Utilizando o BDD e o Selenium

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Os objetivos desta parte são: (I) Entender os objetivos dos testes de aceitação (II) Trabalhar sobre a abordagem de desenvolvimento orientado por comportamento - do inglês Behavior Driven Development (BDD). (III) Construir um exemplo simples utilizando o framework Flask para desenvolvimento web. (IV) Realizar testes para uma aplicação web utilizando o Selenium.

Introdução

As partes anteriores introduziram 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, pois com o auxílio de pessoas que conhecem muito bem as regras de negócio é possível, por meio das estórias dos usuários, do inglês *users stories*, ajudar na construção do software. Outra perspectiva da aplicação do BDD é por meio do **teste de aceitação**, o qual é uma abordagem de testes para verificação do comportamento da aplicação, e com isso, assegurar que a entrega para o usuário final é exatamente o que se é esperado.

Este texto pretende aplicar os conceitos de BDD no processo de desenvolvimento de software para web, para isso será necessário utilizar o Selenium para automatizar o teste de software e integrar com uma ferramenta de BDD.

Objetivos do teste de aceitação

Uma vez construída uma boa cobertura de testes unitários, por que você gastaria tempo e esforço para construir um conjunto de testes de aceitação ? Existe uma diferença importante entre testes de unidade e testes de aceitação. Os testes de unidade asseguram que os elementos individuais da sua aplicação trabalham adequadamente de maneira isolada. Os testes de aceitação agrupam um conjunto de unidades que já foram testadas e asseguram que elas trabalham juntas para suportar uma determinada funcionalidade ou serviço. Ou seja, quando você encadeia classes e métodos para criar uma aplicação totalmente funcional, os testes de aceitação verificam que o comportamento da sua aplicação é exatamente como esperado (SALE, 2014).

Os principais objetivos dos testes de aceitação são:

- Assegurar que sua aplicação entrega a funcionalidade como esperado.
- Fornecer uma descrição sobre a funcionalidade sob teste que pode ser lida e entendida por pessoas envolvidas no projeto.
- Incorporar testes de aceitação no processo de desenvolvimento ágil assegura que tenha um código funcionando entregue e uma aplicação funcionando de acordo com a especificação.
- Force um conjunto de testes de regressão, permitindo que você adicione mais funcionalidade e tendo a certeza de que você não alterou ou quebrou um comportamento que foi previamente entregue.
- Elucida questões ou erros na sua aplicação, como um usuário, antes de ser liberado para a produção.

BDD no desenvolvimento web

Esta parte pretende construir um exemplo de um aplicativo que trata de informações de contas bancárias. Como um exemplo fictício, e para efeitos

didáticos os dados armazenados serão apenas a conta e o saldo. Para este exemplo, observe a estrutura de arquivos e diretórios na Figura 12.1.

Figura 12.1. Estrutura de pastas

Fonte: do autor, 2022.

Para este exemplo, deve-se instalar o framework Flask, que é um micro-framework para desenvolvimento web. Para isso, utilize a seguinte instrução:

```
pip install Flask
```

Para o arquivo conta.py observe a Codificação 12.1.

Codificação 12.1. conta.py

```
class Conta:
    def __init__(self, numero, saldo):
        self.numero = numero
        self.saldo = saldo
```

Fonte: do autor, 2022.

Para o arquivo banco.py observe a Codificação 12.2.

Codificação 12.2. banco.py

```
class Banco:
    def __init__(self):
        self.contas = {}

    def incluir_conta(self,conta):
        self.contas[conta.numero] = conta

    def obter_saldo_da_conta(self, numero):
        return self.contas[numero].saldo
```

Fonte: do autor, 2022.

O exemplo apresenta a classe Banco, o qual armazena contas em um dicionário (em memória). Para incluir uma conta no Banco utilize o método

incluir_conta(), e para obter o valor do saldo utilize o método obter_saldo_da_conta().

Para entender a pequena aplicação, a Figura 12.2 apresenta a tela home, que é simplesmente uma tela inicial sem funcionalidade.

Figura 12.2. Tela home

Fonte: do autor, 2022.

Para os testes locais o Flask utiliza por padrão o endereço http://127.0.0.1:5000/. Para cadastrar uma nova conta utilize o endereço http://127.0.0.1:5000/nova_conta conforme apresenta a Figura 12.3, o qual solicita dois campos o número da conta e o saldo.

Figura 12.3. Cadastrar nova conta

Fonte: do autor, 2022.

Observe a Codificação 12.3 para o form_conta.html que constrói a tela da Figura 12.3. O arquivo html possui um form com o id igual a form_conta com o atributo action para o link de nova conta utilizando o método POST. No formulário, observe dois campos de entrada e um botão Salvar.

