Coding Exercise (WeatherAPI)
Submitted by Shifat Mithila

Problem 1 - Data Modeling

Choose a database to use for this coding exercise (SQLite, Postgres, etc.). Design a data model to represent the weather data records. If you use an ORM, your answer should be in the form of that ORM's data definition format. If you use pure SQL, your answer should be in the form of DDL statements.

—----------

Answer: I am using **Django ORM** with **sqlite3** database. The data definition format is used for designing data models (reference in code: *weatherProject/weatherApp/models.py*).

Example:

```
class WeatherData(models.Model):
    """
    Model for storing all the weather data
    """
    stationID = models.CharField(max_length=20)
    date = models.DateField()
    year = models.IntegerField()
    month = models.IntegerField()
    day = models.IntegerField()
    maxTemperature = models.IntegerField()    # data in tenth of deg C
    minTemperature = models.IntegerField()    # data in tenth of deg C
    precipitation_mm = models.IntegerField()    # data in tenth of mm

    def __str__(self):
```

```
        return str(self.stationID)+" "+str(self.date)
```

Problem 2 - Ingestion

Write code to ingest the weather data from the raw text files supplied into your database, using the model you designed. Check for duplicates: if your code is run twice, you should not end up with multiple rows with the same data in your database. Your code should also produce log output indicating start and end times and number of records ingested.

Answer: In *weatherProject/weatherApp/views.py*, the **get** method in class **WeatheringestView(APIView)** is written to ingest data from raw text files in wx_data directory (provided). Data is ingested to the **WeatherData** model. Each filename is used as a stationID for each station.

Duplicates are checked in models.py, in WeatherData model with **UniqueConstraint(..)** for unique fields stationID and date.

```
class WeatherData(models.Model):
    """
    Model for storing all the weather data
    """
    stationID = models.CharField(max_length=20)
    date = models.DateField()
    year = models.IntegerField()
    month = models.IntegerField()
    day = models.IntegerField()
    maxTemperature = models.IntegerField()    # data in tenth of deg C
    minTemperature = models.IntegerField()    # data in tenth of deg C
    precipitation_mm = models.IntegerField() # data in tenth of mm
```

```
    class Meta:
        """
        For duplicate check on data entry UniqueConstraint() is used
for unique fields stationID and date
        """
        constraints = [
            models.UniqueConstraint(fields=['stationID', 'date'],
                                    name='unique_stationID_date')
        ]

    def __str__(self):
        return str(self.stationID)+" "+str(self.date)
```

The get method in class **WeatheringestView(APIView)** in *weatherApp/views.py* also produces log output and stores them in the **WeatherLog** model. This log has start and end times and number of records for each stationID. Duplicate data entry is prevented with **UniqueConstraint(..)** for unique field stationID.

```
class WeatherLog(models.Model):
    """
    Model for storing all the weather data logs, for each
file/station
    """
    stationID = models.CharField(max_length=20)
    startTime = models.CharField(max_length=50)
    endTime = models.CharField(max_length=50)
    numberOfRecords = models.IntegerField()

    class Meta:
        """
        For duplicate check on data entry UniqueConstraint() is used
for unique field stationID
        """
        constraints = [
            models.UniqueConstraint(fields=['stationID'],
                                    name='unique_stationID')
```

```
        ]

    def __str__(self):
        return str(self.stationID)+" "+str(self.numberOfRecords)
```

You can perform data ingestion and produce logs and store in the database with this url: http://127.0.0.1:8888/weather/ingestdata/
(Please, change your port number accordingly)

Problem 3 - Data Analysis

For every year, for every weather station, calculate:

*** Average maximum temperature (in degrees Celsius)**
*** Average minimum temperature (in degrees Celsius)**
*** Total accumulated precipitation (in centimeters)**

Ignore missing data when calculating these statistics.

Design a new data model to store the results. Use NULL for statistics that cannot be calculated.

Your answer should include the new model definition as well as the code used to calculate the new values and store them in the database.
——————————————————————————————-

Answer: In *weatherProject/weatherApp/views.py*, the **get** method in class **WeatheranalysisView(APIView)** is written for the calculation of average maximum and minimum temperatures, and total precipitation for every station and

for every year; along with unit conversion and storing the calculated data in a new model **WeatherStatistics**.

Calculated data is stored in the datatable using **bulk_create()** to handle large size data and **ignore_conflicts= True** is used to handle errors thrown from **UniqueConstraint()** used in the model for duplicate check on data entry. Duplicate data entry is prevented with **UniqueConstraint(..)** for unique fields stationID and year.

