



Тестирование Web-приложений

Something that is untested is broken.

Виды тестирования



- Unit-тестирование
 - Функциональное тестирование
 - Интеграционное тестирование
 - Нагрузочное тестирование
-
- Client side (selenium)
 - Server side

Виды тестирования. Selenium



```
class SeleniumTests(SeleniumTestCase):
    def test_main(self):
        self.webdriver.find_element_by_xpath('//a[@href="/blog/"]').click()
        elem = self.webdriver.find_element_by_id('blog-create-blog-button')
        assert elem is not None

        elem = self.webdriver.find_element_by_class_name('blog-post-button')
        assert elem is not None

        self.webdriver.find_element_by_link_text('Тренинги').click()

        elem = self.webdriver.find_element_by_link_text('Тренеры')
        assert elem is not None

        # Создаем запись
        self.webdriver.find_element_by_id(u'blog-entry-add').click()
        self.webdriver.find_element_by_name('title').send_keys('Hello')
        self.webdriver.find_element_by_id('blog-post-button').click()
```

XPath (XML Path Language) – язык запросов к элементам XML-документа.

Виды тестирования. TDD



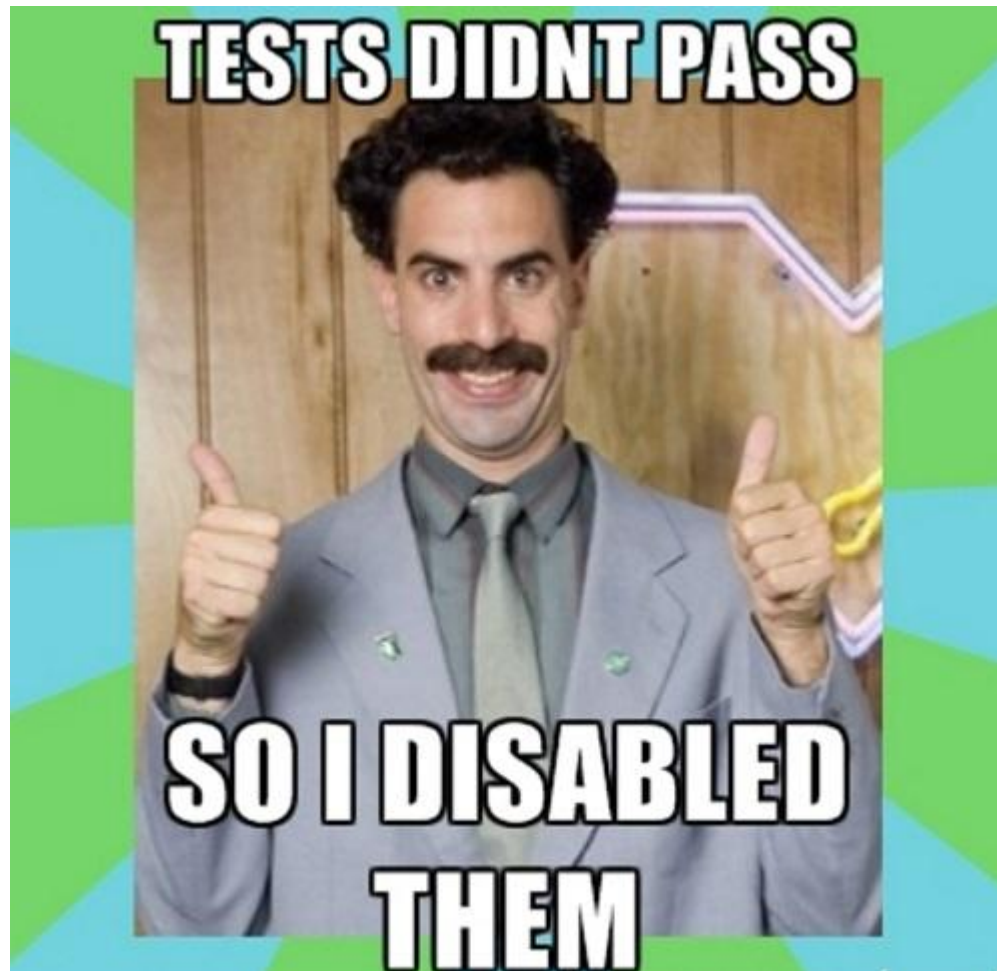
TDD - test driven development - техника разработки ПО, основывается на повторении коротких циклов разработки: пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода

