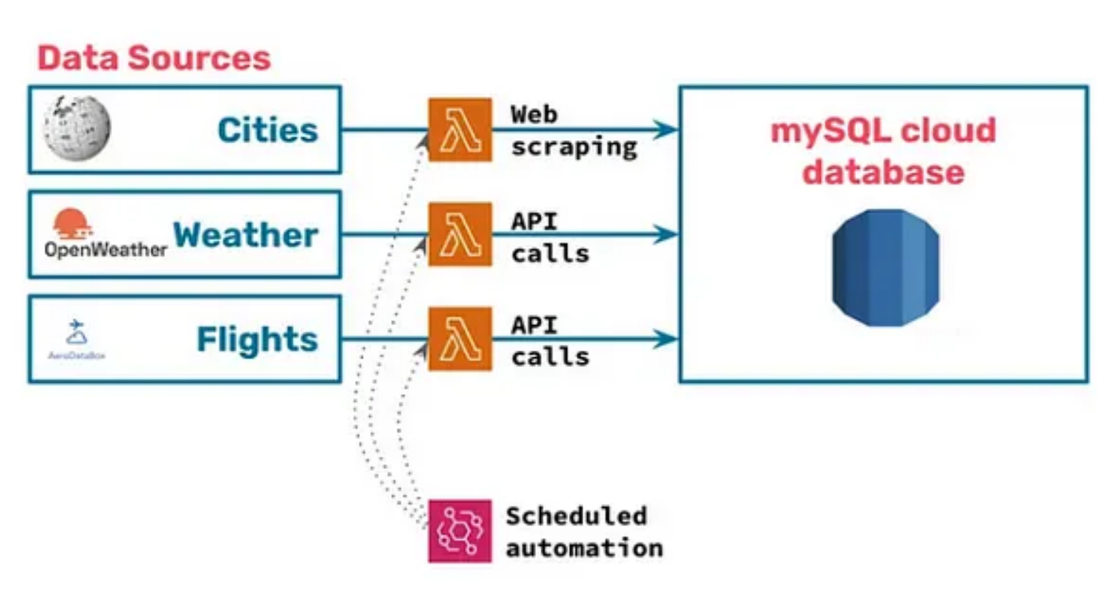
My Journey into Building Scalable Data Pipeline

Haikel Zanina
4 min read · Just now



Introduction
It all started with a fictional company called Gans. My instructor asked me to help them collect data from external sources to predict e-scooter movements. Sounds simple, right? But little did I know, this task would help me build my first end-to-end data pipeline using web scraping, APIs, and databases, and eventually automate the whole process.

In this article, I'll share how data flows from the internet to my database every day, using Python and various libraries.

. . .

Web Scraping: Extracting Data from Wikipedia

The first step was to collect demographic data for the largest German cities. Since Wikipedia doesn't provide an API for this, I turned to **BeautifulSoup**, a Python library for pulling data out of HTML and XML files.

Scraping Wikipedia:

To get the list of German cities by population, I used the following code snippet:

```
import requests
from bs4 import BeautifulSoup

url = ("https://en.wikipedia.org/wiki/List_of_German_cities_by_population_within
response = requests.get(url)
soup = BeautifulSoup(response.content, "html.parser")
```

This pulls the Wikipedia page and parses the HTML using **BeautifulSoup**. The next step was to extract the relevant information from the table:

```
parent = soup.find(class_ = 'wikitable sortable')
children = parent.contents[1]
dictionary = {"City": [], "Country": [], "Population": []}

for i, child in enumerate(children):
    if((i != 0) & (i % 2 == 0)):
        dictionary['City'].append(child.contents[3].get_text("|", strip=True).re
        dictionary['Population'].append(child.contents[7].get_text("|", strip=Tr
        dictionary['Country'].append(child.contents[5].get_text("|", strip=True)
    if(i == 40):
        break
```

Here, I manually parsed the table content, cleaning the data as I went. In practice, scraping can be messy, especially with complex HTML structures like Wikipedia's.

. . .

Collecting Data Using APIs

Next, Gans asked for weather and flight information for major cities. I used **OpenWeatherMap** and **AeroDataBox** APIs for this task. The weather data required getting a forecast for the next day, and flight data was collected based on airport arrivals.

Weather Data from OpenWeatherMap:

After signing up on OpenWeatherMap, I used the following API call to fetch the weather forecast:

```
url = f"http://api.openweathermap.org/data/2.5/forecast?q={city_name},{country_c
response = requests.get(url)
json_data = response.json()
```

I then iterated through the response and extracted the necessary information, while handling possible missing data:

```
try:
    dictionary['Snow'].append(json_data['list'][0]['snow']['3h'])
except KeyError:
    dictionary['Snow'].append('0')
```

Flight Data from AeroDataBox:

Similarly, to get flight arrivals, I used the **AeroDataBox** API. This API provided live flight data, which I then processed into the format needed:

```
url = f"https://aerodatabox.p.rapidapi.com/airports/{icao_code}/flights/arrivals
response = requests.get(url, headers=headers)
json_data = response.json()
```

As with the weather data, I had to handle missing or incomplete information and extract only the relevant fields.

