CM3035 - Advanced Web Development

Midterm Coursework

7th December, 2023

Table of contents

Introduction

The RESTful web application I decided to implement is a website for monitoring air quality of the Australian Capital Territory(ACT region). The project involves choosing a data set, cleaning the data for the database, implementing API endpoints to help users query the database.

Dataset & Database

Dataset

The dataset included in the project is "Air Quality Monitoring Data" provided and updated by ACT government (https://www.data.act.gov.au/Environment/Air-Quality-Monitoring-Data/94a5-zqnn/about_data). Selection of this dataset was due to few reasons:

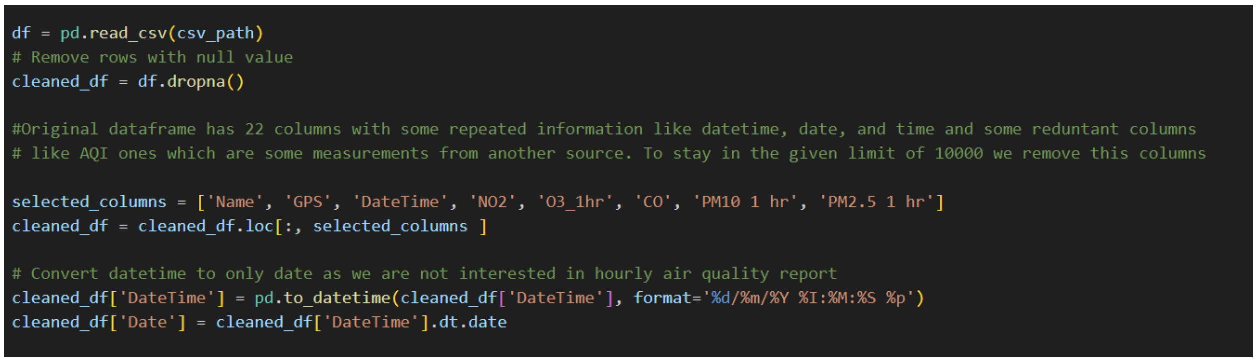
- The dataset is updated and maintained hourly by the government
- Comprehensive data and information essential to build the database
- Open data that is free to use in not commercial projects

Columns in the dataset include the name of the location and GPS where measurements were taken, DateTime, and measurements taken from two different sources (National

Environment Protection Measures (NEPM) and an Air Quality Index (AQI)).

Cleaning the data

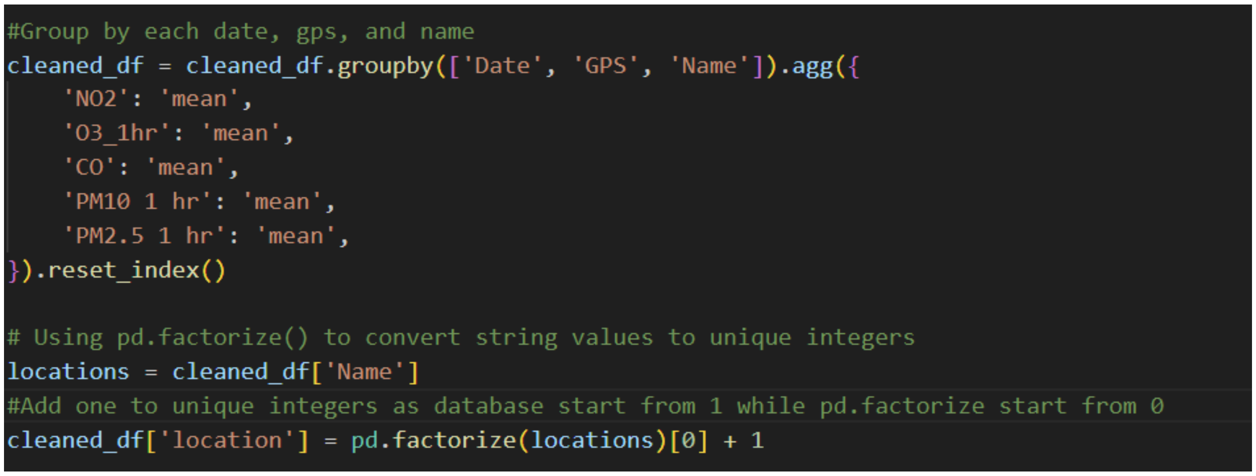
For cleaning purposes, firstly rows with Null or empty fields were removed. After that panda dataframe was built from columns with the relevant data. Removed columns include data and time columns since we already have datetime columns with the same info and measurements taken from AQI which is almost identical to data collected from (NEPM).



```
df = pd.read_csv(csv_path)
# Remove rows with null value
cleaned_df = df.dropna()

#Original dataframe has 22 columns with some repeated information like datetime, date, and time and some reduntant columns
# like AQI ones which are some measurements from another source. To stay in the given limit of 10000 we remove this columns

selected_columns = ['Name', 'GPS', 'DateTime', 'NO2', 'O3_1hr', 'CO', 'PM10 1 hr', 'PM2.5 1 hr']
cleaned_df = cleaned_df.loc[:, selected_columns ]

# Convert datetime to only date as we are not interested in hourly air quality report
cleaned_df['DateTime'] = pd.to_datetime(cleaned_df['DateTime'], format='%d/%m/%Y %I:%M:%S %p')
cleaned_df['Date'] = cleaned_df['DateTime'].dt.date
```

(code piece taken from helper.py in scripts folder)

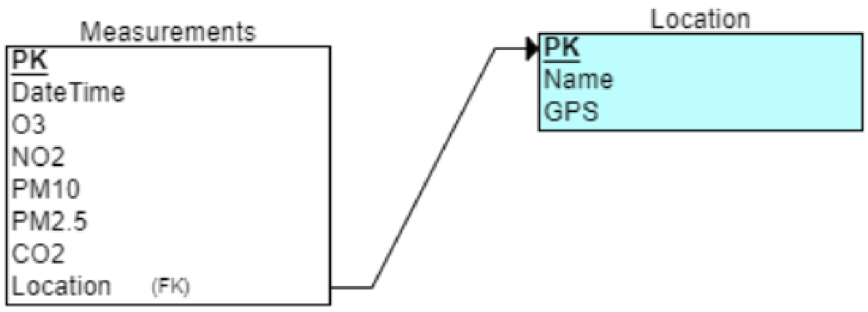
After this hourly measurements were averaged together by their date and location name to create daily average air quality measurement



