Data Engineering Project Report

Gans: data collection, storage and cloud pipelines

Prepared by
Dzmitry Afanasenkau

WBS Coding School
June 2022

1. Project description

Gans is a startup developing an e-scooter-sharing system which operates in the most populous European cities. In each city, it will have hundreds of e-scooters parked in the streets and allows users to rent them by the minute.
The usage of the scooters critically depends on the weather conditions and, in a large extend, on the number of tourists arriving to the city and the demographic situation.
The aim of the project is to **assemble and automate a pipeline in the cloud for collection and storage of the data** regarding *geographical and demographic information* about the cities of interest, *weather forecast* for 5 days and the *number of arriving and departing airplanes* in the nearest airports for each city.
The data collection was organized via different APIs using Python scripts. Data storage was organized using MySQL data base. Both data storage and automated data collection was performed in the cloud using Amazone Web Services (AWS). The complete pipeline is schematically represented in the fig 1.

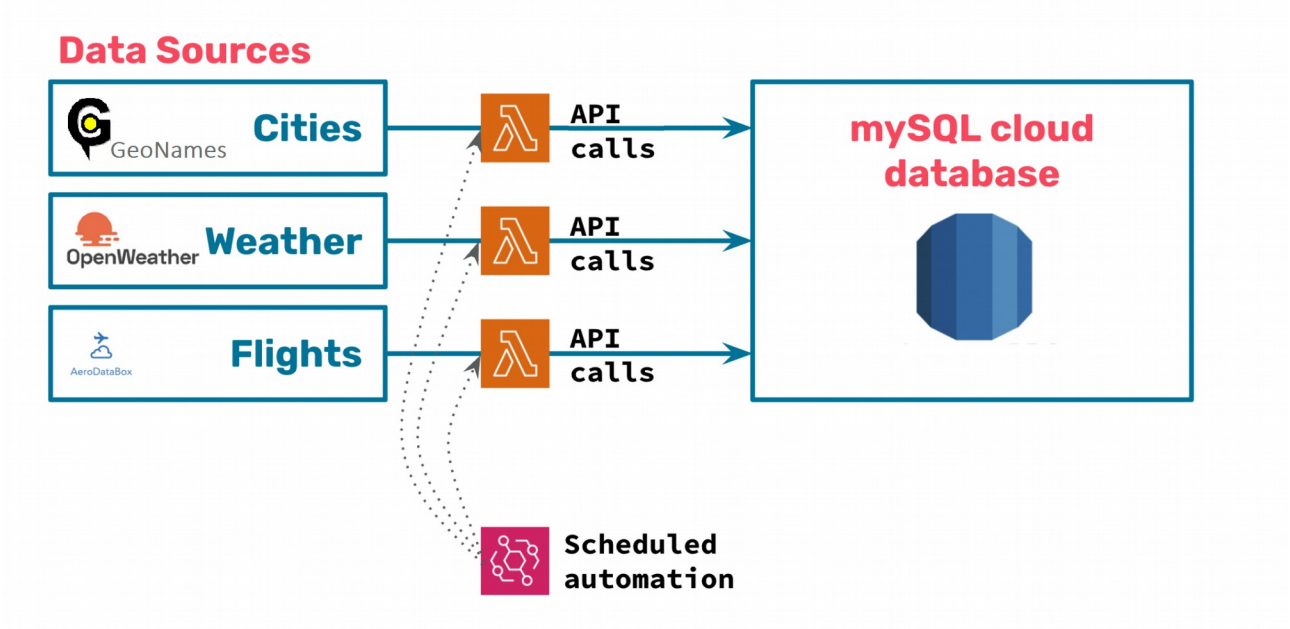

Fig. 1 Schematic representation of the data collection and storage pipeline

2. Data Collection

2.1 General notes

For the propose of this project the 4 groups of data were collected. These groups are

- geographical and demographic data about the cities
- weather forecast
- airports in vicinity of every city
- flights to theses airports was collected

For collection of each group of the data a separate python script was written (Get_cities.py, Get_weather.py, Get_airports.py, Get_flight.py). Each group of data was stored in a separate table in the database. All scripts are available on GitHub.

