Data Engineering: API to AWS to SQL

Florian Bogner

June 17, 2022

This technical report describes the necessary steps to pipe a selection of data into a database. We will deploy various APIs to gather the desired data. These data will be pushed to a SQL database in the cloud. Dynamic data will be collected in the cloud and automatically pushed to the database.

Introduction

The e-scooter company "Gans" seeks to expand into other European markets. E-scooters have to be strategically placed where they will be used a lot. Gans identified three key properties that will influence e-scooter demand: a city's features (population, elevation), weather and airport vicinity. Consequently, the characteristics of these three key aspects have to be investigated by data analysis. In turn, they represent the source of the necessary data.

This technical report documents the build-up of the heuristics to automatically collect data and populate a database with it. The documentation is structured as follows: First, the data collection by means of API calls will be described in Section 1. After that, Section 2 will be concerned with setting up a MySQL database. Finally, Section 3 will demonstrate the transition to the cloud.

1. Collecting Data

As Gans sought to expand into the German speaking markets, we decided to pick the top-ten German and the top-five Austrian cities by population. Initially, we performed web scraping on the Wikipedia pages of the respective cities in order to collect geo and population data. However, there are inconcistencies in the infobox in the Wikipedia entry of the cities which makes it impossible to collect uniform data. Therefore we have decided to use APIs for the entire data collection.

1.1. Get `cities` data

To collect data with a `get_cities(qcities)` function using the GeoDB Cities API from RapidAPI. This API takes the WikiData ID codes of the cities as input and provided us with the city name as well as with information about its elevation, latitudinal and longitudinal geo data and population.

```python
 for qcity in qcities:
  url = f"https://wft-geo-db.p.rapidapi.com/v1/geo/cities/{qcity}"
  headers = {"X-RapidAPI-Host": "wft-geo-db.p.rapidapi.com", "X-RapidAPI-
   ↪ Key": "..."}
  response = requests.request("GET", url, headers=headers)
  time.sleep(2)
  city_df = pd.json_normalize(response.json())
  cities_list.append(city_df)
cities_df = pd.concat(cities_list, ignore_index = True)
```

Listing 1: `for` loop to iterate through all cities, call the API and concatenate the DataFrames.

We have programmed our requests such that all city data are returned to us in one concatenated DataFrame. We have dropped the redundant columns and renamed the rest with more intuitive names. This is necessary since some flattened column names returned from the API include dots which will pose problems while handling them in Python or querying them in MySQL.

1.2. Get `weather` data

To collect weather data with a `get_weather(cities)` function, we deployed the Open-Weather API. This API took the city names from the `get_cities(qcities)` function as input and provided us with information about precipitation probability, temperature, humidity, cloudiness as well as with wind speed and gust.

```python
for city in cities:
  url = f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid
   ↪ =...&units=metric"
  weather = requests.get(url)
  weather_df = pd.json_normalize(weather.json()["list"])
  weather_df["city"] = city
  df_list.append(weather_df)
weather_combined = pd.concat(df_list, ignore_index = True)
```

Listing 2: `for` loop to iterate through all cities, call the API and concatenate the DataFrames.

We have programmed our requests such that all weather data are returned to us in one concatenated DataFrame. We have dropped the redundant columns and renamed the rest with more intuitive names. This is necessary since some flattened column names returned from the API include dots which will pose problems while handling them in Python or querying them in MySQL.

1.3. Get `airports` data

To collect airports data with a `get_airports(lat, lon)` function, we deployed the AeroDataBox API. This API took the latitudinal and longitudinal geo data from the `get_cities(qcities)` function as input and provided us with information about icao codes, airport names, municipality names, county codes as well as latitudinal and longitudinal geo data.

```python
for i in range(len(lat)):
  url = f"https://aerodatabox.p.rapidapi.com/airports/search/location/{lat[i]}/{lon[i]}/km/50/10"
  querystring = {"withFlightInfoOnly":"true"}
  headers = {"X-RapidAPI-Host": "aerodatabox.p.rapidapi.com", "X-RapidAPI-Key": "..."}
        response = requests.request("GET", url, headers=headers, params=querystring)
  airport_df = pd.json_normalize(response.json()["items"])
  airport_list.append(airport_df)
airports_df = pd.concat(airport_list, ignore_index = True)
```

Listing 3: `for` loop to iterate through all longitudinal and latitudinal geo data, call the API and concatenate the DataFrames.

