Recommended Django Project Layout

— by Frank Wiles on Nov 21, 2014

What’s the optimal layout for your Django applications, settings files, and various other associated directories?

When Django 1.4 was released it included an updated project layout(https://docs.djangoproject.com/en/dev/releases/1.4/#updated-default-project-layout-and-manage-py) which went a long way to improving the default Django project’s layout, but here are some tips for making it even better.

This is a question we get asked all of the time so I wanted to take a bit of time and write down exactly how we feel about this subject so we can easily refer clients to this document. Note that this was written using Django version 1.7.1, but can be applied to any Django version after 1.4 easily.

Why this layout is better

The project layout we’re recommending here has several advantages namely:

- Allows you to pick up, repackage, and reuse individual Django applications for use in other projects. Often it isn’t clear as you are building an app whether or not it is even a candidate for reuse. Building it this way from the start makes it much easier if that time comes.
- Encourages designing applications for reuse
- Environment specific settings. No more `if DEBUG==True` nonsense in a single monolithic settings file. This allows to easily see which settings are shared and what is overridden on a per environment basis.
- Environment specific PIP requirements
- Project level templates and static files that can, if necessary, override app level defaults.
- Small more specific test files which are easier to read and understand.

Assuming you have two apps *blog* and *users* and 2 environments dev and prod your project layout should be structured like this:

```
myproject/
    manage.py
    myproject/
    __init__.py
    urls.py
    wsgi.py
    settings/
        __init__.py
        base.py
        dev.py
        prod.py
blog/
    __init__.py
    models.py
    managers.py
    views.py
    urls.py
    templates/
        blog/
            base.html
            list.html
            detail.html
    static/
       …
    tests/
        __init__.py
        test_models.py
        test_managers.py
        test_views.py
users/
    __init__.py
    models.py
    views.py
    urls.py
    templates/
        users/
            base.html
            list.html
            detail.html
    static/
        …
    tests/
        __init__.py
        test_models.py
        test_views.py
 static/
     css/
         …
     js/
         …
 templates/
     base.html
     index.html
 requirements/
     base.txt
     dev.txt
```

```
        test.txt
        prod.txt
```

The rest of this article explains how to move a project to this layout and why this layout is better.

Current Default Layout

We're going to call our example project **foo**, yes I realize it's a very creative name. We're assuming here that we're going to be launching **foo.com** but while we like to have our project names reflect the ultimate domain(s) the project will live on this isn't by any means required.

If you kick off your project using `django-admin.py startproject foo` you get a directory structure like this:

```
foo/
    manage.py
    foo/
       __init__.py
       settings.py
       urls.py
       wsgi.py
```

This layout is a great starting place, we have a top level directory *foo* which contains our manage.py the *project* directory *foo/foo/* inside it. This is the directory you would check into your source control system such as git.

You should think of this *foo/foo/* subdirectory as being **the project** where everything else is either a Django application or ancillary files related to the project.

Fixing Settings

We're on a mission to fix your bad settings files here. We show this layout to new clients and I'm constantly surprised how few people know this is even possible to do. I blame the fact that while everyone knows that settings are just Python code, they don't think about them *as* Python code.

So let's fix up our settings. For our foo project we're going to have 4 environments: dev, stage, jenkins, and production. So let's give each it's own file. The process to do this is:

1. In *foo/foo/* make a settings directory and create an empty `__init__.py` file inside it.
2. Move *foo/foo/settings.py* into *foo/foo/settings/base.py*
3. Create the individual *dev.py*, *stage.py*, *jenkins.py*, and *prod.py* files in *foo/foo/settings/*. Each of these 4 environment specific files should simply contain the following:

   ```
   from base import *
   ```

So why is this important? Well for local development you want `DEBUG=True` , but it's pretty easy to accidentally push out production code with it on, so just open up *foo/foo/settings/prod.py* and after the initial import from base just add `DEBUG=False` . Now if your production site is safe from that silly mistake.

What else can you customize? Well it should be pretty obvious you'll likely have staging, jenkins, and production all pointing at different databases, likely even on different hosts. So adjust those settings in each environment file.

Using these settings

Using these settings is easy, no matter which method you typically use. To use the OS's environment you just do:

```
export DJANGO_SETTINGS_MODULE="foo.settings.jenkins"
```

And boom, you're now using the jenkins configuration.

Or maybe you prefer to pass them in as a commandline option like this:

```
./manage.py migrate —settings=foo.settings.production
```

Same if you're using gunicorn:

```
gunicorn -w 4 -b 127.0.0.1:8001 —settings=foo.settings.dev
```

What else should be customized about settings?

Another useful tip with Django settings is to change several of the default settings *collections* from being tuples to being lists. For example `INSTALLED_APPS`, by changing it from:

```
INSTALLED_APPS = (
   …
)
```

to:

```
INSTALLED_APPS = [
    …
]
```

In *foo/settings/base.py* we can now more easily add and remove apps based on each environment specific settings file. For example, maybe you only want django-debug-toolbar installed in dev, but not your other environments.

