Django Standalone Apps

A developer's fieldguide to writing reusable apps, including testing, model migrations, backwards (and forwards) compatibility, and best practices for extracting existing apps.

Ben Lopatin
Django Standalone Apps

A developer’s fieldguide to developing reusable Django applications

Ben Lopatin

© 2015 - 2016 Ben Lopatin
## Contents

Just a sample ................................................................. i  
Introduction ............................................................... i  

1. Testing, part I: beyond the Django project .................. 1  
   Why test? ................................................................. 1  
   Review: testing apps from a Django project .................. 2  
   Testing the app, theay ................................................. 2  
   Testing outside of a project ........................................ 3  
   Testing application relationships ................................ 6  
   Testing without Django .............................................. 8  


Hello! This book is a work in progress, but I wanted to share a part of it that might give you an idea of what to expect from the complete book, and hopefully to provide some help, too.

In that vein I’ve included the first section on testing. Figuring out how to test a standalone or reusable app was one of the trickier things I had to figure out on my own when I wrote my first reusable app, so hopefully this will keep a few people from ripping their hair out!

After reading through the sample here, make sure to sign up on the Leanpub book page so I can let you know when the first draft of the entire book is ready for the public: https://leanpub.com/standalone-django-apps

And if you have any questions or suggestions, I encourage you to email me: ben@benlopatin.com.

Happy coding,

Ben
Introduction

from standalone_apps import intro

Who this book is for

This book is for Django developers who are either new to writing their own standalone apps or who have written them before but are looking for some common guidance. You don’t need to be a Django guru-ninja-wizard-whatever to write standalone apps but you will need at least moderate familiarity with Django to use this book.

This book is for developers in two categories:

- You have been working with Django for at least a little while and now would like to create a standalone app
- You have written at least one Django standalone app already but there are aspects of creating another or managing your app, from process to corner cases that you’re unsure of.

How the book is organized.

Django Standalone Apps is divided into four sections.

1. A basic standalone app from start to finish

The first section, Getting Started, is geared toward the developer who has never written a standalone app and maybe who finds the idea of putting together, and publishing, a Python package a bit daunting. What we cover in this section is designed to be enough to get you started with a basic but working and publishable standalone app.
2. The strategy and mechanics of extracting an app

The second section is a little less code centric. Here we’re concerned with the concepts around pulling standalone apps out from existing projects.

In this situation you seemingly have an advantage since you already have working code written, however if you want to extract for reuse you have to identify where to draw the line between likely coupled code in your own project. More challenging is the prospect of not just making a reusable app based on your code, but replacing your original code with your new standalone app.

This section deals with the questions, the decisions, and some strategies for these situations.

3. Dealing with more complex project considerations

The third section, “Advanced Considerations”, carries forward the topics in section one including how to work through further issues. How do you best deal with app-specific configuration? How can you support multiple versions of Django or Python?

Some of the topics here overlap with issues you’ll face in any Django project, however the details of implementing them and implementing them well in a standalone app may pose issues. This is especially true when you start considering scenarios beyond your own immediate use case, something you don’t have to worry about in your own projects.

4. Managing a Django app over its lifetime

If the goal of creating a reusable app is to streamline and simplify the process of building Django websites, then here we want to streamline and simplify the process of building and maintaining reusable Django apps.

The fourth section is about managing a Django standalone app in the wild. How do you ensure you have a good release every time? What’s the best way of ensuring a Django version compatible pipeline?

There’s some overlap here, too, with managing any software package, Python or otherwise, with a special focus on Django.
1. Testing, part I: beyond the Django project

Why test?

Everyone says that you should test. It sounds obvious, if testing is good, we should do it. But this begs the question about the benefits of testing.

Testing serves several purposes. Written in conjunction with, or before your application code, tests help provide a working specification against which your code can be verified. In this capacity they can also help reshape the code and interface, as if you’re adding some feature from scratch a test will give you your first chance of using it.

Once in place, even otherwise trivial tests serve to protect against regressions introduced by seemingly trivial changes to the codebase.

While not their primary use, tests can also provide an example of how to use your code. In this capacity they’re certainly not a replacement for proper documentation, but tests as code examples - especially when tests are run automatically - are a form of documentation that you can at least trust is up to date.

Underlying all of this is the fact that computer programs are written by human beings and we’re terribly unreliable when it comes to writing reliable code on our own (apologies if this does not apply to you). There’s all kinds of stuff we can’t predict, stuff we’re not good at seeing right away, and interactions we don’t see at the surface of our code.

Testing doesn’t solve all of these problems but tests provide a potent tool to remove a lot of uncertainty about our code. Ultimately tests provide confidence, both for you and other users of your app - and don’t forget that “future you” is one of those users!
Review: testing apps from a Django project

Django provides a way to run tests in Django apps by using the `test` management command combined with the app name.

```python
python manage.py test myapp
```

If the app looks like this:

```
myapp/
   __init__.py
   models.py
   tests.py
```

Then the command `python manage.py test myapp` will run all of the tests in `myapp.tests`.

This works well when you’re working from a larger Django project, for example if you’re developing your app in the context of a working project. It’s of much less help if your app is a standalone library where the code is intended to be managed from outside of a project.

Testing the app, theay

If you’ve worked with other Python packages before, you’ll have noticed that they’re mostly tested in a straightforward way. There’s usually a test module and the `setup.py` file defines where the test script is. That works for Django apps, too, with the caveat that much Django functionality must be run from the context of a Django project.

To motivate some reasonable ways of testing a standalone app, let’s consider the most immediately strategy for testing the app: testing from whatever project you’re using the app in (presuming you are extracting it).

