CASOS DE TESTE - API LOCUS

Victor Hugo Mendes[1], Franco Talles, Jõao Menezes, Huan Cruz, David Alejandro

[1]Engenharia de Software – Faculdade Metropolitana de Manaus (FAMETRO)

Abstract. *This document details the application of tests studied in the Software Testing I. The tests were performed using Python and Pytest, in addition to Gherkin - which describes expected behaviors and is widely used in BDD (Behavior-Driven Development) to create automated tests. The testes API belongs to the Locus Project, a website where people can report abandonment, adopt and provide other types of animal-related support.*

Resumo. *Este documento documenta a aplicação de testes aprendidos na disciplina de Teste de Software I. Os testes foram realizados utilizando python e sua biblioteca Pytest, além de Gherkin que serve para descrever comportamentos esperados do teste, amplamente utilizado na metodologia BDD (Behavior-Driven Development) para criar testes automatizados. A API testada foi a API do projeto Locus, que é um site onde pessoas podem reportar abandonos, adotar e outros tipos de apoios relacionados a animais.*

O projeto Locus foi desenvolvido com o objetivo de dar visibilidade a animais em situação de abandono, disponíveis para adoção ou que necessitem de assistência, conectando-os a uma comunidade engajada no bem-estar animal. A plataforma opera como uma rede social de nicho, cujas funcionalidades incluem o gerenciamento de usuários, locais e a publicação de posts solicitando apoio.

A API do projeto - cujo código fonte está disponível em https://github.com/FrancoTalles/Locus/tree/main/backend - interage com um banco de dados relacional estruturado em entidades como *Posts*, *Usuários* e *Locais*. Este trabalho apresenta uma suíte de testes focada nos *endpoints* da API Locus, priorizando a validação dos fluxos principais de sucesso (*happy path*).

O projeto com os testes está no link:

https://github.com/FrancoTalles/Locus-Teste

Casos de Teste - Entidades

1. Local (Victor-Hugo)

Conftest.py

```python
import pytest
import requests
from pytest_bdd import given, parsers, then

@pytest.fixture
def context():
    return {}

@given(parsers.parse('que a url base da API é "{url}"'))
def set_base_url(context, url):
    context['base_url'] = url

@given(parsers.parse('que o endpoint de local é "{endpoint}"'))
def set_endpoint(context, endpoint):
    context['endpoint'] = endpoint

    context['full_url'] = f"{context['base_url']}{context['endpoint']}"

@given(parsers.parse('que tenho locais cadastrados no sistema'))
def ensure_locais_exist(context):
    full_url = f"{context['base_url']}{context['endpoint']}"

    novo_local = {
        "nome": "Local criado",
        "endereco": "Rua Teste, 100",
        "categoria": "Temporario",
        "latitude": "-1.0",
        "longitude": "-1.0"
    }

    response = requests.post(full_url, json=novo_local)
    print(response.json())
    assert response.status_code == 201, "Falha ao criar local de setup"

    context['id_local_criado'] = response.json().get('local_id')
    context['created_local_name'] = response.json().get('nome')

@then(parsers.parse('o sistema deve retornar status "{status_str}"'))
def check_status_code(context, status_str):

    try:
        expected_status = int(status_str)
    except ValueError:
        pytest.fail(f"O valor de status '{status_str}' no Gherkin não é um número válido.")

    assert context['response'].status_code == expected_status

@then(parsers.parse('o campo "{campo}" deve ser igual a "{valor}"'))
def check_field_str_value(context, campo, valor):

    json_data = context['response'].json()

    assert str(json_data.get(campo)) == str(valor), (
        f"Erro no campo '{campo}': Esperado '{valor}', Recebido '{json_data.get(campo)}'"
    )
```

O arquivo define um conjunto de etapas BDD usando **pytest-bdd** para validar o cadastro de um novo local na API.

As funções representam os passos descritos no arquivo **`.feature`** e são usadas para preparar o cenário e verificar o retorno da API.

Configuração inicial

A fixture **`context()`** cria um dicionário vazio para armazenar informações ao longo da execução do teste, como URLs, respostas e IDs criados.

Definição da URL base (@given)

A função **`set_base_url`** recebe a URL principal da API e salva esse valor no contexto.

Definição do endpoint (@given)

A função **`set_endpoint`** guarda no contexto qual é o endpoint responsável pelos locais e já monta a URL completa combinando base + endpoint.

Criação de um local para o cenário (@given)

A função **`ensure_locais_exist`** cria um local de exemplo para garantir que o sistema já possui registros antes de continuar o teste.
Ela:

- monta a URL completa,
- envia um **POST** com os dados do novo local,
- verifica se a API respondeu com **201**,
- e salva no contexto o **ID** do local criado e o nome retornado.

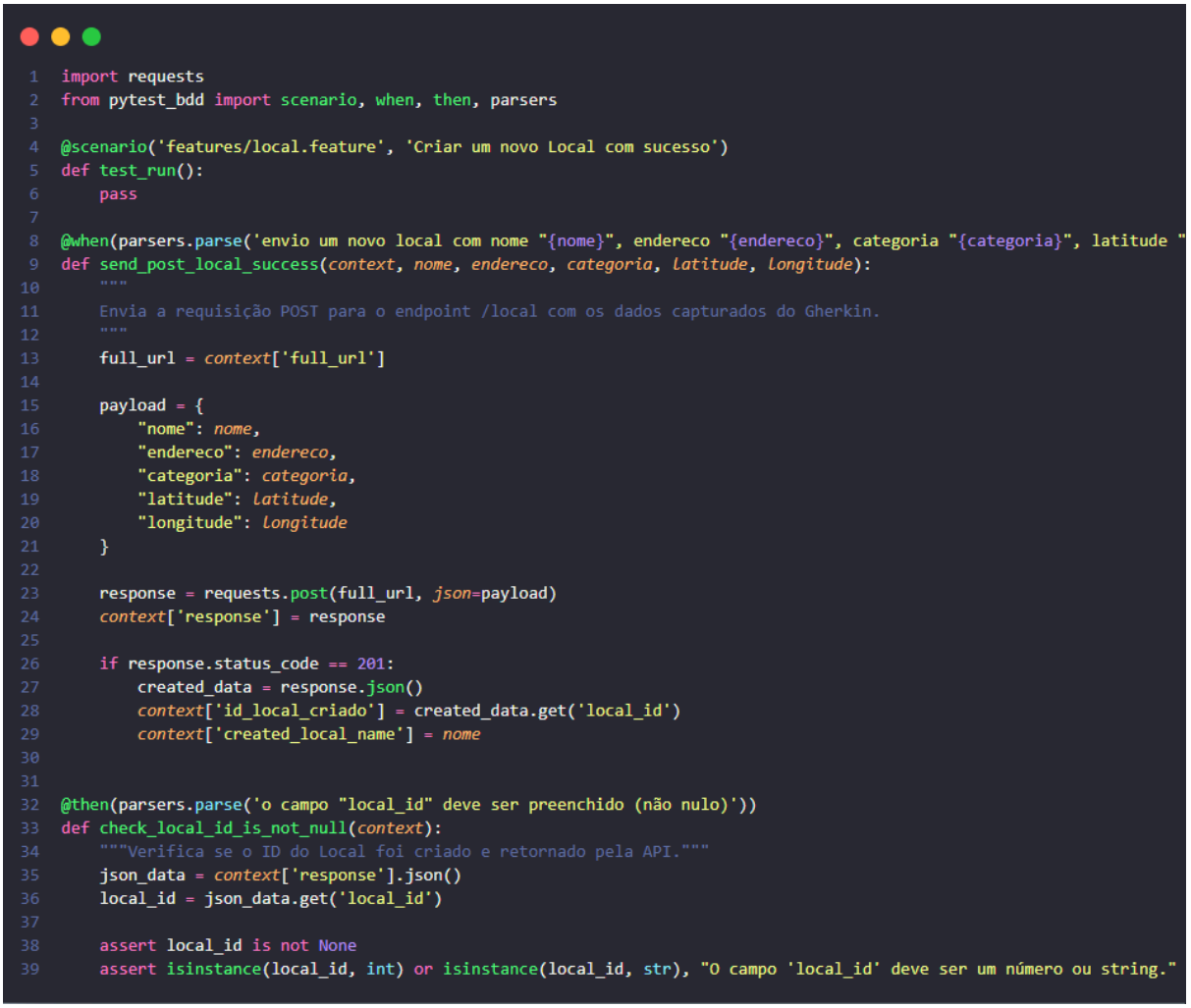
Verificação do status de retorno (@then)

A função **`check_status_code`** confirma se o código de status armazenado no contexto corresponde ao valor esperado informado no Gherkin.

