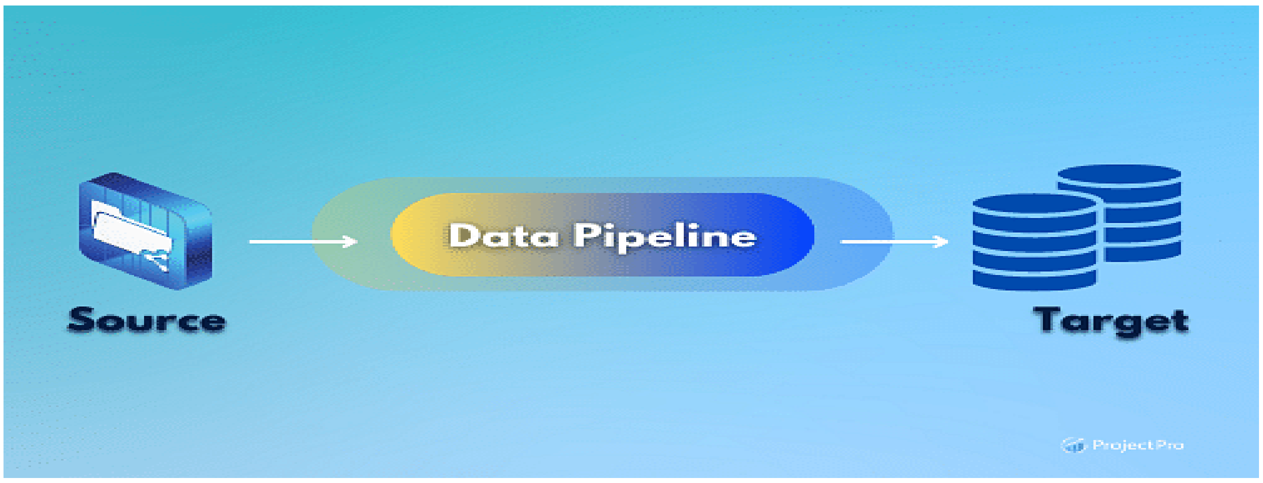
DATA ENGINEERING
BY

Oviawe Iyobosa Matthew

DATA ENGINEERING PIPELINE TASK WITH PYTHON

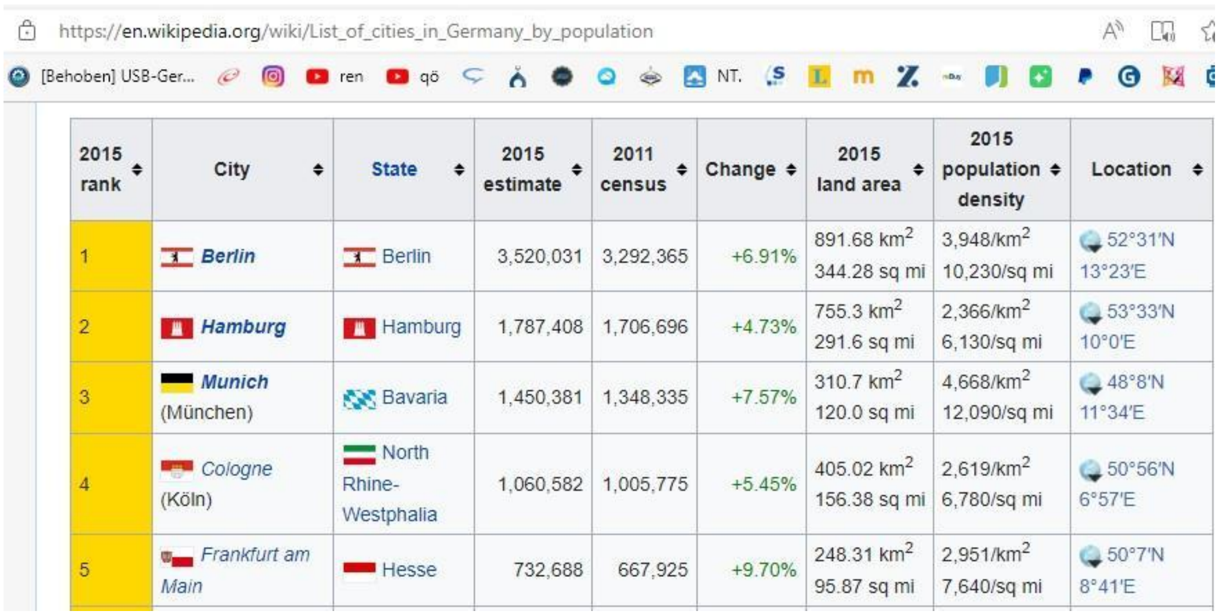


While it all started after I have been employed as a Junior Data Engineer for Gans (a fictional company set up just for learning), an e-scooter-sharing start-up company. It aspires to operate in most populous cities around the world. In each city, the company will have hundreds of e-scooters parked on the streets and allow users to rent them by the minute. Gans has seen that its operational success depends on having its scooters parked where users need them. The company wants to anticipate as many scooter movements as possible. Predictive modelling is certainly on the roadmap, but the first step is to collect more data, transform it and store it appropriately. This is where I come in:

My task will be to collect data from external sources that can potentially help Gans predict e-scooter movement. Since data is needed every day, in real-time, and accessible by everyone in the company, the challenge is going to be to assemble and automate a data pipeline in the cloud. And because Gans is based in Germany and facing already tough e-scooter competitors already, I decided to start the project in Germany.

