Qualidade e Testes

Não podemos considerar nenhum projeto 100% pronto se não tiver testes!

Os testes podem ser escritos em diferentes abordagens, podemos escrever os testes antes, durante ou depois do desenvolvimento, o importante é que eles existam.

Qualidade

Qualidade de Software não é uma coisa relacionada apenas a testes, existem algumas categorias onde a qualidade pode ser aplicada.

- Qualidade de Produto
- Qualidade de Projeto
- Qualidade de Infra
- **Qualidade de Código**
- **Qualidade de software**

Existem profissionais que se especializam a area de qualidade em uma ou em todas as suas vertentes, e este não é um tema só da area de desenvolvimento, qualidade é uma area que existe em empresas de areas diversas como por exemplo na industria.

Qualidade Código

A qualidade de código é uma atividade de responsabilidade e interesse do time de desenvolvimento para facilitar o dia a dia tornando o código mais legivel, descomplicado e fácil de manter.

Mas também de interesse de quem financia o projeto "donos do produto" pois com um código limpo e bem organizado fica mais barato constriur e manter um time de pessoas desenvolvedoras.

Para Python existem alguns padrões e ferramentas que ajudam a manter o código organizado.

PEP8 https://pep8.org/

A PEP8 é um guia de estilo da linguagem Python que é seguido a risca pela maior parte da comunidade a recomendação é seguir a PEP8 sempre que possível e a boa notícia é que podemos usar ferramentas para automatizar esse processo, uma delas é chamada de black.

```
black -l 79 beerlog
```

O comando acima irá formatar automaticamente os arquivos para que fique de acordo com as regras de estilo.

> **NOTA** o `black` não irá renomear seus objetos, as regras de nomenclatura devem ser arrumadas manualmente ou utilizando ferramentas como o `autopep8`

Regras de nomenclatura:

```
# Funções devem usar snake_case >
def nomeDeFuncao()                  >       def nome_de_funcao()
# Classes devem usar PascalCase >
class nomedeclasse                  >       class NomeDeClasse
# Variáveis usam snake_case >
MeuNomeCompleto='Bruno Rocha'       >       meu_nome_completo = "Bruno Rocha"
# Constantes usam SCREAMING_SNAKE_CASE >
screenSize=(800,800)                >       SCREEN_SIZE = 800, 800
```

Além de verificar pelas regras de formatação também é possivel usar ferramentas de análise **linters** para verificar a complexidade do código e até questões de segurança, ferramentas como `flake8`, `pylint`, `bandit`, `mccabe`, `mypy` etc...

Exemplo de uso de algumas ferramentas:

```
# organizar os imports
isort --profile=black -m 3 beerlog/

# Formatar o código
black -l 79 beerlog tests

# Verificar por erros de estilo
flake8 beerlog
```

Qualidade de Software

Em poucas palavras: Testar e garantir que o software funciona conforme a especificação e necessidade do produto.

Ferramentas

Em Python existem uma série de ferramentas interessantes para testes como **selenium**, **unittest**, **lettuce**, **behave**, **ward** e **pytest**

O **pytest** é o framework de testes mais utilizado pela comunidade e com ele podemos escrever diversos tipos de testes.

```
poetry add pytest --dev
```

O pytest por padrão irá procurar qualquer arquivo que tenha seu nome começado com `test_` e também uma pasta chamada `tests` geralmente os projetos adotam esta padronição, dentro da pasta `tests` iremos criar os nossos testes.

Existem 2 categorias principais para testes:

Testes unitários

É usado para testar funções, classes e objetos **com acesso direto ao nosso código**, vamos criar um arquivo para testar as funções que estão em nosso módulo core.

`tests/test_core.py`

```
from beerlog.core import get_beers_from_database, add_beer_to_database
```

```
def test_add_beer_to_database():
    assert add_beer_to_database("Blue Moon", "Witbier", 10, 3, 6)


def test_get_beers_from_database():
    results = get_beers_from_database()
    assert len(results) > 0
```

Cada função de teste tem por objetivo efetuar chamadas aos objetos da nossa aplicação e efetuar um `assert` para garantir o resultado esperado.

> **NOTA** É boa prática que cada `unit test` tenha apenas um `assert` .

Agora podemos executar `pytest` no terminal para obter o resultado dos testes.

```
$ pytest -v
======================= test session starts =======================
collected 2 items

tests/test_core.py::test_add_beer_to_database PASSED        [ 50%]
tests/test_core.py::test_get_beers_from_database PASSED     [100%]

======================== 2 passed in 0.35s ========================
```

O problema que temos nesse caso é que toda vez que rodamos os testes estamos inserindo novos registros no banco de dados `beerlog.db` e isso não é desejavel.

A primeira coisa que podemos fazer é usar um banco de dados especifico para nossa sessão de testes e como nosso projeto utiliza a lib `Dynaconf` isso é fácil.

```
export BEERLOG_DATABASE__url="sqlite:///testing.db"
pytest -v
```

Porém ainda não está bom, se rodarmos várias vezes os testes o banco de dados `testing.db` irá crescer indefinidamente.

Testes não devem causar efeitos colaterais, os dados de um teste precisam ser isolados no próprio testes e para isso podemos usar as `fixtures` do Pytest.

No arquivo `conftest.py` que está na raiz do projeto podemos confurar fixtures.

```
import pytest
from unittest.mock import patch
from sqlmodel import create_engine
```

```
from beerlog import models


@pytest.fixture(autouse=True, scope="function")
def each_test_uses_separate_database(request):
    tmpdir = request.getfixturevalue("tmpdir")
    test_db = tmpdir.join("beerlog.test.db")
    engine = create_engine(f"sqlite:///{test_db}")
    models.SQLModel.metadata.create_all(bind=engine)
    with patch("beerlog.database.engine", engine):
        yield
```

Agora ao executar os testes teremos uma falha!!!

