Ferramentas e Práticas de TDD em Python

Patrick Porto

7 anos de experiência em Python

Praticante de TDD há 5 anos

Software Engineer na Globo.com

TALENTOS.GLOBO.COM

VOCÊ NASCEU PRA ISSO!

Agenda

- Introdução ao TDD
- Ferramentas para testes unitários em Python
- Anatomia de um teste
- Abordagens de TDD em Python
- Mock de objetos
- Considerações finais

Teste é um procedimento que conduz a aceitação ou rejeição

Código que não é testado não funciona

Por que escrever Testes Automatizados?

Escrevendo bons Testes Automatizados

- Devemos projetar organicamente com o código, executando e recebendo feedback constante sobre o seu funcionamento.
-
-
-

Escrevendo bons Testes Automatizados

- Devemos projetar organicamente com o código, executando e recebendo feedback constante sobre o seu funcionamento.
- Devemos escrever nossos próprios testes, pois não podemos esperar 20 vezes por dia para outra pessoa escrever um teste.
-
-

Escrevendo bons Testes Automatizados

- Devemos projetar organicamente com o código, executando e recebendo feedback constante sobre o seu funcionamento.
- Devemos escrever nossos próprios testes, pois não podemos esperar 20 vezes por dia para outra pessoa escrever um teste.
- Nosso ambiente de desenvolvimento deve fornecer resposta rápida a pequenas mudanças.
-

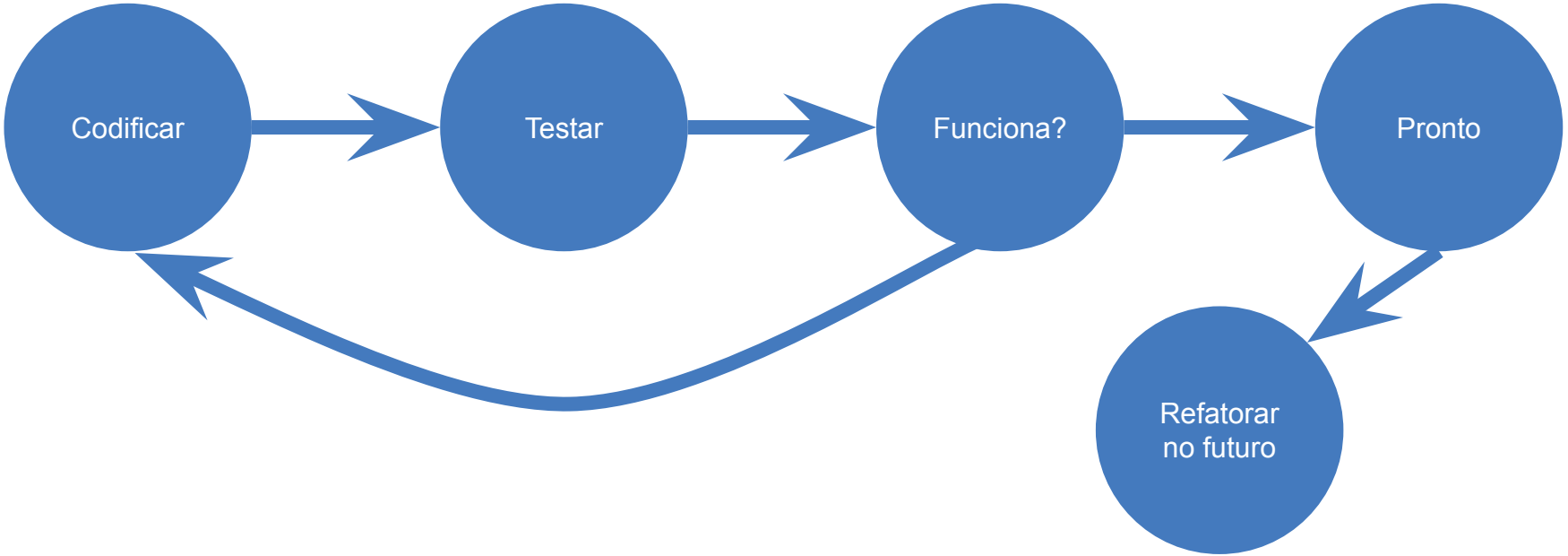
Escrevendo bons Testes Automatizados

- Devemos projetar organicamente com o código, executando e recebendo feedback constante sobre o seu funcionamento.
- Devemos escrever nossos próprios testes, pois não podemos esperar 20 vezes por dia para outra pessoa escrever um teste.
- Nosso ambiente de desenvolvimento deve fornecer resposta rápida a pequenas mudanças.
- Nosso projeto deve consistir em muitos componentes altamente coesos e fracamente acoplados para tornar os testes fáceis.

