My Data Pipelines Project
-Guoying Dai

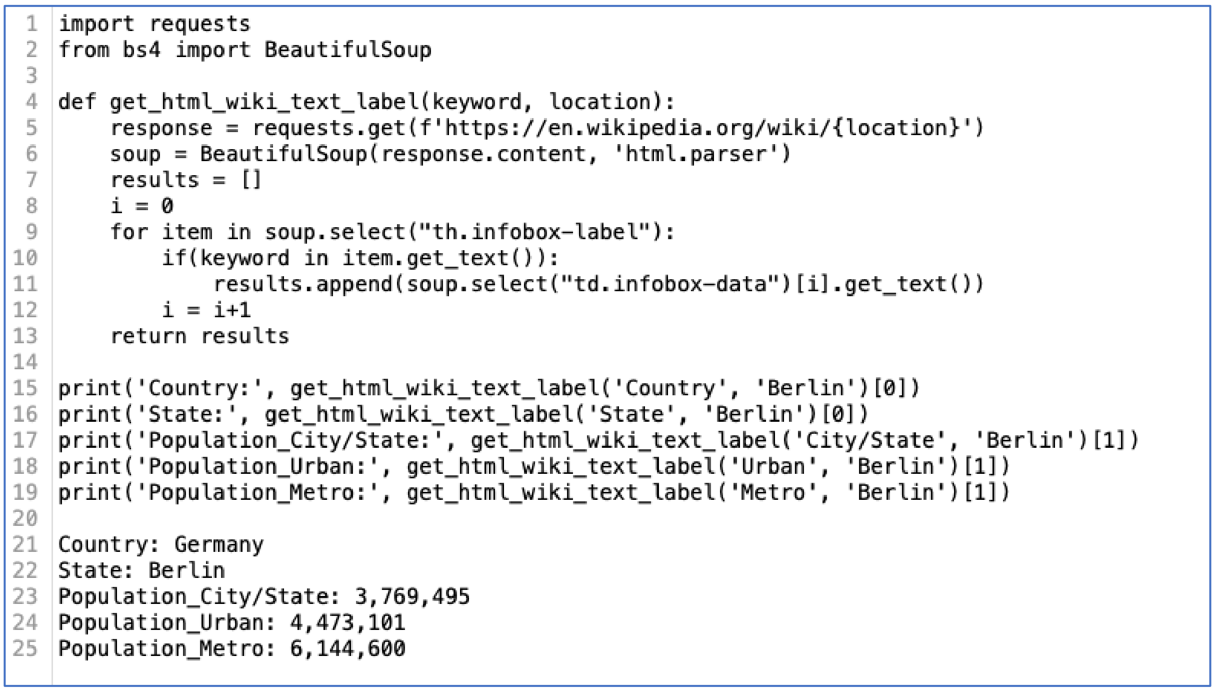
This project is to help a e-scooter company Gans to improve the e-scooter-sharing system. As the populations and weather of a city, as well as the time when people arriving the airport are very important elements that influenced the sharing system. We should collect data from these sources that potentially help Gans e-scooter movement. Since data is needed every day, in real-time and accessible by everyone in the company, the challenge is to assemble and automate a data pipeline in the cloud. There are two major phases of the project: create local pipeline and then create the cloud pipeline.

1. Create Local Pipeline

In this phase, we should gather data from website by execute the scripts in Jupyter notebook, then store them to the local database-MySQL. There are several ways to dedicate data from website.

1) web scraping:

we use the python library :**BeautifulSoup** to access the information we needed by downloading and extracting the HTML code of the website (Fig1).



```
import requests
from bs4 import BeautifulSoup

def get_html_wiki_text_label(keyword, location):
    response = requests.get(f'https://en.wikipedia.org/wiki/{location}')
    soup = BeautifulSoup(response.content, 'html.parser')
    results = []
    i = 0
    for item in soup.select("th.infobox-label"):
        if(keyword in item.get_text()):
            results.append(soup.select("td.infobox-data")[i].get_text())
        i = i+1
    return results

print('Country:', get_html_wiki_text_label('Country', 'Berlin')[0])
print('State:', get_html_wiki_text_label('State', 'Berlin')[0])
print('Population_City/State:', get_html_wiki_text_label('City/State', 'Berlin')[1])
print('Population_Urban:', get_html_wiki_text_label('Urban', 'Berlin')[1])
print('Population_Metro:', get_html_wiki_text_label('Metro', 'Berlin')[1])

Country: Germany
State: Berlin
Population_City/State: 3,769,495
Population_Urban: 4,473,101
Population_Metro: 6,144,600
```