To ignore missing data, **exclude()** function is used. Null is used for statistics that can not be calculated.

Data model to store the calculated results:

```
class WeatherStatistics(models.Model):
    """
    Model for storing all the weather calculation for each
file/station, for each year
    """
    stationID = models.CharField(max_length=20)
    year = models.IntegerField()
    avgMaxTemperature = models.FloatField(
        null=True, blank=True, default=None)    # data in degree C
    avgMinTemperature = models.FloatField(
        null=True, blank=True, default=None)    # data in degree C
    totalPrecipitation_cm = models.FloatField(
        null=True, blank=True, default=None)    # data in cm

    class Meta:
        """
        For duplicate check on data entry UniqueConstraint() is used
for unique fields stationID and year
        """
        constraints = [
            models.UniqueConstraint(fields=['stationID', 'year'],
                                    name='unique_stationID_year')
        ]
```

```
    def __str__(self):
        return str(self.stationID)+" "+str(self.year)
```

Code for calculation:

```
class WeatheranalysisView(APIView):
    """
    Calculates weather data statistics when requested from
/weather/stats url. The calculations are performed and saved in the
database (WeatherStatistics).
    """
    @ classmethod
    def get_extra_actions(cls):
        return []

    def get(self, request, format=None):
        # for every station and for every year
        # calculate averages for minimum and maximum temperatures
        # calculate total precipitation
        # exclude() function ignores missing data
        queryset = WeatherData.objects.values(
            'stationID', 'year').distinct().exclude(
            maxTemperature=-9999).exclude(
            minTemperature=-9999).exclude(
            precipitation_mm=-9999).annotate(
            avgmaxTemp=Avg('maxTemperature'),
            avgminTemp=Avg('minTemperature'),
            totalPrecip=Sum('precipitation_mm'))

        dataList = []
        for item in queryset.iterator():
            # unit conversion
            # average temperatures are divided by 10
            # to convert from "in tenth degree Celsius" to "degree
Celsius" and
```

```
            # total precipitation is divided by 100 to convert from
tenth of "Millimeters"
            # to "Centimeters", 1 mm = 0.1cm, 0.1mm = 0.01cm
            # all values are rounded up to 3 decimal places

            data = {"stationID": item['stationID'],
                    "year": item['year'],
                    "avgMaxTemperature":
round((item['avgmaxTemp']/10), 3),
                    "avgMinTemperature":
round((item['avgminTemp']/10), 3),
                    "totalPrecipitation_cm":
round((item['totalPrecip']/100), 3)}
            dataList.append(WeatherStatistics(**data))

        # calculated data is ingested using bulk_cereate to handle
large data
        # ignore_conflicts = True is used to handle error thrown for
duplicate check
        # from UniqueConstraints used in the model
        WeatherStatistics.objects.bulk_create(dataList,
ignore_conflicts=True)

        return Response({"status": "success"},
                        status.HTTP_201_CREATED)
```

You can perform data calculation and storing with
http://127.0.0.1:8888/weather/calc/

Problem 4 - REST API

Choose a web framework (e.g. Flask, Django REST Framework). Create a REST API with the following GET endpoints:

/api/weather
/api/weather/stats

Both endpoints should return a JSON-formatted response with a representation of the ingested/calculated data in your database. Allow clients to filter the response by date and station ID (where present) using the query string. Data should be paginated.

Include a Swagger/OpenAPI endpoint that provides automatic documentation of your API.

Your answer should include all files necessary to run your API locally, along with any unit tests.

Answer: The REST API is created using Django REST Framework with the following GET endpoints:

- /api/weather
- /api/weather/log
- /api/weather/stats

/api/weather that returns a JSON-formatted response with a representation of the ingested data in your database (WeatherData).

```
GET /api/weather/

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

{
    "count": 1729957,
    "next": "http://127.0.0.1:8888/api/weather/?limit=10&offset=10",
    "previous": null,
    "results": [
        {
            "id": 1894790,
            "stationID": "USC00110072",
            "date": "1985-01-01",
            "year": 1985,
```

```
            "month": 1,
            "day": 1,
            "maxTemperature": -22,
            "minTemperature": -128,
            "precipitation_mm": 94
        },
        {
```


