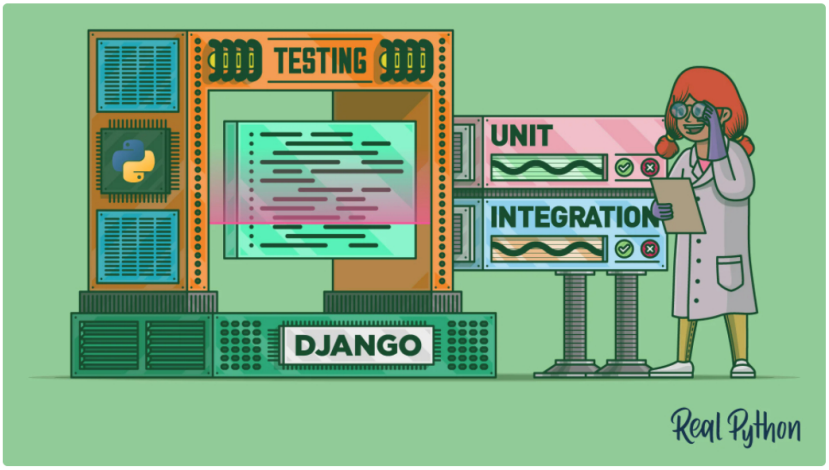


Testing in Django (Part 1) – Best Practices and Examples

by Real Python Aug 06, 2013 17 Comments basics best-practices django testing web-dev

Mark as Completed

Tweet Share Email

Table of Contents

Get more work done. PyCharm. Start free trial JET BRAINS

Remove ads

Testing is vital. Without properly testing your code, you will never know if the code works as it should, now or in the future when the codebase changes. Countless hours can be lost fixing problems caused by changes to the codebase. What's worse, you may not even know that there are problems at all until your end users complain about it, which is obviously not how you want to find out about code breaks.

Having tests in place will *help* ensure that if a specific function breaks you will know about it. Tests also make debugging breaks in code much easier, which saves time and money.

I've literally lost gigs in the past from not properly testing new features against the old codebase. Do not let this happen to you. Take testing seriously. You will have more confidence in your code, and your employer will have more confidence in you. It's essentially an insurance policy.

Testing *helps* you structure good code, find bugs, and write documentation.

In this post, we'll be first looking at a brief introduction that includes best practices before

— FREE Email Series —

Python Tricks

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

No spam. Unsubscribe any time.

Get more work done. PyCharm. Start free trial JET BRAINS PC

Table of Contents

- Intro to Testing in Django
- Examples
- Next Time

Mark as Completed

Tweet Share Email

In this post, we'll be first looking at a brief introduction that includes best practices before looking at a few examples.

Free Bonus: Click here to get access to a free Django Learning Resources Guide (PDF) that shows you tips and tricks as well as common pitfalls to avoid when building Python + Django web applications.

Intro to Testing in Django

Remove ads

Types of tests

Unit and integration are the two main types of tests:

- *Unit Tests* are isolated tests that test one specific function.
- *Integration Tests*, meanwhile, are larger tests that focus on user behavior and testing entire applications. Put another way, integration testing combines different pieces of code functionality to make sure they behave correctly.

Focus on unit tests. Write A LOT of these. These tests are much easier to write and debug vs. integration tests, and the more you have, the less integration tests you will need. Unit tests should be fast. We will look at a few techniques for speeding up tests.

That said, integration tests are sometimes still necessary even if you have coverage with unit tests, since integration tests can help catch code regressions.

In general, tests result in either a Success (expected results), Failure (unexpected results), or an error. You not only need to test for expected results, but also how well your code handles unexpected results.

Best practices

1. If it can break, it should be tested. This includes models, views, forms, templates, validators, and so forth.
2. Each test should generally only test one function.
3. Keep it simple. You do not want to have to write tests on top of other tests.
4. Run tests whenever code is PULLed or PUSHed from the repo and in the staging environment before PUSHing to production.
5. When upgrading to a newer version of Django:
    - upgrade locally,
    - run your test suite,
    - fix bugs,
    - PUSH to the repo and staging, and then
    - test again in staging before shipping the code.

