cookiecutter-django Documentation

Release 2016.51.1

cookiecutter-django

December 19, 2016

Contents

A Cookiecutter template for Django.

Contents:

CHAPTER 1

Project Generation Options

project_name [project_name]: Your human-readable project name, including any capitalization or spaces.

project_slug [project_name]: The slug of your project, without dashes or spaces. Used to name your repo and in other places where a Python-importable version of your project name is needed.

author_name [Your Name]: You! This goes into places like the LICENSE file.

email [Your email]: Your email address.

description [A short description of the project.] Used in the generated README.rst and other places.

domain_name [example.com] Whatever domain name you plan to use for your project when it goes live.

version [0.1.0] The starting version number for your project.

timezone [UTC] Used in the common settings file for the *TIME_ZONE* value.

use_whitenoise [y] Whether to use WhiteNoise for static file serving.

use_celery [n] Whether to use Celery. This gives you the ability to use distributed task queues in your project.

use_mailhog [n] Whether to use MailHog. MailHog is a tool that simulates email receiving for development purposes. It runs a simple SMTP server which catches any message sent to it. Messages are displayed in a web interface which runs at `http://localhost:8025/` You need to download the MailHog executable for your operating system, see the 'Developing Locally' docs for instructions.

use_sentry_for_error_reporting [n] Whether to use Sentry to log errors from your project.

use_opbeat [n] Whether to use Opbeat for preformance monitoring and code optimization.

use_pycharm [n] Adds support for developing in PyCharm with a preconfigured .idea directory.

windows [n] Whether you'll be developing on Windows.

use_python3 [y] By default, the Python code generated will be for Python 3.x. But if you answer *n* here, it will be legacy Python 2.7 code.

use_docker [y] Whether to use Docker, separating the app and database into separate containers.

use_heroku [n] Add configuration to deploy the application to a Heroku instance.

use_compressor [n] Use Django Compressor to minify and combine rendered JavaScript and CSS into cachable static resources.

js_task_runner [1] Select a JavaScript task runner. The choices are:

1. Gulp
2. Grunt

3. Webpack
4. None

use_lets_encrypt [n] Use Let's Encrypt as the certificate authority for this project.

open_source_license [1] Select a software license for the project. The choices are:

1. MIT
2. BSD
3. GPLv3
4. Apache Software License 2.0
5. Not open source

NOTE: *If you choose to use Docker, selecting a JavaScript task runner is not supported out of the box.*

CHAPTER 2

Getting Up and Running Locally

The steps below will get you up and running with a local development environment. We assume you have the following installed:

- pip
- virtualenv
- PostgreSQL

First make sure to create and activate a virtualenv.

Then install the requirements for your local development:

```
$ pip install -r requirements/local.txt
```

Then, create a PostgreSQL database with the following command, where *[project_slug]* is what value you entered for your project's *project_slug*:

```
$ createdb [project_slug]
```

You can now run the usual Django `migrate` and `runserver` commands:

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

At this point you can take a break from setup and start getting to know the files in the project.

But if you want to go further with setup, read on.

(Note: the following sections still need to be revised)

2.1 Setting Up Env Vars for Production

Cookiecutter Django uses the excellent django-environ package, which includes a `DATABASE_URL` environment variable to simplify database configuration in your Django settings.

Rename env.example to .env to begin updating the file with your own environment variables. To add your database, define `DATABASE_URL` and add it to the .env file, as shown below:

`DATABASE_URL=''postgres://<pg_user_name>:<pg_user_password>@127.0.0.1:<pg_port>/<pg_databa`

2.2 Setup your email backend

django-allauth sends an email to verify users (and superusers) after signup and login (if they are still not verified). To send email you need to configure your email backend

In development you can (optionally) use MailHog for email testing. MailHog is built with Go so there are no dependencies. To use MailHog:

1. Download the latest release for your operating system
2. Rename the executable to `mailhog` and copy it to the root of your project directory
3. Make sure it is executable (e.g. `chmod +x mailhog`)
4. Execute mailhog from the root of your project in a new terminal window (e.g. `./mailhog`)
5. All emails generated from your django app can be seen on http://127.0.0.1:8025/

Alternatively simply output emails to the console via: `EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'`

In production basic email configuration is setup to send emails with Mailgun

Live reloading and Sass CSS compilation

If you'd like to take advantage of live reloading and Sass / Compass CSS compilation you can do so with a little bit of prep work.