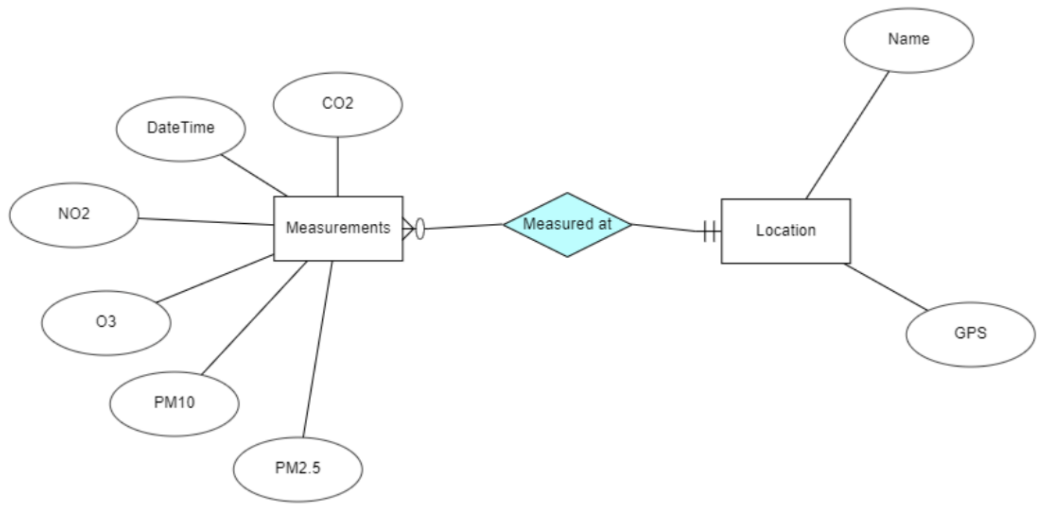
```
#Group by each date, gps, and name
cleaned_df = cleaned_df.groupby(['Date', 'GPS', 'Name']).agg({
    'NO2': 'mean',
    'O3_1hr': 'mean',
    'CO': 'mean',
    'PM10 1 hr': 'mean',
    'PM2.5 1 hr': 'mean',
}).reset_index()

# Using pd.factorize() to convert string values to unique integers
locations = cleaned_df['Name']
#Add one to unique integers as database start from 1 while pd.factorize start from 0
cleaned_df['location'] = pd.factorize(locations)[0] + 1
```

Database

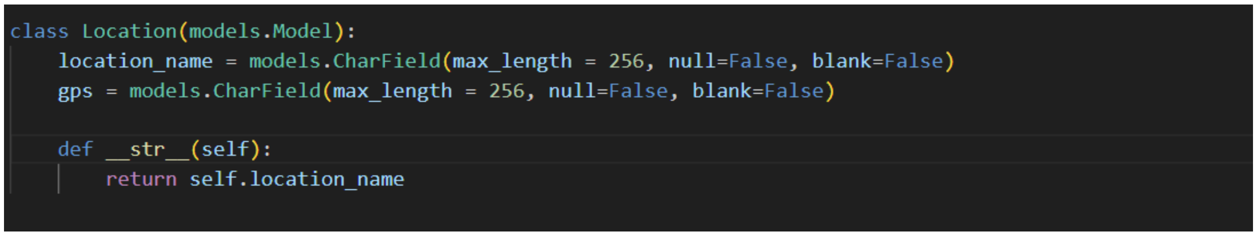
The SQLite3 database for this project has been constructed using Django migration and models. The database contains two tables: “Location” and “Measurements”. Here is relation schema between tables:



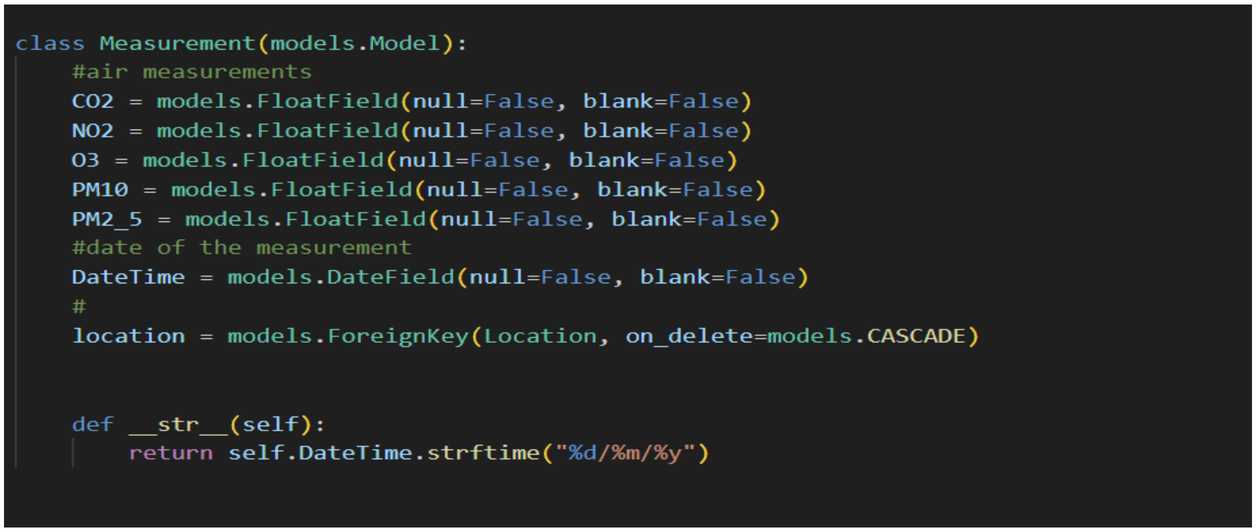
Entity Relation Diagram:



Django models code:

