INSPARI
a valantic company

# A Data Engineers guide to APIs

## Agenda

1\.

What are APIs? What is their role in a Data Platform or Data Automation flow?

2\.

APIs in 2025 - Web Apps, REST and GraphQL

3\.

API basic principles “ELI10”. Common use cases and solutions to common problems. Sync vs. Async.

4\.

Demo of API examples repo with “code snippets” and example usages

5\.

MS Fabric GraphQL

The formalities…

## API - Application Programming Interface

- Messengers for different software systems to communicate with each other
- Allows different programming languages to communicate
- Backbone of interactions on the internet. All Web pages and Web applications are built with them.
- Structured way of serving data (Response) based on a specified input (Request)
- Abstract the underlying complexity of systems, simplify and “containerize” workflows (Micro-service architecture)
- High speed transfer of small data packages often with built in security

## A different type of data ingestion

Sometimes we can’t get access to the underlying Databases and must interact with data via APIs. (External sources)

Or we have ingested data from the usual suspects and need to enhance the data with additional values

(e.g. CVR, geo, weather, financial, SoMe, demographic, traffic etc.)

## A different type of data serving

Sometimes data needs to be accessed by other methods than files, reports and dashboards. Other IT systems might need to retrieve analytical data, so it might be necessary to serve it with an API (Azure function, Fabric GraphQL)

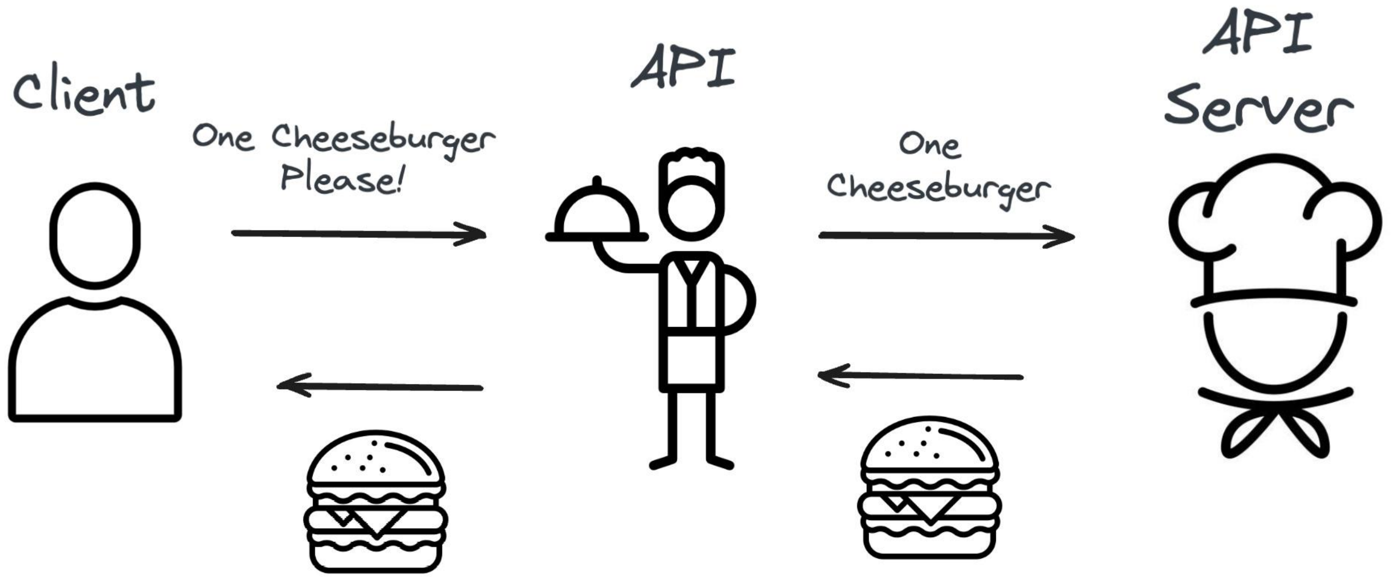
## A crucial part of Agentic Automation

Power Automate, Make, n8n and others all need to communicate via API requests to utilize external tools. Every service on the web you can dream of has an API endpoint you can interact with.

