Building Python APIs with FastAPI

Nikolay Pomytkin - pomytkin@umd.edu
nikolay.pomytkin.com

https://go.umd.edu/bitcamp-fastapi

PLAN

1. CLIENT VS SERVER
2. COMMUNICATION & HTTP
3. URI’s, Requests, Responses
4. Interaction with an API
5. Web development in Python
6. Intro to FastAPI
7. Live coding demo (time permitting)

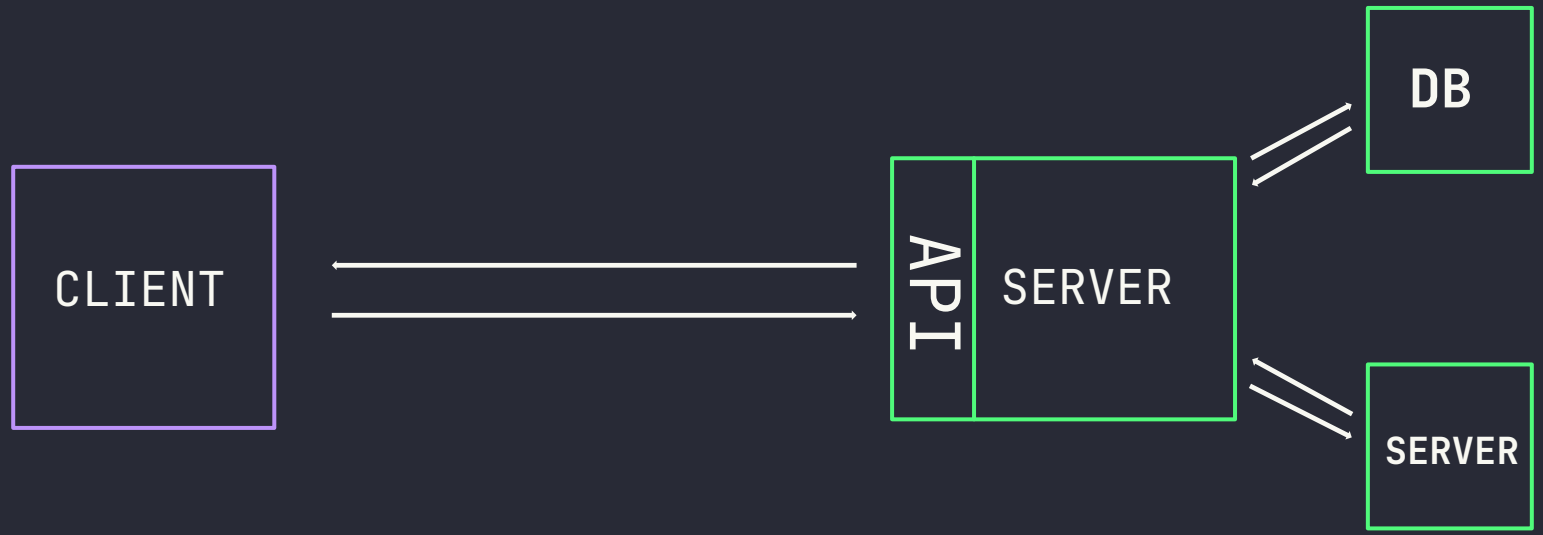
CLIENT

- The browser acts as the client (user).
- The client will always initiate the request to the server (for most basic cases).
- A client allows users to make requests through the web.
- Requests happen asynchronously, meaning the client needs to wait some x time to receive a response from the server.

SERVER

- Servers provide functionality and serve responses to the client.
- A single server can serve multiple clients at the same time.
- A server can contain resources, host applications, and store user and program data.
- A server is always listening to requests from a client, and will then do some computation and return the respective resource.