CHAPTER 3

Getting Up and Running Locally With Docker

The steps below will get you up and running with a local development environment. All of these commands assume you are in the root of your generated project.

3.1 Prerequisites

You'll need at least Docker 1.10.

If you don't already have it installed, follow the instructions for your OS:

- On Mac OS X, you'll need Docker for Mac
- On Windows, you'll need Docker for Windows
- On Linux, you'll need docker-engine

3.2 Attention Windows users

Currently PostgreSQL (`psycopg2` python package) is not installed inside Docker containers for Windows users, while it is required by the generated Django project. To fix this, add `psycopg2` to the list of requirements inside `requirements/base.txt`:

```
# Python-PostgreSQL Database Adapter
psycopg2==2.6.2
```

Doing this will prevent the project from being installed in an Windows-only environment (thus without usage of Docker). If you want to use this project without Docker, make sure to remove `psycopg2` from the requirements again.

3.3 Build the Stack

This can take a while, especially the first time you run this particular command on your development system:

```
$ docker-compose -f dev.yml build
```

If you want to build the production environment you don't have to pass an argument -f, it will automatically use docker-compose.yml.

3.4 Boot the System

This brings up both Django and PostgreSQL.

The first time it is run it might take a while to get started, but subsequent runs will occur quickly.

Open a terminal at the project root and run the following for local development:

```
$ docker-compose -f dev.yml up
```

You can also set the environment variable `COMPOSE_FILE` pointing to `dev.yml` like this:

```
$ export COMPOSE_FILE=dev.yml
```

And then run:

```
$ docker-compose up
```

3.4.1 Running management commands

As with any shell command that we wish to run in our container, this is done using the `docker-compose run` command.

To migrate your app and to create a superuser, run:

```
$ docker-compose -f dev.yml run django python manage.py migrate
$ docker-compose -f dev.yml run django python manage.py createsuperuser
```

Here we specify the `django` container as the location to run our management commands.

3.4.2 Production Mode

Instead of using *dev.yml*, you would use *docker-compose.yml*.

3.5 Other Useful Tips

3.5.1 Make a machine the active unit

This tells our computer that all future commands are specifically for the dev1 machine. Using the `eval` command we can switch machines as needed.

```
$ eval "$(docker-machine env dev1)"
```

3.5.2 Detached Mode

If you want to run the stack in detached mode (in the background), use the `-d` argument:

```
$ docker-compose -f dev.yml up -d
```

3.5.3 Debugging

ipdb

If you are using the following within your code to debug:

```
import ipdb; ipdb.set_trace()
```

Then you may need to run the following for it to work as desired:

```
$ docker-compose run -f dev.yml --service-ports django
```

django-debug-toolbar

In order for django-debug-toolbar to work with docker you need to add your docker-machine ip address (the output of **'Get the IP ADDRESS'_**) to INTERNAL_IPS in local.py

You may need to add the following to your css in order for the django-debug-toolbar to be visible (this applies whether Docker is being used or not):

```
/* Override Bootstrap 4 styling on Django Debug Toolbar */
#djDebug[hidden], #djDebug [hidden] {
    display: block !important;
}

#djDebug [hidden][style='display: none;'] {
    display: none !important;
}
```

3.5.4 Using the Mailhog Docker Container

In development you can (optionally) use MailHog for email testing. If you selected *use_docker*, MailHog is added as a Docker container. To use MailHog:

1. Make sure, that `mailhog` docker container is up and running
2. Open your browser and go to `http://127.0.0.1:8025`

CHAPTER 4

Settings

This project relies extensively on environment settings which **will not work with Apache/mod_wsgi setups**. It has been deployed successfully with both Gunicorn/Nginx and even uWSGI/Nginx.

