

.

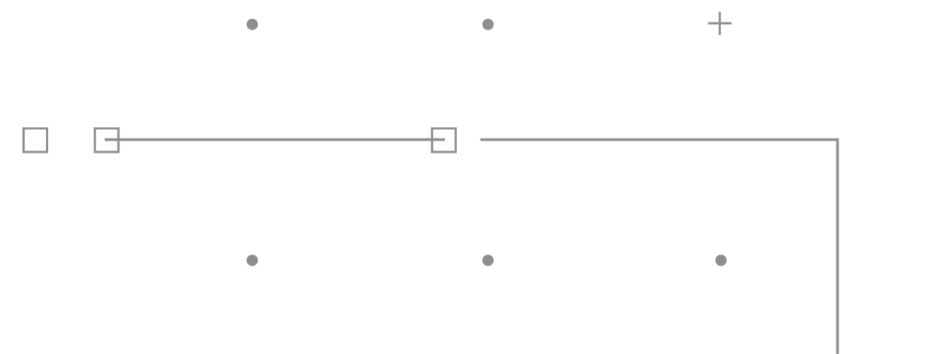
F

-

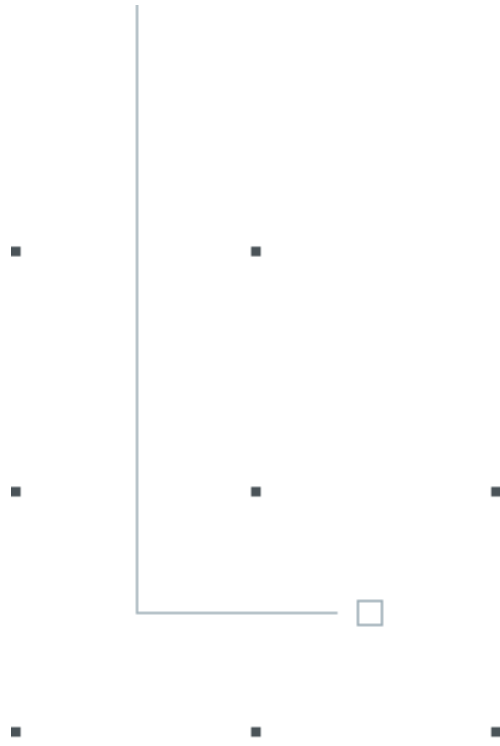
I

A

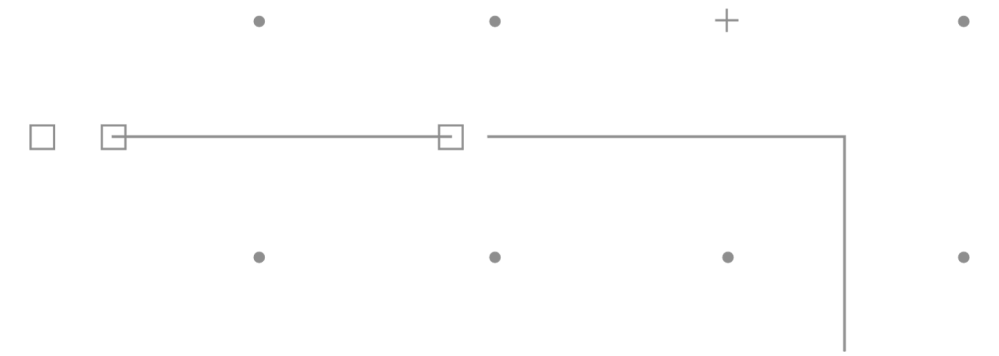
P




# Testes Automatizados




.





# Introdução



Para que sua aplicação tenha uma maior resiliência, é fundamental criar testes automatizados. Imagine o seguinte cenário: Imagine que sua aplicação tem um monte de funcionalidades que foram implementadas ao longo dos anos. Durante o planejamento, decidiram refatorar o código para deixar ele mais eficaz. Só que para garantir que sua aplicação continue funcionando, você vai precisar testar todo o sistema novamente. Não parece algo muito prático, certo?