We have programmed our requests such that all airports data are returned to us in one concatenated DataFrame. We have dropped the redundant columns and renamed the rest with more intuitive names. This is necessary since some flattened column names returned from the API include dots which will pose problems while handling them in Python or querying them in MySQL.

1.4. Get `arrivals` data

To collect arrivals data with a `get_arrivals(icao)` function, we deployed the AeroDataBox API. This API took the icao codes from the `get_airports(lat, lon)` function and returned flight numbers, ICAO codes of the departure airports, scheduled arrival times, airline names and aircraft models.

```python
for code in icao:
  url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{code}/{tomorrow}T10:00/{tomorrow}T22:00"
  querystring = {"withLeg":"false","direction":"Arrival","withCancelled":"false","withCodeshared":"true",
                 "withCargo":"false","withPrivate":"false","withLocation":"false"}
  headers = {"X-RapidAPI-Key": "...", "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"}
  response = requests.request("GET", url, headers = headers, params = querystring)
  arrival_df = pd.json_normalize(response.json()["arrivals"])
  arrival_df["arrival_icao"] = code
  arrival_list.append(arrival_df)
arrivals_df = pd.concat(arrival_list, ignore_index = True)
```

Listing 4: `for` loop to iterate through all icao airport codes, call the API and concatenate the DataFrames.

We have programmed our requests such that all flight arrivals data are returned to us in one concatenated DataFrame. We have dropped the redundant columns and renamed the rest with more intuitive names. This is necessary since some flattened column names returned from the API include dots which will pose problems while handling them in Python or querying them in MySQL.

2. SQL

The gathered data for `cities`, `weather`, `airports` and `arrivals` have to be stored in a MySQL database. This allows for data consistency, reduction of redundancy and for querying and analyzing the data. A new database will be created with the following code:

```
CREATE DATABASE gans;
USE gans;
```

Listing 5: A new database.

To connect to a MySQL database, we used the Python library SQLAlchemy. The code in Listing 6 establishes a connection to a MySQL database to which the DataFrames from Section 1 can be pushed.

```
schema="gans"
host='...rds.amazonaws.com'
user="admin"
password='...'
port=3306
con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'
```

Listing 6: Connection to MySQL database.

SQLAlchemy also handles the push of the DataFrames to the MySQL database with the following code in Listing 7. This will be done with `cities`, `weather`, `airports` and `arrivals` DataFrames.

```
DataFrame.to_sql('tablename', if_exists='append', con=con, index=False)
```

Listing 7: Push DataFrame to connected MySQL database.

This will populate the MySQL database with the respective DataFrames. However, the tables are still not connected and primary and foreign keys are missing. Altering the tables to have primary and foreign keys as well as changing the table's column's datatypes will be described in the following section.

As the DataFrames were pushed to an empty database, keys and datatypes had to be set afterwards. For the example of `weather`, a new column was added to act as a primary key:

```
ALTER TABLE weather
ADD weather_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;
```

Listing 8: Add a new column for the primary key of the `weather` table.

Only after that, it was possible to define the datatypes as shown below:

```sql
ALTER TABLE `gans`.`weather`
CHANGE COLUMN `precip_prob` `precip_prob` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `temperature` `temperature` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `humidity` `humidity` INT NULL DEFAULT NULL ,
CHANGE COLUMN `cloudiness` `cloudiness` INT NULL DEFAULT NULL ,
CHANGE COLUMN `wind_speed` `wind_speed` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `wind_gust` `wind_gust` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `city` `city` VARCHAR(100) NULL DEFAULT NULL ,
CHANGE COLUMN `city_id` `city_id` VARCHAR(10) NULL DEFAULT NULL ;
```

Listing 9: Changing the datatypes of the `weather` table.

Lastly, the foreign key could be set:

```sql
ALTER TABLE `gans`.`weather`
ADD INDEX `weather_city_id_idx` (`city_id` ASC) VISIBLE;
ALTER TABLE `gans`.`weather`
ADD CONSTRAINT `weather_city_id`
  FOREIGN KEY (`city_id`)
  REFERENCES `gans`.`cities` (`city_id`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION;
```

Listing 10: Set foreign key for the `weather` table.