Structure

Structure your tests to fit your Project. I tend to favor putting all tests for each app in the *tests.py* file and grouping tests by what I'm testing - e.g., models, views, forms, etc.

You can also bypass (delete) the *tests.py* file altogether and structure your tests in this manner within each app:

```
└── app_name
    └── tests
        ├── __init__.py
        ├── test_forms.py
        ├── test_models.py
        └── test_views.py
```

Finally, you could create a separate test folder, which mirrors the entire Project structure, placing a *tests.py* file in each app folder.

Larger projects should use one of the latter structures. If you know your smaller project will eventually scale into something much larger, it's best to use one of the two latter structures as well. I favor the first and the third structures, since I find it easier to design tests for each app when they are all viewable in one script.

Third-party packages

Use the following packages and libraries to assist with writing and running your test suite:

- django-webtest: makes it much easier to write functional tests and assertions that match the end user's experience. Couple these tests with Selenium tests for full coverage on templates and views.
- coverage: is used for measuring the effectiveness of tests, showing the percentage of your codebase covered by tests. If you are just starting to set up unit tests, coverage can help offer suggestions on what should be tested. Coverage can also be used to turn testing into a game: I try to increase the percent of code covered by tests each day, for example.
- django-discover-runner: helps locate tests if you organize them in a different way (e.g, outside of *tests.py*). So if you organize your tests into separate folders, like in the example above, you can use discover-runner to locate the tests.
- factory_boy, model_mommy, and mock: all are used in place of fixtures or the ORM for populating needed data for testing. Both fixtures and the ORM can be slow and need to be updated whenever your model changes.

Examples

In this basic example, we will be testing:

- models,
- views,
- forms, and
- the API.

Download the Github repo here to follow along.

Remove ads

Setup

Install coverage and add it to your INSTALLED_APPS:

Shell

```
$ pip install coverage==3.6
```

Run coverage:

Shell

```
$ coverage run manage.py test whatever -v 2
```

> Use verbosity level 2, `-v 2`, for more detail. You can also test your entire Django Project at once with this command: `coverage run manage.py test -v 2`.

Build your report to see where testing should begin:

Shell

```
$ coverage html
```

Open *django15/htmlcov/index.html* to see the results of your report. Scroll to the bottom of the report. You can skip all rows from the virtualenv folder. Never test anything that is a built-in Python function or library since those are already tested. You can move your virtualenv out of the folder to make the report cleaner after it's ran.

Let's start by testing the models.

Testing Models

Within the coverage report, click the link for "whatever/models". You should see this screen:

Essentially, this report is indicating that we should test the title of an entry. Simple.

Open *tests.py* and add the following code:

Python

```python
from django.test import TestCase
from whatever.models import Whatever
from django.utils import timezone
from django.core.urlresolvers import reverse
from whatever.forms import WhateverForm

# models test
class WhateverTest(TestCase):

    def create_whatever(self, title="only a test", body="yes, this is only a test"):
        return Whatever.objects.create(title=title, body=body, created_at=timezone.nou

    def test_whatever_creation(self):
        w = self.create_whatever()
        self.assertTrue(isinstance(w, Whatever))
        self.assertEqual(w.__unicode__(), w.title)
```

What's going on here? We essentially created a `Whatever` object and tested whether the created title matched the expected title - which it did.

> Note: Make sure your function names start with `test_`, which is not only a common convention but also so that django-discover-runner can locate the test. Also, write tests for *all* of the methods you add to your model.

Re-run coverage:

Shell

```
$ coverage run manage.py test whatever -v 2
```

You should see the following results, indicating the test passed:

Shell

```
test_whatever_creation (whatever.tests.WhateverTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.002s
```

```
OK
```

Then if you look at the coverage report again, the models should now be at 100%.

sonarqube
Your Python code: Powerful and Secure
Find Vulnerabilities and Security Hotspots early & fix them fast!
Discover Now
Remove ads

Testing Views

Testing views can sometimes be difficult. I generally use unit tests to check status codes as well as Selenium Webdriver for testing AJAX, Javascript, etc.

