djet Documentation

Release 1.0-dev

Sunscrapers

Apr 02, 2020

Basics

CHAPTER 1

Introduction

Django Extended Tests is a set of helpers for easy testing of Django apps. Main features:

- easy unit testing of Django views (`ViewTestCase`)
- useful assertions provides as mixin classes:
  - response status codes (`StatusCodeAssertionsMixin`)
  - emails (`EmailAssertionsMixin`)
  - messages (`MessagesAssertionsMixin`)
  - model instances (`InstanceAssertionsMixin`)
- handy helpers for testing file-related code (`InMemoryStorageMixin` and others)
- smooth integration with Django REST Framework authentication mechanism (`APIViewTestCase`)

1.1 Installation

Simply install using `pip`:

```
$ pip install djet
```

1.2 Requirements

All of provided versions are validated via testing pipeline to ensure that they are supported:

- **Python**: 2.7, 3.4+
- **Django**: 1.10+
- **Django REST Framework**: 3.7+ (optional)

CHAPTER 2

Getting started

Django test client performs integration tests. All middleware classes, resolvers, decorators and so on are tested. Just a single failure in a middleware can break all the view tests.

One technique of performing the tests was presented at DjangoCon Europe 2013. We, at Sunscrapers have decided to do it in slightly different way, which is why **djet** has been created.

2.1 Testing views

djet makes performing unit tests for your views easier by providing `ViewTestCase`. Instead of `self.client`, you can use `self.factory`, which is an extended `RequestFactory` with overridden shortcuts for creating requests (eg. `path` is not required parameter).

Sometimes you would need middleware to be applied in order to test the view. There is an option that helps specify which middleware should be used in a single test or a whole test case by applying `middleware_classes` argument. This argument should be a list of middleware classes (e.g. `SessionMiddleware`) or tuples where first argument is middleware class and rest items are middleware types (from `MiddlewareType` class). In this case only indicated middleware methods will be call.

2.2 Assertions

djet also provides additional assertions via mixin classes within `djet.assertions` module. They have been created to simplify common testing scenarios and currently there is `StatusCodeAssertionsMixin`, `EmailAssertionsMixin`, `MessagesAssertionsMixin` and `InstanceAssertionsMixin` full of useful assertions.

Remember that if you want to use assertions e.g. from `MessagesAssertionsMixin` you must also add `middleware_classes` required by messages to your test case. We do not add them for you in mixin, because we believe those mixin classes shouldn't implicitly mess with middleware, because it would make it harder to understand what and why exactly is happening in your tests.

2.3 Testing file uploads

There are three primary issues, while testing file-related code in Django and `djet.files` module attempts to solve all of these.

First thing - you won't need any files put somewhere next to fixtures anymore. `create_inmemory_file` and `create_inmemory_image` are ready to use. Those helpful functions are taken from great blog post by Piotr Maliński with just a few small changes.

You can also use `InMemoryStorage` which deals with files being saved to disk during tests and speed ups tests by keeping them in memory.

`InMemoryStorageMixin` does another great thing. It replaces `DEFAULT_FILE_STORAGE` with `InMemoryStorage` for you and also removes all files after test `tearDown`, so you will no longer see any files crossing between tests. You can also provide any storage you want, it should only implement `clear` method which is invoked after tearDown. `InMemoryStorageMixin` cannot be used with bare `unittest.TestCase` - you have to use `TestCase` from Django or `ViewTestCase` from **djet**.

CHAPTER 3

Function Based Views

If you want to test function-based view you should do it like this:

```
from djet import testcases

from fooapp.views import foo_view

class FooViewTest(testcases.ViewTestCase):
    view_function = foo_view

    def test_foo_view_get(self):
        request = self.factory.get()
        # assertions for request

        response = self.view(request)
        # assertions for response
```

CHAPTER 4

Class Based Views

If you want to test class-based view you should do it like this:

```
from djet import testcases

from fooapp.views import foo_view

class FooViewTest(testcases.ViewTestCase):
    view_class = foo_view

    def test_foo_view_get(self):
        request = self.factory.get()
        # assertions for request

        response = self.view(request)
        # assertions for response
```

There is special `create_view_object` helper for testing single view methods, which applies the view_kwargs specified to created view object. You can also provide request, args and kwargs here and they will be bounded to view, like it normally happens in dispatch method.