Verificação de campos retornados (@then)

A função **`check_field_str_value`** compara o valor de um campo específico no JSON de resposta com o valor informado no cenário, garantindo que o retorno está correto.

test_create.py



```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('features/local.feature', 'Criar um novo Local com sucesso')
def test_run():
    pass

@when(parsers.parse('envio um novo local com nome "{nome}", endereco "{endereco}", categoria "{categoria}", latitude "
def send_post_local_success(context, nome, endereco, categoria, latitude, longitude):
    """
    Envia a requisição POST para o endpoint /local com os dados capturados do Gherkin.
    """
    full_url = context['full_url']

    payload = {
        "nome": nome,
        "endereco": endereco,
        "categoria": categoria,
        "latitude": latitude,
        "longitude": longitude
    }

    response = requests.post(full_url, json=payload)
    context['response'] = response

    if response.status_code == 201:
        created_data = response.json()
        context['id_local_criado'] = created_data.get('local_id')
        context['created_local_name'] = nome


@then(parsers.parse('o campo "local_id" deve ser preenchido (não nulo)'))
def check_local_id_is_not_null(context):
    """Verifica se o ID do Local foi criado e retornado pela API."""
    json_data = context['response'].json()
    local_id = json_data.get('local_id')

    assert local_id is not None
    assert isinstance(local_id, int) or isinstance(local_id, str), "O campo 'local_id' deve ser um número ou string."
```

O arquivo define um cenário BDD usando **pytest-bdd** para validar a criação de um novo local na API.
As etapas seguem o fluxo descrito no arquivo **`.feature`**, organizando o teste em ações (when) e validações (then).

Carregamento do cenário

A função **`test_run()`** apenas associa o teste ao cenário *"Criar um novo Local com sucesso"* definido no arquivo **`local.feature`.**

Envio dos dados do novo local (@when)

A função **`send_post_local_success`** é responsável por montar e enviar a requisição **POST** para criar um local usando os valores fornecidos no Gherkin.

Ela realiza as seguintes ações:

- recupera a URL completa salva no contexto,
- monta o corpo da requisição (payload) com nome, endereço, categoria, latitude e longitude,
- envia o POST para o endpoint,
- salva a resposta retornada no contexto para validações posteriores.

Se a API retornar **201**, a função também:

- captura o **`local_id`** retornado,
- armazena o **`id_local_criado`** e o nome criado no contexto.

Validação do ID retornado (@then)

A função **`check_local_id_is_not_null`** verifica se o campo **`local_id`** está presente na resposta e se ele não é nulo.

Além disso, garante que o ID tenha um tipo válido, aceitando tanto número quanto string, conforme o comportamento da API.

test_delete.py

```python
import requests
from pytest_bdd import scenario, when, then

@scenario('features/local.feature', 'Deletar um Local existente com sucesso')
def test_run():
    pass

@when('removo o local criado anteriormente')
def delete_dynamic_post(context):
    local_id = context['id_local_criado']

    print(local_id)

    full_url = f"{context['base_url']}{context['endpoint']}/{local_id}"

    context['response'] = requests.delete(full_url)

@then('o sistema deve retornar status 204')
def check_status_204(context):
    assert context['response'].status_code == 204

@then('ao pesquisar pelo ID removido o sistema deve retornar 404')
def verify_item_deleted(context):
    local_id = context['id_local_criado']
    full_url = f"{context['base_url']}{context['endpoint']}/{local_id}"

    response = requests.get(full_url)
    assert response.status_code == 404
```

O arquivo define um cenário BDD usando **pytest-bdd** para validar a remoção de um local previamente criado.

As etapas seguem o fluxo definido no arquivo **`.feature`**, organizando a operação de exclusão e suas verificações.

Carregamento do cenário

A função **`test_run()`** apenas liga o teste ao cenário *"Deletar um Local existente com sucesso"* descrito em **`local.feature`**.

Remoção do local previamente criado (@when)

A função **`delete_dynamic_post`** é responsável por realizar a requisição de deleção.

Ela:

- recupera do contexto o **`id_local_criado`,**
- monta a URL completa adicionando o ID ao final do endpoint,
- envia a requisição **DELETE** para a API,
- armazena a resposta no contexto.

Validação do status de deleção (@then)

A função **`check_status_204`** confirma que a API retornou **204**, indicando que o local foi removido corretamente e que não há conteúdo na resposta.

Verificação da remoção (@then)

A função **`verify_item_deleted`** garante que o local realmente não existe mais.
Ela:

- monta novamente a URL com o ID removido,
- envia um **GET** para consultar o item,
- e valida que a API retorna **404**, indicando que o recurso não está mais disponível.

test_findAll.py

```
mport pytest
import requests
from pytest_bdd import scenario, given, when, then, parsers
```

```python
# --- CENÁRIO - (GET) ---

@scenario('features/local.feature', 'Listar todos os Locais Cadastrados')
def test_listar_locais():
    """Define e executa o cenário de listagem de locais."""
    pass


@when(parsers.parse('envio uma requisição GET para /local'))
def send_get_all_request(context):
    try:
        context['response'] = requests.get(context['full_url'])
    except requests.exceptions.ConnectionError as e:
        pytest.fail(f"Falha ao conectar à api em {context['full_url']}: {e}")


@then(parsers.parse('o sistema deve retornar status 200'))
def check_status_200(context):

    assert context['response'].status_code == 200

@then(parsers.parse('deve retornar uma lista de locais'))
def check_response_is_list(context):

    try:
        json_data = context['response'].json()
        assert isinstance(json_data, list)
        #lista não está vazia
        assert len(json_data) > 0, "A lista de locais retornada está vazia."
    except requests.exceptions.JSONDecodeError:
        pytest.fail("O corpo da resposta não é um JSON válido.")

@then(parsers.parse('cada local deve conter os campos "{campos_string}"'))
def check_local_schema(context, campos_string):

    json_data = context['response'].json()

    # Processa os campos esperados (separados por vírgula e removendo aspas)
    campos_esperados = {campo.strip().replace('"', '') for campo in
campos_string.split(',')}
```

```
    for i, local in enumerate(json_data):
        chaves_recebidas = set(local.keys())


        missing_fields = campos_esperados.difference(chaves_recebidas)

        assert not missing_fields, (
            f"Local no índice {i} ({local.get('local_id', 'ID desconhecido')})
está faltando os campos: {missing_fields}"
        )
```

O arquivo implementa um cenário BDD usando **pytest-bdd** para validar a listagem de todos os locais cadastrados na API.
As etapas seguem o fluxo definido no arquivo **`.feature`**, organizando a chamada GET e suas validações.

Carregamento do cenário

A função **`test_listar_locais()`** apenas associa o teste ao cenário *"Listar todos os Locais Cadastrados"* presente no arquivo **`local.feature`**.

Envio da requisição GET (@when)

A função **`send_get_all_request`** é responsável por consultar o endpoint que retorna a lista completa de locais.

Ela:

- envia uma requisição **GET** para a URL salva no contexto,
- armazena a resposta para as validações seguintes,
- e trata erros de conexão, falhando o teste caso a API esteja indisponível.

Validação do status de retorno (@then)

A função **`check_status_200`** verifica se a API respondeu com **status 200**, indicando que a listagem foi processada com sucesso.

Validação do formato da resposta (@then)

A função **`check_response_is_list`** garante que o corpo da resposta:

- é um JSON válido,
- contém uma lista,
- e que essa lista não está vazia.

Isso confirma que existem locais cadastrados e que o endpoint retorna a estrutura esperada.

Validação do esquema dos itens retornados (@then)

A função **`check_local_schema`** verifica se cada item da lista contém todos os campos obrigatórios informados no cenário.

Ela:

- interpreta os campos esperados fornecidos no Gherkin,
- compara os campos recebidos com os obrigatórios,
- e falha o teste caso algum campo esteja faltando.

test.findOne.py

```python
import pytest
import requests
from pytest_bdd import scenario,  when, then, parsers


# --- CENÁRIO (GET por ID) ---

@scenario('features/local.feature', 'Listar um Local ao buscar por ID')
def test_buscar_local_por_id_existente():
    """Define e executa o cenário de busca por ID existente."""
    pass


@when(parsers.parse('pesquiso pelo local com ID {local_id:d}'))
def send_get_by_id_request(context, local_id):
    context['local_id'] = local_id
    url_with_id = f"{context['full_url']}/{local_id}"
    try:
        context['response'] = requests.get(url_with_id)
    except requests.exceptions.ConnectionError as e:
        pytest.fail(f"Falha ao conectar à API em {url_with_id}: {e}")


@then(parsers.parse('o sistema deve retornar status 200'))
def check_status_200(context):
    assert context['response'].status_code == 200

@then(parsers.parse('o campo "local_id" deve ser igual a {expected_id:d}'))
def check_id_match(context, expected_id):
    """Verifica se o ID retornado no corpo do JSON é o ID esperado (1)."""
    json_data = context['response'].json()
    assert json_data.get('local_id') == expected_id, (
        f"Esperado local_id: {expected_id}, Recebido: {json_data.get('local_id')}"
    )

@then(parsers.parse('o campo "nome" do local deve ser "{expected_name}"'))
def check_local_name(context, expected_name):
    """Verifica se o nome do local retornado é o nome esperado."""
    json_data = context['response'].json()
    assert json_data.get('nome') == expected_name, (
        f"Esperado nome: {expected_name}, Recebido: {json_data.get('nome')}"
    )
```

O código mostra um conjunto de **testes BDD usando pytest-bdd**. Ele simula a busca de um “local” na API a partir de um ID e valida se o retorno está correto.

Cenário BDD

A função **`test_buscar_local_por_id_existente()`** liga o teste ao cenário descrito no arquivo **`local.feature`.**

Ação (WHEN) – Fazer a requisição

No passo **`send_get_by_id_request`,** o teste:

- recebe o ID informado no cenário,
- monta a URL com esse ID,
- faz um GET usando **`requests.get`,**
- salva a resposta dentro do **`context`** para que os outros passos possam usar.

Validação do status (THEN)

O passo **`check_status_200`** garante que a API respondeu com código HTTP **200**, indicando sucesso.

Validação do ID retornado

O passo **`check_id_match`** pega o JSON retornado e verifica se o campo **`local_id`** é o mesmo ID pedido no cenário.