After changing all of the table's column's datatypes and defining primary and foreign keys, the database was set. Figure 1 shows the final schema with all necessary relations between the tables. The entire MySQL script that leads to this very schema is appended in the Listing 11 in Appendix A.1.

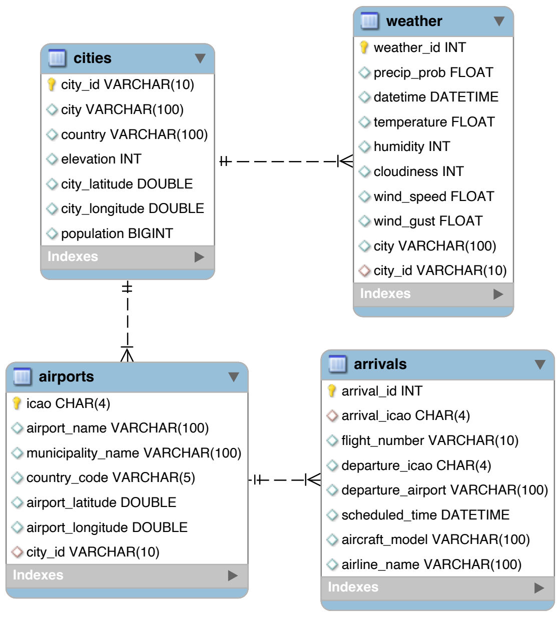

Figure 1: MySQL schema.

3. AWS

Gathering dynamic data can be facilitated by cloud computing and cloud storage. Both the prior Python code as well als the MySQL database can be transferred to the cloud. This enables API calls to be periodically triggered as well as their data to be automatically pushed to a cloud instance of a MySQL database. The service provider of choice will be Amazon Web Services (AWS)

In order to have a cloud instance of a MySQL database, we deployed one in AWS' Relational Database Service. Now, it would not be any local database that would be filled with data but the one in the AWS cloud servers.

Similarly, Python code would not be executed locally but also in the cloud. Therefore, we used AWS' Lambda functions. Their Lambda handler functions allowed us to execute the API calls, database connections and data pushes to this very database in one go for `cities`, `weather`, `airports` and `arrivals`. The contents for all four of them are appended in Appendix A.2.

As a final step, two of the Lambda functions had to be automated. While `cities` and `airports` are rather static in nature `weather` and flight `arrivals` data are dynamic and thus subject to rapid changes over time. Therefore, we set for the latter two Lambda functions a custom trigger, namely an EventBridge (CloudWatch Events) that pulled data from the OpenWeather and AeroDataBox APIs on a daily basis by deploying `59 21 ? * * *` as a Cron expression.

Conclusion

This technical report documented the data flow from API to AWS to SQL. Figure 2 illustrates the pipeline. Data is pulled from APIs for `cities`, `weather`, `airports` and `arrivals` data. The API for `cities` takes WikiData IDs and returns the respecitve city name as well as information about its elevation, latitudinal and longitudinal geo data and population. The city name is then piped into the API for the `weather` call. This provides us with information about precipitation probability, temperature, humidity, cloudiness as well as with wind speed and gust. Apart from that, latitudinal and longitudinal data from `cities` is also used by the API for `airports` which returns information about icao codes, airport names, municipality names, county codes as well as latitudinal and longitudinal geo data. Finally, the ICAO airport codes from `airports` are used to gather flights `arrivals` data about flight numbers, ICAO codes of the departure airports, scheduled arrival times, airline names and aircraft models.

Dynamic `weather` and flight `arrivals` data were pulled on a daily basis. All data was pushed to a MySQL database instance on AWS' RDS.

This pipeline allows the database to be queried, analyze and visualized by Python pandas.

