

SUMÁRIO

O QUE VEM POR AÍ?	3
HANDS ON	4
SAIBA MAIS.....	5
O QUE VOCÊ VIU NESTA AULA?	10
REFERÊNCIAS.....	11

EMSE

O QUE VEM POR AÍ?

Agora que já sabemos programar, vamos entender como criar uma interface de programação de aplicações, também conhecida como API. Abordaremos os dois frameworks mais utilizados para o desenvolvimento destas aplicações: Flask e FastAPI.

Lembrando que as APIs conectam soluções e serviços sem a necessidade de saber como os elementos internos foram implementados.

HANDS ON

Nesta aula prática, demonstraremos como criar e publicar uma biblioteca no PyPI. Também exploraremos a importância dessa prática para a organização e a reutilização de código, além de aprender a criar nossos próprios módulos e bibliotecas de maneira eficiente.



SAIBA MAIS

Agora que passamos pelos conceitos básicos do desenvolvimento em Python, vamos avançar em um dos conceitos que podem nos ajudar no nosso dia a dia, a criação de uma API.

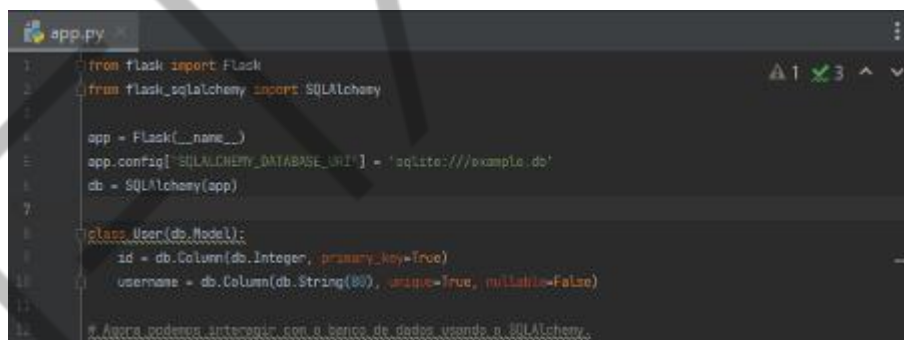
Para isso existem algumas bibliotecas em Python, mas nesta aula abordaremos duas das mais utilizadas: o Flask e o FastAPI.

Iniciando pelo Flask, ele é conhecido por sua simplicidade e flexibilidade, mas também oferece recursos avançados para devs experientes. Vamos explorar alguns desses conceitos.

Extensões Flask

As extensões no Flask são módulos adicionais que estendem as funcionalidades do framework. Vamos considerar uma extensão popular, o “Flask_SQLAlchemy”, que facilita a integração com bancos de dados.

Exemplo:

A screenshot of a code editor window titled 'app.py'. The code is in Python and demonstrates the setup of a Flask application with a database using SQLAlchemy. The code includes imports for Flask and SQLAlchemy, the creation of a Flask app, configuration of the database URI, and the creation of a SQLAlchemy database instance. It also shows a class definition for a 'User' model with columns for 'id' and 'username'.

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
6 db = SQLAlchemy(app)
7
8 class User(db.Model):
9     id = db.Column(db.Integer, primary_key=True)
10    username = db.Column(db.String(80), unique=True, nullable=False)
11
12 # Agora podemos interagir com o banco de dados usando o SQLAlchemy.
```

Figura 1 – Exemplo de código-fonte Python (Flask integração com banco de dados)
Fonte: Elaborado pelo autor (2024)

Interpretação

No código anterior importamos a classe “Flask” do Flask e a classe “SQLAlchemy” do Flask_SQLAlchemy. O Flask_SQLAlchemy é uma extensão que simplifica a integração do SQLAlchemy com o Flask, facilitando a interação com bancos de dados.

Um objeto Flask chamado app é criado. O primeiro argumento passado para o construtor é “__name__”, que representa o nome do módulo ou pacote principal. Isso é necessário para que o Flask saiba onde procurar por templates e arquivos estáticos.

A configuração do aplicativo é ajustada para definir a URL do banco de dados. Neste caso, estamos usando o “SQLite” e o banco de dados será um arquivo chamado “example.db” na mesma pasta do script.

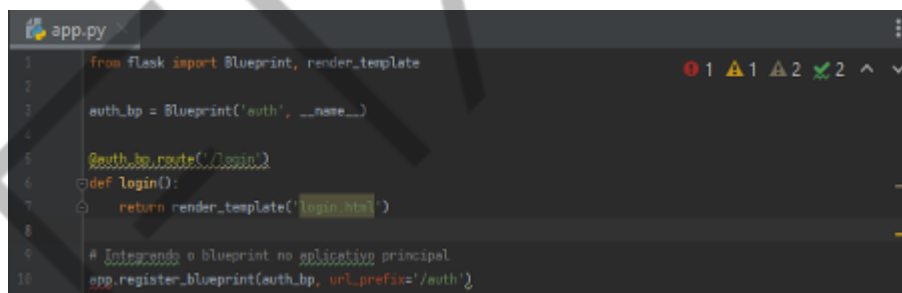
Um objeto chamado “db” é criado e associado ao aplicativo Flask. Isso configura a extensão para que ela saiba como interagir com ele.

Uma classe “User” é definida, que herda da classe “db.Model” do SQLAlchemy. Esta classe representa uma tabela no banco de dados. No exemplo, a tabela terá uma coluna “id” como chave primária e uma coluna “username” que deve ser única e não pode ser nula.

Blueprints e Organização de Código

Para projetos maiores, organizar o código de maneira eficiente é crucial. Os blueprints no Flask ajudam a estruturar o aplicativo.

Exemplo:

A screenshot of a code editor window titled 'app.py'. The code defines a Flask blueprint named 'auth_bp' and registers it with the main application. The code is as follows:

```
1 from flask import Blueprint, render_template
2
3 auth_bp = Blueprint('auth', __name__)
4
5 @auth_bp.route('/login')
6 def login():
7     return render_template('login.html')
8
9 # Integrando o blueprint no aplicativo principal
10 app.register_blueprint(auth_bp, url_prefix='/auth')
```

Figura 2 – Exemplo de código-fonte Python (Flask com blueprints)
Fonte: Elaborado pelo autor (2024)

Interpretação

O código importa duas classes do Flask: “Blueprint” e “render_template”. Assim, Blueprint é uma maneira de organizar um conjunto de rotas relacionadas e views em um aplicativo Flask.

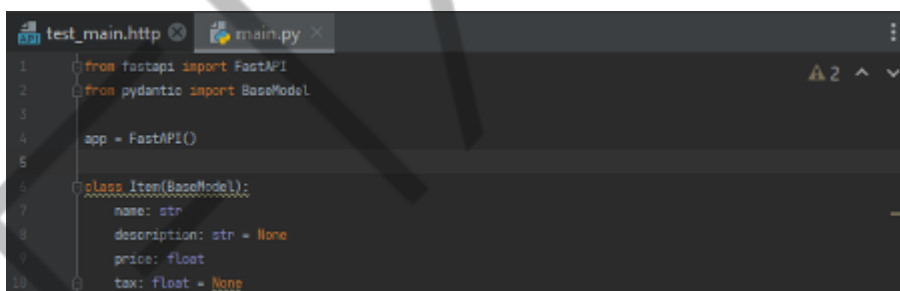
Um objeto Blueprint chamado “auth_bp” é criado. O primeiro argumento é o nome do blueprint e o segundo é o nome do módulo ou pacote. O blueprint é utilizado para agrupar rotas e views relacionadas.

Aqui, uma rota é definida para o caminho “/login”. Quando alguém acessa essa rota no navegador, a função login é executada. Neste caso, a função simplesmente retorna o resultado da função “render_template”, que renderiza o modelo HTML chamado “login.html”. Isso geralmente significa que, ao acessar a rota “/login”, será exibida uma página HTML contida no arquivo “login.html”.

Aqui, o blueprint “auth_bp” é registrado no aplicativo Flask principal (app). A opção url_prefix=’/auth’ significa que todas as rotas definidas no blueprint terão o prefixo “/auth”. Portanto, a rota “/login” do blueprint será acessível através de “/auth/login” no aplicativo principal.

Já o FastAPI, construído sobre o Starlette e o Pydantic, oferece uma abordagem moderna para o desenvolvimento de APIs. Para quem não conhece, o Pydantic é integrado ao FastAPI para a validação automática de dados.

Exemplo:

A screenshot of a code editor with two tabs: 'test_main.http' and 'main.py'. The 'main.py' tab is active, showing the following Python code:

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 app = FastAPI()
5
6 class Item(BaseModel):
7     name: str
8     description: str = None
9     price: float
10    tax: float = None
```

Figura 3 – Exemplo de código-fonte Python (FastAPI com PyDantic)

Fonte: Elaborado pelo autor (2024)

Neste exemplo nós importamos as classes “FastAPI” e “BaseModel” do FastAPI e Pydantic, respectivamente.