Validação do nome do local

O passo **`check_local_name`** verifica se o campo `nome` do JSON é o que o cenário espera.

test_update.py

```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('features/local.feature', 'Atualizar um Local existente com sucesso')
def test_run():
    pass

@when(parsers.parse('atualizo o local criado anteriormente com a nova categoria "{nova_categoria}"'))
def update_dynamic_post(context, nova_categoria):
    local_id = context['id_local_criado']

    full_url = f"{context['base_url']}{context['endpoint']}/{local_id}"

    payload = {
        "categoria": nova_categoria,
    }

    context['response'] = requests.patch(full_url, json=payload)
    print(context['response'].json())

@then('o sistema deve retornar status 200')
def check_status_200(context):
    assert context['response'].status_code == 200

@then(parsers.parse('o campo "{nome}" deve permanecer o original'))
def check_field_str_original(context):
    expected_name = context['created_local_name']
    json_data = context['response'].json()

    assert json_data.get('nome') == expected_name, (
        f"Erro: O nome foi alterado. Esperado: '{expected_name}', Recebido: '{json_data.get('nome')}'")
```

Este código define um cenário BDD que testa a **atualização de um Local já existente** usando um método PATCH.

1. Cenário do teste

A função **`test_run()`** só liga o teste ao cenário descrito no arquivo **`local.feature`**.

O cenário representa a atualização da categoria de um local previamente criado.

2. Ação (WHEN) – Atualizar o Local

No passo **`update_dynamic_post`** o teste:

- pega do `context` o ID do local que foi criado antes,
- monta a URL completa usando **`base_url`**, **`endpoint`** e o ID,

- cria um payload contendo a nova categoria enviada pelo cenário,

- envia uma requisição **`PATCH`** para atualizar esse campo,
- guarda a resposta no **`context`** para os próximos passos.

Validação do status (THEN)

O passo **`check_status_200`** garante que a API devolveu **status 200**, indicando que a atualização ocorreu sem erro.

Validação de campo que não deve mudar

O passo **`check_field_str_original`** verifica se o campo nome permanece igual ao original, pois o teste atualiza apenas a categoria.
Ele compara o nome retornado pela API com o nome salvo no **`context`**.

2. Usuários (Huan-Cruz)

Os testes foram desenvolvidos utilizando pytest, pytest-bdd e a biblioteca requests.

A abordagem segue o modelo BDD (Behavior Driven Development), onde os requisitos são descritos em linguagem natural através de cenários (Given, When, Then).

Cada cenário descrito no arquivo **`.feature`** aciona funções Python responsáveis por executar requisições HTTP reais na API, validar respostas e registrar os resultados.

O objetivo geral é garantir que o CRUD de usuários funcione corretamente:

- Criar
- Listar todos
- Listar por ID
- Atualizar
- Deletar

Arquivo de Cenários – TCs-Huan-Cruz.feature

Este arquivo descreve os casos de teste no formato Gherkin.
Ele não contém código executável – apenas o comportamento esperado da aplicação.

Estrutura do arquivo:

No início do arquivo, temos um *Background*, que é executado antes de cada cenário:

`Given que a url base da API é "http://localhost:3000"`
`And que o endpoint de usuarios é "/usuarios"`

Isso significa:

- Definir a URL base da API
- Definir o endpoint que será usado em todos os cenários

Esses valores serão usados pelos arquivos de teste.

```
Feature: CRUD de Usuários - Huan Cruz
  Como usuário da API
  Quero realizar operações de CRUD no recurso de Usuários

  Background:
    Given que a url base da API é "http://localhost:3000"
    And que o endpoint de usuarios é "/usuarios"

  Scenario: Listar todos os Usuários Cadastrados
    Given que tenho usuários cadastrados no sistema
    When envio uma requisição GET para /usuarios
    Then o sistema deve retornar status 200
    And deve retornar uma lista de usuários

  Scenario: Listar um usuário ao buscar por ID
    Given que tenho usuários cadastrados no sistema
    When pesquiso pelo usuário com ID 2
    Then o sistema deve retornar status 200
    And o campo "usuario_id" deve ser igual a 2

  Scenario: Criar um novo usuário com sucesso
    When envio um novo usuário com nome "Usuario Teste", email "usuario.teste@example.com" e senha "123456"
    Then o sistema deve retornar status 201
    And o campo "nome" deve ser igual a "Usuario Teste"

  Scenario: Atualizar um usuário existente com sucesso
    Given que tenho usuários cadastrados no sistema
    When atualizo o usuário criado anteriormente com o novo nome "Nome Atualizado"
    Then o sistema deve retornar status 200
    And o campo "nome" deve ser igual a "Nome Atualizado"

  Scenario: Deletar um usuário existente com sucesso
    Given que tenho usuários cadastrados no sistema
    When removo o usuário criado anteriormente
    Then o sistema deve retornar status 204
    And ao pesquisar pelo ID removido o sistema deve retornar 404

```

Cenários

1. Listar todos os usuários

Descrição:

- O cenário cria um usuário automaticamente
- Envia um GET para **`/usuarios`**
- Verifica se o status é 200
- Verifica se a resposta é uma lista

2. Buscar usuário por ID

- Cria um usuário antes
- Envia GET para **`/usuarios/2`**
- Espera status 200
- Valida que o campo **`usuario_id`** é igual a 2

3. Criar um novo usuário

- **Envia POST com nome, email e senha**
- **Verifica status 201**
- **Valida o campo retornado no JSON**

4. Atualizar usuário

- Cria usuário para garantir que existe
- Envia PATCH alterando o nome
- Espera status 200
- Confere que o campo do JSON foi atualizado

5. Deletar usuário

- Cria usuário
- Envia DELETE para **`/usuarios/{id}`**
- Valida o status
- Faz um GET para o mesmo ID e espera 404

3. Arquivo conftest.py

```python
import pytest
import requests
import time
from pytest_bdd import given, parsers

@pytest.fixture
def context():
    return {}

@given(parsers.parse('que a url base da API é "{url}"'))
def set_base_url(context, url):
    context['base_url'] = url

@given(parsers.parse('que o endpoint de usuarios é "{endpoint}"'))
def set_endpoint(context, endpoint):
    context['endpoint'] = endpoint

@given('que tenho usuários cadastrados no sistema')
def create_user_for_setup(context):
    full_url = f"{context['base_url']}{context['endpoint']}"

    email_unique = f"setup_{int(time.time()*1000)}@example.com"

    payload = {
        "nome": "Usuario Setup",
        "email": email_unique,
        "senha_hash": "123456",
        "foto_perfil": ""
    }

    r = requests.post(full_url, json=payload)

    assert r.status_code in (200, 201), f"Falha ao criar usuário de setup: {r.text}"

    result = r.json()
    user_id = result.get("id") or result.get("usuario_id")

    assert user_id is not None, "Resposta não contém ID de usuário"

    context["id_usuario"] = user_id

```

Este é O MAIS IMPORTANTE arquivo de suporte.

Ele contém:

Fixture **`context`**

```
@pytest.fixture
def context():
    return {}
```

Esse contexto é um dicionário compartilhado entre todos os passos do teste.
Ele é o equivalente a uma “memória” para passar valores entre Given → When → Then.

Given: definir URL base

```
@given(parsers.parse('que a url base da API é "{url}"'))
def set_base_url(context, url):
    context['base_url'] = url
```

O valor "**`http://localhost:3000`**" do arquivo **`.feature`** é injetado aqui**.**

Given: definir endpoint

```
@given(parsers.parse('que o endpoint de usuarios é
"{endpoint}"'))
def set_endpoint(context, endpoint):
    context['endpoint'] = endpoint
```

Esse passo armazena o endpoint **`/usuarios`.**

Given: criar usuário automaticamente antes de alguns cenários

```
@given('que tenho usuários cadastrados no sistema')
def create_user_for_setup(context):
```

Sempre que um cenário começa com essa frase, essa função é executada.

O que ela faz?

1. Constrói a URL completa:
   `http://localhost:3000/usuarios`

2. Cria um email único baseado no timestamp (evita duplicidade)

3. Envia um POST real para a API criando um usuário

4. Valida se o status é 200 ou 201

5. Captura o ID retornado pela API e salva em:
   `context["id_usuario"]`

Esse ID será usado nos testes de atualização, busca ou exclusão.

4.test_create_user.py – Criar Usuário

```
import pytest
import requests
import time
from pytest_bdd import given, parsers

@pytest.fixture
def context():
    return {}

@given(parsers.parse('que a url base da API é "{url}"'))
def set_base_url(context, url):
    context['base_url'] = url

@given(parsers.parse('que o endpoint de usuarios é "{endpoint}"'))
def set_endpoint(context, endpoint):
    context['endpoint'] = endpoint
@given('que tenho usuários cadastrados no sistema')
def create_user_for_setup(context):
    full_url = f"{context['base_url']}{context['endpoint']}"

    email_unique = f"setup_{int(time.time()*1000)}@example.com"

    payload = {
        "nome": "Usuario Setup",
        "email": email_unique,
        "senha_hash": "123456",
        "foto_perfil": ""
    }

    r = requests.post(full_url, json=payload)

    assert r.status_code in (200, 201), f"Falha ao criar usuário de
setup: {r.text}"

    result = r.json()
    user_id = result.get("id") or result.get("usuario_id")

    assert user_id is not None, "Resposta não contém ID de usuário"

    context["id_usuario"] = user_id
```

Scenario

Vincula a função `test_run()` ao cenário do .feature.

