~~Everything I know about Python... (/)~~

> Learn to Write Pythonic Code!
>
> Check out the book *Writing Idiomatic Python*! (https://www.jeffknupp.com/writing-idiomatic-python-ebook/)

Writing Idiomatic Python (//www.jeffknupp.com/writing-idiomatic-python-ebook/)

Learn to Write "Pythonic" Code (//www.jeffknupp.com/writing-idiomatic-python-ebook/)

Best of jeffknupp.com:

Open Sourcing a Python Project the Right Way (http://www.jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/)
Python and Flask are Ridiculously Powerful (http://www.jeffknupp.com/blog/2014/01/18/python-and-flask-are-ridiculously-powerful/)
Starting a Django 1.6 Project the Right Way (http://www.jeffknupp.com/blog/2013/12/18/starting-a-django-16-project-the-right-way/)
Python's Hardest Problem (http://www.jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/)
Writing Idiomatic Python (http://www.jeffknupp.com/blog/2012/10/04/writing-idiomatic-

python/)

Starting a Django 1.6 Project the Right Way (/blog/2013/12/18/starting-a-django-16-project-the-right-way)

Back in February of 2012, I wrote an article titled 'Starting a Django Project the Right Way' (http://www.jeffknupp.com/blog/2012/02/09/starting-a-django-project-the-right-way/), and later followed up with 'Starting a Django 1.4 Project the Right Way' (http://www.jeffknupp.com/blog/2012/10/24/starting-a-django-14-project-the-right-way/). Both of these articles still draw a consistent audience and are referenced in numerous StackOverflow answers, corporate wikis, and tweets. With 1.5 and *1.6 already out, now seems like an appropriate time to update the article again.*

The beginning of a project is a critical time, when choices are made that have long term consequences. There are a number of tutorials about how to get started with the Django framework, but few that discuss how to use Django in a professional way, using industry accepted best practices to make sure your project's development practices scale as your application grows. A small bit of planning goes a *long* way towards making your life (and the lives of any coworkers) easier in the future.

By the end of this post, you will have

1. A fully functional Django 1.6 project
2. All resources under source control (with git or Mercurial)
3. Automated regression and unit testing (using the unittest library)
4. An environment independent install of your project (using virtualenv)
5. Automated deployment and testing (using Fabric)
6. Automatic database migrations (using South)
7. A development work flow that scales with your site.

None of these steps, except for perhaps the first, are covered in the official tutorial. **They should be**. If you're looking to start a new, production ready Django 1.6 project, look no further.

Prerequisites

A working knowledge of Python is assumed. Also, some prior experience with Django would be incredibly helpful, but not strictly necessary. You'll need git (http://www.git-scm.com) or Mercurial (http://mercurial.selenic.com/) for version control. That's it!

Preparing To Install

I'm assuming you have Python installed. If you don't head over to python.org (http://www.python.org) and find the install instructions for your architecture/os. I'll be running on a 64-bit Arch server installation hosted by Linode (http://www.linode.com/?r=ae1808f234f8e219de24842336fada09ef81d52f), with whom I'm very happy.

So, what's the first step? Install Django, right? Not quite. One common problem with installing packages directly to your current site-packages area is that, if you have more than one project or use Python on your machine for things other than Django, you may run into dependency issues between your applications and the installed packages. For this reason, we'll be using virtualenv (http://pypi.python.org/pypi/virtualenv) and the excellent extension virtualenvwrapper (http://virtualenvwrapper.readthedocs.org/en/latest/) to manage our Django installation. This is common, recommended practice among Python and Django users alike.

If you're using pip to install packages (and I can't see why you wouldn't), you can get both virtualenv and virtualenvwrapper by simply installing the latter.

```
$ pip install virtualenvwrapper
```

After it's installed, add the following lines to your shell's start-up file (.zshrc, .bashrc, .profile, etc).

