Fast API

Summary	In this codelab, we will learn how to create Flask based Web apps in Python.
URL	https://fastapi.tiangolo.com/
Category	Web
Status	Published
Author	Nikhil Kohli

Synchronous vs Asynchronous python
- Synchronous vs Asynchronous python
  - Synchronous Server
  - Asynchronous Server setup
  - Some of the Asynchronous frameworks -

FastAPI Introduction
- What is FastAPI?
  - The key features of FastAPI are:
  - Which Companies are using FastAPI?

Requirements & Installation
- Requirements
- Installation

FastAPI Features
- 1. Automatic interactive API documentation
- 2. Supports Async IO operations
- 3. Helps quickly create REST endpoints for our application

Running a Basic Example Application
  - Interactive API docs
  - Alternative Documentation

Synchronous vs Asynchronous python

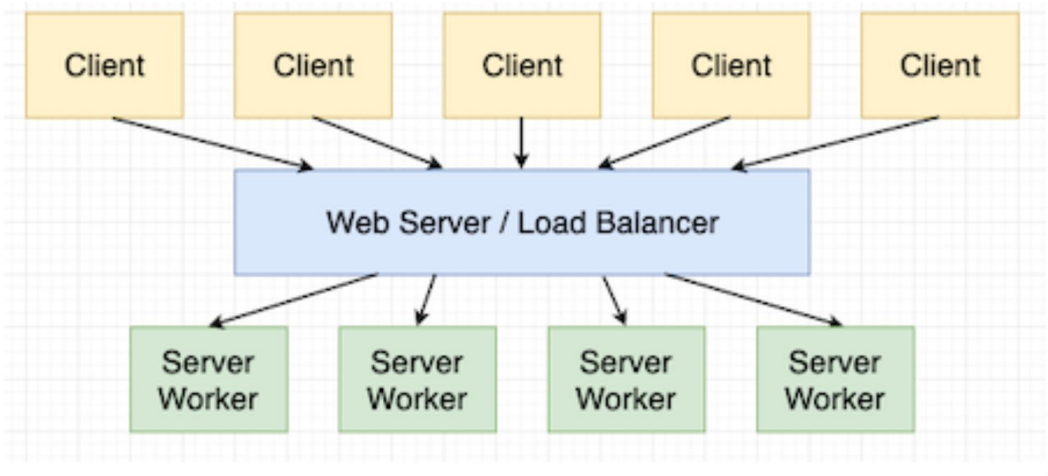
First things first, Why are we talking about Fast API? We already know Flask and we can create our web applications using that easily. Then why fastAPI?

Synchronous vs Asynchronous python

Web applications often have to deal with many requests, all arriving from different clients and within a short period of time. To avoid processing delays it is considered a must that they should be able to handle several requests in parallel, something commonly known as concurrency.
The terms "sync" and "async" refer to two ways in which to write applications that use concurrency. The so called "sync" servers use the underlying operating system support of threads and processes to implement this concurrency.

Synchronous Server

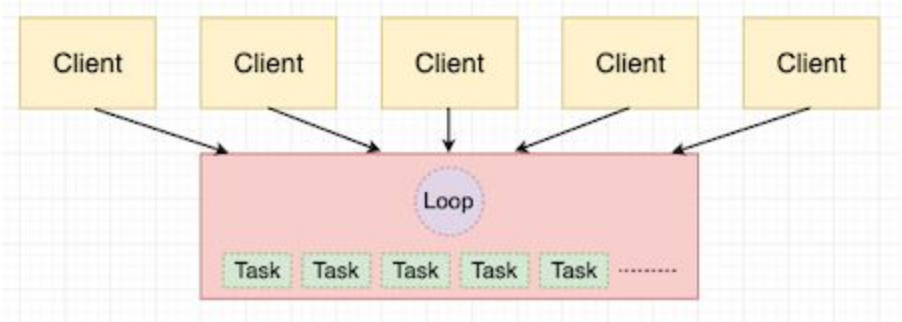
Here is a diagram of how a sync deployment might look:



In this situation we have five clients, all sending requests to the application. The public access point for this application is a web server that acts as a load balancer by distributing the requests among a pool of server workers, which might be implemented as processes, threads or a combination of both. The workers execute requests as they are assigned to them by the load balancer. The application logic, which you may write using a web application framework such as Flask or Django, lives in these workers.

In terms of disadvantages, the diagram above clearly shows what the main limitation of this approach is. We have five clients, but only four workers. If these five clients send their requests all at the same time, then the load balancer will be able to dispatch all but one to workers, and the request that lost the race will have to remain in a queue while it waits for a worker to become available. So four of the five clients will receive their responses timely, but one of them will have to wait longer for it.