# Unitários x Integrados

Quando se fala de testes automatizados, existem dois tipos de testes:

**Unitários** => Testa unidades individuais do seu código

**Integrados** => Testa uma combinação de funcionalidades da aplicação

# unittest

No Python, ele já tem um módulo específico para trabalhar com testes chamado **unittest**.  
Veremos a seguir uma forma simples de se criar um teste unitário usando o **unittest**

# httpx

Essa biblioteca é uma alternative para fazer requisições http.  
Quando se desenvolve utilizando **FastAPI**, ela é recomendada pela sua facilidade e praticidade.  
Para instalar, rode o comando:

```
pip install httpx
```

# unittest

Na estrutura do seu diretório, crie um arquivo chamado `test_main.py`

```
from unittest import mock, TestCase
from fastapi.testclient import TestClient

from main import app, get_db

def override_get_db(): ...

app.dependency_overrides[get_db] = override_get_db
client = TestClient(app)

class UserTestCase(TestCase):
    @mock.patch('database.crud.get_users')
    def test_get_users(self, get_users):
        users = [{
            "id": 1,
            "email": "profleonardo.bragatti@fiap.com.br",
            "is_active": False,
            "items": []
        }]
        get_users.return_value = users
        response = client.get("/users/")
        assert response.status_code == HTTPStatus.OK
        assert response.json() == users
```

# unittest

Explicando as linhas:

1 -> Importa o módulo `mock` e a classe `TestCase`. Para o **unittest** entender quais testes devem ser executados, você deve criar classes que são herdeiras da **TestCase**

3 -> Importa uma classe que simula requisições no **FastAPI** chamado `TestClient`

5 -> Importa a aplicação (`app`) e a função que inicia uma sessão no banco de dados (`get_db`)

7 -> Importa uma classe que contém todos os códigos de status de retorno de uma requisição HTTP

10-11 -> Cria uma função "false" apenas para fingir que vai conectar no banco de dados

13 -> Sobrescreve a função `get_db` utilizada na aplicação para utilizar a nova função "falsa"

14 -> Cria um objeto que irá simular as requisições na aplicação

16 -> Cria uma classe que irá conter os teste referentes as funcionalidades de usuário

17 -> Utiliza o módulo `mock` para falsificar o método `get_users`, que a função onde busca os dados do usuários no banco de dados

18 -> Cria um teste que vai buscas os usuários e validar se está tudo ok

27 -> Define qual o retorno que deve vir da função falsificada

29 -> Simula uma requisição no endpoint `/users/`

30-31 -> Verifica se o código de status e os dados da requisição retornaram respectivamente o valor esperado



# unittest

Uma vez criado o teste, basta executar o comando:

```
python -m unittest
```

# mock

O módulo `mock` é muito utilizado em situações onde você quer substituir e imitar objetos dentro de um ambiente de testes.

Você pode controlar o comportamento do seu teste usando `mock` para validar se as funcionalidades estão se comportando conforme o esperado.

# mock

Conforme o teste descrito no slides anteriores, tem o seguinte bloco de código:

```
@mock.patch("database.crud.get_users")
def test_get_user(self, get_users):
    ...
```

Decoramos a função `test_get_users`, que vai receber do `mock.patch` a função falsa `get_users` que originalmente está no caminho `database.crud.get_users`.

O que vai acontecer é que durante esse teste unitário, toda vez que a função `get_users`, que está dentro de `database.crud`, na verdade invocará a função que foi alterada pelo `mock.patch`. Essa função falsificada é passada como argumento para a função `test_get_user` e dentro deste teste nos alteramos o seu retorno, pois quando a `get_users` é invocada, ela retorna uma lista de usuários, porém é informado qual retorno vai ser esperado pelo teste.