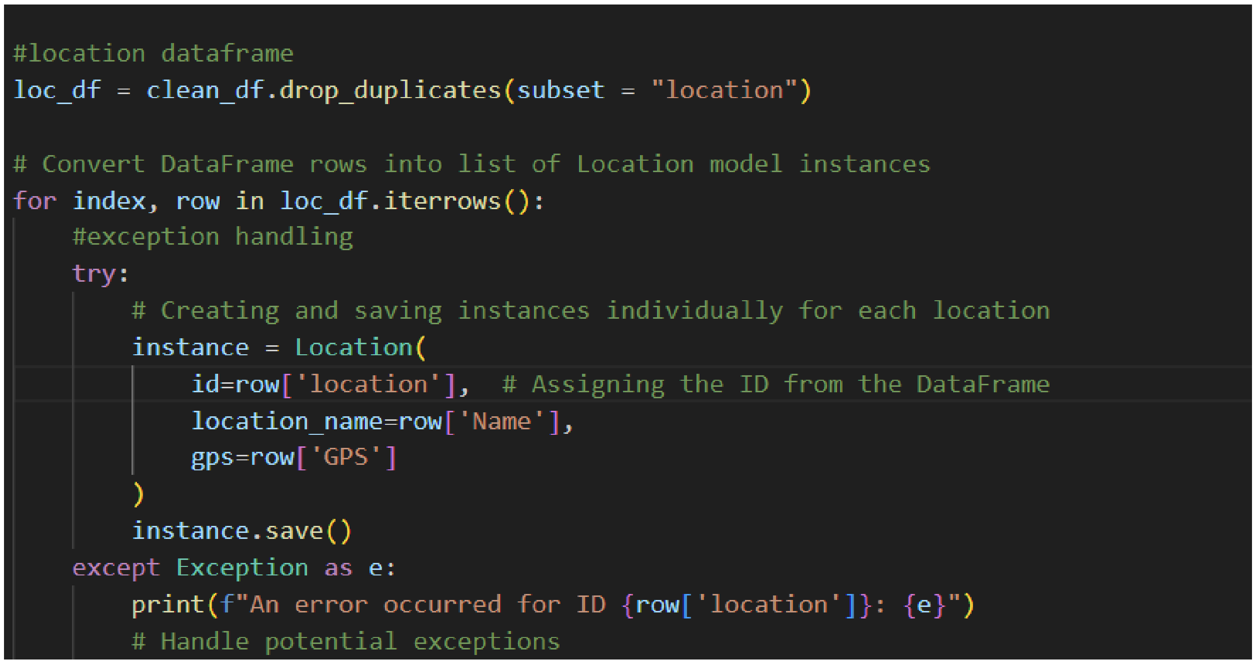


```python
class Location(models.Model):
    location_name = models.CharField(max_length = 256, null=False, blank=False)
    gps = models.CharField(max_length = 256, null=False, blank=False)

    def __str__(self):
        return self.location_name
```



```python
class Measurement(models.Model):
    #air measurements
    CO2 = models.FloatField(null=False, blank=False)
    NO2 = models.FloatField(null=False, blank=False)
    O3 = models.FloatField(null=False, blank=False)
    PM10 = models.FloatField(null=False, blank=False)
    PM2_5 = models.FloatField(null=False, blank=False)
    #date of the measurement
    DateTime = models.DateField(null=False, blank=False)
    #
    location = models.ForeignKey(Location, on_delete=models.CASCADE)


    def __str__(self):
        return self.DateTime.strftime("%d/%m/%y")
```

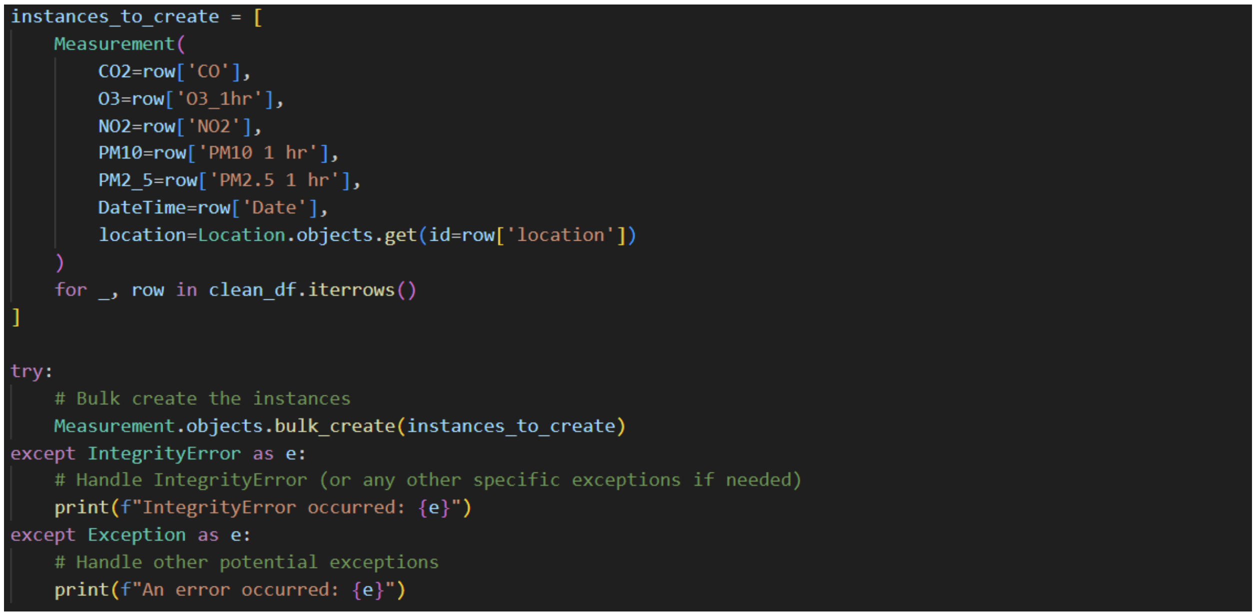
Then the database had been populated from the dataset using the populate.py script file. Location table instances have been created and saved individually using a new data frame created from our cleaned dataframe using .drop_duplicates() panda function.



```python
#location dataframe
loc_df = clean_df.drop_duplicates(subset = "location")

# Convert DataFrame rows into list of Location model instances
for index, row in loc_df.iterrows():
    #exception handling
    try:
        # Creating and saving instances individually for each location
        instance = Location(
            id=row['location'],  # Assigning the ID from the DataFrame
            location_name=row['Name'],
            gps=row['GPS']
        )
        instance.save()
    except Exception as e:
        print(f"An error occurred for ID {row['location']}: {e}")
        # Handle potential exceptions
```