When

`envio um novo usuário com nome "{nome}", email "{email}" e senha "{senha}"`

O arquivo monta o payload e enviando um POST real.

Then

Verifica que:

- O status da resposta é 201
- O campo retornado é exatamente o mesmo passado no .feature

Este teste valida se o endpoint de criação funciona corretamente.

test_delete_user.py – Deletar Usuário

```python
import requests
from pytest_bdd import scenario, when, then


@scenario("features/TCs-Huan-Cruz.feature", "Deletar um usuário existente com sucesso")
def test_run():
    pass

@when('removo o usuário criado anteriormente')
def delete_user(context):
    id_user = context["id_usuario"]
    url = f"{context['base_url']}{context['endpoint']}/{id_user}"
    context["response"] = requests.delete(url)

@then('o sistema deve retornar status 204')
def status_204(context):

    assert context["response"].status_code in (200, 204)

@then('ao pesquisar pelo ID removido o sistema deve retornar 404')
def ensure_deleted(context):
    id_user = context["id_usuario"]
    url = f"{context['base_url']}{context['endpoint']}/{id_user}"
    r = requests.get(url)
    assert r.status_code == 404
```

Neste caso, o usuário já foi criado no **`conftest.py`**.

When

`removo o usuário criado anteriormente`

O código:

- Recupera o ID criado no contexto
- Monta a URL **`/usuarios/{id}`**
- Envia DELETE

Then

1. Valida status 204 (ou 200 dependendo da API)
2. Realiza novo GET para o mesmo usuário
3. Espera um 404, confirmando exclusão

test_findAll_user.py – Listar Todos

```
import requests
from pytest_bdd import scenario, when, then


@scenario("features/TCs-Huan-Cruz.feature", "Listar todos os Usuários
Cadastrados")
def test_run():
    pass

@when('envio uma requisição GET para /usuarios')
def get_all(context):
    url = f"{context['base_url']}{context['endpoint']}"
    context['response'] = requests.get(url)

@then('o sistema deve retornar status 200')
def status_200(context):
    assert context["response"].status_code == 200

@then('deve retornar uma lista de usuários')
def is_list(context):
    assert isinstance(context["response"].json(), list)
```

Este teste verifica somente:

- Status deve ser 200
- Corpo deve ser uma lista

A requisição é simples, mas valida se o endpoint existe e responde corretamente.

test_findOne_user.py – Buscar por ID

```python
import requests
from pytest_bdd import scenario, when, then, parsers


@scenario("features/TCs-Huan-Cruz.feature", "Listar um usuário ao buscar por ID")
def test_run():
    pass

@when(parsers.parse("pesquiso pelo usuário com ID {id_user}"))
def get_by_id(context, id_user):

    url = f"{context['base_url']}{context['endpoint']}/{id_user}"
    context["response"] = requests.get(url)

@then('o sistema deve retornar status 200')
def status_200(context):
    assert context["response"].status_code == 200

@then(parsers.parse('o campo "{campo}" deve ser igual a 2'))
def validate_field(context, campo):

    assert context["response"].json().get(campo) == 2
```

Aqui o ID é fornecido no próprio .feature:

`When pesquiso pelo usuário com ID 2`

O teste:

- Envia GET para **`/usuarios/2`**

- Espera status 200
- Valida que o campo "**usuario_id**" é igual a 2

Isso verifica:

- A API permite buscar por ID
- O retorno contém o campo esperado

test_update_user.py – Atualizar Usuário

```
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario("features/TCs-Huan-Cruz.feature", "Atualizar um usuário
existente com sucesso")
def test_run():
    pass

@when(parsers.parse('atualizo o usuário criado anteriormente com o novo
nome "{novo_nome}"'))
def update_user(context, novo_nome):
    id_user = context["id_usuario"]
    url = f"{context['base_url']}{context['endpoint']}/{id_user}"

    context["response"] = requests.patch(url, json={"nome": novo_nome})

@then('o sistema deve retornar status 200')
def status_200(context):
    assert context["response"].status_code == 200

@then(parsers.parse('o campo "{campo}" deve ser igual a "{valor}"'))
def check_field(context, campo, valor):
    assert context["response"].json()[campo] == valor
```

When:

Atualiza o usuário criado no setup:

```
context["response"] = requests.patch(url, json={"nome": novo_nome})
```

Then:

- Status deve ser 200
- Campo “nome” deve refletir o novo valor enviado

Este teste garante que o update funciona e que o endpoint PATCH está funcionando.

3. Post - Franco Talles

Métodos de Criação, Busca, Atualização e Deleção de Posts da API

Método	Rota	O que faz
GET	`/post`	Lista todos os posts
GET	`/post/:id`	Lista um post por ID
POST	`/post`	Cria um novo post
PATCH	`/post/:id`	Atualiza um post
DELETE	`/post/:id`	Delete um post

Background para todos os casos

Antes e depois de cada cenário, vai rolar os comandos que estão em ***franco_posts/conftest.py***

Explicação: Esses comandos pegam e salvam em um contexto as variáveis da API e da rota. E a cada cenário, ele vai criar um post, usar esse post criado em cada caso de teste e depois vai excluir ele.

Primeiro Cenário:

Scenario: Listar todos os Posts Cadastrados
Given que tenho posts cadastrados no sistema
When envio uma requisição GET para /post
Then o sistema deve retornar status 200
And deve retornar uma lista de posts
And cada post deve conter os campos "post_id", "descricao", "imagem", "local_id", "usuario_id", "created_at", "updated_at"

Código: ***franco-posts/test_findAll.py***

Explicação: Essa rota é para testar o findAllPosts, entao ele vai requisitar na rota que busca todos os posts, vai verificar se deu tudo certo com o Status 200, vai verificar se retornou uma lista e verificar se cada item da lista contém os campos passados no cenário.

Segundo Cenário:

Scenario: Listar um post ao buscar por ID
Given que tenho posts cadastrados no sistema
When pesquiso pelo post com ID 1
Then o sistema deve retornar status 200
And o campo "post_id" deve ser igual a 1

Código: ***franco-posts/test_findOne.py***

Explicação: Essa rota é para testar o findOnePost, entao ele vai requisitar na rota que busca somente um post por ID único, vai verificar se deu tudo certo com o Status 200 e vai verificar se o campo “usuario_id” é igual ao ID passado

Terceiro Cenário:

Scenario: Criar um novo post com sucesso
When envio um novo post com descricao "Novo Post", usuario 2 e local 3
Then o sistema deve retornar status 201
And o campo "descricao" deve ser igual a "Novo Post"
And o campo "usuario_id" deve ser igual a 2

Código: ***franco-posts/test_create.py***

Explicação: Essa rota é para testar a criação de um novo post, então é feito uma requisição que insere de acordo com os valores do cenário e verifica se deu tudo certo com o Status 201. A rota de criação retorna o objeto que foi criado no banco com seu ID, então é feito a verificação se o campo “descricao” é o mesmo que foi passado e se o campo “usuario_id” também é o mesmo que foi passado.

Quarto Cenário:

Scenario: Atualizar um post existente com sucesso
Given que tenho posts cadastrados no sistema
When atualizo o post criado anteriormente com a nova descricao "100% atualizado"
Then o sistema deve retornar status 200
And o campo "descricao" deve ser igual a "100% atualizado"

Código: ***franco-posts/test_update.py***

Explicação: Esta rota valida a atualização de um registro existente. Conforme mencionado, um post é criado previamente com dados padrão; o teste então envia

uma requisição HTTP PATCH alterando a descrição original para '100% atualizado'. Por fim, valida-se o recebimento do status 200 e verifica-se, no objeto retornado pela API, se a alteração foi persistida corretamente.

Quinto Cenário:

Scenario: Deletar um post existente com sucesso
    Given que tenho posts cadastrados no sistema
    When removo o post criado anteriormente
    Then o sistema deve retornar status 200
    And ao pesquisar pelo ID removido o sistema deve retornar 404

Código: ***franco-posts/test_delete.py***

Explicação: Essa rota é para testar o delete de um post. Como dito anteriormente, um post sempre é criado para testar ele. Então, com o id do post criado, você manda para a rota de delete esse id, verifica se deu tudo certo com o status 200 e após isso, para fins de validação se ocorreu tudo certo, o teste verifica se o id do post existe. Se deu tudo certo, deve retornar o status 404.

4. Comentários (Joao-Menezes)

Conftest.py

```python
import pytest
import requests
from pytest_bdd import given, parsers

@pytest.fixture
def context():
    return {}

@given(parsers.parse('que a url base da API é "{url}"'))
def set_base_url(context, url):
    context['base_url'] = url

@given(parsers.parse('que o endpoint de comentarios é "{endpoint}"'))
def set_endpoint(context, endpoint):
    context['endpoint'] = endpoint

@given(parsers.parse('que tenho comentarios cadastrados no sistema'))
def ensure_comentarios_exist(context):
    full_url = f"{context['base_url']}{context['endpoint']}"

    novo_comentario = {
        "comentario": "Comentario criado",
        "usuario_id": 4,
        "post_id": 2,
    }

    response = requests.post(full_url, json=novo_comentario)

    assert response.status_code == 201, "Falha ao criar comentario"

    context['id_comentario_criado'] = response.json().get('comentario_id')
```

O arquivo define alguns passos de teste usando pytest e pytest-bdd para validar a criação de comentários na API Locus.

