COMP SCI 7098 Master of Computing & Innovation Project
2021 Semester 1

Project: A Map Management Platform for Self-Driving Cars

Technical Specifications Document

Version 1.0

Revision History

Date	Version	Description	Author
15/May/21	0.1	Document creation	Nhu Quynh Hoa
25/May/21	0.2	Document amendment: 1. Introduction, 2. System Overview, 3.2. Backend – Django, 3.3. Database, 3.4. Django REST Framework, 3.5. Django REST API – React Integration 3.8. Frontend – Django & React	Nhu Quynh Hoa
27/May/21	0.3	Document amendment: 3.1. Connecting to Mapillary, 3.6. Mapbox, 3.7. Deck.gl, 4.1.1. Fetching users and sequences from Mapillary, 4.1.3. Scheduler, 4.2.2. Direction calculation, 4.3.1. Select a region, 4.3.3. Zoom and layer interactions 4.3.6. Select a date range	Jonhatan Cotes
28/May/21	0.4	Document amendment: 4.3.2. Choose different layer types, 4.3.3. Select a position object, 4.3.5. Interact with sliding image frame 4.3.7. Loading	Huu Thanh Nguyen
31/May/21	0.5	Document amendment: 4.1.2. Downloading images from Mapillary, 4.2.1. Nearest neighbors	Aryaman Dhawan
10/Jun/21	1.0	Document finalisation	Nhu Quynh Hoa

Table of Contents

1. Introduction 5

1.1 Purpose of this document 5
1.2 References 5

2. System Overview 5
3. System Configurations 7

3.1 Connecting to Mapillary 7
3.2 Backend - Django 7

3.2.1 URL Patterns 7
3.2.2 Views 8
3.2.3 Models 8
3.2.4 Templates 11

3.3 Database 11
3.4 Django REST Framework 11

3.4.1 Serializers 11
3.4.2 API Views 12

3.5 Django REST API – React Integration 15

3.5.1 Axios 15
3.5.2 webpack 17
3.5.3 Babel 18

3.6 Mapbox 18
3.7 Deck.gl 19
3.8 Frontend – Django & React 20

3.8.1 URL Patterns 20
3.8.2 Views 20
3.8.3 Templates 21

4. System Requirements 21

4.1 Mapillary API – Django integration 21

4.1.1 Fetching users and sequences from Mapillary 21
4.1.2 Downloading images from Mapillary 23
4.1.3 Scheduler 25

4.2 Backend – Django views 26

4.2.1 Nearest neighbors 26
4.2.2 Direction calculation 31

4.3 Frontend – React 33

4.3.1 Select a region 33
4.3.2 Choose different layer types 35
4.3.3 Zoom and layer interactions 36
4.3.4 Select a position object 38
4.3.5 Interact with sliding image frame 40
4.3.6 Select a date range 42
4.3.7 Loading 43

Technical Specifications Document

1. Introduction

1.1 Purpose of this document

The technical specifications document provides a list of technical requirements that are addressed by the project "A Map Management Platform for Self-Driving Cars". The project is carried out by a group of 4 students (Aryaman Dhawan, Huu Thanh Nguyen, Jonhatan Cotes Calderon, Nhu Quynh Hoa) taking the course COMP SCI 7098 Master of Computing & Innovation Project during semester 1, 2021 at The University of Adelaide.

1.2 References

Reference	Document/Link
axios	https://axios-http.com/docs/intro
Babel	https://babeljs.io/docs/en/
Deck.gl	https://deck.gl/docs
Django	https://docs.djangoproject.com/en/3.2/
Django REST framework	https://www.django-rest-framework.org/
Mapbox	https://docs.mapbox.com/
Mapillary API	https://www.mapillary.com/developer/api-documentation/
React	https://reactjs.org/docs/getting-started.html
webpack	https://webpack.js.org/

2. System Overview

The system is a web-based application of map management for self-driving cars that can be continuously updated to represent a changing environment. The platform uses an SQLite database that contains geographic data and images downloaded from Mapillary, a crowdsourced image service. The data is processed using Quadtree based algorithms in the backend to identify all the nearest neighbor images having the same direction. The backend, which uses Django, renders the data to Django REST API from which the frontend using React consumes and processes the data. The frontend also uses Mapbox to render the map and Deck.gl to visualise layers.

Figure 1 below shows the solution architecture of the application.

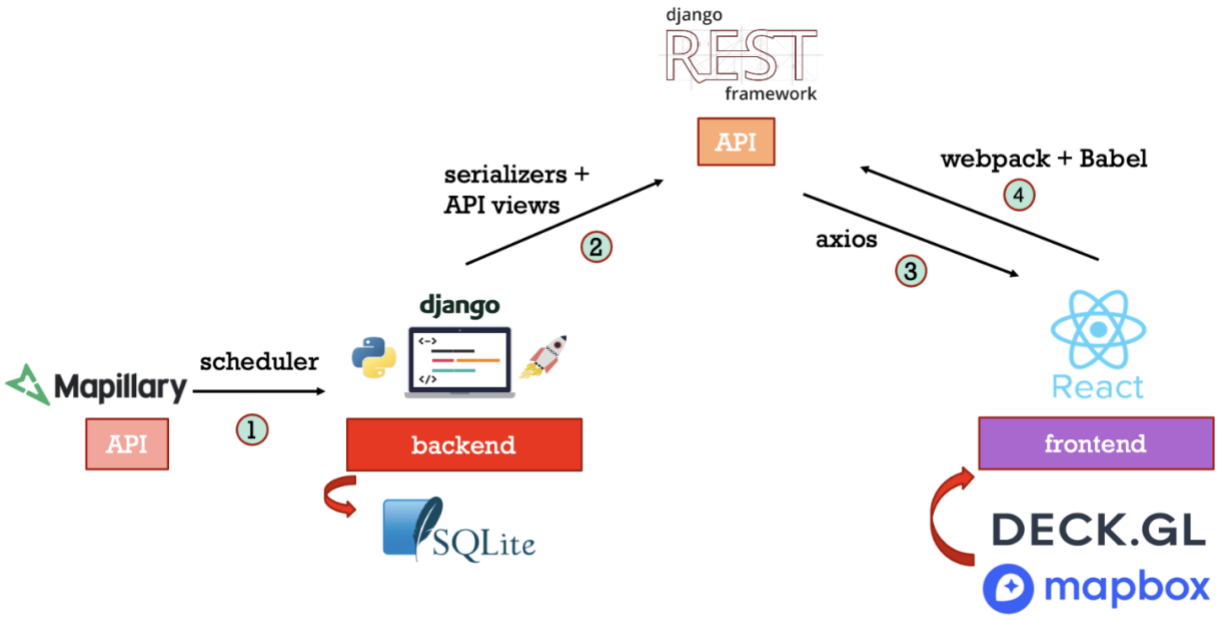


Figure 1. Solution architecture of the application

For Django web framework, the underlying architecture, or conceptual structure is MVC, or Model-View-Controller software design pattern. However, Django uses some different names for these, which are URL patterns, views, models, and templates.

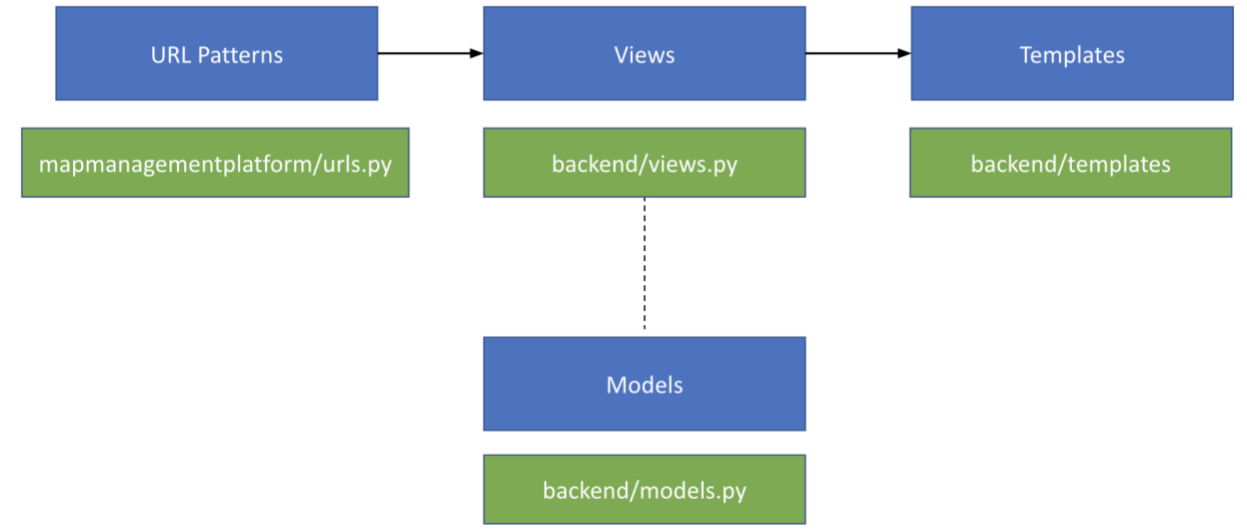


Figure 2. Django MVC architecture of the application

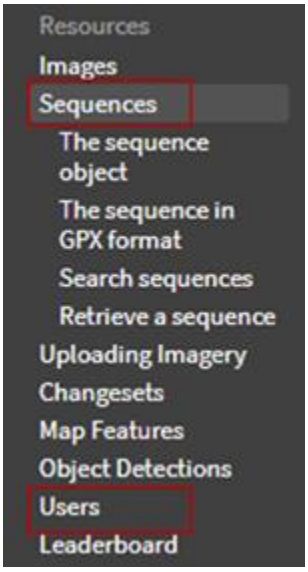
3. System Configurations

3.1 Connecting to Mapillary

To connect to Mapillary the web application needs to have a Client id as stated in Mapillary's guide (Accessing imagery and data through the Mapillary API – Mapillary). The application currently uses the following data for the connection but that can be changed in the *backend/views.py*:

- Client id: dm95M3VFaHJkZ2dDS1RDZzlaVXN1RjpkMDU4NTU5ZDdhMDM4MmZi

To integrate with the API we followed the documentation available in Mapillary API Documentation - Version 3. Our integration only uses the users and sequences, the details of the information fetched are detailed in *4.1.1 Fetching users and sequences from Mapillary*



3.2 Backend - Django

3.2.1 URL Patterns

When a Django application receives a web request, it uses the URL patterns to decide which view to pass the request to for handling. In the project, the URL patterns are defined in *mapmanagementplatform/urls.py*.

If a request comes into Django with a root path of just a slash, Django will first look at the URL patterns to find in *urls.py* to find what view it should use. The URL patterns will then route to *frontend.urls*, which will be explained in detail in section *3.8. Frontend – Django & React*.

```
urlpatterns = [
    # path('', include('backend.urls')), #home path says if you land at 'http://127.0.0.1:8000/' take you to frontend
    path('', include('frontend.urls')),
    path('admin/', admin.site.urls),
    # path('backend', views.home, name='backend'),
    path('response', views.response, name='response'),
    path('users', views.users, name='users'),
    path('sequences', views.sequences, name='sequences'),
    path('neighbors', views.neighbors, name='neighbors'),
    path('sanitydirection', views.sanitydirection, name='sanitydirection'),
    #path('mapillaryupdate', views.mapillaryupdate, name='mapillaryupdate'),
    path('deltaneighbors', views.deltaneighbors, name='deltaneighbors'),
    # path('index', views.index, name='index'),
    path('api/coordinates/', api_views.CoordinatePropertyView.as_view()),
    path('api/regions/', api_views.RegionView.as_view()),
    path('api/sequences/', api_views.SequenceView.as_view()),
]
```

3.2.2 Views

Views provide the logic or control flow portion of the project. A view is a Python callable, such as a function that takes an HTTP request as an argument and returns an HTTP response for the webserver to return. Our views are defined at *backend/views.py*.

3.2.3 Models

To perform queries against the database, each view can leverage Django models as needed. We define our models for the backend app in *backend/models.py*. A Django model is a class with attributes that define a database table's schema or underlying structure. These classes provide built-in methods for making queries on the associated tables.

