mockstar Documentation

Release 0.1

Konstantine Rybnikov

January 09, 2017

Contents

Mockstar is a small enhance on top of Mock library that gives you declarative way to write your unit-tests.

- author: Konstantine Rybnikov.
- main repository on bitbucket: https://bitbucket.org/k_bx/mockstar
- mirror on github: https://github.com/k-bx/mockstar

CHAPTER 1

Philosophy

Usually, **unit under test** is something simple, like function or method. It's result is dependent on it's arguments and calls to some external dependencies (side-effects). For example, here:

```
# file sample_app/blog/forms.py

class PostForm(forms.Form):
    title = forms.CharField()
    content = forms.TextField()

    def clean(self):
        if is_post_exist(self.cleaned_data['title']):
            raise ValidationError(_(u"Post with this title already exists"))
        return self.cleaned_data
```

Unit under test is `PostForm` class (more precicely, it's `clean` method here), it has one side-effect, which is `is_post_exist` function.

Usually, you create a single test module for single code module (well, I do). So in this example **module under test** would be `myapp.blog.forms`.

CHAPTER 2

Minimal test-case example

Your minimal test case would look something like this:

```
import unittest
from mockstar import prefixed_p
from sample_app.blog import forms

ppatch = prefixed_p("sample_app.blog.forms")  # module under test


class TestPostForm(unittest.TestCase):
    @ppatch('is_post_exist')      # list / describe side-effects
    def side_effects(self, se):
        se.is_post_exist.return_value = False  # default side-effects behavior
        return self.invoke(se)

    def test_should_be_valid_for_simple_data(self, se):
        form = forms.PostForm({'title': 'foo', 'content': 'bar'})

        self.assertTrue(form.is_valid())

    def test_should_get_error_on_existing_post_title(self, se):
        se.is_post_exist.return_value = True
        form = forms.PostForm({'title': 'foo', 'content': 'bar'})

        self.assertFalse(form.is_valid())
        self.assertEquals(dict(form.errors),
                          ["Post with this title already exists"])
```

CHAPTER 3

More detailed introduction

So, you want to implement and test your unit. Let's say it's a function `create_user()` that will look like this when it is done:

```
# -*- coding: utf-8 -*-

"""app.bl.user"""

from app.bl import mail
from app.tasks.friendship import discover_possible_friends
from app.models import User
from app.utils.security import not_md5_and_has_salt


def create_user(email, password, full_name):
    user = User(email=email,
                password=not_md5_and_has_salt(password),
                full_name=full_name)
    user.save()
    score = count_score(user)
    if score < 10:
        choose_low_quality_avatar(user)
    else:
        choose_high_quality_avatar(user)
        mail.send_welcome_email(user)
        discover_possible_friends(user)
    return user


def count_score(user):
    pass


def choose_low_quality_avatar(user):
    pass


def choose_high_quality_avatar(user):
    pass
```

This unit consists of input-parameters:

- email
- password

- full_name

and seven side-effects:

- `User` model
- `not_md5_and_has_salt` function
- `count_score` function
- `choose_low_quality_avatar` function
- `choose_high_quality_avatar` function
- `mail` business-logic
- `discover_possible_friends` function

So, to test this unit in isolation we would need to mock-out all side-effects, on every test put some return-values so that they will fit our if-else clauses, and finally, generate suitable input-parameters.