WEB SCRAPING

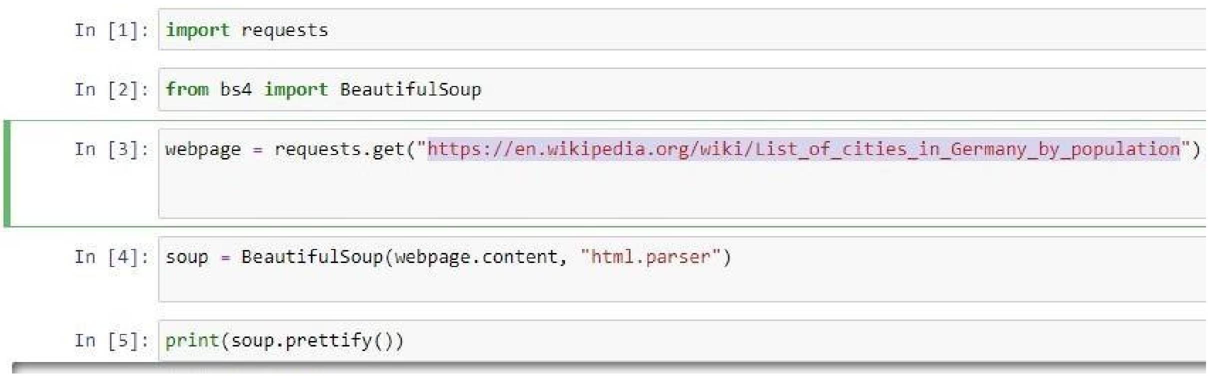
In the first phase of the project, data was scraped from the Internet and collected from APIs. Demographic data were collected for some regional strategic cities in Germany. This was done by web scraping the Wikipedia page (see Figure: 1) using a Python library called Beautiful Soup (see Figure: 2), a Python library for parsing HTML and XML documents. It creates a parse tree for parsed pages that can be used to extract data from HTML and XML. The find_all method was deployed in Python script to locate the tag in the HTML where my interested table was located, and then a code was written to extract the table of my interest.


https://en.wikipedia.org/wiki/List_of_cities_in_Germany_by_population

2015 rank	City	State	2015 estimate	2011 census	Change	2015 land area	2015 population density	Location
1	Berlin	Berlin	3,520,031	3,292,365	+6.91%	891.68 km^2 344.28 sq mi	3,948/km^2 10,230/sq mi	52°31′N 13°23′E
2	Hamburg	Hamburg	1,787,408	1,706,696	+4.73%	755.3 km^2 291.6 sq mi	2,366/km^2 6,130/sq mi	53°33′N 10°0′E
3	Munich (München)	Bavaria	1,450,381	1,348,335	+7.57%	310.7 km^2 120.0 sq mi	4,668/km^2 12,090/sq mi	48°8′N 11°34′E
4	Cologne (Köln)	North Rhine-Westphalia	1,060,582	1,005,775	+5.45%	405.02 km^2 156.38 sq mi	2,619/km^2 6,780/sq mi	50°56′N 6°57′E
5	Frankfurt am Main	Hesse	732,688	667,925	+9.70%	248.31 km^2 95.87 sq mi	2,951/km^2 7,640/sq mi	50°7′N 8°41′E

Figure 1: Some demographic data in Germany

The extracted data were saved in a data frame. The data were later processed for storage.


```
In [1]: import requests

In [2]: from bs4 import BeautifulSoup

In [3]: webpage = requests.get("https://en.wikipedia.org/wiki/List_of_cities_in_Germany_by_population")

In [4]: soup = BeautifulSoup(webpage.content, "html.parser")

In [5]: print(soup.prettify())
```

