify(payload), 200
```

Figure 6.8.5 - GET API Route to Retrieve All Hotspots at a Given Location in a Tour.

- An API route that allows an informational hotspot to be added to a specified location within a specified tour.

```python
@app.route('/api/tour/<int:tour_id>/location/<int:location_id>/infohotspots/add', methods=['POST'])
def api_add_informational_hotspot(tour_id, location_id):
    """
        Add informational hotspot to current location in current tour.
    """
    if request.is_json:
        data = request.get_json()
        infoHotspot = InfoHotSpot(tour_id=tour_id,
                                  location_id=location_id,
                                  text_content=data['text_content'],
                                  position_x=data['position_x'],
                                  position_y=data['position_y'],
                                  position_z=data['position_z'])
        db.session.add(infoHotspot)
        db.session.commit()
        payload = {
            'message': f'Informational hotspot has been created successfully.'
        }
        return jsonify(payload), 201
    else:
        payload = {
            'error': 'The request payload is not JSON format.'
        }
        return jsonify(payload), 404
```

Figure 6.8.6 - POST API Route to Add a Hotspot to a Given Location in a Tour.

- An API route that allows a user to update an informational hotspot at a specified location within a specified tour. The update would include both the position coordinates (position_x, position_y, and position_z) and the text_content properties.

```python
@app.route('/api/tour/location/infohotspot/update/<int:infoHotSpot_id>', methods=['POST'])
def api_update_informational_hotspot(infoHotSpot_id):
    """
        Update informational hotspot by id.
    """
    if request.is_json:
        data = request.get_json()
        infoHotspot = InfoHotSpot.query.get_or_404(infoHotSpot_id)
        infoHotspot.text_content = data['text_content']
        infoHotspot.position_x = data['position_x']
        infoHotspot.position_y = data['position_y']
        infoHotspot.position_z = data['position_z']
        db.session.commit()
        payload = {
            'message': f'Informational hotspot has been updated successfully.'
        }
        return jsonify(payload), 200
    else:
        payload = {
            'error': 'The request payload is not JSON format.'
        }
        return jsonify(payload), 404
```

Figure 6.8.7 - POST API Route to Update a Hotspot at a Selected Location in a Tour.

- An API route to delete an informational hotspot at a specified location within a specified tour.

```python
@app.route('/api/tour/location/infohotspot/delete/<int:infoHotspot_id>', methods=['DELETE'])
def api_delete_informational_hotspot(infoHotspot_id):
    """
        Delete informational hotspot by id.
    """
    infoHotspot = InfoHotSpot.query.get_or_404(infoHotspot_id)
    db.session.delete(infoHotspot)
    db.session.commit()
    payload = {
        'message': f'Informational hotspot was successfully deleted.',
    }
    return jsonify(payload), 200
```

Figure 6.8.8 - DELETE API Route to Delete a Hotspot from a Selected Location in a Tour.

7. Project File Structure

The structure of the directory and files can be as follows:

- Main directory of the project: The main directory of the project is placed as the root of the file structure, and this directory is created with the name of the project.

virotour/

- The app directory: The app directory contains all the files related to the Flask program and its various modules.
- app/api directory: The api directory contains all the files related to the application API.

project_name/

├── app/

│ ├── api/

- app/api/__init__.py file: This file is one of the main files for the api module that must exist. In this file, we can import the required modules for the API and configure our program.

virotour /

├── app/

│ ├── api/

│ │ ├── init.py

- app/api/views.py file: This file contains all the views related to the API that must be defined in this module.

virotour /

├── app/

│ ├── api/

│ │ ├── init.py

│ │ ├── views.py

- app/templates directory: The templates directory contains all the HTML files that are used to provide output to the application's users.

virotour /

├── app/

│ ├── templates/

- app/static directory: The static directory contains all static files, such as CSS files, JavaScript, images, and other files related to the design of pages and programs.

virotour /

├── app/

│ ├── init.py

│ ├── api/

│ │ ├── init.py

│ │ ├── views.py

│ ├── templates/

│ └── static/

- run.py file: This file is used to run the program. In this file, we first create the Flask application and then run it according to the settings configured in the init.py file.

virotour /

```
├── app/
│   ├── init.py
│   ├── api/
│   │   ├── init.py
│   │   ├── views.py
│   ├── templates/
│   └── static/
├── run.py
```

- requirements.txt file: This file contains all program dependencies that must be installed using Python's dependency management tool (such as pip).

```
virotour /
├── app/
│   ├── init.py
│   ├── api/
│   │   ├── init.py
│   │   ├── views.py
│   ├── templates/
│   └── static/
├── run.py
└── requirements.txt
```

- Tests directory: The tests directory contains all unit tests and integral tests of the program.

```
virotour /
├── app/
│   ├── init.py
│   ├── api/
│   │   ├── init.py
```

```
│   │   ├── views.py
│   ├── templates/
│   └── static/
├── run.py
├── requirements.txt
└── tests/
```

- tests/__init__.py file: This file is one of the main files for the tests module that must exist.

