RedRover Documentation

Release 0.7.2

Dustin Farris

January 15, 2013

CONTENTS

RedRover is a behavior-driven testing utility suite for Django. It wraps other powerful tools such as Nose and Selenium into a clean, readable syntax that is easy to use.

CHAPTER

ONE

USER'S GUIDE

1.1 Quickstart

Basic requirements to run RedRover:

- Python 2.7 (Working on 3.x but not quite there)
- Django 1.4+ (tested with all micro-releases of 1.4)

1.1.1 Dependencies

These will be automatically installed for you:

- django_nose
- rednose
- splinter

It is also highly recommended that you install and use FactoryBoy. While not strictly required to run RedRover, it is a very nice compliment to RedRover's operation and is used in many of the examples.

1.1.2 Install RedRover

Installing is easy via pip:

```
pip install redrover
```

1.1.3 Configuration

Add `'redrover'` to the bottom of your `'INSTALLED_APPS'`. Also add the following to your settings.py:

```
TEST_RUNNER = 'redrover.RedRoverRunner'
```

1.1.4 Usage

Run your tests the same as usual. Django will use the custom test- runner you specified in settings.py which gives RedRover the reigns.:

```
python manage.py test
```

1.1.5 Test Discovery

RedRover will discover your tests be recursively detecting Python modules and then looking for any sub-module(file) ending in "_test.py". Because of this, it is recommended that you organize your tests in a "tests" directory in your root project directory.

A sample Django project might look like this:

```
myproject
  |
  |-- manage.py
  |
  |-- myapp
  |     |
  |     |-- __init__.py
  |     |-- models.py
  |     |-- views.py
  |     +-- urls.py
  |
  |-- myproject
  |     |
  |     |-- __init__.py
  |     |-- settings.py
  |     |-- urls.py
  |     +-- wsgi.py
  |
  +-- tests
        |
        |-- __init__.py
        |-- factories.py
        |
        |-- models
        |     |
        |     |-- __init__.py
        |     +-- myapp_mymodel_test.py
        |
        +-- requests
              |
              |-- __init__.py
              +-- myapp_pages_test.py
```

Within your 'test' Python files, you should import RedRover and subclass the RedRoverTest or RedRoverLiveTest classes like this:

```
# myapp_my_model_test.py

from redrover import *

from myapp.models import MyModel


class MyModelTest(RedRoverTest):
```

When you use the RedRoverTest or RedRoverLiveTest superclass you are provided the "describe" decorator. When you decorate your individual test cases with this decorator, they will be discovered by the test runner regardless of whether the characters "test" appear in the method name:

```
class MyModelTest(RedRoverTest):

  @describe
  def addition(self):
    assert_equal(2, 1 + 1)
```

CHAPTER

TWO

WRITING TESTS

2.1 Writing Tests

2.1.1 Introduction

The whole point of RedRover is to make writing Django tests easy and fun. To that end, much of the traditional Python unittest syntax has been bent or broken; but, RedRover does ultimately inherit from the unittest library — if it works in unittest, it will work in RedRover.

Remember that tests are divided into 3 main categories:

Unit Tests Tests that check the a very specific piece of code is behaving as it should.

These haven't changed much. You can use the a few of the RedRover goodies to make them look a bit nicer, but the concept is pretty much the same.

Integration Tests Tests verifying that a given input to the program/website produces the expected output.

This is where RedRover really steps up. Wrapping powerful Selenium functionality into seemless syntax, you'll be able to test your Django pages, forms, and APIs with ease.

Functional Tests Tests that watch the program/website in action and report any odd behavior.

For a Django website, these would consist of server pings, automated browsing, and database querying. These are usually not part of the Django project and are set up on a server or some service provider.

We'll focus on Unit Tests and Integration Tests since those are the two you will most likely be including in your project code. When writing tests for a Django project, we'll divide them into three areas:

- Unit Tests
- Model Tests
- Request Tests

Unit Tests will, again, test specific pieces of your Django project. Model Tests, which could also be consider unit tests, will test that your individual models are set up properly and behave as expected. Finally, Request Tests will check that you Django project serves the right pages and that they contain the right content as you navigate, fill out forms, etc. These are your integration tests.

In your root project directory, you should a tests directory to contain these. It would look something like this:

```
tests
  |
  |--  __init__.py
  |
  |--  factories.py
  |
  |
  |--  unit
  |      |
  |      |--  __init__.py
  |      |
  |      |--  some_method_test.py
  |
  |
  |--  models
  |      |
  |      |--  __init__.py
  |      |
  |      |--  someapp_model_test.py
  |
  |
  |--  requests
         |
         |--  __init__.py
         |
         |--  someapp_pages_test.py
```

You may be wondering about the "factories.py" file. That will hold FactoryBoy factories which are an awesome replacement for fixtures. Use it.

2.1.2 Subjects and Assertions

The general syntax of RedRover works like this.

