django-skel Documentation

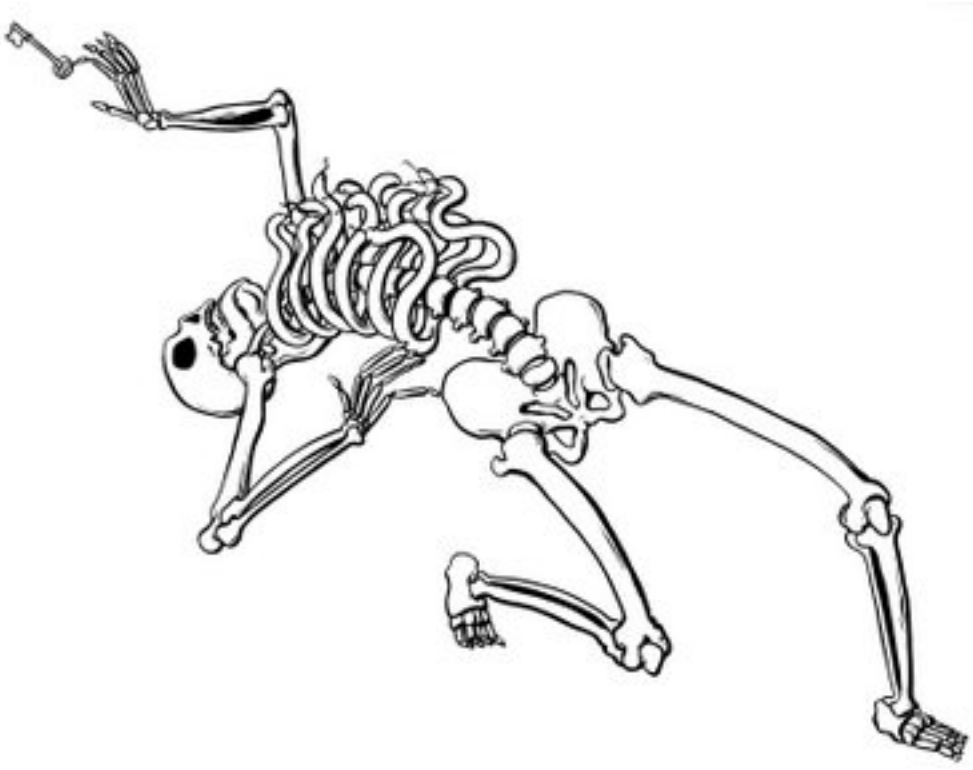
Release 1.5

Randall Degges

January 30, 2014

Contents

A modern Django 1.5 project skeleton.



Django is a great framework. Unfortunately, like any framework, it is only as useful as the tools you use with it. This is where `django-skel` *really* shines.

`django-skel` gives you a great project skeleton, complete with:

- Database migrations via South.
- Static file management via django-compressor.
- Task queueing via Celery.
- Helper utilities for working on the command line, via Fabric.
- Fancy documentation generation via Sphinx.
- Awesome local debugging and analysis via django-debug-toolbar.
- Amazon S3 integration (for publishing static assets: css, js, images, etc.) via django-storages.
- CSS compression (for production environments) via cssmin.
- JS compression (for production environments) via jsmin.
- Memcache caching support via django-heroku-memcacheify.
- PostgreSQL support via django-heroku-postgresify.
- A blazing fast WSGI server for serving production traffic via gunicorn and gevent.
- Production application performance monitoring and usage statistics via newrelic.
- All the best practices I've come to learn with more than 4 years of Django experience.
- Built in support for production deployments on Heroku's platform.

But, more importantly, `django-skel` gives you a really clean, simple, and reliable project template for developers of any experience level.

If you want a best practices approach to Django, use `django-skel` and you won't be disappointed!

CHAPTER 1

Follow the Guide Below to Victory!