# unittest

Uma vez o teste para obter a lista de usuários foi feito, é bom testar se a criação de usuário vai se comportar conforme o esperado.

```
@mock.patch('database.crud.get_user_by_email')
@mock.patch('database.crud.create_user')
def test_create_user(self, create_user, get_user_by_email):
    user_data = {
        "id": 1,
        "email": "profleonardo.bragatti@fiap.com.br",
        "is_active": False,
        "items": []
    }
    create_user.return_value = user_data
    get_user_by_email.return_value = None

    response = client.post("/users/", json={
        "email": "profleonardo.bragatti@fiap.com.br",
        "password": "123"
    })
    assert response.status_code == HTTPStatus.OK
    assert response.json() == user_data
```

# unittest

No teste anterior, nota-se que são criados dois `mocks`. Um para a função `create_user` e outro para a função `get_user_by_email`. Ambos são utilizados durante a requisição de criar usuário, então precisamos falsificar ambas para ter o retorno esperado.

Interessante ressaltar que pode-se criar quantos mocks forem necessários para criar seus testes.

--

Mas nem tudo são flores, certo? Durante a criação do usuário, podemos ter requisições que não vem no formato esperado, retornando um erro 4XX. E para esse tipo de cenário, também se deve criar um teste para garantir que sua aplicação é consistente.

```
def test_invalid_user_create(self):
    response = client.post('/users/', json={})
    assert response.status_code == HTTPStatus.UNPROCESSABLE_ENTITY
```

# unittest

No exemplo a seguir, um teste de como validar se um usuário existe

```
@mock.patch("database.crud.get_user_by_email")
def test_user_exists(self, get_user_by_email):
    get_user_by_email.return_value = self.user

    response = client.post("/users/", json={"email": "profleonardo.bragatti@fiap.com.br", "password": "123"})
    assert response.status_code == HTTPStatus.BAD_REQUEST
    assert response.json() == {"detail": "Email already registered"}
```

# pytest

Uma outra opção para realizarmos testes é a biblioteca [pytest](#).  
Ele simplifica a forma de se escrever os testes.  
Para utiliza-lo, basta instalar no seu ambiente:

```
pip install pytest
```

# pytest

Com o **pytest**, escrever os testes fica mais simples, pois ele não utiliza classes. Basta escrever uma função. Reescrevendo a função de obter uma lista de usuários, o arquivo ficaria dessa forma:

```
from http import HTTPStatus
from unittest import TestCase, mock

from fastapi.testclient import TestClient
from main import app, get_db

def override_get_db():
    pass

app.dependency_overrides[get_db] = override_get_db
client = TestClient(app)

@mock.patch('database.crud.get_users')
def test_get_users(get_users):
    users = [{
        "id": 1,
        "email": "profleonardo.bragatti@fiap.com.br",
        "is_active": False,
        "items": []
    }]
    get_users.return_value = users
    response = client.get("/users/")
    assert response.status_code == HTTPStatus.OK
    assert response.json() == users
```



# pytest

E para executar os testes usando o pytest, basta executar o comando:

```
pytest
```

A saída será algo conforme abaixo:

```
===== test session starts =====
platform linux -- Python 3.10.12, pytest-7.4.0, pluggy-1.2.0
rootdir: /mnt/6769bdb4-48cc-4fb6-af4f-4b745e2d4ae4/code/Python/mastering/fastapi
plugins: anyio-3.7.0
collected 1 item

test_main.py . [100%]
```

# pytest

E assim como utilizamos no `unittest`, podemos utilizar o `mock` quantas vezes for necessário para escrever seus testes, seguindo os mesmos critérios que no `unittest`

```
@mock.patch("database.crud.get_user_by_email")
@mock.patch("database.crud.create_user")
def test_create_user(create_user, get_user_by_email):
    create_user.return_value = {"id": 1, "email": "profleonardo.bragatti@fiap.com.br", "is_active": False, "items": []}
    get_user_by_email.return_value = None

    response = client.post("/users/", json={"email": "profleonardo.bragatti@fiap.com.br", "password": "123"})
    assert response.status_code == HTTPStatus.OK
    assert response.json() == self.user
```

# pytest

Um teste para validar se a criação do usuário está inválida, fica bem simples:

```
def test_invalid_user_create():  
    response = client.post("/users/")  
    assert response.status_code == HTTPStatus.UNPROCESSABLE_ENTITY
```

