AULA 08

FASTAPI

TESTES AUTOMATIZADOS

O QUE VEREMOS HOJE

01 O QUE SÃO TESTES AUTOMATIZADOS

02 TIPOS DE TESTES

03 TESTES NO FASTAPI

04 ESTRUTURAS DOS ARQUIVOS DE TESTES

05 TESTES DE ERRO

TESTE AUTOMATIZADOS

Testes são processos sistemáticos de verificação e validação que garantem que um sistema, software ou produto atenda aos requisitos e funcione corretamente.

Em APIs, os testes visam verificar o funcionamento correto de endpoints, garantindo que a aplicação responda conforme esperado.

TIPOS DE TESTES

Testes de Unidade: verificam o funcionamento de componentes individuais da API, como endpoints, funções e métodos, de forma isolada, garantindo que cada parte esteja funcionando corretamente.

Testes de Integração: validam a interação entre diferentes componentes, incluindo banco de dados e serviços externos.

Testes de Ponta a Ponta: simulam o uso real da API, verificando todo o fluxo de uma aplicação, desde a entrada de dados até a saída de resultados, garantindo o funcionamento da API em um cenário real.

TESTES NO FASTAPI

Para aplicar testes automatizados no FastAPI é comum o uso de bibliotecas voltadas para testes.

Uma das mais conhecidas é o Pytest que facilita o desenvolvimento de testes robustos e escaláveis.

Para usar a biblioteca, precisamos instalar usando o comando:

pip install pytest

Além do Pytest, também vamos usar a ferramenta **TestClient** do FastAPI que permite simular requisições HTTP.

pytest

ESTRUTURA DOS ARQUIVOS DE TESTES

Os arquivos de teste devem seguir o padrão de nome **test_*.py (test_main.py) ou *_test.py (user_test.py)** para que o Pytest detecte os arquivos automaticamente.

Cada função de teste deve começar com o prefixo **test_**. Isso também serve para que o Pytest identifique a função como um teste.

E o Pytest também tem como padrão o uso da palavra-chave **assert** para verificar as condições dos testes são verdadeiras.

CRIAÇÃO DE TESTES NO PYTEST

Para executar os testes, precisamos importar o TestClient do FastAPI e inicializar nossa aplicação dentro dele.

Em cada teste, criamos funções que representam os casos de teste. Dentro delas, usamos o **client** para fazer requisições aos endpoints e verificamos as respostas esperadas com o assert.

O **assert** é uma forma de fazer uma verificação ou 'checagem'. Ele serve para confirmar se uma determinada condição é verdadeira.

```python
from fastapi.testclient import TestClient
from main import app

# Criação de um cliente de teste para a aplicação
client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200 # Verifica se o status code da resposta é 200
    assert response.json() == {"message": "api running"} # Verifica se o conteúdo da resposta é o esperado
```

O QUE É ASSERT?

assert é uma maneira de confirmar que uma condição é verdadeira.

Se a condição for verdadeira, o teste continua. Se for falsa, o teste para e exibe uma mensagem de erro, indicando que algo deu errado.

O assert é a parte do teste que realmente confirma se o código está funcionando como esperado.

```python
def test_read_home():
    response = client.get("/")
    assert response.status_code == 200 # Assert serve para verificar se a condição é verdadeira
    assert response.json() == {"message": "api running"}

```

TESTES NA API

Com o Pytest podemos testar todos os endpoints da API simulando requisições reais, por isso é importante fazer o teste enviando todos os dados que a API espera e podemos verificar o retorno.

É comum validarmos o status code que está sendo retornado e o tipo de dado.

O **isinstance** é uma função nativa do Python que verifica se um objeto é de um determinado tipo ou classe. Ela retorna True se o objeto for do tipo especificado, e False caso contrário.

```python
from fastapi.testclient import TestClient
from main import app

# Criação de um cliente de teste para a aplicação
client = TestClient(app)

def test_get_users():
    response = client.get("/users")
    assert response.status_code == 200
    assert isinstance(response.json(), list) # Verifica se o conteúdo da resposta é uma lista
```

TESTES NA API

Já nos endpoints que exigem o envio de dados, como o POST, precisamos simular uma solicitação real enviando um JSON com os dados esperados pela API.