Fig. 1 The code for getting the city's (Berlin) information: populations and state and country

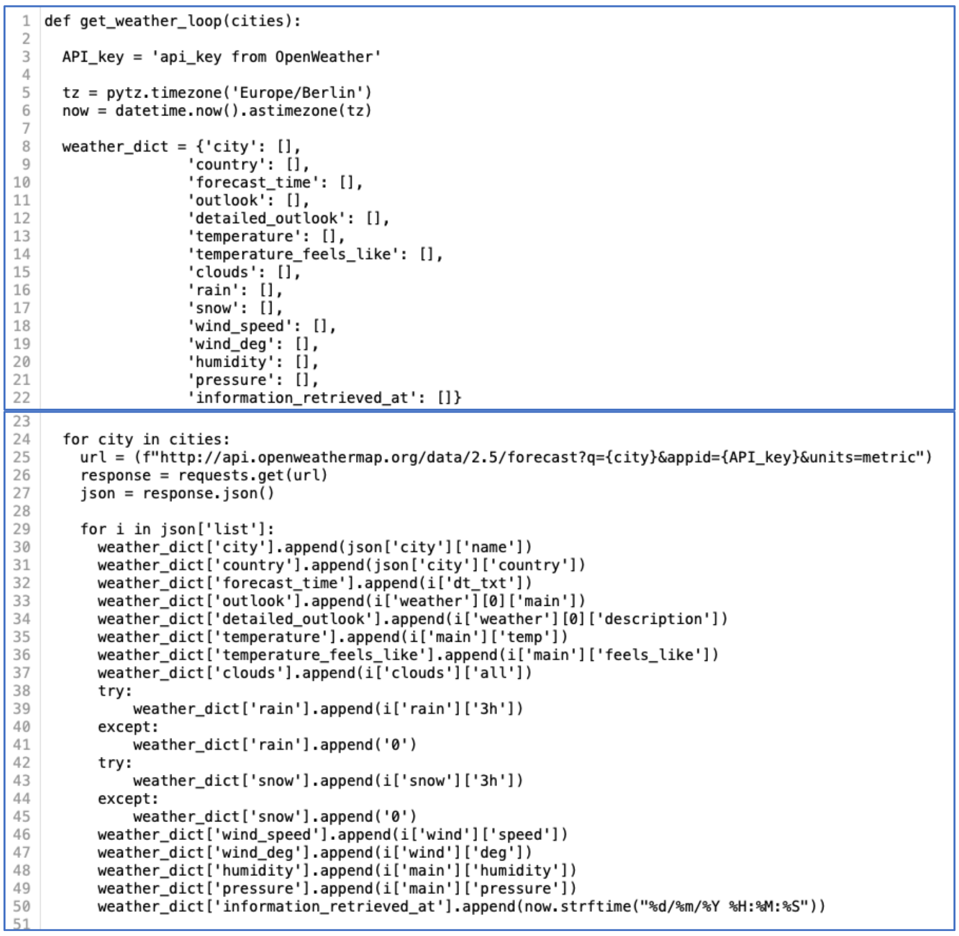
2) with APIs or particular API

Application programming interface (API) is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software.(from Wikipedia) .

In this project, we use Python's ***request library*** to interact with APIs and collected weather and flights data by calling APIs.

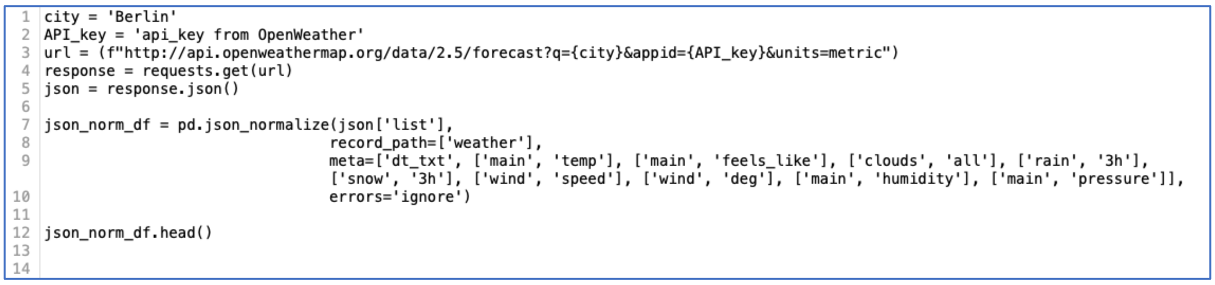
- Weather:

Firstly sign up for an account in **OpenWeather**, then we got the *api keys.* Secondly, we make a API call. There are two ways to get the data by APIs ways: using a "for" loop (Fig 2), or using '.json_normalize ()' (Fig 3).



```
def get_weather_loop(cities):

  API_key = 'api_key from OpenWeather'

  tz = pytz.timezone('Europe/Berlin')
  now = datetime.now().astimezone(tz)

  weather_dict = {'city': [],
                  'country': [],
                  'forecast_time': [],
                  'outlook': [],
                  'detailed_outlook': [],
                  'temperature': [],
                  'temperature_feels_like': [],
                  'clouds': [],
                  'rain': [],
                  'snow': [],
                  'wind_speed': [],
                  'wind_deg': [],
                  'humidity': [],
                  'pressure': [],
                  'information_retrieved_at': []}

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

    for i in json['list']:
      weather_dict['city'].append(json['city']['name'])
      weather_dict['country'].append(json['city']['country'])
      weather_dict['forecast_time'].append(i['dt_txt'])
      weather_dict['outlook'].append(i['weather'][0]['main'])
      weather_dict['detailed_outlook'].append(i['weather'][0]['description'])
      weather_dict['temperature'].append(i['main']['temp'])
      weather_dict['temperature_feels_like'].append(i['main']['feels_like'])
      weather_dict['clouds'].append(i['clouds']['all'])
      try:
          weather_dict['rain'].append(i['rain']['3h'])
      except:
          weather_dict['rain'].append('0')
      try:
          weather_dict['snow'].append(i['snow']['3h'])
      except:
          weather_dict['snow'].append('0')
      weather_dict['wind_speed'].append(i['wind']['speed'])
      weather_dict['wind_deg'].append(i['wind']['deg'])
      weather_dict['humidity'].append(i['main']['humidity'])
      weather_dict['pressure'].append(i['main']['pressure'])
      weather_dict['information_retrieved_at'].append(now.strftime("%d/%m/%Y %H:%M:%S"))
```

Fig.2 using a "for" loop to collect the weather information

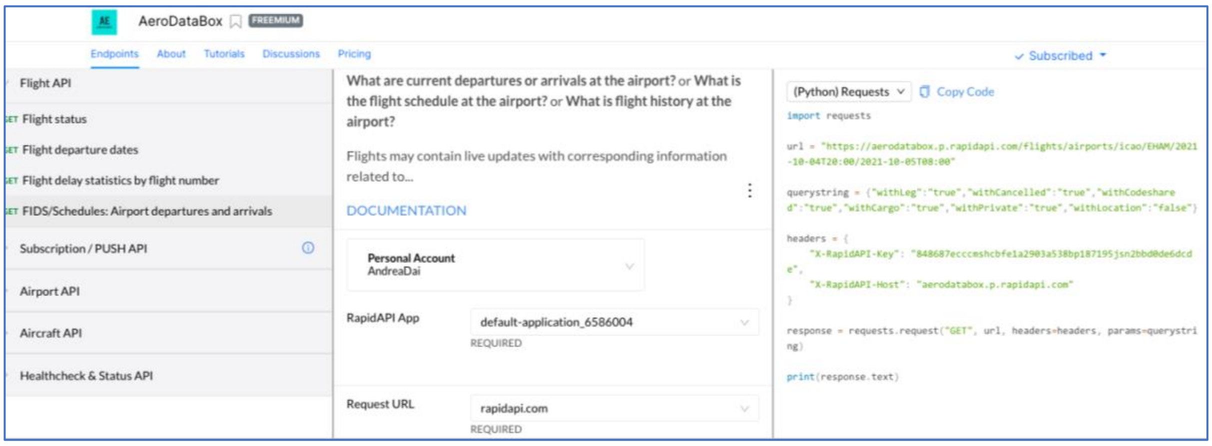


```python
city = 'Berlin'
API_key = 'api_key from OpenWeather'
url = (f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid={API_key}&units=metric")
response = requests.get(url)
json = response.json()

json_norm_df = pd.json_normalize(json['list'],
                                 record_path=['weather'],
                                 meta=['dt_txt', ['main', 'temp'], ['main', 'feels_like'], ['clouds', 'all'], ['rain', '3h'],
                                 ['snow', '3h'], ['wind', 'speed'], ['wind', 'deg'], ['main', 'humidity'], ['main', 'pressure']],
                                 errors='ignore')

json_norm_df.head()
```

