




Automation using AWS

PIPELINES TO THE CLOUDS

WBS Coding School

Balu govind



PIPELINES TO THE CLOUD

Ever since the dawn of time, mankind was in constant search for efficiency and ease, leading to the invention and development of means of automation, mechanical and digital. Today there is no need to reinvent the wheel. The opportunity to automate is virtually endless. In this project we are looking at one of these methods of automation.

The simple task of getting from A to B is radically changing as more and more modes of transportation are being developed. The recent and one of the big hits in the metropolitans of the world is the E-Scooter. This project attempts to calculate and predict strategic placement of E-Scooters using the power of AWS cloud Lambda Functions.

WHAT ARE WE LOOKING FOR?

As a Data Engineer the key area that needed focus was the Data itself. "What to look for?" was question that needed to be answered. So a list of data types were made.

- Demographic
- Weather
- Airports
- Flights

OBJECTIVES OF THE PROJECT

- Data Collection
- Database creation
- Pipeline to the Cloud
- Automate Pipeline

METHODOLOGY-DEMOGRAPHICS DATA COLLECTION

To keep our attempts simplified we had restricted ourselves to German cities, giving particular interest to ones with airports. From these cities the City population was one of the key aspects we looked for. Along with it, the city name, state and geographical location were also collected. These details were collected by manual webscraping of wikipedia website pages and then compiled into a Data Frame.

The following code depicts the city of Frankfurt am Main but can be further extrapolated by adding list of cities if needed in the future.

```
import requests
import pandas as pd
import json
from bs4 import BeautifulSoup
import re
import time
!pip install python-dotenv
%load_ext dotenv
%dotenv
import os
import pytz
from datetime import datetime
import mysql.connector
import sqlalchemy
```

```
cityd=['Frankfurt am Main']
API_key = api_key
def demo(cities):
    cities_id = []
    dfList = []
    for city in cities:
        time.sleep(2)
        url1 =
'https://en.wikipedia.org/wiki/{}'.format(city)
        cite = requests.get(url1, 'html.parser')

        if BeautifulSoup(cite.content) != None:
            soup = BeautifulSoup(cite.content)
            if soup.find('li', {'id':'t-
wikibase'}).find('a')['href'] != None:
                wikidata_link = soup.find('li', {'id':'t-
wikibase'}).find('a')['href']
            city_id = re.search('Q\d+', wikidata_link).group()
            cities_id.append(city_id)

        url2 = "https://wft-geo-
db.p.rapidapi.com/v1/geo/cities/{}".format(city_id)
        headers = {
            "X-RapidAPI-Key": API_key,
            "X-RapidAPI-Host": "wft-geo-db.p.rapidapi.com"
        }
        response = requests.request("GET", url2,
headers=headers)
        print(response.json())
```

```

cit_dic = {}
    cit_dic['City'] = response.json()['data']['name']
    cit_dic['Country'] =
response.json()['data']['country']
    cit_dic['CountryCode'] =
response.json()['data']['countryCode']
    cit_dic['WikiDataId'] =
response.json()['data']['wikiDataId']
    cit_dic['Latitude'] =
round(response.json()['data']['latitude'], 4)
    cit_dic['Longitude'] =
round(response.json()['data']['longitude'], 4)
    cit_dic['Population'] =
response.json()['data']['population']
    cit_dic['Timezone'] =
response.json()['data']['timezone']

    dfList.append(cit_dic)
    df_demo = pd.DataFrame(dfList)
    return df_demo
demo(citd) #run the whole thing – good luck!

```

WEATHER DATA COLLECTION

The weather data was collected by utilizing the open weather APIs and a 5-day forecast with 3 hour intervals was collected. Again, this code can work for any European city that can be added to the list.

```

citd = ['Frankfurt am Main']
def weather(cities):
    API_key = api_key

    tz = pytz.timezone('Europe/Berlin')
    now = datetime.now().astimezone(tz) #for the timestamp
    time of sampling

    a = {'City': [], 'Country': [], 'ForecastTime': [],
'Outlook': [], 'Temperature': [], 'Clouds': [], 'Rain': [],
'Snow': [], 'WindSpeed': [], 'DateInformation': []} #usefull
form of a dic, gives the right form when change to df, see
cell below

    for city in cities: #runs a loop over the cities
        url =
(f"http://api.openweathermap.org/data/2.5/forecast?q={city}&a
ppid={API_key}&units=metric")
        response = requests.get(url)
        json = response.json()

```

Here a dictionary is defined to collect the required keys from the JSON response which will form the columns of the data frame. To properly view and navigate the JSON response, use the following code

```
from IPython.display import JSON
import json
JSON(response.json())
```

The following is a part of the weather data frame that was created.