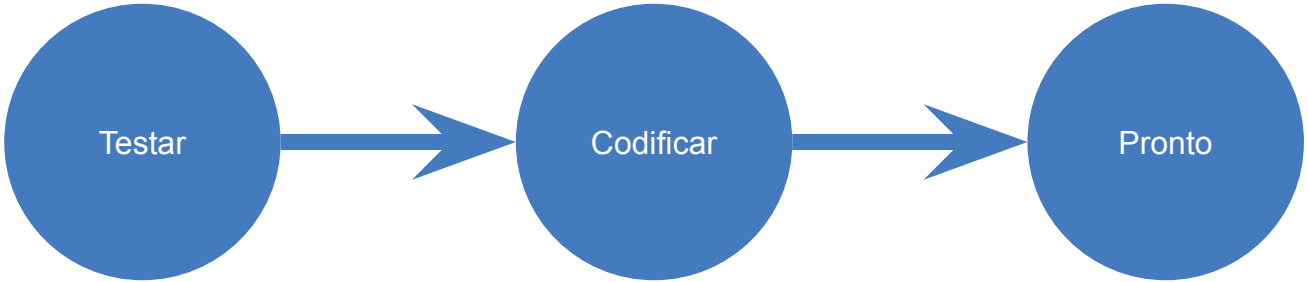
O que é Test-Driven Development?

Test First

Feedback sem TDD



Feedback com TDD



TDD não é Teste Unitário

TDD não é uma Técnica de Teste

O que guia um desenvolvimento que não tem TDD?

"Esse é um problema difícil e eu não consigo ver o fim a partir do começo".
~ Bob