1.1 Prerequisites

Before we go any further, I'm going to assume that you've got the following things:

- You're running some flavor of Linux or Mac (untested) as your desktop OS.
- You've got Django 1.5 installed somewhere (inside a virtualenv, preferably).
- You have an Amazon Web Services account. This is required if you want to use our production deployment tools. If you don't want to run your code in production, don't worry about it.
- You have a Heroku account. Heroku is the best python web host on the internet. If you'd like to deploy your site to production, having an account there will be extremely useful.
- You have the Heroku toolbelt installed. This only applies to you if you plan on deploying your stuff to production (same as the above).

1.2 Getting Started

So you're ready to start your next Django project! Let me be your guide. Over the next few minutes we'll be taking a magical journey together >:)

1.2.1 Creating a New Project

To create your new project, run the following command, substituting `woot` for whatever you'd like to name your new project:

```
$ django-admin.py startproject --template=https://github.com/rdegges/django-skel/zipball/master woot
$ cd woot
$ ls
docs/  fabfile.py  gunicorn.py.ini  manage.py  Procfile  README.md  reqs/ requirements.txt  woot/  ws
```

The next thing you'll probably want to do is remove my project docs:

```
$ rm -rf docs README.md
```

That way you don't get the documentation you're reading right now in your new project.

Next, create your first django app for this project:

```
$ mkdir woot/apps/myapp
$ django-admin.py startapp myapp woot/apps/myapp
```

Lastly, create a Git repository for your new project, and commit everything:

```
$ git init
Initialized empty Git repository in /home/rdegges/Code/ex/woot/.git/
$ git add .; git commit -m 'First commit using django-skel!'
...
```

Easy, right?!

1.2.2 Install All the Dependencies!

Before I start writing code, I like to setup a virtualenv for myself–this allows me to install all my project dependencies in a local installation, as opposed to installing all of them globally on my box.

To install the local dependencies (that you'll need to run your site locally), run the following command:

```
$ pip install -r reqs/dev.txt
...
```

Note: If the pip command above fails, it means you're missing some C libraries that are required for some of the Python libraries to work. The ones you need (on Ubuntu) are:

- libevent-dev
- libpq-dev
- libmemcached-dev
- zlib1g-dev
- libssl-dev
- python-dev
- build-essential

I also recommend you install `postgresql-client`, even though it isn't required.

Bam!

1.2.3 Running Your Site Locally

Before you start coding, let's bootstrap our SQLite database (for local development), and test our the Django admin panel just to make sure everything's working:

```
$ python manage.py syncdb
...
$ python manage.py migrate
...
$ python manage.py runserver
...
```

Assuming everything's working, you should now be able to visit http://localhost:8000/admin/ in your web browser, and log in.

The `syncdb` command here just initializes our database, and the `migrate` command applies our South migrations.

From now on, whenever you want to run your site locally for testing, you can follow these standard Django conventions.

1.3 Layout

Before we move on, I'd like to give you a quick tour of `django-skel`'s file layout:

```
.
-- fabfile.py
-- gunicorn.py.ini
-- manage.py
-- Procfile
-- reqs
|   -- common.txt
|   -- dev.txt
|   -- prod.txt
-- requirements.txt
-- woot
|   -- apps
|   |   -- __init__.py
|   -- __init__.py
|   -- libs
|   |   -- __init__.py
|   -- settings
|   |   -- common.py
|   |   -- dev.py
|   |   -- __init__.py
|   |   -- prod.py
|   -- templates
|   |   -- 404.html
|   |   -- 500.html
|   -- urls.py
-- wsgi.py

6 directories, 19 files
```

`fabfile.py` is a utility script (written using Fabric) that adds some helpful shortcut commands. It can automatically bootstrap a Heroku app for you, and a number of other useful things. You can run `fab --list` from the command line to see its usage.

`gunicorn.py.ini` is our gunicorn web server configuration file. It is optimized for large scale sites, and should

work well in any environment.

`manage.py` is our default Django management script.

`Procfile` is our Heroku process file–which tells Heroku what our three types of services are: `web`, `scheduler`, and `worker`. To learn more about this, see Heroku's Procfile documentation.

`reqs` is a directory which contains all of our pip requirement files, broken into categories by the environment in which they're used. The `common.txt` file contains all of our 'shared' requirements, the `dev.txt` file contains all of our local development requirements, and the `prod.txt` file contains our production requirements. This modular approach is taken to make development as flexible (and intuitive) as possible.

