django-reusable-app-docs Documentation

Release 0.1.0

Brian Rosner

May 12, 2017

Contents

This is a living document of best practices in developing and deploying with the Django Web framework. These should not be seen as the *right* way or the *only* way to work with Django, but instead best practices we've honed after years of working with the framework.

It is a fork of the great django-reusable-app-docs project started by Brian Rosner and Eric Holscher regarding best practices for writing and maintaining reusable Django apps.

Note: The source code for this documentation lives on GitHub as django-best-practices and can be built in a number of formats using Sphinx.

CHAPTER 1

Feedback

See a problem? Disagree with something? Want to see other topics covered? **We'd love to hear your feedback!** Email us or file an issue.

CHAPTER 2

Table of Contents

Coding Style

In general, code should be clean, concise and readable. The Zen of Python (PEP 20) is a great introduction to best coding practices for Python.

- Follow the Style Guide for Python Code (PEP 8) as closely as reasonable.
- Follow the Django coding style.

Django Projects

At its core, a Django project requires nothing more than a settings file. In practice, almost every project consists of the following items:

- *Settings*
- *URLconf*
- *WSGI File*
- *Local Applications*
- *Templates*
- *Static Media*
- manage.py

Settings

The settings module is the only true requirement for a Django project. Typically, it lives in the root of your project as `settings.py`.

Handling Settings for Multiple Environments

Django's `startproject` command gives you a single `settings.py` file. If you're new to Django, stick with the single file while you learn the ropes. As you start to deploy production sites and work with more than one developer, you'll realize the benefit in maintaining multiple settings files. For example, you probably want to run with `DEBUG` on locally, but not in production.

There are numerous ways to handle multiple settings. Whatever solution you choose, it should meet the following requirements:

- All the important **settings files are version controlled**. If the settings change on your production site, you'll want to know who made the changes and when they were made.
- **All settings inherit from a common base**. If you want to add `django-debug-toolbar` to your `INSTALLED_APPS`, you should be able to do it without redefining all your `INSTALLED_APPS`.

If you don't want to think about it, simply use our Django project template when starting new projects. It is ready to support multiple projects out of the gate:

```
django-admin.py startproject --template=https://github.com/lincolnloop/django-layout/
↪tarball/master -e py,rst,example,gitignore my_project_name
```

See also:

Django's Split Settings Wiki Examples of handling multiple settings

Handling File Paths

One function of your settings is to tell Django where to find things such as your static media and templates. Most likely they'll already live inside your project. If so, let Python generate the absolute path names for you. This makes your project portable across different environments.

```
import os
DIRNAME = os.path.dirname(__file__)
# ...
STATIC_ROOT = os.path.join(DIRNAME, 'static')
```

URLconf

By default, you'll find your URLconf in the root of your project as `urls.py`. It defines how requests should be routed for your project.

Keep it Simple

Your project URLconf should simply include URLconfs from your applications whenever possible. This keeps your application logic inside your application and your project simply serves as a pointer to it.

See also:

Django URL dispatcher documentation Including other URLconfs

Handling URLconfs for Multiple Environments

Just like your settings module, eventually, you'll come across the need to run different URLconfs for different environments. You may want to use `admin` locally, but not once deployed. Django already provides an easy way for you to do this with the ROOT_URLCONF setting.

This is basically the same scenario as having *multiple settings*. You can use the same solution here:

```
myproject
  ...
  settings/
        __init__.py
        base.py          <-- shared by all environments
        def.py
        production.py
  urls/
    __init__.py
        base.py                  <-- shared by all environments
    dev.py
    production.py
  ...
```

See also:

Our django-layout template

WSGI File

The WSGI file tells your WSGI server what it needs to do to serve your project on the web. Django's default `wsgi.py` is sufficient for most applications.

Local Applications

Local applications are Django applications that are domain-specific to your project. They typically live inside the project module and are so closely tied to your project, they would have little use outside of it.

Local vs. Third Party

There are hundreds[1] of open source Django applications available. Before you reinvent the wheel, make sure somebody hasn't already solved your problem by searching on Google or Django Packages. If you find something that will work do **not** put it your project code, instead add it to your *pip requirements*.

