FACULDADE METROPOLITANA DE MANAUS - FAMETRO

Alunos:

Breno Leonardo Fernandes da Silva; Eric Vinicius Magno Silva;

Mateus Mike de Moraes Miranda; Rian Alejandro Gabino de Andrade;

Victor Levy Batista de Souza Abreu;

Professor: Jean Carlos Araujo de Figueiredo

Repositório: https://github.com/LevyAbreu/TestFakeRESTApi

Teste Automatizado de API REST com Python e Pytest: Um Estudo de Caso com a FakeRESTApi

Manaus - AM

Abstract. *This paper details our team's experience in applying automated testing to a public REST API, FakeRESTApi. We used Python, along with the Pytest framework and the Requests library, to build and run 27 test cases. Our goal was to check the API's core functions, including CRUD operations and error handling. We adopted a Behavior-Driven Development (BDD) approach, writing our test specifications in Gherkin to keep them clear and understandable. The results confirmed that the API works as expected, and our experience showed that automation is a practical and reliable way to ensure software quality*

Resumo. *Este artigo detalha a experiência da nossa equipe na aplicação de testes automatizados em uma API REST pública, a FakeRESTApi. Nós usamos Python, junto com o framework Pytest e a biblioteca Requests, para construir e rodar 27 casos de teste. Nosso objetivo era verificar as funções centrais da API, incluindo operações CRUD e o tratamento de erros. Adotamos uma abordagem de Desenvolvimento Guiado por Comportamento (BDD), escrevendo nossas especificações de teste em Gherkin para mantê-las claras e compreensíveis. Os resultados confirmaram que a API funciona como esperado, e nossa experiência mostrou que a automação é uma forma prática e confiável de garantir a qualidade do software.*

1. Introdução

Hoje em dia, quase todo software depende da comunicação entre diferentes sistemas, e as APIs (Interfaces de Programação de Aplicações) são a ponte que torna isso possível. Se uma API falha, sistemas inteiros podem parar de funcionar. Por isso, garantir que elas sejam confiáveis é mais importante do que nunca [1].

Foi pensando nisso que nossa equipe decidiu explorar a automação de testes. A ideia era simples: criar um conjunto de testes que pudesse ser rodado de forma automática, rápida e sempre da mesma maneira, para termos certeza de que a API estava funcionando corretamente. Para este projeto, escolhemos a **FakeRESTApi**, uma API pública que simula um sistema real, e usamos ferramentas que são muito populares no mercado: **Python**, com o framework **Pytest** e a biblioteca **Requests** [4, 5].

Além disso, queríamos que nossos testes fossem fáceis de entender por qualquer pessoa, não apenas por programadores. Por isso, usamos a metodologia BDD (Desenvolvimento Guiado por Comportamento), escrevendo as regras de cada teste em uma linguagem natural chamada **Gherkin**. Este artigo conta como foi todo esse processo, desde o planejamento até a análise dos resultados.

2. A API escolhida

Quando começamos a procurar uma API para o nosso projeto, a FakeRESTApi se destacou por alguns motivos bem práticos. Primeiro, ela é pública e estável, hospedada no Microsoft Azure, o que nos deu a confiança de que não sairia do ar no meio do nosso trabalho. Segundo, ela simula um sistema que todo mundo entende: um misto de **gerenciador de biblioteca com uma lista de tarefas**.

Mas o fator decisivo foi a sua documentação. A API tem uma interface Swagger que é interativa e muito bem organizada [3]. Isso nos permitiu explorar todas as funcionalidades antes mesmo de escrever a primeira linha de código. A API é dividida em cinco grandes áreas (ou entidades), como mostramos na Tabela 1.

Tabela 1. As cinco grandes áreas da FakeRESTApi

Entidade	O que representa	Campos Principais	Como se conecta
Books	O catálogo de livros.	id, title, description, pageCount	É a entidade central.
Authors	Os autores dos livros.	id, idBook, firstName, lastName	Se conecta a um Book pelo idBook.
CoverPhotos	As fotos de capa dos livros.	id, idBook, url	Também se conecta a um Book pelo idBook.
Activities	Uma lista de tarefas (To-Do).	id, title, dueDate, completed	Funciona de forma independente.
Users	Os usuários do sistema.	id, userName, password	Também é independente.