Figure 2: Beautiful Soup

API DATA COLLECTION

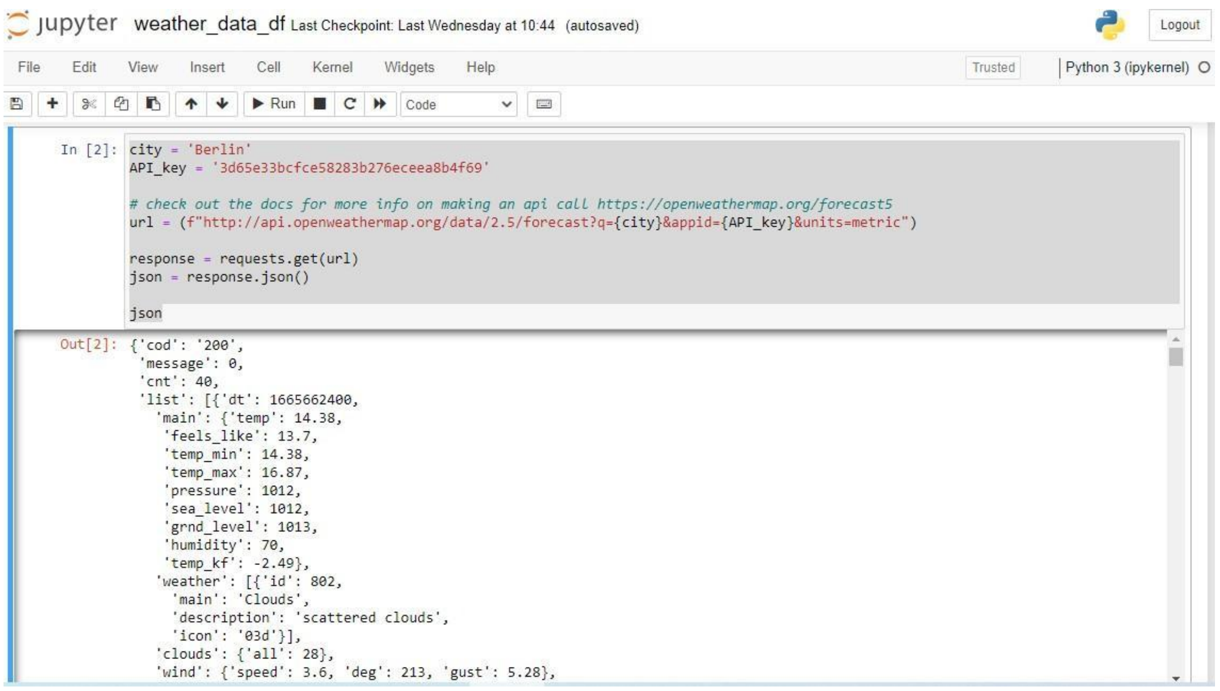
Data about Airport locations and flights into and out of the Airports and the Weather of the cities of Gans interest were collected from APIs. And since Gan is interested in building a predictive model of their e-Scooter movements, these data sets should be collected and automated to reflect today's weather conditions and airport flight activities and predict tomorrow's weather conditions and flight activities in and around our interested cities. This is seen in the code below:

today = datetime.date.today()

tomorrow = today + datetime.timedelta(days=1)

WEATHER API — OpenWeatherMap

Weather data was collected using OpenWeatherMap web API. OpenWeatherMap provides global weather data via API, including current weather data, forecasts, nowcasts, and historical weather data for any geographical location. An API key was given to me after 8 hours of signing up for an account with them which I used to exploit their data. I made an API call for their 4 days Hourly Forecast by City and Country name. The request code is seen in Figure 3 below.



```
In [2]: city = 'Berlin'
        API_key = '3d65e33bcfce58283b276eceea8b4f69'

        # check out the docs for more info on making an api call https://openweathermap.org/forecast5
        url = (f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid={API_key}&units=metric")

        response = requests.get(url)
        json = response.json()

        json
```

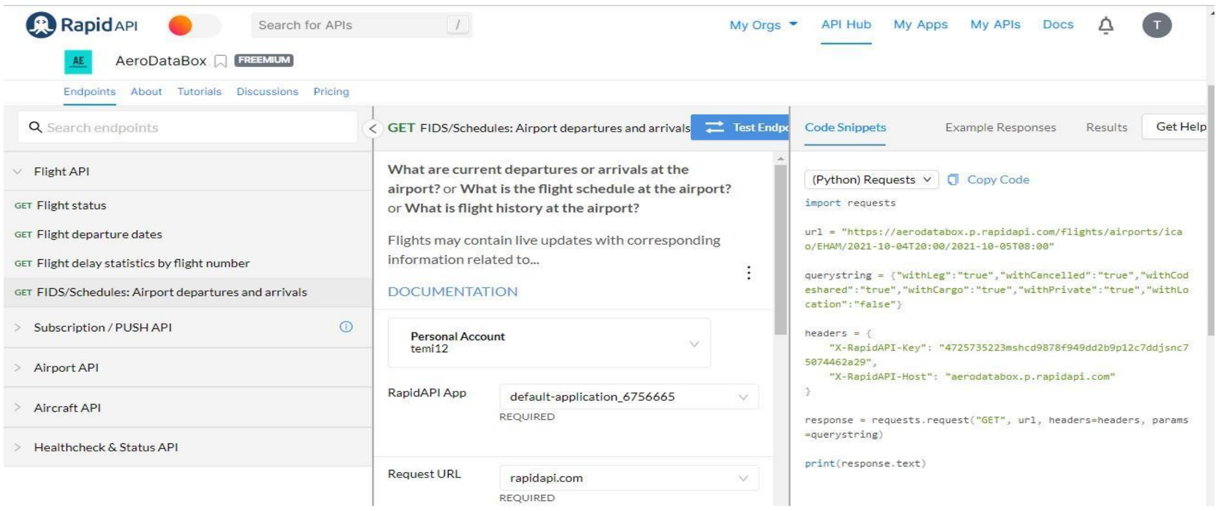
```
Out[2]: {'cod': '200',
         'message': 0,
         'cnt': 40,
         'list': [{'dt': 1665662400,
           'main': {'temp': 14.38,
            'feels_like': 13.7,
            'temp_min': 14.38,
            'temp_max': 16.87,
            'pressure': 1012,
            'sea_level': 1012,
            'grnd_level': 1013,
            'humidity': 70,
            'temp_kf': -2.49},
           'weather': [{'id': 802,
             'main': 'Clouds',
             'description': 'scattered clouds',
             'icon': '03d'}],
           'clouds': {'all': 28},
           'wind': {'speed': 3.6, 'deg': 213, 'gust': 5.28},
```