The Namespace

How local applications should be imported into your project is a source of ongoing debate in the Django community[2]. Fortunately, with the release of Django 1.4, the default `manage.py` no longer changes the `PYTHONPATH`[3], making this much less of an issue.

At Lincoln Loop, we put project applications inside the project namespace. This prevents polluting the global namespace and running into potential naming conflicts.

[1] http://djangopackages.com/categories/apps/
[2] Discussion on django-developers mailing list regarding project namespaces in the tutorial
[3] Django 1.4 manage.py changes

Templates

Location

Templates typically live in one of two places, inside the application or at the root level of a project. We recommend keeping all your templates in the project template directory unless you plan on including your application in multiple projects (or developing it as a open source "reusable" application). In that case, it can be helpful to ship with a set of sample templates in the application, allowing it to work out-of-the-box or serving as an example for other developers.

Naming

Django's generic views provide an excellent pattern for naming templates. Following design patterns already found in Django can be helpful for a couple reasons.

1. They have been well thought out and tested.
2. It makes your code immediately understandable to new developers picking up your Django code.

Most generic view templates are named in the format:

```
[application]/[model]_[function].html
```

For example, creating a template to list all of the contacts (`Contact` model) in my address book (`address_book` application), I would use the following template:

```
address_book/contact_list.html
```

Similarly, a detail view of a contact would use:

```
address_book/contact_detail.html
```

Not every template you create will map so closely to a single model, however. In those cases, you're on your own for naming, but should still keep your templates in a directory with the same name as your application.

When using inclusion tags or other other functionality to render partial templates, keep them in an `includes` directory inside the application template directory. For example, if I had an inclusion tag to render a contact form inside my address book application, I would create a template for it at:

```
address_book/includes/contact_form.html
```

There is no rule (anymore) that templates must have an `html` file extension. If you are rendering something else (plain text, JSON, XML, etc), your templates file extension should match that of the content you are generating.

Static Media

Static media encompasses all the non-dynamic content needed for your website: CSS, images, JavaScript, Flash, etc. It comes in two flavors, user-generated content and the media needed to render your site. Best practice dictates that *your* static media lives inside your project and your version control system. Certainly, we don't want stuff our users' uploads to go to the same place. As such, we always use `django.contrib.staticfiles`[4].

In addition to some other slick features, `staticfiles` gives you a `static` template tag[5] that will properly locate your static files whether they are on your local computer or in a non-local storage on your production system. This leaves `MEDIA_URL` and `MEDIA_ROOT` to manage user generated content.

[4] https://docs.djangoproject.com/en/dev/ref/contrib/staticfiles/
[5] https://docs.djangoproject.com/en/dev/ref/contrib/staticfiles/#std:templatetag-staticfiles-static

See also:

On Static Media and Django

Django Applications

A Django project typically consists of many applications declared in `INSTALLED_APPS`. Django applications should follow the Unix philosopy of, "Do one thing and do it well."[1], with a focus on being small and modular, mirroring Django's "loose coupling" design philosophy[2].

James Bennett's Reusable Apps talk at the first DjangoCon is an excellent primer on the subject of building good Django applications.

Code Organization

The only requirement of a Django application is that it provides a `models.py` file. In practice, however, Django applications are made up of many different files. When building your own applications, follow common file naming conventions. Start with the framework Django provides via `manage.py startapp <foo>` and build out from there as needed.

- `__init__.py`
- `admin.py`
- `context_processors.py`
- `feeds.py`
- `forms.py`
- `managers.py`
- `middleware.py`
- `models.py`
- `receivers.py`
- `signals.py`
- `templates/app_name/`
- `templatetags/`
  - `__init__.py`
  - `app_name.py`
- `tests.py` or `tests/`
- `urls.py`
- `views.py`

What lives in each of these files should be self-explanatory. Let's dive into some of the meatier ones though.