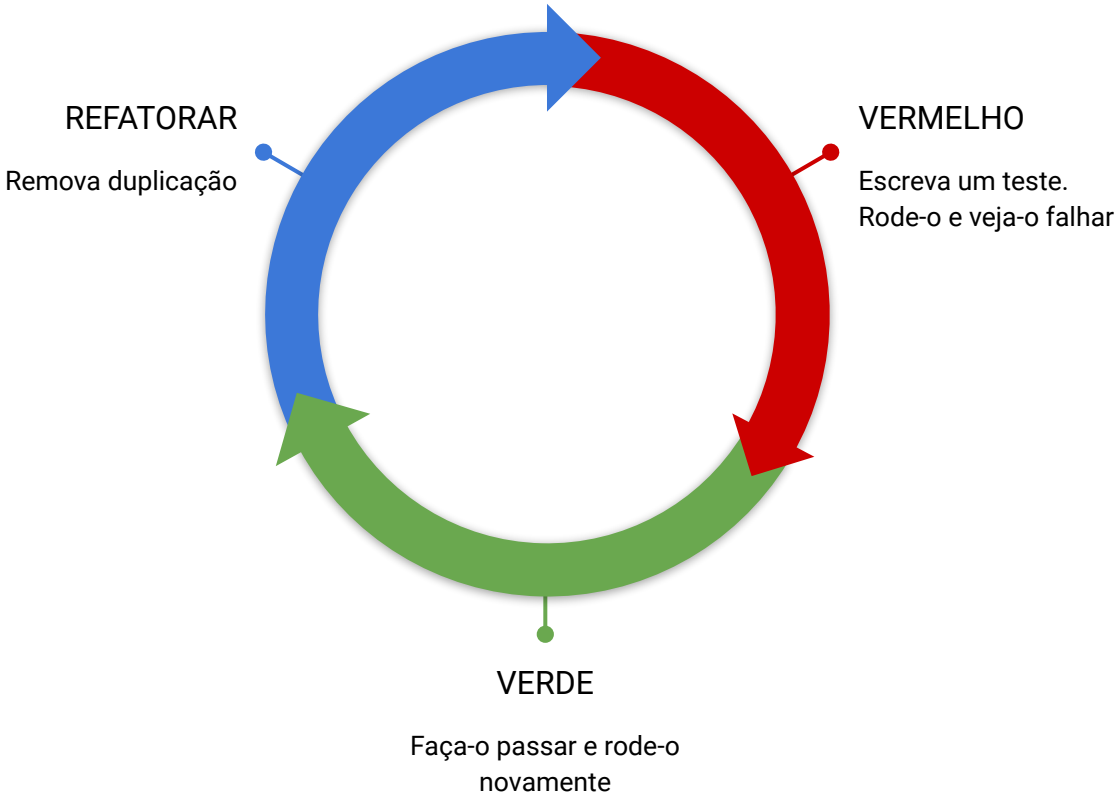
Regras do TDD

- Escrevemos código novo apenas se um teste automatizado falhou
-

Regras do TDD

- Escrevemos código novo apenas se um teste automatizado falhou
- Eliminamos duplicação

Test-Driven Development em Python



Ferramentas para Testes Unitários em Python

unittest — Unit testing framework

Source code: Lib/unittest/__init__.py

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:

test fixture
: A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case
: A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite
: A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that

pytest

About pytest

pytest is a mature full-featured Python testing tool that helps you write better programs.

Table Of Contents

pytest: helps you write better programs

The `pytest` framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

An example of a simple test:

```
# content of test_sample.py
def inc(x):
    return x + 1


def test_answer():
    assert inc(3) == 5
```

To execute it:

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                        [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________
```

standardise testing in Python

Star 984

Searc

quicklinks

home examples
install changelog
config issues[gh]
support plugins/hooks

Table of Contents

Welcome to the tox automation project

Fork me on GitHub

vision: standardize testing in Python

`tox` aims to automate and standardize testing in Python. It is part of a larger vision of easing the packaging, testing and release process of Python software.

What is tox?

tox is a generic virtualenv management and test command line tool you can use for:

- checking your package installs correctly with different Python versions and interpreters
- running your tests in each of the environments, configuring your test tool of choice
- acting as a frontend to Continuous Integration servers, greatly reducing boilerplate and merging CI and shell-based testing.

Basic example

First, install `tox` with `pip install tox`. Then put basic information about your project and the test environments you want your project to run in into a `tox.ini` file residing right next to your `setup.py` file:

v: latest

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py27,py36
```

Coverage.py

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

Coverage measurement is typically used to gauge the effectiveness of tests. It can show which parts of your code are being exercised by tests, and which are not.

The latest version is coverage.py 4.5.3, released March 9, 2019. It is supported on:

- Python versions 2.6, 2.7, 3.3, 3.4, 3.5, 3.6, 3.7, and alpha 3.8.
- PyPy2 6.0 and PyPy3 6.0.
- Jython 2.7.1, though only for running code, not reporting.
- IronPython 2.7.7, though only for running code, not reporting.

Anatomia de um Teste

Anatomia de um teste

A expectativa vem em primeiro lugar

```
assert Money(100 / 2 * (1-0.015), "GBP") == result
```

Anatomia de um teste

```
result = bank.convert(Money(100, "USD"), "GBP")

assert Money(100 / 2 * (1-0.015), "GBP") == result
```

Anatomia de um teste

```
bank = Bank()
bank.add_rate("USD", "GBP", STANDARD_RATE)
bank.commission(STANDARD_COMMISSION)

result = bank.convert(Money(100, "USD"), "GBP")

assert Money(100 / 2 * (1-0.015), "GBP") == result
```

Anatomia de um teste

```
# arrange
bank = Bank()
bank.add_rate("USD", "GBP", STANDARD_RATE)
bank.commission(STANDARD_COMMISSION)
# act
result = bank.convert(Money(100, "USD"), "GBP")
# assert
assert Money(100 / 2 * (1-0.015), "GBP") == result
```

Teste interativo por exemplos

```
def factorial(n):
    """Return the factorial of n, an exact integer >= 0.
    >>> factorial(5)
    120
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0
    """
    if not n >= 0:
        raise ValueError("n must be >= 0")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result
```