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/directory-you-do-development-in
source /usr/local/bin/virtualenvwrapper.sh
```

Reload your start up file (e.g. `source .zshrc`) and you're ready to go.

Creating a New Environment

Creating a virtual environment is simple. Just type

```
$ mkvirtualenv django_project
```

where `django_project` is whatever name you give to your project.

You'll notice a few things happen right away:

- Your shell is prepended by `(django_project)`
- distribute and pip were automatically installed

This is an extremely helpful part of virtualenvwrapper: it automatically prepares your environment in a way that lets you start installing packages using pip right away. The `django_project` portion is a reminder that you're using a virtualenv instead of your

system's Python installation. To exit the virtual environment, simply type `deactivate`. When you want to resume work on your project, it's as easy as `workon django_project`. Note that unlike the vanilla virtualenv tool, *where* you run these commands doesn't matter.

Installing Django

"Wait, 'Installing Django'? I already have Django installed!" Fantastic. You aren't going to use it. Instead, we'll use one managed by virtualenv that can't be messed up by other users (or yourself) working elsewhere on the machine. To install Django under virtualenv, just type:

```
$ pip install django
```

This should give you the latest version of Django which will be installed in your virtualenv area. You can confirm this by doing:

```
$ which django-admin.py
```

Which should point to your `$HOME/.virtualenvs/` directory. If it doesn't, make sure you see `django_project` before your prompt. If you don't, activate the virtualenv using `workon django_project`.

Setting Up The Project

Before we actually start the project, we need to have a little talk. I've consulted on a number of Django/Python projects and spoken to numerous developers in the last few years. Overwhelmingly, the ones having the most difficulty are those that do not use any form of version control. It may sound unbelievable (considering the popularity of GitHub (http://www.github.com)), but developers have simply never been exposed to version control. Others think that since "this is a small project," that it's not necessary. **Wrong**.

None of the tools listed here will pay greater dividends then the use of a version control system.

Previously, I only mentioned git as a (D)VCS. However, this project being in Python, Mercurial is a worthy Python based alternative. Both are popular enough that learning resources abound online. Make sure you have either git or Mercurial installed. Both are almost certainly available via your distro's packaging system.

If you plan on using git, GitHub (http://www.github.com) is an obvious choice for keeping a remote repository. With Mercurial, Atlassian's Bitbucket (https://bitbucket.org/) is a fine choice (it supports git as well, so you could use it in either case).

(source) Controlling Your Environment

Even though we haven't actually done anything yet, we know we're going to want everything under source control. We have two types of 'things' we're going to be committing: our code itself (including templates, etc) and supporting files like database fixtures, South migrations (more on that later), and a `requirements.txt` file, which lists all of the packages your project depends on and allows automated construction of environments (without your having to `pip install` everything again).

Let's go ahead and create our project directory. Use the `startproject` command supplied by `django-admin.py` to get it set up.

```
$ django-admin.py startproject django_project
```

We'll see a single directory created: `django_project`. Within the `django_project` directory, we'll see *another* `django_project` directory containing the usual suspects: `settings.py`, `urls.py`, and `wsgi.py`. At the same level as the second `django_project` directory is `manage.py`.

Intermezzo: Projects vs. Apps

You may be wondering why, back in Django 1.4, the `startproject` command was added alongside the pre-existing `startapp` command. The answer lies in the difference between Django "projects" and Django "apps". Briefly, a *project* is an entire web site or application. An "app" is a small, (hopefully) self-contained Django application that can be used in any Django project. If you're building a blogging application called "Super Blogger", then "Super Blogger" is your Django project. If "Super Blogger" supports reader polls, "polls" would be an Django app used by "Super Blogger". The idea is that your polls app should reusable in any Django project requiring polls, not just within "Super Blogger". A project is a collection of apps, along with project specific logic. An app can be used in multiple projects.

While your natural inclination might be to include a lot of "Super Blogger" specific code and information within your "polls" app, avoiding this has a number of benefits. Based on the principle of *loose coupling*, writing your apps as standalone entities prevents design decisions and bugs in your project directly affecting your app. It also means that, if you wanted to, you could pass of the development of any of your apps to another developer without them needing to access or make changes to your main project.

Like many things in software development, it takes a bit of effort up-front but pays huge dividends later.

Setting Up Our Repos

Since we have some "code" in our project now (really just some stock scripts and empty config files, but bear with me), now is as good a time as any to initialize our repositories in source control. Here's how to do that in git and Mercurial.

git

```
$ git init
```

This creates a git repository in the current directory. Lets stage all of our files to git to be committed.

```
$ git add django_project
```

Now we actually commit them to our new repo:

```
$ git commit -m 'Initial commit of django_project'
```

Mercurial

```
$ hg init
```

This creates a Mercurial repository in the current directory. Lets stage all of our files to git to be committed.

