

A ferramenta Behave

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Os objetivos desta parte são: (I) Trabalhar sobre a abordagem de desenvolvimento orientado por comportamento - do inglês Behavior Driven Development (BDD). (II) Entender o que é o behave (III) Criar um projeto utilizando behave. (IV) Utilizar parametrizações com Scenarios Outlines no behave.

Introdução

Assim como na parte 10, esta parte tem como objetivo trabalhar com a abordagem de Desenvolvimento Orientado a Comportamento, do inglês *Behavior Driven Development* (BDD), que é uma abordagem de desenvolvimento ágil que encoraja a colaboração de pessoas técnicas e não técnicas dentro do time de desenvolvimento do software. Nesta parte, a ferramenta a ser utilizada é o behave que é uma ferramenta de BDD bastante conhecida na comunidade Python e que, de maneira semelhante ao `pytest-bdd`, gera o arquivo de testes a partir de uma especificação na linguagem Gherkin (ENGEL *et al*, 2021), (TUTORIALSPPOINT, 2022).

A ferramenta behave

O behave é uma ferramenta de *Behavior Driven Development* (BDD) na linguagem Python. Utilizando-se da linguagem Gherkin para a escrita das funcionalidades, o behave segue a mesma filosofia seguida pelo `pytest-bdd`.

Para a instalação do behave, utilize a seguinte instrução:

pip install behave

Isto é necessário para se utilizar das funções disponíveis na ferramenta behave.

Criando um projeto com behave

Esta seção apresenta uma aplicação prática do BDD sobre um problema conhecido pelos desenvolvedores de software denominado por FizzBuzz, o qual foi estudado na parte 4 deste curso. Este problema consiste em exibir uma lista de números de 1 até 100, um em cada linha, com as seguintes condições:

- Números divisíveis por 3 devem aparecer como "Fizz" ao invés do número;
- Números divisíveis por 5 devem aparecer como "Buzz" ao invés do número;
- Números divisíveis por 3 e por 5 devem aparecer como "FizzBuzz" ao invés do número.

Estrutura de um projeto em behave

Para o primeiro exemplo, observe a estrutura de pastas de acordo com a Figura 11.1.

Figura 11.1. Estrutura de pastas de um projeto com behave



Fonte: do autor, 2022.

O arquivo fizzbuzz.py tem apenas a função de fizzbuzz, como se pode ver na Codificação 11.1. Este é o mesmo código produzido na parte 04 sobre TDD.

Codificação 11.1. fizzbuzz.py

```
def fizzbuzz(n):
```

```
    if n % 3 == 0 and n % 5 == 0:
```

```

    return "FizzBuzz"
if n % 5 == 0:
    return "Buzz"
if n % 3 == 0:
    return "Fizz"
return str(n)

```

Fonte: do autor, 2022.

Para o arquivo `fizzbuzz.feature`, observe a Figura 11.2, que define apenas um cenário de teste para o exercício do fizzbuzz.

Figura 11.2. `fizzbuzz.feature`



Fonte: do autor, 2022.

Com o arquivo `fizzbuzz.feature` criado, agora será necessário criar o arquivo de teste. Diferente do `pytest`, o `behave` não fornece um comando de geração, porém ao tentar executar o `behave` ele devolve a saída como apresenta a Figura 11.3.

Figura 11.3. Sugestão do `behave` como arquivo de teste original



Fonte: do autor, 2022.

Seguindo a sugestão do `behave` observe o código `test_fizzbuzz.py` pela Codificação 11.2.

Codificação 11.2. `test_fizzbuzz.py`

```

"""fizzbuzz feature tests."""
from behave import *
@given(u'o número 5')
def step_impl(context):
    raise NotImplementedError(u'STEP: Given o número 5')
@when(u'eu executo o FizzBuzz')

```

```
def step_impl(context):
    raise NotImplementedError(u'STEP: When eu executo o FizzBuzz')
@then(u'eu tenho como saída Buzz')
def step_impl(context):

    raise NotImplementedError(u'STEP: Then eu tenho como saída Buzz')
```

Fonte: do autor, 2022.

Agora que o arquivo de teste foi criado, basta implementar as regras utilizando a linguagem python. Neste exemplo, é preciso realizar a importação do behave para que os decorators given, when e then possam ser utilizados, e também do arquivo fizzbuzz.py e implementar as etapas especificadas pelo teste. Note que para cada passo do arquivo de requisitos, gerou-se uma função que indica o passo da regra de negócio. Observe a implementação completa do teste na Codificação 11.3.

Codificação 11.3. test_fizzbuzz.py - 2

```
"""fizzbuzz feature tests."""
from behave import *
from fizzbuzz import fizzbuzz
@given(u'o número 5')
def step_impl(context):
    context.num = 5
@when(u'eu executo o FizzBuzz')
def step_impl(context):
    context.saida = fizzbuzz(context.num)
@then(u'eu tenho como saída Buzz')
def step_impl(context):

    assert context.saida == 'Buzz'
```

Fonte: do autor, 2022.

Como se pode perceber, o behave gera em suas funções um contexto que são utilizados nas funções de teste, o qual serve para comunicar uma função de teste com outra.