For configuration purposes, the following table maps environment variables to their Django setting:

Environment Variable	Django Setting	Development Default	Production Default
DJANGO_ADMIN_URL	n/a	r'^admin/'	raises error
DJANGO_CACHES	CACHES (default)	locmem	redis
DJANGO_DATABASES	DATABASES (default)	See code	See code
DJANGO_DEBUG	DEBUG	True	False
DJANGO_SECRET_KEY	SECRET_KEY	CHANGEME!!!	raises error
DJANGO_SECURE_BROWSER_XSS_FILTER	SECURE_BROWSER_XSS_FILTER	n/a	True
DJANGO_SECURE_SSL_REDIRECT	SECURE_SSL_REDIRECT	n/a	True
DJANGO_SECURE_CONTENT_TYPE_NOSNIFF	SECURE_CONTENT_TYPE_NOSNIFF	n/a	True
DJANGO_SECURE_FRAME_DENY	SECURE_FRAME_DENY	n/a	True
DJANGO_SECURE_HSTS_INCLUDE_SUBDOMAINS	SECURE_HSTS_INCLUDE_SUBDOMAINS	n/a	True
DJANGO_SESSION_COOKIE_HTTPONLY	SESSION_COOKIE_HTTPONLY	n/a	True
DJANGO_SESSION_COOKIE_SECURE	SESSION_COOKIE_SECURE	n/a	False
DJANGO_DEFAULT_FROM_EMAIL	DEFAULT_FROM_EMAIL	n/a	"your_project_name <noreply@your_domain_name>"
DJANGO_SERVER_EMAIL	SERVER_EMAIL	n/a	"your_project_name <noreply@your_domain_name>"
DJANGO_EMAIL_SUBJECT_PREFIX	EMAIL_SUBJECT_PREFIX	n/a	"[your_project_name] "
DJANGO_ALLOWED_HOSTS	ALLOWED_HOSTS	['*']	['your_domain_name']

The following table lists settings and their defaults for third-party applications, which may or may not be part of your project:

Environment Variable	Django Setting	Development Default	Production Default
DJANGO_AWS_ACCESS_KEY_ID	AWS_ACCESS_KEY_ID	n/a	raises error
DJANGO_AWS_SECRET_ACCESS_KEY	AWS_SECRET_ACCESS_KEY	n/a	raises error
DJANGO_AWS_STORAGE_BUCKET_NAME	AWS_STORAGE_BUCKET_NAME	n/a	raises error
DJANGO_SENTRY_DSN	SENTRY_DSN	n/a	raises error
DJANGO_SENTRY_CLIENT	SENTRY_CLIENT	n/a	raven.contrib.django.raven_compat.DjangoClient
DJANGO_SENTRY_LOG_LEVEL	SENTRY_LOG_LEVEL	n/a	logging.INFO
DJANGO_MAILGUN_API_KEY	MAILGUN_ACCESS_KEY	n/a	raises error
DJANGO_MAILGUN_SERVER_NAME	MAILGUN_SERVER_NAME	n/a	raises error
MAILGUN_SENDER_DOMAIN	MAILGUN_SENDER_DOMAIN	n/a	raises error
NEW_RELIC_APP_NAME	NEW_RELIC_APP_NAME	n/a	raises error
NEW_RELIC_LICENSE_KEY	NEW_RELIC_LICENSE_KEY	n/a	raises error
DJANGO_OPBEAT_APP_ID	OPBEAT['APP_ID']	n/a	raises error
DJANGO_OPBEAT_SECRET_TOKEN	OPBEAT['SECRET_TOKEN']	n/a	raises error
DJANGO_OPBEAT_ORGANIZATION_ID	OPBEAT['ORGANIZATION_ID']	n/a	raises error

4.1 Other Environment Settings

DJANGO_ACCOUNT_ALLOW_REGISTRATION (=True) Allow enable or disable user registration through *django-allauth* without disabling other characteristics like authentication and account management. (Django Setting: ACCOUNT_ALLOW_REGISTRATION)

CHAPTER 5

Linters

5.1 flake8

To run flake8:

```
$ flake8
```

The config for flake8 is located in setup.cfg. It specifies:

- Set max line length to 120 chars
- Exclude .tox,.git,*/migrations/,/static/CACHE/*,docs,node_modules

5.2 pylint

This is included in flake8's checks, but you can also run it separately to see a more detailed report:

```
$ pylint <python files that you wish to lint>
```

The config for pylint is located in .pylintrc. It specifies:

- Use the pylint_common and pylint_django plugins. If using Celery, also use pylint_celery.
- Set max line length to 120 chars
- Disable linting messages for missing docstring and invalid name
- max-parents=13

5.3 pep8

This is included in flake8's checks, but you can also run it separately to see a more detailed report:

```
$ pep8 <python files that you wish to lint>
```

The config for pep8 is located in setup.cfg. It specifies:

- Set max line length to 120 chars
- Exclude .tox,.git,*/migrations/,/static/CACHE/*,docs,node_modules

CHAPTER 6

Live reloading and Sass CSS compilation

If you'd like to take advantage of live reloading and Sass / Compass CSS compilation you can do so with a little bit of prep work.