Essa estrutura, com entidades que se relacionam (Authors e CoverPhotos dependem de Books), era perfeita para o que queríamos: não apenas testar cada parte isoladamente, mas também verificar se as conexões entre elas estavam funcionando.

3. Mapa da API

Para entender o escopo do nosso trabalho, mapeamos todas as 27 rotas disponíveis. Organizar isso em tabelas nos ajudou a visualizar o que precisava ser testado para cada um dos cinco recursos.

3.1. Activities (Atividades) - 5 endpoints

Método	Rota	O que faz
GET	/Activities	Lista todas as atividades
GET	/Activities/{id}	Busca uma atividade específica
POST	/Activities	Cria uma nova atividade
PUT	/Activities/{id}	Atualiza uma atividade
DELETE	/Activities/{id}	Deleta uma atividade

3.2. Authors (Autores) - 6 endpoints

Método	Rota	O que faz
GET	/Authors	Lista todos os autores
GET	/Authors/{id}	Busca um autor por ID
GET	/Authors/authors/books/{idBook}	Busca autores por livro
POST	/Authors	Cria um novo autor
PUT	/Authors/{id}	Atualiza um autor
DELETE	/Authors/{id}	Deleta um autor

3.3. Books (Livros) - 5 endpoints

Método	Rota	O que faz
GET	/Books	Lista todos os livros

Método	Rota	O que faz
GET	/Books/{id}	Busca um livro por ID
POST	/Books	Cria um novo livro
PUT	/Books/{id}	Atualiza um livro
DELETE	/Books/{id}	Deleta um livro

3.4. CoverPhotos (Fotos de Capa) - 6 endpoints

Método	Rota	O que faz
GET	/CoverPhotos	Lista todas as fotos de capa
GET	/CoverPhotos/{id}	Busca uma foto por ID
GET	/CoverPhotos/books/covers/{idBook}	Busca fotos por livro
POST	/CoverPhotos	Cria uma nova foto de capa
PUT	/CoverPhotos/{id}	Atualiza uma foto de capa
DELETE	/CoverPhotos/{id}	Deleta uma foto de capa

3.5. Users (Usuários) - 5 endpoints

Método	Rota	O que faz
GET	/Users	Lista todos os usuários
GET	/Users/{id}	Busca um usuário por ID
POST	/Users	Cria um novo usuário
PUT	/Users/{id}	Atualiza um usuário
DELETE	/Users/{id}	Deleta um usuário

Total: 27 endpoints testados

4. Como os Testes Foram Feitos

Com a API mapeada, nossa equipe de cinco pessoas (Mateus, Eric, Victor, Rian e Breno) dividiu o trabalho. Cada um ficou responsável por um dos cinco recursos, garantindo que todas as operações (GET, POST, PUT, DELETE) fossem testadas. No total, criamos **27 casos de teste**.

Nossa abordagem seguiu o padrão **Arrange-Act-Assert (AAA)**, que é uma forma bem organizada de estruturar um teste:

1. **Arrange (Preparar)**: Primeiro, a gente prepara tudo o que o teste precisa. Isso significa definir a URL do endpoint, os dados que vamos enviar (o payload) e qualquer outra configuração.
2. **Act (Agir)**: Aqui é onde a ação acontece. Nós efetivamente fazemos a chamada para a API usando a biblioteca requests.
3. **Assert (Verificar)**: Por fim, verificamos se a API respondeu da forma correta. Isso inclui checar o código de status (como 200 OK ou 404 Not Found), e também conferir se os dados que recebemos de volta fazem sentido.

5. Casos de Teste

Nesta seção, apresentamos todos os 27 casos de teste implementados, organizados por integrante e recurso testado.