Calma.. é uma falha esperada

```
pytest -v
======================== test session starts ========================
collected 2 items

tests/test_core.py::test_add_beer_to_database PASSED          [ 50%]
tests/test_core.py::test_get_beers_from_database FAILED       [100%]

============================ FAILURES ============================
____________________ test_get_beers_from_database ____________________

    def test_get_beers_from_database():
        # add_beer_to_database("Blue Moon", "Witbier", 10, 3, 6)
        results = get_beers_from_database()
>       assert len(results) > 0
E       assert 0 > 0
E        +  where 0 = len([])

tests/test_core.py:11: AssertionError
====================== short test summary info ======================
FAILED tests/test_core.py::test_get_beers_from_database - assert...
=================== 1 failed, 1 passed in 0.25s ====================
```

Como os testes agora usam bancos de dados diferentes, se quisermos testar o `get_beers_from_database` teremos que primeiro adicionar cervejas usando o `add_beer_to_database` alterando o `test_core.py`

```
def test_get_beers_from_database():
    add_beer_to_database("Blue Moon", "Witbier", 10, 3, 6)  # NEW
    results = get_beers_from_database()
    assert len(results) > 0
```

E agora sim os testes irão passar e cada teste irá usar um banco de dados isolado.

Testes funcionais (ou de integração)

Estes testes se caracterizam principalmente pela caracteristica de não terem acesso direto ao código do projeto, enquanto no teste de unidade podemos importar coisas com `from beerlog.` os testes funcionais devem **imitar** um usuário ou cliente do projeto e usar protocolos e interfaces.

Em nosso caso por exemplo, o teste funcional pode importar objetos apenas das nossas interfaces como `cli` e `api` mas não deve ter acesso a módulos internos como `core`, `database` e `models`.

Vamos criar um arquivo `tests/test_functional_cli.py` para testar a interface de linha de comando.

Da mesma forma que podemos rodar `beerlog add Skol KornPA --flavor=1 --image=1 --cost=2` diretamente no terminal como um usuário do programa, podemos automatizar no pytest usando o CliRunner da lib Typer.

```
from typer.testing import CliRunner

from beerlog.cli import main

runner = CliRunner()


def test_add_beer():
    result = runner.invoke(main, ["add", "Skol", "KornPA", "--flavor=1", "--ima
    assert result.exit_code == 0
    assert "Beer added" in result.stdout
```

E agora fazemos a mesma coisa para testar a API.

Para testar a API precisamos instalar o cliente `requests`

```
poetry add requests --dev
```

E então em `tests/test_functional_api.py`

```
from fastapi.testclient import TestClient

from beerlog.api import api
```

```
client = TestClient(api)


def test_create_beer_via_api():
    response = client.post(
        "/beers",
        json={
            "name": "Skol",
            "style": "KornPA",
            "flavor": 1,
            "image": 1,
            "cost": 2
        },
    )
    assert response.status_code == 201
    result = response.json()
    assert result["name"] == "Skol"
    assert result["id"] == 1


def test_list_beers():
    response = client.get("/beers")
    assert response.status_code == 200
    result = response.json()
    assert len(result) == 0
```

e então

```
$ pytest -v
======================== test session starts ========================
collected 5 items

tests/test_core.py::test_add_beer_to_database PASSED          [ 20%]
tests/test_core.py::test_get_beers_from_database PASSED       [ 40%]
tests/test_functional_api.py::test_create_beer_via_api PASSED [ 60%]
tests/test_functional_api.py::test_list_beers PASSED          [ 80%]
tests/test_functional_cli.py::test_add_beer PASSED            [100%]

======================== 5 passed in 0.37s ==========================
```

> **NOTA** Repare que em testes das categorias **funcionais**, **integration**, **end-to-end** podemos usar multiplos `assert` dentro de um mesmo teste e o motivo disso é a economia de recursos e também o fato de que para "imitar" os passos de um usuário geralmente multiplas verificações precisam ser feitas.

CI

Continuous Integration é o nome dado a uma prática de rodar testes a cada nova alteração no repositório, a idéia é que desenvolvedores distribuidos possam `integrar` suas alterações continuamente pelo menos uma vez por dia.

Na prática não é bem assim que acontece, mas nós continuamos usando a sigla `CI` na verdade nós resignificamos a sigla `CI` e hoje essa palavra quer dizer `esteira de testes automatizados` .

No Github podemos configurar uma esteira de testes usando github actions.

em `.github/workflows/ci.yaml` podemos declarar no formato YAML o passo a passo para instalar, configurar e testar a aplicação.

```
mkdir -p .github/workflows
```

e então criarmos nesta pasta um arquivo `.github/workflows/main.yaml`

```yaml
name: CI
on:
  push:
    branches:
      - "*"
  pull_request:
    branches:
      - "*"
  workflow_dispatch:

jobs:
  test:
    strategy:
      fail-fast: true
      matrix:
        python-version: ['3.8', '3.10']
        os: [ubuntu-latest]
    runs-on: ${{matrix.os}}
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-python@v2
        with:
          python-version: ${{matrix.python-version}}
      - name: Install Poetry
        run: pip install --upgrade pip && pip install poetry

      - name: Install Project
        run: poetry install

      - name: Look for style errors
        run: poetry run flake8 beerlog
```

```yaml
      - name: Look for auto format errors
        run: poetry run black -l 79 --check --diff beerlog tests

      - name: Run tests
        run: poetry run pytest -v --junitxml=test-result.xml

      - name: publish junit results
        uses: EnricoMi/publish-unit-test-result-action@v1
        if: always()
        with:
          files: test-result.xml
          check_name: Test Result (Python ${{matrix.python-version}})
```