Passo 1: Preparar o Ambiente

Instalar o FastAPI e Uvicorn

Primeiro, você precisa instalar o **FastAPI** e o **Uvicorn**, que será o servidor que rodará sua aplicação.

1. Crie um ambiente virtual (opcional, mas recomendado):

bash

CopiarEditar

python -m venv venv

source venv/bin/activate # No Windows: venv\Scripts\activate

2. Instale o **FastAPI** e o **Uvicorn**:

bash

CopiarEditar

pip install fastapi uvicorn

Passo 2: Criar o Arquivo Principal da API

Agora, vamos criar o arquivo principal da aplicação. Vamos chamá-lo de main.py.

Código do Arquivo main.py:

python

CopiarEditar

```
# Importa a classe FastAPI para criar a aplicação
from fastapi import FastAPI


# Criação da instância da aplicação FastAPI
app = FastAPI()


# Rota raiz (endpoint "/") para retornar uma mensagem simples
@app.get("/")
def read_root():
  return {"message": "Olá, Mundo!"}
```

Explicação:

1. **from fastapi import FastAPI**: Importa a classe FastAPI que é utilizada para criar a aplicação.
2. **app = FastAPI()**: Cria uma instância da classe FastAPI, que é a nossa aplicação.
3. **@app.get("/")**: Define uma rota do tipo GET para o endpoint /. Quando o usuário acessar esse endpoint, a função read_root() será executada.
4. **return {"message": "Olá, Mundo!"}**: A função read_root retorna um dicionário Python, que será automaticamente convertido em formato JSON pelo FastAPI.

Passo 3: Rodar a API

Para rodar a aplicação, usamos o **Uvicorn** como servidor. Execute o seguinte comando no terminal:

bash

CopiarEditar

uvicorn main:app --reload

- **main**: Refere-se ao nome do arquivo main.py.
- **app**: Refere-se à variável app onde criamos a instância do FastAPI.
- **--reload**: Faz com que a aplicação recarregue automaticamente sempre que houver alterações no código (útil durante o desenvolvimento).

Agora, abra o navegador e acesse http://127.0.0.1:8000/. Você verá o seguinte JSON de resposta:

json

CopiarEditar

{

"message": "Olá, Mundo!"

}

Passo 4: Criar Endpoints com Diferentes Métodos HTTP

Agora, vamos adicionar alguns endpoints com diferentes métodos HTTP, como GET e POST.

Exemplo 1: Endpoint GET com Parâmetros de URL

Vamos criar um endpoint que aceita parâmetros diretamente na URL.

python

CopiarEditar

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

Explicação:

1. **@app.get("/items/{item_id}")**: Define uma rota GET para o endpoint /items/{item_id}, onde {item_id} é um parâmetro de URL dinâmico.
2. **item_id: int**: O parâmetro item_id será automaticamente convertido para um tipo int pelo FastAPI. O FastAPI faz a validação de tipo automaticamente.
3. **q: str = None**: O parâmetro q é opcional, e se não for fornecido, seu valor será None.
4. **return {"item_id": item_id, "q": q}**: Retorna um JSON com os valores de item_id e q.

Testando:

Acesse a URL http://127.0.0.1:8000/items/42?q=fastapi, e você verá a seguinte resposta:

json

CopiarEditar

```
{
  "item_id": 42,
  "q": "fastapi"
}
```

Passo 5: Criar Endpoint POST com Body

Agora, vamos criar um endpoint POST que aceita dados no corpo da requisição.

python

CopiarEditar

```
# Importa a classe BaseModel do Pydantic, que será usada para validar os dados
```

```
from pydantic import BaseModel


# Definição do modelo Item para validar os dados que o cliente vai enviar
class Item(BaseModel):
    name: str
    description: str = None
    price: float
    tax: float = None


# Rota POST para criar um novo item
@app.post("/items/")
def create_item(item: Item):
    return {"name": item.name, "price": item.price}
```

Explicação:

1. **from pydantic import BaseModel**: O Pydantic é uma biblioteca usada para validação de dados. O FastAPI a utiliza internamente para validar os dados que são enviados para os endpoints.
2. **class Item(BaseModel)**: Aqui, estamos criando um modelo Item, que define como os dados enviados pelo cliente devem ser estruturados.
    - name: Um campo obrigatório do tipo str.
    - description: Um campo opcional do tipo str.
    - price: Um campo obrigatório do tipo float.
    - tax: Um campo opcional do tipo float.
3. **@app.post("/items/")**: Define uma rota POST para criar um novo item.
4. **def create_item(item: Item)**: A função create_item recebe um objeto item que é automaticamente validado pelo FastAPI com base na classe Item.
5. **return {"name": item.name, "price": item.price}**: Retorna o nome e o preço do item recebido.