1. Declare a subject
2. Define the subject
3. Refer to the subject using various assertions

You declare a subject by assigning it as a string to the 'subject' class variable. You define it in the test-case's 'setUp' method. You refer to it in your various tests within the test-case.

The Subject

Here is an example of us declaring the subject of a test case to be "mynumber" and defining it in the setUp method:

```
class SomeTest(RedRoverTest):

  subject = 'mynumber'

  def setUp(self):
    self.mynumber = 4
```

The Assertions

In the individual tests you will refer to the subject as 'it'. "It" has two primary tests: "should" and "should_not". Within these tests we pass an assertion that 'should' or 'should not' pass, and any additional arguments that assertion might need.

The syntax looks like this:

it.should|should_not(*assertion_name*, [*arg1, arg2,* ...])

To enable this syntax we use the "describe" decorator. An example test would look like this:

```
@describe
def adding_up_to_mynumber(self):
  it.should(equal, 2 + 2)
```

In this case, the assertion is "equal" and it *should* pass. Behind the scenes "equal" compares self.mynumber — which in this case is 4 — and 2 + 2. The do in fact equal, they "should" equal according to our test, so this test passes.

You can also refer to attributes of the subject. Suppose we have some object:

```
class MyObject(object):

  def __init__(self, name)
    self.name = name

  @property
  def greeting(self):
    return "My name is %s" % self.name
```

We could test the behavior of this object with a test-case that looks like this:

```
class MyObjectTest(RedRoverTest):

  subject = 'myobject'

  def setUp(self):
    self.myobject = MyObject("bob")

  @describe
  def myobjects_name(self):
    its('name').should(be, "bob")
```

You can also test properties:

```
@describe
def my_objects_greeting(self):
  its('greeting').should(be, "My name is bob")
```

2.1.3 Behavior-driven Testing

The notion of "behavior-driven" testing is that you first define how something behaves, and then you write the code to make it happen. To define how something behaves you must first define what that something *is*. That "something" becomes the name of our test-case class.

Model Tests

Suppose we have a Django app called "people" and we want to define the behavior of a model within this app called "Person". Our test would go in a file named something like `tests/models/people_person_model_test.py` and would be called something like PersonModel-Test here:

```
from redrover import *

class PersonModelTest(RedRoverTest):
```

The next step is to define what aspect of the Person model we care about. In this case the answer is straight-forward, we care about an instantiated version of the Person model. That becomes the *subject* of our test case.

To define the "subject" of a test case, first declare it using a string, then define it in the setUp method of the test case:

```
from redrover import *

from tests.factories import *


class PersonModelTest(RedRoverTest):

  subject = 'person'

  def setUp(self):
    self.person = PersonFactory.build()
```

Now you can use RedRover assertions to define the behavior this subject should have. A commonly expected behavior of a Django class would be that it responds to certain attributes and methods.:

```
class PersonModelTest(RedRoverTest):

  subject = 'person'

  def setUp(self):
    self.person = PersonFactory.build()

  @describe
  def attributes(self):
    it.should(respond_to, 'first_name')
    it.should(respond_to, 'last_name')
    it.should(respond_to, 'full_name')
    it.should(respond_to, 'age')
    it.should(respond_to, 'gender')

    it.should(be_valid)
```

We would further expect that if certain fields were blank or otherwise invalid, the model would not validate:

```
@describe
def when_first_name_is_not_present(self):
  self.person.first_name = ""
  it.should_not(be_valid)

@describe
def when_last_name_is_not_present(self):
  self.person.last_name = ""
  it.should_not(be_valid)

@describe
def when_age_is_not_present(self):
  self.person.age = None
  it.should_not(be_valid)

@describe
```

```
def when_gender_is_not_present(self):
  self.person.gender = ""
  it.should_not(be_valid)

@describe
def when_gender_is_invalid(self):
  self.person.gender = "X"
  it.should_not(be_valid)
```

Also, since full_name will probably be a property that combines first_name and last_name, we'll want to check that it behaves as it should.:

```
@describe
def full_name(self):
  self.person.first_name = "Charles"
  self.person.last_name = "Dickens"
  its('full_name').should(equal, "Charles Dickens")
```

If you were to run `manage.py test` with this test in place, it would probably give a bunch of (nice-looking) errors since the Person model doesn't even exist yet.

To get these tests to pass, we would create a model that looks something like this:

```
class Person(models.Model):
  GENDER_CHOICES = [
    ('M', 'Male'),
    ('F', 'Female')]
  first_name = models.CharField(max_length=20)
  last_name = models.CharField(max_length=20)
  age = models.IntegerField()
  gender = models.CharField(max_length=1, choices=GENDER_CHOICES)

  @property
  def full_name(self):
    return '%s %s' % (self.first_name, self.last_name)
```

Request Tests

CHAPTER

THREE

REFERENCE

CHAPTER

FOUR

RESOURCES

- Bug Tracker
- Code