Technical Report - Data Engineering for company Gans

Connect to database and create tables

We import all required packages and set a connection to the the database in the cloud:

In [ ]:

```
# For this project, we need to import the following packages:
import requests
import sqlalchemy
import pandas as pd
from pandas.io.json import json_normalize
import json
from bs4 import BeautifulSoup
import re
import numpy as np
import time
import datetime
from datetime import date
import mysql.connector



############ name of the database we want to create
dbname = "project5"

############ list of cities we want to consider
citylist = ['Frankfurt am Main', "Munich"]
# citylist = ['Baden-Baden', "Berlin",  "Bonn", 'Bremen', 'Dresden', 'Dortmund', 'Dü


# At this point, we need to connect to the database in the cloud (we use the service
############ AWS
sethost = "vdreiffm.cxmmedv6eu66.eu-central-1.rds.amazonaws.com"
setuser = "admin"
setpassword = "0123456789"
setport = 3306

############ LOCAL
# sethost = "127.0.0.1"
# setuser = "root"
# setpassword = "z7tKXYB#"
# setport = 3306



########### Access for SQLAlchemy
host = sethost
user = setuser
password = setpassword
port = setport
con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{dbname}'



############ Access for SQLConnector
import mysql.connector
def connect():
    return(mysql.connector.connect(
    user=setuser,
    password=setpassword,
    host=sethost,
    ))

cnx = connect()
cursor = cnx.cursor()
```

At this point, we want to create the database "dbname" and its tables. To do so, we use SQL commands embeddet in python code via the mysql-connector. We also set primary and foreign keys for each table. These attributes and internal security routines in mysql ensure the consistency of the tables relations and consistency of new data.

In [18]:

```
cursor.execute(f"CREATE DATABASE IF NOT EXISTS {dbname}")

cursor.execute(f"USE {dbname}")

cursor.execute(
    "CREATE TABLE IF NOT EXISTS cities("
    "name VARCHAR(255),"
    "country VARCHAR(255),"
    "country_code VARCHAR(255),"
    "wiki_data_id VARCHAR(255),"
    "latitude NUMERIC,"
    "longitude NUMERIC,"
    "population INT,"
    "timezone VARCHAR(255),"
    "PRIMARY KEY(name)   )"
)

cursor.execute(
    "CREATE TABLE IF NOT EXISTS weather("
    "id INT AUTO_INCREMENT,"
    "city VARCHAR(255),"
    "date_time date,"
    "temperature INT,"
    "rain VARCHAR(6),"
    "clouds VARCHAR(255),"
    "PRIMARY KEY(id),"
    "FOREIGN KEY(city) REFERENCES cities(name)   )"
)

cursor.execute(
    "CREATE TABLE IF NOT EXISTS airports("
    "icao VARCHAR(4),"
    "airport VARCHAR(255),"
    "city VARCHAR(255),"
    "PRIMARY KEY(icao),"
    "FOREIGN KEY(city) REFERENCES cities(name)   )"
)

cursor.execute(
    "CREATE TABLE IF NOT EXISTS arrivals("
    "flightnumber VARCHAR(15),"
    "status VARCHAR(255),"
    "departure_airport_icao     VARCHAR(255),"
    "departure_airport_iata     VARCHAR(255),"
    "departure_airport_name     VARCHAR(255),"
    "arrival_scheduledTimeLocal DATETIME,"
    "arrival_actualTimeLocal DATETIME,"
    "arrival_scheduledTimeUtc DATETIME,"
    "arrival_actualTimeUtc DATETIME,"
    "arrival_terminal VARCHAR(255),"
    "aircraft_model     VARCHAR(255),"
    "airline_name VARCHAR(255),"
    "arrival_airport_icao VARCHAR(255),"
    "PRIMARY KEY(flightnumber),"
    "FOREIGN KEY(arrival_airport_icao) REFERENCES airports(icao)   )"
)
```

We visualize the schema:

fishy

Fill tables with collected data

Now, we want to fill the tables with data. Due to the primary and foreign key relations, we have to start with the table "cities". Its primary key "name" is the foreign key of the tables "weather" and "airports". That means that one can only add new lines to the latter tables if a matching primary key (the city name) in "cities" exists. To fill each table, we will create a function that collects the favored data.