There are 4 models defined for the project: Region, User (for Mapillary users), Sequence, and Coordinate_property.

```python
class Region(models.Model):
    name = models.CharField(max_length=200) #Region name
    min_longitude = models.FloatField() # Minimum Long of the image
    min_latitude = models.FloatField() # Minimum Latitude of the image
    max_longitude = models.FloatField() # Minimum Long of the image
    max_latitude = models.FloatField() # Maximum Latitude of the image
    view_longitude = models.FloatField() # View of the Region in the map
    view_latitude = models.FloatField() # View of the Region in the map
    last_update = models.DateTimeField(null=True, editable=False) # When the region was last updated
    stored_at = models.DateTimeField() # When data was stored in database

    def __str__(self):
        return self.name

class User(models.Model):
    about = models.TextField(blank=True) # A short description about the user
    avatar = models.CharField(max_length=1000) # URL to the user's profile photo
    created_at = models.DateTimeField() # When user joined Mapillary
    key = models.CharField(unique=True, max_length=200) # Unique identifier of the user
    username = models.CharField(max_length=200) # Login username
    stored_at = models.DateTimeField() # When data was stored in database

    def __str__(self):
        return self.username

class Sequence(models.Model):
    camera_make = models.CharField(max_length=200) # Camera manufacture
    captured_at = models.DateTimeField() # When sequence was captured
    created_at = models.DateTimeField() # When sequence was uploaded
    key = models.CharField(unique = True, max_length=200) # Unique Identifier of the sequence
    pano = models.BooleanField() # Whether the sequence is panorama or not
    user_key = models.ForeignKey(User, on_delete=models.CASCADE) # User who captured the sequence; 1 user - many sequences
    region_key = models.ForeignKey(Region, on_delete=models.CASCADE) #Region where the sequence is stored
    stored_at = models.DateTimeField() # When data was stored in database
    points_in_seq = models.IntegerField() # Stores how many points are in the sequence
    def __str__(self):
        return self.key

class Coordinate_property(models.Model):
    ca = models.FloatField(null=True) # Camera angles either in [0, 360) degrees, or -1 which indicates the corresponding CA is missing
    image_key = models.CharField(unique = True, max_length=200) # Image key for coordinates
    sequence_key = models.ForeignKey(Sequence, on_delete=models.CASCADE) # Identifier of the sequence
    longitude = models.FloatField(null=True) # Longitude of the image
    latitude = models.FloatField(null=True) # Latitude of the image
    direction = models.FloatField(null=True) #Direction angles either in [0, 360) degrees, generated by algorithm
```

Together with other standard Django models, we have the UML class diagram from Django models for the project as follows:

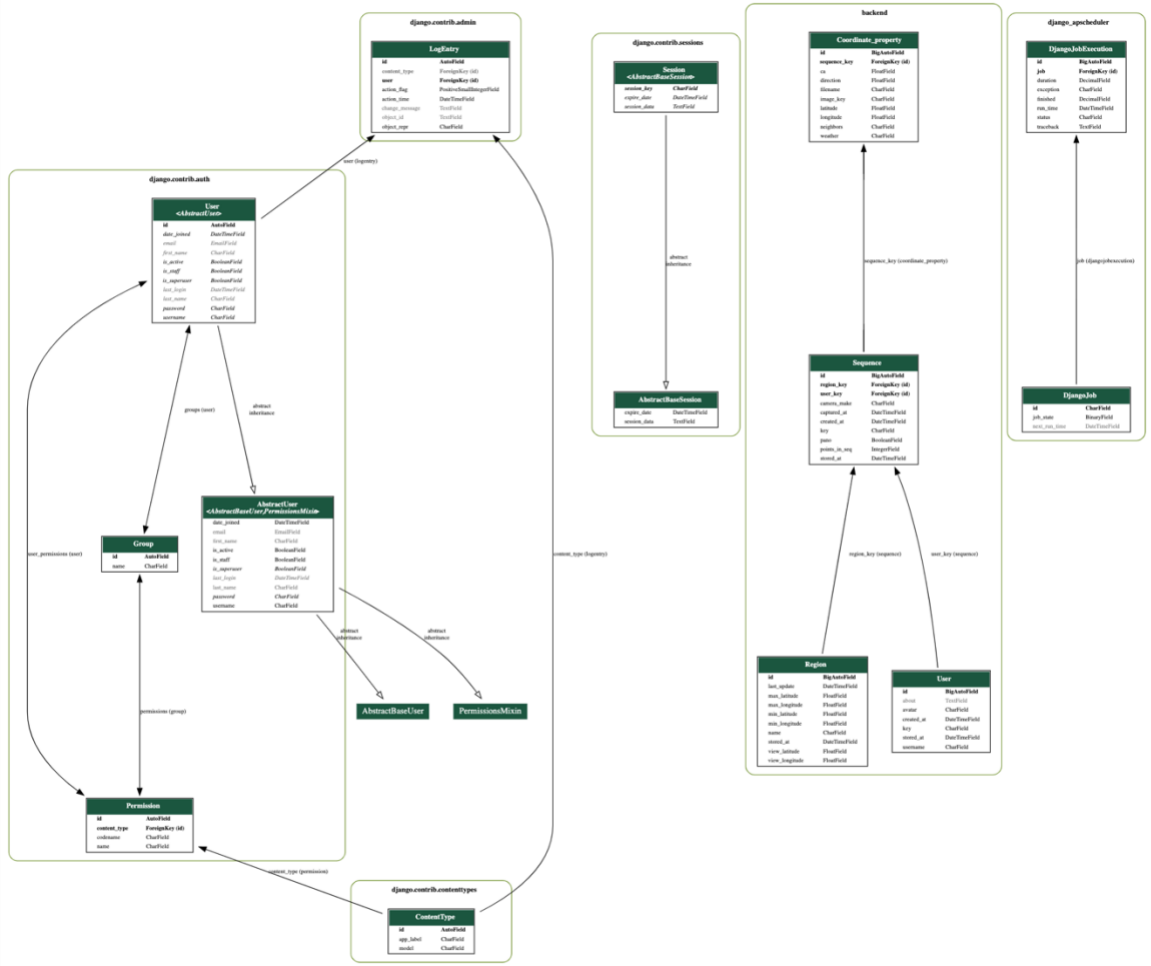


Figure 3 UML class diagram from Django models

3.2.4 Templates

Each view we define can also leverage templates, which help with the presentation layer of the HTML response. Each template is a separate file that consists of HTML along with some extra template syntax for variables, loops, and other control flow. Our template files are placed in a folder that we create called "templates", and it is inside the backend folder (*backend/templates*).

3.3 Database

By default, the database setup in Django is SQLite. The configuration can be found in *mapmanagementplatform/settings.py*.

```
# Database
# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

3.4 Django REST Framework

3.4.1 Serializers

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. In the project, the 3 models Coordinate_property, Region, and Sequence are serialized to convert the data to JSON content type that is served through Django REST API for React to consume. Details can be found in *backend/serializers.py*.

```
from rest_framework import serializers
from .models import Coordinate_property, Region, Sequence

# Serialize the Coordinate property and expose the data from the db to the API for the front to consume
class CoordinatePropertySerializer(serializers.ModelSerializer):
    coordinates = serializers.SerializerMethodField()

    class Meta:
        model = Coordinate_property
        fields = ('id', 'image_key', 'sequence_key', 'direction', 'neighbors', 'filename', 'coordinates',)
    def get_coordinates(self,obj):
      coordinate = obj.longitude, obj.latitude
      return coordinate

# Serialize the Region model
class RegionSerializer(serializers.ModelSerializer):
    class Meta:
        model = Region
        fields = '__all__'

# Serialize the Sequence model
class SequenceSerializer(serializers.ModelSerializer):
    coordinates = serializers.SerializerMethodField()
    class Meta:
        model = Sequence
        fields = ('id', 'camera_make', 'captured_at', 'key', 'pano', 'points_in_seq', 'coordinates',)
    def get_coordinates(self,obj):
      coordinates = Coordinate_property.objects.filter(sequence_key=obj.id)
      y_coords = coordinates.values_list('latitude',flat = True)
      x_coords = coordinates.values_list('longitude',flat = True)
      coordinate = list(zip(x_coords,y_coords))
      return coordinate
```

3.4.2 API Views

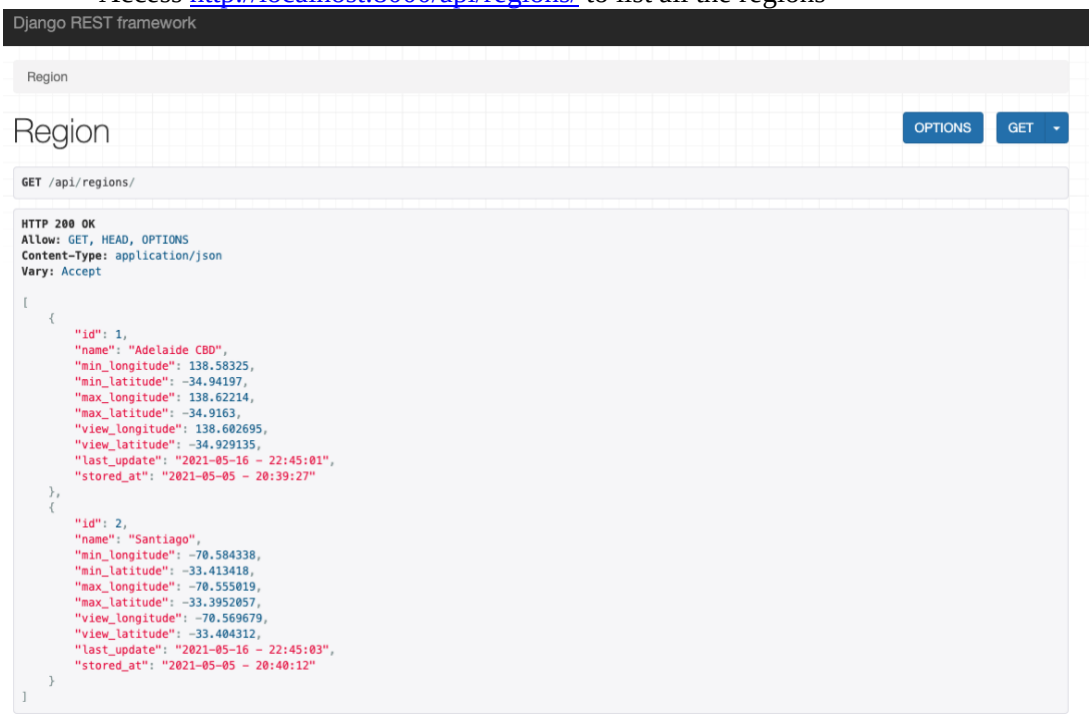
From the Serializers for Coordinate_property, Region, and Sequence, a List API View, which is a generic class-based view, is used to return the lists of coordinates, regions, and sequences and render them into JSON on Django REST API. The code for API views can be found in *backend/api_views.py*.

```python
# List the Coordinate properties on the API
class CoordinatePropertyView(generics.ListAPIView):
    serializer_class = CoordinatePropertySerializer

    def get_queryset(self):

        # Get URL parameter as a string, if exists
        ids = self.request.query_params.get('ids', None)
        minlon = self.request.query_params.get('min_lon', None)
        minlat = self.request.query_params.get('min_lat', None)
        maxlat = self.request.query_params.get('max_lat', None)
        maxlon = self.request.query_params.get('max_lon', None)
        datefrom = self.request.query_params.get('from',None)
        dateto = self.request.query_params.get('to',None)
        sequenceids = self.request.query_params.get('sequence_ids', None)

        # Get points for ids if they exist
        if ids is not None:
            # Convert parameter string to list of integers
            ids = [ int(x) for x in ids.split(',') ]
            # Get objects for all parameter ids
            queryset = Coordinate_property.objects.filter(pk__in=ids)
        else:
            if minlon is not None and minlat is not None and maxlat is not None and maxlon is not None and datefrom is None and datefrom is None:
                queryset = Coordinate_property.objects.filter(longitude__gte=minlon, longitude__lte=maxlon, latitude__gte=minlat, latitude__lte=maxlat)
            else:
                if minlon is not None and minlat is not None and maxlat is not None and maxlon is not None and datefrom is not None and datefrom is not None:
                    # Convert dates to Django Format
                    dt_format = '%Y-%m-%d'
                    f = datetime.strptime(datefrom, dt_format).replace(tzinfo=pytz.UTC)
                    t = datetime.strptime(dateto, dt_format).replace(tzinfo=pytz.UTC)
                    t = t + timedelta(days=1)
                    # Get sequences within the dates
                    coordinates = Coordinate_property.objects.filter(longitude__gte=minlon, longitude__lte=maxlon, latitude__gte=minlat, latitude__lte=maxlat).values('sequence_key').distinct()
                    queryset = Sequence.objects.filter(captured_at__range=[f, t],pk__in=coordinates)
                    queryset = Coordinate_property.objects.filter(sequence_key__in=queryset)
                else:
                    if sequenceids is not None :
                        sequenceids = [ int(x) for x in sequenceids.split(',') ]
                        queryset = Coordinate_property.objects.filter(sequence_key__in=sequenceids)
                    else:
                        # Else no parameters, return all objects
                        queryset = Coordinate_property.objects.all()
        return queryset