(code piece from populate.py in scripts folder)

However, the Measurement table has been populated using the bulk_create function to reduce the run time as there are over 3000 rows in the dataframe.



```
instances_to_create = [
    Measurement(
        CO2=row['CO'],
        O3=row['O3_1hr'],
        NO2=row['NO2'],
        PM10=row['PM10 1 hr'],
        PM2_5=row['PM2.5 1 hr'],
        DateTime=row['Date'],
        location=Location.objects.get(id=row['location'])
    )
    for _, row in clean_df.iterrows()
]

try:
    # Bulk create the instances
    Measurement.objects.bulk_create(instances_to_create)
except IntegrityError as e:
    # Handle IntegrityError (or any other specific exceptions if needed)
    print(f"IntegrityError occurred: {e}")
except Exception as e:
    # Handle other potential exceptions
    print(f"An error occurred: {e}")
```

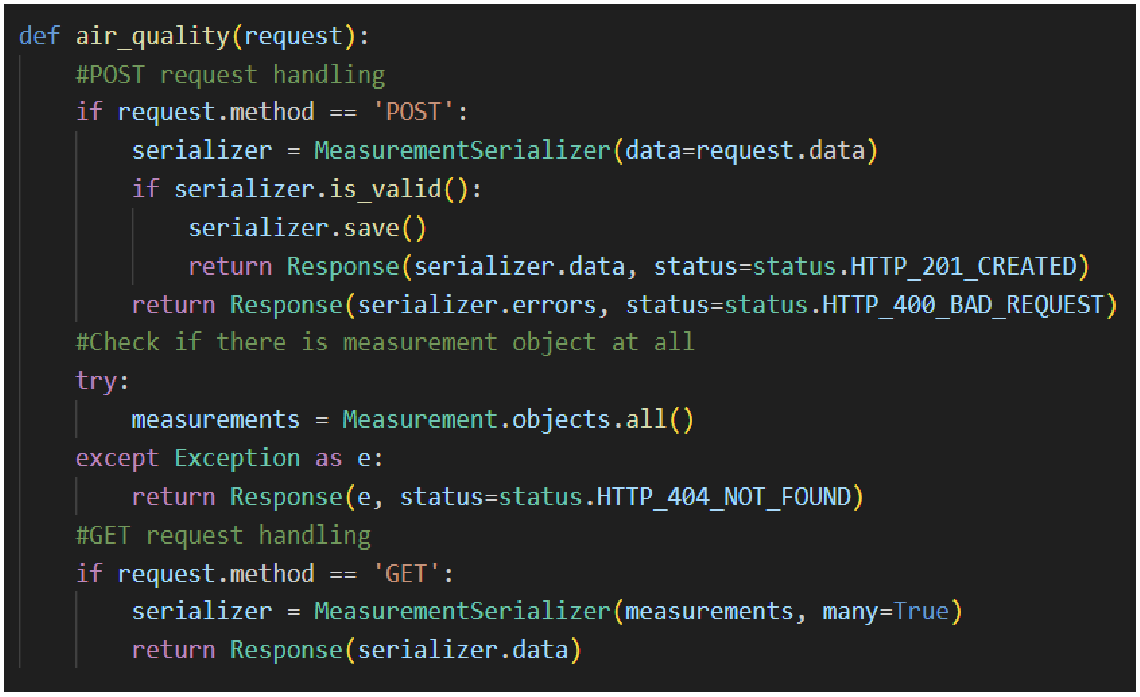
(code piece from populate.py in scripts folder)

API Endpoints

API points

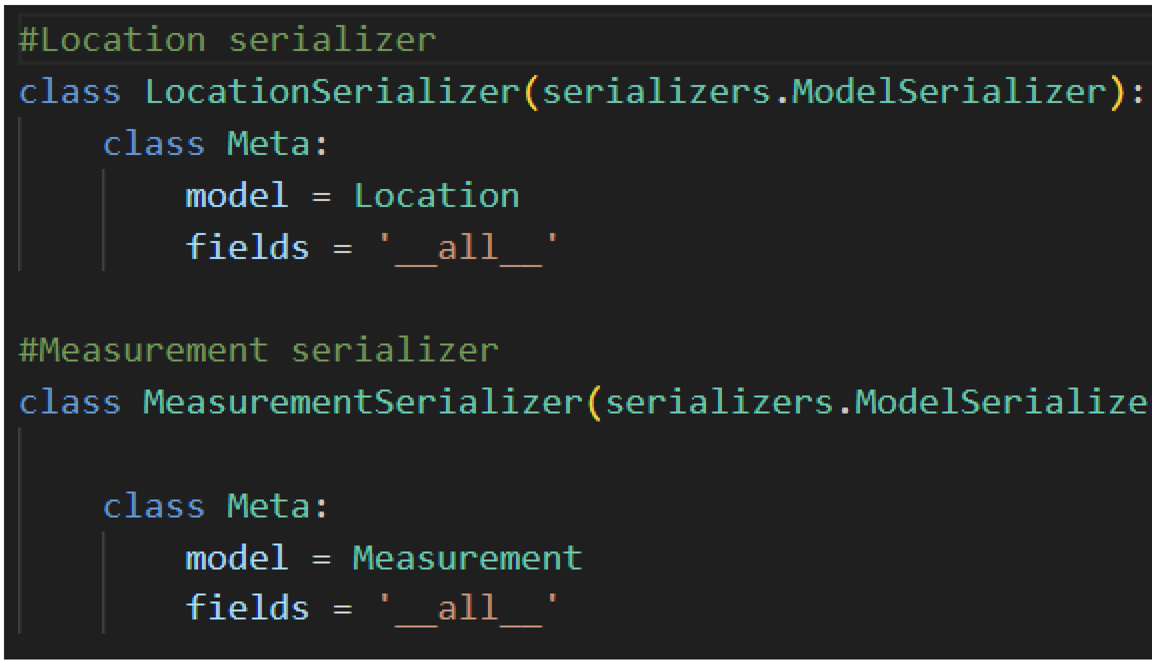
1. **air_quality:**

   API point that receives GET and POST request that shows all the data in the Measurement table and also creates new measurements using given information from the user.



```
def air_quality(request):
    #POST request handling
    if request.method == 'POST':
        serializer = MeasurementSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    #Check if there is measurement object at all
    try:
        measurements = Measurement.objects.all()
    except Exception as e:
        return Response(e, status=status.HTTP_404_NOT_FOUND)
    #GET request handling
    if request.method == 'GET':
        serializer = MeasurementSerializer(measurements, many=True)
        return Response(serializer.data)
```