Table cities

In [3]:

```
# required packages
import requests
import sqlalchemy
import pandas as pd
import json
from bs4 import BeautifulSoup
import re
import time



# the strategy is first to get the wikidataid and than collect the data from geoDB
def demo(cities): # as input, we use a list of cities
    cities_id = [] # initiate an empty id list
    dfList = []
    for city in cities:
        #retrieve the wikidataId
        time.sleep(1) # slows down the execution therby the server don't block our q
        url1 = f'https://en.wikipedia.org/wiki/{city}' #go to the wiki site of the c
        citem = requests.get(url1, 'html.parser') # get the html
        if BeautifulSoup(citem.content) != None:
            soup = BeautifulSoup(citem.content)   # soup the content
        if soup.find('li', {'id':'t-wikibase'}).find('a')['href'] != None:
            wikidata_link = soup.find('li', {'id':'t-wikibase'}).find('a')['href'] #
        # wl.append(wikidata_link)
        # \d+ is a regular expression and means one digit or more, the wiki data id
        #for group() in re see: https://www.tutorialspoint.com/What-is-the-groups-me
        city_id = re.search('Q\d+', wikidata_link).group()
        cities_id.append(city_id)
        #use the wikidataId to retrieve infrormation from geoDB
        url2 = "https://wft-geo-db.p.rapidapi.com/v1/geo/cities/{}".format(city_id)
        headers = {
        "X-RapidAPI-Key": "3cd15bf266msh2331a2a034ea490p1c96a2jsn828e9dce3a50",
        "X-RapidAPI-Host": "wft-geo-db.p.rapidapi.com"
        }
        response = requests.request("GET", url2, headers=headers)# gets a json-like
        cit_dic = {}#make a dictionary to retrieve the information
        cit_dic['name'] = response.json()['data']['name']
        cit_dic['country'] = response.json()['data']['country']
        cit_dic['country_code'] = response.json()['data']['countryCode']
        cit_dic['wiki_data_id'] = 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) #put it in a list
        df_demo = pd.DataFrame(dfList) # transform the list to df

    return df_demo



demodata = demo(citylist)  # stores the collected data in demodata
```

In [30]:

```
# optionally, we save the data in a .csv file
# demodata.to_csv('demodata.csv', index=False)
```

In [19]:

```
# campare with the row above. We read the data from the genereted .csv
demodata = pd.read_csv('demodata.csv')
```

In [20]:

```
# we connect to the database and insert our data in the table "cities"
demodata.to_sql("cities", con=con, index=False, if_exists='append')
```

Table weather

In [5]:

```
# required packages
import json
import sqlalchemy
import requests
import pandas as pd
from pandas.io.json import json_normalize
import numpy as np



def weather(cities): # as input, we use a list of cities
    storage = pd.DataFrame() # here we store the data from each city
    API_key = "d3645498e615aef5ea56aba13e895b36"   # the key for the API service

    for city in cities: # we iterate over each city
        url = f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid={API_
        response = requests.get(url)
        wf = pd.DataFrame(response.json()["list"])  # collecting the data
        wf.drop(["dt", "clouds", "wind", "visibility", "pop", "sys"], axis=1, inplac
        for i in range(wf.shape[0]):     # each element in the column weather is in
            wf["weather"][i] = wf["weather"][i][0]
        wf = pd.concat([wf, pd.json_normalize(wf.main), pd.json_normalize(wf.weather
        wf.drop(["main", "weather", "feels_like", "pressure", "sea_level", "grnd_lev
        wf["city"] = city  # generate a column with the city name
        if 'rain' not in wf.columns:  #some cities have a column rain and some have
            wf["rain"] = pd.Series(dtype='object')
        wf["rain"] = wf.apply(lambda x: "0" if x["rain"] is np.nan else "1", axis=1)
        wf = wf.sort_index(axis=1)
        wf.columns = ["city", "clouds", "date_time", "rain", "temperature"] # set ne
        wf = wf.sort_values(by="date_time", ascending=True) # sort the data in respe
        storage = pd.concat([storage, wf]).reset_index(drop=True) # we reset the ind
    return storage



weatherdata = weather(citylist)
```

In [33]:

```
# optionally, we save the data in a .csv file
# weatherdata.to_csv('weatherdata.csv', index=False)
```

In [35]:

```
# campare with the row above. We read the data from the genereted .csv
weatherdata = pd.read_csv('weatherdata.csv')
```

In [27]:

```
# we connect to the database and insert our data in the table "weather"
weatherdata.to_sql("weather", con=con, index=False, if_exists='append') # we use "in
```

Table airports

In order to get the the arriving flights for a certain airport, we need its ICAO id. We will store the ids in the table "airports". This enables us to convert a airports name in the ICAO necessary for our querries. source: https://de.wikipedia.org/wiki/Liste_von_ICAO-Codes_in_Deutschland_mit_Flugplatzangaben