Цели тестирования

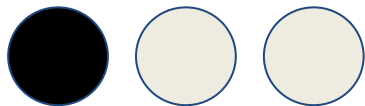


- Проверка правильности реализации
- Проверка обработки внештатных ситуаций и граничных условий
- Минимизация последствий

Цели тестирования



Степень покрытия тестами (test coverage)



Лендинговая страница



Web-приложение с бизнес-логикой



Фреймворки и библиотеки

coverage - библиотека для проверки покрытия тестами
`pip install coverage`

общий случай

```
coverage run tests.py
coverage report -m
coverage html
```

django

```
coverage run --source='.' manage.py test myapp
coverage report
```

Инструменты для тестирования в Python



- doctest
- pytest
- hypothesis
- unittest

doctest



```
'''
Module showing how doctests can be included with source code
Each '>>>' line is run as if in a python shell, and counts as a
test.
The next line, if not '>>>' is the expected output of the previous
line.
If anything doesn't match exactly (including trailing spaces), the
test fails.
'''
```

```
def multiply(a, b):
    """
    >>> multiply(4, 3)
    12
    >>> multiply('a', 3)
    'aaa'
    """
    return a * b
```

Занык

`python -m doctest <file>`

<https://docs.python.org/2/library/doctest.html>

pytest



```
class TestClass(object):
    def test_one(self):
        x = 'this'
        assert 'h' in x

    def test_two(self):
        x = 'hello'
        assert hasattr(x, 'check')
```

Запуск

pytest <file>

<https://docs.pytest.org/en/latest/>

hypothesis



```
from hypothesis import given
from hypothesis.strategies import text

@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

<https://hypothesis.readthedocs.io/en/latest/>

unittest



```
from django.test import Client, TestCase

class SomeTest(TestCase):
    def setUp(self):
        self.client = Client()

    def test_smth(self):
        response_1 = self.client.post('/url1/', {'id': 123})
        response_2 = self.client.get('/url2/')
```

Тестовый http-клиент. Имитирует работу браузера, может отправлять http-запросы

Запрашивает относительный путь, например /login/

Тестирование в Django. unittest



```
import json
from django.test import TestCase, Client

from video.models import Category

class VideoAPITest(TestCase):

    def setUp(self):
        self.client = Client()
        self.category = Category.objects.create(name='test category')
        self.response_map = {
            'name': 'test category',
            'video_count': 0
        }

    def test_category_values(self):
        response = self.client.get('/api/video_category/')
        content = json.loads(response.data)
        self.assertEqual(content['data']['category']['title'], 'test_category')

    def tearDown(self):
        print ('I am done')
```

unittest. Расширенный набор проверок assert



```
assertEqual(a, b)
assertNotEqual(a, b)
assertTrue(x)
assertFalse(x)
assertIs(a, b)
assertIsNot(a, b)
assertIsNone(x)
assertIn(a, b)
assertIsInstance(a, b)
assertLessEqual(a, b)
assertListEqual(a, b)
assertDictEqual(a, b)
assertRaises(exc, fun, *args, **kwargs)
assertJSONEqual(a, b)
```

mock



Mock - подменяет объекты (функции, классы) на так называемые мок-объекты, заглушки.

```
class TestUserSubscription(TestCase):  
    @patch('users.views.get_subscription_status')  
    def test_subscription(self, get_subscription_status_mock):  
        get_subscription_status_mock.return_value = True  
  
        ...
```



Атрибуты объекта Mock с информацией о вызовах

`called` – вызывался ли объект вообще
`call_count` – количество вызовов
`call_args` – аргументы последнего вызова
`call_args_list` – список всех аргументов
`method_calls` – аргументы обращений к вложенным методам и атрибутам
`mock_calls` – то же самое, но в целом и для самого объекта, и для вложенных

пример

```
self.assertEqual(get_subscription_status_mock.call_count, 1)
```




Библиотека `factory_boy` служит для генерации тестовых объектов (в т.ч. связанных) по заданным параметрам.

