- FastAPI is an open source Python web API framework.

- Using it, we can create fully functional CRUD operation-supporting APIs, and their documentation are generated automatically.

- Based on OpenAPI standards.

- Installing FastAPI with all dependencies:-

```
pip install fastapi [all]
```

- Then, for use, importing it as follows:-

```
from fastapi import FastAPI
```

↓

case should be same

- Then, we mainly create instance of FastAPI, to use it.

```
app = FastAPI
```

↳ can be any name here, but it is just a convention

This "app" object will now be used to set endpoint paths and define functions that will

be seen when that specific endpoint is requested with the specified HTTP method. These functions that will handle the request, are actually the API's.

Simplest API :-

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

PATH operation / Routing

Save this py file with any name, say main.py

This is the API, that can be now tested by running the uvicorn live server :-

```
uvicorn main:app
```

main : name of the Python file in which the API is coded

app : name of the object, using which we coded the API

Suppose, the API is already running, & we made

some changes in code of API, then to reload the API, the above command is run using following flag

`uvicorn main:app --reload`

⇒ If we run the above command (with flag), then, after each change in code (& saved), the API will get automatically reloaded, & we do not have to do it manually.

- After running the command, in the terminal it will be shown at which URL / endpoint on our local machine, the API is running.

- Opening that url in browser, will show the object that we returned from our API (function).

Interactive Documentation :-

a) Of the API that we just created, we can get the documentation that is automatically generated, as follow :-

`localhost:8000/docs`

At this URL, we can test all our API's interactively.

also, alternative documentation URL :- `localhost:8000/redoc`

- Thus, the Path Operation / Routing consists of 2 parts:
  - a) Decorator
  - b) Function.

◇ Function :-

- It is a simple python function.
- If the request sent to the API is to be made as Asynchronous, the "async" keyword is used

If is optional, and if not used, the request made to the API will be synchronous.

- Name of the function can be anything — does not matter But, the name should be descriptive.

- In the function body, we do the coding that should get executed when the URL is requested.
  It may be storing some data into database, or fetching some data from database.

- What we return from the function is returned to the user as the response of the API.
  Fast API converts the response to JSON automatically.

◇ Decorator :-

- The decorator actually makes the Python function act like API, i.e, converts it to Path Operation.

- The decorator annotation syntax (in general) is :-

@ fastAPI_instance . http_method (endpoint path)

name of the object

get, post

endpoint path in quotes

- At the place of endpoint path, there should be a string, starting with "/", which means it is relative path, after the domain name & port no.

Some ex':

"/" → http://localhost:8000/

"/test/" → http://localhost:8000/test/

The domain will get changed acc. to hosting, but the relative path mentioned will become the endpoint of that new domain (in which the API will be hosted)

- If we define multiple path operations for single URL endpoint, the one that is coded above in the file will get executed.

Thus, for convinience, all Path Operations should have different URL endpoints set.

POST requests :-

- POST requests are mainly used to send data to server, securely.

Getting request data inside Path Operation func.:

importing Body for using it.

```
from fastapi.params import Body

@app.post("/createposts")
def create_posts(payload: dict = Body(...)):
    print(payload)
```

this is type check/ conversion

this variable can be named anything

Body(...) extracts all of the field's data from the requests' body, and converts it into a dictionary, and will store into the variable (which is named "payload" here).

For testing, we are printing the dictionary that we obtained, to check what gets received.

Now, while testing this API using POSTMAN, send the data as Python dictionary object, in the "raw" section of "Body" tab.

Also, you must change the type sent from [Text v] to [JSON v] from the dropdown menu.

In real applications, we generally store the data obtained from user, using POST request.

⇒ Suppose, the user sends "first_name" and "last_name" as request body. So, we can code something like this:-

```
...
def handle_data(data: dict = Body(...)):
    print(type(data), data.keys(), data.values())
    return {"msg": f"welcome {data['first_name']}"}
```

dict functions

simply accessing the dictionary value

⊛ Need for Schema :-

⇒ Without a schema, client can send whatever data they want.

⇒ The data is not getting validated.

⇒ Thus, we ultimately want to force the client to send data in a schema that we expect.

⇒ Thus, to efficiently work with API's, we need to explicitly mention how & what data should be provided, and restrict entry of others types of data.

For this, we will use a library called PYDANTIC.