How to Build a Data Pipeline for API Integration Using Python and PostgreSQL

A hands-on approach to fetching, storing, and analyzing data from APIs

Satyam Sahu

Python in Plain English

Nov 18, 2024

If you’ve ever worked with **data integration**, you know how crucial it is to have a streamlined process for fetching, storing, and analyzing data from **APIs**. Whether you’re dealing with real-time data from social media, stock markets, IOT sensors, or weather forecasting APIs, building a robust data pipeline can save you hours of manual effort.

In this hands-on guide, I'll walk you through building a complete **data pipeline** using **Python** and **PostgreSQL**. By the end of this guide, you'll know how to:

1. Fetch data from APIs using Python's requests library.
2. Transform the data for efficient storage.
3. Load it into a PostgreSQL database for further analysis.

Let's dive right in and build a data pipeline that can automate your data integration tasks!

Step 1: Setting Up Your Environment

Before getting started, ensure you have **installed Python and PostgreSQL**. You'll also need these Python libraries:

```
pip install requests psycopg2 pandas
```

- **requests**: For making HTTP requests to fetch API data.
- **psycopg2**: To interact with your PostgreSQL database. (PostgreSQL adapter for the Python programming language)
- **pandas**: For data manipulation and transformation.

Setting Up PostgreSQL Database

First, create a new database where you'll store the API data. Open your PostgreSQL terminal and run:

```
CREATE DATABASE api_data;
```

This database will store our weather data fetched from an API.

Step 2: Fetching Data from an API

Fetching data from an API is the first step. We'll use Python's requests library to get data from a weather API.

Python Code to Fetch Data

```
import requests

# Define the API endpoint
api_url = "https://api.open-meteo.com/v1/forecast"
params = {
  "latitude": 35.6895,
  "longitude": 139.6917,
  "hourly": "temperature_2m"
}
```

```
# Make a GET request to fetch data
response = requests.get(api_url, params=params)

# Check if the request was successful
if response.status_code == 200:
   data = response.json()
   print("Data fetched successfully!")
else:
   print("Failed to fetch data:", response.status_code)
```

Output:

```
Data fetched successfully!
```

Pro Tip: Always check the response status code. If you don't handle potential errors, your pipeline might break unexpectedly.

Step 3: Transforming Data for Storage

APIs often return data in **JSON** format, which needs to be cleaned and structured before storing it in a database. We'll use **Pandas** for this step.

Transforming JSON Data into a DataFrame

```
import pandas as pd

# Extract the relevant data from the JSON response
temperature_data = data['hourly']['temperature_2m']
timestamps = data['hourly']['time']

# Create a DataFrame
df = pd.DataFrame({'timestamp': timestamps, 'temperature': temperature_data})

# Convert timestamp to datetime format
df['timestamp'] = pd.to_datetime(df['timestamp'])
print(df.head())
```

Output:

```
timestamp              temperature
0 2024-10-18 00:00:00       15.2
1 2024-10-18 01:00:00       14.8
2 2024-10-18 02:00:00       14.4
3 2024-10-18 03:00:00       13.9
4 2024-10-18 04:00:00       13.5
```

Step 4: Creating a Table in PostgreSQL

Now, we need to set up a table in PostgreSQL to store the transformed data.

```
CREATE TABLE weather_data (
  id SERIAL PRIMARY KEY,
  timestamp TIMESTAMP,
  temperature FLOAT
);
```

Step 5: Loading Data into PostgreSQL

With our data cleaned and ready, it's time to load it into the PostgreSQL database.

Python Code to Insert Data

```
import psycopg2

# Connect to PostgreSQL
conn = psycopg2.connect(
  dbname="api_data", user="your_user", password="your_password", host="localhost"
)
cur = conn.cursor()

# Insert DataFrame into PostgreSQL
for _, row in df.iterrows():
  cur.execute(
    "INSERT INTO weather_data (timestamp, temperature) VALUES (%s, %s)",
    (row['timestamp'], row['temperature'])
  )

# Commit changes and close connection
conn.commit()
cur.close()
conn.close()
print("Data inserted into PostgreSQL successfully!")
```

Output:

```
Data inserted into PostgreSQL successfully!
```

Pro Tip: Always commit your changes using conn.commit() to ensure your data is saved in the database.

Step 6: Automating the Data Pipeline

Automating your data pipeline ensures that you can fetch, transform, and store data on a schedule without manual intervention. This is especially useful for regularly updated APIs like weather data or stock prices.

Why Automate?

- **Consistency**: Ensures data is fetched at regular intervals.
- **Scalability**: Saves time by automating repetitive tasks.
- **Reliability**: Reduces the chance of human error.

Setting Up Automation Using Python and Cron (Linux/Mac)

Create a Python script, data_pipeline.pythat includes all the steps above. Then, automate its execution using **cron**.

1. **Open the crontab editor**:

```
crontab -e
```

2. Schedule your script to run every day at midnight:

```
0 0 * * * /usr/bin/python3 /path/to/data_pipeline.py
```

3. Save and exit.

Automating with Task Scheduler (Windows)

If you're on Windows, use **Task Scheduler**:

1. Open **Task Scheduler**.
2. Create a new task and set a trigger for the desired schedule.
3. Set the action to run your Python script:

- **Program**: python
- **Arguments**: C:\path\to\data_pipeline.py

4. Save the task, and your pipeline is now automated!

Pro Tip: Test your automated pipeline to ensure it's working correctly. You can log errors using Python's logging module:

```
import logging
logging.basicConfig(filename='pipeline.log', level=logging.ERROR)
```

Example Output (Log File)

```
2024-10-18 00:00:01 - Data fetched successfully!
2024-10-18 00:00:03 - Data inserted into PostgreSQL successfully!
```

Common Mistake: Forgetting to set environment variables (like database credentials) in your automation script can lead to unexpected failures. Always test your automated job manually before scheduling it.

Step 7: Querying Data for Analysis

Now that your pipeline is automated, you can query the data whenever you need insights.

Python Code to Fetch Data for Analysis

```
conn = psycopg2.connect(
   dbname="api_data", user="your_user", password="your_password", host="localhost"
)
cur = conn.cursor()

cur.execute("SELECT * FROM weather_data WHERE timestamp > NOW() - INTERVAL '1 day'")
rows = cur.fetchall()

for row in rows:
   print(row)

cur.close()
conn.close()
```

Conclusion: Building an Efficient Data Pipeline

Creating a data pipeline is essential for automating data integration tasks. With this guide, you've learned how to fetch, transform, and store API data using Python and PostgreSQL. Automating the pipeline ensures that your data is always up-to-date, saving time and reducing manual errors.

Whether you're a data analyst, data engineer, or someone interested in automation, this guide provides the foundation you need to streamline your workflows. Now go ahead and apply these techniques to your projects!

Link to this page : [How to Build a Data Pipeline for API Integration Using Python and PostgreSQL | by Satyam Sahu | Python in Plain English]