Delivery API 

O que é o projeto?

Faremos uma API com Flask, para um serviço de delivery, onde teremos o CRUD de pedido que contém:

- id
- cliente
- produto
- valor

O que é uma API?

Uma API (Interface de Programação de Aplicações) é como uma "ponte" que permite que diferentes programas ou sistemas "conversem" entre si e troquem informações.

Imagine assim:

Suponha que você vá a um restaurante. O menu representa os serviços disponíveis, e você escolhe o que quer. O garçom é como uma API. Você não entra na cozinha nem precisa saber como a comida é feita.

O garçom pega seu pedido e leva para a cozinha, onde os cozinheiros (o sistema por trás) preparam a comida. Depois, o garçom traz sua refeição pronta. Você apenas faz o pedido e recebe o que quer, sem se preocupar com os detalhes do processo.

Na tecnologia, funciona de forma semelhante:

- A API permite que um programa peça informações ou serviços de outro.
- Por exemplo, um aplicativo de clima pode usar a API de um serviço meteorológico para obter a previsão do tempo.
- O aplicativo faz um "pedido" à API, e a API "responde" com os dados (como a previsão).

Benefícios:

- Facilita a comunicação entre diferentes softwares ou serviços, sem que o usuário ou desenvolvedor precise entender os detalhes técnicos do sistema que está sendo acessado.
- As APIs podem ser usadas para conectar tudo, desde sites e aplicativos móveis até dispositivos e serviços online.

Em resumo, uma API é uma maneira fácil para diferentes programas ou sistemas trocarem informações e funcionalidades entre si, como um garçom que pega pedidos e entrega resultados!

O que é um CRUD?

CRUD é um acrônimo que se refere às quatro operações básicas usadas em sistemas de gerenciamento de dados, especialmente em bancos de dados e APIs. Essas operações são:

- Create (Criar): Inserção de novos dados no sistema, como adicionar um novo registro em um banco de dados.
- Read (Ler): Recuperação de dados existentes. Isso inclui a busca e visualização de informações armazenadas.
- Update (Atualizar): Modificação de dados já existentes. Pode ser a alteração parcial ou completa de um registro.
- Delete (Excluir): Remoção de dados do sistema.

Essas operações são a base para qualquer sistema que gerencie dados persistentes, seja um banco de dados relacional, uma API REST ou uma aplicação web.

Preparando o Ambiente

Você precisa ter instalado em seu computador:

- Python 3, com o pip (instalador de pacotes) e o venv (ambientes virtuais)
- MySQL (tutorial de instalação e configuração)

Na pasta do seu projeto crie um novo ambiente virtual, com o comando:

```
python -m venv venv
```

Ative o ambiente virtual, com o comando (para Windows):

```
venv\Scripts\activate
```

Para Linux:

```
source venv/bin/activate
```

Com o ambiente virtual ativado, instale o Flask com o comando:

```
pip install Flask
```

Código inicial

Na pasta do projeto, crie um arquivo chamado main.py e adicione o código a seguir:

```
from flask import Flask, request, Response

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

app.config['DEBUG'] = True
app.run()
```

O que esse código faz?

- Importa a classe Flask do módulo flask. O Flask é um framework leve para construir aplicações web em Python. Ele facilita o processo de lidar com requisições HTTP e rotas de URLs.

- Cria uma instância chamada app da classe Flask. O __name__ é uma variável especial em Python que se refere ao nome do módulo atual (arquivo). Ele diz ao Flask onde procurar recursos como templates e arquivos estáticos.

- Define uma rota para a aplicação (“/”). Quando o navegador acessar a página principal (/), ele executará a função abaixo deste decorator. O @app.route('/') é um decorator que vincula a função logo abaixo dele a uma URL específica. Nesse caso, a URL é '/', que representa a página raiz do seu site. O Flask utiliza decoradores para mapear URLs às funções que devem ser executadas quando essa URL é acessada.

- Define uma função chamada hello_world(). la será executada sempre que alguém acessar a URL definida no decorator ('/'). A linha return 'Hello World!' retorna uma string 'Hello World!'. O Flask automaticamente transforma essa string em uma resposta HTTP que será enviada de volta ao cliente (navegador ou outra ferramenta).

- Coloca a aplicação em modo debug. Isso significa que a aplicação será executada em modo de depuração. No modo de depuração a aplicação reinicia automaticamente quando você faz alterações no código e mensagens detalhadas de erro serão mostradas no navegador em caso de falhas.
  Objetivo: Facilitar o desenvolvimento, permitindo que o código seja testado e modificado rapidamente com melhor feedback de erros.