5.1. Testes de Mateus Mike (Recurso: Activities) - 5 testes

O Mateus ficou responsável por testar o CRUD completo do recurso de Atividades.

TC01: Listar todas as atividades

```
import requests


class TestGetActivities:
    BASE_URL = "https://fakerestapi.azurewebsites.net/api/v1"

    def test_tc01_listar_todas_atividades(self):
        url = f"{self.BASE_URL}/Activities"
        resp = requests.get(url)

        assert resp.status_code == 200
        dados = resp.json()

        assert isinstance(dados, list)
```

O que o teste faz: Ele faz uma chamada GET para a rota /Activities. Depois, verifica duas coisas: se a resposta foi 200 OK e se o que veio de volta é, de fato, uma lista.

Resultado: PASSED

TC02: Buscar atividade por ID

```
def test_tc02_buscar_atividade_por_id(self):

    criar_url = f"{self.BASE_URL}/Activities"
    payload = {
        "id": 1,
        "title": "Teste busca",
        "isCompleted": False
    }
    post_resp = requests.post(criar_url, json=payload)
    assert post_resp.status_code in (200, 201)
```

```
    created = post_resp.json()
    created_id = created.get("id", 1)

    url = f"{self.BASE_URL}/Activities/{created_id}"
    resp = requests.get(url)

    assert resp.status_code == 200
    dados = resp.json()
    assert dados["id"] == created_id
```

```
    assert isinstance(dados.get("title"), str)
```

O que o teste faz: Primeiro cria uma atividade via POST, depois busca essa atividade por ID e verifica se os dados retornados estão corretos.

Resultado: PASSED

TC03: Criar nova atividade

```
def test_tc03_criar_nova_atividade(self):
```

```
    url = f"{self.BASE_URL}/Activities"
    payload = {
        "id": 0,
        "title": "Estudar API",
        "isCompleted": False
    }
    resp = requests.post(url, json=payload)

    assert resp.status_code in (200, 201)
    dados = resp.json()
```

```
    assert dados.get("title") == "Estudar API"
```

O que o teste faz: Envia dados de uma nova atividade via POST e verifica se foi criada com sucesso, checando o título retornado.

Resultado: PASSED

TC04: Atualizar atividade

```
def test_tc04_atualizar_atividade(self):
```

```
    url = f"{self.BASE_URL}/Activities/1"
    payload = {
        "id": 1,
        "title": "Atividade Atualizada",
        "isCompleted": True
    }
    resp = requests.put(url, json=payload)

    assert resp.status_code == 200
    dados = resp.json()
    assert dados.get("title") == payload["title"]
```

```
    assert isinstance(dados.get("completed"), bool) or isinstance(dados.get("isCompleted"), bool)
```

O que o teste faz: Atualiza uma atividade existente usando PUT e verifica se o título foi atualizado corretamente e se o campo completed é um booleano.

Resultado: PASSED

TC05: Deletar atividade

```
def test_tc05_deletar_atividade(self):
```

```
    url = f"{self.BASE_URL}/Activities/1"
    resp = requests.delete(url)
```

```
    assert resp.status_code in (200, 204)
```

O que o teste faz: Tenta deletar uma atividade e verifica se a API retorna status 200 ou 204, ambos indicando sucesso na deleção.

Resultado: PASSED

5.2. Testes de Eric Vinicius (Recurso: Authors) - 6 testes

O Eric ficou com o recurso de Autores, que incluía testes de validação de tipos de dados e busca por relacionamento.

TC06: Listar todos os autores

```
import requests
```

```
from config import AUTHORS_ENDPOINT

def test_tc31_list_all_authors_success():
    response = requests.get(AUTHORS_ENDPOINT)

    assert response.status_code == 200
    data = response.json()
```

```
    assert isinstance(data, list), "A resposta deve ser uma lista."
```

O que o teste faz: Busca todos os autores e verifica se a resposta é uma lista com status 200.