## Restaurant analogy

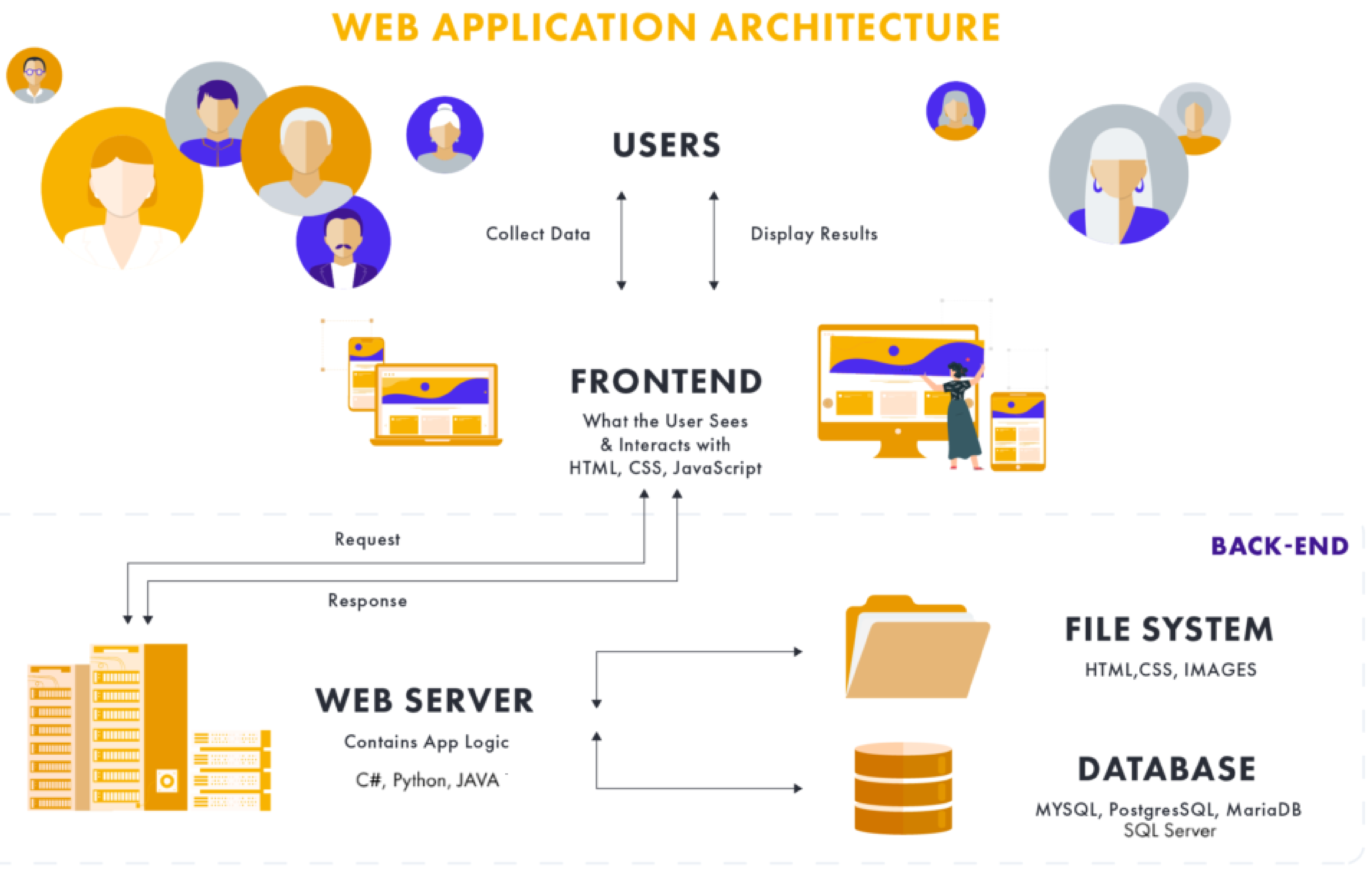
Calling APIs is like ordering specific dishes from a restaurant, you specify what you want (request) and a "server" returns a finished result (response) created by the kitchen (back-end).

Our usual loading of data from databases is like ordering ingredients in bulk from a distributor (ingestion), cooking a dish (transformation) and serving it (presentation).