Clients can filter the response by **date and stationID** using the query string. Data is paginated with page size 10.

```
GET /api/weather/?stationID=USC00110072&date=2011-11-11

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

{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 1904508,
            "stationID": "USC00110072",
            "date": "2011-11-11",
            "year": 2011,
            "month": 11,
            "day": 11,
            "maxTemperature": 44,
            "minTemperature": -39,
            "precipitation_mm": 0
        }
```

/api/weather/log that returns a JSON-formatted response with a representation of the calculated data log with start and end time and number of records in your database (WeatherLog).

```
GET /api/weather/log/

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

{
    "count": 167,
    "next": "http://127.0.0.1:8888/api/weather/log/?limit=10&offset=10",
    "previous": null,
    "results": [
        {
            "id": 216,
```

```
            "stationID": "USC00110072",
            "startTime": "1679539015.4599073",
            "endTime": "1679539016.694612",
            "numberOfRecords": 10865
        },
```


Clients can filter the response by **stationID** using the query string. Data is paginated with page size 10.

```
GET /api/weather/log/?stationID=USC00110072

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

{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 216,
            "stationID": "USC00110072",
            "startTime": "1679539015.4599073",
            "endTime": "1679539016.694612",
            "numberOfRecords": 10865
        }
    ]
}
```

/api/weather/stats that returns a JSON-formatted response with a representation of the calculated data in your database (WeatherStatistics).

```
GET /api/weather/stats/

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

{
    "count": 4791,
    "next": "http://127.0.0.1:8888/api/weather/stats/?limit=10&offset=10",
    "previous": null,
    "results": [
        {
            "id": 143915,
            "stationID": "USC00110072",
            "year": 1985,
            "avgMaxTemperature": 15.375,
            "avgMinTemperature": 4.326,
            "totalPrecipitation_cm": 77.43
```

```
        },
```

Clients can filter the response by **year and stationID** using the query string. Data is paginated with page size 10.

```
GET /api/weather/stats/?stationID=USC00110072&year=2011

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

{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 143941,
            "stationID": "USC00110072",
            "year": 2011,
            "avgMaxTemperature": 15.389,
            "avgMinTemperature": 4.495,
            "totalPrecipitation_cm": 85.73
        }
    ]
}
```

For automatic documentation of our API, use endpoint:
http://127.0.0.1:8888/api/v1/swagger/schema/

Testing:
WeatherApp/tests.py is written for all the unit tests. 3 tests are done with 3 test classes called **TestWeatherData(APITestCase), TestWeatherLog(APITestCase) ,** and **TestWeatherStatistics(APITestCase)** for 3 endpoints: **api/weather, api/weather/log,** and **api/weather/stats** respectively. For example, the Code snippet below shows the TestWeatherData class written for testing endpoint /api/weather.

```
class TestWeatherData(APITestCase):
    """
    Unit testing for /api/weather endpoint
    """
    url = "/api/weather/"
```

```
    def setUp(self):
        WeatherData.objects.create(stationID="USC0001",
date="2020-04-20", year=2020, month=4, day=20, maxTemperature=-111,
minTemperature=-217, precipitation_mm=94)

    def test_get_weather(self):

        response = self.client.get(self.url)
        result = response.json()
        print(result)

        self.assertEqual(response.status_code, 200)
        self.assertIsInstance(result['results'], list)
        self.assertEqual(result['results'][0]["stationID"],
"USC0001")
```

For running the API locally, clone the project from:
https://github.com/Shifat11420/weather-API

Extra Credit - Deployment

(Optional.) Assume you are asked to get your code running in the cloud using AWS. What tools and AWS services would you use to deploy the API, database, and a scheduled version of your data ingestion code? Write up a description of your approach.

Answer:
To get my code running or deploy my Django REST API project in the cloud using AWS, I would need few AWS services and tools such as:

- **Amazon Elastic Beanstalk** to deploy and scale web applications
- **Amazon RDS** (Relational Database Service) for database
- **Amazon VPC** (Virtual Private Cloud) for virtual network
- **Amazon Lambda** for scheduling data ingestion
- **CloudWatch EventBridge** for setting up a trigger for Amazon Lambda
- **Amazon SES** (Simple Email Service) to verify if Lambda is getting triggered periodically

The following description gives an overall idea of how we can deploy our Django REST API in AWS cloud.