Codificação 12.3. form_conta.html

```
<!DOCTYPE html>
<html>
   <head>
      <title>Cadastrar conta</title>
   </head>
   <body>
      <form  id="form_conta" action="/nova_conta" method="POST">
         <label>Número da conta:</label>
```

```
                    <input type="text" id="numero" name="numero"
autocomplete="off" /> <br />
        <label>Saldo:</label>
          <input type="text" id="saldo" name="saldo" autocomplete="off"
/>
        <button type="submit">Salvar</button>
      </form>
   </body>
</html>
```

Fonte: do autor, 2022.

Ao clicar no botão salvar, o site armazena a nova conta em memória e direciona para uma nova tela, que é dada pelo endereço http://127.0.0.1:5000/consulta, como apresenta a Figura 12.4.

Figura 12.4. Tela de consulta

Fonte: do autor, 2022.

Observe a Codificação 12.4 para o arquivo form_consulta_conta.html. O arquivo html apresenta o formulário solicitando um número de conta. Ao enviar para o servidor, o programa consulta a base de dados contendo as contas e saldos e caso a consulta obtiver sucesso, o saldo é apresentado na tela, caso contrário a mensagem de conta não encontrada é apresentada.

Codificação 12.4. form_consulta_conta.html

```
<!DOCTYPE html>
<html>
   <head><title>Consultar conta</title></head>
<body>
   <form id="form_consulta_conta">
      <label>Número da conta:</label>
      <input name="numero_conta" type="text">
```

```
    <input type="submit" />
  </form>

  {% if saldo %}
  <p> Saldo: {{ saldo }} </p>
  {% endif %}

  {% if conta_nao_encontrada %}
  <p> Conta não encontrada! </p>
  {% endif %}

</body>
</html>
```

Fonte: do autor, 2022.

Além disso, é possível observar a lista de contas pelo link http://127.0.0.1:5000/contas, que lista todas as contas que foram cadastradas no repositório de dados. A Figura 12.5 apresenta a tela de listagem de contas com uma conta cadastrada.

Figura 12.5. Tela de listagem de contas

Fonte: do autor, 2022.

Observe a Codificação 12.5 para o arquivo lista_conta.html. O arquivo html apresenta uma tabela cujo conteúdo é uma lista de contas que apresenta o número e o saldo.

Codificação 12.5. lista_conta.html

```
<!DOCTYPE html>
<html>
  <head><title>Contas</title></head>
  <body>
    <h1>Contas:</h1>
    <table>
```

```
        <tr>
          <td>Número</td>
          <td>Saldo</td>
        </tr>
        {% for conta in contas %}
          <tr>
            <td>{{conta['numero']}}</td>
            <td>{{conta['saldo']}}</td>
          </tr>
        {% endfor %}
      </table>
    </body>
</html>
```

Fonte: do autor, 2022.

O arquivo app.py, Codificação 12.6, contém a programação para o servidor.

Codificação 12.6. app.py

```
from flask import Flask, render_template, request, redirect, url_for

from banco.banco import Banco
from banco.conta import Conta

app = Flask(__name__)
contas_banco = Banco()


@app.route("/")
def home():
    return '<h1>Aplicativo de banco</h1>'

@app.route("/nova_conta", methods = ["GET"])
def form_criar_conta():
    return render_template("form_conta.html")

@app.route("/nova_conta", methods = ["POST"])
```

```
def criar_conta():
    numero = request.form["numero"]
    saldo = int(request.form["saldo"])
    contas_banco.incluir_conta(Conta(numero, saldo))
    return redirect('/consulta')

@app.route("/consulta")
def consulta():
    numero_conta = request.args.get('numero_conta')
    try:
        saldo = contas_banco.obter_saldo_da_conta(numero_conta)
        return render_template('form_consulta_conta.html', saldo = saldo)
    except:
        return render_template('form_consulta_conta.html',
                               conta_nao_encontrada = numero_conta)

@app.route("/contas")
def listar_contas():
    contas = []
    for conta in contas_banco.contas.items():
        c = {}
        c['numero'] = conta[0]
        c['saldo'] = conta[1].saldo
        contas.append(c)

    contas = sorted(contas, key=lambda d: d['numero'])
    return render_template("lista_conta.html", contas = contas)

if __name__ == "__main__":
    app.run(debug=True)
```

Fonte: do autor, 2022.

Como se pode observar, o arquivo app.py possui o código para o tratamento do back end, em que todas as requisições RESTs são tratadas neste arquivo.

Dada a simplicidade do projeto, e também, para facilitar a didática, escolheu-se colocar todo tratamento em um arquivo. No entanto, para projetos grandes, o Flask sugere uma arquitetura de projetos própria, que não vem ao caso para este assunto de testes automatizados.

Pela Codificação 12.6, observe que a função home() devolve uma string que é uma descrição do aplicativo. Após isso, observe que ao acessar o caminho nova_conta, o Flask direciona o método GET para a função form_criar_conta o qual direciona para a página form_conta.html. Entretanto, no caminho nova_conta e método POST, o Flask direciona para o método criar_conta que aloca uma nova conta e armazena no banco de contas.