Add the following code to the WhateverTest class in *tests.py*:

Python

```python
# views (uses reverse)

    def test_whatever_list_view(self):
        w = self.create_whatever()
        url = reverse("whatever.views.whatever")
        resp = self.client.get(url)

        self.assertEqual(resp.status_code, 200)
        self.assertIn(w.title, resp.content)
```

Here we fetch the URL from the client, store the results in the variable `resp` and then test our assertions. First, we test whether the response code is 200, and then we test the actual response back. You should get the following results:

Shell

```
test_whatever_creation (whatever.tests.WhateverTest) ... ok
test_whatever_list_view (whatever.tests.WhateverTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.052s

OK
```

Run your report again. You should now see a link for “whatever/views”, showing the following results:

You can also write tests to make sure something fails. For example, if a user needs to be logged in to create a new object, the test would succeed if it actually fails to create the object.

Let's look at a quick selenium test:

Python

```python
# views (uses selenium)

import unittest
from selenium import webdriver

class TestSignup(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def test_signup_fire(self):
        self.driver.get("http://localhost:8000/add/")
        self.driver.find_element_by_id('id_title').send_keys("test title")
        self.driver.find_element_by_id('id_body').send_keys("test body")
        self.driver.find_element_by_id('submit').click()
        self.assertIn("http://localhost:8000/", self.driver.current_url)

    def tearDown(self):
        self.driver.quit

if __name__ == '__main__':
    unittest.main()
```

Install selenium:

Shell

```
$ pip install selenium==2.33.0
```

Run the tests. Firefox should load (if you have it installed) and run the test. We then assert that the correct page is loaded upon submission. You could also check to ensure that the new object was added to the database.

Testing Forms

Add the following methods:

Python

```python
def test_valid_form(self):
    w = Whatever.objects.create(title='Foo', body='Bar')
    data = {'title': w.title, 'body': w.body,}
    form = WhateverForm(data=data)
    self.assertTrue(form.is_valid())

def test_invalid_form(self):
    w = Whatever.objects.create(title='Foo', body='')
    data = {'title': w.title, 'body': w.body,}
    form = WhateverForm(data=data)
    self.assertFalse(form.is_valid())
```

Notice how we are generating the data for the form from JSON. This is a fixture.

You should now have 5 passing tests:

Shell

```
test_signup_fire (whatever.tests.TestSignup) ... ok
test_invalid_form (whatever.tests.WhateverTest) ... ok
test_valid_form (whatever.tests.WhateverTest) ... ok
test_whatever_creation (whatever.tests.WhateverTest) ... ok
test_whatever_list_view (whatever.tests.WhateverTest) ... ok

----------------------------------------------------------------------
Ran 5 tests in 12.753s
```

```
OK
```

You could also write tests that assert whether a certain error message is displayed based on the validators in the form itself.

Remove ads

Testing the API

First, you can access the API from this URL: http://localhost:8000/api/whatever/?format=json. This is a simple setup, so the tests will be fairly simple as well.

Install lxml and defused XML:

Shell

```
$ pip install lxml==3.2.3
$ pip install defusedxml==0.4.1
```

Add the following test cases:

Python

```
from tastypie.test import ResourceTestCase

class EntryResourceTest(ResourceTestCase):

    def test_get_api_json(self):
        resp = self.api_client.get('/api/whatever/', format='json')
        self.assertValidJSONResponse(resp)

    def test_get_api_xml(self):
        resp = self.api_client.get('/api/whatever/', format='xml')
        self.assertValidXMLResponse(resp)
```

We are simply asserting that we get a response in each case.

Next Time

In the next tutorial, we'll be looking at a more complicated example as well as using model_mommy for generating test data. Again, you can grab the code from the repo.

Free Bonus: Click here to get access to a free Django Learning Resources Guide (PDF) that shows you tips and tricks as well as common pitfalls to avoid when building Python + Django web applications.

Got something to add? Leave a comment below.

Mark as Completed

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

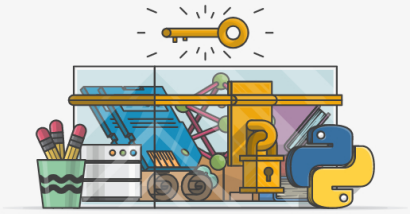
About The Team

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Michael

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: basics best-practices django testing web-dev

Join the discussion…

Pedro Gabriel Lancelloti Pinto • 4 years ago • edited

Nvm just checked the github repo

Help