# List the Regions on the API
class RegionView(generics.ListAPIView):
    queryset = Region.objects.all()
    serializer_class = RegionSerializer
```

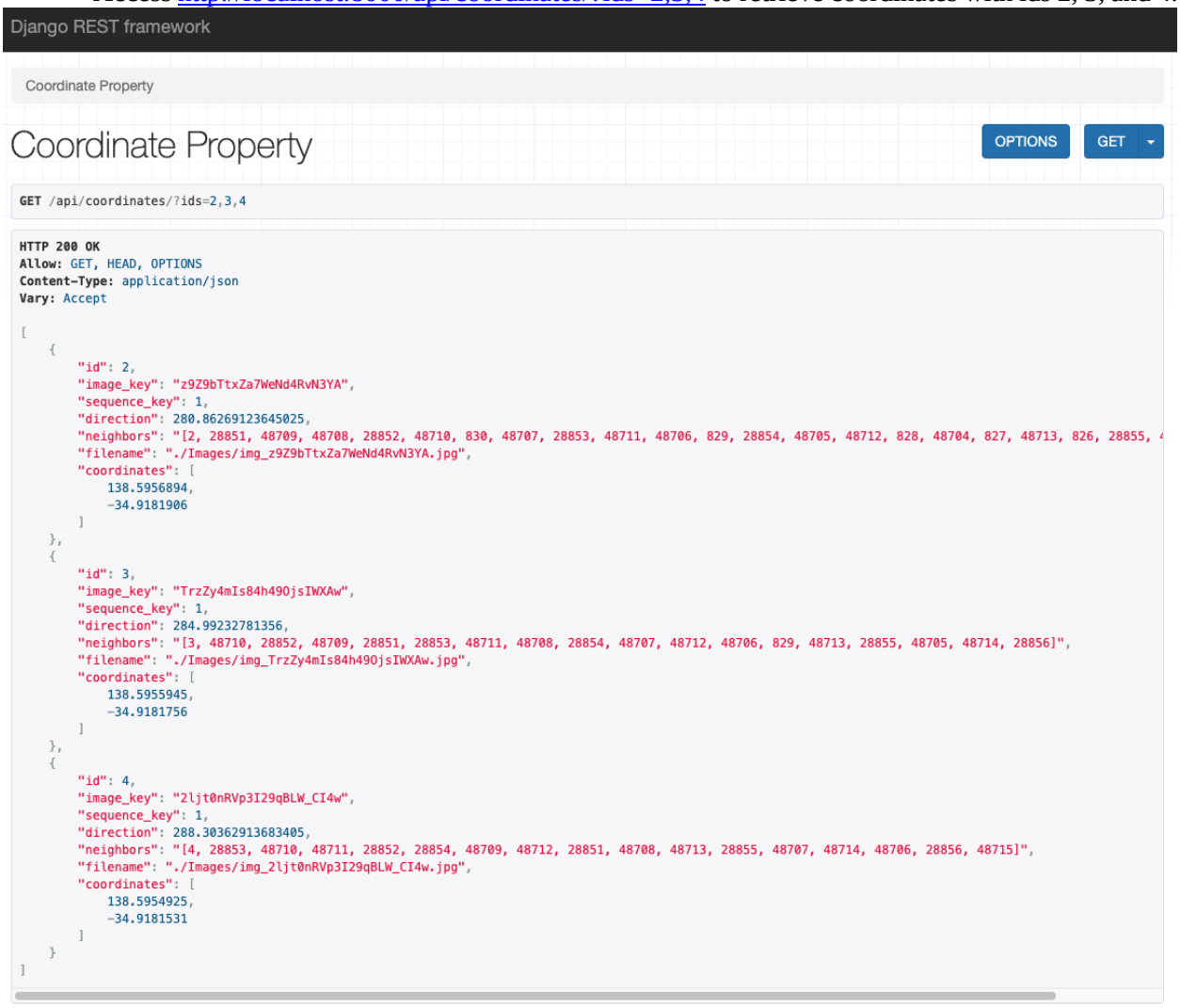
```python
# List the Sequences on the API
class SequenceView(generics.ListAPIView):
    serializer_class = SequenceSerializer

    def get_queryset(self):

        # Get URL parameter as a string, if exists
        ids = self.request.query_params.get('ids', None)
        datefrom = self.request.query_params.get('from', None)
        dateto = self.request.query_params.get('to', None)
        minlon = self.request.query_params.get('min_lon', None)
        minlat = self.request.query_params.get('min_lat', None)
        maxlat = self.request.query_params.get('max_lat', None)
        maxlon = self.request.query_params.get('max_lon', None)
        coordinates_ids = self.request.query_params.get('coordinates_ids', None)

        # Get sequences for ids if they exist
        if ids is not None:
            # Convert parameter string to list of integers
            ids = [ int(x) for x in ids.split(',') ]
            # Get objects for all parameter ids
            queryset = Sequence.objects.filter(pk__in=ids)
        else:
            if minlon is not None and minlat is not None and maxlat is not None and maxlon is not None and datefrom is not None and datefrom is None:
                coordinates = Coordinate_property.objects.filter(longitude__gte=minlon, longitude__lte=maxlon, latitude__gte=minlat, latitude__lte=maxlat).values('sequence_key').distinct()
                queryset = Sequence.objects.filter(pk__in=coordinates)
            else:
                if datefrom is not None and dateto is not None and minlon is not None and minlat is not None and maxlat is not None and maxlon is not None:
                    # Convert dates to Django Format
                    dt_format = '%Y-%m-%d'
                    f = datetime.strptime(datefrom, dt_format).replace(tzinfo=pytz.UTC)
                    t = datetime.strptime(dateto, dt_format).replace(tzinfo=pytz.UTC)
                    t = t + timedelta(days=1)
                    # Get sequences within the dates
                    coordinates = Coordinate_property.objects.filter(longitude__gte=minlon, longitude__lte=maxlon, latitude__gte=minlat, latitude__lte=maxlat).values('sequence_key').distinct()
                    queryset = Sequence.objects.filter(captured_at__range=[f, t],pk__in=coordinates)
                else:
                # Else no parameters, return all objects
                    queryset = Sequence.objects.all()
        return queryset
```

With this, we can query the lists of coordinates, regions, and sequences on Django REST API as required. For example:

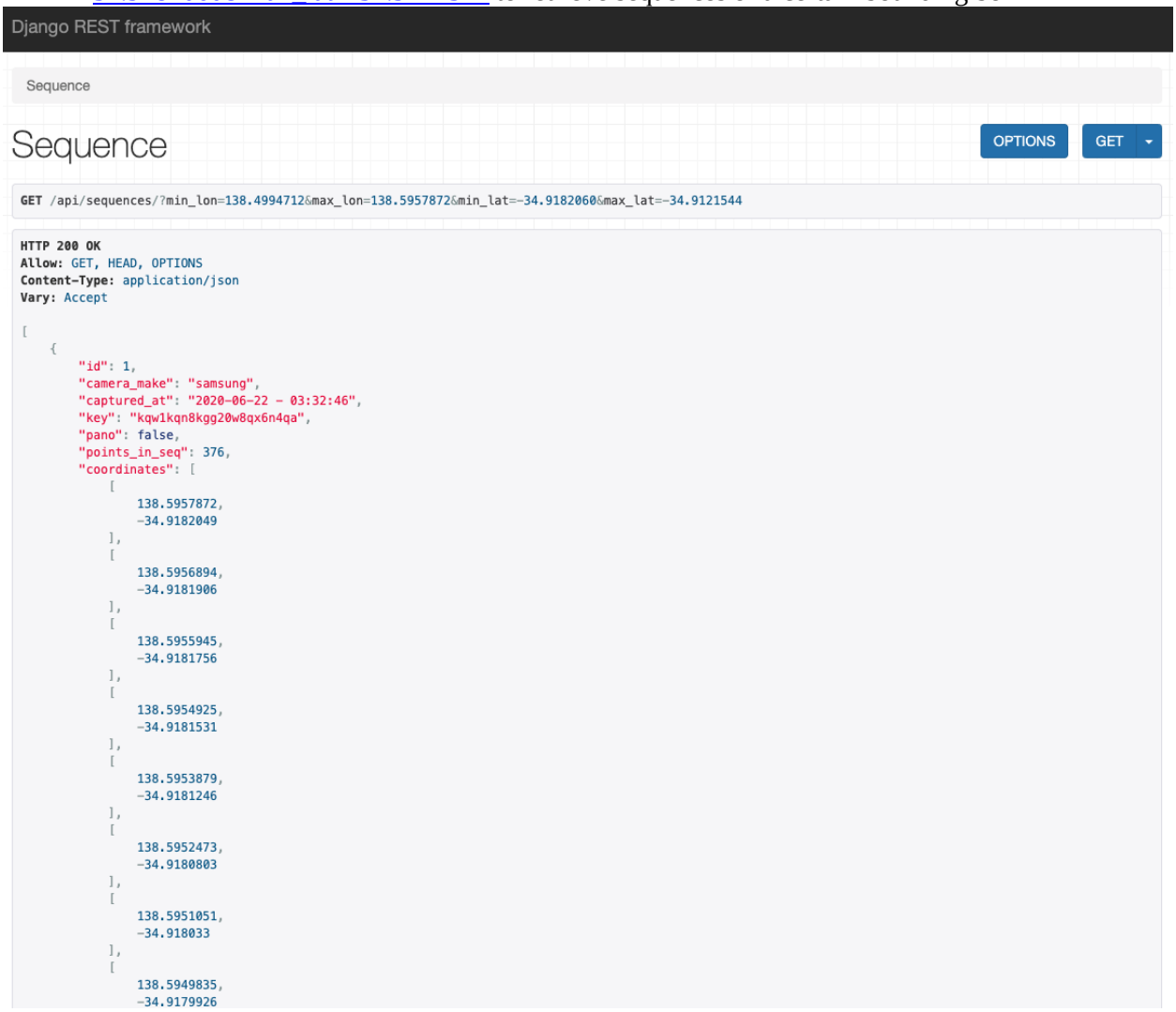
- Access http://localhost:8000/api/regions/ to list all the regions



- Access http://localhost:8000/api/coordinates/?ids=2,3,4 to retrieve coordinates with ids 2, 3, and 4.



- Access http://localhost:8000/api/sequences/?min_lon=138.4994712&max_lon=138.5957872&min_lat=-34.9182060&max_lat=-34.9121544 to retrieve sequences of a certain bounding box



3.5 Django REST API – React Integration

3.5.1 Axios

Axios is a promise-based HTTP Client for node.js and the browser. In our project, we use axios for the following purposes:

- Fetch coordinates from Django REST API (in *frontend/src/App.js*)

```
const getApi = async (id) => {
  return await axios
    .get(`api/coordinates/?ids=${id}`)
    .then((response) => {
      return response.data[0];
    })
    .catch((error) => {
      console.log('Error', error);
    });
};
```

- Fetch regions from Django REST API (in *frontend/src/layouts/topMenu.js*)

```
const getRegions = async () => {
await axios
    .get(`/api/regions`)
    .then((response) => {
        emptyregion(response.data);
    })
    .catch((error) => {
    console.log('Error', error);
    });
};
```

The configuration for axios can be found in *frontend/package.json* under "dependencies".

```
{
  "name": "webmapmanagement",
  "version": "0.1.0",
  "private": true,
  "proxy": "http://localhost:8000",
  "dependencies": {
    "@ant-design/icons": "^4.6.2",
    "@deck.gl/mapbox": "^8.4.1",
    "@luma.gl/test-utils": "^8.4.5",
    "@probe.gl/test-utils": "^3.3.1",
    "@testing-library/jest-dom": "^5.11.9",
    "@testing-library/react": "^11.2.5",
    "@testing-library/user-event": "^12.8.3",
    "antd": "^4.15.4",
    "axios": "^0.21.1",
    "debounce": "^1.2.1",
    "deck.gl": "^8.4.1",
    "gl": "^4.9.0",
    "mobx": "^6.1.8",
    "mobx-react": "^7.1.0",
    "moment": "^2.29.1",
    "probe.gl": "^3.3.1",
    "react-datepicker": "^3.8.0",
    "react-map-gl": "^6.1.10",
    "react-scripts": "4.0.3",
    "styled-components": "^5.2.1",
    "tape": "^5.2.2",
    "tape-catch": "^1.0.6",
    "web-vitals": "^1.1.0"
  },
```

3.5.2 webpack

webpack is a static module bundler for modern JavaScript applications. When webpack processes our application, it internally builds a dependency graph which maps every module our project needs and generates bundles under *frontend/static/frontend*.

The configuration file for webpack can be found in *frontend/webpack.config.js*. In React, webpack is configured in *frontend/package.json* under "main" and "devDependencies".

