Python Online Free Ramzan Course 2025
Taught by: Shaida Muhammad

Working with APIs

Day 17 - Python Basics

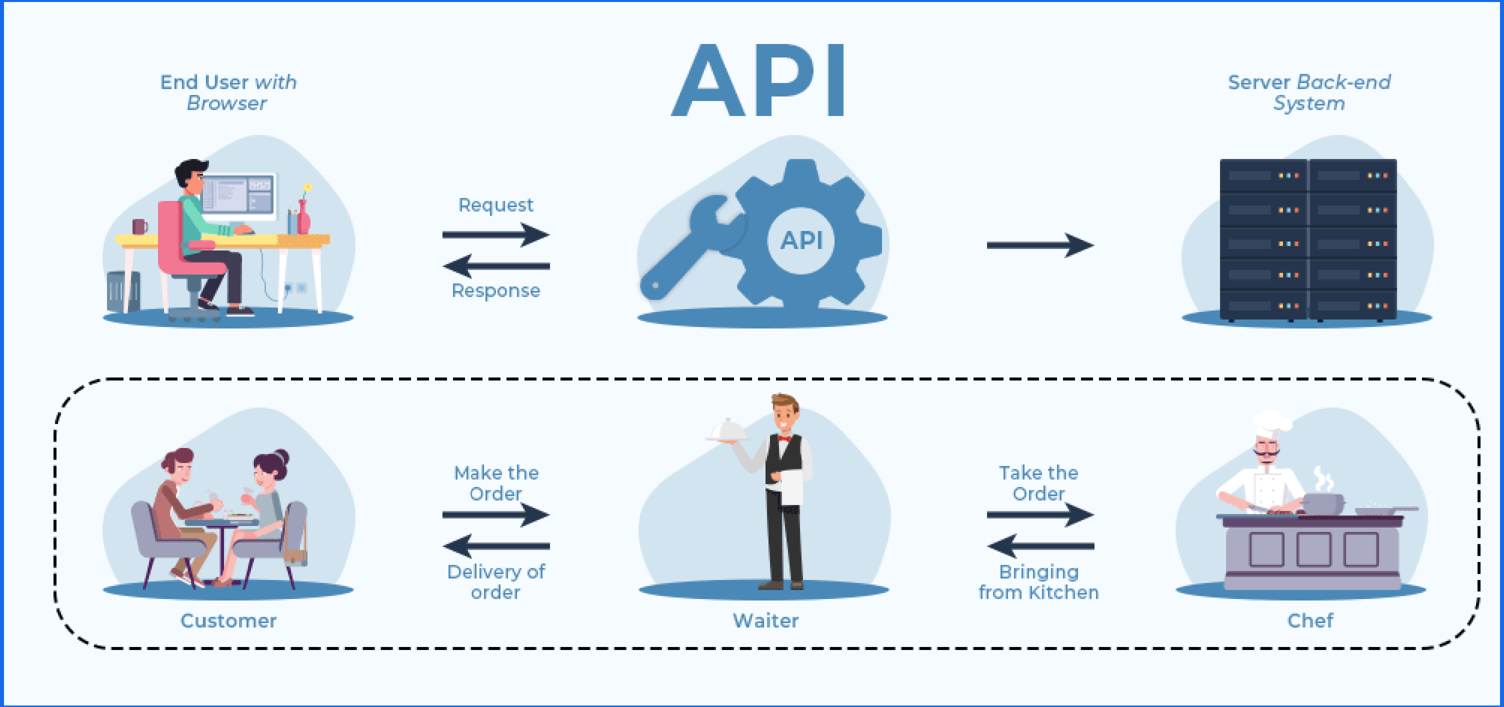
Shaida Muhammad

Tuesday, March 25, 2025

ShaidaSherpao@gmail.com

Agenda

What is an API?



What is an API?

- **Definition:** An API (Application Programming Interface) defines how software components interact.
- **Analogy:** Like a restaurant menu which provides a list of operations you can request.
- **Examples:**
  - Fetching weather data
  - Integrating payment systems
  - Retrieving social media posts

Types of APIs

1. **Web APIs** (REST, GraphQL, SOAP)
2. **Library APIs** (e.g., `requests`, `pandas`)
3. **OS APIs** (Windows API, POSIX)
4. **Hardware APIs** (e.g., GPU acceleration)

Feature	**Web APIs**	**Library APIs**	**OS APIs**	**Hardware APIs**
Scope	Network	Software	Operating System	Physical Hardware
Protocol	HTTP/HTTPS	Language calls	System calls	Driver interfaces
Example	Twitter API	pandas DataFrame	Windows API	CUDA
Complexity	Medium	Low-Medium	Medium-High	High