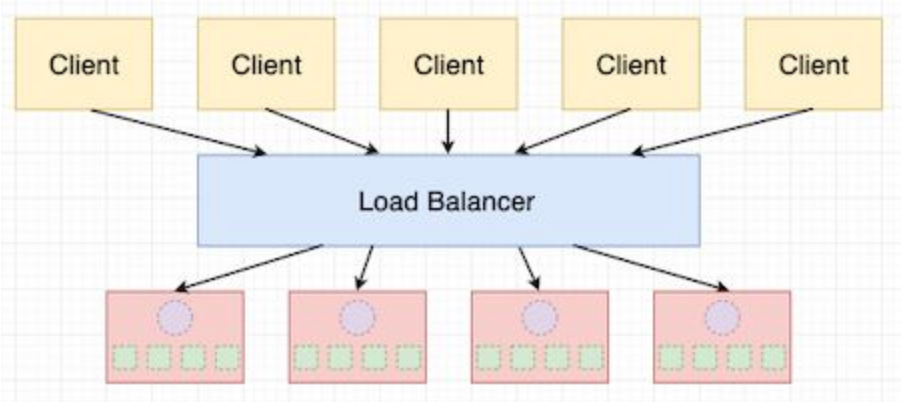
Asynchronous Server setup



This type of server runs in a single process that is controlled by a loop. The loop is a very efficient task manager and scheduler that creates tasks to execute the requests that are sent by clients. Unlike server workers, which are long lived, an async task is created by the loop to handle a specific request, and when that request is completed the task is destroyed. At any given time an async server may have hundreds or even thousands of active tasks, all doing their own work while being managed by the loop.

You may be wondering how the parallelism between async tasks is achieved. This is the interesting part, because an async application relies exclusively on cooperative multitasking for this. What does this mean? When a task needs to wait for an external event, like for example, a response from a database server, instead of just waiting like a sync worker would do, it tells the loop what it needs to wait for and then returns control to it. The loop then is able to find another task that is ready to run while this task is blocked by the database. Eventually the database will send a response, and at that point the loop will consider that first task ready to run again, and will resume it as soon as possible.

To maximize the utilization of multiple CPUs when using an async server, it is common to create a hybrid solution that adds a load balancer and runs an async server on each CPU, as shown in the following diagram:



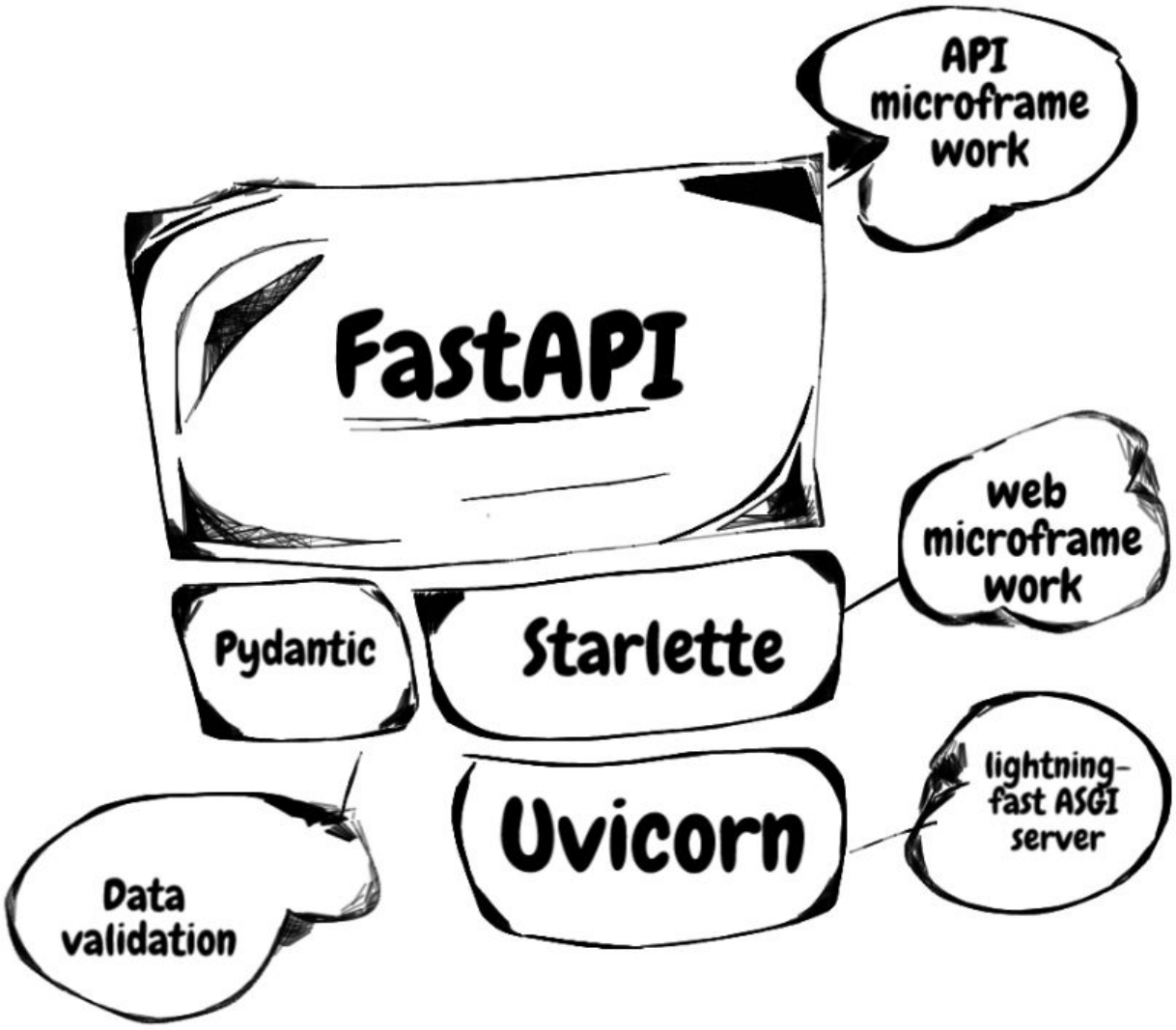
Some of the Asynchronous frameworks -

1. Tornado
2. Sanic
3. Vibora
4. FastAPI
5. Quart

FastAPI Introduction

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."



The key features of FastAPI are:

- **Fast:** Very high performance, on par with NodeJS and Go (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- **Fast to code:** Increase the speed to develop features by about 200% to 300%. *
- **Fewer bugs:** Reduce about 40% of human (developer) induced errors.
- **Intuitive:** Great editor support. Completion everywhere. Less time debugging.
- **Easy:** Designed to be easy to use and learn. Less time reading docs.
- **Short:** Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- **Robust:** Get production-ready code. With automatic interactive documentation.

Which Companies are using FastAPI?

Uber, Microsoft, Netflix, Explosion AI etc.

Requirements & Installation

Requirements

- Python 3.6+
- Uvicorn

FastAPI stands on the shoulders of giants:

- Starlette for the web parts.
    - Starlette is a lightweight ASGI (Asynchronous Server Gateway Interface) framework/toolkit, which is ideal for building high performance services
- Pydantic for the data parts.
    - Data validation and settings management using python type annotations. It enforces type hints at runtime, and provides user friendly errors when data is invalid.

Uvicorn is a lightning-fast ASGI server implementation, ASGI (Asynchronous Server Gateway Interface) is a spiritual successor to WSGI, intended to provide a standard interface between async-capable Python web servers, frameworks, and applications.

ASGI helps enable an ecosystem of Python web frameworks that are highly competitive against Node and Go in terms of achieving high throughput in IO-bound contexts. It also provides support for HTTP/2 and WebSockets, which cannot be handled by WSGI.

Installation

- Creating a Virtual Environment

```
python -m venv fastdemo
```

```
dir
```

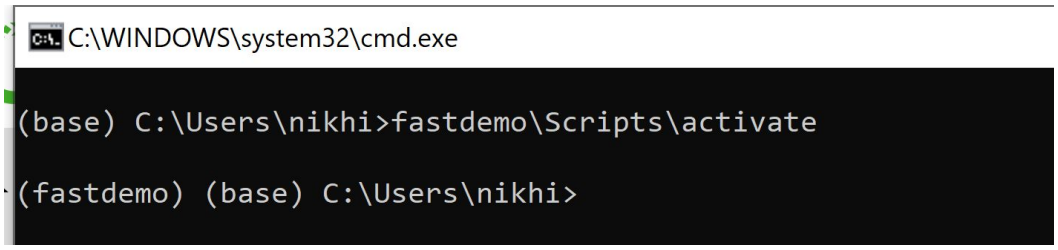
- Activate Virtual Environment

Windows:

```
fastdemo\Scripts\activate
```

Mac:

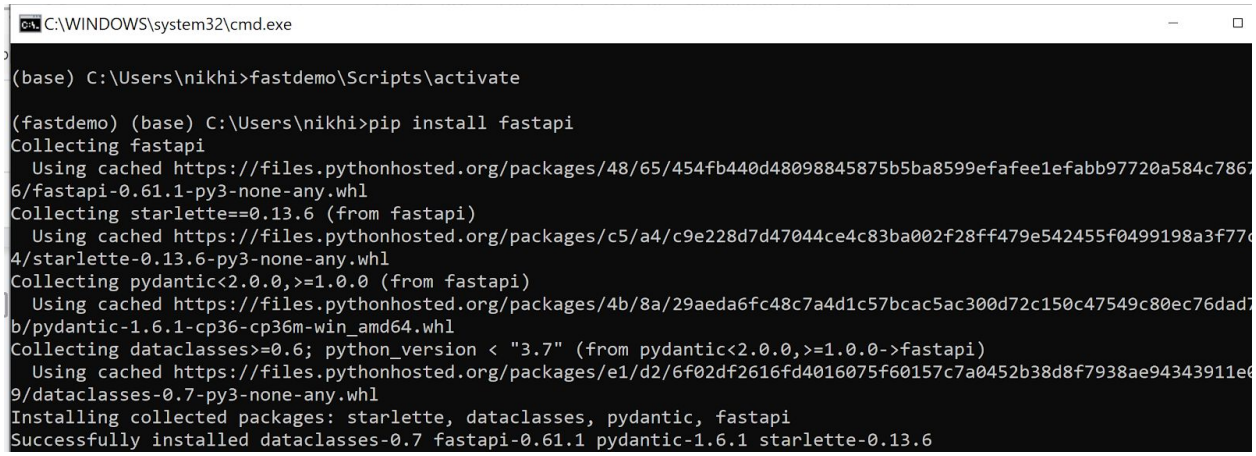
```
Source demo_env/bin/activate
```



We do not need to install Starlette and pydantic explicitly, these will be installed with fastAPI

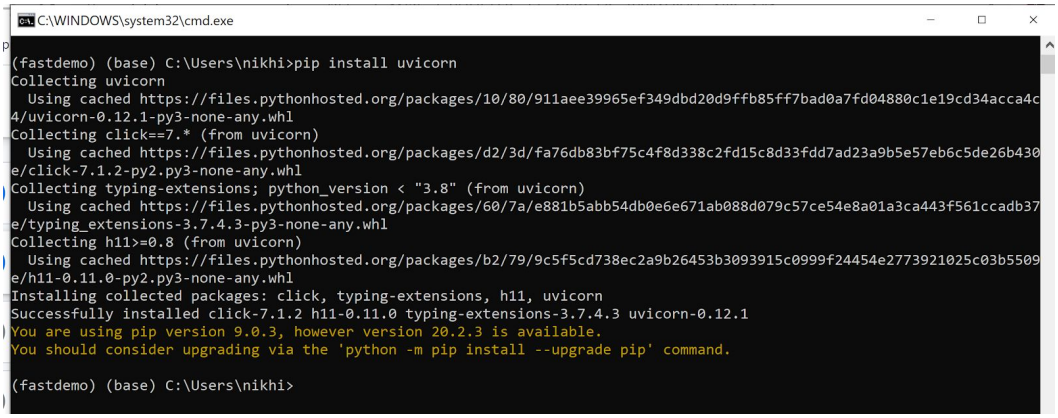
- **Install fastAPI**

```
pip install fastapi
```



- Install Uvicorn

```
pip install uvicorn
```



FastAPI Features

1. Automatic interactive API documentation

- The FastAPI is based on the open standards for OpenAPI and JSON Schema. As a result, it can automatically build a SwaggerUI for our web API without any additional coding as long as we use the appropriate Pydantic data types.
- SwaggerUI is interactive in nature and allows us to test an API directly from the browser.
- FastAPI also lets us validate the input data from the external callers and generates automatic errors to the clients when it receives invalid data.
- The OpenAPI automatically generates a schema for the application. FastAPI also supports data serialisation and input data parsing.

2. Supports Async IO operations

- It's vital to build a web API that does not block its callers whilst it is performing operations. The asyncio library in Python offers a large number of async capabilities.
- The FastAPI library supports Async IO operations using the Starlette library.
- The Starlette library itself sits on top of the Uvicorn. Uvicorn is a lightning-fast ASGI server that supports asyncio frameworks.

3. Helps quickly create REST endpoints for our application

- Flask users have to install the Flask-restplus package to create REST endpoints for their data science application.

- The FastAPI supports the GET, PUT, POST, DELETE, OPTIONS, HEAD, PATCH and TRACE Rest operations without any additional packages. All of the routes along with their operations are automatically generated in the documentation.