# pytest

Um recurso muito interessante do `pytest` é que caso você tenha testes escritos com o `unittest`, ele também irá executar estes testes  
Então se pegarmos o exemplo anterior:

```
class UserTestCase(TestCase):
    def setUp(self):
        self.user = {"id": 1, "email": "profleonardo.bragatti@fiap.com.br", "is_active": False, "items": []}
        self.users = [self.user]

    @mock.patch("database.crud.get_users")
    def test_get_users(self, get_users):
        get_users.return_value = self.users
        response = client.get("/users/")
        assert response.status_code == HTTPStatus.OK
        assert response.json() == self.users
```

E rodarmos o `pytest` você verá que o teste `test_get_users` será executado

# Coverage

Coverage é uma abordagem utilizada para determinar quanto seus testes estão cobrindo as funcionalidades implementadas no seu código.

No Python, existe a biblioteca [coverage](#) que pode ser instalada com o comando:

```
pip install coverage
```

# Coverage

Uma vez instalado, você precisa primeiro executar os testes utilizando o **coverage**. Se você utiliza o **unittest**, rode o comando:

```
coverage run -m unittest discover
```

A saída será igual quando o **unittest** é executado:

```
.  
-----  
Ran 1 test in 0.007s  
  
OK
```

# Coverage

Caso esteja utilizando o `pytest`:

```
coverage run -m pytest
```

A saída também será a mesma como se tivesse executado apenas o `pytest`

```
===== test session starts =====
platform linux -- Python 3.10.12, pytest-7.4.0, pluggy-1.2.0
rootdir: /mnt/6769bdb4-48cc-4fb6-af4f-4b745e2d4ae4/code/Python/mastering/fastapi
plugins: anyio-3.7.0
collected 1 item

test_main.py . [100%]
```

# Coverage

`coverage` fará uma análise de todos os testes executados e das funcionalidades no seu código e ver o quanto coberto ele está.

Para ter certeza que o comando foi executado com sucesso, repare que na pasta será gerado um arquivo `.coverage`



# Coverage

Após executar os testes utilizando o `coverage`, é possível ver um relatório do que ele analisou, para isso basta rodar:

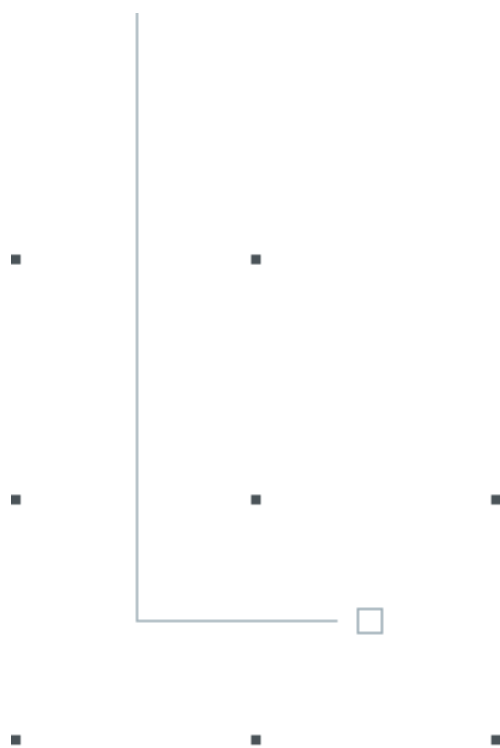
```
coverage report -m
```

A saída desse relatório será algo do tipo:

Name	Stmts	Miss	Cover	Missing
auth.py	40	22	45%	25-28, 32-44, 48-55
database/__init__.py	11	4	64%	14-18
database/crud.py	23	15	35%	7, 11, 15, 19-24, 28, 32-36
database/models.py	21	0	100%	
database/schemas.py	22	0	100%	
main.py	36	12	67%	22-25, 36-40, 45, 50, 55, 60
test_main.py	55	34	38%	32-72
TOTAL	208	87	58%	

# Coverage

No relatório, a coluna **Cover** diz quanto do arquivo está coberto de testes e a coluna **Missing** mostra quais as linhas que não estão cobertas por testes. É sempre bom se atentar nessas duas colunas, pois te dão uma boa ideia do que falta para garantir que sua aplicação seja consistente



.

F

-

I

A

P