`requirements.txt` is a Heroku specific file which tells Heroku to install our production requirements *only*.

`woot` is the base Django site. Everything inside this directory is considered your actual Django code.

`woot/apps` is a directory meant to hold all of your local Django applications. If you wanted to create a `blog` app, for instance, you'd put it here.

`woot/libs` is a directory meant to hold all of your local Django libraries–code which doesn't really fit into 'applications'. This usually includes stuff like templatetags that are used in various place, or other helpful utility functions.

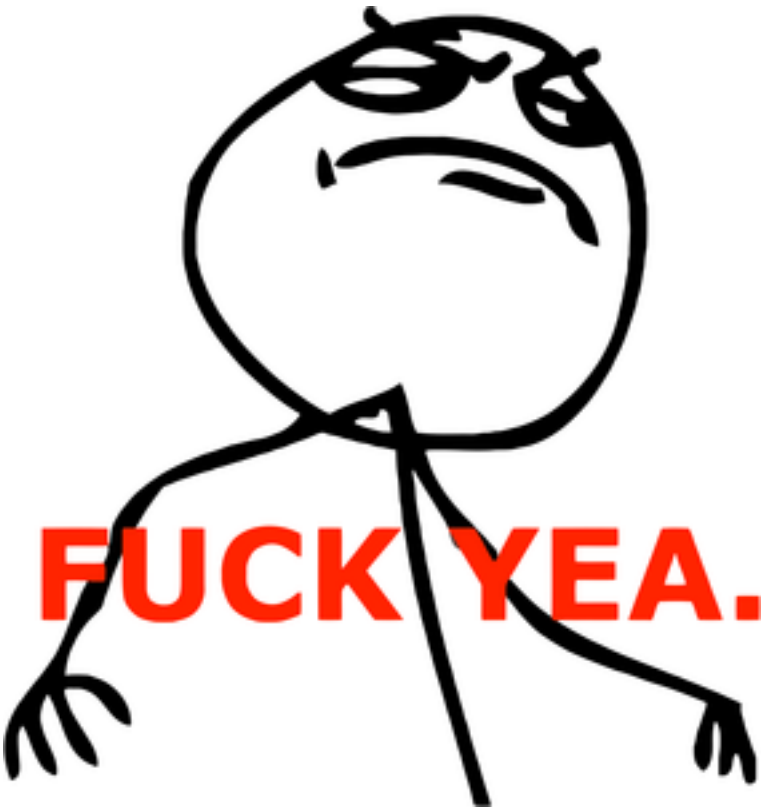
`woot/settings` is a directory which holds all of your Django settings files! Much like our pip requirements, there is a settings file for each environment: `dev.py`, `prod.py`, and `common.py` (shared settings). Feel free to edit and tweak these to your specific needs.

`woot/templates` is a directory that holds all your Django templates. By default, we only include a 404.html and 500.html, since those are used in all Django projects.

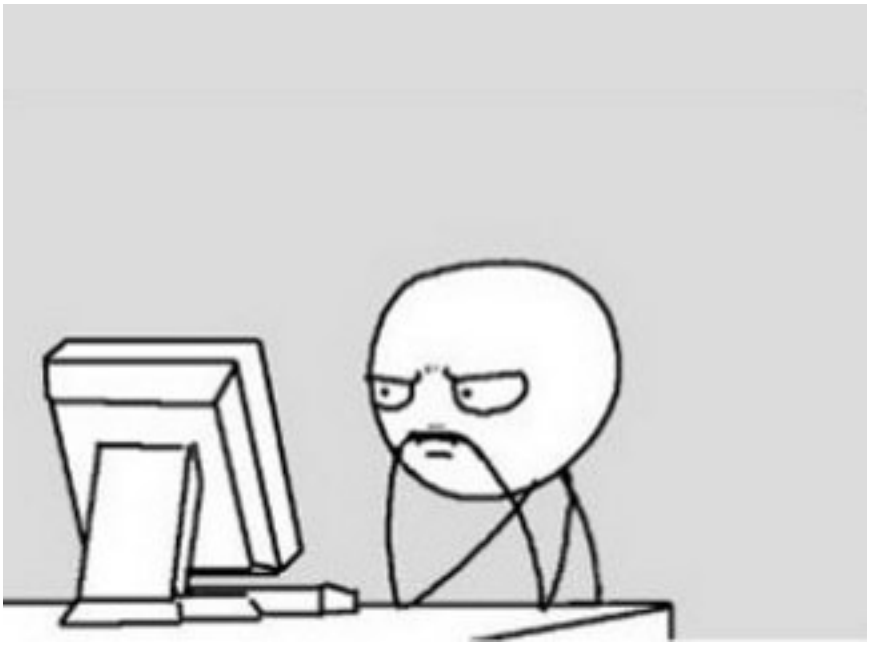
`woot/urls.py` is your standard Django urlconf.