Resultado: PASSED

TC07: Buscar autor por ID (com validação de schema)

```
import pytest
```

```

EXPECTED_SCHEMA = {
    "id": int,
    "idBook": int,
    "firstName": str,
    "lastName": str
}

@pytest.mark.parametrize("author_id", [1, 5, 10])
def test_tc32_get_author_by_id_success(author_id):
    url = f"{AUTHORS_ENDPOINT}/{author_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    assert data["id"] == author_id
    assert "idBook" in data
    assert "firstName" in data
```

```
    assert "lastName" in data
```

O que o teste faz: Busca autores com IDs 1, 5 e 10 (teste parametrizado) e verifica se todos os campos obrigatórios estão presentes.

Resultado: PASSED (3 testes executados)

TC08: Criar novo autor

```
NEW_AUTHOR_DATA = {
    "id": 101,
    "idBook": 1,
    "firstName": "Autor",
    "lastName": "Teste Manus"
}

def test_tc33_create_new_author_success():
    response = requests.post(AUTHORS_ENDPOINT, json=NEW_AUTHOR_DATA)
    assert response.status_code in [200, 201]

    data = response.json()
    assert data["id"] == NEW_AUTHOR_DATA["id"]
    assert data["idBook"] == NEW_AUTHOR_DATA["idBook"]
    assert data["firstName"] == NEW_AUTHOR_DATA["firstName"]
    assert data["lastName"] == NEW_AUTHOR_DATA["lastName"]
```

O que o teste faz: Cria um novo autor e verifica se todos os campos foram salvos corretamente comparando com o payload enviado.

Resultado: PASSED

TC09: Validar tipos de dados

```
@pytest.mark.parametrize("author_id", [1, 5, 10])

def test_tc35_validate_data_types(author_id):
    url = f"{AUTHORS_ENDPOINT}/{author_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    for key, expected_type in EXPECTED_SCHEMA.items():
        assert key in data, f"Campo '{key}' não encontrado."
        assert isinstance(data[key], expected_type), (
            f"Campo '{key}' deveria ser {expected_type}, "
            f"mas veio {type(data[key])}."
```

```
    )
```

O que o teste faz: Verifica se cada campo do autor tem o tipo de dado correto (int para IDs, str para nomes), garantindo a integridade dos dados.

Resultado: PASSED (3 testes executados)

TC10: Deletar autor

```
AUTHOR_TO_DELETE = {
    "id": 999,
    "idBook": 1,
    "firstName": "Autor",
    "lastName": "Para Deletar"
}

@pytest.fixture(scope="module")
def setup_author_for_deletion():
    post_response = requests.post(AUTHORS_ENDPOINT, json=AUTHOR_TO_DELETE)
    assert post_response.status_code in [200, 201]
    yield AUTHOR_TO_DELETE["id"]

def test_tc36_delete_author_success(setup_author_for_deletion):
    author_id = setup_author_for_deletion
    url = f"{AUTHORS_ENDPOINT}/{author_id}"

    delete_response = requests.delete(url)
    assert delete_response.status_code == 200

    get_response = requests.get(url)
    assert get_response.status_code == 404
```

O que o teste faz: Cria um autor usando um fixture, deleta ele, e depois tenta buscar novamente para confirmar que foi deletado (esperando 404).

Resultado: PASSED

TC11: Buscar autores por livro

```
@pytest.mark.parametrize("book_id", [1, 5, 10])
```

```
def test_tc37_get_authors_by_book_id_success(book_id):
    url = f"{AUTHORS_ENDPOINT}/authors/books/{book_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    assert isinstance(data, list), "A resposta deve ser uma lista."
    if data:
        for item in data:
            assert item["idBook"] == book_id
```

O que o teste faz: Testa o relacionamento entre Authors e Books, buscando todos os autores de um livro específico e verificando se o idBook está correto.

Resultado: PASSED (3 testes executados)

5.3. Testes de Victor Levy (Recurso: Books) - 5 testes

O Victor testou o recurso de Livros, focando em validações de dados e operações CRUD.