Running a Basic Example Application

- Create a python file main.py

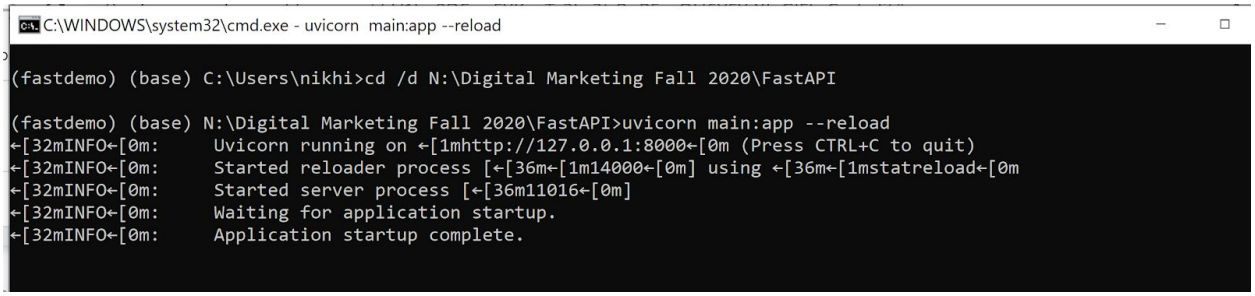
```
from typing import Optional
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

Run the file using below command

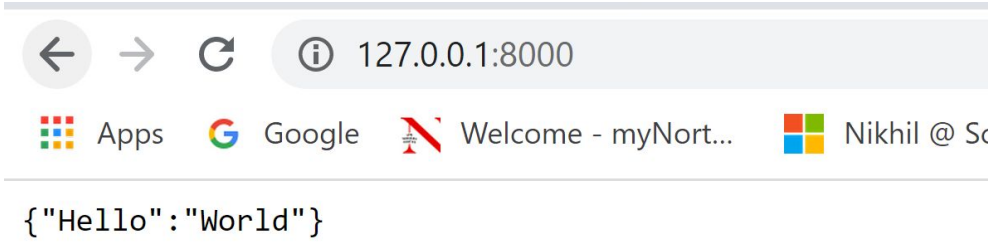
```
Uvicorn main:app --reload
```

The command uvicorn main:app refers to:

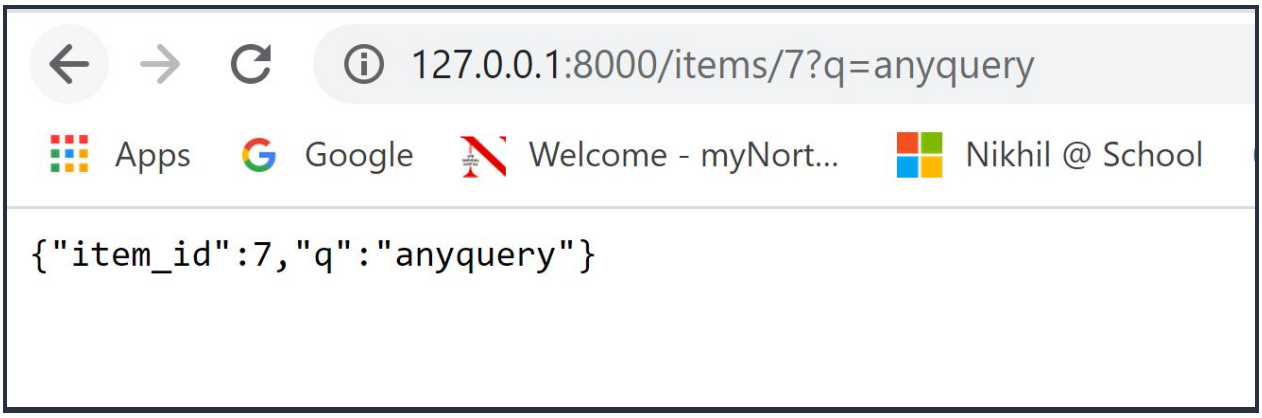
- main: the file main.py (the Python "module").
- app: the object created inside of main.py with the line app = FastAPI().
- --reload: make the server restart after code changes. This for development only.



Go to the path - http://127.0.0.1:8000



http://127.0.0.1:8000/items/7?q=anyquery



You will see the JSON response as:

```
{"item_id": 7, "q": "anyquery"}
```

You created an API that:

- Receives HTTP requests in the *paths* `/` and `/items/{item_id}`.
- Both *paths* take `GET` *operations* (also known as HTTP *methods*).
- The *path* `/items/{item_id}` has a *path parameter* `item_id` that should be an `int`.
- The *path* `/items/{item_id}` has an optional `str` *query parameter* `q`.

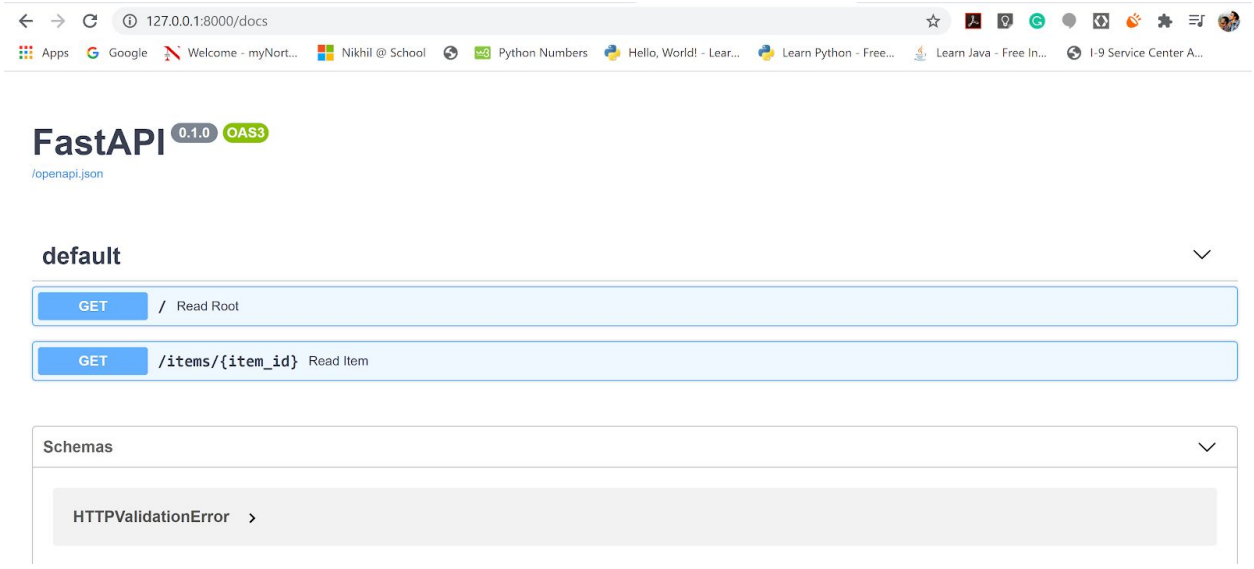
Interactive API docs

Remember going to Postman and testing your API? Here we can test the API in our browser itself with the help of Swagger UI.

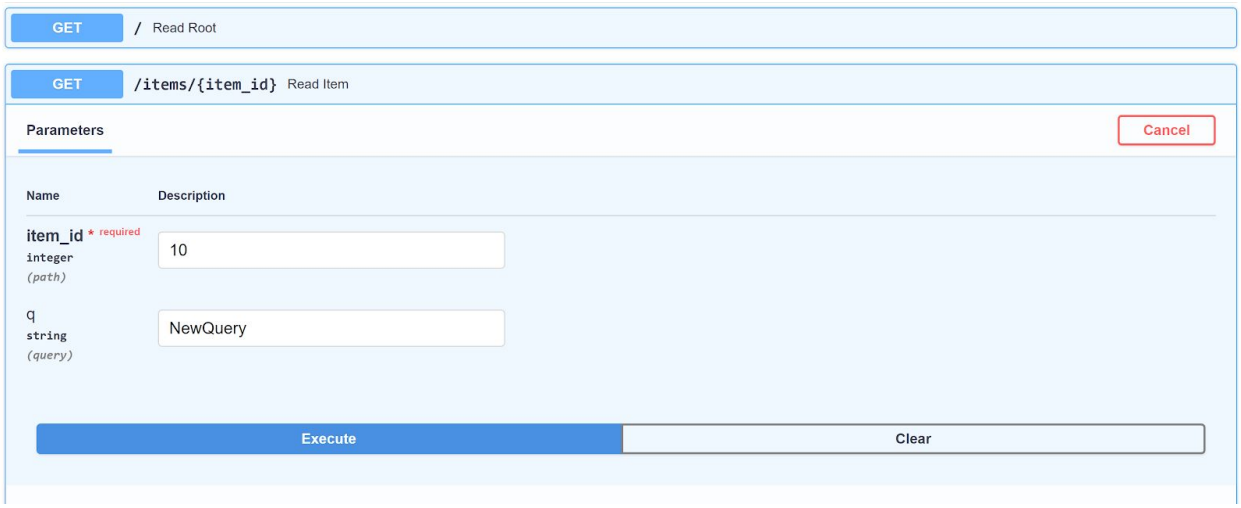
Now go to

