



## AULA 03

PUT, DELETE E INTRODUÇÃO A BANCO DE DADOS

# O QUE VEREMOS HOJE

**01** PUT/PATCH

**02** DELETE

**03** BANCO DE DADOS

**04** SQL

**05** CONEXÃO COM BANCO DE DADOS

**06** MODELS

# PUT

O método PUT é utilizado para atualizar um recurso existente no servidor.

Em APIs, isso significa modificar um item que já foi criado anteriormente.

O PUT geralmente é usado com um **identificador na URL**, como um ID, para indicar exatamente qual recurso será atualizado.




```
1 produtos = [  
2     {"id": 1, "nome": "Camisa", "preco": 49.90},  
3     {"id": 2, "nome": "Tênis", "preco": 199.90},  
4     {"id": 3, "nome": "Mochila", "preco": 129.90}  
5 ]  
6  
7 @app.route("/produtos/<int:id>", methods=["PUT"])  
8 def atualizar_produto(id):  
9     dados = request.json  
10  
11     for produto in produtos:  
12         if produto["id"] == id:  
13             produto["nome"] = dados.get("nome", produto["nome"])  
14             produto["preco"] = dados.get("preco", produto["preco"])  
15             return jsonify(produto), 200  
16  
17     return jsonify({"erro": "Produto não encontrado"}), 404  
18
```

# PATCH

O método PATCH também é utilizado para atualizar um recurso no servidor, assim como o PUT.

A principal diferença é que o PATCH realiza uma atualização parcial, ou seja, apenas modifica os campos enviados na requisição, sem substituir o objeto inteiro.

Enquanto o PUT espera que todos os campos sejam reenviados, o PATCH permite alterar apenas o necessário.



```
1
2 @app.route("/produtos/<int:id>", methods=["PATCH"])
3 def atualizar_parcial(id):
4     dados = request.json
5
6     for produto in produtos:
7         if produto["id"] == id:
8             if "nome" in dados:
9                 produto["nome"] = dados["nome"]
10            if "preco" in dados:
11                produto["preco"] = dados["preco"]
12            return jsonify(produto), 200
13
14    return jsonify({"erro": "Produto não encontrado"}), 404
```

# DELETE

O método DELETE é utilizado para remover um recurso do servidor. Em uma API, isso significa apagar permanentemente um item do sistema.

Assim como no PUT e no PATCH, o DELETE geralmente usa um identificador na URL (como o ID do item) para indicar qual recurso deve ser excluído.



```
1 @app.route("/produtos/<int:id>", methods=["DELETE"])
2 def deletar_produto(id):
3     for produto in produtos:
4         if produto["id"] == id:
5             produtos.remove(produto)
6             return jsonify({"mensagem": "Produto removido com sucesso"}), 200
7
8     return jsonify({"erro": "Produto não encontrado"}), 404
9
```

# ATIVIDADE PRÁTICA

Utilizando a lista de tarefas simuladas (com id, titulo, descrição e status), implemente duas novas rotas:

## **Rota PUT /tarefas/<id>**

- Atualize os campos titulo e/ou status da tarefa com aquele ID
- Valide se o campo titulo foi enviado
- Se faltar o título, retorne uma mensagem de erro com status 400
- Se o ID não existir, retorne uma mensagem de erro com status 404
- Se atualizar com sucesso, retorne os dados da tarefa atualizada com status 200

## **Rota DELETE /tarefas/<id>**

- Remova a tarefa com o ID informado
- Se encontrar e remover, retorne uma mensagem de confirmação com status 200
- Se não encontrar, retorne uma mensagem de erro com status 404

# BANCO DE DADOS

Um banco de dados é uma estrutura usada para armazenar informações de forma organizada, com o objetivo de facilitar o acesso, a busca e a atualização desses dados.

Ele é muito utilizado em sistemas e aplicações que precisam guardar dados de forma persistente, como cadastros de usuários, produtos, pedidos, mensagens, entre muitos outros.

Ao contrário de listas ou variáveis que usamos na memória do programa, os dados em um banco continuam salvos mesmo depois que o sistema é desligado.



# BANCO DE DADOS RELACIONAL

O banco de dados relacional organiza os dados em tabelas, parecidas com planilhas.

Cada tabela é formada por linhas (que representam os registros salvos) e colunas (que representam os campos de informação, como nome, data ou status).

Esse modelo é chamado de "relacional" porque as tabelas podem se relacionar entre si por meio de chaves e identificadores.