```
virotour /
├── app/
│   ├── init.py
│   ├── api/
│   │   ├── init.py
│   │   ├── views.py
│   ├── templates/
│   └── static/
├── run.py
├── requirements.txt
```

- **__init__.py:** This file is an empty file that tells Python that this directory should be considered as a package. This file can be used to initialize any configuration or set up that needs to be done before the package is used.
- **auth.py:** This file contains code for user authentication and authorization. It defines functions for registering a new user, logging in, and generating access tokens. The functions are used in the Flask routes to restrict access to certain endpoints and to verify the identity of the user making the request.
- **errors.py:** This file contains code for handling errors and exceptions that can occur during the execution of the API. The file defines custom error classes that inherit from the Exception class and are raised when certain errors occur. The Flask application then handles these errors and returns appropriate error responses to the client.
- **routes.py:** This file contains the Flask routes for the API. It defines the endpoints that clients can use to interact with the server. Each endpoint is associated with a URL path

and HTTP method, and the functions that handle the requests and responses are defined in this file.

- **utils.py:** This file contains utility functions that are used by other modules in the API. These functions can be used for common tasks such as parsing request parameters, generating JSON responses, and validating user inputs.

Compute module structure:

compute

├── __init__.py

├── models.py

├── schemas.py

├── service.py

└── utils.py

└── neighbor_util

└── pano_util

└── text_util

- **__init__.py:** This file is an empty file that tells Python that this directory should be considered as a package. This file can be used to initialize any configuration or set up that needs to be done before the package is used.
- **models.py:** This file contains the data models used in the compute module. These models are usually database tables and define the structure of the data that will be stored in the database. The models are defined using SQLAlchemy, a popular ORM (Object-Relational Mapping) library for Python.
- **schemas.py:** This file contains the data schemas used in the compute module. These schemas define how the data will be serialized and deserialized when it is sent over the network. The schemas are defined using Marshmallow, a popular serialization/deserialization library for Python.
- **service.py:** This file contains the business logic of the compute module. It defines functions for performing calculations or computations, based on the data stored in the database. The functions defined in this file are usually called from the Flask routes defined in the routes.py file.
- **utils.py:** This file contains utility functions that are used by other modules in the compute module. These functions can be used for common tasks such as parsing request parameters, generating JSON responses, and validating user inputs.

Overall, this file structure is typical of a Flask project's module that handles computations or calculations based on data stored in a database. The models.py file defines the structure of the data to be stored in the database, the schemas.py file defines how the data will be serialized and

deserialized when it is sent over the network, the service.py file contains the business logic for performing calculations based on the data in the database, and the utils.py file contains utility functions that are used by the other modules in the compute module.

Neighbor module structure:

neighbor_util

```
├── __init__.py
├── data
│   ├── __init__.py
│   ├── neighborhood_geodata.geojson
│   └── neighborhood_geodata.json
├── models.py
├── schemas.py
├── service.py
└── utils.py
```

- **init__.py:** This file is an empty file that tells Python that this directory should be considered as a package. This file can be used to initialize any configuration or set up that needs to be done before the package is used.
- **data folder:** This folder contains the data files used in the neighbor_util module. The __init__.py file in this folder makes the directory a package.
- **neighborhood_geodata.geojson:** This file contains geographic data for neighborhoods in a certain area. The data is in the GeoJSON format, which is a popular format for encoding geographic data structures.
- **models.py:** This file contains the data models used in the neighbor_util module. These models are usually database tables and define the structure of the data that will be stored in the database. The models are defined using SQLAlchemy, a popular ORM (Object-Relational Mapping) library for Python.
- **schemas.py:** This file contains the data schemas used in the neighbor_util module. These schemas define how the data will be serialized and deserialized when it is sent over the network. The schemas are defined using Marshmallow, a popular serialization/deserialization library for Python.
- **service.py:** This file contains the business logic of the neighbor_util module. It defines functions for performing operations on the neighborhood data, such as finding the closest neighborhood to a given point. The functions defined in this file are usually called from the Flask routes defined in the routes.py file.

- **utils.py:** This file contains utility functions that are used by other modules in the neighbor_util module. These functions can be used for common tasks such as parsing request parameters, generating JSON responses, and validating users.

8. Data and Backend

SQL Alchemy is an ORM (object-relational mapper) that allows python class objects to be mapped to database tables in SQLite. Data is stateless and persists through the application by a construct known as the "session." The session is what establishes a connection to the database. CRUD (Create-Retrieve-Update-Delete) operations are accessible and executed throughout the session. Data manipulations and other database transactions are not stored until they have been committed. The name and path of the SQLite database is defined in _init_.py. (SQLALCHEMY 2.0 documentation, n.d.)

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(basedir, 'app.db')
```

Figure 9.1 - Configuration Handling.