Fig.3 using '.json_normalize ()' to collect the weather information

- Flights:

  Firstly sign up for an account in *Rapid API*. Secondly go to the Aerodatabox api and subscribe. I choose the Basic and free plan. Thirdly go to the *Endpoints section* of the API and select the 'FIDS/Schedules: Airport departures and arrivals'. Then set the parameters and choose the 'Python-request', finally copy the code to my Jupter Notebook (Fig 4).



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
AndreaDai

RapidAPI App: default-application_6586004
REQUIRED

Request URL: rapidapi.com
REQUIRED

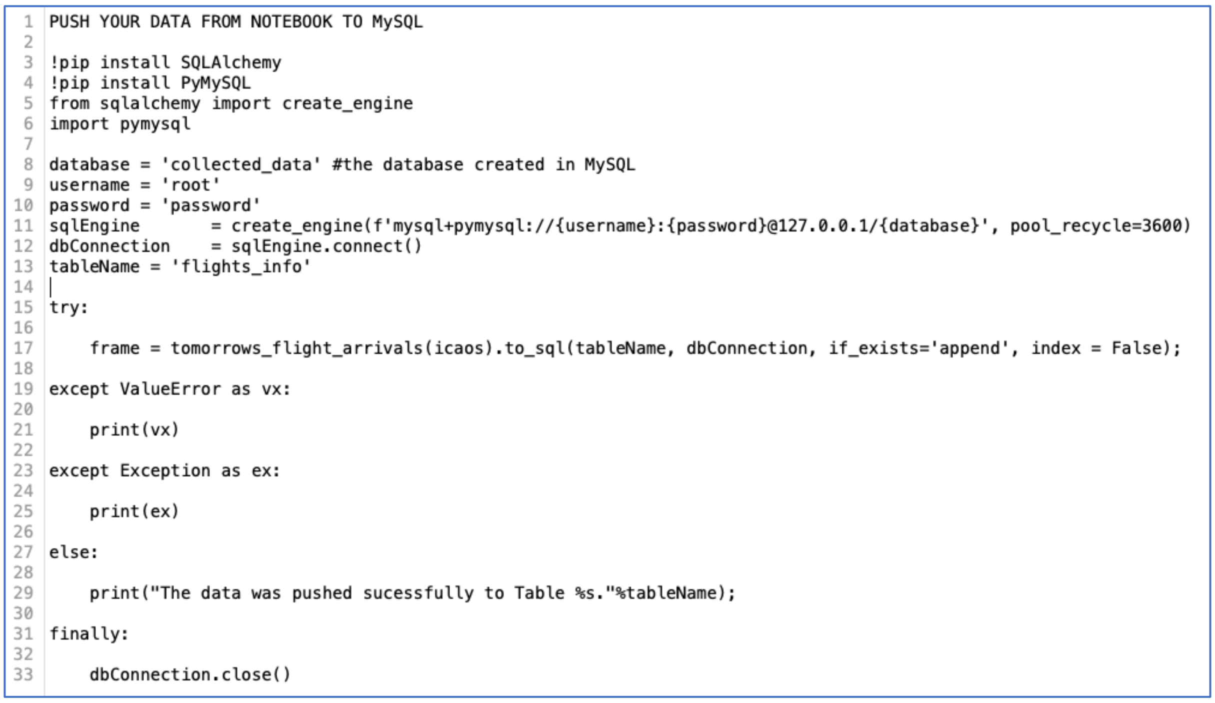
```python
import requests

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

querystring = {"withLeg":"true","withCancelled":"true","withCodeshared":"true","withCargo":"true","withPrivate":"true","withLocation":"false"}

headers = {
    "X-RapidAPI-Key": "848687ecccmshcbfe1a2903a538bp187195jsn2bbd0de6dcde",
    "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"
}

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

print(response.text)
```

Fig. 4 setting the API from Aerodatabox

3) store the data in a database

Push the data from Jupyter Notebook to local MySQL.