Abordagens de TDD em Python

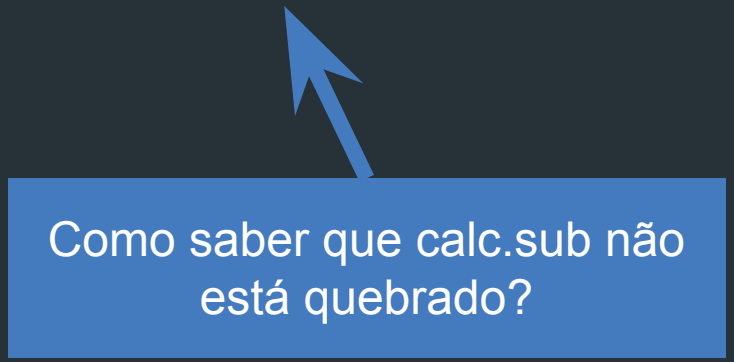
Um Só Passo

RUIM

```
def test_sum():

    assert 4 == calc.add(2, 2)

    assert 0 == calc.sub(2, 2)
```



BOM

```
def test_sum():

    assert 4 == calc.add(2, 2)


def test_sub():

    assert 0 == calc.sub(2, 2)
```

Dados evidentes

RUIM

```
# arrange
bank = Bank()
bank.add_rate("USD", "GBP", STANDARD_RATE)
bank.commission(STANDARD_COMMISSION);
# act
result = bank.convert(Money(100, "USD"), "GBP")
# assert
assert result == Money(49.25, "GBP")
```

BOM

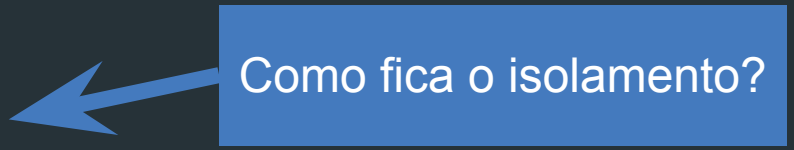
```
# arrange
bank = Bank().
bank.add_rate("USD", "GBP", STANDARD_RATE)
bank.commission(STANDARD_COMMISSION);
# act
result = bank.convert(Money(100, "USD"), "GBP")
# assert
assert result == Money(100 / 2 * (1-0.015), "GBP")
```

Fixture

```python
import pytest


@pytest.fixture
def bank()
    bank = Bank()
    bank.add_rate("USD", "GBP", STANDARD_RATE)
    bank.commission(STANDARD_COMMISSION)


def test_bank_convert(bank)
    result = bank.convert(Money(100, "USD"), "GBP")
    assert result == Money(100 / 2 * (1-0.015), "GBP")
```



Fixture externa

```python
import pytest

@pytest.fixture
def bank()
    bank = Bank.objects.create()
    bank.add_rate("USD", "GBP", STANDARD_RATE)
    bank.commission(STANDARD_COMMISSION)
    yield bank
    bank.delete()

def test_bank_convert(bank)
    result = bank.convert(Money(100, "USD"), "GBP")
    assert Money(100 / 2 * (1-0.015), "GBP") == result
```

Cria o registro no banco de dados

Apaga o registro no banco de dados

Asserções customizadas

```
import pytest

def assert_gbp(expected, actual):
    assert Money(expected, "GBP") == actual

def test_bank_convert(bank)
    bank = Bank()
    bank.add_rate("USD", "GBP", STANDARD_RATE)
    bank.commission(STANDARD_COMMISSION)
    result = bank.convert(Money(100, "USD"), "GBP")
    assert_gbp(100 / 2 * (1-0.015), result)
```

Teste de aprendizado

```
def test_json_response():
    response = requests.get("https://httpbin.org/json")
    data = response.json()
    assert data is not None
    assert "Yours Truly" == data["slideshow"]["author"]
```