In [19]:

```
airportdata = [
    ('EDSB', 'Karlsruhe/Baden-Baden', 'Baden-Baden'),
    ('EDDB', 'Berlin Brandenburg', 'Berlin'),
    ("EDDT", "Berlin-Tegel","Berlin"),
    ("EDDK", "Köln/Bonn", "Bonn"),
    ("EDDW", "Bremen", "Bremen"),
    ("EDDC", "Dresden", "Dresden"),
    ("EDLW", "Dortmund", "Dortmund"),
    ("EDDL", "Düsseldorf", "Düsseldorf"),
    ("EDLE", "Verkehrslandeplatz Essen/Mülheim", "Essen"),
    ("EDDF", "Frankfurt am Main", "Frankfurt am Main"),
    ("EDDH", "Hamburg", "Hamburg"),
    ("EDDV", "Hannover-Langenhagen", "Hanover"),
    ("EDDP", "Leipzig/Halle", "Leipzig"),
    ("EDDM", "München", "Munich"),
    ("EDDG", "Münster/Osnabrück", "Münster"),
    ("EDDN", "Nürnberg", "Nuremberg"),
    ("EDDS", "Stuttgart", "Stuttgart")
]

aiports_df=pd.DataFrame(airportdata, columns = ['icao' , 'airport', 'city']) # creat



# we connect to the database and insert our data in the table "airports"
aiports_df.to_sql("airports", con=con, index=False, if_exists='append')
```

Table arrivals

In [20]:

```
# required packages
import sqlalchemy
import requests
import pandas as pd
from pandas.io.json import json_normalize
import json
import numpy as np
import datetime
from datetime import date
```

```
def arrivals(citylist): # as input, we use a list of cities

    def city2iaco(citylist): # the API needs the ICAO code of the airport and not it
        newcitylist = []
        for city in citylist:
            cnx = connect() # this function is defined at the beginning of this note
            cursor = cnx.cursor()
            cursor.execute(f"USE {dbname}")   # again we use SQL
            cursor.execute(f"SELECT icao FROM airports WHERE city = '{city}'")
            cityname = cursor.fetchone()[0]  # the save the result of the query
            newcitylist.append(cityname)
        return(newcitylist)
    airportlist = city2iaco(citylist) # now we proceed with the list of ICAOs

    storage = pd.DataFrame()   # here we store the data from each airport
    # key = "7a4bc5ce0bmshd49770f6283961fp1c45a6jsn405d55120cc4"  # Marvin's key (no
    # key = "515339d6fmsh81d78dce0cb28cap1cbb53jsnb31e102cdb8c"   # Balus's key (not
    key = "3cd15bf266msh2331a2a034ea490p1c96a2jsn828e9dce3a50"    # Joachims's key
    tomorrow = (date.today() + datetime.timedelta(days=1)).strftime("%Y-%m-%d") # we

    for airport in airportlist:
        url = f"https://aerodatabox.p.rapidapi.com/flights/airports/icao/{airport}/{
        querystring = {"withLeg":"true","direction":"Arrival","withLocation":"true"}
        headers = {
        "X-RapidAPI-Host": "aerodatabox.p.rapidapi.com",
        "X-RapidAPI-Key": key}

        response = requests.request("GET", url, headers=headers, params=querystring)
        arrivaldata = response.json()["arrivals"]  # storing the data
        arrivaldata = pd.DataFrame(pd.json_normalize(arrivaldata))
        arrivaldata["arrival.airport.icao"] = airport  # generate a column with the
        arrivaldata.drop(["codeshareStatus", "isCargo", "departure.quality", "arriva
        arrivaldata.columns = ["flightnumber", "status", "departure_airport_icao", "
        # convert date-time columns in the right format:
        arrivaldata['arrival_scheduledTimeLocal'] = pd.to_datetime(arrivaldata['arri
        arrivaldata['arrival_actualTimeLocal'] = pd.to_datetime(arrivaldata['arrival
        arrivaldata['arrival_scheduledTimeUtc'] = pd.to_datetime(arrivaldata['arriva
        arrivaldata['arrival_actualTimeUtc'] = pd.to_datetime(arrivaldata['arrival_a
        storage = pd.concat([storage, arrivaldata]).reset_index(drop=True)

    return storage



arrivaldata = arrivals(["Frankfurt am Main", "Munich"])
```

In [9]:

```
# optionally, we save the data in a .csv file
# arrivaldata.to_csv('arrivaldata.csv', index=False)
```

In [72]:

```
# campare with the row above. We read the data from the genereted .csv
arrivaldata = pd.read_csv('arrivaldata.csv')
```

In [73]:

```
# we connect to the database and insert our data in the table "arrivals"
arrivaldata.to_sql("arrivals", con=con, index=False, if_exists='append')
```

Prepare the cloud (AWS): Setting a lambda function and CloudWatch

I created two lambda functions. One for the weather data and a manual use and one second test function with automation.

