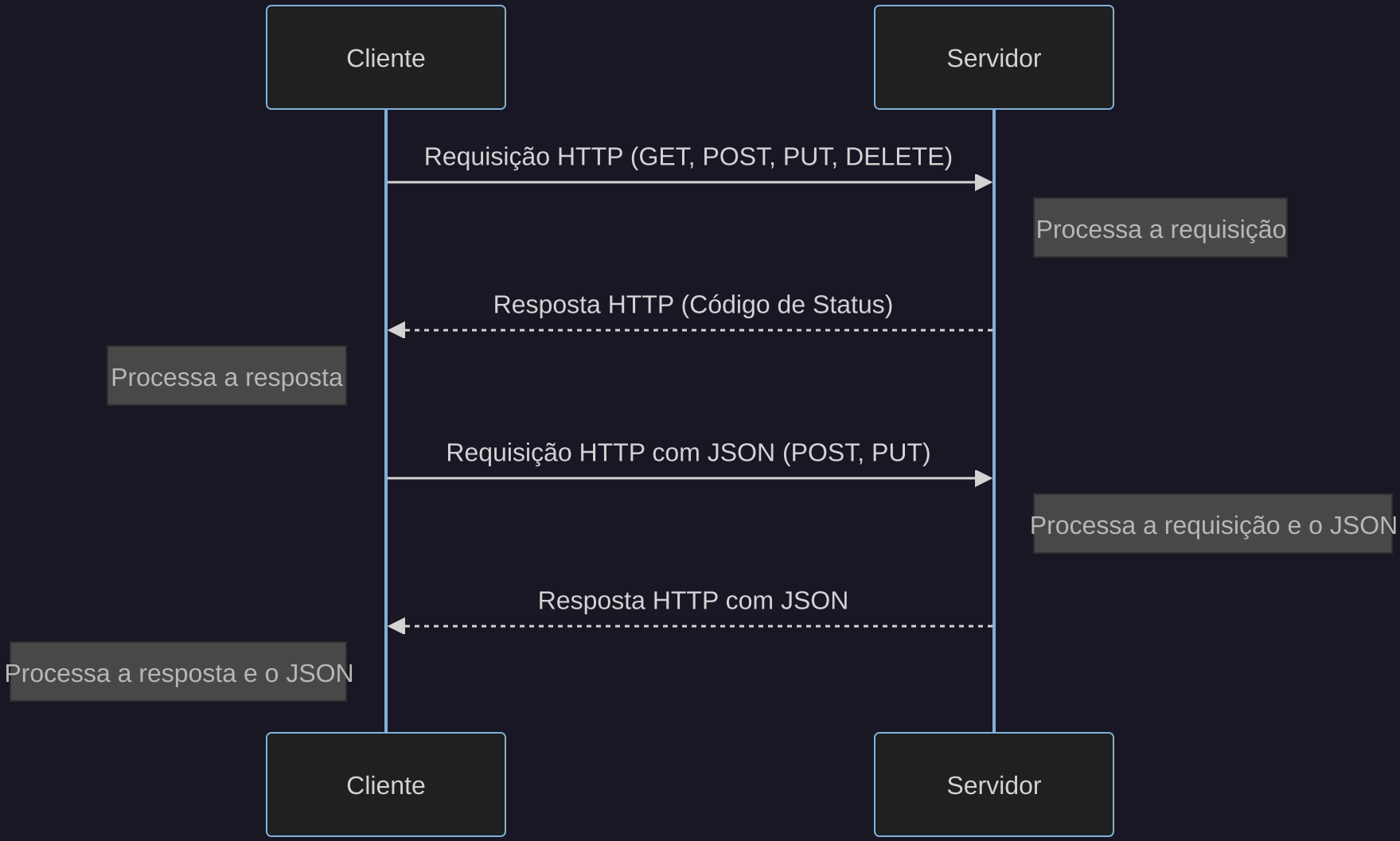
Estruturando o Projeto e Criando Rotas do CRUD

> https://fastapidozero.dunossauro.com/4.0/03/

Objetivos dessa aula:

- Aplicação prática dos conceitos da aula passada
  - http, verbos, status codes, schemas, ...
- Como estruturar rotas CRUD (Criar, Ler, Atualizar, Deletar)
- Aprofundar no Pydantic
- Escrita e execução de testes para validar o comportamento das rotas
- Um gerenciamento mínimo de cadastro de pessoas

Na aula passada



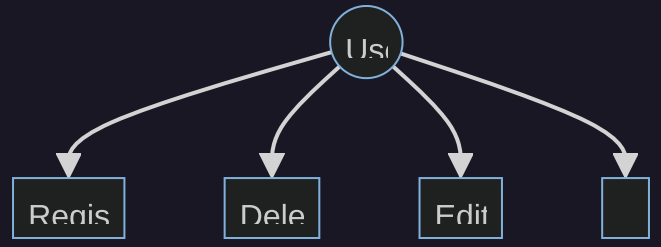
O que vamos criar nessa aula?

Endpoints para cadastro, recuperação, alteração e deleção de usuários

Um tipo de recurso

Quando queremos manipular um tipo especifico de dados, precisamos fazer algumas operações com ele.

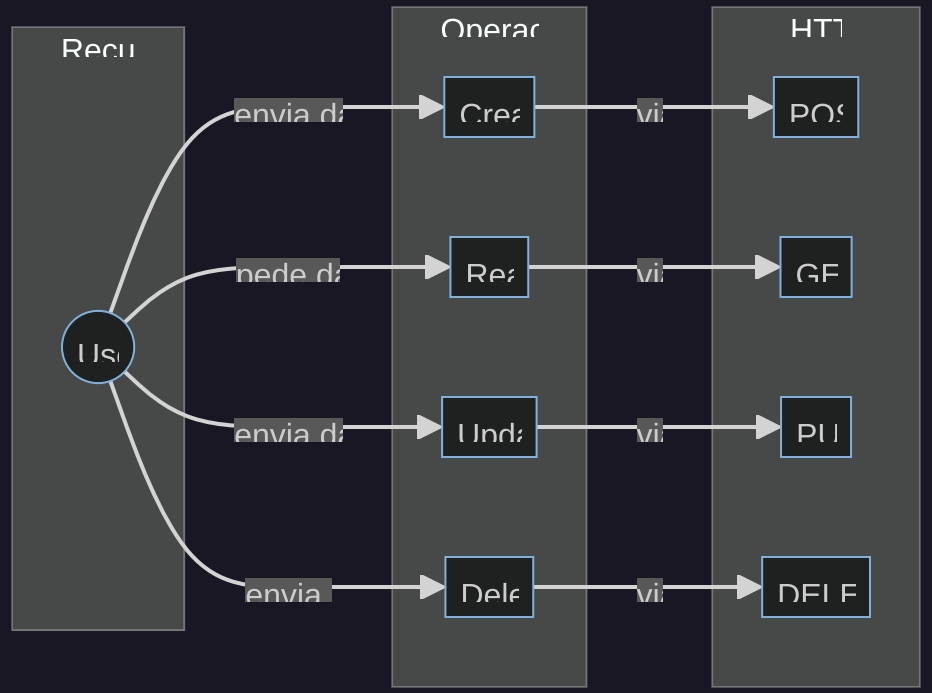
Por exemplo, vamos pensar na manipulação de `users`:



Operações com dados

- **C**reate (Criar): Adicionar novos registros
- **R**ead (Ler): Recuperar registros existentes
- **U**pdate (Atualizar): Modificar registros existentes
- **D**elete (Excluir): Remover registros existentes

Associações com HTTP



A estrutura dos dados

Se quisermos trocar mensagens via HTTP, precisamos definir um formato para transferir esse dado

Imagino um JSON como esse:

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

Pydantic

A responsabilidade de entender os schemas de contrato e a validação para saber se os dados estão no formato do schema, vai ficar a cargo do pydantic.

O json:

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

A classe do pydantic:

```
from pydantic import BaseModel


class UserSchema(BaseModel):
    username: str
    email: str
    password: str
```

Temos um de-para de chaves e tipos.