Teste de regressão

- Escreva um teste para um bug encontrado
-
-

Teste de regressão

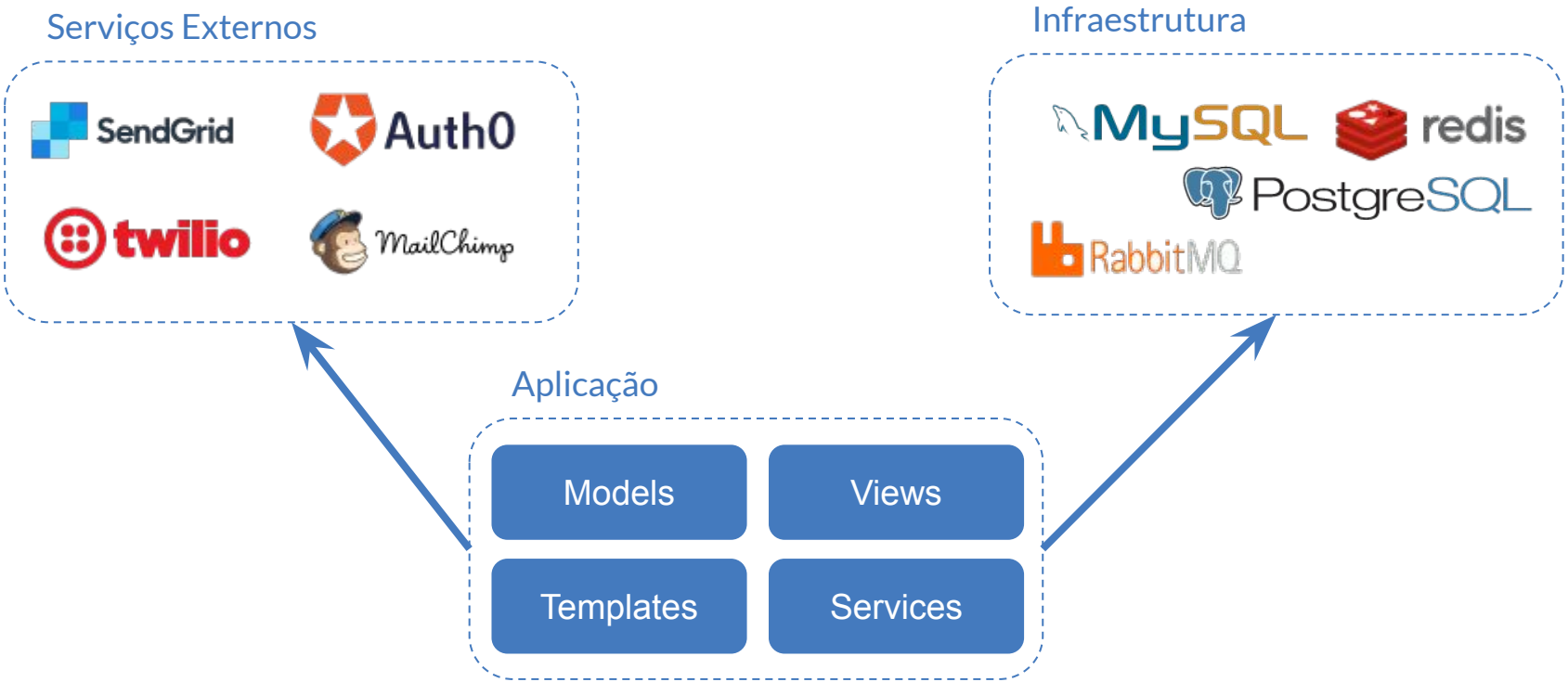
- Escreva um teste para um bug encontrado
- Faça o menor teste possível que reproduza o problema
-

