
FIAP

FastAPI

Introdução

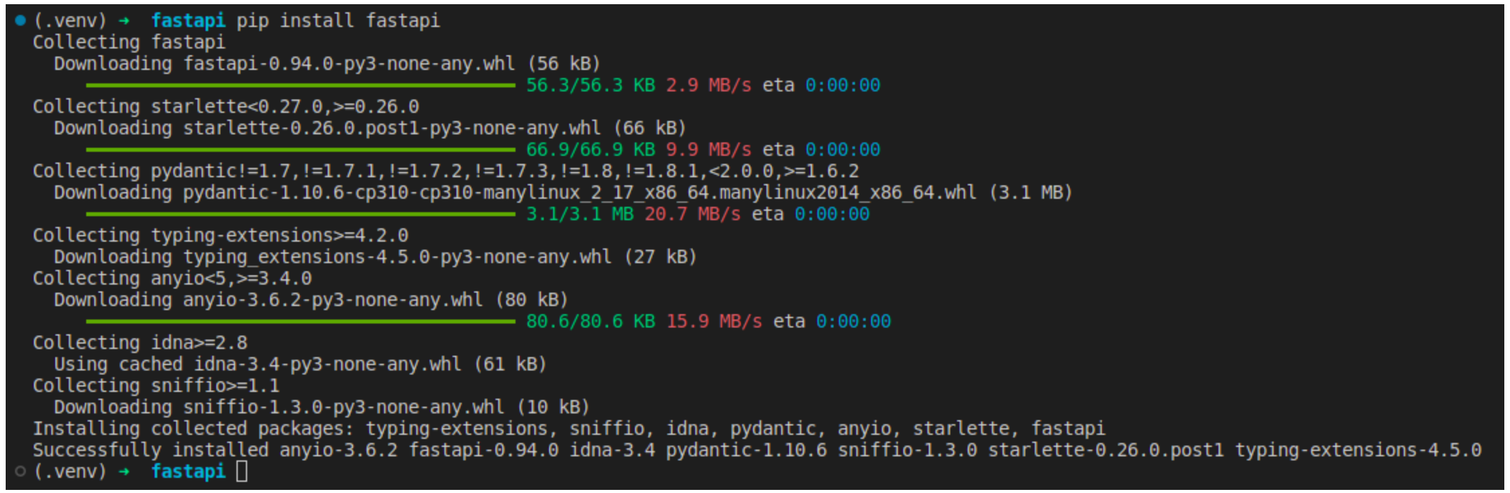
Sua aplicação precisa ser robusta e estável de tal forma que sua aplicação fique rodando 24/7.
Muitas aplicações tem bem definida uma estrutura de backend e de frontend. Ao utilizar o FastAPI, criamos uma estrutura de backend onde não compromete o frontend e vice-versa.
Sua aplicação vai precisar armazenar e manipular dados. E quando precisamos desse tipo de recurso, utilizamos banco de dados, seja ele SQL ou NoSQL, vai depender muito da sua necessidade.

FastAPI

FastAPI é um framework Python para desenvolvimento de APIs de forma ágil, rápida e minimalista. Tem uma boa aceitação da comunidade e tem se destacado por ser robusta, intuitiva e fácil de se utilizar.

Após criarmos nosso virtualenv, para instalar basta rodar o comando:

```
pip install fastapi
```



FastAPI

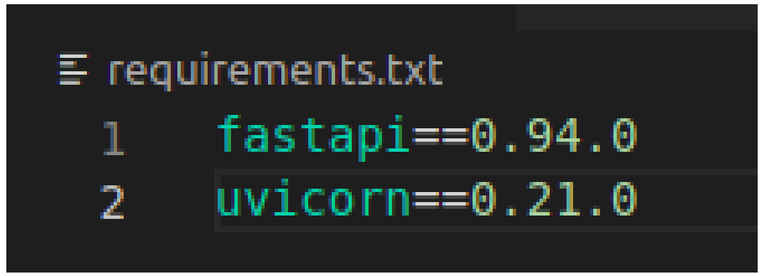
Também precisamos instalar o uvicorn. Ele é uma implementação de minimalista de servidor web para Python.
Para instalar, basta rodar o comando:

```
pip install "uvicorn[standard]"
```