O pydantic têm tipos além do python

Validação de emails podem ser melhores:

```python
from pydantic import BaseModel, EmailStr


class UserSchema(BaseModel):
    username: str
    email: EmailStr
    password: str
```

Dito tudo isso

vamos implementar a criação do user

A rota

```python
from http import HTTPStatus

from fastapi import FastAPI
from fast_zero.schemas import UserSchema

# ...

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

- `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
```

Juntando ao endpoint

Usando esse schema como resposta do nosso endpoint:

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

# código omitido

@app.post('/users/', status_code=status_code=HTTPStatus.CREATED, response_model=UserPublic)
def create_user(user: UserSchema):
    return user
```

Criando um banco de dados falso

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

# ...

database = []  # provisório para estudo!


@app.post('/users/', status_code=status_code=HTTPStatus.CREATED, 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 enpoint

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

Pedindo os dados a API

Agora que já temos nosso "banco de dados", podemos criar um endpoint que nos mostra **todos** os recursos que já cadastramos na base.

O endpoint:

```python
@app.get('/users/', response_model=UserList)
def read_users():
    return {'users': database}
```

O schema para N users:

```python
class UserList(BaseModel):
    users: list[UserPublic]
```

Testando

```
def test_read_users(client):
    response = client.get('/users/')

    assert response.status_code == HTTPStatus.OK
    assert response.json() == {
        'users': [
            {
                'username': 'alice',
                'email': 'alice@example.com',
                'id': 1,
            }
        ]
    }
```

Alterando registros!

Antes de implementar o endpoint de fato, temos que aprender sobre parametrização na URL:

```
@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(user_id: int, user: UserSchema):
```

- `{user_id}`: cria uma "variável" na url
- `user_id: int`: diz que esse valor vai ser validado como um inteiro

A implementação

```python
from fastapi import FastAPI, HTTPException

# ...

@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(user_id: int, user: UserSchema):
    user_with_id = UserDB(**user.model_dump(), id=user_id)
    database[user_id - 1] = user_with_id

    return user_with_id
```

Um problema complicado

Imagine que tentemos alterar um id que não exista no banco de dados ou então pior, um valor menor do que 1, que é nosso id inicial.

```
from fastapi import FastAPI, HTTPException

# ...

@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(user_id: int, user: UserSchema):
    if user_id > len(database) or user_id < 1:
        raise HTTPException(
            status_code=HTTPStatus.NOT_FOUND, detail='User not found'
        )
    # ...
```

HTTPException

Quando queremos expor um erro ao cliente, devemos levantar (raise) uma Exception de HTTP.

Isso se transforma em um schema do pydantic para erros. A única chave disponível é `detail`.

```
raise HTTPException(status_code=404, detail='NOT FOUND')
```

> http://localhost:8000/docs

Testando o caminho feliz

```
def test_update_user(client):
    response = client.put(
        '/users/1',
        json={
            'username': 'bob',
            'email': 'bob@example.com',
            'password': 'mynewpassword',
        },
    )
    assert response.status_code == HTTPStatus.OK
    assert response.json() == {
        'username': 'bob',
        'email': 'bob@example.com',
        'id': 1,
    }
```

O delete!

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

Exercícios

1. Escrever um teste para o erro de `404` (NOT FOUND) para o endpoint de PUT;
2. Escrever um teste para o erro de `404` (NOT FOUND) para o endpoint de DELETE;
3. Crie um endpoint GET para pegar um único recurso como `users/{id}` e faça seus testes para `200` e `404`.

> Obviamente, não esqueça de responder ao quiz da aula

Suplementar / Para próxima aula

Para próxima aula, caso você não tenha nenhuma familiaridade com o SQLAlchemy ou com o Alembic, recomendo que assista a essas lives para se preparar e nivelar um pouco o conhecimento sobre essas ferramentas:

- SQLAlchemy: conceitos básicos, uma introdução a versão 2 | Live de Python #258
- Migrações, bancos de dados evolutivos (Alembic e SQLAlchemy) | Live de Python #211

Outro recurso que usaremos na próxima aula e pode te ajudar saber um pouco, são as variáveis de ambiente. Tema abordado em:

- Variáveis de ambiente, dotenv, constantes e configurações | Live de Python #207

Commit

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