

FastAPI

QXD0099 - Desenvolvimento de Software para Persistência

Universidade Federal do Ceará - *Campus* Quixadá

Prof. Francisco Victor da Silva Pinheiro
victorpinheiro@ufc.br



Agenda

- O que é FastAPI?
- Principais características
- Python versões
- FastAPI
- Gerenciadores de pacotes Python
- Setup do ambiente e primeiros passos
- Primeiro exemplo básico
- main.py
- HTTP - Códigos de resposta
- Códigos de resposta mais usados
- Uvicorn
- Ruff
- main2.py
- main3.py

O que é FastAPI?

- FastAPI é um framework web moderno e rápido para construção de APIs em Python, projetado com foco em alto desempenho e simplicidade.
- Ele utiliza as capacidades do Python tipo-assinaturas (type hints) para fornecer verificação de tipos e documentação automática, o que facilita a criação e o gerenciamento de APIs.
- Comparação com outros frameworks (Flask, Django, etc.).
- Destaca-se pela criação de APIs RESTful.



FastAPI

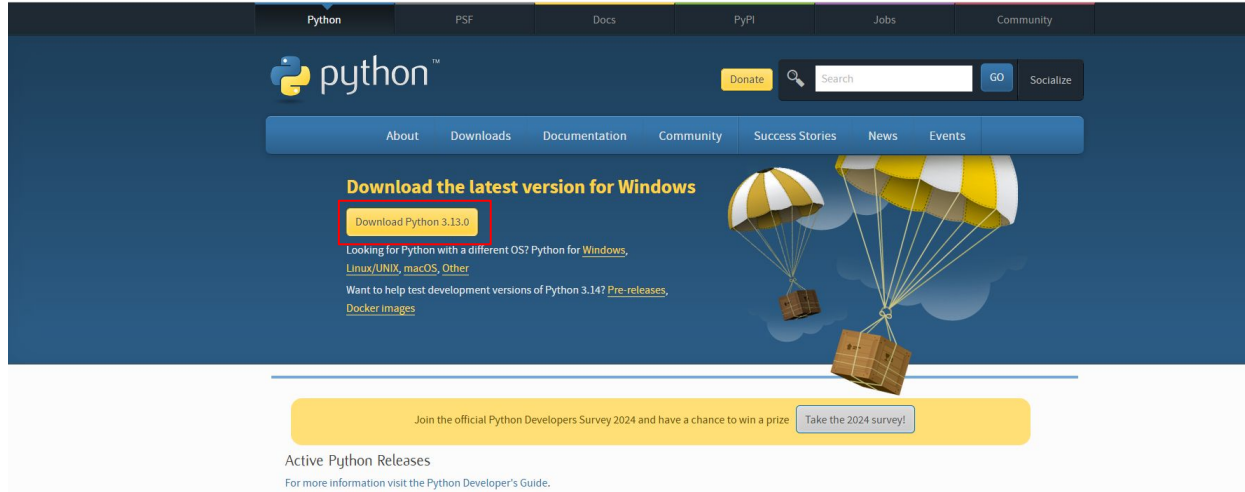


Principais características

- **Alto desempenho:** É uma das opções mais rápidas para construção de APIs em Python, comparável a Node.js e Go, devido à utilização do ASGI (Asynchronous Server Gateway Interface) e do framework Starlette.
- **Tipagem forte:** FastAPI usa o Pydantic para validação de dados, o que permite o uso de tipagem estática e facilita o tratamento e a validação de dados, reduzindo erros.
- **Documentação automática:** Com base nas anotações de tipos, ele gera automaticamente a documentação Swagger e Redoc para as APIs, facilitando o entendimento e o uso por parte dos desenvolvedores.
- **Suporte a operações assíncronas:** Ele permite o uso de funções async para suportar operações assíncronas, tornando-se ideal para aplicações que requerem alta performance e escalabilidade.

Python versões

- 3.10 – 04/out/2021
- 3.11 – 24/out/2022
- 3.12 – 02/out/2023
- 3.13 – 07/out/202



FastAPI

- Site: <https://fastapi.tiangolo.com/>
- Github: <https://github.com/fastapi/fastapi>
- Requisitos:
 - Starlette – framework web leve.
 - Pydantic – biblioteca para validação de dados.