1. **Fixture `context()`**

   Cria um dicionário vazio que é usado para guardar informações compartilhadas entre os passos do teste.

2. Passo **`set_base_url`**

   Recebe a URL base da API e salva essa informação no contexto.

3. Passo **`set_endpoint`**

   Guarda no contexto qual é o endpoint específico responsável pelos

comentários.

4. Passo **`ensure_comentarios_exist`**

   Monta a URL completa juntando `base_url` + `endpoint` e faz uma requisição POST criando um comentário de teste.
   Depois verifica se a API respondeu com status 201 e guarda no contexto o `id` do comentário criado.

test_create.py

```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('features/comentarios.feature', 'Criar um novo comentario')
def test_run():
    pass

@when(parsers.parse('envio um novo comentario "{comentario}" do usuario {usuario_id:d} no post {post_id:d}'))
def send_comentario_request(context, comentario, usuario_id, post_id):
    full_url = f"{context['base_url']}/comentarios"

    payload = {
        "comentario": comentario,
        "usuario_id": usuario_id,
        "post_id": post_id
    }

    context['response'] = requests.post(full_url, json=payload)
    print(context['response'].json())


@then('o sistema deve retornar status 201')
def check_status_201(context):
    assert context['response'].status_code == 201


@then(parsers.parse('o campo "{campo}" deve ser igual a "{valor}"'))
def check_field_str_value(context, campo, valor):
    json_data = context['response'].json()
    assert json_data[campo] == valor


@then(parsers.parse('o campo "{campo}" deve ser igual a {valor:d}'))
def check_field_int_value(context, campo, valor):
    json_data = context['response'].json()
    assert json_data[campo] == valor
```

Este teste automatiza o cenário BDD responsável por validar o processo de criação de um novo comentário no sistema. Ele utiliza pytest-bdd para organizar as ações e as verificações realizadas.

1. Associação ao cenário
   A função **`test_run()`** apenas vincula este arquivo ao cenário descrito em **`comentarios.feature`**, que trata especificamente da criação de um comentário.

2. Envio da requisição de criação **(`@when`)**
   Na etapa **`send_comentario_request`**, são recebidos o texto do comentário, o ID do usuário e o ID do post definido no cenário.

   - A URL da API é montada usando **`base_url`** e o endpoint `/comentarios`.
   - É criado um payload com os dados necessários para o novo comentário.
   - Em seguida, o teste envia uma requisição POST e armazena a resposta no **`context`**.

3. Validação do status da criação **(`@then`)**
   A etapa **`check_status_201`** verifica se a API retorna **HTTP 201**, que é o status esperado quando um recurso é criado com sucesso.

4. Verificação dos dados retornados – campos de texto **(`@then`)**
   A função **`check_field_str_value`** confirma se um campo específico da resposta possui um valor textual igual ao que foi passado no cenário. Isso assegura que o dado foi registrado corretamente.

5. Verificação dos dados retornados – campos numéricos **(`@then`)**
   A etapa **`check_field_int_value`** faz a mesma validação para campos numéricos (como IDs), garantindo que o valor retornado pela API corresponde ao enviado na criação.

test_delete.py

```python
import requests
from pytest_bdd import scenario, when, then

@scenario('features/comentarios.feature', 'Deletar um comentário existente')
def test_run():
    pass

@when('removo o comentário criado anteriormente')
def delete_dynamic_comentario(context):
    id_comentario = context['id_comentario_criado']

    print(id_comentario)

    full_url = f"{context['base_url']}{context['endpoint']}/{id_comentario}"

    context['response'] = requests.delete(full_url)

@then('o sistema deve retornar status 200')
def check_status_200(context):
    assert context['response'].status_code == 200

@then('ao pesquisar pelo ID removido o sistema deve retornar 404')
def verify_item_deleted(context):
    id_comentario = context['id_comentario_criado']
    full_url = f"{context['base_url']}{context['endpoint']}/{id_comentario}"

    response = requests.get(full_url)
    assert response.status_code == 404
```

Este conjunto de etapas implementa o cenário BDD responsável por validar a exclusão de um comentário já existente no sistema. O teste utiliza pytest-bdd para estruturar as ações de remoção e as verificações necessárias.

1. Associação ao cenário
   A função **`test_run()`** apenas conecta o arquivo ao cenário descrito no **`.feature`** que trata da remoção de um comentário previamente criado.

2. Remoção do comentário **(`@when`)**
   A função **`delete_dynamic_comentario`** recupera do contexto o ID do comentário criado anteriormente.

   - Com esse ID, é montada a URL completa para a chamada exclusão.
   - Em seguida, o teste envia uma requisição DELETE para a API e armazena a resposta no **`context`**.

3. Validação do status da remoção **(`@then`)**
   A etapa **`check_status_200`** confirma que a API responde com HTTP 200, indicando que a remoção foi processada corretamente.

4. Verificação de que o item não existe mais **(`@then`)**
   Por fim, a função **`verify_item_deleted`** faz uma nova requisição GET usando o mesmo ID.

   - O objetivo é garantir que o objeto realmente foi excluído.
   - A API deve retornar **404**, indicando que o comentário não está mais disponível.

test_findAll.py

```python
import requests
from pytest_bdd import scenario, given, when, then, parsers

@scenario('features/comentarios.feature', 'Listar todos os Comentários Cadastrados')
def test_run():
    pass

@given('que tenho comentarios cadastrados no sistema')
def set_endpoint(context):
    context['endpoint'] = "/comentarios"

@when(parsers.parse('envio uma requisição GET para /comentarios'))
def send_get_request(context):
    full_url = f"{context['base_url']}{context['endpoint']}"
    context['response'] = requests.get(full_url)
    print(context['response'].json())

@then(parsers.parse('o sistema deve retornar status 200'))
def check_status_200(context):
    assert context['response'].status_code == 200

@then(parsers.parse('deve retornar uma lista de comentários'))
def check_response_is_list(context):
    json_data = context['response'].json()
    assert isinstance(json_data, list)

@then(parsers.parse('cada comentário deve conter os campos "{campos_string}"'))
def check_comentarios_schema(context, campos_string):
    json_data = context['response'].json()

    campos_esperados = {campo.strip().replace('"', '') for campo in campos_string.split(',')}

    for comentario in json_data:
        chaves_recebidas = set(comentario.keys())
        assert chaves_recebidas == campos_esperados, \
            f"Esperado: {campos_esperados}, Recebido: {chaves_recebidas}"
```

Este conjunto de passos implementa o cenário BDD responsável por validar a listagem completa dos comentários cadastrados no sistema. O teste utiliza pytest-bdd para estruturar as etapas e garantir que a API esteja retornando os dados de forma consistente.

1. Associação ao cenário
   A função **`test_run()`** apenas vincula o arquivo ao cenário “Listar todos os

Comentários Cadastrados”, descrito no arquivo **`.feature`**.

2. Configuração do endpoint **(`@given`)**
   A etapa **`set_endpoint`** define no contexto o endpoint base utilizado para as operações de consulta (`/comentarios`). Isso permite que as próximas etapas montem a URL final corretamente.

3. Envio da requisição GET **(`@when`)**
   A função **`send_get_request`** monta a URL completa combinando **`base_url`** e o endpoint configurado.

   - Envia uma requisição GET para a API.
   - Armazena a resposta no **`context`**, permitindo o uso do retorno nas validações seguintes.

4. Validação do status HTTP (**`@then`**)
   Em **`check_status_200`**, o teste garante que a API responde com HTTP 200, indicando que a listagem foi executada com sucesso.

5. Verificação de que o retorno é uma lista **(`@then`)**
   A etapa **`check_response_is_list`** acessa o JSON retornado é válida se ele está no formato de lista, já que a API deve retornar vários comentários.

6. Verificação do esquema dos comentários **(`@then`)**
   Por fim, **`check comentários schema`** válida se cada comentário retornado possui todos os campos obrigatórios informados no cenário.

   - Os campos esperados são extraídos da string recebida pelo passo.
   - Para cada comentário, é feita uma comparação entre as chaves retornadas e as previstas.
   - Caso falte algum campo, o teste falha mostrando a diferença.

test_findOne.py

```python
import requests
from pytest_bdd import scenario, given, when, then, parsers

@scenario('features/comentarios.feature', 'Listar um comentário ao buscar por ID')
def test_run():
    pass

@given('que tenho comentarios cadastrados no sistema')
def set_endpoint(context):
    context['endpoint'] = "/comentarios"

@when(parsers.parse('pesquiso pelo comentário com ID {id_comentario}'))
def get_comentario_by_id(context, id_comentario):
    full_url = f"{context['base_url']}{context['endpoint']}/{id_comentario}"
    context['response'] = requests.get(full_url)
    print(context['response'].json())

@then(parsers.parse('o sistema deve retornar status 200'))
def check_status_200(context):
    assert context['response'].status_code == 200

@then(parsers.parse('o campo "{campo}" deve ser igual a {valor:d}'))
def check_field_int_value(context, campo, valor):
    json_data = context['response'].json()
    assert json_data[campo] == valor
```

Este código implementa o cenário BDD responsável por validar a busca de um comentário específico na API, utilizando o pytest-bdd para estruturar as etapas.

1. Associação ao cenário
   A função **`test_run()`** apenas conecta o teste ao cenário definido no arquivo **`comentarios.feature`**, que descreve a operação de listar um comentário pelo seu ID.