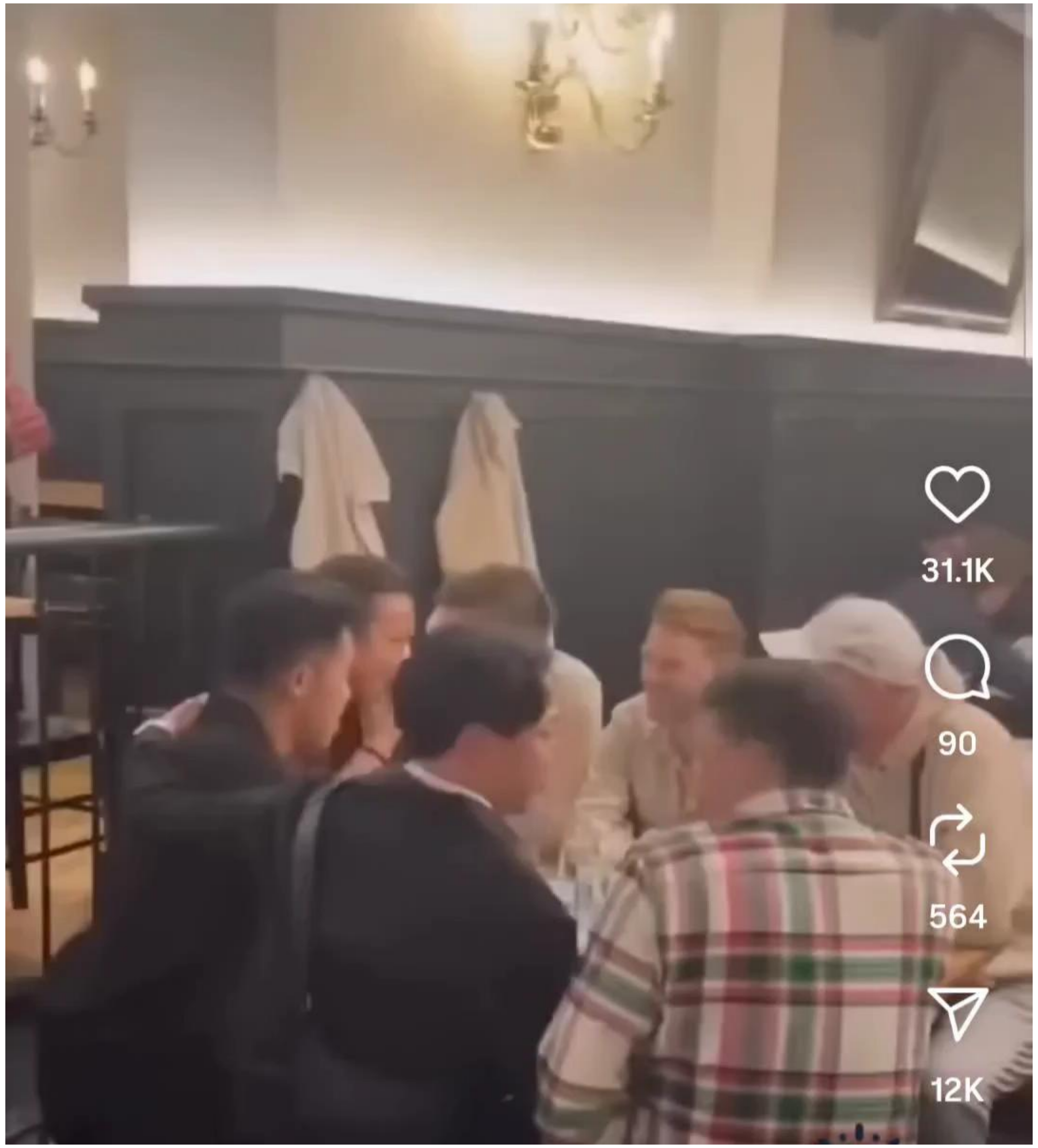


API documentation -> Menu!

- WHAT ARE API'S?



- WHAT ARE API'S?