Para executar o teste observe a Figura 11.4, basta utilizar a instrução `behave`.

Figura 11.4. Executando o teste com behave



Fonte: do autor, 2022.

Observe que ao executar o comando `behave`, o arquivo `fizzbuzz.feature` é mapeado com cada linha do arquivo `test_fizzbuzz.py`. O relatório indica uma *feature*, com um cenário e três passos foram executados e passaram sem erros.

Agrupando vários exemplos de testes

Para o segundo exemplo, observe a estrutura de pastas de acordo com a Figura 11.5. Este exemplo é uma extensão do primeiro exemplo, portanto, segue a mesma estrutura.

Figura 11.5. Estrutura de pastas de um projeto com behave - 2



Fonte: do autor, 2022.

Para este problema considere um plano de testes conforme apresenta a Tabela 11.1. Esse plano de teste contém alguns casos de testes para que se possa pensar nas funções para o tratamento desses casos de testes.

Tabela 11.1. Planejando os casos de testes



Fonte: do autor, 2022.

Com o plano de testes, construa o arquivo `fizzbuzz.feature`, como apresenta a Figura 11.6.

Figura 11.6. Arquivo de requisitos da fizzbuzz



Fonte: do autor, 2022.

Como visto na parte 10, as palavras *Scenario Outline* permitem agrupar cenários de maneira mais concisa, sendo possível realizar testes fornecidos como entrada pelo uso da palavra *Examples*. Os valores de exemplos serão substituídos por parâmetros delimitados por < e >. Então, um *Scenario Outline* deve necessariamente conter um ou mais seções *Examples*.

A Figura 11.4 apresenta um *Scenario Outline* para o problema FizzBuzz, e logo abaixo na seção *Examples* há um conjunto de testes de entrada e também uma saída para ser validada na função de teste.

Para o test_fizzbuzz.py observe a Codificação 11.4.

Codificação 11.4. test_fizzbuzz.py - 3

```
from behave import *
from fizzbuzz import fizzbuzz
@given(u'o número {entrada:d}')
def step_impl(context, entrada):
    context.entrada = entrada
@when(u'eu executo o FizzBuzz')
def step_impl(context):
    context.saida = fizzbuzz(context.entrada)
@then(u'eu tenho como saída {saida}')
def step_impl(context, saida):
    assert saida == context.saida
```

Fonte: do autor, 2022.

O arquivo fizzbuzz.py contém apenas a função de fizzbuzz, como se pode ver na Codificação 11.5.

Codificação 11.5. test_fizzbuzz.py - 4

```
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return "FizzBuzz"
```

```
if n % 5 == 0:
    return "Buzz"
if n % 3 == 0:
    return "Fizz"

return str(n)
```

Fonte: do autor, 2022.

Como se pode observar pela Figura 11.6, existem alguns detalhes para o tratamento de números inteiros. Como a entrada deveria ser necessariamente um número inteiro a cláusula **@given** possui uma parametrização **{entrada:d}**, o qual inclui informação sobre o tipo de dados inteiro especificado pela letra d. Como a saída é apenas uma string, a cláusula **@then** inclui a parametrização **{saída}**, o qual não há necessidade de especificar o tipo de dados, pois por padrão é uma string. Agora, além da variável context, as parametrizações são passadas como parâmetros formais na função. Para executar os testes utilize a instrução behave conforme apresenta a Figura 11.7.

Figura 11.7. Execução dos testes automatizados com behave



Fonte: do autor, 2022.

Considerações finais

Esta parte apresentou a ferramenta behave, que é uma ferramenta que implementa os conceitos de Behave Driven Development (BDD). É uma ferramenta bastante utilizada por desenvolvedores Python e segue a filosofia semelhante ao pytest-bdd. O behave tem uma desvantagem sobre o pytest-bdd pois não há gerador de código para a feature, apenas uma sugestão de código que pode ser copiado para o arquivo de teste.

Referências

ENGEL, Jens; RICE, Benno; JONES, Richard. **Welcome to behave**. 2021. Disponível em: <<https://behave.readthedocs.io/en/latest/>>. Acesso em: 07 ago. 2022.

ENGEL, Jens; RICE, Benno; JONES, Richard. **Behave documentation release 1.2.7.dev2**. 2022. Disponível em: <<https://buildmedia.readthedocs.org/media/pdf/behave/latest/behave.pdf>>. Acesso em: 07 ago. 2022.

PYTEST. **Documentação versão de python 3.7+**. 2015. Disponível em: <<https://docs.pytest.org/en/7.1.x/index.html>>. Acesso em: 11 jun. 2022.

PYTHON. **Documentação versão de python 3.10.5**. The python standard library. Development tools. unittest.mock - mock object library. Versão em inglês. Disponível em: <<https://docs.python.org/3/library/unittest.mock.html>>. Acesso em: 08 jul. 2022.

TUTORIALSPPOINT. **Behave** - quick guide. 2022. Disponível em: <https://www.tutorialspoint.com/behave/behave_quick_guide.htm>. Acesso: 07 ago. 2022.