	City	Country	ForecastTime	Outlook	Temperature	Clouds	Rain	Snow	WindSpeed	DateInformation
0	Frankfurt am Main	DE	2022-06-15 12:00:00	clear sky	26.08	8	0	0	0.91	15/06/2022 11:02:40
1	Frankfurt am Main	DE	2022-06-15 15:00:00	clear sky	28.16	7	0	0	0.77	15/06/2022 11:02:40
2	Frankfurt am Main	DE	2022-06-15 18:00:00	clear sky	26.88	6	0	0	2.16	15/06/2022 11:02:40
3	Frankfurt am Main	DE	2022-06-15 21:00:00	clear sky	21.00	5	0	0	1.53	15/06/2022 11:02:40
4	Frankfurt am Main	DE	2022-06-16 00:00:00	few clouds	17.99	23	0	0	2.19	15/06/2022 11:02:40
5	Frankfurt am Main	DE	2022-06-16 03:00:00	scattered clouds	16.32	37	0	0	2.25	15/06/2022 11:02:40
6	Frankfurt am Main	DE	2022-06-16 06:00:00	scattered clouds	19.78	47	0	0	3.07	15/06/2022 11:02:40
7	Frankfurt am Main	DE	2022-06-16 09:00:00	clear sky	24.81	4	0	0	3.17	15/06/2022 11:02:40
8	Frankfurt am Main	DE	2022-06-16 12:00:00	clear sky	27.92	7	0	0	3.12	15/06/2022 11:02:40
9	Frankfurt am Main	DE	2022-06-16 15:00:00	clear sky	28.27	10	0	0	4.18	15/06/2022 11:02:40
10	Frankfurt am Main	DE	2022-06-16 18:00:00	few clouds	24.45	11	0	0	3.85	15/06/2022 11:02:40

AIRPORTS DATA COLLECTION

The next order of business was to find all the major airports in Germany. Our research showed that there are 15 major civilian airports in Germany. Since manual web scraping of the Wikipedia and official airport website pages turned out to be a Herculean task, we had to rely on API key providers such as Rapid API-AeroDatabox. Although this provider has easy to use codes for API calls, the major downside was the hard limit of the number of calls available per account. So extra care was taken not to exhaust the quota.

Here instead of giving the city name, the ICAO codes for the airports should be given to the list.

```
ac = ['EDDF']
def airp(cities):
    airport = []
    API_KEY = api_key
    a = {'ICAO': [], 'IATA': [], 'Name': [], 'City': [],
        'Country': [], 'Continent': [], 'Time Zone': [], 'Website':
        []}
    for i in ac:
        url =
        f"https://aerodatabox.p.rapidapi.com/airports/icao/{i}"

        querystring = {"withRunways": "true"}

        headers = {
            "X-RapidAPI-Key": API_KEY,
            "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"
        }
```

```

response = requests.request("GET", url, headers=headers,
params=querystring)
air_json = response.json()
a['ICAO'] = air_json['icao']
a['IATA'] = air_json['iata']
a['Name'] = air_json['shortName']
a['City'] = air_json['municipalityName']
a['Country'] = air_json['country']['name']
a['Continent'] = air_json['continent']['name']
a['Time Zone'] = air_json['timeZone']
try:
    a['Website'] = air_json['urls']['webSite']
except:
    a['Website'] = 'Not Available'

air = pd.DataFrame(a, index=[0])

```

FLIGHT DATA COLLECTION

One of the most important details to be collected was the flight details so as to have a sense of when the scooters are going to be most used, since the company customer service reports majority of the users utilize our commodities to commute to and from the airports. Again, we had to rely on Rapid API services. With these keys we can analyse the flight schedules for the coming 12 hours and deduce the rush hour by calculating the number of flights arriving per hour.

```

cit = ['EDDF']

def flights(cities):
    flight_details = []
    a = {'City': [], 'FlightNumber': [], 'FlightStatus': [],
'OriginAirport': [], 'ScheduledTime': [], 'AirLine': []}
    url =
f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{c
ity}/2022-06-15T06:00/2022-06-15T17:00"
    querystring =
{"direction":"Arrival","withCodeshared":"true","withCargo":"f
alse","withPrivate":"false","withLocation":"true"}
    headers = {
        "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com",
        "X-RapidAPI-Key": api_key
    }