The following binds an instance of the Flask application to SQLAlchemy:

```
db = SQLAlchemy(app)
```

Figure 9.2 - Creating Flask Application Instance.

When the Flask application runs, run.py calls the following statement to generate all of the objects that are defined in models.py:

```
db.create_all()
```

Figure 9.3 - Creating All Object Instances Within the Database.

Our class objects are stored in models.py. Each of the classes contain an _init_() function. This is the constructor and initializes an instance of the class object. The _repr_() function is used to return a description of the class.

The Tour class stores the unique identifier for the tour, name of the tour, and a description.

```
class Tour(db.Model):
    __tablename__ = 'tour_table_v1'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    description = db.Column(db.String(255))

    Ivelin Tchangalov +1
    def __init__(self, name=None, description=None):
        self.name = name
        self.description = description

    Ivelin Tchangalov
    def __repr__(self):
        return '<Tour %r>' % self.name
```

Figure 9.4 - Tour Object Model.

A single tour can contain multiple panoramic images. Within our data model, a panoramic image is the same as a location. The Location class stores a unique identifier for the location, the associated tour id, the file path to the panoramic image, the transitional hotspots, the state that the current panoramic image/location is in and any filter settings that were applied to the panoramic image. The states that the panoramic can undergo are original, panoramic, and blurred. When the location is in the “original” state, images have not been stitched to form a panorama and are in their raw state. When the location is in the “panoramic” state, the image has been stitched. When the image is in ‘blurred” state, the blurred algorithm has been executed on the panorama.

```python
class Location(db.Model):
    __tablename__ = 'locations_table_v1'

    location_id = db.Column(db.Integer, primary_key=True)
    tour_id = db.Column(db.Integer)
    pano_file_path = db.Column(db.String(255))
    neighbors = db.Column(db.PickleType)
    state = db.Column(db.String(255))
    filter_id = db.Column(db.Integer)

    Viet Nguyen +1
    def __init__(self, tour_id=None, pano_file_path=None, neighbors=None, state=None, filter_id=None):
        if neighbors is None:
            neighbors = []
        self.tour_id = tour_id
        self.pano_file_path = pano_file_path
        self.neighbors = neighbors
        self.state = state
        self.filter_id = filter_id

    Ivelin Tchangalov
    def __repr__(self):
        return '<Locations %r>' % self.location_id
```

Figure 9.5 - Location Object Model.

The image class stores the raw images that were uploaded to our server. A single location can contain multiple images. The image class stores the unique identifier for the image, the associated tour id, the associated location id, and the file path of the image.

```python
class Image(db.Model):
    __tablename__ = 'images_table_v1'

    image_id = db.Column(db.Integer, primary_key=True)
    tour_id = db.Column(db.Integer)
    location_id = db.Column(db.Integer)
    file_path = db.Column(db.String(255))

    Ivelin Tchangalov +1
    def __init__(self, tour_id=None, location_id=None, file_path=None):
        self.tour_id = tour_id
        self.location_id = location_id
        self.file_path = file_path

    Ivelin Tchangalov
    def __repr__(self):
        return '<Images %r>' % self.image_id
```

Figure 9.6 - Image Object Model.

The Text class stores the unique identifier of the text, the associated tour id, the associated location id, text content, and coordinates that were extracted from the text extraction algorithm.

```
class Text(db.Model):
    __tablename__ = 'extracted_text_table_v1'

    text_id = db.Column(db.Integer, primary_key=True)
    tour_id = db.Column(db.Integer)
    location_id = db.Column(db.Integer)
    text_content = db.Column(db.String(8000))
    position_x = db.Column(db.Integer)
    position_y = db.Column(db.Integer)
    position_z = db.Column(db.Integer)

    Ivelin Tchangalov +1
    def __init__(self, tour_id, location_id=None, text_content=None, position_x=None, position_y=None, position_z=None):
        self.tour_id = tour_id
        self.location_id = location_id
        self.text_content = text_content
        self.position_x = position_x
        self.position_y = position_y
        self.position_z = position_z

    Ivelin Tchangalov
    def __repr__(self):
        return '<Text %r>' % self.text_id
```

Figure 9.7 - Text Object Model.

As panoramic images are processed, different filters can be applied to enhance a panorama. Currently, the Filter class is used to store the glow effect filter and its setting.

```
class Filter(db.Model):
    __tablename__ = 'filter_table_v1'

    filter_id = db.Column(db.Integer, primary_key=True)
    filter_name = db.Column(db.String(255))
    setting = db.Column(db.Integer)

    Viet Nguyen
    def __init__(self, setting=None):
        self.setting = setting

    Viet Nguyen
    def __repr__(self):
        return '<States %r>' % self.filter_id
```

Figure 9.8 - Filter Object Model.

9. Publishing

Please refer to the Deployment and Operations Guide for instructions on how the ViroTour Server Processor is deployed and operated.