DEPARTAMENTO	
Departamento Id	Nome do Departamento
1	Recursos Humanos
2	TI
3	Financeiro

FUNCIONÁRIO		
Funcionario_Id	Nome do Funcionário	Departamento_Id
101	Alice	1
102	Bruno	1
103	Carla	2
104	Daniel	3



# SQL

SQL (Structured Query Language) é a linguagem usada para **criar, consultar, atualizar e apagar dados em um banco de dados relacional**.

Com SQL, é possível fazer comandos como:

- Inserir um novo registro em uma tabela
- Buscar dados filtrando por algum critério
- Atualizar informações de um item
- Remover um registro do banco

É uma linguagem padronizada e compatível com os principais bancos relacionais, como MySQL, PostgreSQL e SQLite.



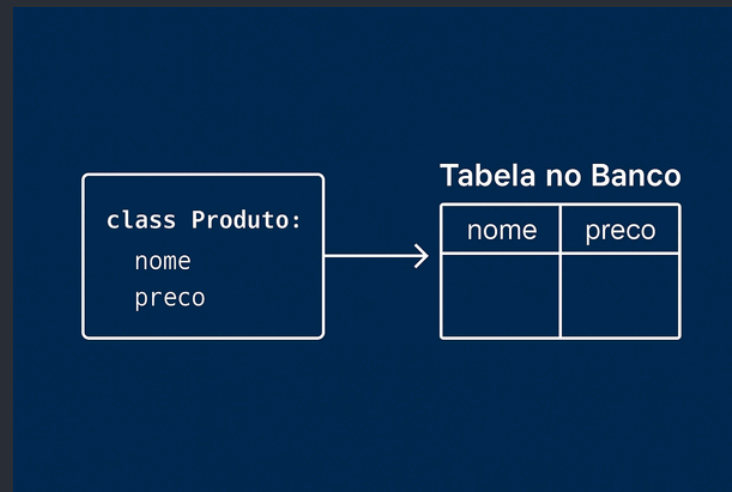
# FLASK E BANCO DE DADOS

O Flask não possui suporte nativo para bancos de dados, então utilizamos bibliotecas externas como o SQLAlchemy, um ORM (Object Relational Mapper).

Ele conecta objetos em Python a bancos de dados relacionais, permitindo que os desenvolvedores:

- Criem classes Python para representar tabelas
- Utilizem objetos para manipular dados (inserir, buscar, atualizar, deletar)

O ORM converte objetos em registros no banco e vice-versa, facilitando a interação com os dados.




# CONEXÃO COM O BANCO DE DADOS

Para conectar o Flask a um banco de dados, usamos o SQLAlchemy, que precisa ser instalado no projeto.

Além disso, é necessário instalar o driver específico do banco que será utilizado.

A configuração é feita no início do código, onde informamos a URL do banco e criamos uma instância da extensão.

A depender do banco de dados utilizado, pode ser necessário criar manualmente antes da conexão.



```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://root:1234@localhost/tarefas_db'
6 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
7
8 db = SQLAlchemy(app)
9
```

# MODELS

Em uma aplicação Flask com banco de dados, chamamos de modelo (ou model) a classe que representa uma tabela no banco.

Cada modelo define:

- O nome da tabela
- Os campos (colunas) que essa tabela vai ter
- Os tipos de dados de cada campo

Ao criar um modelo com SQLAlchemy, essa estrutura em Python será convertida em uma tabela real no banco quando chamarmos **db.create\_all()**.

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://root:1234@localhost/tarefas_db'
6 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
7
8 db = SQLAlchemy(app)
9
10 class Produto(db.Model):
11     id = db.Column(db.Integer, primary_key=True)
12     nome = db.Column(db.String(100), nullable=False)
13     preco = db.Column(db.Float, nullable=False)
14
15
16
```

# TESTE DA CONEXÃO E CRIAÇÃO DE TABELAS

Após definir um modelo, podemos criar a tabela correspondente no banco usando o comando **db.create\_all()**.

Para fins de teste, podemos usar o decorador **@app.before\_first\_request**, que garante que a criação aconteça automaticamente antes da primeira requisição recebida pela aplicação.

Também podemos criar uma rota simples para verificar se a conexão está funcionando corretamente.

```
1 from flask import Flask
2 from flask_sqlalchemy import SQLAlchemy
3 import config
4
5 app = Flask(__name__)
6 app.config.from_object(config)
7
8 db = SQLAlchemy(app)
9
10 class Produto(db.Model):
11     id = db.Column(db.Integer, primary_key=True)
12     nome = db.Column(db.String(100), nullable=False)
13     preco = db.Column(db.Float, nullable=False)
14
15 @app.before_first_request
16 def criar_tabelas():
17     db.create_all()
18
19 @app.route("/testar")
20 def testar_conexao():
21     return {"mensagem": "Conexão com o banco funcionando!"}
22
```