BASIC INTERACTION



URIs (Uniform Resource Identifiers)

https://api.com/v1/wizards

LOCATION

RESOURCE

REQUESTS

GET → READ
POST → CREATE
PATCH → UPDATE
DELETE → DESTROY

STATUS CODES

2** → GOOD
4** → YOU'RE AT FAULT
5** → SERVER AT FAULT

REQUEST ANATOMY

POST	/wizards	HTTP/1.1
METHOD	URI	HTTP -V

```
HOST: gryf.hogwarts.com
Content-Type: Content/Json
Date: Tuesday, 01 Feb 2022
```

HEADERS

```
{
    name : “potter”
}
```

BODY

RESPONSE ANATOMY

HTTP/1.1	200	Ok
HTTP -V	Code	Message

```
Server: Apache
Content-Type: Content/Json
Date: Tuesday, 01 Feb 2022
```

HEADERS

```
{
    wizards: [...]
}
```

BODY

Web Development in Python

- Most popular options for building web apps today:
  - Django (Django REST Framework for APIs)
  - Flask (Flask-RESTful for APIs)
- Django is a huge framework that requires a lot of time to learn and it is not very flexible
- Flask was designed to be a lightweight alternative (and it’s great for building full-stack apps), however it lacks built-in functionality for API development
- Solution: FastAPI

What is FastAPI?

“FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.”

- Uses pydantic for data validation, type hints, automatic documentation generation
- Uses starlette as the underlying web library
- Uses uvicorn as an asynchronous Python web server

Documentation:

https://fastapi.tiangolo.com/

Why FastAPI?

- Super fast 🚀
  - Fastest Python web framework (independent benchmarks: https://www.techempower.com/benchmarks/)
  - On-par with Node.js and Go
- Super easy to learn and use
- Super powerful feature set (yet somehow not bloated)
  - Supports async python with asyncio
  - Type hints
  - Data validation
  - Automatic documentation generation with OpenAPI (supports both Swagger-UI and ReDoc)
  - Integrated security and authentication features

Make an API with 5 Lines of Code:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/wizard/{wizard_id}")
async def get_wizard(wizard_id: int):
    return {"wizard_id": wizard_id}
```

- In a few lines of code, we've already taken advantage of many of FastAPI's features:
  - Data validation: type hint in function parameter
  - Path parameters: `@app.get("/user/{user_id}")` → `get_user(user_id: int)`
  - Implicit JSON conversion: returning a python dictionary

Quick Aside: Some resources to learn from

- This presentation is not the best way to learn how to build APIs with FastAPI
  - I only started using it a few months ago, so I'm not an expert
  - I learned about it from the official docs and code samples on Github
- Great learning resources:
  - Official tutorial from FastAPI docs: https://fastapi.tiangolo.com/tutorial/
  - Full-stack FastAPI + Vue.js + Postgres template: https://github.com/tiangolo/full-stack-fastapi-postgresql
  - Real Python tutorial: https://realpython.com/fastapi-python-web-apis/
  - Google! :)

Setup

Step 1: Install required libraries

- Need to have Python installed (version 3.6+)
- Install fastapi and uvicorn

```
pip3 install fastapi uvicorn
```

Step 2: Run the local web server

- Use the `--reload` uvicorn option to enable auto reload

```
uvicorn --reload main:app
```

- This will run the “app” instance in “main.py”

Path Parameters

- Can pass data through URI
- Type enforcement through parameter type hint
- Used for accessing a specific resource

```python
@app.get("/wizard/{wizard_id}")
async def get_wizard(wizard_id: int):
    return {"wizard_id": wizard_id}
```

- ReDoc / Swagger to test (or an application like Insomnia)

Query Parameters

- A client can pass an indefinite amount of key,value pairs through the URI (up to a limit of 2048 characters)
- Generally used to pass additional information, data filters, etc in GET requests

```
from typing import Optional
@app.get("/spell/{spell_name}")
async def get_spell(spell_name: str, q: Optional[str] = None):
    if q:
        return {"spell_name": spell_name, "q": q}
    return {"spell_name": spell_name}
```

Pydantic Data Models

- Pydantic's `BaseModel` allows you to easily define schemas for data using Python classes
- We can then use this as a type hint in our route functions to validate POST request bodies

```python
from pydantic import BaseModel

class Wizard(BaseModel):
    first_name: str
    last_name: str
    age: int
    hogwarts_house: Optional[str]
```

POST request bodies

- As we saw earlier, POST requests are special because we can pass data without using the URL
- Similar to path and query parameters, we can validate this data
- `response_model` helps us with generating documentation

```
@app.post("/wizard", response_model=Wizard)
async def add_wizard(wizard: Wizard):
    return wizard
```