[1] http://en.wikipedia.org/wiki/Unix_philosophy#McIlroy:_A_Quarter_Century_of_Unix
[2] https://docs.djangoproject.com/en/dev/misc/design-philosophies/#loose-coupling

Models

Style

Follow Django's defined conventions for model code.

Make 'em Fat

A common pattern in MVC-style programming is to build thick/fat models and thin controllers. For Django this translates to building models with lots of small methods attached to them and views which use those methods to keep their logic as minimal as possible. There are lots of benefits to this approach.

1. **DRY**: Rather than repeating the same logic in multiple views, it is defined once on the model.
2. **Testable**: Breaking up logic into small methods on the model makes your code easier to unit test.
3. **Readable**: By giving your methods friendly names, you can abstract ugly logic into something that is easily readable and understandable.

For a good example of a fat model in Django, look at the definition of `django.contrib.auth.models.User`.

Managers

Similar to models, it's good practice to abstract common logic into methods on a manager. More specifically, you'll probably want a chainable method that you can use on any queryset. This involves some boilerplate that I always forget, so here's an example for (mostly) cutting and pasting:

```
import datetime
from django.db import models
from django.db.models.query import QuerySet

class PostQuerySet(QuerySet):
    def live(self):
        """Filter out posts that aren't ready to be published"""
        now = datetime.datetime.now()
        return self.filter(date_published__lte=now, status="published")

class PostManager(models.Manager):
    def get_query_set(self):
        return PostQuerySet(self.model)
    def __getattr__(self, attr, *args):
        # see https://code.djangoproject.com/ticket/15062 for details
        if attr.startswith("_"):
            raise AttributeError
        return getattr(self.get_query_set(), attr, *args)

class Post(models.Model):
    # field definitions...
    objects = PostManager()
```

This code will let you call our new method *live* both directly on the manager `Post.objects.live()` and chain it on a queryset `Post.objects.filter(category="tech").live()`. At the time of writing, there is an open bug to make this less painful.

Deployment

Contents:

Project Bootstrapping

Filesystem Layout

Note: This document is heavily biased towards Unix-style filesystems and may require additional effort to use in other operating systems.

Virtualenv is a must for Python projects. It provides a method to isolate different Python environments. We typically host our production sites from `/opt/webapps/<site_name>` and our development sites from `~/webapps/<site_name>`. Each individual project gets its own `virtualenv` that also serves as the directory for all the source files associated with the project. We use `pip` to populate the `virtualenv` with the necessary packages.

The bootstrap process looks like this:

```
cd /opt/webapps
virtualenv mysite.com
cd mysite.com
source bin/activate
pip install -r path/to/requirements.txt
```

Tip: For convenience, you can use virtualenvwrapper which provides some helpers to make working with virtualenvs more friendly.

Packaging

One of the keys to successful deployment is to ensure that the software you develop on is as close as possible to the software you deploy on. Pip provides a simple repeatable method allowing you to consistently deploy Python projects across many machines. Every application that requires third-party libraries should include a pip requirements file called `requirements.txt`. Projects should aggregate the application requirements files adding any additional requirements as needed.

What to include in your requirements files

In short, everything. While your operating system may provide some Python packages, nearly everything can be installed cleanly with `pip` these days. By installing everything into your virtualenv, you can isolate your environment and prevent system packages from causing version conflicts.

Warning: **Pin your dependencies!** Pip makes it easy to install from a VCS, or just grab whatever version it finds on PyPI. This also makes it easy for your deployments to have different versions of different libraries which can lead to unexpected results. Make sure you specify a version for PyPI libs or a specific commit/tag for VCS checkouts. Examples: `django==1.4.1` or `-e git+https://github.com/toastdriven/django-tastypie.git@v0.9.9#egg=django-tastypie`

Servers

Note: Deployment arcitectures vary widely depending on the needs and traffic of the site. The setup described below is minimally configured and works well for most instances.

We serve Django on Ubuntu Linux with a PostgreSQL database backend via gunicorn or uWSGI from behind an Nginx frontend proxy. For simplicity, we'll only be discussing Gunicorn/Nginx here.