- Chama o método run() que inicia o servidor web integrado do Flask, permitindo que a aplicação seja acessada em um navegador. O servidor ficará aguardando por requisições até ser interrompido. Por padrão, ela estará acessível no endereço http://127.0.0.1:5000/ (localhost na porta 5000).

Como rodar esse código?

- Salve o código.
- Abra o terminal e rode o comando:
  ```
  python main.py
  ```
- Abra seu navegador no endereço indicado, e voilá! Temos a mensagem de Hello World!

Construindo a API

Configurando o banco de dados

Nossa API utilizará um banco de dados MySQL. Então precisamos criar um banco chamado delivery. E uma tabela nesse banco com as informações do pedido. Para isso, em seu terminal, acesse o MySQL com o comando:

```
mysql -u root -p
```

Informe a senha do usuário root.

E no terminal do MySQL, rode o código a seguir:

```
CREATE DATABASE delivery;

USE delivery;

CREATE TABLE pedido (
    id INT NOT NULL AUTO_INCREMENT,
    cliente VARCHAR(100) NOT NULL,
    produto VARCHAR(100) NOT NULL,
    valor FLOAT NOT NULL,

    CONSTRAINT pedido PRIMARY KEY(id)

)ENGINE = InnoDB AUTO_INCREMENT = 1;
```

Para saber mais sobre SQL, você pode fazer os cursos da formação Consultas com MySQL.

Agora, vamos instalar as bibliotecas necessárias para lidar com o banco de dados na API. Em seu terminal rode os comandos a seguir:

```
pip install mysql-connector-python
pip install flask-sqlalchemy
pip install flask-marshmallow
pip install marshmallow-sqlalchemy
```

Para que serve cada uma dessas bibliotecas?

- **mysql-connector-python**: É uma biblioteca que permite que o Python se conecte e interaja com um banco de dados MySQL. Ela fornece uma interface para executar consultas, inserir e manipular dados, e trabalhar diretamente com MySQL em projetos Python.

- **flask-sqlalchemy:** Integra o SQLAlchemy, uma biblioteca de mapeamento objeto-relacional (ORM), com o Flask. Ele facilita o uso de bancos de dados relacionais dentro de aplicações Flask, permitindo que você trabalhe com o banco de dados usando classes e objetos Python, em vez de escrever SQL diretamente.

- **flask-marshmallow:** É uma biblioteca que integra o Marshmallow (uma ferramenta de serialização/deserialização de dados) com o Flask. Ela ajuda a converter objetos Python (como resultados de consultas) em formatos como JSON, o que é útil para APIs, e vice-versa, transformando dados recebidos em objetos Python.

- **marshmallow-sqlalchemy:** Expande o Marshmallow para funcionar diretamente com SQLAlchemy. Ele facilita a serialização (transformação de objetos em JSON, por exemplo) e a desserialização (transformação de dados JSON em objetos) de modelos SQLAlchemy, o que é muito útil em APIs que usam bancos de dados.

Integrando o banco de dados ao código

Importe as bibliotecas de bancos de dados no código, conforme mostrado a seguir. Também importe a biblioteca json, pois utilizaremos esse formato de dados para lidar com as informações em nossa API.

```
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
```

```
import json
```

```

Apague o trecho de código que traz a rota e a função de hello_world().

Faça a configuração da aplicação para se conectar a um banco de dados usando SQLAlchemy e Marshmallow:

```
app.config['SQLALCHEMY_DATABASE_URI'] = \
    '{SGBD}://{usuario}:{senha}@{servidor}/{database}'.format(
        SGBD = 'mysql+mysqlconnector',
        usuario = 'root',
        senha = 'root',
        servidor = 'localhost',
        database = 'delivery'
)
db = SQLAlchemy(app)
ma = Marshmallow(app)
```

Esse código:

- Define  a URL de conexão com o banco de dados para o SQLAlchemy. O Flask utiliza essa configuração para saber como e onde se conectar ao banco de dados.

- A URL de conexão tem o formato: `{SGBD}://{usuario}:{senha}@{servidor}/{database}`, onde:

    - **SGBD:** Especifica o Sistema de Gerenciamento de Banco de Dados que será utilizado. Neste caso, mysql+mysqlconnector, que indica o uso do MySQL com o conector mysql-connector-python.
    - **usuario:** O nome de usuário para acessar o banco de dados (aqui é root).
    - **senha:** A senha correspondente ao usuário (aqui também é root).
    - **servidor:** O endereço do servidor onde o banco de dados está hospedado (aqui é localhost, ou seja, o banco de dados está rodando localmente).
    - **database:**O nome do banco de dados que será usado (neste caso, delivery)

- **`db = SQLAlchemy(app)`**: Cria uma instância do SQLAlchemy, associando-a à aplicação Flask.

- **`ma = Marshmallow(app)`**: Cria uma instância do Marshmallow associada à aplicação Flask.
```

Em seguida, vamos definir um modelo de banco de dados e um esquema de serialização para um Pedido.