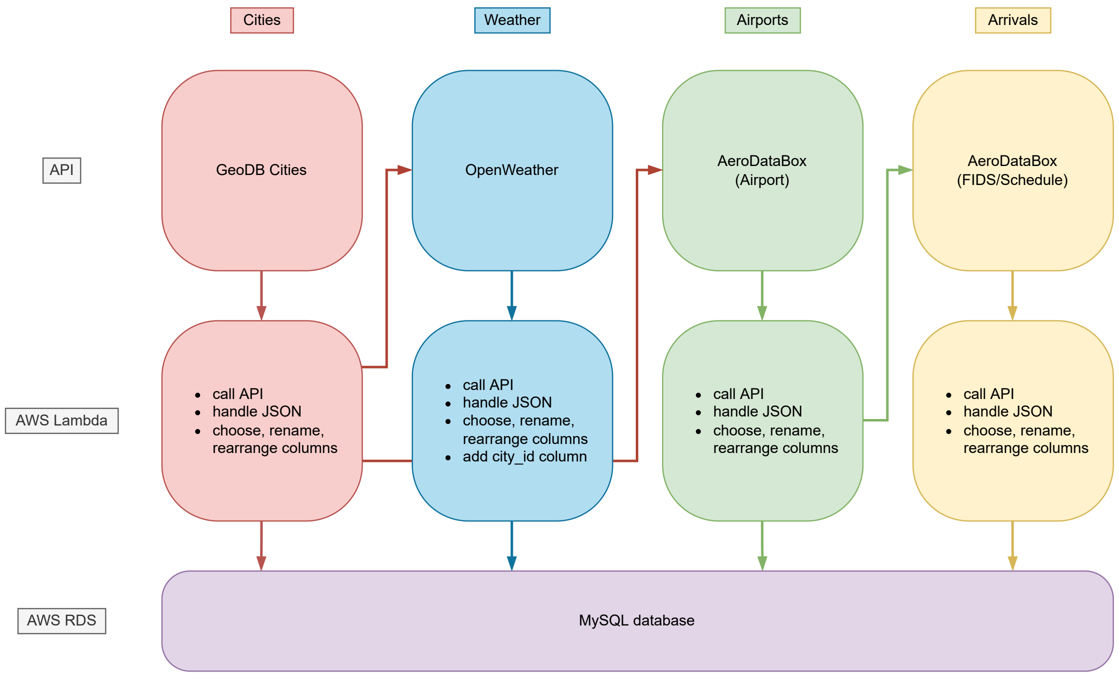

Figure 2: Pipeline from API to database.