Virtual network:

To start with, we will make sure that we already have a default VPC or, create a default VPC in case we don't have one. **Amazon VPC or amazon Virtual Private Cloud** enables us to launch aws resources into our defined private network. We are going to deploy our app and create an Amazon RDS database in the default VPC. This virtual network closely resembles a traditional network where we will operate in our own data center. This facilitates the use of scalable infrastructure of AWS.

Security groups:

Next step, we will create two security groups: one for EC2 instances in the Elastic Beanstalk (eb) environment and the other for Amazon RDS database. We set up the inbound rules in the security group for EC2 instances for allowing SSH, HTTP and HTTPS access from source Anywhere-IPv4. We also set up the inbound rules for the Amazon RDS database for type PostgreSQL to allow PostgreSQL client access from eb environment.
This configuration now will allow only EC2 instances in the Elastic Beanstalk environment to connect the Amazon RDS database.

Database:

Next, we create a **PostgreSQL database on Amazon RDS** and attach it with the security group we created. In this step, we have to choose a subnet group and

create a database with a few specifications, for example, engine type, templates, username, password etc. We will set a database name and it should give us a hostname under the endpoint. We will need this information to pass as environment variables to connect our app to the database.

Deployment:

Now, it's time we can deploy our app. For this, we will use **Amazon Elastic Beanstalk service**.
Our django weatherApp is using SQLite3 on development. On production, we can use the PostgreSQL database that we created on Amazon RDS. We just need to configure the production settings with the environment variables: Django_SECRET_KEY, WEBSITE_HOSTNAME, DB_HOST, DB_NAME, DB_USER, DB_PASSWORD.

Since we have used a virtual environment, we will generate a requirements.txt file with 'pip freeze > requirements.txt' for Elastic Beanstalk to determine the packages to be installed. We create a virtual environment for elastic Beanstalk and install all the dependency from requirements.txt.

We will also need to create a directory called .ebextensions where a config file ebapp.config will be written that will attach the security groups for the EC2 instances and set the path of the app's WSGI object. We can now commit the file to the repository.

Next steps are:

- Deploy our site with **EB CLI (Elastic Beanstalk Command Line Interface)**. For this, we need to have awsebcli installed.
- Then immediately after we deploy, we need to edit Django's configuration file to add the domain name that Elastic Beanstalk assigned to our application in Django's ALLOWED_HOSTS. Then redeploy the application. This is **Django's security requirement** that is designed to prevent HTTP header attacks.
- **Initialize EB CLI repository**. This step will ask for some information such as: default location, app name, programming language, platform branch, Set up SSH, keypair etc. With that we now have the hostname

of our app, database hostname, database name, database user, database user's password.

- **Setup environment variables for Elastic Beanstalk environment**. We pass the above information as the environment variables to the EB environment.
- **Run database migration command on deployment.** We need to modify the .ebextensions/ebapp.config file with the migration commands.
- **Commit the changes in the repository** with git add and git commit.
- **Redeploy** with 'eb deploy' command.

Testing:

We can now test our REST API using **HTTPie** or **Postman** and play around with retrieving and filtering data.

Connecting database instance with psql:

If we need to do some database management or run SQL queries, we may need to connect the Amazon RDS instance with a database client, for example, psql. **Psql** is a terminal based frontend to PostgreSQL. Psql will enable us to type in queries interactively, communicate with PostgreSQL and see the query results. We can do this connection of the database to psql from EC2 instance with ssh command.

Scheduling data ingestion from code:

We can use **Amazon Lambda** and then create a timed trigger so that Lambda runs at a certain schedule. AWS Lambda is an event-driven, serverless computing platform/service that runs code in response to events and automatically manages the computing resources required by that code.

We will write the Lambda function which will read the data (from a given source) and push/migrate it into the database. Then we can use a trigger using **CloudWatch EventBridge**. EventBridge allows us to set periodic triggers with rate() trigger. For example, if we set the schedule expression as rate(15 minutes),

this will cause EventBridge to send a trigger to the Lambda function every 15 minutes to perform the scheduled action, ingest data in this case.

We can verify if Lambda is getting triggered periodically as planned using another service called **Amazon SES** (Simple Email Service). We can send email using Amazon SES each time the code is being executed. With the above example, an email will be sent every 15 minutes when the code is executed and data is ingested successfully.

Cleaning up:

Cleaning up is necessary to save instance hours and other AWS resources between development sessions. We can do this using the ‘eb terminate’ command. This terminates the Elastic Beanstalk environment , and doesn't delete the application. We can always create more environments with the same configurations running ‘eb create’ command.

Thank you!