```

```

responses = requests.request("GET", url, headers=headers,
params=querystring)
    json = responses.json()
    for i in json['arrivals']:
        a['City'] = city
        a['FlightNumber'].append(i['number'])
        a['FlightStatus'].append(i['status'])

a['OriginAirport'].append(i['movement']['airport']['name'])

a['ScheduledTime'].append(i['movement']['scheduledTimeLocal'
])
        a['AirLine'].append(i['airline']['name'])
    c = pd.DataFrame(a)
    return c
flights(cit)

```

The data frame for the flights looks like below

	City	FlightNumber	FlightStatus	OriginAirport	ScheduledTime	AirLine
0	EDDF	AC 9430	Arrived	Casablanca	2022-06-15 05:55+02:00	Air Canada
1	EDDF	UA 8833	Arrived	District of Columbia	2022-06-15 05:45+02:00	United
2	EDDF	A3 1837	Expected	Athens	2022-06-15 06:45+02:00	Aegean
3	EDDF	AA 8554	Expected	Madrid	2022-06-15 06:30+02:00	American
4	EDDF	AC 6128	Arrived	Bangkok	2022-06-15 06:15+02:00	Air Canada

DATABASE CREATION

After the task of data collection, the new job at hand was to create a database locally and save these data frames as tables in the same. The platform that is used here is MySQL.

The procedure was to first create a Schema using MySQL workbench and then export all the data frames to the said Schema by using the power of Python. For this, we utilize the PyMySQL and SQLAlchemy client libraries. If these modules are not installed, a pip installation would suffice.

To connect Python with MySQL, use the following code

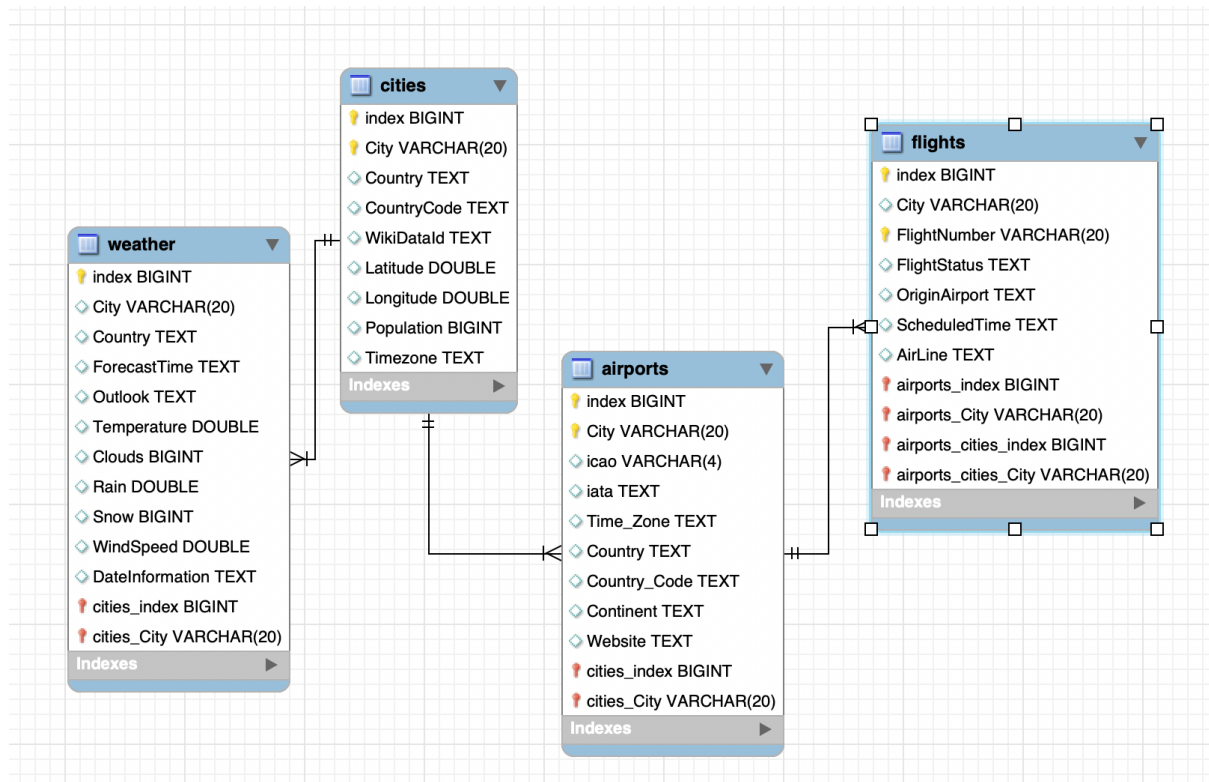
```
cnx = mysql.connector.connect(
    user='root',
    password='password', #type your root password here
    host='local', # to connect to your local instance
    database='gans' #type the name of the database you want
to use here
)

# connect to database
cursor = cnx.cursor()
schema="gans"
host="local"
user="root"
password="password"
port=3306
con =
f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'
```