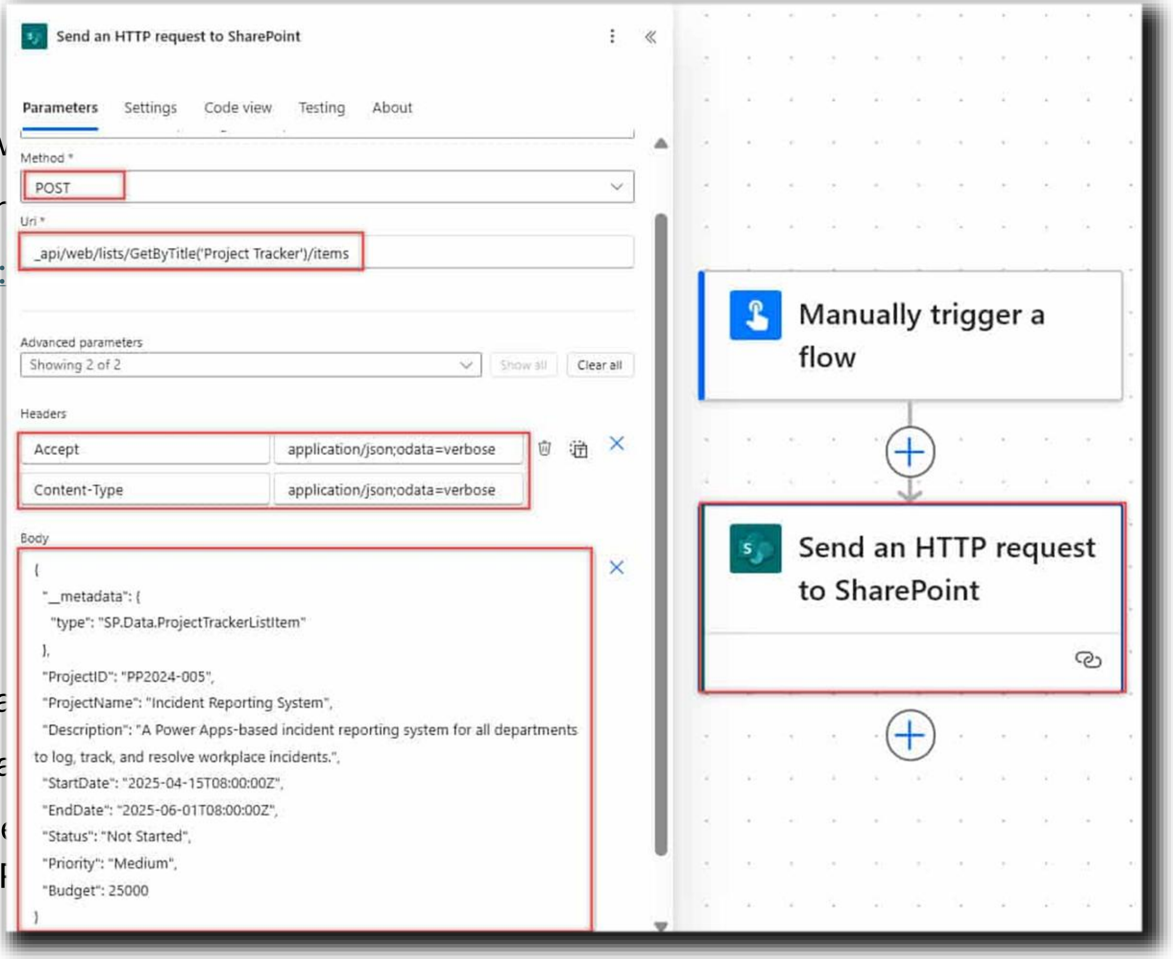


## Components of a request

- URL - The address that specifies where the API resource is located on the w
- Method - The action to be performed (like GET, POST, (PUT), (DELETE)) on th
- Query Parameters (optional) – Parameter to specify what data to get (https:
- Header - Contains metadata about the request/response, such as content cookie in the browser).
- Body (optional) - The data sent to the API, usually in JSON or XML format.

## Components of a response

- Header - Metadata about the response, like content type, caching rules, or a
- Body - The data received from the API, usually in JSON or XML format (but ca
- Status Code - A number indicating the result of the request (e.g., 200 - succe server error). Rule of thumb: 2xx (great!), 4xx (you made an error), 5xx (the AI



Method *: POST

Uri *: _api/web/lists/GetByTitle('Project Tracker')/items

Headers

| | |
|---|---|
| Accept | application/json;odata=verbose |
| Content-Type | application/json;odata=verbose |

Body

```
{
 "__metadata": {
  "type": "SP.Data.ProjectTrackerListItem"
 },
 "ProjectID": "PP2024-005",
 "ProjectName": "Incident Reporting System",
 "Description": "A Power Apps-based incident reporting system for all departments
to log, track, and resolve workplace incidents.",
 "StartDate": "2025-04-15T08:00:00Z",
 "EndDate": "2025-06-01T08:00:00Z",
 "Status": "Not Started",
 "Priority": "Medium",
 "Budget": 25000
}
```

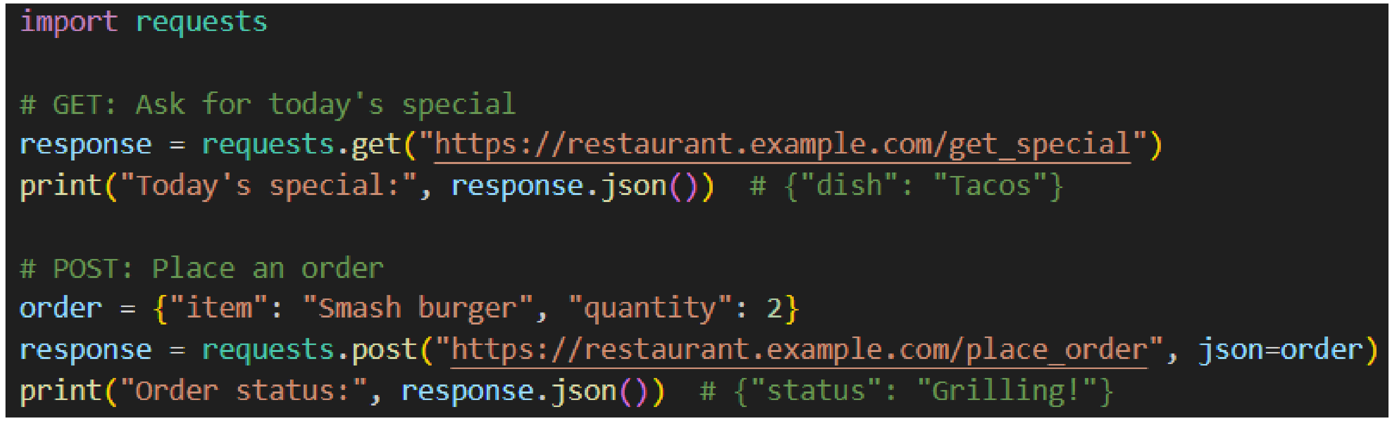
## HTTP (Hypertext Transfer Protocol)

Primary mode of communication for the World Wide Web.

•**Endpoint-Centric**: URLs mapped directly to server-side actions (e.g., {baseURL}/getDish?id=123)

•**Verb-Based methods**: GET (fetch, retrieve data) and POST (submit data)

•**Statelessness**: Each request contains all necessary info; no server-side session storage.



```python
import requests

# GET: Ask for today's special
response = requests.get("https://restaurant.example.com/get_special")
print("Today's special:", response.json())  # {"dish": "Tacos"}

# POST: Place an order
order = {"item": "Smash burger", "quantity": 2}
response = requests.post("https://restaurant.example.com/place_order", json=order)
print("Order status:", response.json())  # {"status": "Grilling!"}
```