With Mock library, you would do something like this:

```
# -*- coding: utf-8 -*-

import unittest
from mock import patch
from app.bl.user import create_user


class TestCreateUser(unittest.TestCase):
    @patch('app.bl.user.choose_low_quality_avatar', autospec=True)
    @patch('app.bl.user.count_score', autospec=True)
    @patch('app.bl.user.not_md5_and_has_salt', autospec=True)
    @patch('app.bl.user.User', autospec=True)
    def test_should_create_save_and_return_user(
        self, user_mock, not_md5_and_has_salt_mock, count_score_mock,
        choose_low_quality_avatar_mock):

        count_score_mock.return_value = 0
        user = user_mock.return_value

        # do
        rv = create_user("foo@bar.com", "pwd", "Foo Bar")
        user_mock.assert_called_with(
            email="foo@bar.com",
            password=not_md5_and_has_salt_mock.return_value,
            full_name="Foo Bar")
        not_md5_and_has_salt_mock.assert_called_with("pwd")
        user.save.assert_called_with()
        self.assertIs(rv, user)

    @patch('app.bl.user.choose_low_quality_avatar', autospec=True)
    @patch('app.bl.user.count_score', autospec=True)
    @patch('app.bl.user.not_md5_and_has_salt', autospec=True)
    @patch('app.bl.user.User', autospec=True)
    def test_should_choose_low_quality_avatar_on_small_score(
        self, user_mock, not_md5_and_has_salt_mock, count_score_mock,
        choose_low_quality_avatar_mock):

        count_score_mock.return_value = 9
        user = user_mock.return_value
```

```
        # do
        create_user("foo@bar.com", "pwd", "Foo Bar")

        count_score_mock.assert_called_with(user)
        choose_low_quality_avatar_mock.assert_called_with(user)

    @patch('app.bl.user.discover_possible_friends', autospec=True)
    @patch('app.bl.user.mail', autospec=True)
    @patch('app.bl.user.choose_high_quality_avatar', autospec=True)
    @patch('app.bl.user.count_score', autospec=True)
    @patch('app.bl.user.not_md5_and_has_salt', autospec=True)
    @patch('app.bl.user.User', autospec=True)
    def test_should_choose_high_quality_avatar_on_big_score(
        self, user_mock, not_md5_and_has_salt_mock, count_score_mock,
        choose_high_quality_avatar_mock, mail_mock,
        discover_possible_friends_mock):

        # ok, I'm bored already
        pass
```

Problems I see:

- need to repeat mocked names as test parameters
- need to write autospec=True again and again
- need to write module prefix `app.bl.user` on every patch call
- need to patch on every test case
- need to add common return_values and assign to some variables (like user) that we'll use later in asserts
- side_effects take a lot of space in our testing code, I want to separate them

With mockstar your test would look something like this:

```
# -*- coding: utf-8 -*-

from mockstar import BaseTestCase
from mockstar import prefixed_p
from app.bl.user import create_user

ppatch = prefixed_p('app.bl.user', autospec=True)


class TestCreateUser(BaseTestCase):
    @ppatch('discover_possible_friends')
    @ppatch('mail')
    @ppatch('choose_high_quality_avatar')
    @ppatch('choose_low_quality_avatar')
    @ppatch('count_score')
    @ppatch('not_md5_and_has_salt')
    @ppatch('User')
    def side_effects(self, se):
        se.user = se.User.return_value
        se.secure_pwd = se.not_md5_and_has_salt.return_value
        se.score = se.count_score.return_value
        return self.invoke(se)

    def test_should_create_save_and_return_user(self, se):
        # do
```

```
        rv = create_user("foo@bar.com", "pwd", "Foo Bar")

        se.User.assert_called_with(
            email="foo@bar.com",
            password=se.secure_pwd,
            full_name="Foo Bar")
        se.not_md5_and_has_salt.assert_called_with("pwd")
        self.assertIs(rv, se.user)

    def test_should_choose_low_quality_avatar_on_small_score(self, se):
        se.count_score.return_value = 9

        # do
        create_user("foo@bar.com", "pwd", "Foo Bar")

        se.count_score.assert_called_with(se.user)
        se.choose_low_quality_avatar.assert_called_with(se.user)

    def test_should_choose_high_quality_avatar_on_big_score(self, se):
        se.count_score.return_value = 11

        # do
        create_user("foo@bar.com", "pwd", "Foo Bar")

        se.choose_high_quality_avatar.assert_called_with(se.user)

    # I'm not so bored now :)
```

I hope you like mockstar's aspiration to get declarative way of writing unit-tests and reduce of copypasta.

CHAPTER 4

Installation

To install mockstar, just type:

```
pip install mockstar
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search