A. Appendix

A.1. SQL

```
CREATE DATABASE gans;
USE gans;

-----------------------------------------------------------------------

-- SETTING UP CITIES:
-- Alter table datatypes of the columns and set primary key:
ALTER TABLE `gans`.`cities`
CHANGE COLUMN `city_id` `city_id` VARCHAR(10) NOT NULL ,
CHANGE COLUMN `city` `city` VARCHAR(100) NULL DEFAULT NULL ,
CHANGE COLUMN `country` `country` VARCHAR(100) NULL DEFAULT NULL ,
CHANGE COLUMN `elevation` `elevation` INT NULL DEFAULT NULL ,
ADD PRIMARY KEY (`city_id`);

-----------------------------------------------------------------------

-- SETTING UP WEATHER:
-- BEFORE altering datatypes and setting keys: Weather has duplicate
   ↪ datetime values; therefore a new column with unique values is
   ↪ necessary:
ALTER TABLE weather
ADD weather_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;
```

```

-- Alter table datatypes of the columns and set primary key:
ALTER TABLE `gans`.`weather`
CHANGE COLUMN `precip_prob` `precip_prob` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `temperature` `temperature` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `humidity` `humidity` INT NULL DEFAULT NULL ,
CHANGE COLUMN `cloudiness` `cloudiness` INT NULL DEFAULT NULL ,
CHANGE COLUMN `wind_speed` `wind_speed` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `wind_gust` `wind_gust` FLOAT NULL DEFAULT NULL ,
CHANGE COLUMN `city` `city` VARCHAR(100) NULL DEFAULT NULL ,
CHANGE COLUMN `city_id` `city_id` VARCHAR(10) NULL DEFAULT NULL ;

-- Setting foreign key:
ALTER TABLE `gans`.`weather`
ADD INDEX `weather_city_id_idx` (`city_id` ASC) VISIBLE;
ALTER TABLE `gans`.`weather`
ADD CONSTRAINT `weather_city_id`
  FOREIGN KEY (`city_id`)
  REFERENCES `gans`.`cities` (`city_id`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION;

-------------------------------------------------------------------------

-- SETTING UP ARRIVALS:
-- BEFORE altering datatypes and setting keys: Arrivals has duplicate
    ↪ datetime values; therefore a new column with unique values is
    ↪ necessary:
ALTER TABLE arrivals
ADD arrival_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;

-- Alter table datatypes of the columns and set primary key:
ALTER TABLE `gans`.`arrivals`
CHANGE COLUMN `arrival_icao` `arrival_icao` CHAR(4) NULL DEFAULT NULL ,
CHANGE COLUMN `flight_number` `flight_number` VARCHAR(10) NULL DEFAULT
    ↪ NULL ,
CHANGE COLUMN `departure_icao` `departure_icao` CHAR(4) NULL DEFAULT NULL
    ↪  ,
CHANGE COLUMN `departure_airport` `departure_airport` VARCHAR(100) NULL
    ↪ DEFAULT NULL ,
CHANGE COLUMN `scheduled_time` `scheduled_time` DATETIME NULL DEFAULT
    ↪ NULL ,
CHANGE COLUMN `aircraft_model` `aircraft_model` VARCHAR(100) NULL DEFAULT
    ↪  NULL ,
CHANGE COLUMN `airline_name` `airline_name` VARCHAR(100) NULL DEFAULT
    ↪ NULL ;

-- Setting foreign key:
ALTER TABLE `gans`.`arrivals`
ADD INDEX `arrivals_icao_idx` (`arrival_icao` ASC) VISIBLE;
ALTER TABLE `gans`.`arrivals`
ADD CONSTRAINT `arrivals_icao`
  FOREIGN KEY (`arrival_icao`)
  REFERENCES `gans`.`airports` (`icao`)
```

```sql
  ON DELETE NO ACTION
  ON UPDATE NO ACTION;

-----------------------------------------------------------------------

-- SETTING UP AIRPORTS:
-- Alter table datatypes of the columns and set primary key:
ALTER TABLE `gans`.`airports`
CHANGE COLUMN `icao` `icao` CHAR(4) NOT NULL ,
CHANGE COLUMN `airport_name` `airport_name` VARCHAR(100) NULL DEFAULT
    ↪ NULL ,
CHANGE COLUMN `municipality_name` `municipality_name` VARCHAR(100) NULL
    ↪ DEFAULT NULL ,
CHANGE COLUMN `country_code` `country_code` VARCHAR(5) NULL DEFAULT NULL
    ↪ ,
CHANGE COLUMN `city_id` `city_id` VARCHAR(10) NULL DEFAULT NULL ,
ADD PRIMARY KEY (`icao`);

-- Drop localCode column from airports:
ALTER TABLE `gans`.`airports`
DROP COLUMN `localCode`;

-- Setting foreign key:
ALTER TABLE `gans`.`airports`
ADD INDEX `airports_city_id_idx` (`city_id` ASC) VISIBLE;
ALTER TABLE `gans`.`airports`
ADD CONSTRAINT `airports_city_id`
  FOREIGN KEY (`city_id`)
  REFERENCES `gans`.`cities` (`city_id`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION;
```

Listing 11: MySQL script to set up database schema.

A.2. AWS Lambda functions

A.2.1. Cities

```python
import requests
import pymysql
import sqlalchemy
import pandas as pd
import json
import time
from datetime import datetime, date, timedelta
from pytz import timezone

def lambda_handler(event, context):
    # Define cities
    qcities = [# 10 German cities by population:
    'Q64',      # Berlin
    #'Q1055',   # Hamburg
    #'Q1726',   # München
    'Q365',    # Köln
    #'Q1794',  # Frankfurt am Main
    #'Q1022',  # Stuttgart
    'Q1718',   # Düsseldorf
    #'Q1295',  # Dortmund
    #'Q2066',  # Essen
    #'Q2079',  # Leipzig

               # 5 Austrian cities by population:
    #'Q1741',  # Wien
    #'Q13298', # Graz
    #'Q41329', # Linz
    #'Q34713', # Salzburg
    #'Q1735'   # Innsbruck
    ]

    # Get cities:
    def get_cities(qcities):

        # Connect to database:
        schema="gans"
        host='...rds.amazonaws.com'
        user="admin"
        password='...'
        port=3306
        con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'

        # Make cities_df:
        cities_list = []
        for qcity in qcities:
            url = f"https://wft-geo-db.p.rapidapi.com/v1/geo/cities/{qcity}"
            headers = {"X-RapidAPI-Host": "wft-geo-db.p.rapidapi.com", "X-RapidAPI-Key": "..."}
            response = requests.request("GET", url, headers=headers)
            time.sleep(2)
            city_df = pd.json_normalize(response.json())
            cities_list.append(city_df)
        cities_df = pd.concat(cities_list, ignore_index = True)
        cities_df = cities_df[["data.wikiDataId",
                         "data.city",
                         "data.country",
                         "data.elevationMeters",
                         "data.latitude",
                         "data.longitude",
                         "data.population"]]
        cities_df.rename(columns = {'data.wikiDataId': 'city_id',
                            'data.city': 'city' ,
                            'data.country': 'country',
                            'data.elevationMeters': 'elevation',
                            'data.latitude': 'city_latitude',
                            'data.longitude': 'city_longitude',
                            'data.population': 'population'},
                        inplace = True)

        # Push to MySQL:
        cities_df.to_sql('cities', if_exists='append', con=con, index=
    ↪ False)

    get_cities(qcities)
```