Teste de regressão

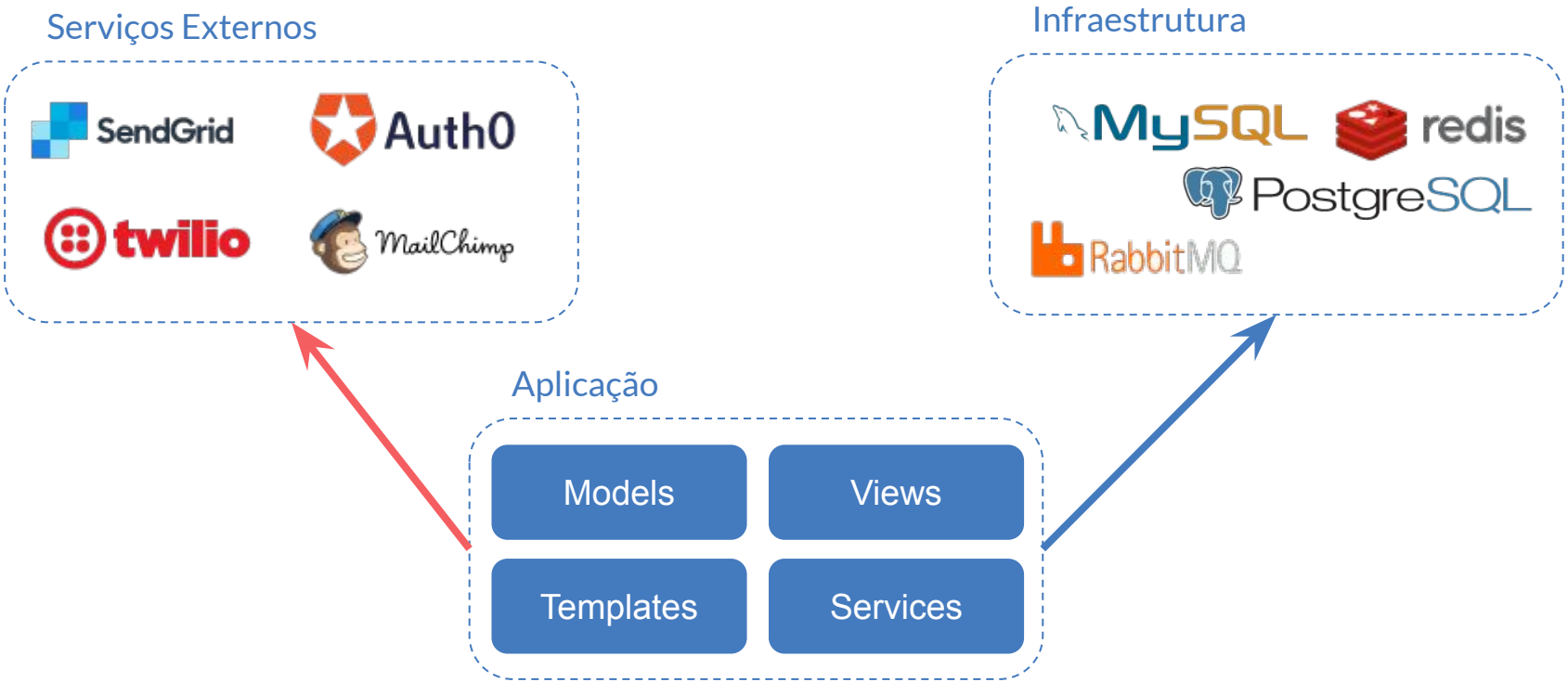
- Escreva um teste para um bug encontrado
- Faça o menor teste possível que reproduza o problema
- Você deveria ter escrito esse teste antes

Mock de Objetos

Simulação de recursos caros



Simulação de recursos caros (Classicista)



Simulação de recursos caros (Mockista)