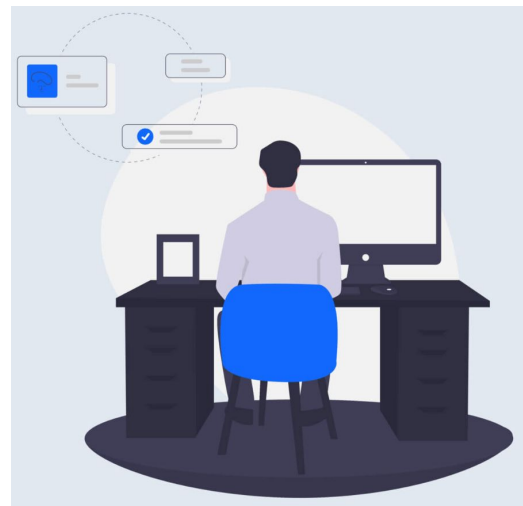


Gerenciadores de pacotes Python

- pip
- conda
- poetry
- uv - <https://docs.astral.sh/uv/>
 - gerenciador de pacotes extremamente rápido e gerenciador de projetos, escrito em Rust.
 - 10-100x mais rápido que o pip.
 - Substitui pip, pip-tools, pipx, poetry, pyenv, twine, virtualenv, etc.
 - Instalação Linux / Mac:
 - `curl -LsSf https://astral.sh/uv/install.sh | sh`

Setup do ambiente e primeiros passos

- Instalação:
 - Usando ***pip install fastapi e uvicorn*** (para execução).
 - ***pip install fastapi uvicorn***
- Primeiro exemplo básico:
 - Escreva uma aplicação básica que responde com "Hello, World!".
- Pra rodar:
 - ***uvicorn main:app --reload***
 - ou Execute o Uvicorn diretamente com Python
 - ***python -m uvicorn main:app --reload***



Exemplo básico

```
from fastapi import FastAPI

app = FastAPI()

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

- Este exemplo cria um endpoint que retorna uma mensagem JSON. Com apenas essa configuração, o FastAPI já cria automaticamente a documentação interativa da API, acessível em <http://localhost:8000/docs>.
- FastAPI é ideal para APIs RESTful, microserviços e sistemas que exigem alta taxa de requisições e baixo tempo de resposta, e é amplamente usado em aplicações de machine learning e big data devido à sua eficiência e facilidade de integração com bibliotecas de dados do Python.

main.py

```
from typing import Union
from fastapi import FastAPI

app = FastAPI()

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

@app.get("/itens/{item_id}")
def read_item(item_id: int, nome: Union[str, None] = None):
    return {"item_id": item_id, "nome": nome}
```

- O `@app.get("/")` define um endpoint de método GET na raiz (/), que serve como ponto de entrada padrão.
- O `@app.get("/itens/{item_id}")` define um endpoint GET com um parâmetro de caminho `item_id`, permitindo a passagem de um identificador para o item e opcionalmente um nome.
- Acessar GET / retorna `{"msg": "Hello World"}`.
- Acessar GET `/itens/1?nome=Teste` retorna `{"item_id": 1, "nome": "Teste"}`.

main.py

1. **Endpoint raiz (/):**
 - **Rota:** "/"
 - **Método HTTP:** GET
 - **Função:** `read_root()`
 - **Descrição:** Quando você acessa `http://localhost:8000/`, essa rota retorna um JSON com a mensagem `{"msg": "Hello World"}`.

2. **Endpoint de item (/itens/{item_id}):**
 - **Rota:** `/itens/{item_id}`, onde `{item_id}` é uma variável de caminho.
 - **Método HTTP:** GET
 - **Função:** `read_item(item_id: int, nome: Union[str, None] = None)`
 - **Parâmetros:**
 - `item_id` (int): Um parâmetro de caminho obrigatório que deve ser um número inteiro.
 - `nome` (Union[str, None]): Um parâmetro de consulta opcional que pode ser uma string ou `None`.
 - **Descrição:** Esse endpoint retorna um JSON com o `item_id` e o `nome` (caso fornecido). Por exemplo:
 - GET `/itens/1?nome=Teste` retornaria `{"item_id": 1, "nome": "Teste"}`.
 - GET `/itens/2` retornaria `{"item_id": 2, "nome": null}` (pois `nome` não foi fornecido).

FastAPI

- Execute assim para acesso local:
 - `fastapi dev main.py`
- Abrindo a aplicação para o “mundo”:
 - `fastapi dev main.py --host 0.0.0.0`
- No navegador acesse:
 - <http://127.0.0.1:8000/>
 - <http://127.0.0.1:8000/items/1?q=monitor>
 - Documentação:
 - <http://localhost:8000/docs>
 - <http://localhost:8000/redoc>



HTTP - Códigos de resposta

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