We used two typical ways, here I choose the simple way, which could push the DataFrame data directly to MySQL database (Fig. 5).



```
PUSH YOUR DATA FROM NOTEBOOK TO MySQL

!pip install SQLAlchemy
!pip install PyMySQL
from sqlalchemy import create_engine
import pymysql

database = 'collected_data' #the database created in MySQL
username = 'root'
password = 'password'
sqlEngine       = create_engine(f'mysql+pymysql://{username}:{password}@127.0.0.1/{database}', pool_recycle=3600)
dbConnection    = sqlEngine.connect()
tableName = 'flights_info'

try:

    frame = tomorrows_flight_arrivals(icaos).to_sql(tableName, dbConnection, if_exists='append', index = False);

except ValueError as vx:

    print(vx)

except Exception as ex:

    print(ex)

else:

    print("The data was pushed sucessfully to Table %s."%tableName);

finally:

    dbConnection.close()
```

Fig. 5 Push the data in ‘DataFrame’ format to the local MySQL

After pushing the data from the Jupyter notebook, I got the database in my local MySQL (Fig 6). Every time I execute the python code , the new collected data will be inserted to table. But in this way give it a disadvantage, that we need to refresh the data in the database manually if we need the real-time information of weather and flights, which will be time consuming and maybe not so specific correct. Therefore, I started the second project, moving the data pipeline to the cloud, in which way, I could automatically gather data from website and insert to local database in MySQL.

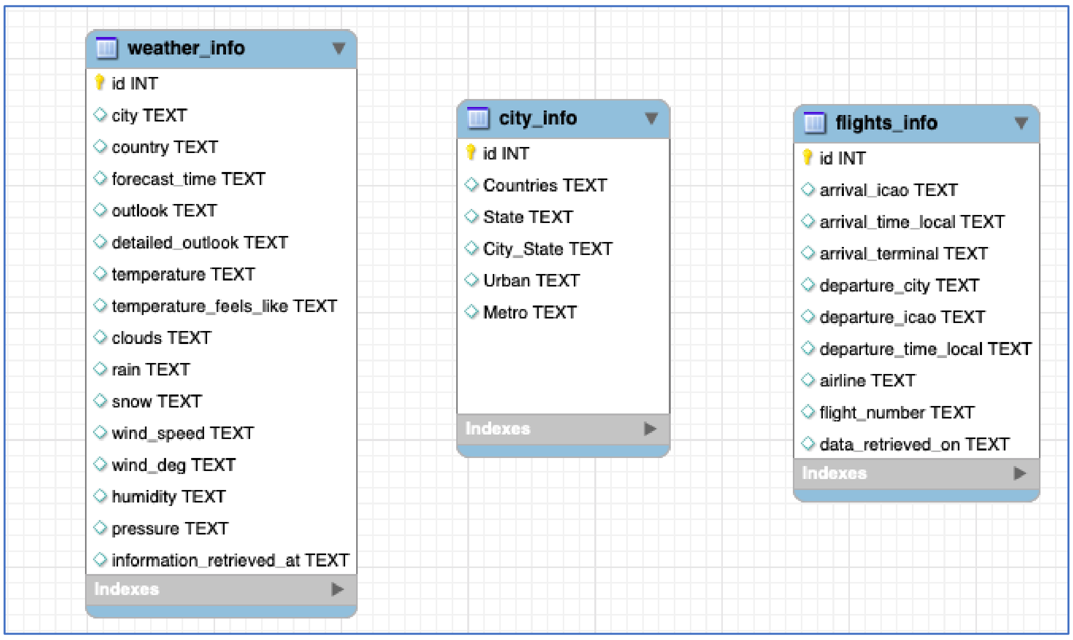


Fig. 6 collected information of city, weather and flights are stored as 3 tables in MySQL

2. Cloud Pipeline Project

In this section we use Amazon Web Service (AWS) to add a could pipeline. The process contains the following 3 steps:

1) Sign up for an AWS account.