Primeiro, crie uma classe que define o modelo de banco de dados para um Pedido, que será armazenado em uma tabela no banco de dados:

```
class Pedido(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    cliente = db.Column(db.String(100), nullable=False)
    produto = db.Column(db.String(100), nullable=False)
    valor = db.Column(db.Float, nullable=False)
```

Essa classe utiliza o db.Model, que é uma classe base fornecida pelo SQLAlchemy que indica que a classe Pedido é um modelo de banco de dados. Cada instância da classe representa uma linha (registro) na tabela do banco.

Nessa classe temos os atributos id, cliente, produto e valor. Cada um desses atributos representa uma coluna na tabela Pedido do banco de dados.

Depois, crie uma classe que define o esquema de dados do Pedido, conforme mostrado a seguir:

```
class PedidoSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = Pedido
        load_instance = True
        sqla_session = db.session
```

A classe `PedidoSchema` usa o Marshmallow para definir como os dados do modelo `Pedido` serão **serializados** e **desserializados** (convertidos para e a partir de formatos como JSON).

A classe ma.SQLAlchemyAutoSchema é uma classe base fornecida pelo Marshmallow que automaticamente gera esquemas de serialização/deserialização com base em um modelo SQLAlchemy.

`class Meta:`: Dentro da `Meta` class, você define configurações para o esquema.

- **`model = Pedido`**: Indica que o esquema será baseado no modelo `Pedido`. Assim, ele saberá que as colunas `id`, `cliente`, `produto` e `valor` devem ser incluídas no esquema.
- **`load_instance = True`**: Informa ao Marshmallow que ele deve retornar uma instância do modelo `Pedido` quando desserializar os dados (como ao criar um pedido a partir de um JSON recebido em uma API).

- **`sqla_session = db.session`**: Define que a sessão do SQLAlchemy usada será `db.session`. Isso é necessário para que o Marshmallow possa interagir corretamente com o banco de dados, especialmente ao salvar ou modificar registros.

Criando endpoints na API

Agora vamos construir os endpoints da API.

O que é um endpoint?

Um endpoint é um ponto de acesso em uma API onde um cliente (como um navegador ou outro programa) pode enviar solicitações e receber respostas. Basicamente, é o endereço (ou URL) de uma API que permite a comunicação entre o cliente e o servidor.

Endpoint GET de todos os pedidos

Adicione o trecho de código a seguir, que define uma rota que retorna uma lista de pedidos em formato JSON:

```
@app.route('/')
def lista_pedidos():
    pedidos = Pedido.query.all()
    pedidos_schema = PedidoSchema(many=True)
    return pedidos_schema.dump(pedidos)
```

Análise:

- **`@app.route('/')`**: Define uma rota na URL raiz (`/`) da aplicação. Quando alguém acessa essa URL, a função `lista_pedidos()` é chamada.

- **`def lista_pedidos():`**: Define a função que será executada quando a rota for acessada.

- **`pedidos = Pedido.query.all()`**: Usa o SQLAlchemy para consultar e recuperar todos os registros da tabela `pedido` no banco de dados, armazenando-os na variável `pedidos`.

- **`pedidos_schema = PedidoSchema(many=True)`**: Cria uma instância do esquema `PedidoSchema`, configurada para processar múltiplas instâncias (muitos pedidos).

- **`return pedidos_schema.dump(pedidos)`**: Serializa os objetos `pedidos` em formato JSON usando o esquema `PedidoSchema` e retorna essa lista JSON como resposta à requisição.

Endpoint POST de um pedido

Adicione o trecho de código a seguir, que define uma rota em uma aplicação Flask para criar um novo pedido:

```
@app.route('/criapedido', methods=['POST'])
def cria_pedido():
    cliente = request.json['cliente']
    produto = request.json['produto']
    valor = request.json['valor']

    novo_pedido = Pedido(cliente=cliente, produto=produto, valor=valor)
    db.session.add(novo_pedido)
    db.session.commit()

    pedido_schema = PedidoSchema()
    return pedido_schema.dump(novo_pedido)
```

Análise:

- **`@app.route('/criapedido', methods=['POST'])`**: Define uma rota para a URL `/criapedido`, permitindo apenas requisições do tipo **POST**. Essa rota é usada para criar novos pedidos.