You can always create view object with different kwargs by using `self.view_class` constructor.

```
class YourViewObjectMethodTest(testcases.ViewTestCase):
    view_class = YourView
    view_kwargs = {'redirect_url': '/'}

    def test_some_view_method(self):
        request = self.factory.get()
        view_object = self.create_view_object(request, 'some arg', pk=1)

        view_object.some_method()

        self.assertTrue(view_object.some_method_called)
```

CHAPTER 5

Assertions

We encourage you to import whole djet modules, not classes.

```
from djet import assertions, testcases
from django.contrib import messages
from django.contrib.messages.middleware import MessageMiddleware
from django.contrib.sessions.middleware import SessionMiddleware
from yourapp.views import YourView
from yourapp.factories import UserFactory

class YourViewTest(assertions.StatusCodeAssertionsMixin,
                   assertions.MessagesAssertionsMixin,
                   testcases.ViewTestCase):
    view_class = YourView
    view_kwargs = {'some_kwarg': 'value'}
    middleware_classes = [
        SessionMiddleware,
        (MessageMiddleware, testcases.MiddlewareType.PROCESS_REQUEST),
    ]

    def test_post_should_redirect_and_add_message_when_next_parameter(self):
        request = self.factory.post(data={'next': '/'}, user=UserFactory())

        response = self.view(request)

        self.assert_redirect(response, '/')
        self.assert_message_exists(request, messages.SUCCESS, 'Success!')
```

You can also make assertions about the lifetime of model instances. The `assert_instance_created` and `assert_instance_deleted` methods of `InstanceAssertionsMixin` can be used as context managers. They ensure that the code inside the `with` statement resulted in either creating or deleting a model instance.

```
from django.test import TestCase
from djet import assertions
from yourapp.models import YourModel
```

(continues on next page)

(continued from previous page)

```
class YourModelTest(assertions.InstanceAssertionsMixin, TestCase):

    def test_model_instance_is_created(self):
        with self.assert_instance_created(YourModel, field='value'):
            YourModel.objects.create(field='value')
```

CHAPTER 6

Files

An example of test using all files goodies from **djet**:

```
from djet import files
from django.core.files.storage import default_storage
from django.test.testcases import TestCase

class YourFilesTests(files.InMemoryStorageMixin, TestCase):

    def test_creating_file(self):
        created_file = files.create_inmemory_file('file.txt', 'Avada Kedavra')

        default_storage.save('file.txt', created_file)

        self.assertTrue(default_storage.exists('file.txt'))
```

CHAPTER 7

Integration with DRF

Below there is an example of Django REST Framework authentication mocking. Pay attention to `djet.restframework.APIViewTestCase` base class and `user` parameter in request factory call.

```
from django.contrib.auth import get_user_model
from djet import assertions, utils, restframework
import views

class SetUsernameViewTest(restframework.APIViewTestCase,
                          assertions.StatusCodeAssertionsMixin):
    view_class = views.SetUsernameView

    def test_post_should_set_new_username(self):
        password = 'secret'
        user = get_user_model().objects.create_user(username='john',␣
↪password=password)
        data = {
            'new_username': 'ringo',
            'current_password': password,
        }
        request = self.factory.post(user=user, data=data)

        response = self.view(request)

        self.assert_status_equal(response, status.HTTP_200_OK)
        user.refresh_from_db()
        self.assertEqual(data['new_username'], user.username)
```

For more comprehensive examples we recommend to check out how djoser library tests are crafted.

CHAPTER 8

Indices and tables

- genindex
- search