Figure 3: The request code used to collect Weather API data

The response I got from the request was saved in JSON format, this can, however, be changed to HTML or XML format by adding a mode parameter. The response data in the JSON file was iterated, and only the interested data were selected. The selected data after iteration were stored in a dictionary and converted to a data frame for processing and storage.

FLIGHT API - AeroDataBox

Airport locations and Flight arrivals and departures to these Airport locations data were collected using AeroDataBox API. Access was given to this API data after I signed up for an account and subscribed for AeroDataBox API basic plan. I searched for Airport departures and arrivals data in the search section. See below, the search criteria created. I also searched for Airport locations in the Endpoints (see Figure: 4) and the codes from these searches were copied and pasted into the Jupyter Notebook where a request was made to the AeroDataBox API URL and the corresponding responses were stored in JSON files. Python codes were used to iterate through the JSON files and the interested data sets were extracted and saved in data frames.



What are current departures or arrivals at the airport? or What is the flight schedule at the airport? or What is flight history at the airport?

Flights may contain live updates with corresponding information related to...

Personal Account
temi12

RapidAPI App: default-application_6756665 REQUIRED

Request URL: rapidapi.com REQUIRED

```
import requests

url = "https://aerodatabox.p.rapidapi.com/flights/airports/ica
o/EHAM/2021-10-04T20:00/2021-10-05T08:00"

querystring = {"withLeg":"true","withCancelled":"true","withCod
eshared":"true","withCargo":"true","withPrivate":"true","withLo
cation":"false"}

headers = {
    "X-RapidAPI-Key": "4725735223mshcd9878f949dd2b9p12c7ddjsnc7
5074462a29",
    "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"
}

response = requests.request("GET", url, headers=headers, params
=querystring)

print(response.text)
```

Figure 4: AeroDataBox API Endpoints

DATA STORING

The second phase of this project was to store the extracted data sets from web scraping and APIs. These data were stored in a MySQL database, and I used MySQL Workbench (a graphical tool for working with MySQL servers and databases) to create and virtualize the database (see Figure: 5). The database, and its tables were created by running a written script on MySQL Workbench and connected using primary and foreign keys. The script for creating the database can be seen on my Github.