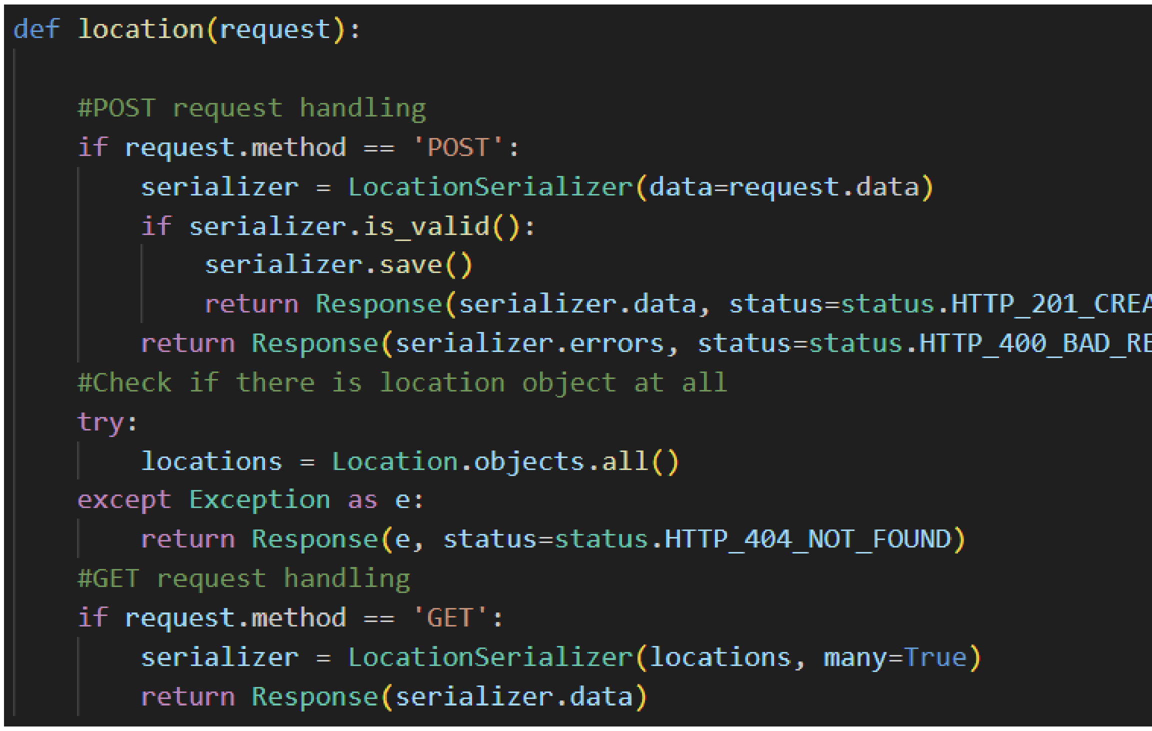
Handles exceptions using try and except functions and validates the new data using serializer.



```
#Location serializer
class LocationSerializer(serializers.ModelSerializer):
    class Meta:
        model = Location
        fields = '__all__'

#Measurement serializer
class MeasurementSerializer(serializers.ModelSerialize

    class Meta:
        model = Measurement
        fields = '__all__'
```

2. location:

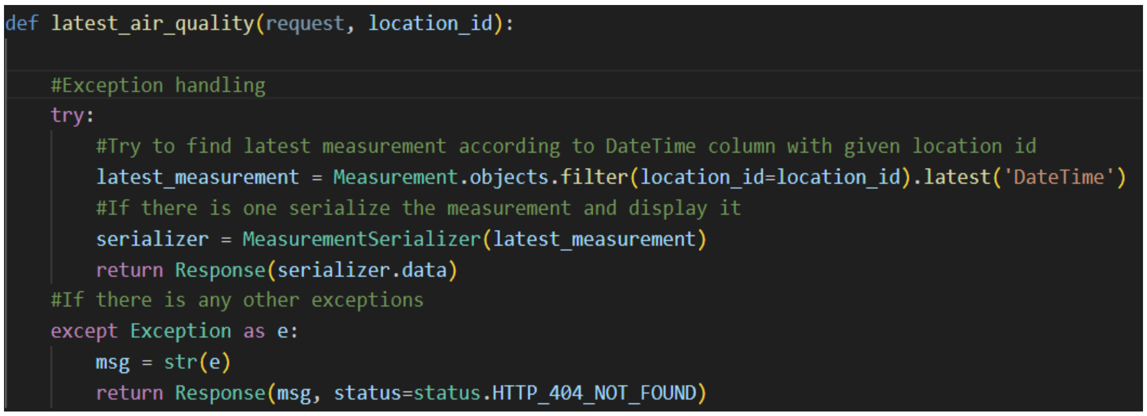
API point that receives GET and POST request that shows all the data in the Location table and also creates new location using given information.



```
def location(request):

    #POST request handling
    if request.method == 'POST':
        serializer = LocationSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREA
        return Response(serializer.errors, status=status.HTTP_400_BAD_RE
    #Check if there is location object at all
    try:
        locations = Location.objects.all()
    except Exception as e:
        return Response(e, status=status.HTTP_404_NOT_FOUND)
    #GET request handling
    if request.method == 'GET':
        serializer = LocationSerializer(locations, many=True)
        return Response(serializer.data)
```

3. latest_air_quality:

API that shows latest air quality measurements for a specific location using parameters passed through URL. Useful when user want to quickly display the most recent air quality.


```
def latest_air_quality(request, location_id):

    #Exception handling
    try:
        #Try to find latest measurement according to DateTime column with given location id
        latest_measurement = Measurement.objects.filter(location_id=location_id).latest('DateTime')
        #If there is one serialize the measurement and display it
        serializer = MeasurementSerializer(latest_measurement)
        return Response(serializer.data)
    #If there is any other exceptions
    except Exception as e:
        msg = str(e)
        return Response(msg, status=status.HTTP_404_NOT_FOUND)
```