`wsgi.py` is your standard Django wsgi configuration file. Our webserver uses this to figure things out :)

As you can see–everything is very straightforward. All standard Django knowledge you have should easily apply to `django-skel`!



1.4 Developing



Now that you've got your project running locally and the basics covered, let's talk about development.

`django-skel` is optimized for a simple workflow:

- Develop code locally on your laptop using SQLite.
- Run your production code remotely on Heroku.
- Upload your static files (css, javascript, images, etc.) to Amazon S3 so that they are served extremely fast to your end-users.
- Compress all your static files (css and javascript) so that end-users can download them quicker. This also helps prevent copycats from copy+pasting your code, since minified code is much more difficult to reverse engineer.

I've found that using this workflow is the most effective way for me to write code. If your needs differ from mine, you can easily tweak `django-skel`'s settings by editing the files found in `project_name/settings` to your liking. All the options you'll find there are documented, and easy to understand.

With that said, let's discuss development!

1.4.1 Managing Your Settings

Managing your settings using `django-skel` is simple. Follow the rules below, and you can't go wrong:

1. Place all your 'common' settings in `settings/common.py`. This includes stuff like: Django apps you need to use in all environmnets (development, production, etc.), global variables, etc.
2. Place all your development-specific settings in `settings/dev.py`. 'Nuff said.
3. Place all your production-specific settings in `settings/prod.py`.
4. If you're confused, follow the documentation links! I've heavily documented the settings files, and included reference links to all the relevant documentation. If you've got a question, or are confused about something, consult the docs first!

1.4.2 Storing Static Assets

Always place your static assets (images, javascript, css, etc.) into a sub-directory of your project folder called `assets`. If your project is named `woot`, for instance, then you should place all your static files inside of `woot/assets`.

The way I like to organize this is by doing something like:

```
$ mkdir woot/assets
$ cd woot/assets
$ mkdir {css,js,img}
```

Then I'll place all my css files in `woot/assets/css`, my js files in `woot/assets/js`, and my images into `woot/assets/img`.

This way, you've got a clear directory hierarchy, and anyone else that looks at your code will immediately recognize what's going on.

1.4.3 CSS Best Practices

One really great feature of `django-skel` is that it's already optimized for handling CSS files in the most optimial way possible. What this means for you, as a developer, is that if you're planning on writing / using CSS in your Django project, you should keep the following in mind.

When you include a CSS file in your HTML, it normally looks something like this:

```
<html>
  <head>
    <link rel="stylesheet" href="{{ STATIC_URL }}css/style.css" />
  </head>
</html>
```

That's great and all, but by doing things that way you'll miss out on a powerful feature: CSS templating. Wouldn't it be nice if you could use `{{ STATIC_URL }}` *inside* of your CSS files as well? That way you could write nifty rules like:

```
body {
  background: url({{ STATIC_URL }}img/omgyea.png);
}
```

The above code snippet is great because it will work in both local development mode (by having Django serve your image locally), as well as production mode (by having Amazon S3 serve your image through its CDN). To make use of this awesome functionality, all you have to do is modify your HTML template like so:

```
{% load compress %}
<html>
  <head>
    {% compress css %}
      <link rel="stylesheet" href="{{ STATIC_URL }}css/style.css" />
    {% endcompress %}
  </head>
</html>
```

Using django-compressor you get this functionality out of the box! Behind the scenes, django-compressor will run your CSS files through the Django templating engine, which allows you do the cool stuff mentioned above.