Firstly we need sigh up an account on AWS, a tricky thing is that you need a valid credit card for it even you may not charge for anything. On AWS platform, there are lots of detail knowledge of AWS service, if you do not familiar with the service like Lambda, you could learn on AWS plateform. Meanwhile, it provides many specific training options, for example data science and machine learning etc.

Secondly we should set up a database on AWS RDS. The setting process contains many detailed steps, I will not list every steps here.

The last step will be to connect to the instance we created so that we can create a database with all of the tables to store the information that collected online.

2) Automate the data pipeline

After setting up a cloud MySQL instance, now we use AWS Lambda to run the Python scripts for data collection. We need three steps to create a Lambda function.

- Create a Role called 'LambdaAdminAccess' with the permission 'AdministratiorAccess'
- Create a Lambda function called 'weather_info' with the role 'LambdaAdminAccess'.
- Connect the Lambda function to RDS instance.

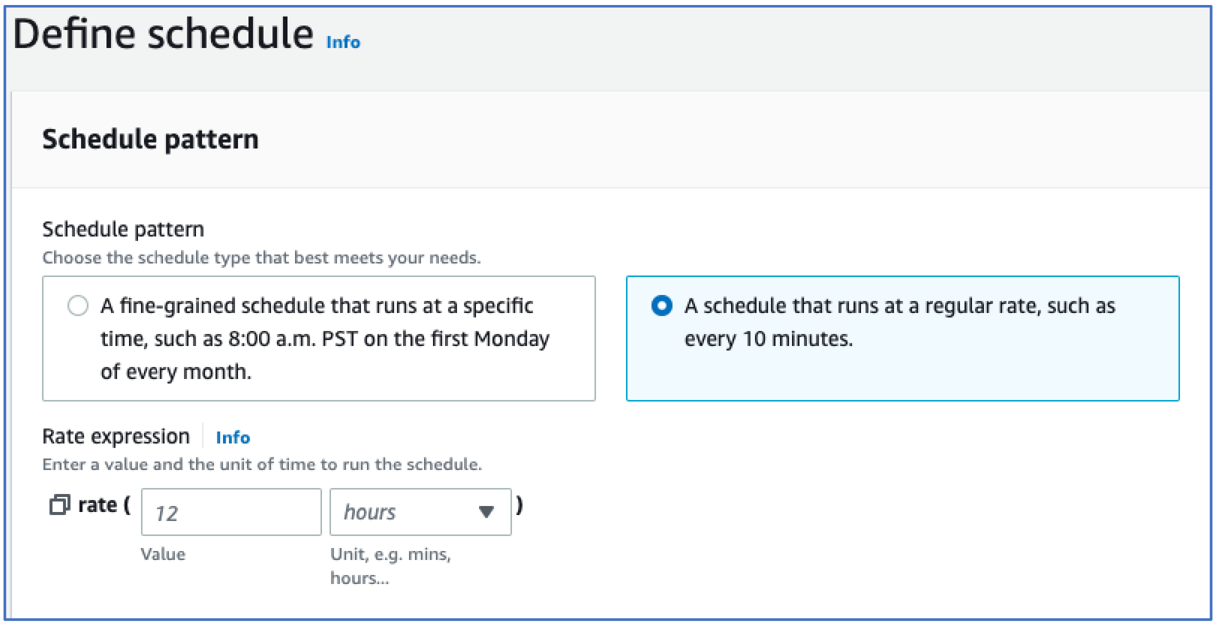
The Python scripts we run in Lambda function is almost the same with the one we push to the local database manfully from Jupter notebook. There are two main differences: one is we should add 'def lambda_handler(event, context):" in it , another is adding 'return' at the end of the function.

By run the Lambda functions, we have got our data from the internet and insert it onto the RDS cloud instance.

3) Collection data from website to local database automatedly

We use AWS EventBridge to set up a trigger event.
The schedule of running the Lambda function could be stetted in the 'Rule-Define schedule' (Fig. 7).

Now the cloud pipeline was settled down. We could collect the data automatically and in real-time.


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.]

Rate expression Info
Enter a value and the unit of time to run the schedule.

rate (12 hours)
Value — Unit, e.g. mins, hours...

Fig. 7 define the scheldule for Lambda function to execute