Para o caminho consulta, o Flask direciona para o método consulta(), o qual verifica se o número digitado no formulário encontra-se no repositório de contas. Caso seja encontrado, a função direciona para o arquivo form_consulta_conta.html com o saldo. Caso contrário, direciona para esse mesmo arquivo com a mensagem de que a conta não foi encontrada.

Por fim, o caminho contas direciona para a função listar_contas, que realiza uma iteração sobre as contas do repositório de contas e constrói uma lista adequada para ser apresentada no arquivo lista_conta.html.

Para executar o programa utilize, na linha de comando, a instrução flask run, conforme apresenta a Figura 12.6. O endereço e a porta são apresentados, com isso basta acessar esse link pelo navegador para a tela home.

Figura 12.6. Executando o programa em Flask

Fonte: do autor, 2022.

Para o arquivo de feature observe a Figura 12.7.

Figura 12.7. Arquivo banco.feature

Fonte: do autor, 2022.

Com o arquivo de banco.feature gerado, agora é necessário criar o arquivo de testes, para isso utilize a seguinte instrução:

pytest-bdd generate tests/features/banco.feature > tests/test_consulta_banco.py

Como resultado, observe a Codificação 12.7.

Codificação 12.7. test_consulta_banco.py

```
"""Aplicação web para banco feature tests."""

from pytest_bdd import (
    given,
    scenario,
    then,
    when,
)

@scenario('features/banco.feature', 'Obter o saldo da conta')
def test_obter_o_saldo_da_conta():
    """Obter o saldo da conta."""


@given('eu visito o app do banco')
def eu_visito_o_app_do_banco():
    """eu visito o app do banco."""
    raise NotImplementedError


@when('eu entro com o número da conta "1111"')
def eu_entro_com_o_número_da_conta_1111():
    """eu entro com o número da conta "1111"."""
    raise NotImplementedError


@then('eu obtenho saldo igual a 50')
```

```
def eu_obtenho_saldo_igual_a_50():
    """eu obtenho saldo igual a 50."""
    raise NotImplementedError
```

Fonte: do autor, 2022.

Com o arquivo de teste foi gerado, basta implementar as regras utilizando a linguagem python. Observe a Codificação 12.8 para a implementação do teste.

Codificação 12.8. test_consulta_banco.py - 2

```
"""Aplicação web para banco feature tests."""

import pytest
from pytest_bdd import (
    given,
    scenario,
    then,
    when,
)

from selenium.webdriver import Chrome
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options

from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.service import Service

@pytest.fixture
def navegador():
    opts = Options()
    opts.headless = True
    servico = Service(ChromeDriverManager().install())
    chrome = Chrome(service=servico, options=opts)
    yield chrome
    chrome.quit()
```

```
@scenario('features/banco.feature', 'Obter o saldo da conta')
def test_obter_o_saldo_da_conta(navegador):
    """Obter o saldo da conta."""

@given('eu visito o app do banco')
def eu_visito_o_app_do_banco(navegador):
    """eu visito o app do banco."""
    navegador.get('http://127.0.0.1:5000/nova_conta')
    form = navegador.find_element(By.ID, "form_conta")
    num_conta = navegador.find_element(By.ID, "numero")
    num_conta.send_keys("1111")
    saldo = navegador.find_element(By.ID, "saldo")
    saldo.send_keys("50")
    form.submit()

@when('eu entro com o número da conta "1111"')
def eu_entro_com_o_número_da_conta_1111(navegador):
    """eu entro com o número da conta "1111"."""
    navegador.get('http://127.0.0.1:5000/consulta')
    form = navegador.find_element(By.ID, "form_consulta_conta")
    num_conta = navegador.find_element(By.NAME, "numero_conta")
    num_conta.send_keys("1111")
    form.submit()

@then('eu obtenho saldo igual a 50')
def eu_obtenho_saldo_igual_a_50(navegador):
    """eu obtenho saldo igual a 50."""
    result = navegador.find_element(By.TAG_NAME, 'p')

    assert result.text == 'Saldo: 50'
```

Fonte: do autor, 2022.

Agora execute o teste utilizando o pytest, conforme apresenta a Figura 12.8.

Figura 12.8. Executando o teste

Fonte: do autor, 2022.

Considerações finais

Esta parte apresentou um exemplo de desenvolvimento de software utilizando Flask. Para os testes utilizou-se a abordagem de *Behave Driven Development* (BDD). Para os testes do sistema, utilizou-se o Selenium para interagir com as páginas web, e o *test runner* utilizado foi o pytest, que é uma ferramenta versátil para vários tipos de testes em aplicações Python. Com isso, conclui-se o curso de testes automatizados de software, há muito que se aprender sobre automação, mas com esses passos iniciais é possível aplicar em diversos cenários de desenvolvimento de sistemas.

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 ago. 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: 08 jul. 2022.

SALE, D. **Testing python:** applying unit testing, TDD, BDD, and accepting testing. Wiley, 2014.

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