As an added benefit, in production mode, it will also minify your CSS files for you (removing whitespace to save space). But more on that later!

1.4.4 Javascript Best Practices

Much like CSS best practices, `django-skel` is optimized for handling Javascript code in the same way that it does for CSS (see the previous section for details).

To make use of both the Django templating engine (so that you can use stuff like `{{ STATIC_URL }}` in your Javascript code) as well as Javascript minification and obfuscation, change your HTML templates from this:

```
<html>
  <head>
    <script src="{{ STATIC_URL }}js/script.js" type="text/javascript"></script>
  </head>
</html>
```

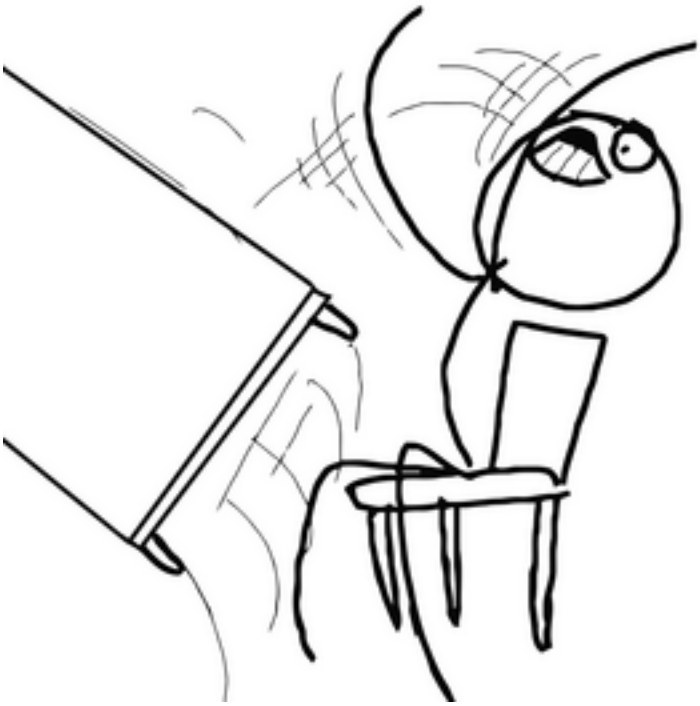
To this:

```
{% load compress %}
<html>
  <head>
    {% compress js %}
      <script src="{{ STATIC_URL }}js/script.js" type="text/javascript"></script>
    {% endcompress %}
  </head>
</html>
```

And that's all there is to it!

1.5 Running on Heroku

Normally, deploying a Django site would make you want to flip your desk:



Luckily for us, Heroku has made the process a complete joy! If you're aren't familiar with Heroku–they are the **best** web host, and you will love them if you don't already.

`django-skel` ships with a production ready Heroku configuration module, and this section will walk you through creating your Heroku app, and getting your site running in production.

While this section is quite long, don't be intimidated! It's only long because I'm explaining everything along the way–the reality of it is that deploying your site this way really only consists of a couple commands.

If you'd like to read some official documentation on the topic, check out Heroku's Django documentation.

1.5.1 Step 1 - Create Your Heroku Application

The first step in getting your site running on Heroku is, as I'm sure you've guessed, to create a Heroku app! Let's do it now:

```
$ heroku create [your_app_name_here]
```

If you don't specify an app name, one will be automatically assigned to you. I like to name my apps explicitly, because I have a bunch of them, and it's a lot easier to track.

The next thing you'll need to do is push your project code to Heroku. When you ran the `heroku create` command above, the `heroku` command added a new Git remote to your project. To push your code to Heroku, all you do is push to the `heroku` remote:

```
$ git push heroku master
```

That will 'deploy' your code straight to Heroku! From now on, whenever you want to deploy your code, just run this command.

1.5.2 Step 2 - Install the Addons

Now that you've got your Heroku application going, let's install some Heroku Addons. Heroku is a modular system. The core of Heroku allows you to run your code, but doesn't provide any extra infrastructure services.