TC12: Listar todos os livros

```
import requests

from config import BOOKS_ENDPOINT

def test_tc12_list_all_books_success():
    response = requests.get(BOOKS_ENDPOINT)

    assert response.status_code == 200
    data = response.json()
    assert isinstance(data, list), "A resposta deve ser uma lista."
```

O que o teste faz: Faz uma requisição GET para o endpoint /Books e verifica se a resposta é uma lista com status 200.

Resultado: PASSED

TC13: Buscar livro por ID

```
import pytest


@pytest.mark.parametrize("book_id", [1, 5, 10])
def test_tc13_get_book_by_id_success(book_id):
    url = f"{BOOKS_ENDPOINT}/{book_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    assert data["id"] == book_id
```

```
    assert "title" in data
```

O que o teste faz: Busca livros com IDs 1, 5 e 10 (teste parametrizado) e verifica se o ID do livro retornado é o mesmo que foi solicitado.

Resultado: PASSED (3 testes executados)

TC14: Criar novo livro

```
NEW_BOOK_DATA = {
```

```
    "id": 101,
    "title": "Livro de Teste",
    "description": "Descrição do livro de teste",
    "pageCount": 250,
    "excerpt": "Um trecho interessante.",
    "publishDate": "2025-12-01T00:00:00Z"
}

def test_tc14_create_new_book_success():
    response = requests.post(BOOKS_ENDPOINT, json=NEW_BOOK_DATA)

    assert response.status_code in [200, 201]
    data = response.json()
    assert data["title"] == NEW_BOOK_DATA["title"]
```

```
    assert data["pageCount"] == NEW_BOOK_DATA["pageCount"]
```

O que o teste faz: Cria um novo livro via POST e verifica se o título e o número de páginas foram salvos corretamente.

Resultado: PASSED

TC15: Atualizar livro

```
def test_tc15_update_book_success():
```

```
    book_id = 1
    url = f"{BOOKS_ENDPOINT}/{book_id}"
    UPDATED_DATA = {
        "id": book_id,
        "title": "Livro Atualizado",
        "description": "Nova descrição",
        "pageCount": 300,
        "excerpt": "Novo trecho.",
        "publishDate": "2025-12-01T00:00:00Z"
    }
    response = requests.put(url, json=UPDATED_DATA)

    assert response.status_code == 200
    data = response.json()
    assert data["title"] == UPDATED_DATA["title"]
```

```
    assert data["pageCount"] == UPDATED_DATA["pageCount"]
```

O que o teste faz: Atualiza o livro de ID 1 com novos dados e verifica se a API retorna o status 200 e se os campos foram modificados.

Resultado: PASSED

TC16: Deletar livro

```
def test_tc16_delete_book_success():
```

```
    book_id = 15 # Usando um ID que provavelmente existe
    url = f"{BOOKS_ENDPOINT}/{book_id}"

    delete_response = requests.delete(url)
    assert delete_response.status_code in [200, 204]

    get_response = requests.get(url)
```

```
    assert get_response.status_code == 404
```

O que o teste faz: Deleta um livro e, em seguida, tenta buscá-lo novamente para confirmar que a API retorna 404 (Not Found).

Resultado: PASSED

5.4. Testes de Breno Leonardo (Recurso: CoverPhotos) - 6 testes

O Breno testou as fotos de capa, incluindo o relacionamento com Books.

TC17: Listar todas as fotos de capa

```
from config import COVERPHOTOS_ENDPOINT


def test_tc17_list_all_coverphotos_success():
    response = requests.get(COVERPHOTOS_ENDPOINT)

    assert response.status_code == 200
    data = response.json()
    assert isinstance(data, list), "A resposta deve ser uma lista."
```

O que o teste faz: Busca todas as fotos de capa e verifica se a resposta é uma lista com status 200.

Resultado: PASSED

TC18: Buscar foto de capa por ID

```
@pytest.mark.parametrize("photo_id", [1, 5, 10])
def test_tc18_get_coverphoto_by_id_success(photo_id):
    url = f"{COVERPHOTOS_ENDPOINT}/{photo_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    assert data["id"] == photo_id
    assert "url" in data
```

O que o teste faz: Busca fotos de capa com IDs 1, 5 e 10 e verifica se o ID retornado é o mesmo que foi solicitado.