2. Definição do endpoint **(`@given`)**
   Na etapa **`set_endpoint`**, é configurado no contexto o endpoint base utilizado para as operações de comentário (`/comentarios`). Isso garante que as próximas etapas montem a URL corretamente.

3. Busca do comentário **(`@when`)**
   A função `get_comentario_by_id` monta a URL completa combinando `base_url`, o endpoint configurado e o ID informado no cenário.

   - Em seguida, envia uma requisição GET para a API.

- A resposta recebida é armazenada no **`context`** para ser usada nas validações seguintes.

4. Validação do retorno **(`@then`)**
   A etapa **`check_status_200`** confirma que a API retorna o código 200, indicando que o comentário foi encontrado com sucesso.

5. Verificação dos dados retornados **(`@then`)**
   A função **`check_field_int_value`** acessa o corpo JSON da resposta e valida se um campo específico possui o valor esperado (geralmente usado para conferir se o ID retornado condiz com o ID solicitado).

test.update.py

```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('features/comentarios.feature', 'Atualizar um comentario existente')
def test_run():
    pass

@when(parsers.parse('atualizo o comentário criado anteriormente com o novo texto "{novo_comentario}"'))
def update_dynamic_comentario(context, novo_comentario):
    id_comentario = context['id_comentario_criado']

    full_url = f"{context['base_url']}{context['endpoint']}/{id_comentario}"

    payload = {
        "comentario": novo_comentario,
    }

    context['response'] = requests.patch(full_url, json=payload)
    print(context['response'].json())

@then('o sistema deve retornar status 200')
def check_status_200(context):
    assert context['response'].status_code == 200

@then(parsers.parse('o campo "{campo}" deve ser igual a "{valor}"'))
def check_field_str_value(context, campo, valor):
    json_data = context['response'].json()
    assert json_data[campo] == valor
```

O arquivo implementa um cenário de teste BDD usando pytest-bdd para validar a atualização de um comentário já existente na API.

1. **Carregamento do cenário**
   A função **`test_run()`** apenas vincula o teste ao cenário “Atualizar um comentário existente”, descrito no arquivo **`comentarios.feature`.**
2. **Etapa de atualização (`@when`)**
   A função **`update_dynamic_comentario`** é responsável por enviar a requisição PATCH para atualizar o texto de um comentário já criado anteriormente.

   - Recupera o ID do comentário salvo no contexto.
   - Monta a URL usando os valores de `base_url` e `endpoint`.
   - Envia a requisição passando o novo texto no corpo.
   - Armazena a resposta no **`context`**.

3. **Validação do status (`@then`)**
   A função **`check_status_200`** garante que a API retornou HTTP 200, indicando que a atualização foi processada com sucesso.

4. **Validação do campo alterado (`@then`)**
   A função **`check_field_str_value`** verifica se o campo informado realmente foi atualizado na resposta JSON, comparando o valor retornado com o esperado.

5. Seguidores - David Alejandro Acosta Noguera

```
Feature: CRUD de Seguidores - David Acosta
    Como usuário da API
    Quero realizar operações de CRUD no recurso de Posts

  Background:
    Given que a url base da API é "http://localhost:3000"
    And que o endpoint de seguidores é "/seguidores"

  Scenario: Seguir um Usuário com Sucesso
    Given que eu tenho 2 usuarios cadastrados no sistema
    When eu envio uma requisição POST para "/seguidores" com usuario "1" seguindo usuario "2"
    Then o sistema deve retornar status 201
    And o campo "seguidor_id" deve ser igual a "1"
    And o campo "seguido_id" deve ser igual a "2"

  Scenario: Deixar de Seguir um Usuário com Sucesso
    Given que eu sigo o usuario "2" com o meu usuario "1"
    When eu envio uma requisição DELETE para "/seguidores" com usuario "1" deixando de seguir "2"
    Then o sistema deve retornar status 200
    And ao tentar buscar o seguidor "1" seguindo "2" o sistema deve retornar 404

  Scenario: Listar Usuários Seguidos por um Usuário
    Given que o usuario "3" segue os usuarios "1" e "2"
    When eu envio uma requisição GET para "/seguidores" do usuario "3"
    Then o sistema deve retornar status 200
    And o retorno deve ser uma lista de 2 seguidos
    And o campo "seguido_id" deve conter "1" e "2"

  Scenario: Tentar Seguir um Usuário que Já é Seguido (Conflito 409)
    Given que o usuario "1" já segue o usuario "2"
    When eu envio uma requisição POST para "/seguidores" com usuario "1" seguindo usuario "2"
    Then o sistema deve retornar status 409

  Scenario: Tentar Deixar de Seguir uma Relação Inexistente (404)
    Given que o usuario "1" nao segue o usuario "9999"
    When eu envio uma requisição DELETE para "/seguidores" com usuario "1" deixando de seguir "9999"
    Then o sistema deve retornar status 404
```

1. Cenários BDD Este arquivo BDD (`.feature`) em Português descreve os requisitos para o CRUD de Seguidores em uma API (endpoint: `/seguidores`).

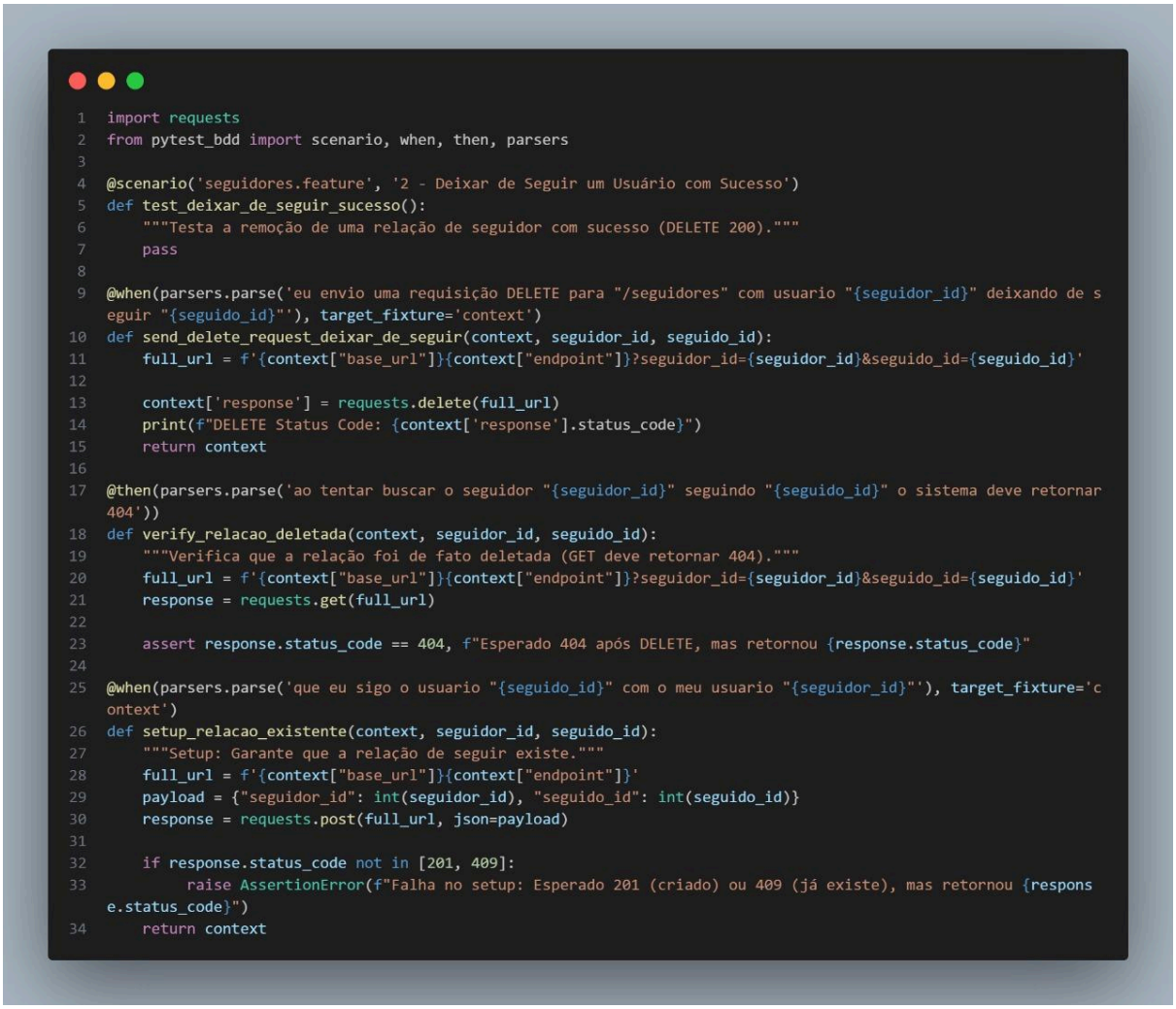
- **Endpoint:** `/seguidores` **(Base URL:** `http://localhost:3000`**)**
- **Cenários Principais:**
    - **Seguir um Usuário com Sucesso: Faz um POST e espera Status 201.**
    - **Deixar de Seguir um Usuário com Sucesso: Faz um DELETE e espera Status 200, e uma busca (GET) subsequente retorna 404.**
    - **Listar Usuários Seguidos por um Usuário: Faz um GET e espera Status 200 com uma lista de seguidores.**
    - **Tentar Seguir um Usuário que Já é Seguido (Conflito): Faz um POST e espera Status 409.**
    - **Tentar Deixar de Seguir uma Relação Inexistente: Faz um DELETE e espera Status 404.**

```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('seguidores.feature', '1 - Seguir um Usuário com Sucesso')
def test_seguir_sucesso():
    """Testa a criação de uma relação de seguidor com sucesso (POST 201)."""
    pass

@when(parsers.parse('eu envio uma requisição POST para "/seguidores" com usuario "{seguidor_id}" seguindo usuario "{seguido_id}"'), target_fixture='context')
def send_post_request_seguir(context, seguidor_id, seguido_id):
    full_url = f'{context["base_url"]}{context["endpoint"]}'

    payload = {
        "seguidor_id": int(seguidor_id),
        "seguido_id": int(seguido_id)
    }

    context['response'] = requests.post(full_url, json=payload)
    print(f"POST Response: {context['response'].json()}")
    return context
```

Teste: Seguir um Usuário com Sucesso

Este código implementa o passo `@when` para o cenário **"1 - Seguir um Usuário com Sucesso"** (POST 201).