To get things like PostgreSQL, memcache, RabbitMQ, etc.–you need to install Heroku addons to do what you want.

Let's install our required addons now–these addons are all free (you can upgrade them at any time in the future). `django-skel` already supports all of these, and requires most of them to function:

```
$ heroku addons:add cloudamqp:lemur
$ heroku addons:add heroku-postgresql:dev
$ heroku addons:add scheduler:standard
$ heroku addons:add memcachier:dev
$ heroku addons:add newrelic:standard
$ heroku addons:add pgbackups:auto-month
$ heroku addons:add sentry:developer
```

cloudamqp is a hosted RabbitMQ service. This is what makes our task queueing (via Celery) possible.

heroku-postgresql is a hosted PostgreSQL service that kicks ass.

scheduler is a cron replacement.

memcachier is a hosted memcache service.

newrelic is the best application monitoring tool ever created.

pgbackups is an excellent PostgreSQL backup tool that stores backups automatically to S3, and lets you download and manage your backups easily.

sentry is a pretty neat error aggregation and searching tool that makes debugging issues simple.

Just for the record, if you'd like to upgrade any of these free addons, you can do so by running the `heroku addons:upgrade` command. For example–to switch from the free newrelic addon to their paid addon which has lots more features, you can simply run:

```
$ heroku addons:upgrade newrelic:professional
```

Bam!

The last thing you'll need to do is specify a default PostgreSQL database (`django-skel` requires this). To do this, run:

```
$ heroku pg:info
```

And you should see a database name, something like `HEROKU_POSTGRESQL_NAVY`. Once you've got that name, run:

```
$ heroku pg:promote HEROKU_POSTGRESQL_NAVY
```

To set your database as the default.

1.5.3 Step 3 - Configure the Environment

Heroku operates via environment variables. This is the preferred place to store all those secret things (passwords, API keys, etc.) that you don't want lurking around your version control system.

`django-skel` requires several environment variables be set. To set these variables, run the following commands:

```
# Your AWS security credentials:
$ heroku config:add AWS_ACCESS_KEY_ID=xxx
$ heroku config:add AWS_SECRET_ACCESS_KEY=xxx
$ heroku config:add AWS_STORAGE_BUCKET_NAME=xxx

# Replace 'woot' with the name of your project:
$ heroku config:add DJANGO_SETTINGS_MODULE=woot.settings.prod

# A random long (40 characters or so) string:
$ heroku config:add SECRET_KEY=xxx
```

Note: Not sure what to use for your `SECRET_KEY` setting? You can always do something like:

```
from random import choice
print ''.join([choice('abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)') for i in range(50)])
```

And copy the resulting string for usage :)

If you'd like to, you can also enable email support out of the box by setting the optional email environment variables as well:

```
$ heroku config:add EMAIL_HOST=xxx
$ heroku config:add EMAIL_HOST_PASSWORD=xxx
$ heroku config:add EMAIL_HOST_USER=xxx
$ heroku config:add EMAIL_PORT=xxx
```

Note: `EMAIL_HOST` and `EMAIL_PORT` will default to the proper settings for Google apps, so if you're using that–feel free to leave those out.

1.5.4 Step 4 - Spin It Up!

Now that everything is configured and ready to go, let's spin up our backend!

Instead of spinning up 'servers', Heroku allows us to spin up 'dynos', which are essentially locked-down virtual server instances. The `Procfile` defined at the root of your `django-skel` project defines our three service types:

- `web` - The service that runs our Django application behind gunicorn.
- `scheduler` - The service that runs a Celery worker and the Celerybeat daemon.
- `worker` - The service that runs a Celery worker **only**.

To spin up a web dyno, run: `heroku scale web=1`. You can confirm that everything is working by running `heroku ps` afterwards. That will run a single web dyno.

If you'd like run a Celery worker, run: `heroku scale scheduler=1`. If you need more than one worker, you can add additional power by spinning up the `worker` dynos, via `heroku scale worker=1`.