Nginx

Nginx makes for a great frontend server due to its speed, stability and low resource footprint. The typical Nginx configuration for a site looks like this:

```
# Gunicorn server
upstream django {
  server          domain.com:9000;
}

# Redirect all requests on the www subdomain to the root domain
server {
  listen      80;
  server_name www.domain.com;
  rewrite ^/(.*) http://domain.com/$1 permanent;
}

# Serve static files and redirect any other request to Gunicorn
server {
  listen       80;
  server_name  domain.com;
  root        /var/www/domain.com/;
  access_log  /var/log/nginx/domain.com.access.log;
  error_log  /var/log/nginx/domain.com.error.log;

  # Check if a file exists at /var/www/domain/ for the incoming request.
  # If it doesn't proxy to Gunicorn/Django.
  try_files $uri @django;

  # Setup named location for Django requests and handle proxy details
  location @django {
    proxy_pass          http://django;
    proxy_redirect      off;
    proxy_set_header    Host             $host;
    proxy_set_header    X-Real-IP        $remote_addr;
    proxy_set_header    X-Forwarded-For  $proxy_add_x_forwarded_for;
  }
}
```

What Does it Do?

The first block tells Nginx where to find the server hosting our Django site. The second block redirects any request coming in on `www.domain.com` to `domain.com` so each resource has only one canonical URL. The final section is the one that does all the work. It tells Nginx to check if a file matching the request exists in `/var/www/domain.com`. If it does, it serves that file, if it doesn't, it proxies the request to the Django site.

SSL

Another benefit to running a frontend server is SSL termination. Rather than having two Django instances running for SSL and non-SSL access, we can have Nginx act as the gatekeeper redirecting all requests back to a single non-SSL WSGI instance listening on the `localhost`. Here's what that would look like:

```
server {
  listen       67.207.128.83:443; #replace with your own ip address
  server_name  domain.com;
  root        /var/www/domain.com/;
  access_log  /var/log/nginx/domain.com.access.log;

  ssl on;
  ssl_certificate /etc/nginx/ssl/certs/domain.com.crt;
  ssl_certificate_key /etc/nginx/ssl/private/domain.com.key;
  ssl_prefer_server_ciphers       on;

  # Check if a file exists at /var/www/domain/ for the incoming request.
  # If it doesn't proxy to Gunicorn/Django.
  try_files $uri @django;

  # Setup named location for Django requests and handle proxy details
  location @django {
    proxy_pass          http://django;
    proxy_redirect      off;
    proxy_set_header    Host              $host;
    proxy_set_header    X-Real-IP         $remote_addr;
    proxy_set_header    X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header    X-Forwarded-Protocol ssl;
  }

}
```

You can include this code at the bottom of your non-SSL configuration file.

Gunicorn

Gunicorn is a lightweight WSGI server that can scale from small deploys to high-traffic sites. You can install it via `pip install gunicorn`. Since Nginx will be listening for HTTP(S) requests, you'll need to bind Gunicorn to a different port. While you're at it, you can tell it to only respond to the `localhost`. A simple gunicorn process might look like this:

```
$ gunicorn --workers=4 --bind=127.0.0.1:9000 my_project.wsgi:application
```

This spawns a gunicorn process with 4 workers listening on http://127.0.0.1:9000. If your project doesn't already have a `wsgi.py` file, you'll want to add one. See the Django WSGI docs or django-layout for an example.

Process Management

You want to be sure that gunicorn is always running and that it starts up automatically after a server reboot. If you are deploying to Ubuntu, `upstart` is probably the easiest way to get started. Here is a sample config:

```
# logs to /var/log/upstart/my_project.log

description "my_project"
```

```
start on startup
stop on shutdown

respawn

# start from virtualenv path
exec /opt/webapps/my_project/bin/gunicorn  -w 4 -b 127.0.0.1:9000 my_project.
↪wsgi:application
setuid www-data
```

Save this file to `/etc/init/gunicorn.conf` and run `sudo start gunicorn`. For troubleshooting, your logs will be visible at `/var/log/upstart/gunicorn.log`.

Note: Supervisor is a pure Python option if you don't have access to `upstart`.

Index