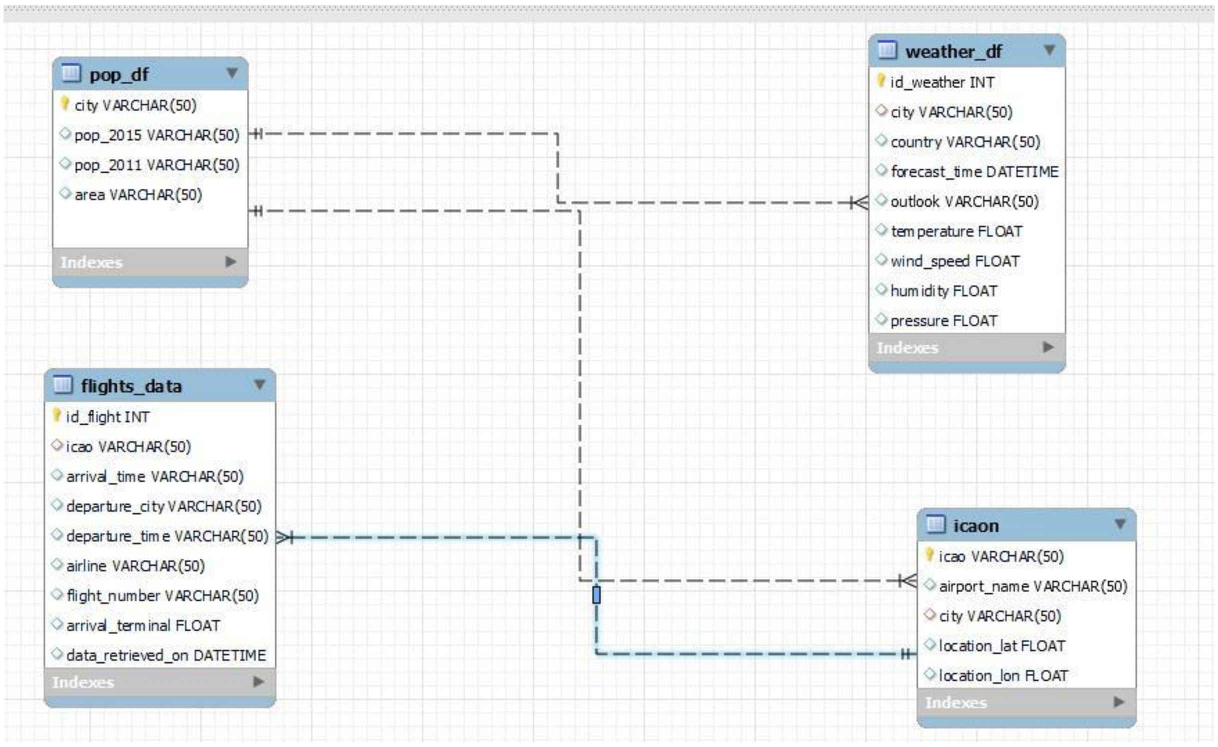
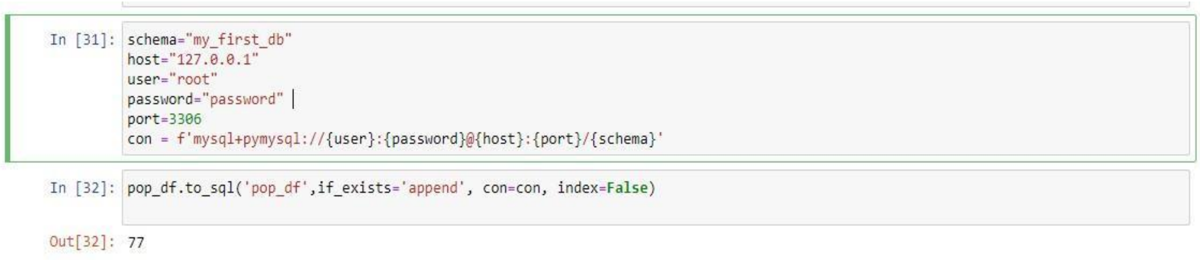


Figure 5: The EER Diagram for my project database containing 4 tables connected through primary and foreign keys

It was time to transfer the data from data frames in Jupyter Notebook to my newly created database in MySQL. In achieving this, I used a Python library called SQLAlchemy (a Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL). In the connection code (see Figure 6), schema is the name of the database I am connecting to, the host is the location of the database server. The user and password are the login details to the database server, The port refers to the optional database port. The mysql+pymysql refers to the type of database and the DBAPI I am using. Con variable created and passed to_sql() function, the data frame is sent directly to my created database in MySQL.