Lambda function I (collecting weather data)

In [ ]:

```
import pymysql
import mysql.connector
import sqlalchemy
import requests
import pandas as pd
import numpy as np
import json
from pandas.io.json import json_normalize



def lambda_handler(event, context):

    cnx = pymysql.connect(
        user='admin',
        password='0123456789',
        host='vdreiffm.cxmmedv6eu66.eu-central-1.rds.amazonaws.com',
        database='project5')
    cursor = cnx.cursor()
    ##################################################

    dbname = "project5"
    host="vdreiffm.cxmmedv6eu66.eu-central-1.rds.amazonaws.com"
    user="admin"
    password="0123456789"
    port=3306
    con = f'mysql+pymysql://{user}:{password}@{host}:{port}/{dbname}'


    def weather(cities):
        storage = pd.DataFrame()
        API_key = "d3645498e615aef5ea56aba13e895b36"

        for city in cities:
            url = f"http://api.openweathermap.org/data/2.5/forecast?q={city}&appid={
            response = requests.get(url)
            response
            wf = pd.DataFrame(response.json()["list"])
            wf.drop(["dt", "clouds", "wind", "visibility", "pop", "sys"], axis=1, in
            for i in range(wf.shape[0]):     # each element of column weather is in
                wf["weather"][i] = wf["weather"][i][0]
            wf = pd.concat([wf, pd.json_normalize(wf.main), pd.json_normalize(wf.wea
            wf.drop(["main", "weather", "feels_like", "pressure", "sea_level", "grnd
            wf["city"] = city
            if 'rain' not in wf.columns:  #some cities have a column rain and some n
                wf["rain"] = pd.Series(dtype='object')
            wf["rain"] = wf.apply(lambda x: "0" if x["rain"] is np.nan else "1", axi
            wf = wf.sort_index(axis=1)
            wf.columns = ["city", "clouds", "date_time", "rain", "temperature"]
            wf = wf.sort_values(by="date_time", ascending=True)
            storage = pd.concat([storage, wf]).reset_index(drop=True)
        return(storage)


    citylist = ['Baden-Baden', "Berlin"]
    weatherdata = weather(citylist)

    weatherdata.to_sql("weather", con=con, index=False, if_exists='append')
    # commit changes & close connection
    cnx.commit()
    cursor.close()
    cnx.close()


    ##################################################
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

Lambda function II (test function)

Due to the query limitations of the applied APIs, we use a test function for automation. For this, we create a dummy Database and table, we want to fill with dummy data.

In [ ]:

```
dbname = "auto_db"
tbname = "auto_table"
conauto = f'mysql+pymysql://{user}:{password}@{host}:{port}/{dbname}'

cursor.execute(f"CREATE DATABASE IF NOT EXISTS {dbname}")

cursor.execute(f"USE {dbname}")

cursor.execute(
    f"CREATE TABLE IF NOT EXISTS {tbname} ("
    "time_data VARCHAR(255),"
    "data VARCHAR(255) )"
)
```

The lambda function only connects to the database "auto_db" and inserts the actual date-time in the table "auto_table".

In [ ]:

```
import json
import pymysql
import sqlalchemy
import pandas as pd
import datetime
from datetime import datetime


def lambda_handler(event, context):

    # connect to database
    cnx = pymysql.connect(
        user='admin',
        password='0123456789',
        host='vdreiffm.cxmmedv6eu66.eu-central-1.rds.amazonaws.com',
        database='auto_db')

    cursor = cnx.cursor()



    ##################################################

    dbname = "auto_db"
    host="vdreiffm.cxmmedv6eu66.eu-central-1.rds.amazonaws.com"
    user="admin"
    password="0123456789"
    port=3306
    conauto = f'mysql+pymysql://{user}:{password}@{host}:{port}/{dbname}'


    ##################################################
    now = datetime.now().strftime("%Y-%m-%d %H:%M")  # save the actual date and time

    df = pd.DataFrame([{"time_data": now, "data": "some data"}])  # generate a data
    df.to_sql("auto_table", con=conauto, index=False, if_exists='append')   # the da
    ##################################################



    # commit changes & close connection
    cnx.commit()
    cursor.close()
    cnx.close()


    return {
        'statusCode': 200,
        'body': json.dumps('Läuft bei dir!')
    }
```

Automation

Now we want that the lambda function is called once an hour. We achieved this by creating an CloudWatch Event and connecting it with the lambda function. Typing

In [ ]:

```
USE auto_db;
SELECT * FROM auto_table;
```

in the MySQL Workbench shows that everythin is working.