## HTTP (Hypertext Transfer Protocol)

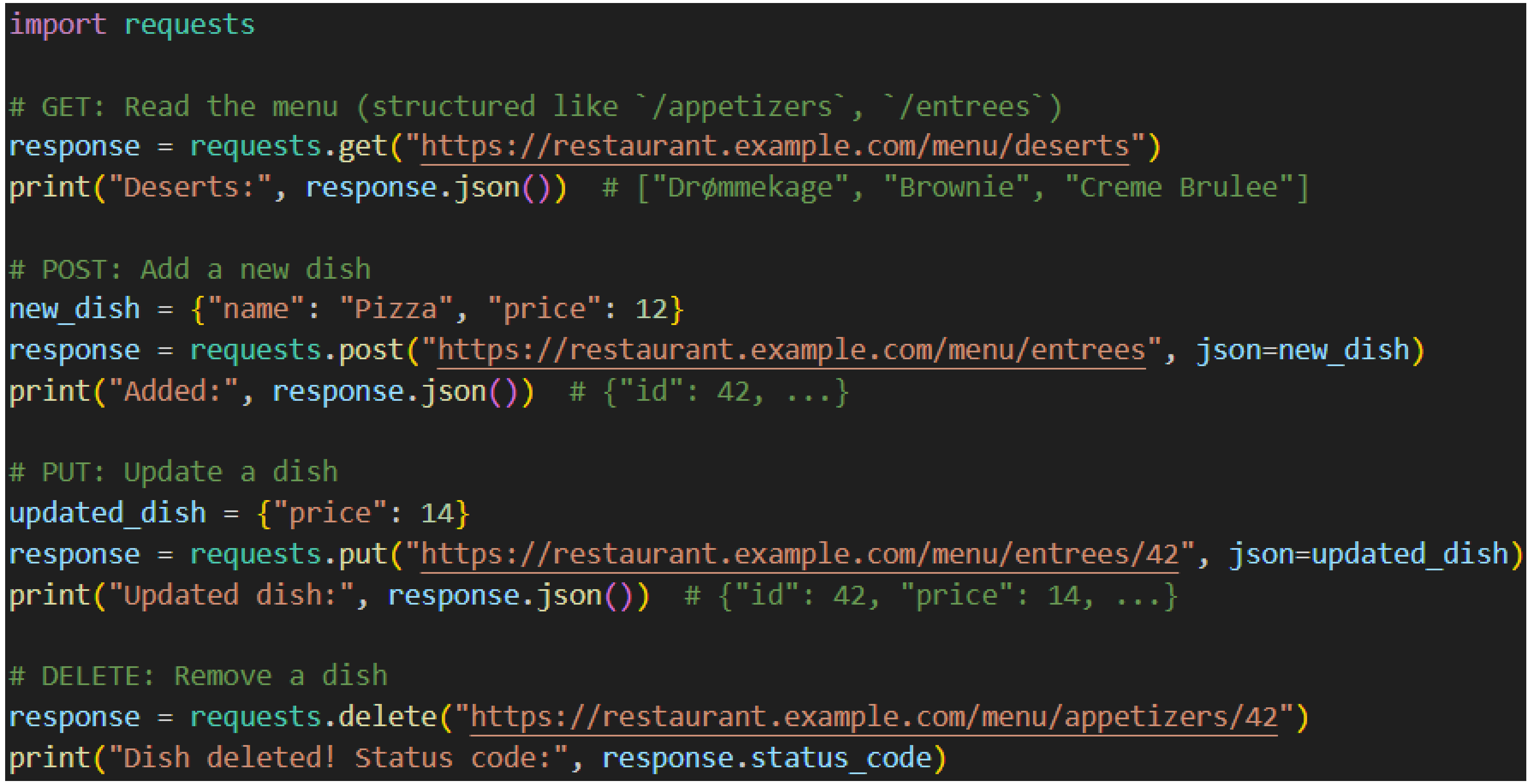
Initial setup and debugging is great in Postman – **DEMO**

## REST (Representational State Transfer)

Modern architectural principle for creating APIs

- **Resource-Oriented**: URLs represent resources *not actions* (e.g. {baseURL}/dishes/123
- **HTTP Verbs as actions:** GET (read), POST (create), PUT (update), DELETE (remove)
- **Stateless & Cacheable**: No client context stored on server; responses are cached.
- **Standardized Interface**: Usually based on JSON.

## REST (Representational State Transfer)



```python
import requests

# GET: Read the menu (structured like `/appetizers`, `/entrees`)
response = requests.get("https://restaurant.example.com/menu/deserts")
print("Deserts:", response.json())  # ["Drømmekage", "Brownie", "Creme Brulee"]

# POST: Add a new dish
new_dish = {"name": "Pizza", "price": 12}
response = requests.post("https://restaurant.example.com/menu/entrees", json=new_dish)
print("Added:", response.json())  # {"id": 42, ...}

# PUT: Update a dish
updated_dish = {"price": 14}
response = requests.put("https://restaurant.example.com/menu/entrees/42", json=updated_dish)
print("Updated dish:", response.json())  # {"id": 42, "price": 14, ...}

# DELETE: Remove a dish
response = requests.delete("https://restaurant.example.com/menu/appetizers/42")
print("Dish deleted! Status code:", response.status_code)
```

