

# Estruturando o Projeto e Criando Rotas do CRUD

| <https://fastapidozero.dunossauro.com/02/>

# Objetivos dessa aula:

- Entendimento dos verbos HTTP, JSON e códigos de resposta
- Compreender a estrutura de um projeto FastAPI e como estruturar rotas CRUD (Criar, Ler, Atualizar, Deletar)
- Aprender sobre a biblioteca Pydantic e sua utilidade na validação e serialização de dados
- Implementação de rotas CRUD em FastAPI
- Escrita e execução de testes para validar o comportamento das rotas

# O que é uma API?

**Acrônimo de Application Programming Interface (Interface de Programação de Aplicações), uma API é um conjunto de regras e protocolos que permitem a comunicação entre diferentes softwares.**

**As APIs servem como uma ponte entre diferentes programas, permitindo que eles se comuniquem e compartilhem informações de maneira eficiente e segura.**

# A arquitetura

O que queremos dizer com cliente servidor



Existe uma aplicação que "Serve" e uma que é cliente de quem serve

# O que é HTTP?

**HTTP, ou Hypertext Transfer Protocol, é o protocolo fundamental na web para a transferência de dados e comunicação entre clientes e servidores.**

**Por exemplo, quando chamamos a rota *ou endpoint* `/` da nossa aplicação no teste**

```
def test_root_deve_retornar_200_e_ola_mundo():  
    client.get('/')
```

# Tipos de requisição HTTP

**Os verbos HTTP indicam a ação desejada a ser executada em um determinado recurso. Os verbos mais comuns são:**

- GET: Recupera recursos
- POST: Cria um novo recurso
- PUT: Atualiza um recurso existente
- PATCH: Atualiza parte de um recurso
- DELETE: Exclui um recurso

# Tipos de resposta HTTP

**As respostas dadas pela API no HTTP vem com códigos para explicar o que aconteceu**

- 200 OK: A solicitação foi bem-sucedida
- 201 Created: A solicitação foi bem-sucedida e um novo recurso foi criado
- 404 Not Found: O recurso solicitado não pôde ser encontrado

```
def test_root_deve_retornar_200_e_ola_mundo():  
    response = client.get('/')  
    assert response.status_code == 200
```