This trick is also often useful for the `TEMPLATE_DIRS` and `MIDDLEWARE_CLASSES` settings.

Another useful trick we often use is to break up your apps into two lists, one your prerequisites and another for your actual project applications. So like this:

```
PREREQ_APPS = [
   ‘django.contrib.auth’,
   ‘django.contrib.contenttypes’,
   …
   ‘debug_toolbar’,
   ‘imagekit’,
   ‘haystack’,
]

PROJECT_APPS = [
   ‘homepage’,
   ‘users’,
   ‘blog’,
]

INSTALLED_APPS = PREREQ_APPS + PROJECT_APPS
```

Why is this useful? For one it helps better distinguish between Django core apps, third party apps, and your own internal project specific applications. However, `PROJECT_APPS` often comes in handy as a list of your specific apps for things like testing and code coverage. You have a list of *your* apps, so you can easily and automagically make sure their tests are run and coverage is recorded just for them, not including any third party apps, without having to maintain the list in two separate places.

Fixing requirements

Most projects have a single `requirements.txt` file that is installed like this:

```
pip install -r requirements.txt
```

This is sufficient for small simple projects, but a little known feature of requirements files is that you can use the `-r` flag to include other files. So we can have a base.txt of all the common requirements and then if we need to be able to run tests have a specific *requirements/test.txt* that looks like this:

```
-r base.txt
pytest==2.5.2
coverage==3.7.1
```

I’ll admit this is not a *HUGE* benefit, but it does help separate out what is a requirement in which environment. And for the truly performance conscience it reduces your pip install time in production a touch by not installing a bunch of things that won’t actually be used in production.

Test Files

Why did we separate out the tests files so much? One main reason, if you’re writing enough tests a single tests.py file per application will end up being one huge honking file. This is bad for readability, but also just for the simple fact you have to spend time scrolling around a lot in your editor.

You’ll also end up with less merge conflicts when working with other developers which is a nice side benefit. Small files are your friends.

URLs

For small projects it’s tempting to put all of your url definitions in *foo/urls.py* to keep them all in one place. However, if your goal is clarity and reusability you want to define your urls in each app and include them into your main project. So instead of:

```
urlpatterns = patterns(‘’,
    url(r’^$’, HomePageView.as_view(), name=‘home’),
    url(r’^blog/$’, BlogList.as_view(), name=‘blog_list’),
    url(r’^blog/(?P<pk>\d+)/$’, BlogDetail.as_view(), name=‘blog_detail’),
    …
    url(r’^user/list/$’, UserList.as_view(), name=‘user_list’),
    url(r’^user/(?P<username>\w+)/$’, UserDetail.as_view(), name=‘user_detail’),
)
```

you should do this:

```
urlpatterns = patterns(‘’,
    url(r’^$’, HomePageView.as_view(), name=‘home’),
    url(r’^blog/‘, include(‘blog.urls’)),
    url(r’^user/‘, include(‘user.urls’)),
)
```

Templates and static media

Having per app *templates/* and *static/* directories gives us the ability to reuse an application basically **as is** in another project.

We get the default templates the app provides and any associated static media like special Javascript for that one cool feature all in one package.

However, it *also* gives us the ability to override those templates on a per project basis in the main *foo/templates/* directory. By adding a `templates/blog/detail.html` template we override, or mask, the default `blog/templates/blog/detail.html` template.

Reusing a Django application

So assuming you’ve been using this layout for awhile, one day you’ll realize that your new project needs a blog and the one from your *foo* project would be perfect for it. So you copy and paste the files in… *pssst WRONG!*. Now you have two copies of the application out there. Bug fixes or new features in one have to manually be moved between the projects, and that assumes you even remember to do that.

Instead, make a new repo for your blog and put the *foo/blog/* directory in it. And adjust both your existing foo project and your new project to pip install it.

They can still both track different versions of the app, if necessary, or keep up to date and get all of your bug fixes and new features as they develop. You still can override the templates and static media as you need to on a per project basis, so there really isn’t any real issues doing this.

Additional Resources

Our friends Danny and Audrey over at CartWheel Web(http://www.cartwheelweb.com) reminded us about Cookie Cutter(https://github.com/audreyr/cookiecutter) and specifically Danny's cookiecutter-django(https://github.com/pydanny/cookiecutter-django) as useful tools for making your initial project creations easier and repeatable.

Also, if you're looking for all around great Django tips and best practices, you can't go wrong with their book Two Scoops of Django: Best Practices For Django 1.6(http://www.amazon.com/gp/product/098146730X/ref=as_li_qf_sp_asin_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=098146730X&linkCode=as2&tag=revosystblog-20) which we recommend to all of our clients.

Feedback

We hope you find this improved project layout useful. If you find any bugs, have a suggestion, or just want to chat feel free to reach out to us. Thanks for reading!

Tags: django(/blog/tags/django/), Featured Posts(/blog/tags/Featured%20Posts/), programming(/blog/tags/programming/)