FastAPI

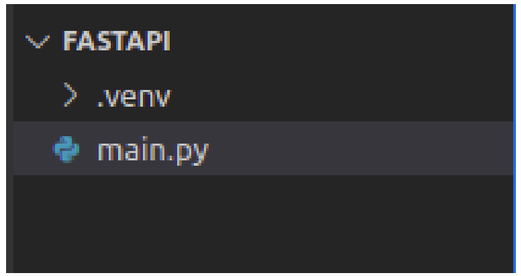
Iremos criar um arquivo chamado **requirements.txt**. É comum criar esse arquivo para colocar quais as bibliotecas que sua aplicação utiliza e se precisar criar um novo ambiente, você usa ele para instalar essas bibliotecas.

No conteúdo do nosso arquivo, terá apenas duas linhas:



FastAPI

Agora iremos criar nossa aplicação. Primeiro, crie um arquivo chamado `main.py` dentro do diretório. No meu exemplo, meu diretório está da seguinte forma:



FastAPI

E vamos criar nosso primeiro endpoint! Para isso, coloque dentro do seu `main.py`, o seguinte conteúdo:

```
from fastapi import FastAPI
app = FastAPI()

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

FastAPI

Só isso? Sim, só isso. Agora vamos o comando a seguir e ver nossa API rodando?

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

E pronto! Acesse no browser as seguintes URLs:

http://localhost:8000/docs -> A documentação das suas APIs
http://localhost:8000 -> O endpoint que irá retornar o conteúdo `{"Hello": "World"}`

FastAPI

Falando um pouco sobre o comando:

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

`main` -> Nome do arquivo que você criou, no caso, main.py

`app` -> O objeto app que foi criado na linha app = FastAPI()

`--reload` -> Qualquer alteração feita no código, sua aplicação irá automaticamente ser reiniciada

FastAPI

Caso você utilize VSCode e debugar o projeto dentro dele, ele já vem com uma opção para isso. Ao clicar no menu **Run** e selecionar a opção **Start Debugging**, irá aparecer uma caixa de diálogo com a opção **FastAPI**. Na sua estrutura de diretórios, deverá ter a pasta **.vscode** com um arquivo **launch.json**. A seguir o conteúdo dele, onde é bom adicionar opção `"--reload"`

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: FastAPI",
            "type": "python",
            "request": "launch",
            "module": "uvicorn",
            "args": [
                "main:app",
                "--reload"
            ],
            "jinja": true,
            "justMyCode": true
        }
    ]
}
```

FastAPI

Como podemos ver no código, fizemos uma requisição do tipo `GET`

```
from fastapi import FastAPI
app = FastAPI()

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

Utilizando o decorador `app.get`, informamos a aplicação que essa função irá receber as requisições do tipo `GET`

FastAPI

Caso queira validar o "`query`" que podemos informar na URL, basta adicionar os argumentos na função.

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def hello_world(name: str):
   return {"Hello": name}
```

FastAPI

Ao fazer a requisição em `http://localhost:8000/?name=Leonardo`. Podemos usar o browser ou pode ser feito pela linha de comando:

```
curl "http://localhost:8000?name=Leonardo"
```

O retorno será o seguinte:

```
{"hello":"Leonardo"}
```

Corpo da requisição

Caso queira mandar dados no corpo da requisição para serem processados, o `FastAPI` tem uma forma muito elegante de lidar com os dados da requisição. Para isso, devemos utilizar uma biblioteca que vem junto com ele, chamada `pydantic`
Vamos supor que você quer receber em uma requisição o email e nome do usuário.
Vamos criar um modelo da seguinte forma:

```
from pydantic import BaseModel

class User(BaseModel):
    email: str
    name: str
```

POST

Além da requisição `GET`, podemos tratar todas as requisições suportadas pelo protocolo HTTP.
Vamos implementar o metódo `POST` utilizando modelo `User` que criamos

```
import random

@app.post("/users")
def create_user(user: User):
    return {
        "id": random.randint(0, 100),
        "name": user.name,
        "email": user.email
    }
```