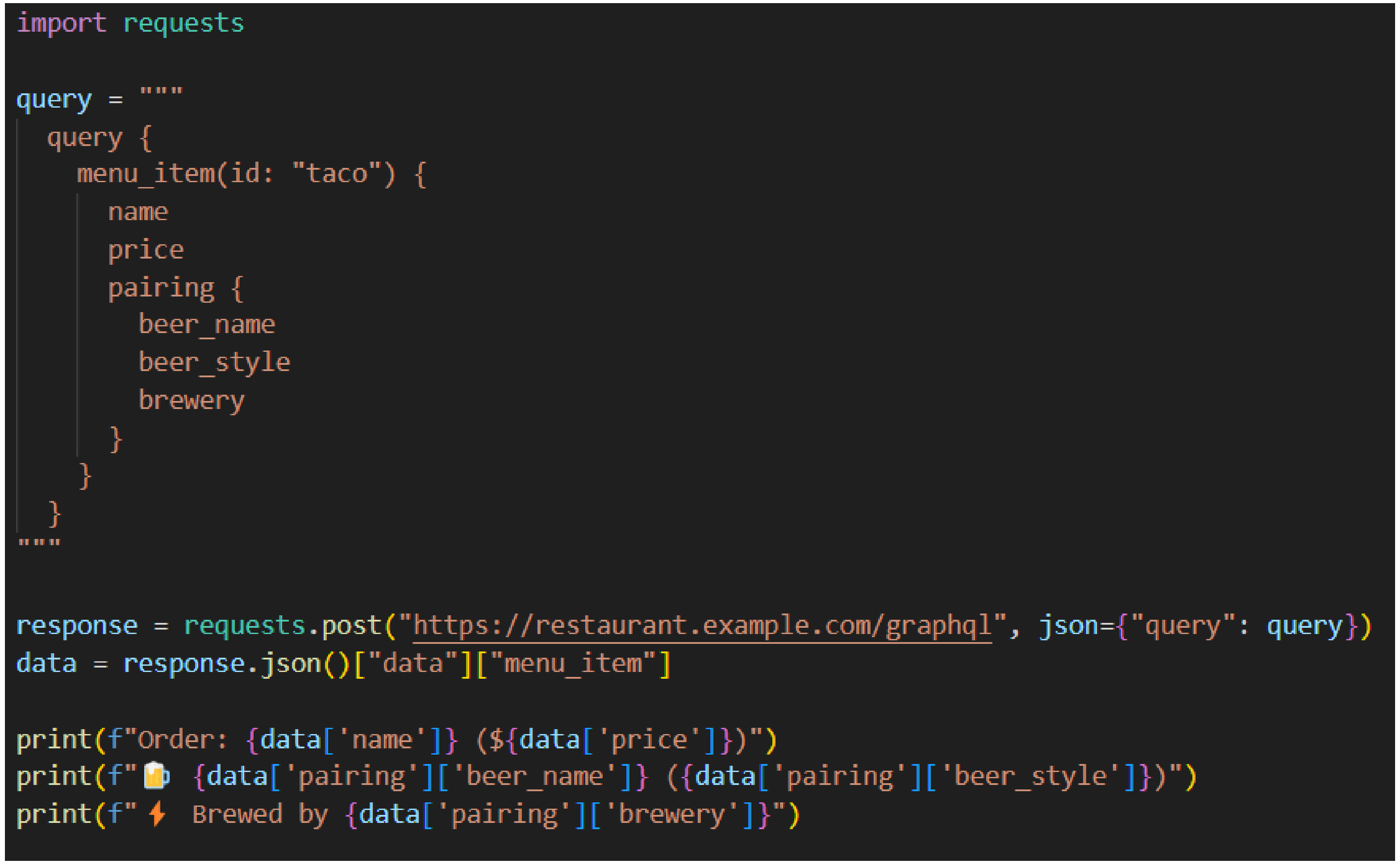
## GraphQL (Graph Query Language)

REST APIs have endpoints for every resource, with static and structured communication with a DB.

GraphQL is flexible, has a single endpoint, and the request body is a query.

- **Query-Driven**: Clients request *exact data needed* (no over/underfetching).
- **Single Endpoint**: POST to a single url (e.g. {baseURL}/graphql).
- **Graph data structure:** Automatically joins data (e.g. Retrieve all dishes and related beer pairings.)

## GraphQL (Graph Query Language)



```python
import requests

query = """
  query {
    menu_item(id: "taco") {
      name
      price
      pairing {
        beer_name
        beer_style
        brewery
      }
    }
  }
"""

response = requests.post("https://restaurant.example.com/graphql", json={"query": query})
data = response.json()["data"]["menu_item"]

print(f"Order: {data['name']} (${data['price']})")
print(f"🍺 {data['pairing']['beer_name']} ({data['pairing']['beer_style']})")
print(f"⚡ Brewed by {data['pairing']['brewery']}")
```

## Sync vs. Async

Which is better?

**Synchronous (sequential)**: You order a dish, the server tells the kitchen to make it, and the same chef then must cook every ingredient in the dish sequentially. She returns the dish to the server before being able to take a new order.

**Asynchronous (concurrent)**:  Multiple chefs work seamlessly, creating a small part of a dish (vegetables, meat, final prep etc.) This enables the possibility of handling multiple concurrent orders.