Listing 12: Lambda function pulls cities data from API, applies post-hoc processing to dataframe, connects to the RDS cloud instance MySQL and pushes the data to the database.

A.2.2. Weather

```
import requests
import pymysql
import sqlalchemy
import pandas as pd
import json
import time
from datetime import datetime, date, timedelta
from pytz import timezone

def get_weather(cities):
    df_list = []
    for city in cities:
        url = f"http://api.openweathermap.org/data/2.5/forecast?q={city}&
    ↪ appid=...&units=metric"
        weather = requests.get(url)
        weather_df = pd.json_normalize(weather.json()["list"])
        weather_df["city"] = city
        df_list.append(weather_df)
    weather_combined = pd.concat(df_list, ignore_index = True)
    weather_combined = weather_combined[["pop", "dt_txt", "main.temp", "
    ↪ main.humidity", "clouds.all", "wind.speed", "wind.gust", "city"]]
    weather_combined.rename(columns = {'pop': 'precip_prob',
                             'dt_txt': 'datetime',
                             'main.temp': 'temperature',
                             'main.humidity': 'humidity',
                             'clouds.all': 'cloudiness',
                             'wind.speed': 'wind_speed',
                             'wind.gust': 'wind_gust'},
                 inplace = True)
    weather_combined['datetime'] = pd.to_datetime(weather_combined['
    ↪ datetime'])
    return weather_combined

def lambda_handler(event, context):
    # Connect to database:
    schema="gans"
    host='...rds.amazonaws.com'
    user="admin"
    password='...'
```

```python
    port=3306
    con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'

    # Make new df for city_id and city columns from SQL query so that we eventually have the city_id as a foreign key in the weather df:
    city_df = pd.read_sql('SELECT city_id, city FROM cities', con=con)

    # Use the queried SQL df to have the city names as a later input for the get_weather function:
    cities = city_df['city'].to_list()

    # Calling the get_weather function:
    weather_output = get_weather(cities)

    # Merging the queried SQL df on 'city' onto the weather_output df which results in an additional column with 'city_id':
    weather_d = weather_output.merge(city_df, how = 'left')

    # Push to MySQL:
    weather_d.to_sql('weather', if_exists='append', con=con, index=False)
```

Listing 13: Lambda function pulls weather data from API, applies post-hoc processing to dataframe, connects to the RDS cloud instance MySQL and pushes the data to the database.