Here, database is queried using location_id which passed through URL like this: air-quality/location/<int:location_id>

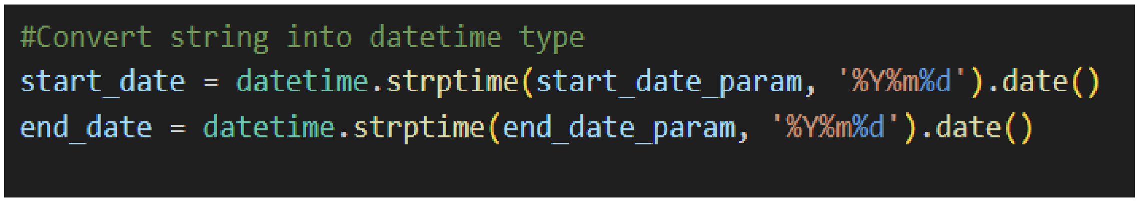
4. air_quality_average:

This API displays average air quality measurements recorded between two dates using GET requests. These two dates passed to this function using query string in the URL e.g:

/air-quality/average/average?start_date={start}&end_date={end}

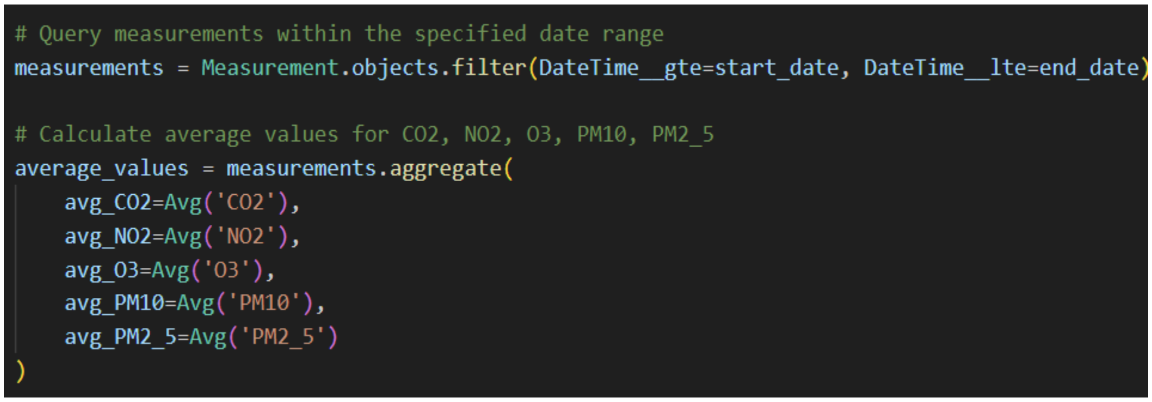
http://54.85.116.86/air-quality/average/average?start_date=20230301&end_date=20230402

Then we turn this two string parameters into datetime type:



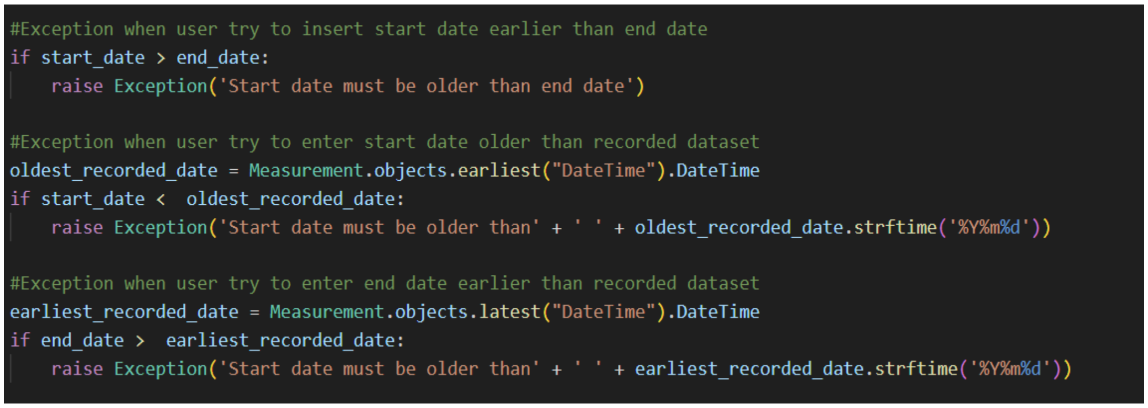
```
#Convert string into datetime type
start_date = datetime.strptime(start_date_param, '%Y%m%d').date()
end_date = datetime.strptime(end_date_param, '%Y%m%d').date()
```

Then we query the database using the datetime variables and average the measurement data:



```
# Query measurements within the specified date range
measurements = Measurement.objects.filter(DateTime__gte=start_date, DateTime__lte=end_date)

# Calculate average values for CO2, NO2, O3, PM10, PM2_5
average_values = measurements.aggregate(
    avg_CO2=Avg('CO2'),
    avg_NO2=Avg('NO2'),
    avg_O3=Avg('O3'),
    avg_PM10=Avg('PM10'),
    avg_PM2_5=Avg('PM2_5')
)
```

Errors have been handled using try and except with custom errors for each possible bad URL:



```
#Exception when user try to insert start date earlier than end date
if start_date > end_date:
    raise Exception('Start date must be older than end date')

#Exception when user try to enter start date older than recorded dataset
oldest_recorded_date = Measurement.objects.earliest("DateTime").DateTime
if start_date <  oldest_recorded_date:
    raise Exception('Start date must be older than' + ' ' + oldest_recorded_date.strftime('%Y%m%d'))

#Exception when user try to enter end date earlier than recorded dataset
earliest_recorded_date = Measurement.objects.latest("DateTime").DateTime
if end_date >  earliest_recorded_date:
    raise Exception('Start date must be older than' + ' ' + earliest_recorded_date.strftime('%Y%m%d'))
```

5. air_quality_history:

API link for showing historical information for given location and date. API function takes two parameters from URL; e.g: /air-quality/history/{location}?date={date}

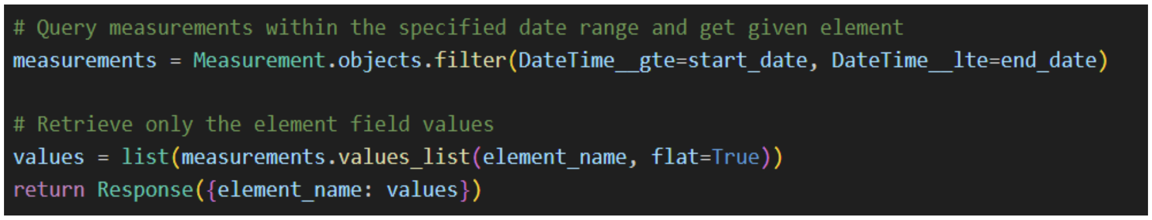
Date parameter is converted from string to datetime like last API and database queried using it.