```
@app.get("/itens/{item_id}")
def read_item(item_id: int, nome: Union[str, None] = None):
    return {"item_id": item_id, "nome": nome}
```

Os códigos de resposta HTTP são usados para indicar o resultado de uma requisição para um endpoint.

HTTP - Códigos de resposta

- **1xx: informativo**
 - utilizado para enviar informações para o cliente de que sua requisição foi recebida e está sendo processada.
- **2xx: sucesso**
 - Indica que a requisição foi bem-sucedida (por exemplo, 200 OK, 201 Created).
- **3xx: redirecionamento**
 - Informa que mais ações são necessárias para completar a requisição (por exemplo, 301 Moved Permanently, 302 Found).
- **4xx: erro no cliente**
 - Significa que houve um erro na requisição feita pelo cliente (por exemplo, 400 Bad Request, 404 Not Found).
- **5xx: erro no servidor**
 - Indica um erro no servidor ao processar a requisição válida do cliente (por exemplo, 500 Internal Server Error, 503 Service Unavailable).



- 1XX
INFORMATIVO
- 2XX
SUCESSO
- 3XX
REDIRECIONAMENTO
- 4XX
ERRO DO CLIENTE
- 5XX
ERRO DO SERVIDOR

Códigos de resposta mais usado

- **200 OK**
 - A solicitação foi bem-sucedida. O significado exato depende do método HTTP utilizado na solicitação.
- **201 Created**
 - A solicitação foi bem-sucedida e um novo recurso foi criado como resultado.
- **404 Not Found**
 - O recurso solicitado não pôde ser encontrado, sendo frequentemente usado quando o recurso é inexistente.
- **422 Unprocessable Entity**
 - Usado quando a requisição está bem-formada, mas não pode ser seguida devido a erros semânticos.
 - É comum em APIs ao validar dados de entrada.
- **500 Internal Server Error**
 - Quando ocorre um erro na nossa aplicação.

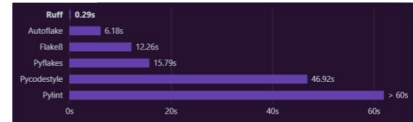
Uvicorn

- <https://www.uvicorn.org>
- Servidor de aplicação ASGI (Asynchronous Server Gateway Interface).
- Ao contrário do WSGI (usado por frameworks como Flask e Django), o ASGI permite suporte a conexões assíncronas e WebSockets, muito úteis para aplicações modernas, que demandam comunicação em tempo real.
- Projetado para rodar aplicações Python assíncronas, especialmente aquelas desenvolvidas com frameworks modernos como FastAPI e Starlette.
- Leve e rápido
 - Ideal para situações em que a performance e a capacidade de lidar com conexões simultâneas são cruciais.
- A aplicação pode rodar assim também:
 - `uvicorn main:app`



Ruff

- <https://docs.astral.sh/ruff/>
- Linter
 - Analisar o código de forma estática
 - Efetuar a verificação se estamos programando de acordo com boas práticas do python.
- Formatter
 - Efetuar a verificação do código para padronizar um estilo de código pré-definido.
- Instalar no projeto
 - `uv add ruff`
- Comandos
 - `ruff check` .: Faz a checagem dos termos que definimos antes
 - `ruff format` .: Faz a formatação do nosso código sendo as boas práticas



main2.py

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

app = FastAPI()

class Item(BaseModel):
    nome: str
    valor: float
    is_oferta: Union[bool, None] = None

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

@app.get("/itens/{item_id}")
def le_item(item_id: int, nome: Union[str, None] = None):
    return {"item_id": item_id, "nome": nome}

@app.put("/itens/{item_id}")
def atualiza_item(item_id: int, item: Item):
    return {"item_nome": item.nome, "item_id": item_id}
```

Modelo de Dados (Item):

- Define um item com **nome** (str), **valor** (float), e **is_oferta** (opcional, bool).