```
$ hg add django_project
```

Now we actually commit them to our new repo:

```
$ hg commit -m 'Initial commit of django_project'
```

If you plan on using a service like GitHub or Bitbucket, now would be a good time to push to them.

Using South for Database Migrations

One of the most frustrating aspects of Django is managing changes to models and the associated changes to the database. With the help of South (http://south.readthedocs.org), you can realistically create an entire application without ever writing database specific code. Changes to your models are detected and automatically made in the database through a *migration file* that South creates. This lets you both migrate the database forward for your new change and **backwards** to undo a change or series of changes. It makes your life so much easier, it's a wonder it's not included in the Django distribution.

When to begin using South

In previous articles, I recommended using South from the very beginning of your project. For relatively simple projects, this is fine. If, however, you have a ton of models that are changing rapidly as you prototype, now is not the time to use South. Rather, just blow away and re-create the database whenever you need to. You can write scripts to populate the database with some test data and edit them as needed. Once your models stop changing, however, make the move to South ASAP. It's as easy as `./manage.py convert_to_south <app_name>`.

Installation and Setup

Still in our virtualenv, install South like so:

```
$ pip install south
```

We setup South by adding it to our `INSTALLED_APPS` in the `settings.py` file for the project. Add that now, as well as your database settings for the project, then run `python manage.py syncdb`. You'll be prompted for a superuser name and password (which you can go ahead and enter). More importantly, South has setup the database with the tables it needs.

You may have noticed that we just ran `syncdb` without having adding an app to the project. We do this first so that South is installed from the beginning. All migrations to our own apps will be done using South, including the "initial" migration.

Since we've just made some pretty big changes, now would be a good time to commit. You should get used to committing frequently, as the more granular the commit, the more freedom you have in choosing something to revert to if things go wrong.

To commit, lets see what has changed.

(git)

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   django_project/settings.py
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       django_project/.settings.py.swp
#       django_project/__init__.pyc
#       django_project/settings.pyc
```

(Mercurial)

```
$ hg status
M django_project/django_project/settings.py
? django_project/django_project/.settings.py.swp
? django_project/django_project/__init__.pyc
? django_project/django_project/settings.pyc
```

With both git and Mercurial, you may notice files you don't ever want committed, like the compiled Python .pyc files and vim swap .swp files above. To ignore these files, create a `.gitignore` or `.hgignore` file in your root project directory and add a shell glob pattern to match files you *don't* want to be tracked. For example, the contents of my file might be

```
*.pyc
.*swp
```

Before we commit, we have one more piece of information to track: our installed Python packages. We want to track the name *and version* of the Python packages we're using so that we can seamlessly recreate our environment in our production area. Helpfully, pip has a command that does exactly what we need.

```
$ pip freeze > requirements.txt
```

I piped the output to a file called `requirements.txt`, which we'll add to source control so we always have an updated list of what packages are being used.

Let's stage and commit our settings.py and requirements.txt files to be committed by running

```
$ (git/hg) add django_project/settings.py requirements.txt
$ (git/hg) commit -m 'Added South for database migrations'
```

New-Style Settings

As developers become more comfortable with Django and Python, they realize that the `settings.py` file is simply a Python script, and can thus be "programmed". One common pattern is for the `settings.py` file to be moved from the rather curious project directory to a new directory called `conf` or `config`. Just be aware you'll need to make a slight change to `manage.py` to accommodate the move.

Within `settings.py`, `INSTALLED_APPS` can quickly grow into a morass of third-party packages, in house django apps, and project specific apps. I like to divide `INSTALLED_APPS` into three categories:

- DEFAULT_APPS: Django framework apps installed as part of the default Django install (like the admin)
- THIRD_PARTY_APPS: Like South

- LOCAL_APPS: The applications you create

This makes it much easier to see what third-party applications you're using and what is home-grown. Just remember to eventually have a line similar to the following:

```
INSTALLED_APPS = DEFAULT_APPS + THIRD_PARTY_APPS + LOCAL_APPS
```

Otherwise, Django will complain about not having `INSTALLED_APPS` defined.