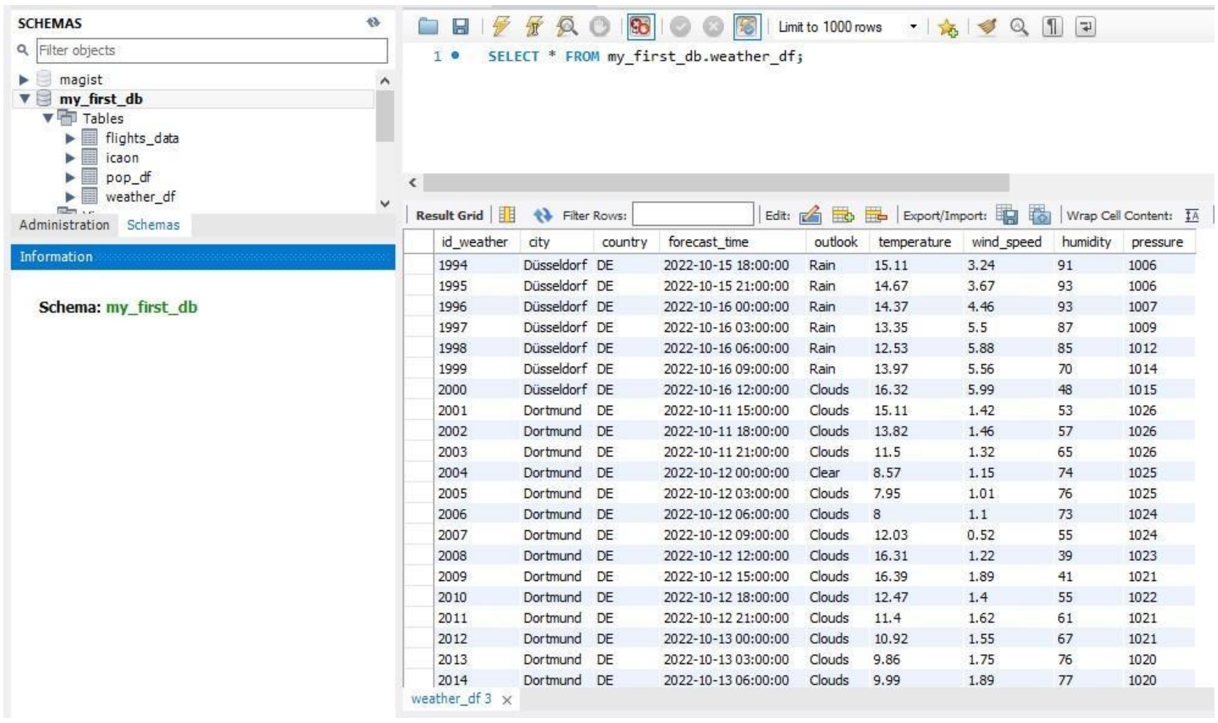

```
In [31]: schema="my_first_db"
         host="127.0.0.1"
         user="root"
         password="password"
         port=3306
         con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'

In [32]: pop_df.to_sql('pop_df',if_exists='append', con=con, index=False)

Out[32]: 77
```

Fig 6: The code used in connecting pop_df to my created database in MySQL

My flight data, flight location data (icaon), weather data, and population data were all sent to my database in MySQL using this connection code, stored there, and ready to be query at any time, see Figure:7. The database was at this time running on my PC’s local server and each time I want to update the dynamic data (flights and flight locations data) I will have to run the python scripts again manually in Jubital Notebook, but this can be eliminated by taking my created database to AWS Crowd.



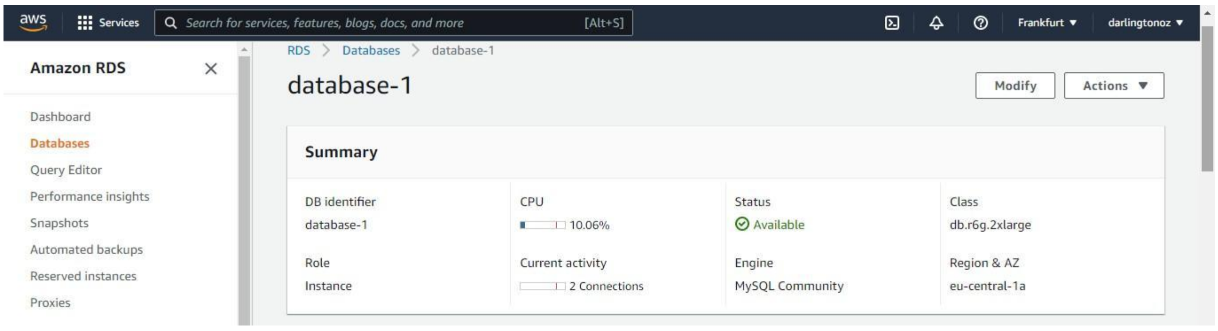
```
SELECT * FROM my_first_db.weather_df;
```

id_weather	city	country	forecast_time	outlook	temperature	wind_speed	humidity	pressure
1994	Düsseldorf	DE	2022-10-15 18:00:00	Rain	15.11	3.24	91	1006
1995	Düsseldorf	DE	2022-10-15 21:00:00	Rain	14.67	3.67	93	1006
1996	Düsseldorf	DE	2022-10-16 00:00:00	Rain	14.37	4.46	93	1007
1997	Düsseldorf	DE	2022-10-16 03:00:00	Rain	13.35	5.5	87	1009
1998	Düsseldorf	DE	2022-10-16 06:00:00	Rain	12.53	5.88	85	1012
1999	Düsseldorf	DE	2022-10-16 09:00:00	Rain	13.97	5.56	70	1014
2000	Düsseldorf	DE	2022-10-16 12:00:00	Clouds	16.32	5.99	48	1015
2001	Dortmund	DE	2022-10-11 15:00:00	Clouds	15.11	1.42	53	1026
2002	Dortmund	DE	2022-10-11 18:00:00	Clouds	13.82	1.46	57	1026
2003	Dortmund	DE	2022-10-11 21:00:00	Clouds	11.5	1.32	65	1026
2004	Dortmund	DE	2022-10-12 00:00:00	Clear	8.57	1.15	74	1025
2005	Dortmund	DE	2022-10-12 03:00:00	Clouds	7.95	1.01	76	1025
2006	Dortmund	DE	2022-10-12 06:00:00	Clouds	8	1.1	73	1024
2007	Dortmund	DE	2022-10-12 09:00:00	Clouds	12.03	0.52	55	1024
2008	Dortmund	DE	2022-10-12 12:00:00	Clouds	16.31	1.22	39	1023
2009	Dortmund	DE	2022-10-12 15:00:00	Clouds	16.39	1.89	41	1021
2010	Dortmund	DE	2022-10-12 18:00:00	Clouds	12.47	1.4	55	1022
2011	Dortmund	DE	2022-10-12 21:00:00	Clouds	11.4	1.62	61	1021
2012	Dortmund	DE	2022-10-13 00:00:00	Clouds	10.92	1.55	67	1021
2013	Dortmund	DE	2022-10-13 03:00:00	Clouds	9.86	1.75	76	1020
2014	Dortmund	DE	2022-10-13 06:00:00	Clouds	9.99	1.89	77	1020