. . .

Storing Data in MySQL

Once I had the data, it was time to store it in a **MySQL** database for further analysis. I used **SQLAlchemy**, a Python library that simplifies database interaction.

Database Setup and Connection:

First, I set up a MySQL database with the following schema:

```
CREATE DATABASE gans;
```

Next, I connected to the database using **SQLAlchemy**:

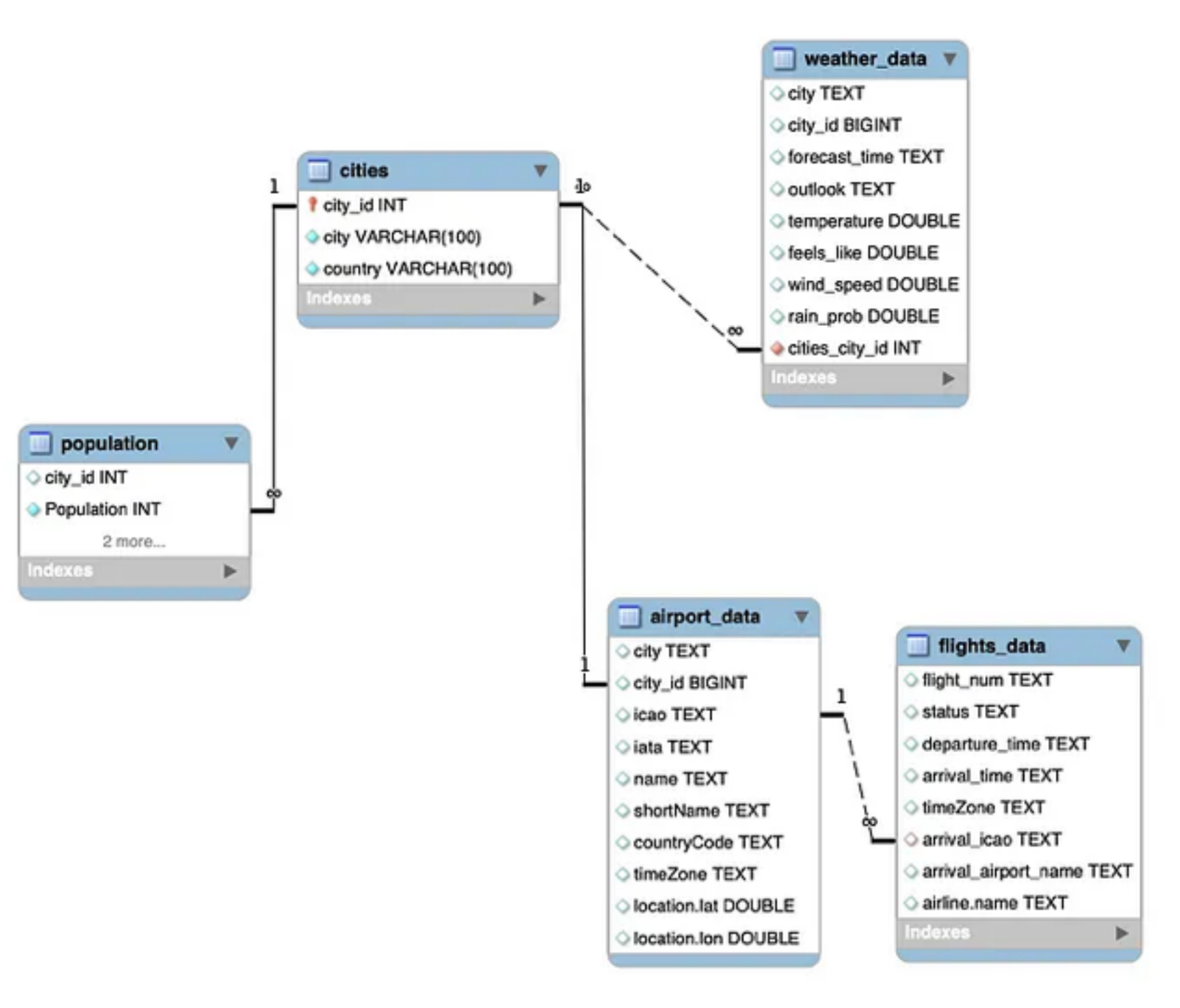
```
from sqlalchemy import create_engine

# Database connection string
engine = create_engine(f'mysql+pymysql://{user}:{password}@{host}:{port}/{databa
df.to_sql('weather', con=engine, if_exists='append', index=False)
```

This code uses **SQLAlchemy** to directly insert the cleaned data into the database. The `df.to_sql()` function stores the data in the **weather** table.

ERR Diagram

Once you know what kind of data you want to store you need to think about how your tables are going to look like and how are you going to connect them.



. . .