Creating Our App

Use `manage.py` to create an app in the normal way (`python manage.py startapp myapp`) and add it under `INSTALLED_APPS`. Also, take the time to make `manage.py` executable (`chmod +x manage.py`) so you can just type `./manage.py <command>` rather than needing to type `python manage.py <command>` all the time. Honestly, so few developers do this. I can't for the life of me figure out why.

The first thing we'll do, before adding models, is tell South we want South to manage changes to our models in the form of migrations:

```
$ python manage.py schemamigration myapp --initial
```

This creates a migration file that can be used to apply our model changes (if we had any) to the database without needing to completely destroy and rebuild it. It also also allows us to *revert* changes if things go sideways on us. We use the migration file to *migrate* the database changes (even though there are none) using :

```
$ python manage.py migrate myapp
```

South is smart enough to know where to look for migration files, as well as remember the last migration we did. You can specify individual migration files, but it's usually not necessary.

When we eventually make changes to our model, we ask South to create a migration using:

```
$ python manage.py schemamigration myapp --auto
```

This will inspect the models in `myapp` and automatically add, delete, or modify the database tables accordingly. Changes can then be applied to the database using the migrate command as above.

Our Development Area

A good habit to get into is to write and test your code separately from where you're serving

your files from, so that you don't accidentally bring down your site via a coding error when you're adding new functionality, for example. git and Mercurial make this simple. Just create a directory somewhere other than where `django_project` is installed for your development area (I just call it `dev`).

In your development (`dev`) directory, clone the current project using git or Mercurial:

```
$ (git/hg) clone /path/to/my/project/
```

Both tools will create an exact copy of the **entire** repository. All changes, branches, and history will be available here. From here on out, you should be working from your development directory.

Since branching with both git and Mercurial is so easy and cheap, create branches as you work on new, orthogonal changes to your site. Here's how to do it each tool:

(git)

```
$ git checkout -b <branchname>
```

Which will both create a new branch named and check it out. Almost all of your development should be done on a branch, so that `master` mimics the "production" (or "version live on your site") `master` and can be used for recovery at any time.

(Mercurial)

```
$ hg branch <branchname>
```

Note that branching is kind of a contentious topic within the Mercurial community, as there are a number of options available but no "obviously correct" choice. Here, I use a named branch, which is probably the safest and most informative style of branching. Any commits after the branch command are done on the branch.

Using Fabric for Deployment

So we have the makings of a Django application. How do we deploy it? **Fabric (http://www.fabfile.org)**. For a reasonable sized project, discussing anything else is a waste of time. Fabric can be used for a number of purposes, but it really shines in deployments.

```
$ pip install fabric
```

Fabric expects a *fabfile* named `fabfile.py` which defines all of the actions we can take. Let's create that now. Put the following in `fabfile.py` in your project's root directory.

```
from fabric.api import local
```

```
def prepare_deployment(branch_name):
    local('python manage.py test django_project')
    local('git add -p && git commit') # or local('hg add && hg commit')
```

This will run the tests and commit your changes, *but only if your tests pass*. At this point, a simple "pull" in your production area becomes your deployment. Lets add a bit more to actually deploy. Add this to your fabfile.py:

```
from fabric.api import lcd, local

def deploy():
    with lcd('/path/to/my/prod/area/'):

        # With git...
        local('git pull /my/path/to/dev/area/')

        # With Mercurial...
        local('hg pull /my/path/to/dev/area/')
        local('hg update')

        # With both
        local('python manage.py migrate myapp')
        local('python manage.py test myapp')
        local('/my/command/to/restart/webserver')
```

This will pull your changes from the development master branch, run any migrations you've made, run your tests, and restart your web server. All in one simple command from the command line. If one of those steps fails, the script stops and reports what happened. Once you fix the issue, there is no need to run the steps manually. Since they're idempotent, you can simply rerun the deploy command and all will be well.

Note that the code above assumes you're developing on the same machine you deploy on. If that's not the case, the file would be mostly the same but would use Fabric's `run` function instead of `local`. See the Fabric documentation (http://docs.fabfile.org/) for details.

So now that we have our `fabfile.py` created, how do we actually deploy? Simple. Just run:

```
$ fab prepare_deployment
$ fab deploy
```

Technically, these could be combined into a single command, but I find it's better to explicitly prepare your deployment and then deploy as it makes you focus a bit more on what you're doing.