After connecting the two platforms, the next step is to export all the data frames to the schema “gans”.

```
weather(citd).to_sql('weather',
                    if_exists='append',
                    con=con,
                    index=False)
airp(ac).to_sql('airports',
               if_exists='append',
               con=con,
               index=False)
demo(citd).to_sql('cities',
                 if_exists='append',
                 con=con,
                 index=False)
flights(cit).to_sql('flights',
                   if_exists='append',
                   con=con,
                   index=False)
```


Now it is time to migrate to the MySQL workbench to further manipulate the created tables. Here the primary and the foreign keys were assigned, and relations were defined.



From MySQL platform these tables could be easily viewed and analyzed with the power of the Structured Query Language.

PIPELINES TO THE CLOUD

With the new age of the cloud, we can utilize the online services for serverless computing. Amazon Web Services (AWS) provides free platforms for small projects and this was perfectly suited for our needs as the scope of the project did not demand large storage or high performance computing.

AWS offers a service called RDS where it allows us our locally made database to be exported to the cloud. The documentation of the said process is available in the console and steps were followed as instructed to make our database online.

Next task was to upload the tables and other data to the database created with RDS and this could be done by utilizing the Lambda functions provided by AWS. For this, an administrator role was created and then the Python code scripts in the Jupiter Lab notebook were transferred to the Lambda function.

AUTOMATE THE PIPELINE

Since the only tables that were subject to change were the weather and flight, automation was only needed for the same.

Within the Lambda function triggers were added with AWS Event Bridge. This asks for Cron expressions which could be easily defined specifying the frequency of these triggers. For the purpose of this project, triggers were designed to fire every 24 hours to gather the weather and flight data.

Define schedule [Info](#)

Schedule pattern

Schedule pattern

Choose the schedule type that best meets your needs.

☒ A fine-grained schedule that runs at a specific time, such as 8:00 a.m. PST on the first Monday of every month.

☐ A schedule that runs at a regular rate, such as every 10 minutes.

Cron expression [Info](#)

Define the cron expression for the schedule

 cron ()

Minutes Hours Day of month Month Day of week Year

Next 10 trigger date(s)

UTC ▼

Fri, 17 Jun 2022 23:59:00 UTC
Sat, 18 Jun 2022 23:59:00 UTC
Sun, 19 Jun 2022 23:59:00 UTC
Mon, 20 Jun 2022 23:59:00 UTC
Tue, 21 Jun 2022 23:59:00 UTC
Wed, 22 Jun 2022 23:59:00 UTC
Thu, 23 Jun 2022 23:59:00 UTC
Fri, 24 Jun 2022 23:59:00 UTC
Sat, 25 Jun 2022 23:59:00 UTC
Sun, 26 Jun 2022 23:59:00 UTC

CONCLUSION

The scope of this project was to learn the techniques that could be used to extract, manipulate and automate data bases using cloud computing services. A major skill set needed in the current job market is the familiarity of AWS console and the participants in this project was introduced to a small window of the platform.

There is scope of expansion as more cities could be added and the details analysed. This project can further expand and investigate the actual flight traffic and weather pattern so as to model a predictive analysis for the strategic placement of the E-Scooters.