6. air_quality_element:

Useful API for getting information on quantity of a single element in the air. URL passes three parameters using query string; e.g:

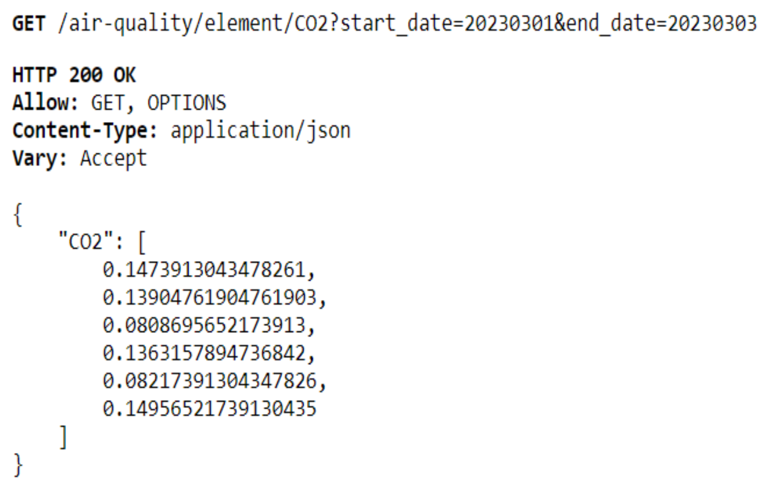
/air-quality/element/{element}?start_date={start}&end_date={end}

API takes the two date parameters and converts them into datetime types and query the database using all two parameters: start_date and end_date. Then, only the values related to {element} parameter are shown to the user.



```
# Query measurements within the specified date range and get given element
measurements = Measurement.objects.filter(DateTime__gte=start_date, DateTime__lte=end_date)

# Retrieve only the element field values
values = list(measurements.values_list(element_name, flat=True))
return Response({element_name: values})
```

Example API JSON output:

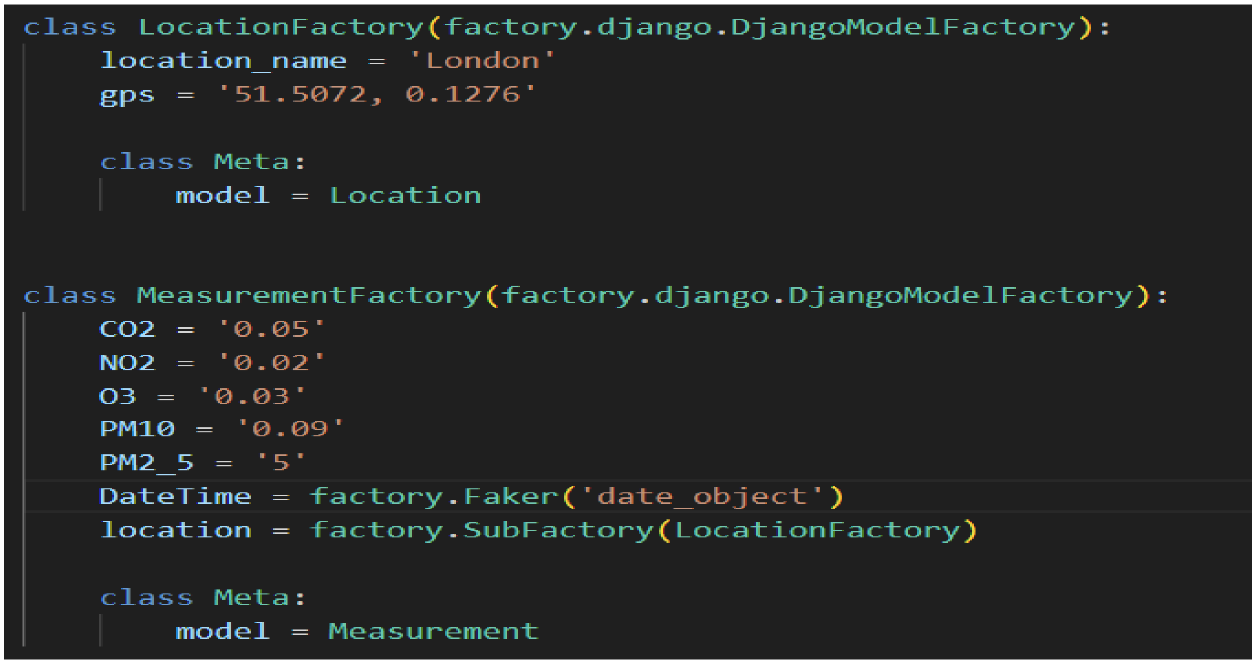


```
GET /air-quality/element/CO2?start_date=20230301&end_date=20230303

HTTP 200 OK
Allow: GET, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "CO2": [
        0.1473913043478261,
        0.13904761904761903,
        0.0808695652173913,
        0.1363157894736842,
        0.08217391304347826,
        0.14956521739130435
    ]
}
```