This means that to test the `myapp` app, it needs to be installed on the same path as your working project, i.e. the same virtual environment, and that it needs to be in your
working project’s INSTALLED_APPS. When it’s time to test changes to myapp you’ll need to go back to the working project to run them.

If this sounds less than sensible, you’re on the right track. It doesn’t allow testing of the app by itself, which means it’s not repeatable for anyone else who isn’t working with your project. And even then it’s a pain in the tucchini.

**Testing outside of a project**

We did this stupid thing because in order to test a Django app we need a lot of stuff from Django. If you try to run a test of some code that imports Django machinery, like models, for instance, you’ll get errors that Django’s improperly configured. So to deal with this we need a Django project.

**Using an example project**

Next step is to create an example project in the package root that will be a stripped down project only including our app. Now we can run manage.py commands directly in our package and test the app. Just add a bash script at the project root that will execute the tests no matter where they’re located.

Here’s what the layout would look like:

```bash
django_project/
    sample_project
        __init__.py
        settings.py
        url.py
        wsgi.py
        __init__.py
        manage.py
    myapp/
        __init__.py
        models.py
        tests.py
    setup.py
```

Then to run the tests for your app you’d run them from the example project just as if it were a production-ready Django project.
cd django_project
python manage.py test myapp

This *works* and is an improvement over the original example, but for most scenarios is cumbersome and unnecessary.

**Using a testing script**

Of course, Django doesn’t demand that we have project scaffolding, just that Django settings are configured. So a better solution is a Python script that configures those minimalist settings and then runs the tests.

The script needs to do three things:

1. Define or configure Django settings
2. Trigger Django initialization (i.e. with `django.setup()`)  
3. Execute the test runner

In the first example here the script configures settings in-place for Django and then uses a test runner from the Django Nose plugin. Testing, Part II will cover alternative test runners in more depth, but for now this suffices to show the overall sequence.

**A runtests.py script using Nose**

```python
#!/usr/bin/env python

import sys

try:
    import django
    from django.conf import settings

    settings.configure(
        DEBUG=True,
        USE_TZ=True,
        DATABASES={
            "default": {
                "ENGINE": "django.db.backends.sqlite3",
```
The Django documentation includes a suggested pattern for testing standalone apps that uses a separate settings module.
Django docs suggested runtests.py

```python
#!/usr/bin/env python
import os
import sys

import django
from django.conf import settings
from django.test.utils import get_runner

if __name__ == '__main__':
    os.environ['DJANGO_SETTINGS_MODULE'] = 'tests.test_settings'
    django.setup()
    TestRunner = get_runner(settings)
    test_runner = TestRunner()
    failures = test_runner.run_tests(['tests'])
    sys.exit(bool(failures))
```

The first example looks more verbose because it includes the settings in place, and because it handles extra arguments for Nose.

The main difference between configuring settings manually and using an environment variable module definition is that in the latter case Django makes changes to the environment variables for the process itself:

Django sets the os.environ['TZ'] variable to the time zone you specify in the TIME_ZONE setting

In Testing, Part II in section 4 we’ll examine a yet better way of managing your tests and test configuration.

Testing application relationships

What do you do if your app absolutely requires interfacing with another Django app - e.g. one that should use your own? In this case you’ll want to create a separate Django app that you can include in your test script.

Let’s say your app provides base models. For our example it’s a very basic e-commerce module that lets people make a product out of any model they want, adding some basic fields like price, a SKU, and whether it’s actively sold or not.
myapp/models.py

class ProductBase(models.Model):
    sku = models.CharField()
    price = models.DecimalField()
    is_in_stock = models.BooleanField()

    class Meta:
        abstract = True

In this case you’ll need to add and include a small test app that depends on your base app. In your test script or wherever you set up Django for testing include this app as an installed app.

For your tests let’s say you want to write an integration test that shows a product on a page. Your app doesn’t actually have any products since you only have an abstract base model, so you’ll need to use a ‘concrete’ model. This is where your example app comes into play.

This app need only provide the bare minimum to be an app.

test_app/
    __init__.py
    models.py

And in your models file define a model using your app’s abstract base model.

test_app/models.py

from myapp.models import ProductBase

class Pen(ProductBase):
    """Product class for writing instruments"""
    name = models.CharField()
    pen_type = models.CharField()

In your testing script, make sure to add the example app to INSTALLED_APPS.
INSTALLED_APPS = [
    'myapp',
    'test_app',
]

Your tests in this case should live in a separate, top level module outside of your app.

myapp/
    __init__.py
    models.py

test_app/
    __init__.py
    models.py

tests/
    __init__.py
    test_models.py

Testing without Django

Lastly, in many cases you can eschew all of this!

The emphasis here is on Django apps, that is, Python modules that can be installed and included in a Django project to use models, template tags, management commands, etc. But in many cases the functionality provided by apps can be tested as plain old Python code.

This will be the case with anything in your app that requires setup, like models. However this isn’t true of every part of Django or every part of your app. And in fact if your app doesn’t have any models, and you don’t have any request related functionality to test - especially at an integration test level - then you can forgo with setting up or using Django’s test modules, sticking to the standard library’s unittest - or any other testing framework you so choose.

In most cases, testing forms, the logic in template tags and filters, etc, is not dependent on any of the parts of Django that require project setup.

Why would you do this? It’s extraordinarily doubtful that the performance gains from using unittest over django.test are going to be noticeable to say nothing of impactful. However if these are the only tests that you need then your testing environment will be simpler to set up and run.