USER
API
BACKEND

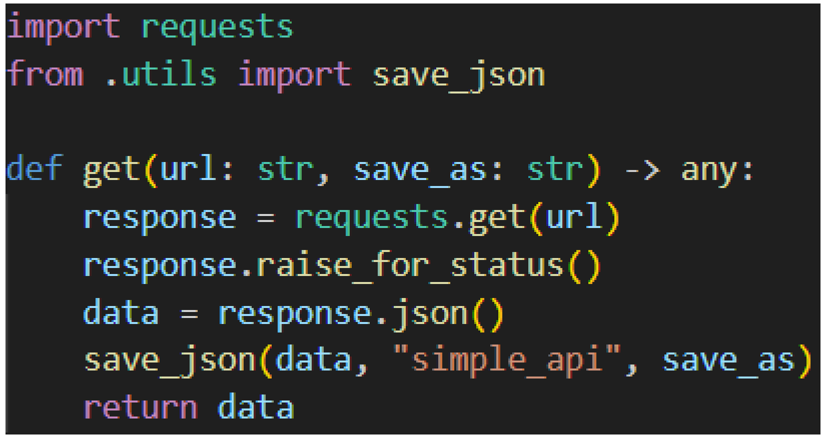
## Sync vs. Async

AIOHTTP VS Requests

Python's "requests" module is good, but… AioHttp is better!

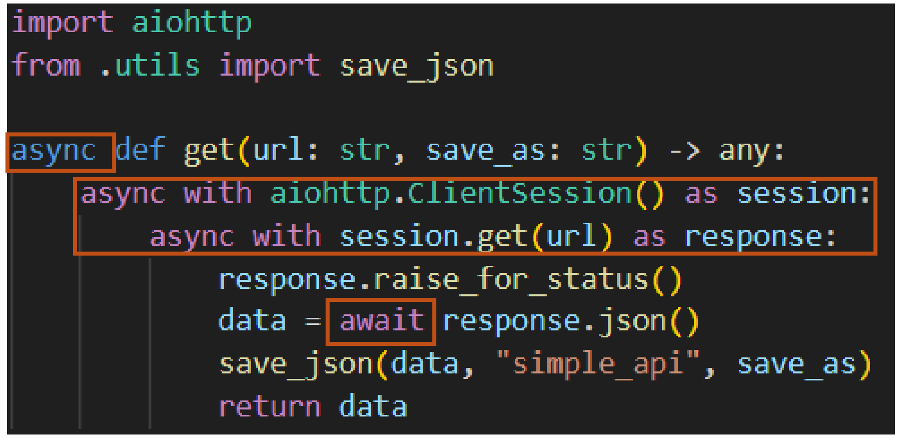
**AIOHTTP**

- Asynchronous http client/server framework built on top of Asyncio (module for async processing in python).
- Due to its non-blocking nature, it can offer superior performance (exponential), especially in I/O-bound and high-concurrency applications.



```python
import requests
from .utils import save_json

def get(url: str, save_as: str) -> any:
    response = requests.get(url)
    response.raise_for_status()
    data = response.json()
    save_json(data, "simple_api", save_as)
    return data
```



```python
import aiohttp
from .utils import save_json

async def get(url: str, save_as: str) -> any:
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            response.raise_for_status()
            data = await response.json()
            save_json(data, "simple_api", save_as)
            return data
```

## Sync vs. Async

**Beware:**

- Async programming syntax is more complicated and needs to be correct to work properly. Consider creating simple unittests
- Mixing sync processes in async code can result in bottlenecks – e.g.  “all chefs need to use the same oven”
- Performance can sometimes be “too good”, resulting in the need for **Rate Limiting**

## Performance demo

## Other common scenarios

**Rate Limiting**

Real: Some APIs have a max requests per minute/hour limit. Some implement request “token currency” to implement this.

Analogy: A kitchen might have a limit on how many orders per hour they can make and therefore implement a seating limit.

**Pagination**

Real: To increase throughput, limit response time, and prevent overloading/overserving data, many APIs implement pagination methods

Analogy: Instead of ordering and delivering all food at once, food is made and served in 3 courses (pages) increasing efficiency

**Error and retry handling**

Real: We can build in error and retry handling to prevent pipelines from failing on common http errors and log important ones

Analogy: If a chef messes up an order, he can retry it before returning it to the server, instead of stopping the entire flow

**Batching**

Real: Passing a list of id’s to retrieve and send the requests in asynchronous batches for higher concurrency

Analogy: Different tables give all their orders to the waiters concurrently, allowing the kitchen to prepare food in batches

**Authentication**

Real: APIs have different authentication methods and tokens, Basic Auth, Oauth 2.0 etc.

Analogy: Not everyone is allowed into the kitchen

## Code snippets demo

## Fabric GraphQL API

- Fabric item that "auto creates" a managed API on specified tables/views, allowing data consumers to query data in Fabric.
- Eliminates the need for REST API development by exposing data via GraphQL endpoints, meaning no extra development time – maximal flexibility and future proof (in case consumer needs other fields/tables etc.)
- Enables **self-service data access** for analysts and engineers.