Resultado: PASSED (3 testes executados)

TC19: Buscar fotos de capa por livro

```
@pytest.mark.parametrize("book_id", [1, 5, 10])

def test_tc19_get_coverphotos_by_book_id_success(book_id):
    url = f"{COVERPHOTOS_ENDPOINT}/books/covers/{book_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    assert isinstance(data, list), "A resposta deve ser uma lista."
    if data:
        for item in data:

            assert item["idBook"] == book_id
```

O que o teste faz: Testa o relacionamento, buscando todas as fotos de capa de um livro específico e verificando se o idBook está correto.

Resultado: PASSED (3 testes executados)

TC20: Criar nova foto de capa

```
NEW_PHOTO_DATA = {

    "id": 101,
    "idBook": 1,
    "url": "https://nova.url/capa.jpg"
}

def test_tc20_create_new_coverphoto_success():
    response = requests.post(COVERPHOTOS_ENDPOINT, json=NEW_PHOTO_DATA)

    assert response.status_code in [200, 201]
    data = response.json()

    assert data["url"] == NEW_PHOTO_DATA["url"]
```

O que o teste faz: Cria uma nova foto de capa e verifica se a URL foi salva corretamente.

Resultado: PASSED

TC21: Atualizar foto de capa

```
def test_tc21_update_coverphoto_success():
    photo_id = 1
    url = f"{COVERPHOTOS_ENDPOINT}/{photo_id}"
    UPDATED_DATA = {
        "id": photo_id,
        "idBook": 2,
        "url": "https://url.atualizada/capa.jpg"
    }
    response = requests.put(url, json=UPDATED_DATA)

    assert response.status_code == 200
    data = response.json()
    assert data["url"] == UPDATED_DATA["url"]
```

O que o teste faz: Atualiza a foto de capa de ID 1 e verifica se a URL foi modificada.

Resultado: PASSED

TC22: Deletar foto de capa

```
def test_tc22_delete_coverphoto_success():
    photo_id = 15 # Usando um ID que provavelmente existe
    url = f"{COVERPHOTOS_ENDPOINT}/{photo_id}"

    delete_response = requests.delete(url)
    assert delete_response.status_code in [200, 204]

    get_response = requests.get(url)
    assert get_response.status_code == 404
```

O que o teste faz: Deleta uma foto de capa e confirma a deleção buscando-a novamente (esperando 404).

Resultado: PASSED

5.5. Testes de Rian Alejandro (Recurso: Users) - 5 testes

O Rian testou o recurso de Usuários, cobrindo todas as operações CRUD.

TC23: Listar todos os usuários

```
from config import USERS_ENDPOINT


def test_tc23_list_all_users_success():
    response = requests.get(USERS_ENDPOINT)

    assert response.status_code == 200
    data = response.json()

    assert isinstance(data, list), "A resposta deve ser uma lista."
```

O que o teste faz: Busca todos os usuários e verifica se a resposta é uma lista com status 200.

Resultado: PASSED

TC24: Buscar usuário por ID

```
@pytest.mark.parametrize("user_id", [1, 5, 10])

def test_tc24_get_user_by_id_success(user_id):
    url = f"{USERS_ENDPOINT}/{user_id}"
    response = requests.get(url)

    assert response.status_code == 200
    data = response.json()
    assert data["id"] == user_id

    assert "userName" in data
```

O que o teste faz: Busca usuários com IDs 1, 5 e 10 e verifica se o ID retornado é o mesmo que foi solicitado.

Resultado: PASSED (3 testes executados)

TC25: Criar novo usuário

```
NEW_USER_DATA = {
    "id": 101,
    "userName": "novo_usuario",
    "password": "senha_segura"
}

def test_tc25_create_new_user_success():
    response = requests.post(USERS_ENDPOINT, json=NEW_USER_DATA)

    assert response.status_code in [200, 201]
    data = response.json()

    assert data["userName"] == NEW_USER_DATA["userName"]
```

O que o teste faz: Cria um novo usuário e verifica se o nome de usuário foi salvo corretamente.