- **Função:** `send_post_request_seguir`
- **Ação:** Envia uma requisição **POST** para o endpoint `/seguidores` com um `payload` contendo o `seguidor_id` (quem segue) e o `seguido_id` (quem é seguido).
- **Resultado:** Armazena o objeto de **resposta** na variável de contexto para que o passo `@then` possa verificar o *Status Code 201* e os dados da resposta.



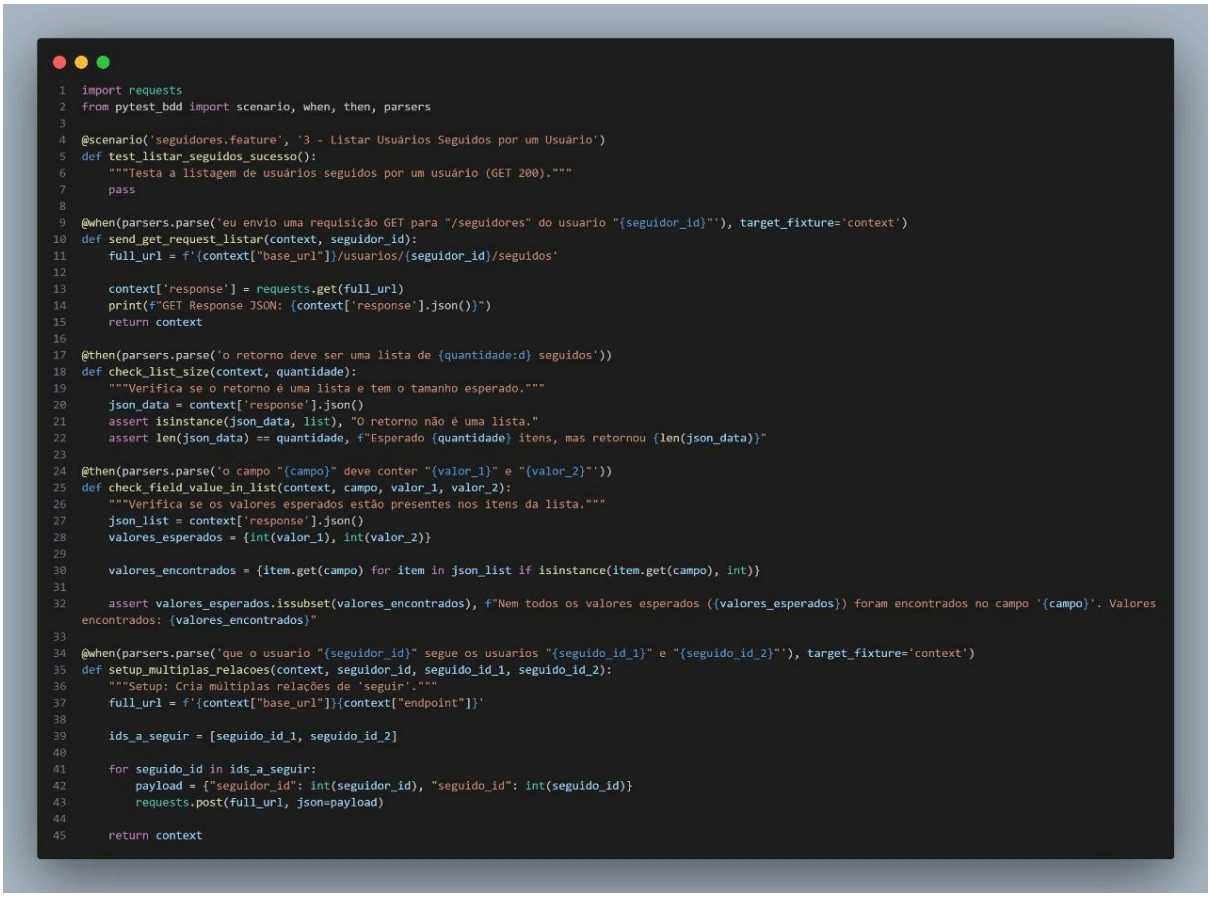
```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('seguidores.feature', '2 - Deixar de Seguir um Usuário com Sucesso')
def test_deixar_de_seguir_sucesso():
    """Testa a remoção de uma relação de seguidor com sucesso (DELETE 200)."""
    pass

@when(parsers.parse('eu envio uma requisição DELETE para "/seguidores" com usuario "{seguidor_id}" deixando de seguir "{seguido_id}"'), target_fixture='context')
def send_delete_request_deixar_de_seguir(context, seguidor_id, seguido_id):
    full_url = f'{context["base_url"]}{context["endpoint"]}?seguidor_id={seguidor_id}&seguido_id={seguido_id}'

    context['response'] = requests.delete(full_url)
    print(f"DELETE Status Code: {context['response'].status_code}")
    return context

@then(parsers.parse('ao tentar buscar o seguidor "{seguidor_id}" seguindo "{seguido_id}" o sistema deve retornar 404'))
def verify_relacao_deletada(context, seguidor_id, seguido_id):
    """Verifica que a relação foi de fato deletada (GET deve retornar 404)."""
    full_url = f'{context["base_url"]}{context["endpoint"]}?seguidor_id={seguidor_id}&seguido_id={seguido_id}'
    response = requests.get(full_url)

    assert response.status_code == 404, f"Esperado 404 após DELETE, mas retornou {response.status_code}"

@when(parsers.parse('que eu sigo o usuario "{seguido_id}" com o meu usuario "{seguidor_id}"'), target_fixture='context')
def setup_relacao_existente(context, seguidor_id, seguido_id):
    """Setup: Garante que a relação de seguir existe."""
    full_url = f'{context["base_url"]}{context["endpoint"]}'
    payload = {"seguidor_id": int(seguidor_id), "seguido_id": int(seguido_id)}
    response = requests.post(full_url, json=payload)

    if response.status_code not in [201, 409]:
        raise AssertionError(f"Falha no setup: Esperado 201 (criado) ou 409 (já existe), mas retornou {response.status_code}")
    return context
```

Deixar de Seguir um Usuário com Sucesso

Este código implementa os passos para o cenário **"2 - Deixar de Seguir um Usuário com Sucesso"** (DELETE 200, GET 404).

- **Passo `@when` (DELETE):**
  - **Função:** `send_delete_request_deixar_de_seguir`
  - **Ação:** Envia uma requisição **DELETE** usando parâmetros de *query* (`?seguidor_id=...&seguido_id=...`) e espera **Status 200** ou **204** (sucesso na remoção).
- **Passo `@then` (GET):**
  - **Função:** `verify_relacao_deletada`
  - **Ação:** Envia uma requisição **GET** com os mesmos parâmetros de *query* para tentar buscar a relação.

    - **Asserção:** Verifica se o *Status Code* é **404**, confirmando que a relação foi de fato removida/não encontrada.
- **Setup (`@when` de contexto):**
    - **Função:** `setup_relacao_existente`
    - **Ação:** Garante que a relação de seguir exista antes do teste de remoção, enviando um **POST** e verificando se o *Status Code* é **201** (criado) ou **409** (já existe).



```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('seguidores.feature', '3 - Listar Usuários Seguidos por um Usuário')
def test_listar_seguidos_sucesso():
    """Testa a listagem de usuários seguidos por um usuário (GET 200)."""
    pass

@when(parsers.parse('eu envio uma requisição GET para "/seguidores" do usuario "{seguidor_id}"'), target_fixture='context')
def send_get_request_listar(context, seguidor_id):
    full_url = f'{context["base_url"]}/usuarios/{seguidor_id}/seguidos'

    context['response'] = requests.get(full_url)
    print(f"GET Response JSON: {context['response'].json()}")
    return context

@then(parsers.parse('o retorno deve ser uma lista de {quantidade:d} seguidos'))
def check_list_size(context, quantidade):
    """Verifica se o retorno é uma lista e tem o tamanho esperado."""
    json_data = context['response'].json()
    assert isinstance(json_data, list), "O retorno não é uma lista."
    assert len(json_data) == quantidade, f"Esperado {quantidade} itens, mas retornou {len(json_data)}"

@then(parsers.parse('o campo "{campo}" deve conter "{valor_1}" e "{valor_2}"'))
def check_field_value_in_list(context, campo, valor_1, valor_2):
    """Verifica se os valores esperados estão presentes nos itens da lista."""
    json_list = context['response'].json()
    valores_esperados = {int(valor_1), int(valor_2)}

    valores_encontrados = {item.get(campo) for item in json_list if isinstance(item.get(campo), int)}

    assert valores_esperados.issubset(valores_encontrados), f"Nem todos os valores esperados ({valores_esperados}) foram encontrados no campo '{campo}'. Valores encontrados: {valores_encontrados}"

@when(parsers.parse('que o usuario "{seguidor_id}" segue os usuarios "{seguido_id_1}" e "{seguido_id_2}"'), target_fixture='context')
def setup_multiplas_relacoes(context, seguidor_id, seguido_id_1, seguido_id_2):
    """Setup: Cria múltiplas relações de 'seguir'."""
    full_url = f'{context["base_url"]}{context["endpoint"]}'

    ids_a_seguir = [seguido_id_1, seguido_id_2]

    for seguido_id in ids_a_seguir:
        payload = {"seguidor_id": int(seguidor_id), "seguido_id": int(seguido_id)}
        requests.post(full_url, json=payload)

    return context
```

Listar Usuários Seguidos por um Usuário

Este código implementa os passos do cenário BDD **"3 - Listar Usuários Seguidos por um Usuário"** (GET 200). O objetivo é verificar se a API retorna corretamente a lista de usuários que um determinado usuário está seguindo.