A.2.3. Airports

```python
import requests
import pymysql
import sqlalchemy
import pandas as pd
import json
import time
from datetime import datetime, date, timedelta
from pytz import timezone

def get_airports(lat, lon):
    airport_list = []
    for i in range(len(lat)):
        url = f"https://aerodatabox.p.rapidapi.com/airports/search/location/{lat[i]}/{lon[i]}/km/50/10"
        querystring = {"withFlightInfoOnly":"true"}
        headers = {"X-RapidAPI-Host": "aerodatabox.p.rapidapi.com", "X-RapidAPI-Key": "..."}
        response = requests.request("GET", url, headers=headers, params=querystring)
        airport_df = pd.json_normalize(response.json()["items"])
        airport_list.append(airport_df)
    airports_df = pd.concat(airport_list, ignore_index = True)
    airports_df.drop_duplicates(subset ='icao', inplace = True)
    airports_df = airports_df[~airports_df.name.str.contains("Air Base", case = False)]
    airports_df.drop(columns = ['iata', 'shortName'], inplace = True)
    airports_df.reset_index(drop = True, inplace = True)
    airports_df['city_id'] = [
                          'Q64',
                          'Q64',
                          #'Q1055',
                          #'Q1726',
                          'Q365',
                          'Q1718',
                          #'Q1794',
                          #'Q1022',
                          #'Q1295',
                          #'Q2079',
                          #'Q1741',
                          #'Q13298',
                          #'Q41329',
                          #'Q34713',
                          #'Q1735'
                          ]
    airports_df.rename(columns = {'name': 'airport_name',
                                'municipalityName': 'municipality_name',
                                'countryCode': 'country_code',
                                'location.lat': 'airport_latitude',
                                'location.lon': 'airport_longitude',
                                },
                                inplace = True)
    return airports_df

def lambda_handler(event, context):
    # Connect to database:
    schema="gans"
    host='...rds.amazonaws.com'
    user="admin"
    password='...'
    port=3306
    con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'

    # Make new df for city_id and city columns from SQL query so that we eventually have the city_id as a foreign key in the weather df:
    city_df = pd.read_sql('SELECT city_latitude, city_longitude FROM cities', con=con)

    # Calling the get_airports function:
    airports_output = get_airports(city_df['city_latitude'].to_list(), city_df['city_longitude'])

    # Push to MySQL:
    airports_output.to_sql('airports', if_exists='append', con=con, index=False)
```

Listing 14: Lambda function pulls cities data from API, applies post-hoc processing to dataframe, connects to the RDS cloud instance MySQL and pushes the data to the database.

A.2.4. Arrivals

```python
import requests
import pymysql
import sqlalchemy
import pandas as pd
import json
import time
from datetime import datetime, date, timedelta
from pytz import timezone

def get_arrivals(icao):
  arrival_list = []
  today = datetime.now().astimezone(timezone('Europe/Berlin')).date()
  tomorrow = (today + timedelta(days = 1))
  for code in icao:
    url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{code}/{tomorrow}T10:00/{tomorrow}T22:00"
    querystring = {"withLeg":"false","direction":"Arrival","withCancelled":"false","withCodeshared":"true",
                    "withCargo":"false","withPrivate":"false","withLocation":"false"}
    headers = {"X-RapidAPI-Key": "...", "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com"}
    response = requests.request("GET", url, headers = headers, params = querystring)
    arrival_df = pd.json_normalize(response.json()["arrivals"])
    arrival_df["arrival_icao"] = code
    arrival_list.append(arrival_df)
  arrivals_df = pd.concat(arrival_list, ignore_index = True)
  arrivals_df.drop(columns = ['isCargo',
                                 'status',
                                 'callSign',
                                 'codeshareStatus',
                                 'movement.airport.iata',
                                 #'movement.actualTimeLocal',
                                 'movement.quality',
                                 'aircraft.reg',
                                 'aircraft.modeS',
                                 'movement.terminal',
                                 'movement.scheduledTimeUtc',
                                 #'movement.actualTimeUtc',
                                 #'movement.baggageBelt',
                                 #'movement.gate'
                                 ],
                      inplace = True)
  arrivals_df.rename(columns = {'number': 'flight_number',
                                   'movement.airport.icao': 'departure_icao',
                                   'movement.airport.name': 'departure_airport',
                                   'movement.scheduledTimeLocal': 'scheduled_time',
                                   'aircraft.model': 'aircraft_model',
                                          'airline.name': 'airline_name'

  },
                            inplace = True)
  arrivals_df['scheduled_time'] = pd.to_datetime(arrivals_df['
    ↪ scheduled_time'])
  return arrivals_df

def lambda_handler(event, context):
    # Connect to database:
    schema="gans"
    host='...rds.amazonaws.com'
    user="admin"
    password='...'
    port=3306
    con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{schema}'

    # Make new df for city_id and city columns from SQL query so that we
    ↪ eventually have the city_id as a foreign key in the weather df:
    icao_df = pd.read_sql('SELECT icao FROM airports', con=con)

    icao = icao_df['icao'].to_list()

    # Call the get_arrivals:
    arrivals_output = get_arrivals(icao)

     # Push to MySQL:
    arrivals_output.to_sql('arrivals', if_exists='append', con=con, index
    ↪ =False)
```

Listing 15: Lambda function pulls flight arrivals data from API, applies post-hoc processing to dataframe, connects to the RDS cloud instance MySQL and pushes the data to the database.