```
"main": "webpack.config.js",
"devDependencies": {
  "@babel/core": "^7.13.15",
  "@babel/plugin-proposal-class-properties": "^7.13.0",
  "@babel/preset-env": "^7.13.15",
  "@babel/preset-react": "^7.13.13",
  "@deck.gl/test-utils": "^8.4.16",
  "babel-loader": "^8.1.0",
  "cypress": "^7.3.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
  "webpack": "^4.44.2",
  "webpack-cli": "^4.6.0"
},
```

Once a code change is made in React frontend in development, the script "npm run dev" is used to call webpack. This script is indicated in *frontend/package.json* under "scripts".

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "e2e": "cypress run",
  "dev": "webpack --mode development --entry ./src/index.js --output-path ./static/frontend"
},
```

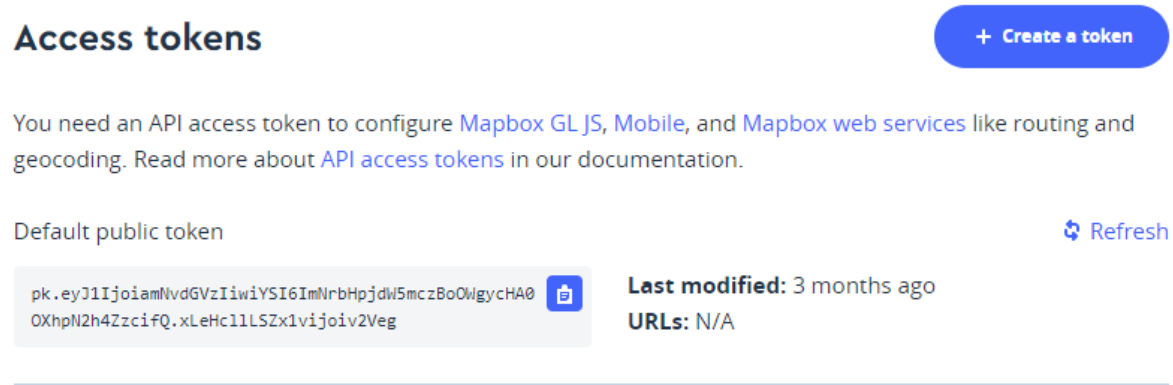
3.5.3 Babel

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.
The configuration file for Babel can be found in *frontend/.babelrc*. In React, Babel is configured in *frontend/package.json* under "devDependencies". The babel-loader package allows transpiling JavaScript files using Babel and webpack.

```
"devDependencies": {
  "@babel/core": "^7.13.15",
  "@babel/plugin-proposal-class-properties": "^7.13.0",
  "@babel/preset-env": "^7.13.15",
  "@babel/preset-react": "^7.13.13",
  "@deck.gl/test-utils": "^8.4.16",
  "babel-loader": "^8.1.0",
  "cypress": "^7.3.0",
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
  "webpack": "^4.44.2",
  "webpack-cli": "^4.6.0"
},
```

3.6 Mapbox

Our web application uses Mapbox to render the map where the layers are then displayed. In order to use Mapbox toolbox, we need to generate an access token using our account in Mapbox Account. Mapbox automatically generates an access token as displayed in the image below.



We use that token when rendering the frontend in Reactjs. The Mapbox access token is set in *frontend/src/App.js*

```
// Set your mapbox access token here
const MAPBOX_ACCESS_TOKEN =
  'pk.eyJ1IjoiamNvdGVzIiwiYSI6ImNrbHpjdW5mczBoOWgycHA0OXhpN2h4ZzcifQ.xLeHcllLSZx1vijoiv2Veg';
```

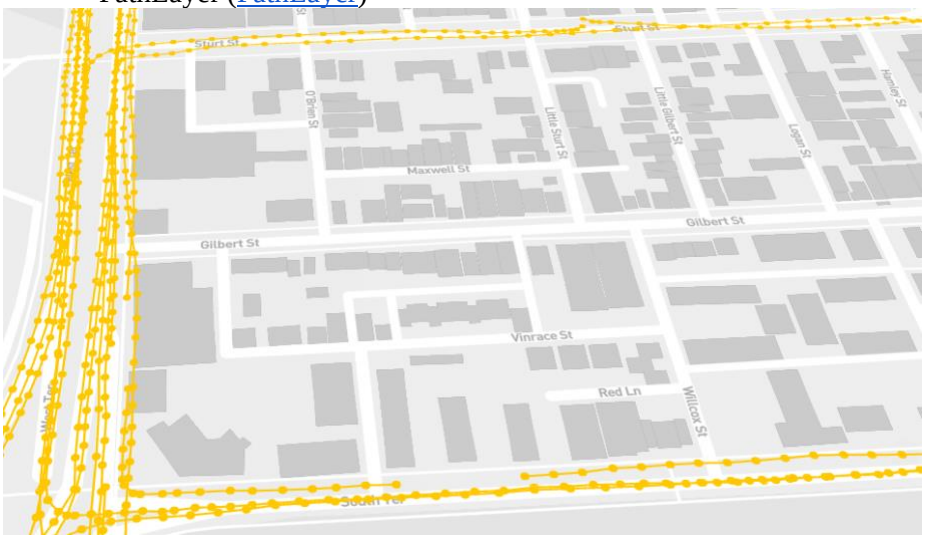
The token is then used when rendering the StaticMap in the same file.

```
<StaticMap mapboxApiAccessToken={MAPBOX_ACCESS_TOKEN} />
```

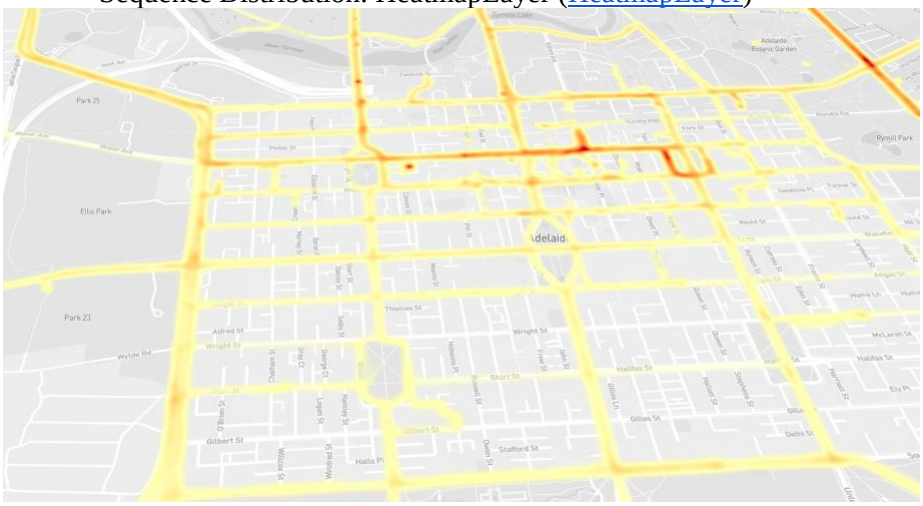
3.7 Deck.gl

Deck.gl is a framework to render large amounts of data in different formats. We are using Deck.gl to render our layers in the map. To properly use the Deck.gl framework we follow the guidelines from Introduction. In particular, we use the following layers:

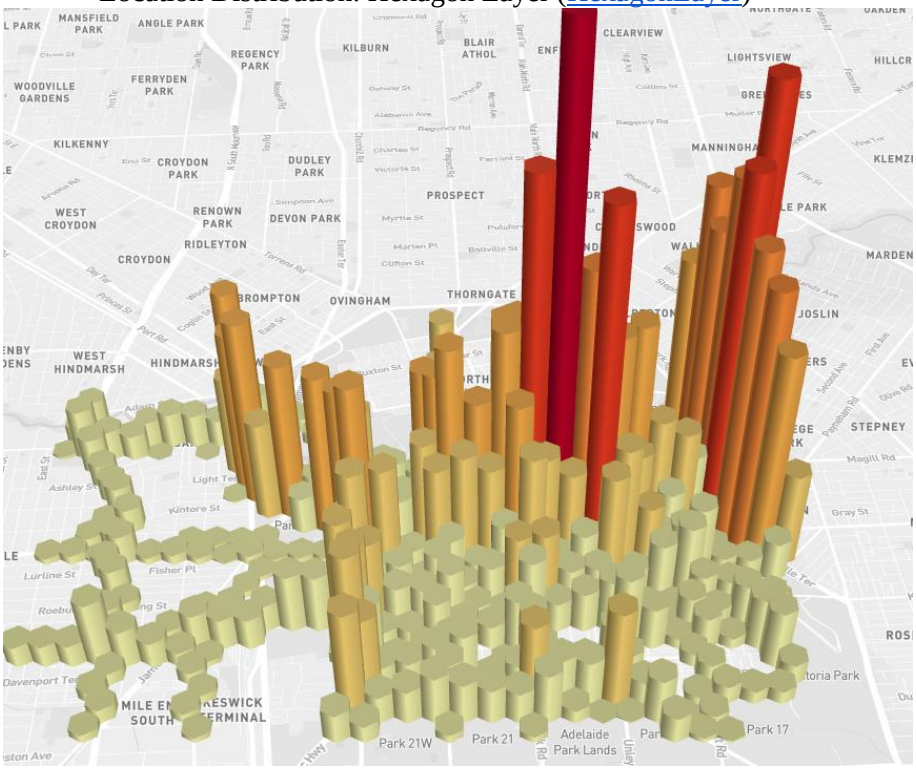
- Location and Sequences: Renders two layers, the ScatterplotLayer (ScatterPlotLayer) and PathLayer (PathLayer)



- Sequence Distribution: HeatmapLayer (HeatmapLayer)



- Location Distribution: Hexagon Layer (HexagonLayer)



3.8 Frontend – Django & React

3.8.1 URL Patterns

When a request comes into Django with a root path of just a slash, the URL patterns in *frontend/urls.py* routes to the function in *frontend/views.py* called *index()*.

```
urlpatterns = [
    path('', views.index ),
]
```

3.8.2 Views

This *index()* function in *frontend/views.py* takes the request as an argument, and it will first make any database calls that are needed, using the models to find in *models.py*. Next, the index view will pass the HTTP request and this data from the database into the template file under *frontend/templates/frontend/index.html*.

```
# Create your views here.
def index(request):
    return render(request, 'frontend/index.html')
```

3.8.3 Templates

At this point, the template in *frontend/templates/frontend/index.html* will render HTML based on the data from React with the source file as *static/frontend/main.js*. The view will return an HTTP response with this HTML as the body, thus completing the processing for the request.

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Map Management WebApp</title>
</head>
<body>
  <div id="root">
    <!-- React will load here -->
  </div>
  <script src="{% static "frontend/main.js" %}"></script>
</body>
</html>
```

4. System Requirements

4.1 Mapillary API – Django integration

4.1.1 Fetching users and sequences from Mapillary

- Overview of the requirement:

For the Map Management platform, the user wants to have an updated version of the available data from Mapillary. To accomplish this, the backend needs to be able to fetch the latest information from Mapillary's API. The user requested to store two types of data, one is the information about users, the other is for the sequences and points in every sequence. The user needs to generate a region in the database from the administration site and trigger a request to download the data.

- Technical details:

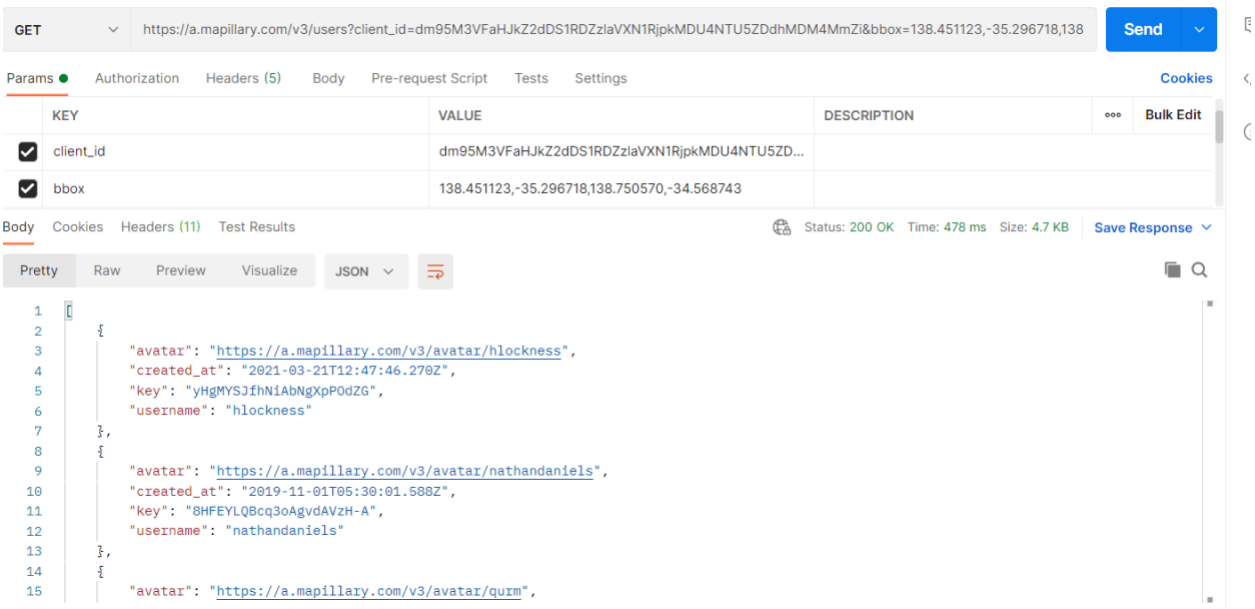
As stated above, the application fetches users and sequences from Mapillary's API and stores that information in our database. The process is triggered by the job function called *mapillaryupdate* in *mapmanagementplatform/urls.py*. It will fetch all the users and sequences from all the regions currently stored in the database using the bounding box as a reference.