Setup (Criação de Relações)

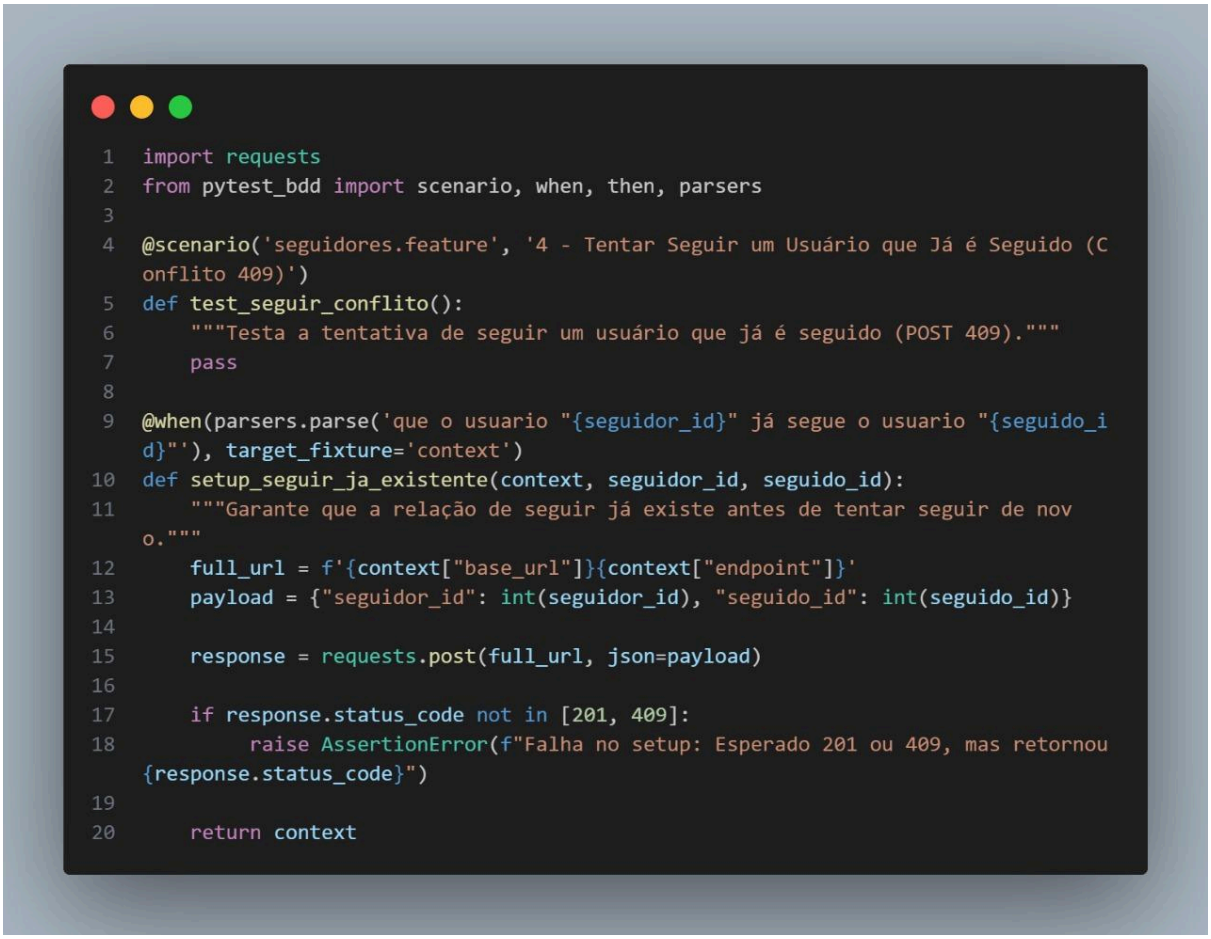
- **Passo `@when` de Contexto:** `setup_multiplas_relacoes`
- **Ação:** Garante que o usuário principal (`seguidor_id`) está seguindo múltiplos usuários (`seguido_id_1` e `seguido_id_2`) antes da execução do teste.
- **Implementação:** Envia requisições **POST** separadas para o endpoint `/seguidores` para criar as relações de seguir necessárias.

Ação Principal (Requisição GET)

- **Passo `@when`:** `send_get_request_listar`
- **Ação:** Envia uma requisição **GET** para o endpoint, usando o ID do seguidor na URL: `/seguidores/{seguidor_id}/seguidos`.
- **Resultado:** Armazena a **resposta** (incluindo a lista de seguidos) na variável de contexto para as verificações subsequentes.

Verificações (Asserções)

- **Passo `@then` (Tamanho da Lista):** `check_list_size`
    - **Ação:** Verifica se a resposta é uma lista e se o **número de itens** na lista de seguidos é igual à `quantidade` esperada.
- **Passo `@then` (Conteúdo da Lista):** `check_field_value_in_list`
    - **Ação:** Verifica se os valores específicos (`valor_1` e `valor_2`) esperados estão **presentes** no campo especificado (`campo`) dentro de cada item da lista retornada. No contexto deste cenário, verifica se os IDs dos usuários seguidos (1 e 2) estão presentes na lista.



```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('seguidores.feature', '4 - Tentar Seguir um Usuário que Já é Seguido (Conflito 409)')
def test_seguir_conflito():
    """Testa a tentativa de seguir um usuário que já é seguido (POST 409)."""
    pass

@when(parsers.parse('que o usuario "{seguidor_id}" já segue o usuario "{seguido_id}"'), target_fixture='context')
def setup_seguir_ja_existente(context, seguidor_id, seguido_id):
    """Garante que a relação de seguir já existe antes de tentar seguir de novo."""
    full_url = f'{context["base_url"]}{context["endpoint"]}'
    payload = {"seguidor_id": int(seguidor_id), "seguido_id": int(seguido_id)}

    response = requests.post(full_url, json=payload)

    if response.status_code not in [201, 409]:
        raise AssertionError(f"Falha no setup: Esperado 201 ou 409, mas retornou {response.status_code}")

    return context
```

Tentar Seguir um Usuário que Já é Seguido

Este código implementa o passo de **setup** e a ação **`@when`** para o cenário **"4 - Tentar Seguir um Usuário que Já é Seguido (Conflito 409)"**.

- **Setup (`@when` de contexto):**
  - **Função:** `setup_seguir_ja_existente`
  - **Ação:** Garante que a relação de seguir já existe antes de tentar criá-la novamente. Envia um **POST** e verifica se o *Status Code* é **201** (para criar) ou **409** (se já existe).
- **Ação Principal (no cenário BDD):**
  - A requisição **POST** duplicada será enviada e o passo **`@then`** verificará se o *Status Code* é **409** (Conflito), conforme esperado no BDD.

```python
import requests
from pytest_bdd import scenario, when, then, parsers

@scenario('seguidores.feature', '5 - Tentar Deixar de Seguir uma Relação Inexistente (404)')
def test_deixar_de_seguir_inexistente():
    """Testa a tentativa de remover uma relação inexistente (DELETE 404)."""
    pass

@when(parsers.parse('que o usuario "{seguidor_id}" nao segue o usuario "{seguido_id}"'), target_fixture='context')
def setup_relacao_inexistente(context, seguidor_id, seguido_id):
    """Setup: Garante que a relação de seguir NÃO existe."""
    full_url = f'{context["base_url"]}{context["endpoint"]}?seguidor_id={seguidor_id}&seguido_id={seguido_id}'
    response = requests.delete(full_url)

    if response.status_code not in [200, 204, 404]:
        raise AssertionError(f"Falha no setup: Esperado 200/204 (deletado) ou 404 (já ausente), mas retornou {response.status_code}")

    return context
```

Tentar Deixar de Seguir uma Relação Inexistente

Este código implementa o passo de **setup** e a ação `@when` para o cenário **"5 - Tentar Deixar de Seguir uma Relação Inexistente (404)"**.

- **Setup (`@when` de contexto):**
  - **Função:** `setup_relacao_inexistente`
  - **Ação:** Garante que a relação de seguir *NÃO* existe antes de tentar removê-la. Envia uma requisição **DELETE** e verifica se o *Status Code* é **200/204** (deletado) ou **404** (já ausente).
- **Ação Principal (no cenário BDD):**
  - A requisição **DELETE** será enviada para a relação inexistente, e o passo `@then` verificará se o *Status Code* é **404** (Não Encontrado), conforme esperado no BDD.