Fig 7: Output of a query from my database in MySQL local server

TRANSFERRING THE PIPELINE TO AWS CLOUD

The last phase of the project was to automate my database in the cloud using Amazon Web Services (AWS) as a service provider. The first step was to open an AWS basic account. After that I created a cloud database using RDS Instance (see Figure 8). I also created a new database in MySQL Workbench and connected it to my created cloud RDS instance with a password and the endpoint of the RDS Instance.



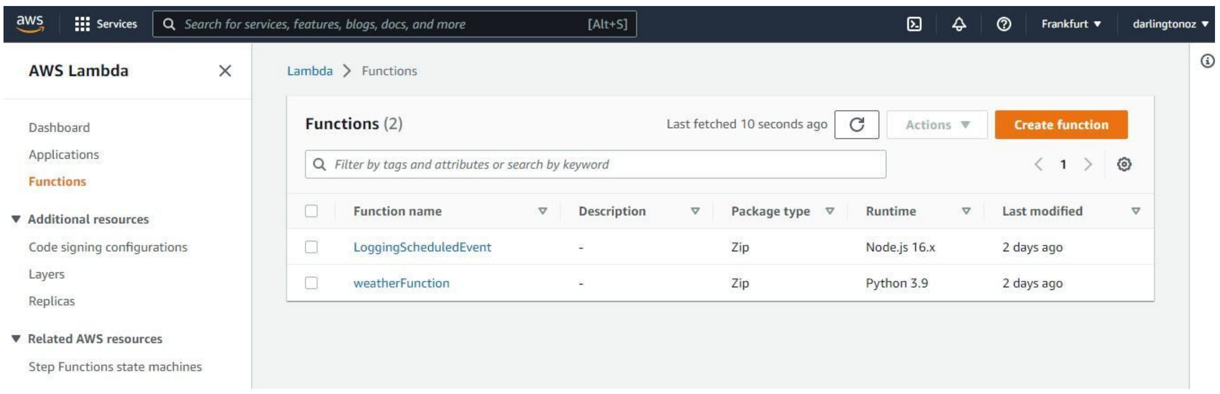
database-1

Summary

DB identifier	CPU	Status	Class
database-1	10.06%	Available	db.r6g.2xlarge
Role	Current activity	Engine	Region & AZ
Instance	2 Connections	MySQL Community	eu-central-1a

Fig 8: My created database in AWS

In the second step, I used the AWS service called Lambda function to transfer my data frames to a cloud database, See Figure: 9. AWS Lambda is a serverless compute service that runs your code in response to events and automatically manages the underlying compute resources for you. These events may include state changes or an update. My Python code used earlier in transferring my data frames to my local server database in MySQL was used again but this time written as a function in a lambda_handler() within the function and whenever the function is run the code within lambda_handler() is triggered. I added a layer to the AWS Lambda for my code to get access to Python libraries that I used in my function code. I then created a new Role in the AWS Identity and Access Management (IAM). This will help me secure who gets access and permission to my created Lambda function.