```
url = 'https://a.mapillary.com/v3/sequences'
#Set up params to pass in the request
client_id = 'dm95M3VFaHJkZ2dDS1RDZzlaVXN1RjpkMDU4NTU5ZDdhMDM4MmZi'
bbox = str(region.min_longitude) + ','+ str(region.min_latitude) + ',' + str(region.max_longitude) + ',' + str(region.max_latitude)
if (region.last_update is not None):
    start_time = region.last_update.strftime("%Y-%m-%d")
    PARAMS = {'client_id': client_id, 'bbox': bbox, 'start_time' : start_time}
else:
    PARAMS = {'client_id': client_id, 'bbox': bbox}
#Create the request using the params previously defined
response = requests.get(url, params = PARAMS, headers={'Content-Type': 'application/json'})
```

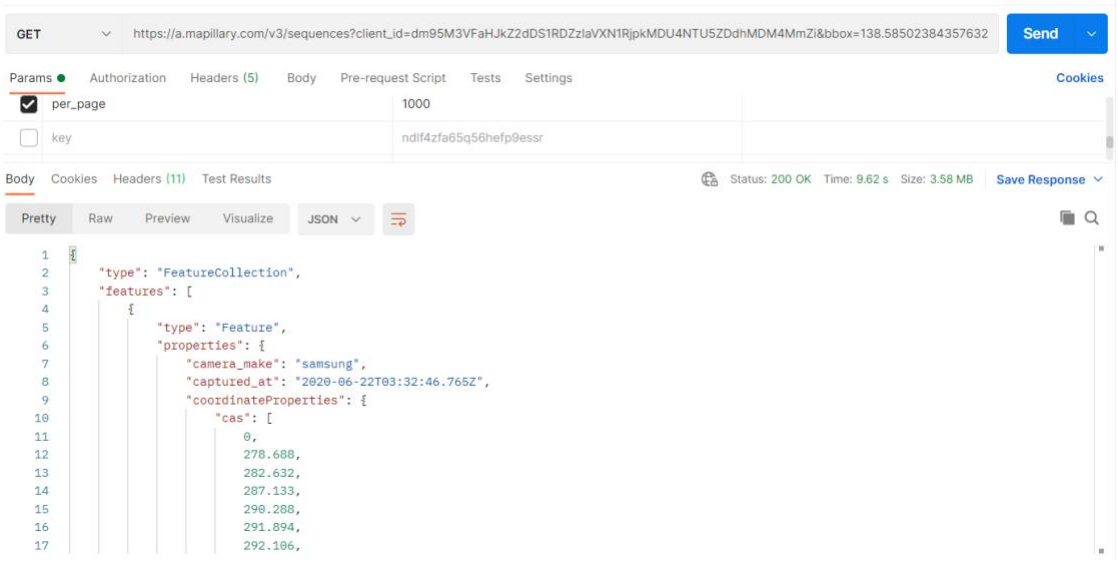
A GET request is generated using the URL constructed following Mapillary's API Guidelines. The process

will bring all the data if there is no previous data from a Region. In case there is already data, it will fetch new information from the last updated date using the *last_update* data stored in the database as shown in the image above. Everytime the process is finalized, the *last_update* will be updated for every Region so that it can be referenced the next time.

Mapillary responds to every GET request with a JSON file that then is stored in the database. The following images display the information in the user requests using Postman (Postman | The Collaboration Platform for API Development) to simulate the request:

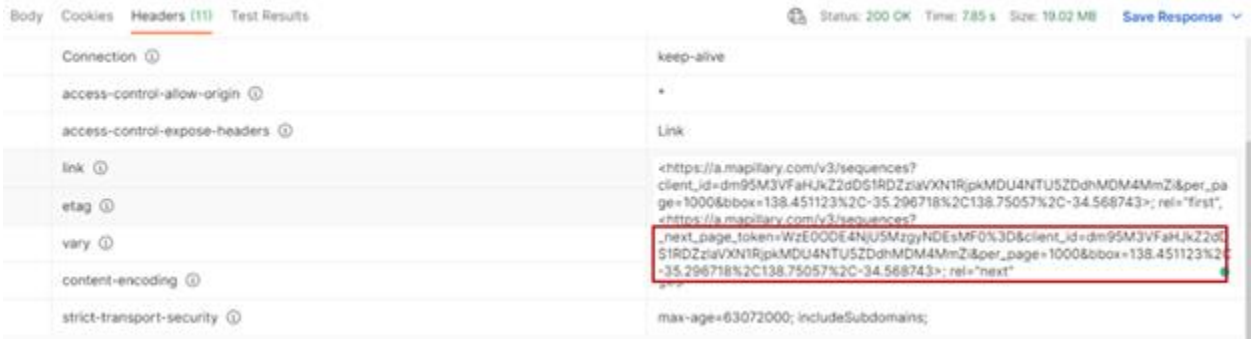


And for the sequence requests:



If the number of objects (users or sequences) in the request is larger than 200, then a “next” url is available

within the Headers of the response as displayed in the following image.



The fetching function will keep retrieving data as long as there's a next in the Headers.

4.1.2 Downloading images from Mapillary

- Overview of the requirement:

Everytime the new data is received, new images corresponding to every new coordinate must be downloaded from Mapillary using an API call based on the image key and stored in `/backend/static/images` as `.jpg` files.

- Technical details:

Using multiprocessing below is the implemented algorithm to download the images by building URl's.
It takes about **10 seconds to download 1000 images in 640px resolution** when **n_processes =30**.
This is done using the `get_images()` method in views.py as follows:

1. Image keys are loaded into memory from the database from new values.

```
317  def get_images():
318      start = time.time()
319      keys = []
320      filename =[]
321      urls=[]
322      # load image keys into variable "[keys]"
323      image_keys = Coordinate_property.objects.values_list('image_key', flat=True)
324      for key in image_keys:
325              row_entry = key
326              keys.append(row_entry)
```

2. Next, urls are generated for all keys

```
336      for i in range (0,len(keys)):
337          url = 'https://images.mapillary.com/'
338          # formatting image name
339          key = str(keys[i])
340          # build image url for resolutionB '640'
341          url = url + key + resolutionB
342          urls.append(url)
```

By default, resolution of images is set to 'resolutionB'= 640px and can be changed from these following options:

```
resolutionA = '/thumb-320.jpg'    //320 px
resolutionB = '/thumb-640.jpg'    //640 px
resolutionC = '/thumb-1024.jpg'   //1024 px
resolutionD = '/thumb-2048.jpg'   //2048 px
```

3. Images are now downloaded by `image_downloader(img_url: str)` which takes url's to all images as a list of strings and is verbose for every image being downloaded and download completed. For every missed image, it tries to download it again 5 times as on line 347. If still unable to, it catches and prints an error.

```
343     def image_downloader(img_url: str):
344         print(f'Downloading: {img_url}')
345         res = requests.get(img_url, stream=True)
346         count = 1
347         while res.status_code != 200 and count <= 5:
348             res = requests.get(img_url, stream=True)
349             print(f'Retry: {count} {img_url}')
350             count += 1
351         # checking the type for image
352         if 'image' not in res.headers.get("content-type", ''):
353             print('ERROR: URL doesnot appear to be an image')
354             return False
355         # Trying to red image name from response headers
356         try:
357             image_name = img_url.split("/")
358             name = "img_" + str(image_name[3]) + ".jpg"
359         except:
360             image_name = str(random.randint(11111, 99999))+'.jpg'
361 
362         i = Image.open(io.BytesIO(res.content))
363         download_location = "./backend/static/1-images"
364         i.save(download_location + '/'+ name)
365         return f'Download complete: {img_url}'
```

4. For speeding up the downloading process, multiprocessing is used. By default, the number of processes allocated is `n_process = 30` on `line 367` and can be changed according to user requirements.

```
367     n_process = 30
368     print(f'MESSAGE: Running {n_process} process')
369     results = ThreadPool(n_process).imap_unordered(image_downloader, urls)
```

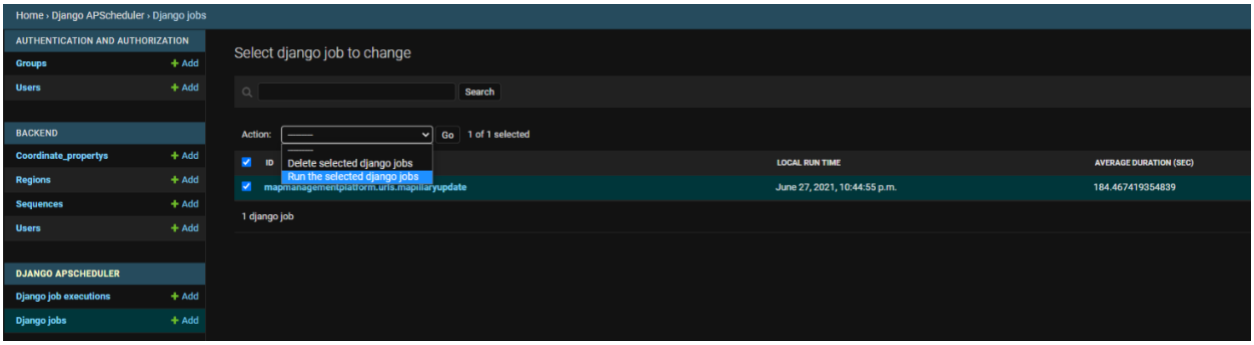
4.1.3 Scheduler

- Overview of the requirement:

As previously stated, the platform needs to be self-updated with the latest data. The web application needs to get the latest users, sequences and images from the stored regions every two weeks.

- Technical details:

To fulfill this requirement we develop a Job Scheduler to run in the background that fetches new data (if there is any). The scheduler can also be triggered manually by the Django admin site in the backend application.



The scheduler uses all the previous defined functions and neighbors as described in 4.2.1 and it is called from *mapmanagementplatform/urls.py* using `@register_job(scheduler, "interval", weeks = 2)`

```
@register_job(scheduler, "interval", weeks = 2)
def mapillaryupdate():
    print('-----------------------------------------')
    print('Updating function - fetching users from Mapillary')
    print('-----------------------------------------')
    users()
    print('-----------------------------------------')
    print('Updating function - fetching sequences from Mapillary')
    print('-----------------------------------------')
    sequences()
    print('-----------------------------------------')
    print('Updating function - Checking directions for all data')
    print('-----------------------------------------')
    sanitydirection()
    print('-----------------------------------------')
    print('Updating function - Calculating neighbors')
    print('-----------------------------------------')
    deltaneighbors()
    print('-----------------------------------------')
    print('Downloading images')
    print('-----------------------------------------')
    get_images()
    print('-----------------------------------------')
    print('Finished Fetching')
    print('-----------------------------------------')

register_events(scheduler)

scheduler.start()
print("Scheduler started!")
```

The *mapillayupdate* will fetch the users and sequences, then check if all the directions are calculated using

the *sanitydirection* and calculate if there are any that have null values. After that it will calculate the neighbors for any new data stored in the database and finally download the images for the new values.

4.2 Backend – Django views

4.2.1 Nearest neighbors

- Overview of the requirement:

Collecting raw and unstructured data from Mapillary needs to be processed and computations need to be done to calculate the nearest neighbors of all the coordinates as the most important requirement for this project. For this project, a ***QuadTree* based custom algorithm** is implemented to cater to geographic data processing considering time and resource constraints.

- Technical details:

A quadtree algorithm that works in **kO(log$_4$N)** where K is the number of neighbors and N is the total number of points, takes roughly *8 hours for finding all nearest neighbors in a 100 sq m area for 3 million points*.
It features the following classes:

`class Point` – Generates points by accepting id of the coordinate in database, longitude and latitude values as id , x and y.

```
6   class Point:
7       def __init__(self,id, x, y, data=None):
8           self.id,self.x, self.y = id,x, y
9           self.data = data
10      def distance_to(self, other):
11          try:
12              other_x, other_y = other.x, other.y
13          except AttributeError:
14              other_x, other_y = other
15
16          dist = DistanceMetric.get_metric('minkowski')
17
18          V=[other_y-90,other_x-180],[self.y-90,self.x-180]
19          distance = dist.pairwise(V)
20          # print ("dist",distance[0][1])
21          return distance[0][1]
```

`class Rect(object)` – Accepts a centroid for rectangle and height/2 and width/2 values to return a rectangle with properties such as `` `contains` ``,`` `intersects` `` and `` `draw` `` to return if the bounding area contains a point, intersects another rectangle or to draw it using plotlib.