Após a solicitação, verificamos se a resposta está de acordo com o esperado. Esse processo de validação da resposta é essencial, pois garante que a API está retornando o que foi definido, tanto em estrutura quanto em conteúdo.

```python
def test_create_user():
    user_data = {"name": "Teste", "email": "teste@gmail.com"}

    response = client.post("/users", json=user_data)

    # colocamos o status code que deve ser retornado quando um novo usuário é criado
    assert response.status_code == 200

    # Recebe os dados da resposta da API como um JSON
    data = response.json()

    assert data["message"] == "User created"  # Verifica se a mensagem de sucesso está correta
    assert data["user"]["name"] == user_data["name"]  # Confirma se o nome do usuário está correto
    assert data["user"]["email"] == user_data["email"]  # Confirma se o email está correto
```

COMO EXECUTAR OS TESTES

Uma vez que os testes foram criados, podemos executá-los usando o pytest.

Rodando pytest na Raiz do Projeto: Se os arquivos de teste estão organizados na pasta raiz do projeto e seguem a nomenclatura esperada pelo pytest (normalmente começando com test_), basta rodar o comando no terminal:

```
pytest
```

Rodando pytest com Testes em Subpastas: Se os arquivos de teste estão dentro de uma pasta (como tests/), execute o pytest apontando para essa pasta:

```
pytest tests/
```

Em alguns casos, ao organizar o projeto em subpastas, pode ser necessário definir o PYTHONPATH para que o pytest encontre todos os módulos corretamente. Neste caso, use:

```
PYTHONPATH=. pytest
```

AUTENTICAÇÃO NOS TESTES

Quando a API exige token de autenticação, como um JWT, é necessário incluir esse token nos testes para que as requisições sejam autenticadas.

O token deve ser passado no header da requisição.

```python
from fastapi.testclient import TestClient
from ..main import app

# Criação de um cliente de teste para a aplicação
client = TestClient(app)

def get_auth_token():
    data = {'email': 'email_teste@gmail.com'}
    response = client.post("/generate-token", json=data)

    # Recebe o token gerado na resposta da API usando o get e a chave que retorna o token
    token = response.json().get("token")
    return token


def test_get_users():
    token = get_auth_token()
    headers = {"Authorization": token }

    # Faz a requisição GET para a rota e inclui o cabeçalho de autorização
    response = client.get("/users", headers=headers)

    assert response.status_code == 200
    assert isinstance(response.json(), list)  # Verifica se o conteúdo da resposta é uma lista
```

ATIVIDADE PRÁTICA

No projeto da API de Registros de Chamados, crie os testes para os endpoints do CRUD (GET, POST, PUT e DELETE).

Para cada teste, verifique o status code e a resposta, conforme definido nos endpoints da API.

TESTES DE ERRO

Os testes de erros são fundamentais para garantir que a API está preparada para lidar com entradas inválidas e recursos inexistentes, além de fornecer respostas informativas e apropriadas aos clientes da API.

Neste tipo de cenário a validação é, principalmente , em torno do status code de identificação do erro.

```python
from fastapi.testclient import TestClient
from ..main import app

# Criação de um cliente de teste para a aplicação
client = TestClient(app)

def test_get_nonexistent_user():
    response = client.get("/users/9999")  # ID inexistente
    assert response.status_code == 404
```

TESTES DE ERRO

Também é possível e essencial testar se a API retorna um erro quando algum dos dados obrigatórios está ausente ou no formato incorreto.

Esses testes garantem que a API trate entradas inválidas adequadamente e não permita a inserção de dados incorretos no banco de dados.

```python
# Criação de um cliente de teste para a aplicação
client = TestClient(app)

def test_create_user_missing_name():
    invalid_data = {"email": "teste@gmail.com"}  # Campo "name" ausente
    response = client.post("/users", json=invalid_data)

    assert response.status_code == 422  # Erro de validação
```

ATIVIDADE PRÁTICA

No projeto da API de Registros de Chamados, implemente o teste de registro não encontrado e de validação dos dados nos endpoints de criação e de atualização do chamado.

Nos testes, verifique se os status code estão vindo como o esperado para esses casos.