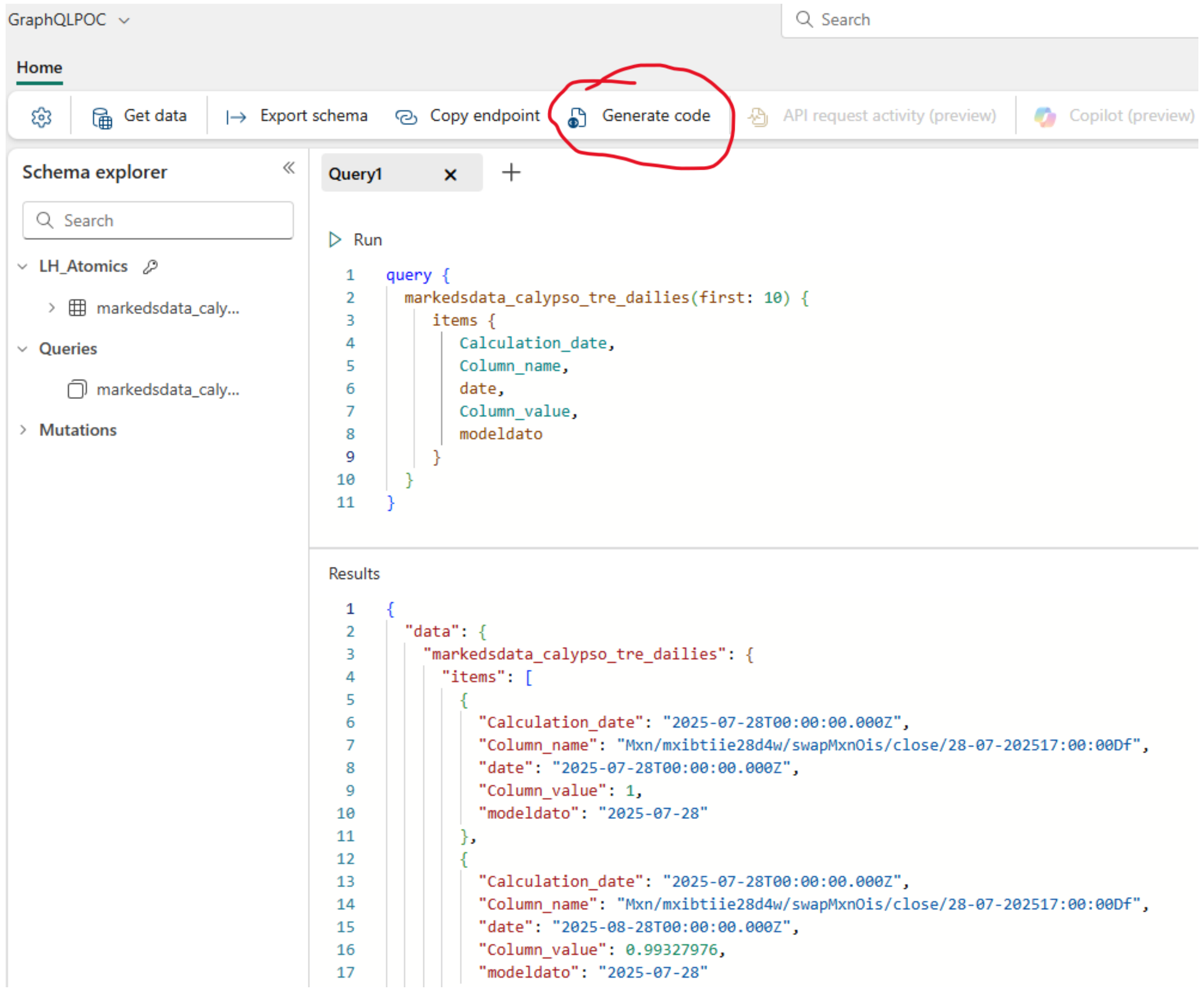
## API for GraphQL in Fabric

**Bridging the gap between data and applications**

| SaaS experience | Fabric data sources | Fabric native |
|---|---|---|
| Quick creation<br>Automated schema<br>Queries, mutations, relationships<br>Intelligent editor | Lakehouse<br>Warehouse<br>Mirrored databases<br>Azure/Fabric SQL | Workspace permissions<br>Lineage, Git integration,<br>Deployment Pipelines,<br>Capacity consumption |

## Fabric GraphQL API

1. Create GraphQL item
2. Choose which tables/views to include
3. Query editor view (right picture) lets us test results of our queries
4. “Generate code” creates a Python code snippet on how to call the API



```
query {
  markedsdata_calypso_tre_dailies(first: 10) {
    items {
      Calculation_date,
      Column_name,
      date,
      Column_value,
      modeldato
    }
  }
}
```

```
{
  "data": {
    "markedsdata_calypso_tre_dailies": {
      "items": [
        {
          "Calculation_date": "2025-07-28T00:00:00.000Z",
          "Column_name": "Mxn/mxibtiie28d4w/swapMxnOis/close/28-07-202517:00:00Df",
          "date": "2025-07-28T00:00:00.000Z",
          "Column_value": 1,
          "modeldato": "2025-07-28"
        },
        {
          "Calculation_date": "2025-07-28T00:00:00.000Z",
          "Column_name": "Mxn/mxibtiie28d4w/swapMxnOis/close/28-07-202517:00:00Df",
          "date": "2025-08-28T00:00:00.000Z",
          "Column_value": 0.99327976,
          "modeldato": "2025-07-28"
```

## Thank you!