# Os dados

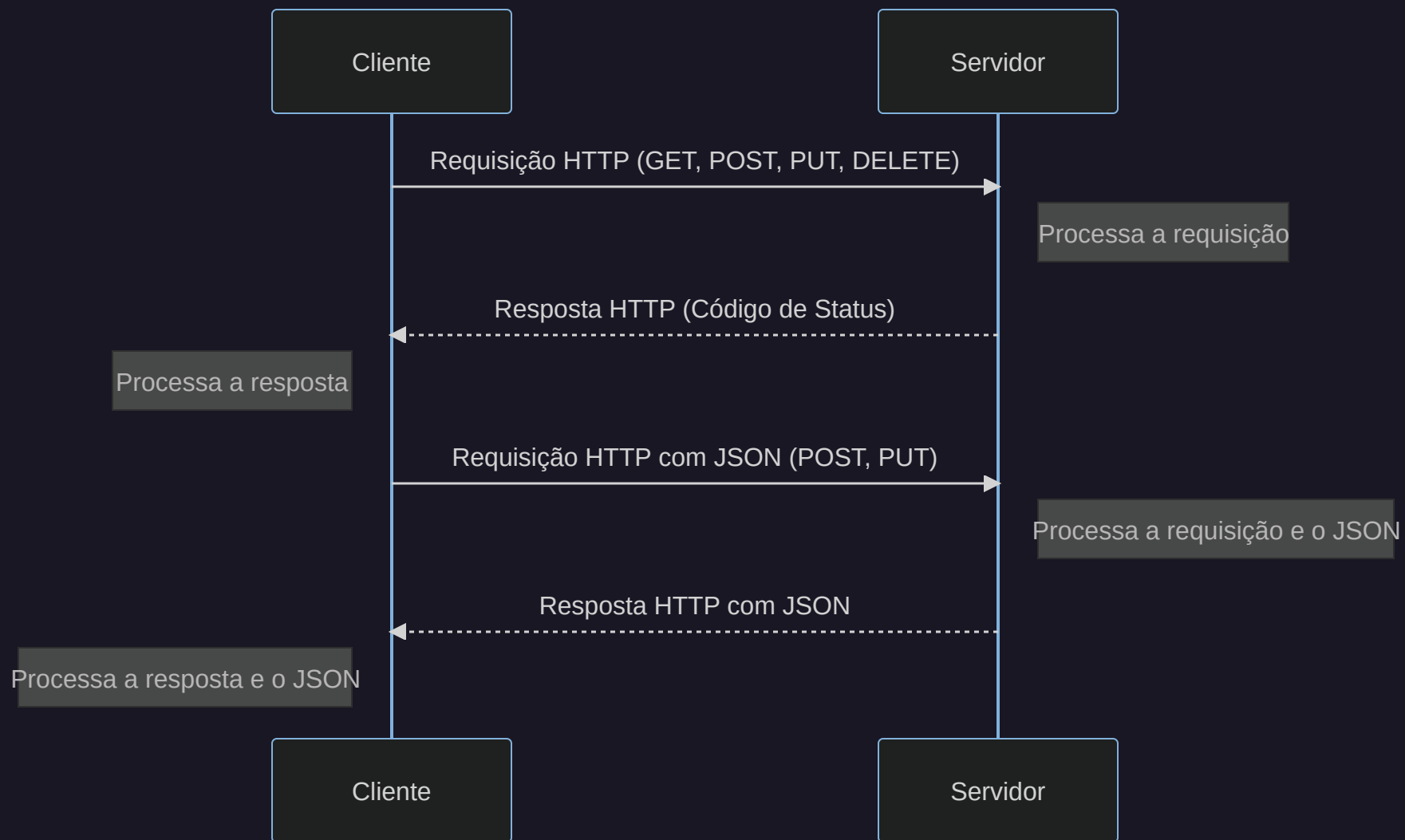
**Quando estamos falando de APIs modernas, estamos quase sempre falando sobre JSON**

```
def test_root_deve_retornar_200_e_ola_mundo():  
    response = client.get('/')  
    assert response.status_code == 200  
    assert response.json() == {'message': 'Olá Mundo!'}
```

**Um formato bastante parecido com um dicionário do python**



# De uma forma geral



# O que vamos criar nessa aula?

**Endpoints para cadastro, recuperação, alteração e deleção de usuários**

# Quando a API e o Banco de dados se encontram

Quando falamos de operações de banco de dados, temos um acrônimo para essas operações chamado **CRUD**:

- **C**reate (Criar): Adicionar novos registros ao banco de dados. No HTTP, essa ação geralmente é associada ao verbo POST.
- **R**ead (Ler): Recuperar registros existentes do banco de dados. No HTTP, essa ação geralmente é associada ao verbo GET.
- **U**ppdate (Atualizar): Modificar registros existentes no banco de dados. No HTTP, essa ação geralmente é associada ao verbo PUT ou PATCH.
- **D**elete (Excluir): Remover registros existentes do banco de dados. No HTTP, essa ação geralmente é associada ao verbo DELETE.

# A estrutura dos dados

**Se quisermos criar um endpoint com fastapi que crie um usuário, precisamos definir como trocaremos essa mensagem.**

**Imagino um JSON como esse:**

```
{  
  "username": "dunossauro",  
  "email": "dunossauro@email.com",  
  "password": "senha-do_dunossauro"  
}
```

# Pydantic e a validação de dados

O Pydantic é uma biblioteca Python que oferece validação de dados e configurações usando anotações de tipos Python. Ela é utilizada extensivamente em FastAPI para lidar com a validação e serialização/desserialização de dados.

O Pydantic tem um papel crucial ao trabalhar com JSON, pois permite a validação dos dados recebidos neste formato, assim como sua conversão para formatos nativos do Python e vice-versa.