2.1 Geographical and demographicdata

Initial idea of collecting geographical city data by web scraping from Wikipedia was completely omitted due to inconsistency of the data positions on the different wiki pages. Instead, it was decided to use GeoNames API to complete this task. A python module *geocoder* was used to interact with the API (page for documentation: https://geocoder.readthedocs.io/index.html). The format of the request is the following

g = geocoder.geonames(c, method='details', key='GEO_NAMES_USER')

where c is the city ID. The returned data contains JSON formatted structure which was used to retrieve required information e.g.

lat = g.geojson['features'][0]['properties']['lat']

Initial table of the names of the cities of interest and their unique ID numbers were taken from SimpleMaps.com where this info was available for free in a convenient format of a .csv file. Using this data additional info was retrieved from GeoNames, namely: c*ountry name, country code, population, time zone, latitude* and *longitude*. The time zone is presented as time shift from UTC in seconds not accounted for the summer time shift.

2.2 Weather forecast

Information about weather forecast for the 5 days was retrieved using openweathermap API (documentation about the API can be found here https://openweathermap.org/forecast5). Communication with API is done using the *requests* python module. The format of the request is the following

url = f"http://api.openweathermap.org/data/2.5/forecastq={city}&appid={API_key}&units=metric"

response = requests.get(url)

The variables which are supplied to the URL string are

city – the name of the city

API_key – a key word identification to use the API

The free subscription API key provides 1 000 000 calls/month (60 calls/minute) which should be enough to get forecast for 15 cities used for this project.

The following data was retrieved from the API: *city_id, time_utc, temperature, humidity, cloudiness_pc, wind_speed, precipitation_prob, rain_volume, snow_volume*. An additional parameter, *local_time*, was calculated for the purpose of convenience on the basis of UTC time and time zone information.

2.3 Airports data

Information about the airports in the vicinity of each city is required to find the number of flights arriving and departing from the city. This info was retrieved from AeroDataBox API (section - Airports API: Search airport by location) which is available via RapidAPI. Communication with API is done using the *requests* python module. The format of the request is the following

```
url = f"https://aerodatabox.p.rapidapi.com/airports/search/location/{lat}/{lon}/km/50/16"

querystring = {"withFlightInfoOnly":"0"}

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

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

The parameters in the URL string:

lat – latitude of the city

lon – longitude of the city

The last two numbers in the url were left constant. They are: 50 – radius around the city to look for airports, 16 - Maximum number of airports to be returned.

2.4 Flights data

Information about flights were collected using AeroDataBox API (section – Flight API: FIDS/Schedules: Airport departures and arrivals by airport ICAO code) which is available via RapidAPI. Communication with API is done using the *requests* python module. The format of the request is the following