Automating the Data Pipeline

To ensure that data is collected and stored daily, I set up a **cron job** to run the Python script every day. Since I didn't use Apache Airflow for orchestration, I leveraged AWS Lambda to schedule the job.

. . .

Challenges and Solutions

Throughout the process, I faced several challenges:

- **Handling messy data**: Scraping HTML data often involved dealing with inconsistent structures, especially from Wikipedia.
- **Missing API data**: Some data wasn't available, so I used `try-except` blocks to prevent errors during processing.
- **Scheduling**: Automating the pipeline involved using AWS Lambda and cron jobs to ensure the script runs daily without manual intervention.

. . .

Conclusion

Building this data pipeline was an exciting journey. From scraping data off the web and making API calls to transforming and storing it in a MySQL database, I learned a lot about the practical applications of data engineering. Now, every day, my code pulls fresh data from the internet, processes it, and loads it into a database automatically, without any manual help.

> *Feel free to leave comments, share your experiences, or check out my GitHub repository for more detailed code examples!*

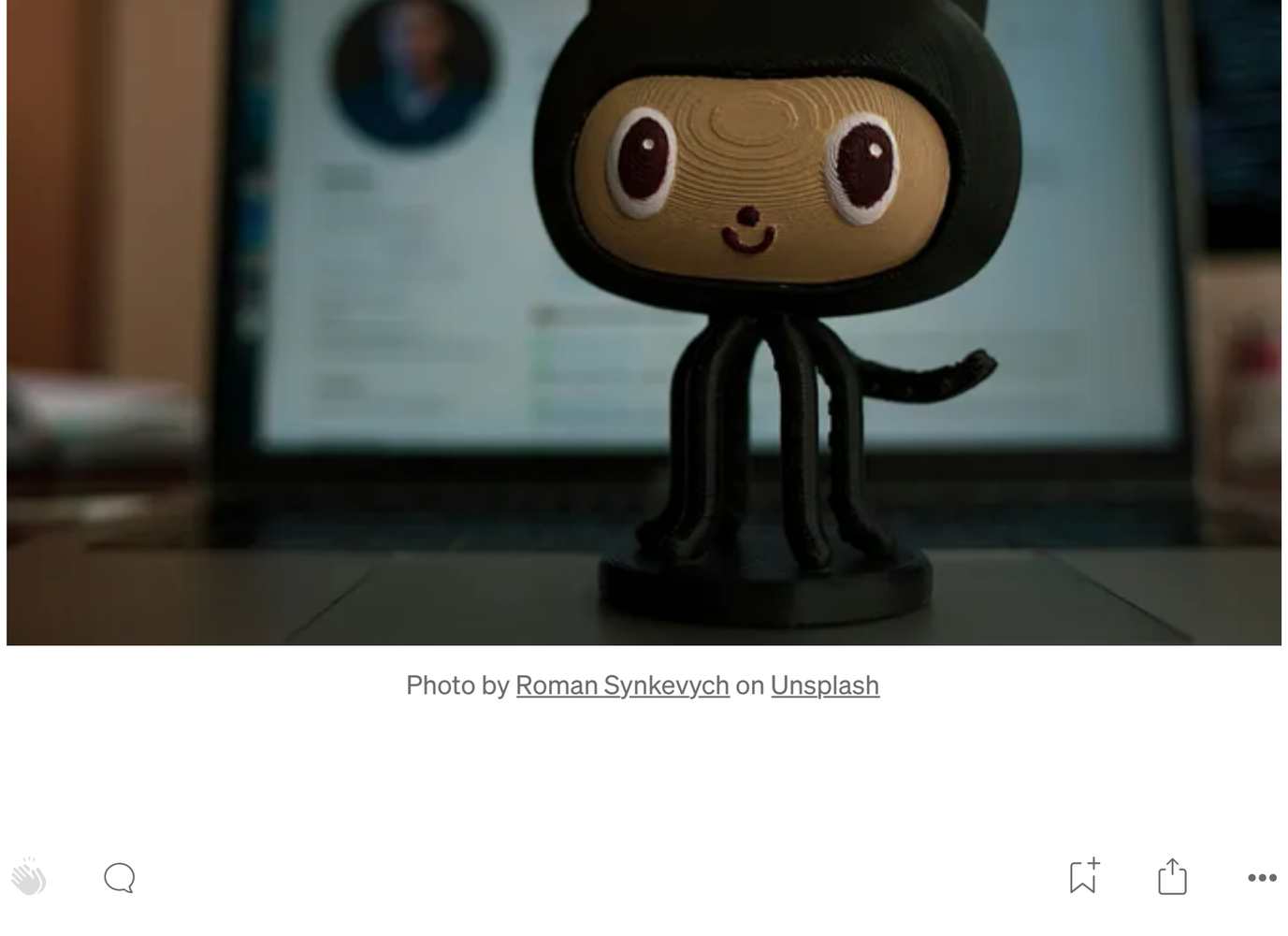


Photo by Roman Synkevych on Unsplash

Written by Haikel Zanina
0 Followers · 1 Following