## Dois conceitos importantes aqui

- **Esquemas:** No contexto da programação, um esquema é uma representação estrutural de um objeto ou entidade. Por exemplo, no nosso caso, um usuário pode ser representado por um esquema que contém campos para nome de usuário, e-mail e senha. Esquemas são úteis porque permitem definir a estrutura de um objeto de uma maneira clara e reutilizável.
- **Validação de dados:** Este é o processo de verificar se os dados recebidos estão em conformidade com as regras e restrições definidas. Por exemplo, se esperamos que o campo "email" contenha um endereço de e-mail válido, a validação de dados garantirá que os dados inseridos nesse campo de fato correspondam a um formato de e-mail válido.

# O pydantic fornece um schema para o json

## O json:

```
{  
  "username": "joao123",  
  "email": "joao123@email.com",  
  "password": "segredo123"  
}
```

## A classe do pydantic:

```
from pydantic import BaseModel, EmailStr  
  
class UserSchema(BaseModel):  
    username: str  
    email: EmailStr  
    password: str
```

**Vamos criar esse schema em um arquivo só de schemas**

`fast_zero/schemas.py` para fins de organização

**Dito isso, vamos implementar a criação do user**



# A rota

```
from fastapi import FastAPI
from fast_zero.schemas import UserSchema

# ...

@app.post('/users/', status_code=201)
def create_user(user: UserSchema):
    return user
```

## Alguns pontos:

- `status_code=201`: é a resposta esperado de uma criação
- `user: UserSchema`: diz ao endpoint qual o schema que desejamos receber

**Vamos ao swagger entender o que aconteceu**

**| <http://localhost:8000/docs>**

# Um problema!

Quando retornamos a requisição, estando expondo a senha, temos que criar um novo schema de resposta para que isso não seja feito.

Um schema que não expõe a senha:

```
class UserPublic(BaseModel):  
    username: str  
    email: EmailStr
```

Usando esse schema como resposta do nosso endpoint:

```
from fast_zero.schemas import UserSchema, UserPublic  
  
# código omitido  
  
@app.post('/users/', status_code=201, response_model=UserPublic)  
def create_user(user: UserSchema):  
    return user
```

# Criando um banco de dados falso

```
from fast_zero.schemas import UserSchema, UserPublic, UserDB

# ...

database = [] # provisório para estudo!

@app.post('/users/', status_code=201, response_model=UserPublic)
def create_user(user: UserSchema):
    user_with_id = UserDB(**user.model_dump(), id=len(database) + 1)
    # Aqui precisamos criar um novo modelo que represente o banco
    # Precisamos de um identificador para esse registro!

    database.append(user_with_id)

    return user
```

## Criando schemas compatíveis

Precisamos alterar nosso schema público para que ele tenha um id e também criar um schema que tenha o id e a senha para representar o banco de dados:

```
class UserPublic(BaseModel):  
    id: int  
    username: str  
    email: EmailStr  
  
class UserDB(UserSchema):  
    id: int
```

# Testando o endpoint

```
def test_create_user():
    client = TestClient(app)
    response = client.post(
        '/users/',
        json={
            'username': 'alice',
            'email': 'alice@example.com',
            'password': 'secret',
        },
    )
    assert response.status_code == 201
    assert response.json() == {
        'username': 'alice',
        'email': 'alice@example.com',
        'id': 1,
    }
```

## Não se repita (DRY)

Você deve ter notado que a linha `client = TestClient(app)` está repetida na primeira linha dos dois testes que fizemos. Repetir código pode tornar o gerenciamento de testes mais complexo à medida que cresce, e é aqui que o princípio de "Não se repita" (**DRY**`{:target="_blank"}`) entra em jogo. DRY incentiva a redução da repetição, criando um código mais limpo e manutenível.

Neste caso, vamos criar uma fixture que retorna nosso `client`. Para fazer isso, precisamos criar o arquivo `tests/conftest.py`. O arquivo `conftest.py` é um arquivo especial reconhecido pelo pytest que permite definir fixtures que podem ser reutilizadas em diferentes módulos de teste dentro de um projeto. É uma forma de centralizar recursos comuns de teste.

```
import pytest
from fastapi.testclient import TestClient
from fast_zero.app import app

@pytest.fixture
def client():
    return TestClient(app)
```



# Outros endpoints

| Agora eu vou no freestyle, sem slides. Me deseje sorte!

# Commit

```
$ git status  
$ git add .  
$ git commit -m "Implementando rotas CRUD"  
$ git push
```