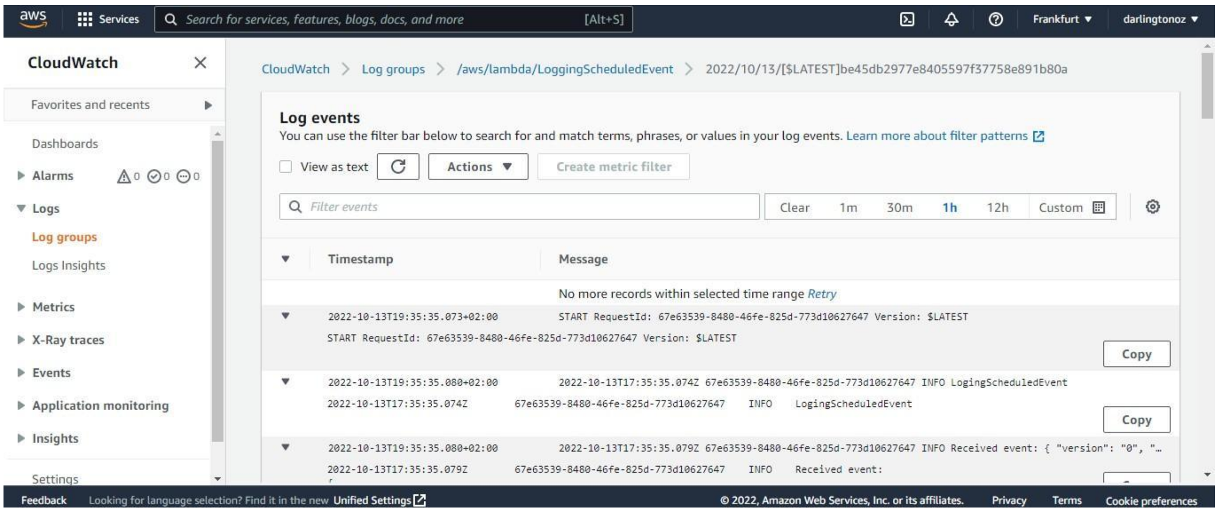

aws
Services
Search for services, features, blogs, docs, and more
[Alt+S]
Frankfurt
darlingtonoz
AWS Lambda
Dashboard
Applications
Functions
Additional resources
Code signing configurations
Layers
Replicas
Related AWS resources
Step Functions state machines
Lambda > Functions
Last fetched 10 seconds ago
Actions
Create function
Filter by tags and attributes or search by keyword

Functions (2)

Function name	Description	Package type	Runtime	Last modified
LoggingScheduledEvent	-	Zip	Node.js 16.x	2 days ago
weatherFunction	-	Zip	Python 3.9	2 days ago

Fig 9: Lambda Function (weatherfunction)

The last step I took in this phase was to Automate my database in the cloud so it gets updated to my pre-set time without me manually running the function code. To achieve this, I used an AWS service called AWS EventBridge (a serverless event bus service that you can use to connect your applications with data from a variety of sources). It enables real-time access to changes in data in AWS services. All that is needed to be done is to set the timing of the EventBridge and at the set time the EventHandler triggers my Lambda function in response, the function code will run, and the database will be updated. In this Project, I set the EventBridge to trigger every day at 6 am. From the Amazon CloudWatch Logs, I was able to monitor, observe and access log files from the automated data pipeline, see Figure:10.


CloudWatch
Favorites and recents
Dashboards
Alarms
Logs
Log groups
Logs Insights
Metrics
X-Ray traces
Events
Application monitoring
Insights
Settings
CloudWatch > Log groups > /aws/lambda/LoggingScheduledEvent > 2022/10/13/[$LATEST]be45db2977e8405597f37758e891b80a
Log events
You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns
View as text
Actions
Create metric filter
Filter events
Clear
1m
30m
1h
12h
Custom

```
Timestamp                       Message
                                No more records within selected time range Retry
2022-10-13T19:35:35.073+02:00   START RequestId: 67e63539-8480-46fe-825d-773d10627647 Version: $LATEST
START RequestId: 67e63539-8480-46fe-825d-773d10627647 Version: $LATEST
2022-10-13T19:35:35.080+02:00   2022-10-13T17:35:35.074Z 67e63539-8480-46fe-825d-773d10627647 INFO LogingScheduledEvent
2022-10-13T17:35:35.074Z     67e63539-8480-46fe-825d-773d10627647    INFO    LogingScheduledEvent
2022-10-13T19:35:35.080+02:00   2022-10-13T17:35:35.079Z 67e63539-8480-46fe-825d-773d10627647 INFO Received event: { "version": "0", "…
2022-10-13T17:35:35.079Z     67e63539-8480-46fe-825d-773d10627647    INFO    Received event:
```

Fig 10: Log events of the automated data pipeline

PROJECT SUMMARY

I have been able to gather needed data for Gans to kick start their data predictive modelling Journey in Germany. These data sets are saved in a database in MySQL, and in AWS Cloud. I have been able to automate this process in AWS Cloud. The function code I set up in AWS Lambda is running in the Cloud every day at 6 am collecting data from the Internet, and saving them to my database. The data in the database will continue to grow bigger, and the Data Scientist will have to start analysing the data for Gans to get some valuable insights about the data. Mission accomplished! It was an insightful journey for me, I was able to learn more than just basic Data Engineering skills. I look forward to the next task from Gans.

Here is a quick recap on how I got it done:

- Wikipedia sites got scraped for information about the population and cities in Germany
- APIs sites used in collecting flight and weather data
- database created and set up in the cloud
- code transformed to function and put in the cloud
- Function code schedule be run whenever needed