```
http://127.0.0.1:8000/docs
```

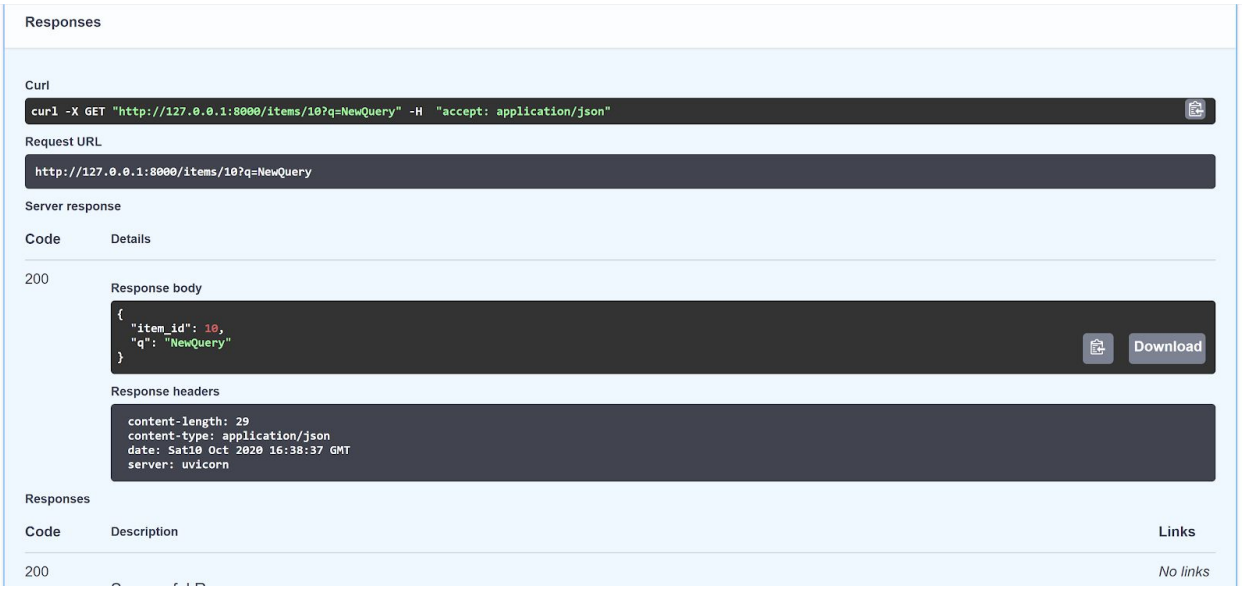
You will see the automatic interactive API documentation (provided by Swagger UI):



Click on try it out



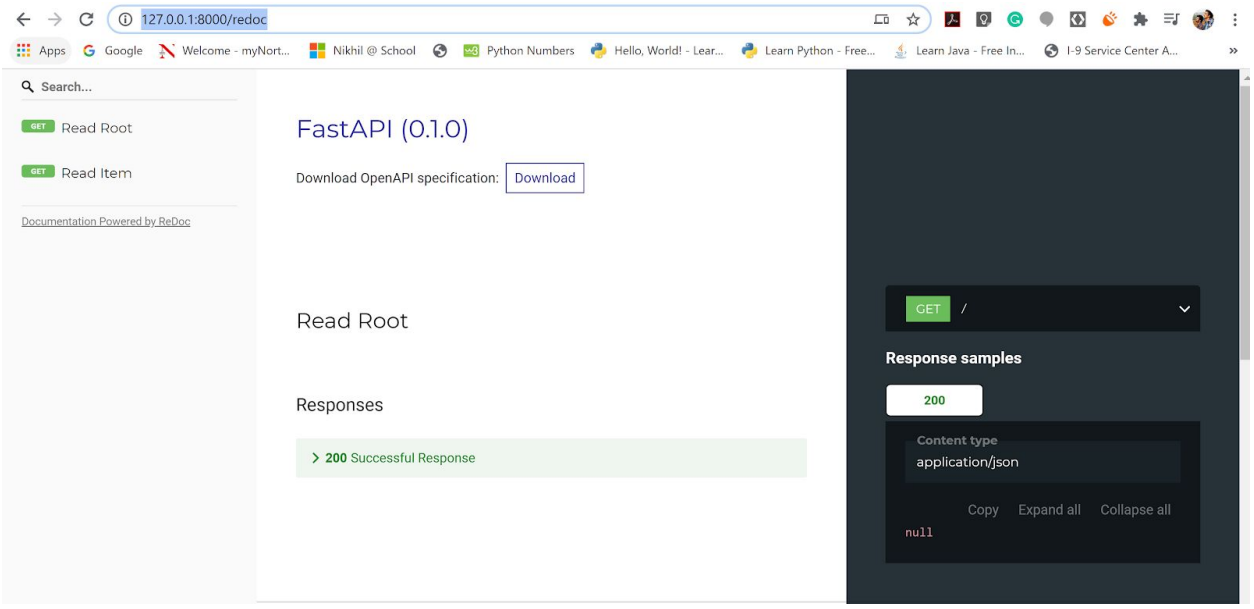
Input your values for item_id and any optional parameter and click Execute.



Alternative Documentation

Go to