Endpoint / (GET):

- Retorna uma mensagem: {"msg": "Hello World"}.

Endpoint /itens/{item_id} (GET):

- Recebe **item_id** (int) e **nome** (opcional, str).
- Retorna {"item_id": item_id, "nome": nome}.

Endpoint /itens/{item_id} (PUT):

- Recebe **item_id** e dados do **Item** no corpo.
- Retorna {"item_nome": item.nome, "item_id": item_id} para atualizar o item.

main2-1.py

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

app = FastAPI()

class Item(BaseModel):
    nome: str
    valor: float
    is_oferta: Union[bool, None] = None

# Dicionário para armazenar itens
items_db = {}

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

@app.get("/items/{item_id}")
def le_item(item_id: int):
    # Verifica se o item existe no dicionário
    if item_id in items_db:
        return items_db[item_id]
    else:
        return {"erro": "Item não encontrado"}

@app.put("/items/{item_id}")
def atualiza_item(item_id: int, item: Item):
    # Salva o item no dicionário
    items_db[item_id] = item
    return {"mensagem": "Item atualizado com sucesso", "item": items_db[item_id]}
```

- **items_db**: Um dicionário para armazenar itens usando **item_id** como chave.
- **le_item**: A função de **GET** agora verifica se o **item_id** está no **items_db** e retorna o item, se encontrado. Caso contrário, retorna uma mensagem de erro.
- **atualiza_item**: A função de **PUT** armazena o item recebido no **items_db** usando **item_id**

main3.py (1/3)

```
from typing import Union
from typing import List
from http import HTTPStatus
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    id: int
    nome: str
    valor: float
    is_oferta: Union[bool, None] = None

itens: List[Item] = []

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

main3.py (2/3)

```
@app.get("/itens/{item_id}", response_model=Item)
def ler_item(item_id: int):
    for indice, item_atual in enumerate(itens):
        if item_atual.id == item_id:
            return item_atual
    raise HTTPException(status_code=HTTPStatus.NOT_FOUND, detail="Item não encontrado.")

@app.get("/itens/", response_model=List[Item])
def listar_itens():
    return itens

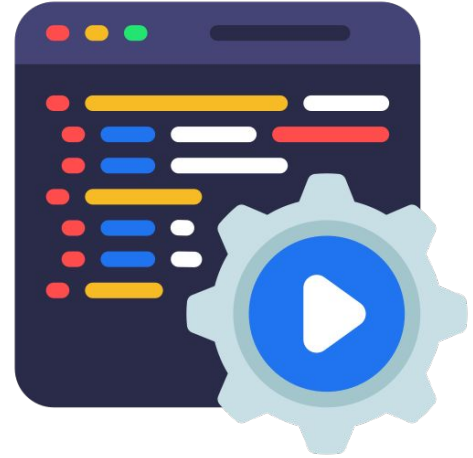
@app.post("/itens/", response_model=Item, status_code=HTTPStatus.CREATED)
def adicionar_item(item: Item):
    if any(item_atual.id == item.id for item_atual in itens):
        raise HTTPException(status_code=400, detail="ID já existe.")
    itens.append(item)
    return item
```

main3.py (3/3)

```
@app.put("/itens/{item_id}", response_model=Item)
def atualizar_item(item_id: int, item_atualizado: Item):
    for indice, item_atual in enumerate(itens):
        if item_atual.id == item_id:
            if item_atualizado.id != item_id:
                item_atualizado.id = item_id
            itens[indice] = item_atualizado
    return item_atualizado
    raise HTTPException(status_code=HTTPStatus.NOT_FOUND, detail="Item não encontrado.")
```

Referências

- <https://fastapi.tiangolo.com/>
- <https://fastapi.tiangolo.com/tutorial/>



Obrigado! Dúvidas?



Universidade Federal do Ceará - *Campus* Quixadá

Prof. Francisco Victor da Silva Pinheiro
victorpinheiro@ufc.br