Note: No matter what, never **EVER** spin up more than one `scheduler`. The scheduler process runs Celerybeat, which schedules background tasks. Having more than one scheduler running can cause serious duplicate task problems. Instead, you should always have one `scheduler` running, and as many `worker` instances as you need.

Need to add more web servers? No problem:

```
$ heroku scale web=100
```

Need to add more workers? No problem:

```
$ heroku scale worker=100
```

Need to check and see how many dynos you have running? Easy:

```
$ heroku ps
```

1.5.5 Step 5 - Deploy Your Static Assets

The last step in successfully deploying your production Django application is to compress and then upload all your static assets to Amazon S3 (css, js, images, etc.).

To do this, simply run the following commands:

```
$ heroku run python manage.py collectstatic --noinput
$ heroku run python manage.py compress
```

And that's it!

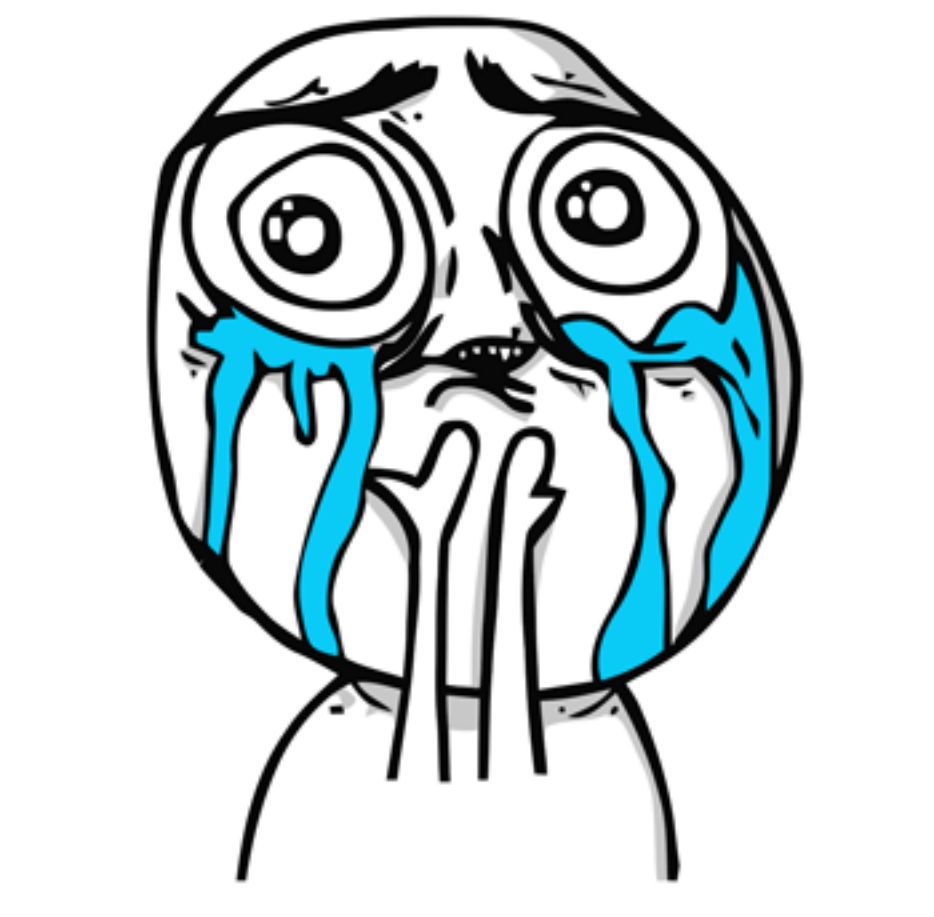
1.5.6 Extra Reading

You are now running a best practices Django website, on top of Heroku, using Amazon S3 to serve your static content!

If you'd like to learn more about Heroku, scaling, and stuff like that, you should probably check out my blog because I write about this stuff all the time >:)

Oh, and also, read Heroku's documentation :)

Now... Go and be happy!



CHAPTER 2

Also...

Need help? Got a question? Want to post random pointless comments? Head over to our GitHub issue tracker and leave a message!

Wanna just hang out with some other bad-ass hackers like yourself? Say hi on #heapify, or you could follow me on twitter.