HTTP & REST Basics

- HTTP Methods:
  - GET (Retrieve data)
  - POST (Create data)
  - PUT/PATCH (Update data)
  - DELETE (Remove data)
- REST Principles:
  - Stateless communication
  - Resource-based URLs (e.g., /users, /posts)
  - JSON/XML responses

API Request Structure

- Components of an HTTP Request

```
POST /api/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN

{
  "name": "John Doe",
  "email": "john@example.com"
}
```

- Key Parts:

  1. Request Line
  - POST: HTTP method (GET/POST/PUT/DELETE)
  - /api/users: Endpoint path
  - HTTP/1.1: Protocol version

  2. Headers
  - Metadata (e.g., Content-Type, Authorization)

  3. Body (Optional)
  - Data sent to the server (JSON/XML/form-data)

API Response Structure

- **Components of an HTTP Response**

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/users/123

{
  "id": 123,
  "name": "John Doe",
  "status": "active"
}
```

- Key Parts:

  1. **Status Line**
     - `HTTP/1.1 201 Created`: Protocol + status code/message
  2. **Headers**
     - Metadata (e.g., `Content-Type`, `Location` for new resources)
  3. **Body (Optional)**
     - Returned data (typically JSON/XML)

Consuming APIs with `requests`

- Basic GET Request

```
import requests
response = requests.get("https://api.github.com/users/ShaidaMuhammad")
print(response.json())
```

- Key Methods:
  - `response.status_code` (200, 404, 500)
  - `response.json()` (Parse JSON data)

Handling API Responses

- Example: Error Handling

```
try:
    response = requests.get("https://api.example.com/data", timeout=5)
    response.raise_for_status()
except requests.exceptions.RequestException as e:
    print(f"Error: {e}")
```

Query Parameters & Headers

- Adding Parameters

```
params = {"q": "Python", "page": 1}
response = requests.get("https://api.example.com/search", params=params)
```

- Custom Headers

```
headers = {"Authorization": "Bearer YOUR_TOKEN"}
response = requests.get("https://api.example.com/protected", headers=headers)
```

Building APIs with FastAPI

Why FastAPI?

- High performance (async support)
- Automatic documentation (/docs and /redoc)
- Data validation with Pydantic

Basic Example

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}
```

API Authentication Methods

1. API Keys
   - **How it works**: Simple unique string passed via:
     - URL params (`?api_key=YOUR_KEY`)
     - Headers (`X-API-Key: YOUR_KEY`)
   - **Pros**: Easy to implement.
   - **Cons**: Low security (exposed in logs/URLs).
   - **Use case**: Public APIs (e.g., weather data).

2. OAuth 2.0
   - **How it works**: Token-based flow with roles:
     - **Client** (your app)
     - **Resource Owner** (user)
     - **Authorization Server** (e.g., Google/Facebook)
   - **Pros**: Secure, delegated access (no password sharing).
   - **Cons**: Complex setup.
   - **Use case**: Social logins ("Sign in with Google").

3. JWT (JSON Web Tokens)
   - **How it works**: Encrypted JSON payload with:
     - **Header** (algorithm)
     - **Payload** (user data)
     - **Signature** (verification).
   - **Pros**: Stateless, self-contained.
   - **Cons**: Token revocation challenges.
   - **Use case**: Modern web/mobile apps.

4. Basic Auth
   - **How it works**: Base64-encoded `username:password` in headers:
     - `Authorization: Basic base64("user:pass")`.
   - **Pros**: Simple HTTP standard.
   - **Cons**: Plaintext credentials (always use HTTPS).
   - **Use case**: Internal/legacy systems.

Securing APIs and Best Practices

- Securing APIs
  - **HTTPS** (Never HTTP)
  - **Rate Limiting** (Prevent abuse)
  - **Input Validation** (Avoid SQL injection)
  - **CORS Policies** (Control cross-origin access)

- API Best Practices
  - **Use RESTful conventions** (e.g., `/users`, `/users/{id}`)
  - **Versioning** (e.g., `/v1/users`)
  - **Documentation** (Swagger/OpenAPI)
  - **Pagination & Filtering** (e.g., `?page=2&limit=10`)
  - **Error Handling** (Proper HTTP status codes)

Examples, Tools, Summary

Real-World API Examples:

1. **GitHub API** (Fetch repositories, user data)
2. **OpenWeatherMap API** (Weather forecasts)
3. **Twitter API** (Retrieve tweets)
4. **Public APIs List** (`https://github.com/public-apis/public-apis`)

Tools for API Development:

- **Postman** (API testing)
- **Swagger UI** (Interactive documentation)
- **HTTPie** (Command-line HTTP client)

Summary:

- APIs enable software communication.
- Python can **consume** (`requests`) and **build** (FastAPI) APIs.
- Security and best practices are critical.
- Real-world APIs power modern applications.

Q&A

- Do you have any questions?
- Share your thoughts.

Closing

Next class: Git & GitHub