```python
class Rect(object):
    def __init__(self, cx, cy, w, h):
        self.cx, self.cy = cx, cy
        self.w, self.h = w, h
        self.west_edge, self.east_edge = cx - w/2, cx + w/2
        self.north_edge, self.south_edge = cy - h/2, cy + h/2

    def contains(self, point):
        try:
            point_x, point_y = point.x, point.y
        except AttributeError:
            point_x, point_y = point

        return (point_x >= self.west_edge and
                point_x <  self.east_edge and
                point_y >= self.north_edge and
                point_y < self.south_edge)

    def intersects(self, other):
        return not (other.west_edge > self.east_edge or
                    other.east_edge < self.west_edge or
                    other.north_edge > self.south_edge or
                    other.south_edge < self.north_edge)

    def draw(self, ax, c='k', lw=1, **kwargs):
        x1, y1 = self.west_edge, self.north_edge
        x2, y2 = self.east_edge, self.south_edge
        ax.plot([x1,x2,x2,x1,x1],[y1,y1,y2,y2,y1], c=c, lw=lw, **kwargs)
```

`class QuadTree:` The main class which accepts parameters; `boundary, max_points, depth` for the quadtree and has functionalities such as `sub_divide, insert and query`.

```python
class QuadTree:
    def __init__(self, boundary, max_points, depth=5):

        self.boundary = boundary
        self.max_points = max_points
        self.points = []
        self.depth = depth
        self.divided = False

    def sub_divide(self):

        cx, cy = self.boundary.cx, self.boundary.cy
        w, h = self.boundary.w / 2, self.boundary.h / 2

        self.nw = QuadTree(Rect(cx - w/2, cy - h/2, w, h),
                           self.max_points, self.depth + 1)
        self.ne = QuadTree(Rect(cx + w/2, cy - h/2, w, h),
                           self.max_points, self.depth + 1)
        self.se = QuadTree(Rect(cx + w/2, cy + h/2, w, h),
                           self.max_points, self.depth + 1)
        self.sw = QuadTree(Rect(cx - w/2, cy + h/2, w, h),
                           self.max_points, self.depth + 1)
        self.divided = True

```

```
    def insert(self, point):
        if not self.boundary.contains(point):
            return False

        if len(self.points) < self.max_points:
            self.points.append(point)
            return True

        if not self.divided:
            self.sub_divide()

        return (self.ne.insert(point) or self.nw.insert(point) or self.se.insert(point) or self.sw.insert(point))

    def query(self, boundary, found_points):
        if not self.boundary.intersects(boundary):
            return False

        for point in self.points:
            if boundary.contains(point):
                found_points.append(int(point.id))

        if self.divided:
            self.nw.query(boundary, found_points)
            self.ne.query(boundary, found_points)
            self.se.query(boundary, found_points)
            self.sw.query(boundary, found_points)
        return found_points
```

For populating all nearest neighbors, `neighbors()` is called, after which for calculating nearest neighbors only for new values, `deltaneighbors()` is called from Scheduler (4.1.3). `neighbors()` and `deltaneighbors()` gets all data from the database and stores them in newcoordinates for further processing.

```
newcoordinates =
Coordinate_property.objects.filter(neighbors__isnull=True).order_by('id')
```

Next, the values are loaded into arrays `coordinates` and `newcoordinates` and prepared for processing.

```
    for coordinate in coordinates:
        row_entry = [coordinate.id, coordinate.longitude, coordinate.latitude,
        coordinate.direction, coordinate.sequence_key_id,coordinate.neighbors]
        data.append(row_entry)
    for newcoordinate in newcoordinates:
        newrow_entry = [newcoordinate.id, newcoordinate.longitude,
        newcoordinate.latitude, newcoordinate.direction, newcoordinate.sequence_key_id]
        newdata.append(newrow_entry)
    # PROCESS DATA
    coords = np.array(data)
    newcoords = np.array(newdata)
    # MAKE A LOCAL COPY IN MEMORY
    original_data = np.copy(coords)
    neworiginal_data = np.copy(newcoords)
    # STORE POINTS(id,long,lat)
    p_coords = coords[:,0:3]
    # DROP ID, etc f`rom coords
    coords = coords[:,1:3]
```

Preparing data feed for the *QuadTree* now, which requires the following:

1. Setting the domain by drawing a rectangle by adding 500 units around the median values of latitude and longitude after normalising the values to be all positive. Adding 500 units ensures to cover all possible values ranging for longitude (0 to 360) and latitude (0 to 180) after normalization.

```
    # EXTRACT MAX and MIN values in long and lat
    max_long_lat,min_long_lat = np.amax(coords, axis=0), np.amin(coords, axis=0)
    # NORMALIZE VALUES OF LAT, LONG AS DOMAIN CAN HANDLE ONLY +VE VALUES
    normalize_long,normalize_lat = 180.0,90.0
    tot_long,tot_lat = 360.0,180.0
    max_long,max_lat = (max_long_lat[0]+normalize_long),(max_long_lat[1]+normalize_lat )
    min_long,min_lat = (min_long_lat[0]+normalize_long), (min_long_lat[1]+normalize_lat )
    coords[:,0],coords[:,1] = (coords[:,0] + normalize_long),(coords[:,1] + normalize_lat )
    # CENTER DOMAIN AT CENTROID OF THE RECTANGLE BOUNDING ALL POINTS IN DB
    center_long = ((max_long + min_long) /2 )
    center_lat = ((max_lat + min_lat) /2 )
    domain = Rect(center_long,center_lat,500,500)
```

2. Generate the points to be fed to *QuadTree* algorithm from the values in `coordinates` and `newcoordinates`.

```
    points = [Point(*coordsp) for coordsp in p_coords]
    points_load_time = time.process_time() - start_generatin_points
```

3. Building a quad tree from the domain and inserting these points into the tree.

```
    # BUILD TREE AND INSERT POINTS
    start_building_tree = time.process_time()
    qtree = QuadTree(domain, 50)
    gen_time = time.process_time() - start_building_tree
    # INSERT POINTS INTO TREE
    start = time.process_time()
    for point in points:
        qtree.insert(point)
    insertion_time = time.process_time() - start
```

Now that the QuadTree is in memory and populated with all the points, we can start querying it to calculate the nearest neighbors after setting the precision to 10^7 and query region of 0.001 x 0.001 around the query point which is roughly 100 sq meters around the query point on line 173 and can be changed to user defined values.

```
157         # QUERYING FOR ALL NEW POINTS
158         query_allpoints = time.process_time()
159         for ncoordinate in newcoordinates:
160             if ncoordinate.id %100 ==0 :
161                 print ("CHECKING FOR ",ncoordinate.id,"th COORDINATE. QUERY TIME TILL NOW = ",
162                 time.process_time()-query_allpoints,"SECONDS")
163             found_points = []
164             distanced_neighbors = []
165             neighs = []
166             i = int(ncoordinate.id-1)
167             # LOAD VALUES FOR DIRECTION AND SEQ.NO. OF QUERY POINT FROM ORIGINAL DATA
168             lon  = coords[i][0]
169             lat  = coords[i][1]
170             dir_querypt  = original_data[i][3]
171             seq_querypt  = original_data[i][4]
172             # SET QUERY REGION OF SEARCH AREA OF "0.001 X 0.001" CENTERED AT LON,LAT AND
173             coord_region = Rect(lon, lat, 0.001, 0.001)
174             # LOG QUERY TIME FOR FINDING i'th NN POINTS
175             querying_nn_points_t = time.process_time()
176             # QUERY TREE FOR i'th POINTS(lon,lat) NN
177             qtree.query(coord_region, found_points)
178             final_neigh=[i+1]
```

With the neighbors id for this query point stored in `found_points` as a list, we now filter these nearest neighbors returned by the algorithm based on view direction within -45 to +45 for current points' view direction in which the image was taken and if the neighbor does not belong to the same sequence, ignore if direction is not available.

```
        # Got ID'S OF i'th points Nearesst Neighbours
        # Filter these considering direction,ditance and seq. no.
        for j in range (0,len(found_points)):
            # SELECT ID OF NN POINT OF QUERY POINT
            id = found_points[j]
            # GET ALL METADATA FROM ORIGINAL_DATA FOR THIS NEAREST NEIGHBOUR TO FIYLTER IT
            # id-1 because numpy array location starts with 0
            longitu = original_data[id-1][1]
            latitu  = original_data[id-1][2]
            diru    = original_data[id-1][3]
            sequ    = original_data[id-1][4]
            if(diru != None and dir_querypt!= None ):
                # CALCULATE ANGLE CHANGE in VIEW DIRECTION
                angledIff = (original_data[id-1][3]-dir_querypt + 180 + 360) % 360 - 180
                if (angledIff <= 45 and angledIff>=-45):
                    # CHECK FOR SEQUENCE NO
                    if (sequ!= seq_querypt):
                        distance =  points[i].distance_to(points[id-1])
                        distanced_neighbours = [distance,id]
                        neighs.append(distanced_neighbours)
            else:
                print("Missing direction values for id : ",id," or ",i+1)
```

Finally we sort the neighbors based on their distance to the query point and save them in the database.

```
        sorted_neighs=sorted(neighs)
        final_neigh+=[x[1] for x in sorted_neighs]
        querying_nn = time.process_time() - querying_nn_points_t
        ncoordinate.neighbors = final_neigh
        ncoordinate.save()
```

4.2.2 Direction calculation

- Overview of the requirement:

Due to the lack of accuracy given from the *ca* (Camera angle) data received directly from Mapillary, the user requested the platform to be able to calculate the direction in which the image was captured by the car. The accuracy of the direction is key in our application because the user wants to see only images that are within a certain range of the angle given a particular point. In our case, the user defined that when he clicks on a point, he only wants to see images that are within 45° of separation from the current point.

- Technical details:

To calculate the direction of a point we need to use the next point in the sequence so that we can determine where the car was pointing at that particular point. We use the geodesic library available for Python (GeographicLib API — geographiclib 1.50 documentation), specifically the *inverse* function that solves the inverse geodesic problem. From the Geodesic documentation we have the following:

Inverse(*lat1*, *lon1*, *lat2*, *lon2*, *outmask=1929*) [source]

Solve the inverse geodesic problem

Parameters:
- **lat1** – latitude of the first point in degrees
- **lon1** – longitude of the first point in degrees
- **lat2** – latitude of the second point in degrees
- **lon2** – longitude of the second point in degrees
- **outmask** – the output mask

Returns: a Geodesic dictionary

Compute geodesic between (*lat1*, *lon1*) and (*lat2*, *lon2*). The default value of *outmask* is STANDARD, i.e., the *lat1*, *lon1*, *azi1*, *lat2*, *lon2*, *azi2*, *s12*, *a12* entries are returned.

The inverse returns the Geodesic dictionary:

Geodesic dictionary

The results returned by `Geodesic.Direct`, `Geodesic.Inverse`, `GeodesicLine.Position`, etc., return a dictionary with some of the following 12 fields set:

- *lat1* = φ_1, latitude of point 1 (degrees)
- *lon1* = λ_1, longitude of point 1 (degrees)
- *azi1* = α_1, azimuth of line at point 1 (degrees)
- *lat2* = φ_2, latitude of point 2 (degrees)
- *lon2* = λ_2, longitude of point 2 (degrees)
- *azi2* = α_2, (forward) azimuth of line at point 2 (degrees)
- *s12* = s_{12}, distance from 1 to 2 (meters)
- *a12* = σ_{12}, arc length on auxiliary sphere from 1 to 2 (degrees)
- *m12* = m_{12}, reduced length of geodesic (meters)
- *M12* = M_{12}, geodesic scale at 2 relative to 1 (dimensionless)
- *M21* = M_{21}, geodesic scale at 1 relative to 2 (dimensionless)
- *S12* = S_{12}, area between geodesic and equator (meters^2)

We need the azimuth (or bearing) of the results and we use that within our *views.py* call:

```
    #Calculate the angle between to two points using the Geodesic library, currently used
def angleFromCoordinateLib(lat1, lng1, lat2, lng2):
    geod = Geodesic.WGS84
    bearing = geod.Inverse(lat1,lng1,lat2,lng2)
    bearing = (bearing['azi1'] + 360) % 360
    return bearing
```

Now that we have the function to calculate the direction between two points in the map, we can use that for every sequence we have stored in the map. To accomplish this, we use the *getdirection* function that receives a sequence and calculates all the directions for the points in that sequence.