Setting Up Unit Tests

If you know anything about me, you probably know I'm crazy about automated tests. Too many Django projects are written without *any* tests whatsoever. This is another one of those things that costs a bit of time up-front but pays *enormous* dividends down the road. If you've ever found yourself debugging your app using `print` statements, having proper tests in place could have saved you a lot of time.

For Django, the Python `unittest` module is perfectly sufficient. The following is a minimal example of tests for a single app:

```
import datetime

from django.test import TestCase
from myapp.models import Post

class BlogPostTestCase(TestCase):
    def setUp(self):
        Post.objects.create(id=1,
            title='Starting a Django 1.6 Project the Right Way',
            date=datetime.datetime.now(),
            category='Django')
        Post.objects.create(id=2,
            title='Python\'s Hardest Problem',
            date=datetime.datetime.now(),
            category='Python')

    def test_posts_have_category(self):
        """Animals that can speak are correctly identified"""
        first_post = Post.objects.get(id=1)
        second_post = Post.objects.get(id=2)
        self.assertEqual(first_post.category, 'Django')
        self.assertEqual(second_post.category, 'Python')
```

You would put this code in a file called `test_<appname>.py` and place it in the same directory as the app it is testing. To run the tests for an app, simply run `./manage.py test <appname>`. The fabfile we created already knows to run the tests before deployment, so no need to make any other changes.

Enjoy Your New Django Application

That's it! You're ready to start your actual development. Now is when the real fun begins. Just remember: commit often, test everything, and don't write code where you serve it from. Regardless of what happens from here on out, you've definitely started a Django 1.6 project the right way!

Posted on Dec 18, 2013 by Jeff Knupp

« Improve Your Python: The Seminar (/blog/2013/12/11/improve-your-python-the-seminar)

Like this article?

Why not sign up for **Python Tutoring**? Sessions can be held remotely using Google+/Skype or in-person if you're in the NYC area. Email jeff@jeffknupp.com (mailto:jeff@jeffknupp.com) if interested.

Sign up for the free jeffknupp.com email newsletter. Sent roughly once a month, it focuses on Python programming, scalable web development, and growing your freelance consultancy. And of course, you'll never be spammed, your privacy is protected, and you can opt out at any time.

Email Address

Subscribe

44 Comments jeffknupp.com Login

Sort by Best Share Favorite

Join the discussion…

Stephan • 5 months ago

Great post. I think the version control strategy with dev and master branch could be explained a little bit more in detail in the later deployment section with fabric. for example, why is it in fabric's domain to do the committing (if I commit the branch to a clean state before running fab prepare_deployment git exits with an error code because there are no files to commit.), when and where occurs merging of the branches ...

For a current project I mixed this articles approach with the three-tier-layout and multiple settings-files-suggestions proposed in „Two Scooops of Django“ and have the feeling that this leads to a perfectly clean setup.

One more question: are there agreed upon best practices for media-directory content under version control? I often think about this and would be interested in different opinions.

5 △ | ▽ • Reply • Share ›

chuphay • 5 months ago

newb here. Had a really hard time with this step (actually more than a hard time, I found it impossible):

After it's installed, add the following lines to your shell's start-up file (.zshrc, .bashrc, .profile, etc).

export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/directory-you-do-development-in
source /usr/local/bin/virtualenvwrapper.sh

4 △ | ▽ • Reply • Share ›

Tom > chuphay • 5 months ago

Assuming you are using Linux or Mac, in your home directory there should be a file with one of those names (try "ls -lah" to list all files including hidden files). If you paste those lines into it, the next time you open a shell they will run automatically, saving you some time. It's a little different on Windows; you'll want to set the first two up as environment variables and virtualenvwrapper's install process should take care of the third line for you.

△ | ▽ • Reply • Share ›

michelleglauser > Tom • 3 months ago

I'm glad that chuphay left this comment because I'm having the same problem. But now I have a follow-up question. If I have all three of those files in my home directory, how do I know which one to add to?

△ | ▽ • Reply • Share ›

Tom > michelleglauser • 3 months ago

Try running `echo $SHELL` in your terminal to see which terminal you're using. In my case, I'm using bash and the .bash_profile is loading .profile so

 (//www.twitter.com/jeffknupp)  (//plus.google.com/+JeffKnupp) 

(//www.github.com/jeffknupp)  (//www.gittip.com/Jeff%20Knupp)

 (http://clicky.com/66535137)