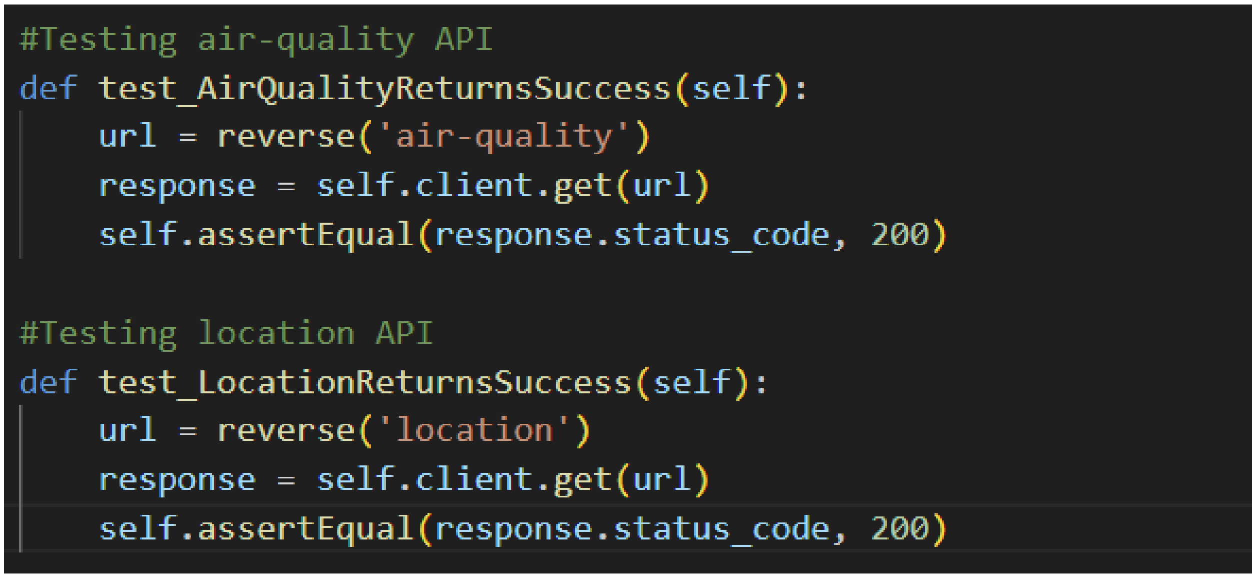
Unit tests:

Each API endpoint has been checked at least once by unit tests written in APITestCase in tests.py. Tests can be run by just using “python manage.py test” without any additional step assuming all the required packages are installed.

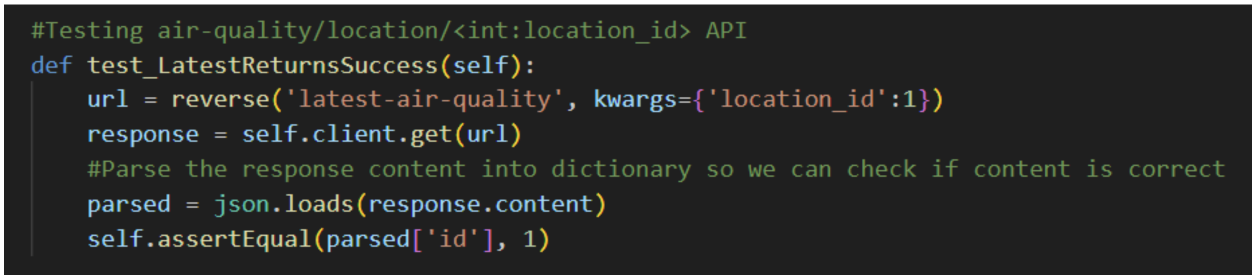
Fake data used in testing has been produced using factory_boy package:


```python
class LocationFactory(factory.django.DjangoModelFactory):
    location_name = 'London'
    gps = '51.5072, 0.1276'

    class Meta:
        model = Location


class MeasurementFactory(factory.django.DjangoModelFactory):
    CO2 = '0.05'
    NO2 = '0.02'
    O3 = '0.03'
    PM10 = '0.09'
    PM2_5 = '5'
    DateTime = factory.Faker('date_object')
    location = factory.SubFactory(LocationFactory)

    class Meta:
        model = Measurement
```

and used in the setUp of our APITestCase function (tests.py line 19-25). As for unit tests, some of them check if the API url is returning correct response status code; eg: 200


```python
#Testing air-quality API
def test_AirQualityReturnsSuccess(self):
    url = reverse('air-quality')
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)

#Testing location API
def test_LocationReturnsSuccess(self):
    url = reverse('location')
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
```

More sophisticated unit tests check if API is returning correct information in JSON form. In this piece of code we check if latest_air_quality API returns correct measurement data:



```python
#Testing air-quality/location/<int:location_id> API
def test_LatestReturnsSuccess(self):
    url = reverse('latest-air-quality', kwargs={'location_id':1})
    response = self.client.get(url)
    #Parse the response content into dictionary so we can check if content is correct
    parsed = json.loads(response.content)
    self.assertEqual(parsed['id'], 1)
```

Unpackage & Running

Required packages to run:

asgiref==3.7.2

Django==5.0

djangorestframework==3.14.0

factory-boy==3.3.0

Faker==20.1.0

numpy==1.26.2

pandas==2.1.3

python-dateutil==2.8.2

pytz==2023.3.post1

six==1.16.0

sqlparse==0.4.4

tzdata==2023.3

Development Environment:

OS: Windows 11

Python version: 3.11.5

Super user information:

http://127.0.0.1:8000/admin for localhost

http://54.85.116.86/admin for AWS deployed website

Username: admin

Password: london1

Test and database populating:

- Testing units can be found in tests.py file provided by django in /actAir folder and helper fake data factories in model_factories.py
- Database populating script can be found in /actAir/scripts folder named populate.py. Database will be already populated; it can be done again by

just running the populate.py. No need to run the helper.py file in the scripts folder.

Deployment

Project has been deployed on AWS EC2 instance running Ubuntu Linux using nginx and gunicorn packages.

IP4 address: http://107.21.56.13/

Administrator login credentials are the same as localhost projects as stated in the “Super user information” section.

Conclusion

Overall, the project was successful in shaping and cleaning dataset, implementing database structure and populating with cleaned data, and serving users with those data using a variety of interesting and complex queries.

In particular, database query complexity for API endpoints extracting up to three parameters from URL has been successful. The robustness of these endpoints using try/catch to prevent application crashes and extensive exception handling also have been noteworthy. However, there are other areas where the project could improve such as:

- Designing more complex and sophisticated databases. In our current implementation we only have two simple table with one-to-many relation
- Creating template and view functions to serve users more stylised data not just JSON

That being said, I believe this is a competent and complex application that will meet the given requirements