POST

Ao fazer uma requisição utilizando a linha de comando da seguinte forma:

```
curl -X "POST" -H 'Content-type: application/json' "http://localhost:8000/users" -d '{"name": "Leonardo", "email": "leonardo@fiap.com.br"}'
```

O retorno será:

```
{"id":21,"name":"Leonardo","email":"leonardo@fiap.com.br"}
```

POST

Criamos um modelo chamado `User`, que tem dois atributos: `email` e `name`
Quando for a requisição é feita no endpoint `/users`, o `FastAPI` automaticamente entende os dados recebidos e converte para a classe `User`, facilitando a manipulação desses dados.

Código de status

No `FastAPI` é possível retornar qualquer código de status HTTP que sua aplicação considerar necessária
Pegando o exemplo anterior de uma requisição `POST`, para informar o código de status, basta alterar da seguinte forma:

```
@app.post("/users", status_code=201)
def create_user(user: User):
    return {
        "id": random.randint(0, 100),
        "name": user.name,
        "email": user.email
    }
```

Ao fazer uma requisição neste endpoint, agora o código de status será 201 ao invés do 200

Código de status

Caso queira retornar algum erro, como um 400, por exemplo, pode ser feito da seguinte forma:

```
from fastapi import HTTPException

@app.post("/users", status_code=201)
def create_user(user:User):
    if "@" not in user.email:
        raise HTTPException(400, "Informe um email válido")

    return {
        "id": random.randint(0, 100),
        "name": user.name,
        "email": user.email
    }
```

Para isso, não se esqueça de importar a classe `HTTPException`

PATCH

Para se receber uma requisição `PATCH`, podemos fazer da seguinte forma:

```
@app.patch("/users/{user_id}")
def partial_update_user(user_id:int, user:User):
    if "@" not in user.email:
        raise HTTPException(400, "Informe um email válido")

    return {
        "id": user_id,
        "name": user.name,
        "email": user.email
    }
```

PUT

Para se receber uma requisição `PATCH`, podemos fazer da seguinte forma:

```
@app.put("/users/{user_id}")
def update_user(user_id:int, user:User):
    if "@" not in user.email:
        raise HTTPException(400, "Informe um email válido")

    return {
        "id": random.randint(0, 100),
        "name": user.name,
        "email": user.email
    }
```

DELETE

Para se receber uma requisição `DELETE`, podemos fazer da seguinte forma:

```
users = {1: {"name": "Leonardo", "email": "email@email.com"}}

@app.delete("/users/{user_id}")
def delete_user(user_id:int):
    try:
        del users[user_id]
    except KeyError:
        pass
```

Parâmetros

Nos exemplos das requisições `PATCH`, `PUT` e `DELETE`, foi utilizado um caminho que recebe como argumento o `user_id`. No `FastAPI`, quando definimos um caminho, podemos definir variáveis que pode ser utilizadas ao receber uma requisição.

```
@app.put("/users/{user_id}")
def update_user(user_id:int, user:User):
    if "@" not in user.email:
        raise HTTPException(400, "Informe um email válido")

    return {
        "id": random.randint(0, 100),
        "name": user.name,
        "email": user.email
    }
```

No exemplo acima, o caminho é definido por `/users/{user_id}`. Ou seja, será passado como argumento para a função o parâmetro `user_id` para ser utilizado conforme for a necessidade

Parâmetros

É importante ressaltar que a ordem dos parâmetros é relevante. Supondo que além do `user_id`, queira receber um novo parâmetro.

```
@app.put("/users/{user_id}/{item_id}")
def update_user(user_id:int, item_id:int, user:User):
    if "@" not in user.email:
        raise HTTPException(400, "Informe um email válido")

    return {
        "id": random.randint(0, 100),
        "name": user.name,
        "email": user.email
    }
```

Como se pode notar no exemplo, o caminho recebe os parâmetros `user_id` e `item_id`. Na função que recebe a requisição, a ordem deve ser equivalente ao passado no caminho. Ou seja, `user_id` é o primeiro argumento da função, enquanto o `item_id` é o segundo argumento.
Além da ordem serem equivalentes, o nome também deve coincidir.


FIAP