- **`def cria_pedido():`**:Define a função que será chamada quando a rota `/criapedido` for acessada.

- **`cliente = request.json['cliente']`**: Extrai o valor do campo `cliente` do corpo da requisição JSON. O `request.json` contém os dados enviados pelo cliente.

- **`produto = request.json['produto']`**: Extrai o valor do campo `produto` da requisição JSON.

- **`valor = request.json['valor']`**:Extrai o valor do campo `valor` da requisição JSON.

- **`novo_pedido = Pedido(cliente=cliente, produto=produto, valor=valor)`**: Cria uma nova instância do modelo `Pedido`, passando os dados extraídos (cliente, produto e valor).

- **`db.session.add(novo_pedido)`**:Adiciona o novo pedido à sessão do banco de dados, preparando-o para ser salvo.

- **`db.session.commit()`**:Executa a transação e salva o novo pedido no banco de dados.

- **`pedido_schema = PedidoSchema()`**:Cria uma instância do esquema `PedidoSchema` para serializar o novo pedido.

- **`return pedido_schema.dump(novo_pedido)`**: Serializa o objeto `novo_pedido` em formato JSON e o retorna como resposta à requisição.

Endpoint GET de um pedido

Adicione o trecho de código a seguir, que define uma rota para ler um pedido específico a partir de seu ID:

```
@app.route('/pedido/<int:id>', methods=['GET'])
def le_pedido(id):
    pedido = Pedido.query.filter(Pedido.id == id).one_or_none()

    if pedido is not None:
        pedido_schema = PedidoSchema()
        return pedido_schema.dump(pedido)
    else:
            return Response(json.dumps({'Erro': 'Nao encontramos um
pedido!'}), status=404)
```

Análise:

- **`@app.route('/pedido/<int:id>', methods=['GET'])`**: Define uma rota para a URL `/pedido/<id>`, onde `<id>` é um parâmetro de URL que deve ser um número inteiro (`int`). Essa rota permite apenas requisições do tipo **GET**.

- **`def le_pedido(id):`**:Define a função que será chamada quando a rota correspondente for acessada. O parâmetro `id` é passado para a função e representa o ID do pedido que se deseja consultar.

- **`pedido = Pedido.query.filter(Pedido.id == id).one_or_none()`**: Usa o SQLAlchemy para consultar o banco de dados e tentar encontrar um pedido com o ID fornecido. O método `one_or_none()` retorna o pedido se ele existir ou `None` se não for encontrado.

- **`if pedido is not None:`**: Verifica se um pedido foi encontrado.
    - **Caso um pedido seja encontrado:**
        - **`pedido_schema = PedidoSchema()`**: Cria uma instância do esquema `PedidoSchema` para serializar o pedido.
        - **`return pedido_schema.dump(pedido)`**: Serializa o objeto `pedido` em formato JSON e o retorna como resposta.
    - **Caso não seja encontrado um pedido:**

- **`return Response(json.dumps({'Erro': 'Nao encontramos um pedido!'}), status=404)`**: Retorna uma resposta JSON com uma mensagem de erro e um status HTTP 404 (não encontrado).

Endpoint UPDATE de um pedido

Adicione o trecho de código a seguir, que define uma rota para atualizar um pedido específico a partir de seu ID:

```
@app.route('/atualizapedido/<int:id>', methods=['PUT'])
def atualiza_pedido(id):
    pedido = Pedido.query.filter(Pedido.id == id).one_or_none()

    if pedido is not None:
        pedido.cliente = request.json['cliente']
        pedido.produto = request.json['produto']
        pedido.valor = request.json['valor']

        db.session.merge(pedido)
        db.session.commit()

        pedido_schema = PedidoSchema()
        return pedido_schema.dump(pedido), 201
    else:
            return Response(json.dumps({'Erro': 'Nao encontramos um
pedido!'}), status=404)
```

Análise:

- **`@app.route('/atualizapedido/<int:id>', methods=['PUT'])`**: Define uma rota para a URL `/atualizapedido/<id>`, onde `<id>` é um parâmetro de URL que deve ser um número inteiro (`int`). Essa rota permite apenas requisições do tipo **PUT**, que geralmente são usadas para atualizar recursos.