```
    #Function called by calculatealldirection, calculate the direction for a particular sequence
def getdirection(sequencetosearch):
    coordinates = Coordinate_property.objects.filter(sequence_key = sequencetosearch.id).order_by('id')
    lastcoordinate = Coordinate_property.objects.filter(sequence_key = sequencetosearch.id).order_by('id').last()
    previousdirection = 0
    #LOOP THROUGH THE COORDINATES FROM THE SEQUENCE
    for coordinate in coordinates:
        #CASE LAST COORDINATE FROM THE PARTICULAR SEQUENCE
        if coordinate == lastcoordinate:
            coordinate.direction = previousdirection
        #OTHER CASES
        else:
            lat1 = coordinate.latitude
            long1 = coordinate.longitude
            nextcoordinate = Coordinate_property.objects.filter(id__gt = coordinate.id).order_by('id').first()
            lat2 = nextcoordinate.latitude
            long2 = nextcoordinate.longitude
            coordinate.direction = angleFromCoordinateLib(lat1, long1, lat2, long2)
            if(coordinate.direction == 0):
                coordinate.direction = previousdirection
            previousdirection = coordinate.direction
        coordinate.save()
```

There are a couple of boundary cases that we need to be aware:

- The car did not move and the two points in the map are the same: in this case, the direction of the second point is the same as the first.
- The last point in the sequence: same as the previous case, the direction of the second point is the same as the first.

4.3 Frontend – React

4.3.1 Select a region

- Overview of the requirement:

The user will have the option to select any region that is stored in the platform. When a region is selected, the map will be moved to that destination.

- Technical details:

When the frontend loads, a request will be triggered to fetch the regions stored in the backend. The fetching is handled by the *frontend/src/layouts/TopMenu.js*

```
const getRegions = async () => {
await axios
    .get(`/api/regions`)
    .then((response) => {
        emptyregion(response.data);
    })
    .catch((error) => {
    console.log('Error', error);
    });
};

useEffect(() => {
getRegions();
}, []);
```

After the fetching is finalized the frontend will generate the dropdown menu (which is provided by the ant design library) for the regions handled by the same js.

```
//REGIONS MENU
const regions = (
  <Menu onClick={handleOnChangeRegions} selectable={true} defaultSelectedKeys={['1']}>
  {
    regionsresult
        .sort((a, b) => a.name > b.name ? 1 : -1)
        .map(x=>{
        return(
            <Menu.Item key={x.id} lat={x.view_latitude} lon={x.view_longitude}>
                {x.name}
            </Menu.Item>
        )})
  }
  </Menu>
);
```

The default selection is the first region stored in the database, ordered by name. When the user clicks on a region from the dropdown menu, the viewstate is updated by the *handleChangeCoordinate* function.

```
//HANDLES THE CHANGES OF REGIONS
const handleOnChangeRegions = (selectedValue) => {
    const selected_lat = selectedValue.item.props.lat;
    const selected_lon = selectedValue.item.props.lon;
    nw[0] = selected_lon - 0.028;
    se[0] = selected_lon + 0.028;
    neweast = se[0] - 0.0379;
    se[1] = selected_lat - 0.0283;
    nw[1] = selected_lat + 0.0289;
    handleChangeCoordinate(selected_lon, selected_lat);
};
```

The function updates the map and sets the viewstate to the selected region.

4.3.2 Choose different layer types

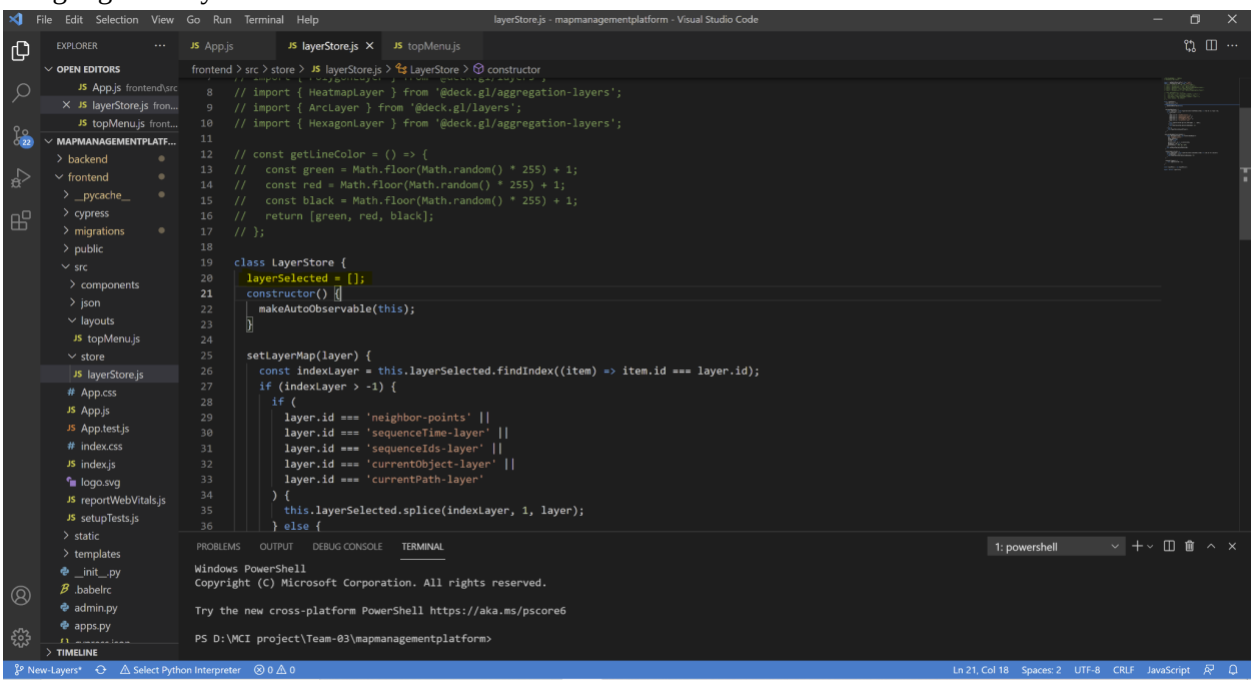
- Overview of the requirement:

There are a total of 4 layers that the user can choose.

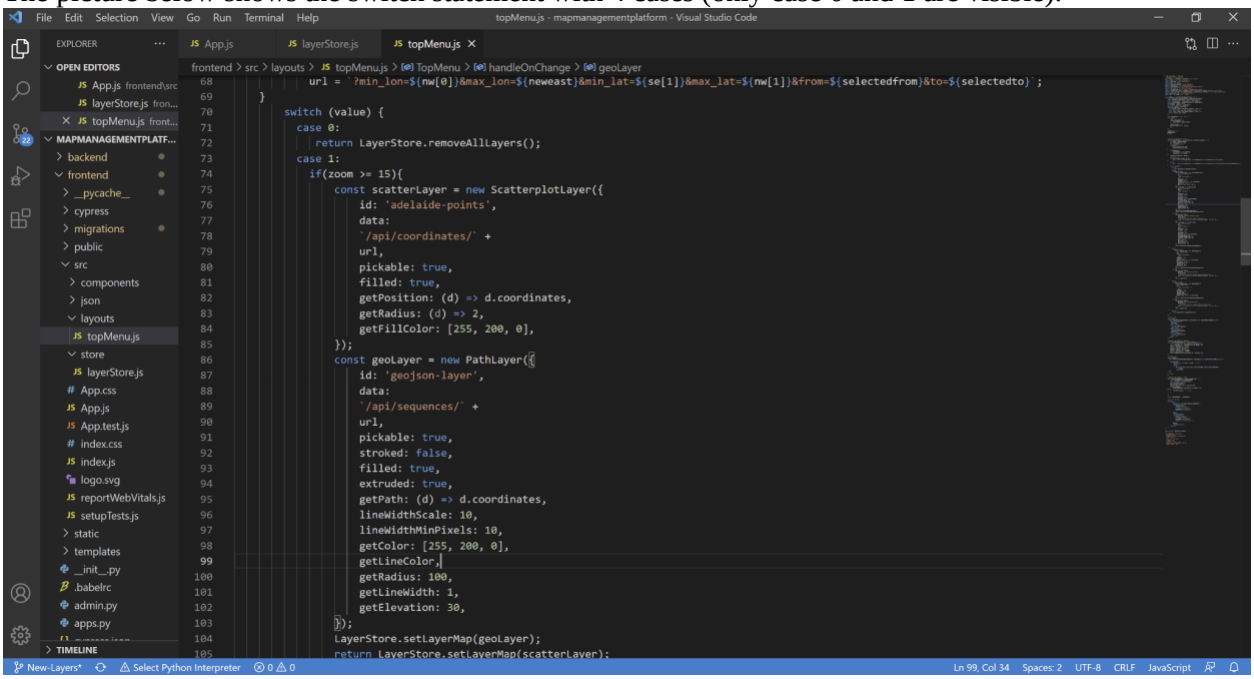
- The first layer is “No Layer”, which will remove all the current layers on the map.
- The second layer is “Location and Sequences”, which shows multiple location objects. Location objects that belong to the same sequence will be linked together by a path, which is called the sequence object. This layer supports users with many features to interact with objects. They can hover on both location or sequence objects to see their specification or click on a location object to detect nearby neighbor objects and their images.
- The third layer is “Sequence Distribution”, which is a heatmap showing distribution of sequence data.
- The fourth layer is “Location Distribution”, which displays distribution of location data in 3D hexagons. Users can hover on a hexagon object to see latitude, longitude and density of location objects.

- Technical details:

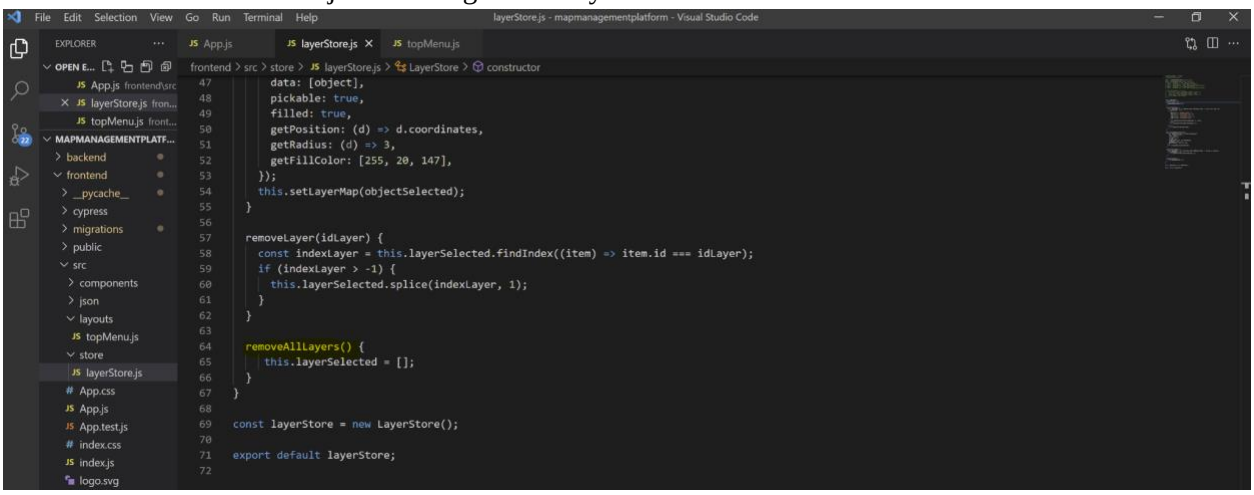
A global array and a function named layerSelected in the file layerStore.js with switch statement in file topMenu.js control the 4 layer options. “Location and Sequences”, “Sequence Distribution” and “Location Distribution” correspond to cases 1, 2 , 3, respectively. In the picture below, the global array in layerStore.js is highlighted in yellow.



The picture below shows the switch statement with 4 cases (only case 0 and 1 are visible).



When a user chooses one of these layers, a new layer object will be created based on the Deck.gl format and data from the API. This object will then be pushed into the global array to display on the map. When a user chooses "No Layer", which is case 0, this triggers a method named removeAllLayers in layerStore.js to remove all the current objects in the global array.



4.3.3 Zoom and layer interactions

- Overview of the requirement:

The user can zoom in and out of the map and then update the selected layer accordingly. Not all layers need

to be shown in all zoom levels. The specific behaviour is as follows:

- Location and Sequences: After a certain zoom level, only the Path layer will be shown hiding the ScatterPlot Layer.
- Sequence Distribution: After a certain zoom level, the layer will not be shown.
- Location Distribution: After a certain zoom level, the layer will not be shown.

In any case, there needs to be a notification to the user letting them know that the layer will not be shown.

- Technical details:

A condition is set to validate the zoom level in the *topMenu.js* to only show layers when the zoom level is greater than 15. The default zoom value is 15 and it is set in *App.js*

```
switch (value) {
  case 0:
    LoadingStore.setLoadingProcess(false);
    return LayerStore.removeAllLayers();
  case 1:
    if(zoom >= 15){
      const dataAdelaide = await axios
      .get(`/api/coordinates/` + url)
      .then((response) => {
        LoadingStore.setLoadingProcess(false);
        return response.data;
      })
      .catch((e) => {
        LoadingStore.setLoadingProcess(false);
        alert('Cannot set layer', e);
      });

    const dataGeojson = await axios
      .get(`/api/sequences/` + url)
      .then((response) => {
        // LoadingStore.setLoadingProcess(false);
        return response.data;
      })
      .catch((e) => {
        LoadingStore.setLoadingProcess(false);
        alert('Cannot set layer', e);
      });
```

In case the zoom level is greater than 15, a notification is sent to the user.

