drf-tester Documentation

Release latest

Nov 05, 2021

CONTENTS

`drf-tester` is a Python module that aims to help developers with testing *DjangoRestFramework* API endpoints.

- Minimize the time (and lines of code) required
- Mantain consistent testing coverage
- Increase productivity!

The philosophy behind the design of this module, is that a developer should only need to:

1. Prepare the `setUp` method of a Test class
2. Choose the correct classes to inheritb for each type of access
3. Smile!!

This saves us developers lots of time writing repetitive, boiler-plate code, while reassuring that our tests are, at the very least, consistent.

CHAPTER

ONE

REQUIREMENTS

The module has been tested to work with the following software versions.

- Python 3.7
- Django 2.2
- DjagnoRESTFramework 3.12.2
- factory-boy 3.1.0

Compatibility likely greater than indicated here (let me know if something else works for you)

CHAPTER TWO

INSTALLATION

To install *drf-tester* in your systems, use pip:

```
pip install drf-tester
```

CHAPTER THREE

TABLE OF CONTENTS

3.1 BaseDrfTest

Located in `drf_tester.utils` this class is the glue that makes it all work.

It contains the `setUp` method to be run before every test, as well as some helper functions and Object variables.

Check out the code for more details.

3.1.1 Built-in Functions

Some generic methods are created for DRYer single-action test classes.

- `get_admin_user(self, data: dict) -> User`
- `get_active_user(self, data: dict) -> User`
- `get_active_staff(self, data: dict) -> User`
- `get_model_instances(self) -> list`

3.1.2 Object Variables

Some Object-level variables are declared outside of `setUp` for convenience.

- `EXACT_AMOUNT`: if is set to an integer value, `get_model_instances` will return a list of instances of exactly that size.
- `MIN` and `MAX`: used as limits when using `random.randint` to create a list of instances of random size. Default: `5` and `10`.

3.1.3 setUp()

The variables required for correct operation:

```
self.endpoint = None    # string with the url of the endpoint
self.factory = None     # factory-boy class to create model instances
self.model = None       # the model accessed through the endpoint
self.instance_data = {}     # dict of valid SERIALIZED data for instance creation
self.view = viewsets.YourViewSet.as_view({"get": "list", "post": "create", "put": "update
↪", "delete": "destroy"})
self.user_data = {}     # Required for authenticated user testing
```

(continues on next page)

(continued from previous page)

```
self.admin_data = {}    # Required for super user testing
self.staff_data = {}    # Required for staff user testing
self.USER_FIELD_NAME = 'creator'    # Required for testing user object access
```

3.1.4 Access Level

Once you know what level of access each kind of user should have, just add those classes to your tests, after `APITestCase`.

Example:

```
from drf_tester.viewsets import anon, admin, auth

class YourViewSetTest(APITestCase, anon.AnonNoAccess, auth.AuthFullAccess, admin.
↪AdminFullAccess):
    """To test a ModelViewSet with IsAuthenticated as the only Permission
    """

    def setUp(self):
        ...
```

3.2 Viewset Tests

All the classes to assist with the development of tests for `ViewSets` are located under `drf_tester.viewsets`, and separated by user type:

- `anon.py`: Anonymous users
- `auth.py`: Authenticated users
- `admin.py`: Admin user (superusers)
- `staff.py`: Staff users

Within each file, there are classes that test the effects of different actions on the endpoint. For example:

```
# drf_tester/viewsets/anon.py

class NoCreate(BaseDrfTest):
    def test_anon_user_cannot_create_instance(self):
        """Anonymous user cannot create new instance"""
        ...
```

These single-action classes are grouped in bigger classes meant to be inherited by the final test cases. For example:

```
class AnonReadOnly(CanList, CanRetrieve, NoCreate, NoUpdate, NoDestroy):
    """
    Anonymous user has only read access to endopint
    """

    pass
```

A few different combinations are provided for your convenience:

- **For anonymous users:**
  - AnonNoAccess
  - AnonReadOnly
  - AnonFullAccess
- **For authenticated users:**
  - AuthFullAccess
  - AuthNoAccess
  - AuthReadOnly
  - AuthOwner: only controls instances linked to user
- **For admin users:**
  - AdminNoAccess
  - AdminReadOnly
  - AdminFullAccess
- **For staff users:**
  - StaffNoAccess
  - StaffReadOnly
  - StaffFullAccess

Custom groups can be made mixing and matching classes according with the level of access expected by each user-type from each endpoint.

3.3 Example

Included in the repository, there's an example illustrating how to implement in your project.

From example_one:

```
class ThingViewSetTest(APITestCase, AnonNoAccess, AuthFullAccess, AdminFullAccess):
    """
    Thing viewset tests
    Permission level: IsAuthenticated
    """

    def setUp(self):
        """Tests setup"""
        self.endpoint = "/api/v1/things/"
        self.factory = factories.ThingFactory
        self.model = models.Thing
        self.view = views.ThingViewSet.as_view({"get": "list", "post": "create", "put":
↪"update", "delete": "destroy"})
        self.instance_data = {...}
        self.user_data = {...}
        self.admin_data = {...}
```

For more details, checkout the project under example/example_one/