- **`def atualiza_pedido(id):`**: Define a função que será chamada quando a rota correspondente for acessada. O parâmetro `id` é passado para a função e representa o ID do pedido que se deseja atualizar.

- **`pedido = Pedido.query.filter(Pedido.id == id).one_or_none()`**: Usa o SQLAlchemy para consultar o banco de dados e tentar encontrar um pedido com o ID fornecido. O método `one_or_none()` retorna o pedido se ele existir ou `None` se não for encontrado.

- `if pedido is not None:`: Verifica se um pedido foi encontrado.
    - **Caso um pedido seja encontrado:**
        - `pedido.cliente = request.json['cliente']`: Atualiza o campo `cliente` do pedido com o valor extraído do corpo da requisição JSON.
        - `pedido.produto = request.json['produto']`: Atualiza o campo `produto` do pedido.
        - `pedido.valor = request.json['valor']`: Atualiza o campo `valor` do pedido.
        - `db.session.merge(pedido)`: Este método é usado para garantir que a instância do pedido na sessão do banco de dados seja atualizada com os novos dados.
        - `db.session.commit()`: Executa a transação e salva as alterações no banco de dados.
        - `pedido_schema = PedidoSchema()`: Cria uma instância do esquema `PedidoSchema` para serializar o pedido atualizado.
        - `return pedido_schema.dump(pedido), 201`: Serializa o objeto `pedido` em formato JSON e retorna os dados atualizados com um status HTTP 201 (criado).
    - **Caso não seja encontrado um pedido:**
        - `return Response(json.dumps({'Erro': 'Nao encontramos um pedido!'}), status=404)`: Retorna uma resposta JSON com uma mensagem de erro e um status HTTP 404 (não encontrado).

Endpoint DELETE de um pedido

Adicione o trecho de código a seguir, que define uma rota para apagar um pedido específico a partir de seu ID:

```
@app.route('/apagapedido/<int:id>', methods=['DELETE'])
def apaga_padido(id):
    pedido = Pedido.query.filter(Pedido.id == id).one_or_none()

    if pedido is not None:
        db.session.delete(pedido)
        db.session.commit()
        pedido_schema = PedidoSchema()
        return pedido_schema.dump(pedido)
    else:
            return Response(json.dumps({'Erro': 'Nao encontramos um
pedido!'}), status=404)
```

Análise:

- `@app.route('/apagapedido/<int:id>', methods=['DELETE'])`: Define uma rota para a URL `/apagapedido/<id>`, onde `<id>` é um parâmetro de URL que deve ser um número inteiro (`int`). Essa rota permite apenas requisições do tipo **DELETE**, que geralmente são usadas para remover recursos.

- `def apaga_pedido(id):`: Define a função que será chamada quando a rota correspondente for acessada. O parâmetro `id` é passado para a função e representa o ID do pedido que se deseja apagar.

- `pedido = Pedido.query.filter(Pedido.id == id).one_or_none()`: Usa o SQLAlchemy para consultar o banco de dados e tentar encontrar um pedido com o ID fornecido. O método `one_or_none()` retorna o pedido se ele existir ou `None` se não for encontrado.
- `if pedido is not None:`: Verifica se um pedido foi encontrado.
    - **Caso um pedido seja encontrado:**
        - `db.session.delete(pedido)`: Remove o pedido da sessão do banco de dados, marcando-o para exclusão.
        - `db.session.commit()`: Executa a transação e confirma a remoção do pedido do banco de dados.
        - `pedido_schema = PedidoSchema()`: Cria uma instância do esquema `PedidoSchema` para serializar o pedido que foi excluído.
        - `return pedido_schema.dump(pedido)`: Serializa o objeto `pedido` em formato JSON e retorna os dados do pedido que foi removido.
    - **Caso não seja encontrado um pedido:**
        - `return Response(json.dumps({'Erro': 'Nao encontramos um pedido!'}), status=404)`: Retorna uma resposta JSON com uma mensagem de erro e um status HTTP 404 (não encontrado).

Testando a API

Utilize algum programa como o Postman para testar cada uma das rotas.

Você pode utilizar os exemplos JSON a seguir:

Exemplo 1:

```
{
  "cliente": "João Silva",
  "produto": "combo",
  "valor": 50.00
}
```

Exemplo 2:

```
{
  "cliente": "Maria Eduarda",
  "produto": "milkshake",
  "valor": 15.00
}
```

Materiais para estudo

- Formação Aprenda a programar em Python com Orientação a Objetos
- Formação Começando com Flask: framework web de Python