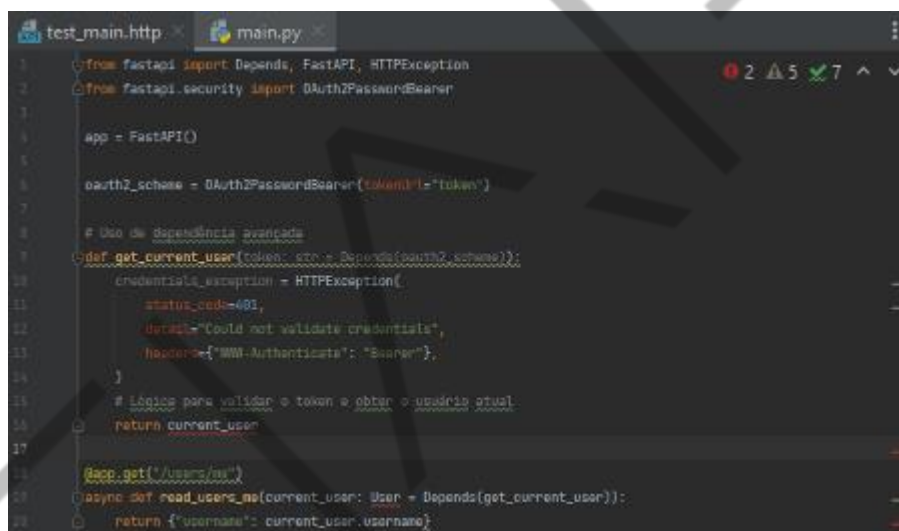
Há a criação de uma instância do aplicativo FastAPI chamada “app” e a definição de uma classe chamada “Item” que herda de BaseModel do Pydantic.

Esta classe é usada para representar a estrutura de dados esperada para os itens manipulados pela API. Cada atributo da classe (name, description, price, tax) corresponde a um campo de dados associado a um item:

- **name:** um campo obrigatório do tipo string.
- **description:** um campo opcional do tipo string (o padrão é None se não fornecido).
- **price:** um campo obrigatório do tipo float.
- **tax:** um campo opcional do tipo float (o padrão é None se não fornecido).

Essa classe é usada para definir automaticamente a validação de dados para as entradas da API. O FastAPI permite o uso de dependências avançadas para organizar o código e adicionar funcionalidades.

Exemplo:



```
1 from fastapi import Depends, FastAPI, HTTPException
2 from fastapi.security import OAuth2PasswordBearer
3
4 app = FastAPI()
5
6 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/token")
7
8 # Uso de dependência avançada
9 def get_current_user(token: str = Depends(oauth2_scheme)):
10     credentials_exception = HTTPException(
11         status_code=401,
12         detail="Could not validate credentials",
13         headers={"WWW-Authenticate": "Bearer"},
14     )
15     # Lógica para validar o token e obter o usuário atual
16     return current_user
17
18 @app.get("/users/me")
19 @async def read_users_me(current_user: User = Depends(get_current_user)):
20     return {"username": current_user.username}
```

Figura 4 – Exemplo de código-fonte Python (FastAPI com autenticação por token)

Fonte: Elaborado pelo autor (2024)

No código anterior, importamos as classes necessárias do FastAPI, incluindo “Depends” para dependências, FastAPI para criar o aplicativo e “HTTPException” para lidar com exceções HTTP. Também importamos “OAuth2PasswordBearer” para autenticação via “OAuth2”.

Um objeto FastAPI chamado “app” é criado. Este é o aplicativo principal que conterá as rotas e funcionalidades da API. Além disso, um esquema OAuth2 é configurado usando a classe OAuth2PasswordBearer.

Este esquema é usado para autenticação por senha, em que clientes podem obter um token de acesso usando um nome de usuário e senha, e o URL para obter o token é definido como `"/token"`.

Uma dependência avançada chamada `"get_current_user"` é definida. Esta dependência usa o esquema OAuth2 definido anteriormente para obter o token de autenticação. A lógica para validar o token e obter o usuário atual deve ser implementada.

Uma rota é definida em `"/users/me"`, que depende da função `get_current_user`. Esta rota está protegida por autenticação e somente usuários autenticados podem acessá-la. O usuário autenticado é passado como um parâmetro chamado `"current_user"`, que é do tipo `"User"` (presumidamente, uma classe de modelo definida em outro lugar no código).

Agora que conhecemos o básico sobre o Flask e o FastAPI, te convido a reassistir nossa videoaula, onde pudemos aprender a como criar uma API utilizando o FastAPI.

O QUE VOCÊ VIU NESTA AULA?

Nessa aula, conhecemos dois frameworks muito utilizados na construção de APIs (Flask e FastAPI) e alguns conceitos avançados, como a integração com banco de dados.

EXEMPLO

REFERÊNCIAS

ADRIANO, T. S. **Create a RESTful API with Python 3 and FastAPI**. 2024. Disponível em: <<https://dev.to/programadriano/create-a-restful-api-with-python-and-fastapi-4h61>>. Acesso em: 15 ago. 2024.

GRINBERG, M. **Flask Web Development**: Developing Web Applications with Python. [s.l.]: O'Reilly Media, 2018.

TITUS, S. **FastAPI**: The complete guide. [s.l.]: Packt Publishing, 2021.

EMANUELO

PALAVRAS-CHAVE

Palavras-chave: Flask. FastAPI. API.

EMSE



POSTECH