Testando:

Usando **Postman** ou **cURL**, envie uma requisição POST para http://127.0.0.1:8000/items/ com o seguinte corpo:

json

CopiarEditar

```
{
  "name": "Produto X",
  "price": 50.0
}
```

Resposta esperada:

json

CopiarEditar

```
{
   "name": "Produto X",
   "price": 50.0
}
```

Passo 6: Validação de Dados com Pydantic

O FastAPI valida automaticamente os dados usando **Pydantic**. Podemos adicionar validações personalizadas no modelo.

python

CopiarEditar

```
from pydantic import BaseModel, validator


class Item(BaseModel):
   name: str
   description: str = None
   price: float
   tax: float = None


   # Validação personalizada para garantir que o preço seja positivo
```

```
@validator('price')
def check_price_positive(cls, value):
    if value <= 0:
        raise ValueError('O preço deve ser maior que zero.')
    return value
```

Explicação:

1. **@validator('price')**: O decorador @validator permite adicionar validações personalizadas para os campos do modelo.
2. **check_price_positive**: Verifica se o valor do preço é maior que zero. Caso contrário, levanta um erro com a mensagem O preço deve ser maior que zero.

Agora, se você tentar enviar um preço negativo ou zero, o FastAPI responderá com um erro de validação:

json

CopiarEditar

```
{
  "detail": [
    {
      "loc": ["body", "price"],
      "msg": "O preço deve ser maior que zero.",
      "type": "value_error"
    }
  ]
}
```

Passo 7: Documentação Automática

O **FastAPI** gera automaticamente documentação para sua API utilizando o **Swagger UI**.

- Acesse http://127.0.0.1:8000/docs para a interface interativa do Swagger, onde você pode testar todos os endpoints diretamente.

- Acesse http://127.0.0.1:8000/redoc para a documentação alternativa gerada pelo **ReDoc**.

A documentação é gerada automaticamente com base nos tipos de dados e nas rotas que você definiu.

Passo 8: Tratamento de Erros

Você pode customizar a resposta para erros utilizando a classe HTTPException.

python

CopiarEditar

```
from fastapi import HTTPException


@app.get("/items/{item_id}")
def read_item(item_id: int):
  if item_id != 42:
    raise HTTPException(status_code=404, detail="Item não encontrado")
  return {"item_id": item_id}
```

Explicação:

1. **HTTPException**: Lança uma exceção HTTP personalizada, permitindo definir o código de status HTTP e a mensagem de erro.
2. **raise HTTPException(status_code=404, detail="Item não encontrado")**: Caso o item_id não seja 42, um erro 404 é retornado.

Passo 9: Conectar com Banco de Dados

Agora, vamos conectar a aplicação a um banco de dados usando **SQLAlchemy**.

1. Instale o **SQLAlchemy**:

bash

CopiarEditar

```
pip install sqlalchemy
```

2. Crie a conexão com o banco de dados e defina o modelo de dados.

python

CopiarEditar

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./test.db"

Base = declarative_base()

class ItemDB(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    description = Column(String, index=True)
    price = Column(Integer)

# Criar banco de dados
engine = create_engine(DATABASE_URL)
Base.metadata.create_all(bind=engine)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Neste exemplo, estamos criando um banco de dados SQLite e um modelo de dados ItemDB.

3. Vamos criar um endpoint POST para salvar itens no banco de dados.

python

CopiarEditar

@app.post("/items/")

def create_item(item: Item, db: Session = Depends(get_db)):

  db_item = ItemDB(name=item.name, description=item.description, price=item.price)

  db.add(db_item)

  db.commit()

  db.refresh(db_item)

  return db_item

Conclusão

Agora você tem uma API completa em Python usando o FastAPI. Você aprendeu a:

- Criar endpoints GET e POST.
- Validar dados com Pydantic.
- Adicionar tratamento de erros.
- Conectar com um banco de dados usando SQLAlchemy.

Além disso, a documentação da API é gerada automaticamente, o que facilita a interação e os testes da API.