```
else{
// LoadingStore.setLoadingProcess(false);
  notification.open({
      message: 'Showing sequences only.',
      description:
        'Zoom in to see the points in the map.',
      icon: <ExclamationCircleFilled style={{ color: '#ffec3d' }} />,
  });
```

4.3.4 Select a position object

- Overview of the requirement:

When a user clicks on a position object, it highlights the onclick object with red color. All the objects belonging to the same sequence with the onclick object also change to green color and are connected via a green line. A sliding frame of image will appear at the bottom right corner of the map, which shows the image of the onclick object.

- Technical details:

Clicking on a location object will create 4 different events:

1. Firstly, a new location layer with all the objects belonging to the same sequence is created. All its properties are the same as the default location layer except the color (green instead of yellow). The below picture shows the newly created location layer of neighboring objects with the getFillColor set to green [58, 235, 52].

```
const sequenceIds = new ScatterplotLayer({
  id: 'sequenceIds-layer',
  data: `/api/coordinates/?sequence_ids=${object.sequence_key}`,
  pickable: true,
  filled: true,
  getPosition: (d) => d.coordinates,
  getRadius: (d) => 3,
  getFillColor: [58, 235, 52],
```

This layer is then pushed into a global array named layerSelected (the same array that controls different layer types). In other words, this new location layer will overlap the previous location layer. The data for each location in this layer is queried by the API:
`/api/coordinates/?sequence_ids=${object.sequence_key}`.

2. Secondly, a new sequence layer is created. This line connects all the newly created location objects. All its properties are the same as the default sequence layer except the color (green instead of yellow). The below picture shows the newly created sequence layer of neighboring objects with the getFillColor set to green [58, 235, 52].

```
const currentPath = new PathLayer({
  id: 'currentPath-layer',
  data: `/api/sequences/?ids=${object.sequence_key}`,
  pickable: true,
  stroked: false,
  filled: true,
  extruded: true,
  getPath: (d) => d.coordinates,
  lineWidthScale: 10,
  lineWidthMinPixels: 10,
  getColor: [58, 235, 52],
  getRadius: 100,
  getLineWidth: 1,
  getElevation: 40,
});
```

This layer is pushed into the global array and overlaps the newly created location layer. The data for this sequence is queried by the API: **`/api/sequences/?ids=${object.sequence_key}`**.

3. Thirdly, a new location layer with the data of the onclick object is created. This layer has the red color and is pushed into the array after the two previous layers. The data for this layer is the onclick object itself. The below picture shows the newly created location layer of the clicked objects with the getFillColor set to red [255, 0, 0].

```
const currentObject = new ScatterplotLayer({
  id: 'currentObject-layer',
  data: [object],
  pickable: true,
  filled: true,
  getPosition: (d) => d.coordinates,
  getRadius: (d) => 3,
  getFillColor: [255, 0, 0],
});
```

4. Lastly, a frame of a sliding image appears at the bottom right of the map. The first image appears is the image of the onclick object. There is a boolean variable named isShowImage in App.js to show the image. If a user clicks on a location object, it will set this bool variable to true, which shows the image. Otherwise, it sets this variable to false, which hides the image. The image is displayed by searching the image key stored in local machine with format **`/static/Images/img_${imageSlide.object.image_key}.jpg`**

4.3.5 Interact with sliding image frame

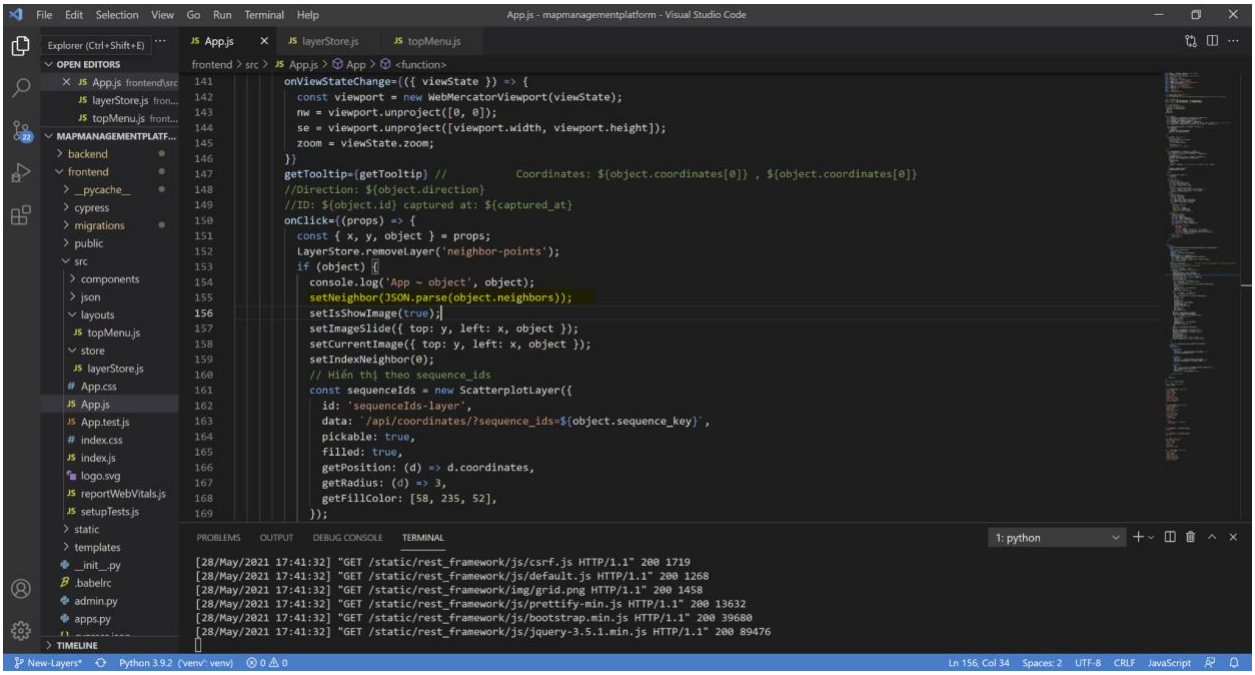
- Overview of the requirement:

When a user clicks a location object, a sliding frame of images appears at the bottom right corner. This sliding image frame has a prev and next button, which allow the user to see the relative position of the neighbor object to the onclick object on map. The sliding image frame also displays the image of the current neighbor object.

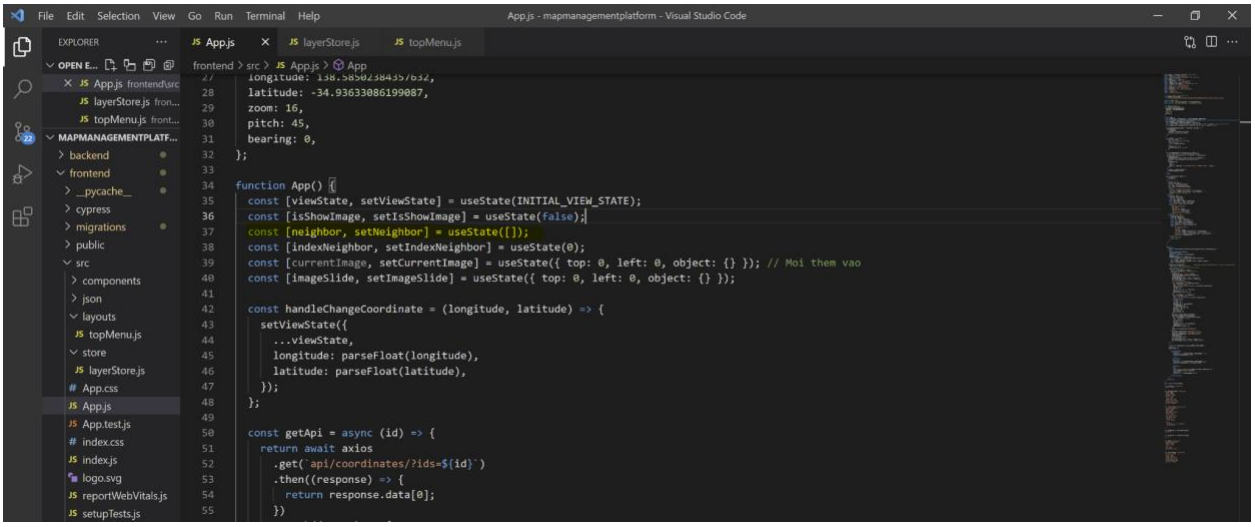
- Technical details:

Each location object in the map will have many properties, one of which is the array containing the id of nearby neighbor objects. The picture below shows the data of the location object with id 1 and the list of its neighboring objects is highlighted.

Every time a user clicks on a location object, a JSON.parse() method will parse this json data into a javascript array called neighbor, which holds the id of neighbor objects of the current onclick object.



The picture above shows that when the user clicks on a location object (the onClick at line 150), the setNeighbor method will parse the JSON data in the array neighbor. The relationship between neighbor and setNeighbor is a useState function in Hook as highlighted in the picture below.



A variable named indexNeighbor with initial value of 0 is used to control the index of the neighbor array. If the object has a neighbor array containing id of 5 neighbor objects (including itself at neighbor[0]). In this case, the sliding frame of images also has 5 images.
When a user clicks the next or prev button, it will increase or decrease the indexNeighbor value. A function named setImageNeighbor in App.js will then query the data of the neighbor object with the id corresponding to neighbor[indexNeighbor]. This function has two main tasks. The first is to get the data of an object with id neighbor[indexNeighbor] to create a new location layer with the pink color, which shows the position of the current neighbor object on map. The second task is to get the image key of the current neighbor object to display in the sliding frame.

4.3.6 Select a date range

- Overview of the requirement:

The user needs to have the ability to filter the information displayed in the map given a certain *from* date and *to* date. The layers will be filtered by such a range every time the user updates a selected range. When there are no selected dates, the layers will only be filtered by the bounding box.

- Technical details:

The *RangePicker* is part of the ant design library and provides the user the selection of a certain date range.

```
<Layout>
    <RangePicker onChange={handleChangeDebut}/>
```

Everytime the user selects a new range of dates, the values of the variables *currentfrom and currentto* is updated and then the *handleOnChange* function is called to update the layers.

```
//HANDLES THE CHANGES OF DATES
const handleChangeDebut = (range,dateStrings) => {
    const currentfrom = dateStrings[0];
    const currentto = dateStrings[1];
    setfrom(dateStrings[0]);
    setto(dateStrings[1]);
    console.log("currentfrom",currentfrom);
    if(currentfrom){
        handleOnChange(0,currentfrom,currentto,true);
    }else{
        handleOnChange(0,"","",true);
    }

};
```

The *handleOnChange* function will then validate if there is a selected date range. If there is then it will add the date range to the request url as *from* and *to.* This will generate a new request to the backend and update the layer.

```
//HANDLE THE CHANGE OF LAYERS
const handleOnChange = (e,currentfrom,currentto,validator) => {
    let value = previousLayer;
    let selectedfrom = from;
    let selectedto = to;
    if(e.key){
      value = parseInt(e.key);
      setpreviousLayer(value);
    }
    if(validator){
       selectedfrom = currentfrom;
       selectedto = currentto;
    }
    LayerStore.removeAllLayers();
    let url = '';
    if(selectedfrom.length  == 0){
       url = `?min_lon=${nw[0]}&max_lon=${neweast}&min_lat=${se[1]}&max_lat=${nw[1]}`;
    }
    else{
       url = `?min_lon=${nw[0]}&max_lon=${neweast}&min_lat=${se[1]}&max_lat=${nw[1]}&from=${selectedfrom}&to=${selectedto}`;
    }
```

4.3.7 Loading

- Overview of the requirement:

The user needs to know if data is being loaded and when the loading is completed.

- Technical details:

A LoadingStore class is defined in loadingStore.js in the folder store and to control the loading spinner. In App.js, we create a LoadingStore component and a css styled component named Loading. This css Loading component will be visible or hidden based on the boolean variable name isLoading of the LoadingStore component. If isLoading is true, it will show the loading spinner. In contrast, if isLoading is false it will hide the loading spinner.

We can set the boolean variable isLoading by the function setLoadingProcess, which is also a property of the LoadingStore component. In topMenu.js, we also create a loadingStore component to control the isLoading variable. In the switch statement of topMenu, case 1, 2 and 3 are used for loading the location & sequence, heatmap and hexagon layer respectively. In each case, the isLoading variable is set to true at the beginning , so it will show the loading spinner as the user chooses case 1,2 or 3. However, after the process of getting data from the api is completed, the isLoading is set to false by the function setLoadingProcess. This ensures the loading spinner will disappear right before the choosing layer appears on the map.

The image below shows the loading spinner before the chosen layer is fully loaded.