```
http://127.0.0.1:8000/redoc
```

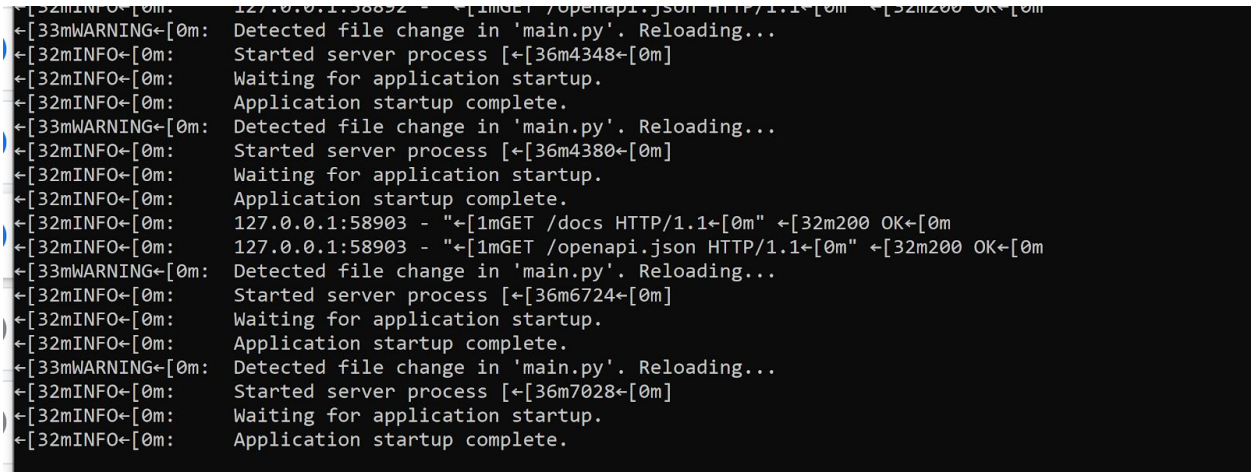


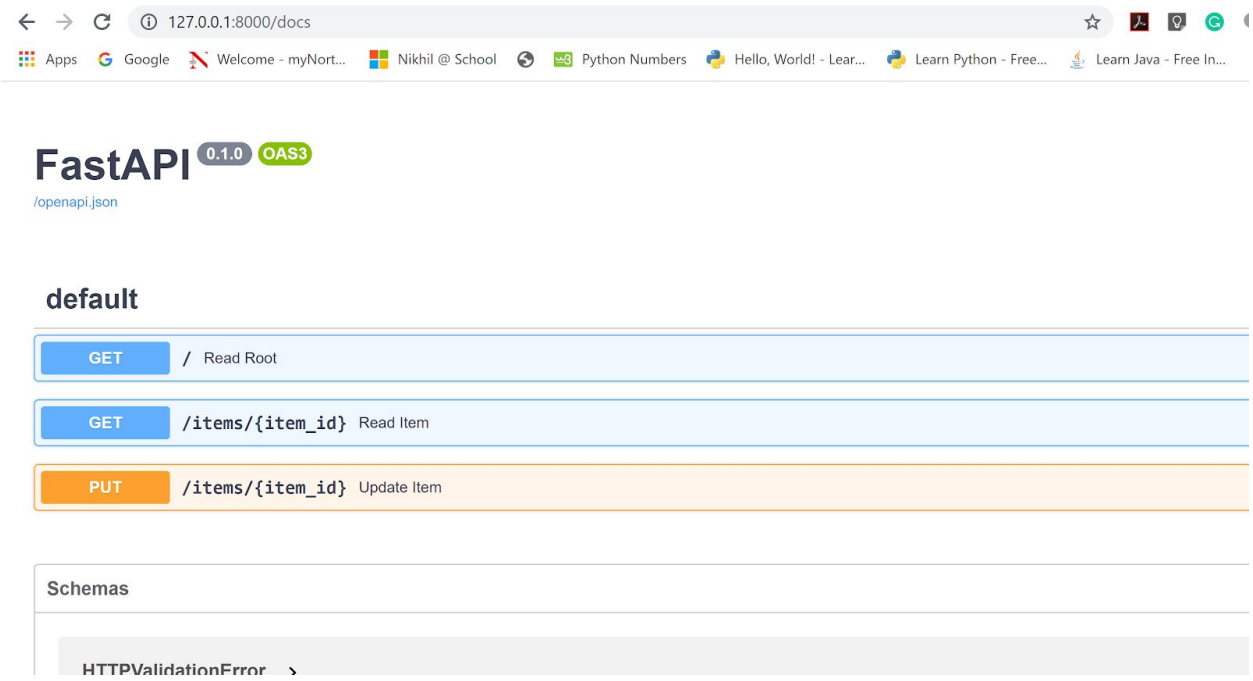
Now modify the file main.py to receive a body from a PUT request.

```
from typing import Optional
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
   name: str
   price: float
   is_offer: Optional[bool] = None


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}



@app.put("/items/{item_id}")
```

```
def update_item(item_id: int, item: Item):
    return {"item_name": item.name, "item_id": item_id}
```

The server should reload automatically as we added --reload to the uvicorn command





You can see the updated documentation with PUT, try to execute it now.

References

https://fastapi.tiangolo.com/

http://www.uvicorn.org/

https://blog.miguelgrinberg.com/post/sync-vs-async-python-what-is-the-difference

https://geekflare.com/python-asynchronous-web-frameworks/