```
headers = {
      "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com",
      "X-RapidAPI-Key": "RAPID_API_KEY"
   }
url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{a_icao}/{date}T{t1}/{date}T{t2}"
response = requests.request("GET", url, headers=headers)
```

where

a_icao – ICAO code of the airport

date – date for which the data is retrieved

t1 and t2 – begin and end time for the time interval for which the data is retrieved

The API allows to get info only for 12 hours time period. To get info for 24 hours two requests have to be send.

AeroDataBox provides only 200 requests per month for free. It was enough for set up and test the pipe line. But for successful functioning of the system more requests are required. For this reason one should consider either to non-free pricing plan (e.g Ultra, $49.99 / month, 30 000 requests/month) or to go for another API.

3. Data Storage

3.1 Database model

Data initially collected with python scripts as Pandas DataFrames were exported to a MySQL data base. The data base named “gans2” contained 4 tables: cities, weathers, airports, and flights. The data base schema is presented in the fig.2

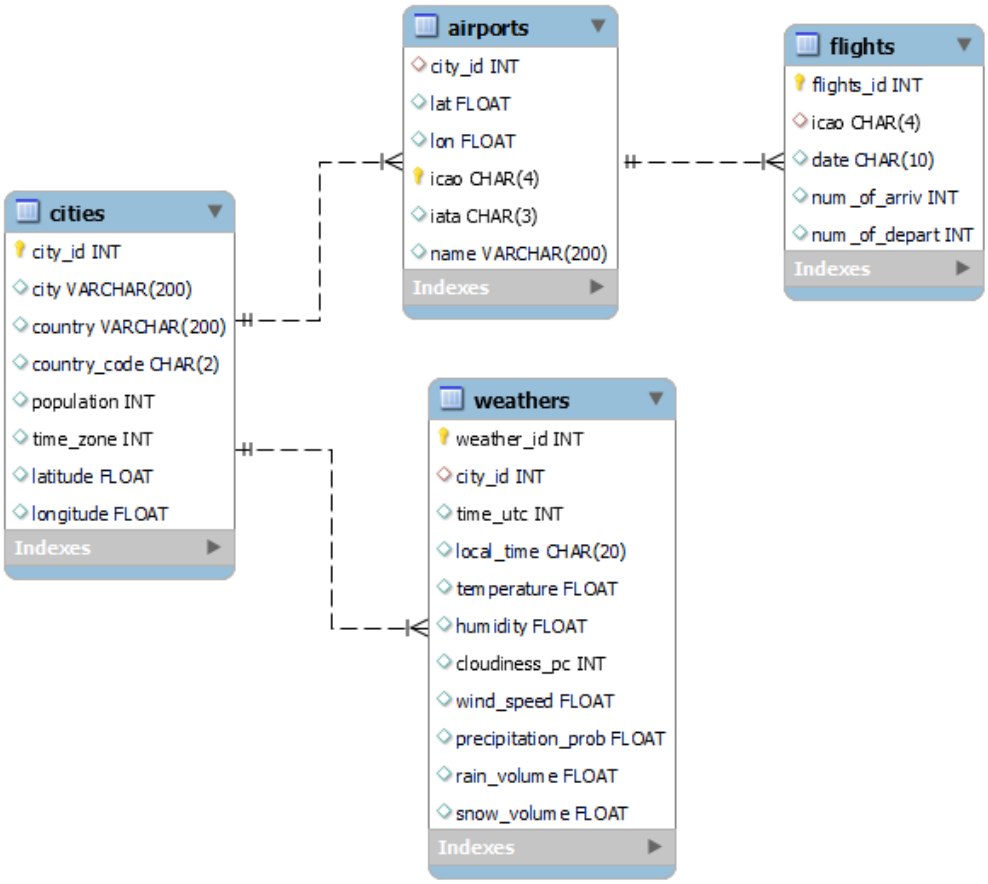

Fig. 2 A diagram representing the gans2 database schema

3.2 Cloud database set up

The RDS service of AWS was used to set up MySQL database in the cloud (DB identifier: wbs-project3-db-da).

4 Cloud pipeline

4.1 AWS Lambda

To run the code collecting data the AWS Lambda service was used. The Lambda function was set up in several steps.

- First a Role with AdministratorAccess was created.
- Then a Lambda called gans_data was set up with this execution Role.
- Lambda was connected to the RDS instance.
- Several layers with the required modules were added to this function.
    - Pandas was added in a layer created using AWSDataWrangler-Python38.

- A layer for SQLAlchemy was added as a so called Klayer from the GitHub repository of Keith Rozario (*Amazon* Resource Names is arn:aws:lambda:us-east-1:770693421928:layer:Klayers-p39-SQLAlchemy:3).
- A layer for such modules as requests and mysql-connector was created manually. Documentation about creating and sharing Lambda layers can be found on the folowing web page https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

The python scripts for weather and flights data collection were added to AWS lambda a separate files. Their content was reorganized into functions which are called from the lambda_handler function. Credentials for the data base access and for APIs were placed in a python file in the body of a function:

```
def get_data(key):
   secrets = {
   'DATABASE_HOST': "host_name",
   'DATABASE_USER': "admin",
   'DATABASE_PASSWORD': 'db_pass',
   ...
  }
   return secrets[key]
```

The data for cities and the airports were retrieved only once and pushed to the database manually.

4.2 Pipeline automation

To automate data collection and storage a trigger, namely EventBridge (CloudWatch Events) was added to the lambda function. The trigger was programmed to fire at 12:00 UTC every day.