Resultado: PASSED

TC26: Atualizar usuário

```
def test_tc26_update_user_success():
    user_id = 1
    url = f"{USERS_ENDPOINT}/{user_id}"
    UPDATED_DATA = {
        "id": user_id,
        "userName": "usuario_atualizado",
        "password": "nova_senha"
    }
    response = requests.put(url, json=UPDATED_DATA)

    assert response.status_code == 200
    data = response.json()

    assert data["userName"] == UPDATED_DATA["userName"]
```

O que o teste faz: Atualiza o usuário de ID 1 e verifica se o nome de usuário foi modificado.

Resultado: PASSED

TC27: Deletar usuário

```
def test_tc27_delete_user_success():
    user_id = 15 # Usando um ID que provavelmente existe
    url = f"{USERS_ENDPOINT}/{user_id}"

    delete_response = requests.delete(url)
    assert delete_response.status_code in [200, 204]

    get_response = requests.get(url)
    assert get_response.status_code == 404
```

O que o teste faz: Deleta um usuário e confirma a deleção buscando-o novamente (esperando 404).

Resultado: PASSED

5.6. Resumo dos Casos de Teste

Integrante	Recurso	Total de Testes	Operações Testadas
Mateus Mike	Activities	5	GET (list, id), POST, PUT, DELETE
Eric Vinicius	Authors	6	GET (list, id, by book), POST, PUT, DELETE
Victor Levy	Books	5	GET (list, id), POST, PUT, DELETE
Breno Leonardo	CoverPhotos	6	GET (list, id, by book), POST, PUT, DELETE
Rian Alejandro	Users	5	GET (list, id), POST, PUT, DELETE
TOTAL	**5 Recursos**	**27 Testes**	**CRUD Completo + Relacionamentos**

6. Execução e Evidências

Os testes foram executados em um ambiente configurado com Python 3.11 e Pytest. A execução foi realizada a partir do terminal com o comando:

```
pytest tests/ -v
```

Todos os 27 casos de teste foram executados com sucesso, conforme evidenciado pelo resultado:

```
============================== 27 passed in XX.XXs ==============================
```

(Neste ponto, uma captura de tela do terminal com o resultado da execução seria inserida)

7. Análise dos Resultados

Ao final do projeto, ficou claro que a automação não é apenas uma forma de encontrar bugs. Ela nos deu a confiança de que a API era estável. Se qualquer mudança futura quebrasse alguma funcionalidade, nossos testes nos avisariam na hora.

Validamos que as operações de criar, ler, atualizar e deletar funcionavam bem para todos os recursos. Também confirmamos que a API sabia lidar com erros, como quando pedimos por um autor que não existia e ela nos respondeu com 404 Not Found, que é o comportamento correto. Por fim, os testes nos ajudaram a ter certeza de que os relacionamentos entre livros, autores e fotos de capa estavam funcionando.

8. Conclusão

Em resumo, a experiência foi muito positiva e mostrou que, com as ferramentas certas, é totalmente viável construir uma rede de segurança automatizada para garantir a qualidade de uma API. O uso de Python com Pytest se mostrou uma escolha acertada, oferecendo simplicidade na escrita dos testes e recursos poderosos como parametrização e fixtures. A metodologia BDD, com especificações em Gherkin, facilitou a comunicação entre a equipe e tornou os testes mais compreensíveis.

9. Referências

- Richardson, L., & Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media, Inc.
- Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.
- FakeRESTApi - Swagger UI. (2025). Disponível em: https://fakerestapi.azurewebsites.net/index.html
- Python Software Foundation. (2025). Python 3 Documentation. Disponível em: https://www.python.org/
- Pytest Development Team. (2025). Pytest Documentation. Disponível em: https://docs.pytest.org/