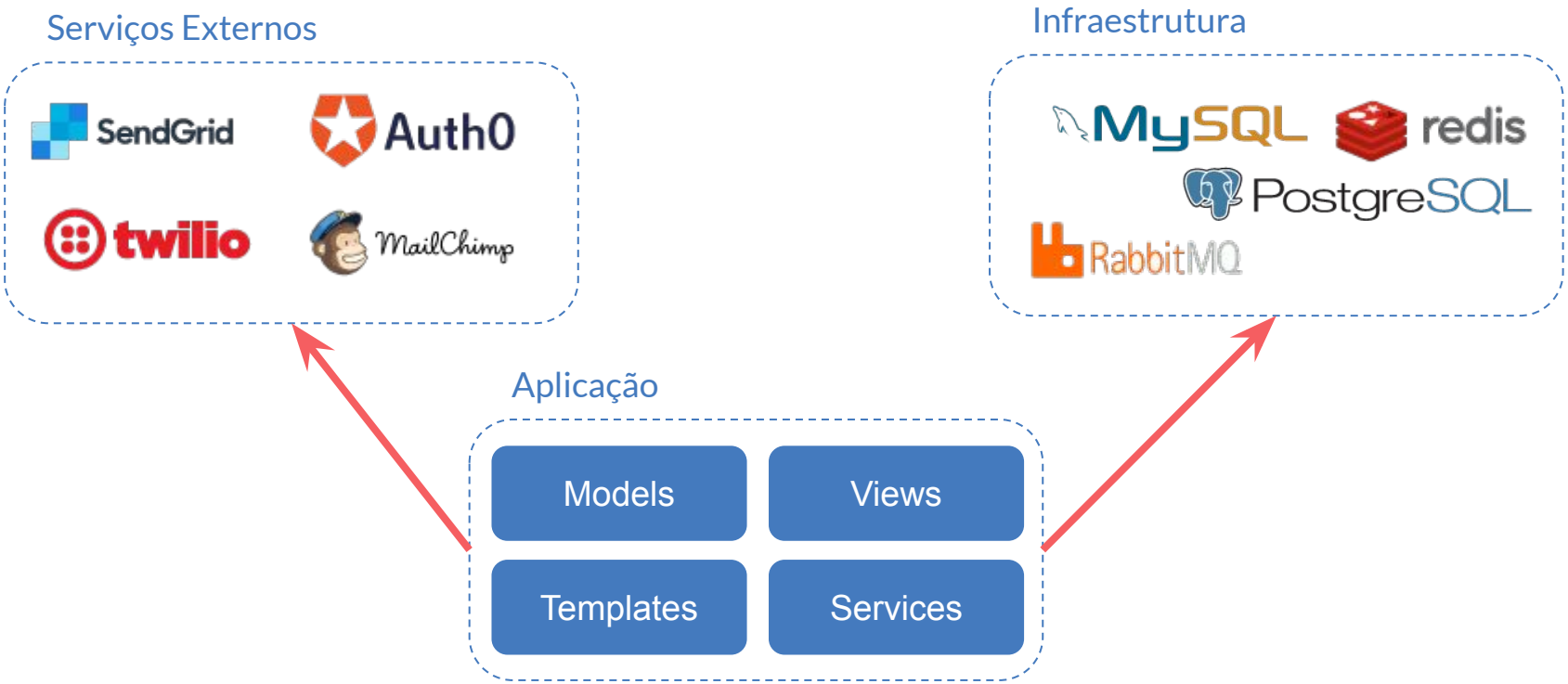


Table of Contents

unittest.mock — mock object library

New in version 3.3.

Source code: Lib/unittest/mock.py

---

unittest.mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

unittest.mock provides a core Mock class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

Additionally, mock provides a patch() decorator that handles patching module and class level attributes within the scope of a test, along with sentinel for creating unique objects. See the quick guide for some examples of how to use Mock, MagicMock and patch().

Mock is very easy to use and is designed for use with unittest. Mock is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

There is a backport of unittest.mock for earlier versions of Python, available as mock on PyPI.

Quick Guide

Mock de Objetos

```
from unittest.mock import MagicMock


def test_form():

    form = Form()
    form.is_valid = MagicMock(return_value=True)

    form.submit()  # True

    form.is_valid.assert_called_once()
```

Modelo de teste de acidentes

```
from unittest.mock import patch
import requests


@patch('requests.get', side_effect=requests.ConnectTimeout)
def test_recursion():
    with pytest.raises(ConnectTimeout) as excinfo:
        requests.get("https://httpbin.org/json")
    assert excinfo is not None
```

Considerações finais

O que devemos testar?

- Condicionais
-
-
-

O que devemos testar?

- Condicionais
- Loops
-
-

O que devemos testar?

- Condicionais
- Loops
- Operações
-

O que devemos testar?

- Condicionais
- Loops
- Operações
- Polimorfismo

Vantagens do TDD

- Você sabe quando está pronto sem ter que se preocupar com uma longa trilha de erros
-
-
-

Vantagens do TDD

- Você sabe quando está pronto sem ter que se preocupar com uma longa trilha de erros
- Você pode escrever códigos melhores
-
-

Vantagens do TDD

- Você sabe quando está pronto sem ter que se preocupar com uma longa trilha de erros
- Você pode escrever códigos melhores
- Melhora a vida dos usuários de seu software
-

Vantagens do TDD

- Você sabe quando está pronto sem ter que se preocupar com uma longa trilha de erros
- Você pode escrever códigos melhores
- Melhora a vida dos usuários de seu software
- Melhora a confiança do time de desenvolvimento

globo.com

Obrigado!

Patrick Porto

Software Engineer | Speaker | Agilist