```
# объявляем фабрику
class RandomUserFactory(factory.Factory):
    class Meta:
        model = models.User

    first_name = factory.Faker('first_name')
    last_name = factory.Faker('last_name')
    email = factory.Sequence(lambda n: 'person{0}@example.com'.format(n))
```

```
# Returns a User instance that's not saved
user = RandomUserFactory.build()

# Returns a saved User instance
user = RandomUserFactory.create()

# Returns a stub object (just a bunch of attributes)
obj = RandomUserFactory.stub()
users = RandomUserFactory.build_batch(10, first_name="Joe")
```

Запуск тестов



Общий случай

```
# найти и выполнить все тесты  
python -m unittest discover
```

```
# тесты нескольких модулей  
python -m unittest test_module1 test_module2
```

```
# тестирование одного кейса - набора тестов  
python -m unittest tests.SomeTestCase
```

```
# тестирование одного метода  
python -m unittest tests.SomeTestCase.test_some_method
```

Django

```
./manage.py test
```

```
./manage.py test app_name
```

```
./manage.py test app_name.tests.TestCaseName
```

```
./manage.py test app_name.tests.TestCaseName.test_function
```

Оптимизация. Тестовое окружение



```
# настроить verbose
python -m unittest --verbose tests

# запустить тесты для быстрого обнаружения ошибок
python -m unittest --failfast tests

# не запускать миграции при каждом запуске тестов (для Django)
./manage.py test --keepdb

# распараллелить запуск на несколько БД (для Django)
./manage.py test --parallel=2
```

Можно использовать легковесную БД при тестировании (если возможно),
переопределив настройки в конфиге

Оптимизация. Тестовое окружение



```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'mydatabaseuser',
        'NAME': 'mydatabase',
        'TEST': {
            'NAME': 'mytestdatabase',
        },
    },
}

# settings.py

TESTING = 'test' in sys.argv or 'jenkins' in sys.argv
```

Фикстуры



```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

Фикстуры



```
# скачать
```

```
django-admin dumpdata app_label > file_name.json
```

```
# загрузить
```

```
django-admin loaddata mydata.json
```

Фикстуры



```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
        # Test definitions as before.
        call_setup_methods()

    def test_fluffy_animals(self):
        # A test that uses the fixtures.
        call_some_test_code()
```



```
from freezegun import freeze_time
from datetime import datetime
from django.test import TestCase

class TestWithDatetime(TestCase):

    def setUp(self):
        self.dt = datetime(
            year=2018, month=8, day=1, hour=12, minute=00
        )

    def test_something(self):
        meeting_id = 1
        data = {'data': 'some_data'}
        with freeze_time(self.dt):
            update_meeting(meeting_id, data)
```


Почувствуй себя интерпретатором



```
@mock.patch("confroom.notification_helpers.send_templated_email")
@mock.patch("confroom.notification_helpers.send_sms")
def test_create(self, send_email_mock, send_sms_mock):
    """
    дано: 2 пользователя, у обоих стоит уведомления по смс,
    у одного стоит уведомлять по email
    """
    self.create_meeting()

    self.assertEqual(send_sms_mock.call_count, 2)
    self.assertEqual(send_email_mock.call_count, 1)
```

Почувствуй себя интерпретатором



```
def test_error1(self):  
    with self.assertRaises(SyntaxError):  
        int(b)  
  
def test_error2(self):  
    with self.assertRaises(TypeError):  
        int('b')  
  
def test_error3(self):  
    with self.assertRaises(KeyError):  
        {}['key']
```

Домашнее задание



- Покрыть тестами все вьюхи и по желанию другие функции
- Написать selenium тест (найти элемент + клик на элемент)
- Использовать mock-объект при тестировании
- Использовать factory boy
- Использовать тестовую базу данных
- Узнать степень покрытия тестами с помощью библиотеки coverage



ТЕХНОТРЕК

**Спасибо за
внимание!**