Make sure that nodejs is installed. Then in the project root run:

```
$ npm install
```

If you don't already have it, install *compass* (doesn't hurt if you run this command twice):

```
gem install compass
```

Now you just need:

```
$ grunt serve
```

The base app will now run as it would with the usual `manage.py runserver` but with live reloading and Sass compilation enabled.

To get live reloading to work you'll probably need to install an appropriate browser extension

CHAPTER 7

Deployment on PythonAnywhere

7.1 Overview

Full instructions follow, but here's a high-level view.

First time config:

1. Pull your code down to PythonAnywhere using a *Bash console* and setup a virtualenv
2. Set your config variables in the *postactivate* script
3. Run the *manage.py* `migrate` and `collectstatic` commands
4. Add an entry to the PythonAnywhere *Web tab*
5. Set your config variables in the PythonAnywhere *WSGI config file*

Once you've been through this one-off config, future deployments are much simpler: just `git pull` and then hit the "Reload" button :)

7.2 Getting your code and dependencies installed on PythonAnywhere

Make sure your project is fully commited and pushed up to Bitbucket or Github or wherever it may be. Then, log into your PythonAnywhere account, open up a **Bash** console, clone your repo, and create a virtualenv:

```
git clone <my-repo-url>  # you can also use hg
cd my-project-name
mkvirtualenv --python=/usr/bin/python3.5 my-project-name # or python2.7, etc
pip install -r requirements/production.txt  # may take a few minutes
```

7.3 Setting environment variables in the console

Generate a secret key for yourself, eg like this:

```
python -c 'import random; print("".join(random.SystemRandom().choice("abcdefghijklmnopqrstuvwxyz0123
```

Make a note of it, since we'll need it here in the console and later on in the web app config tab.

Set environment variables via the virtualenv "postactivate" script (this will set them every time you use the virtualenv in a console):

```
vi $VIRTUAL_ENV/bin/postactivate
```

TIP: *If you don't like vi, you can also edit this file via the PythonAnywhere "Files" menu; look in the ".virtualenvs" folder.*

Add these exports

```
export DJANGO_SETTINGS_MODULE='config.settings.production'
export DJANGO_SECRET_KEY='<secret key goes here>'
export DJANGO_ALLOWED_HOSTS='<www.your-domain.com>'
export DJANGO_ADMIN_URL='<not admin/>'
export DJANGO_MAILGUN_API_KEY='<mailgun key>'
export DJANGO_MAILGUN_SERVER_NAME='<mailgun server name>'
export MAILGUN_SENDER_DOMAIN='<mailgun sender domain (e.g. mg.yourdomain.com)>'
export DJANGO_AWS_ACCESS_KEY_ID=
export DJANGO_AWS_SECRET_ACCESS_KEY=
export DJANGO_AWS_STORAGE_BUCKET_NAME=
export DATABASE_URL='<see below>'
```

NOTE: *The AWS details are not required if you're using whitenoise or the built-in pythonanywhere static files service, but you do need to set them to blank, as above.*

7.4 Database setup:

Go to the PythonAnywhere **Databases tab** and configure your database.

- For Postgres, setup your superuser password, then open a Postgres console and run a *CREATE DATABASE my-db-name*. You should probably also set up a specific role and permissions for your app, rather than using the superuser credentials. Make a note of the address and port of your postgres server.
- For MySQL, set the password and create a database. More info here: https://help.pythonanywhere.com/pages/UsingMySQL
- You can also use sqlite if you like! Not recommended for anything beyond toy projects though.

Now go back to the *postactivate* script and set the `DATABASE_URL` environment variable:

```
export DATABASE_URL='postgres://<postgres-username>:<postgres-password>@<postgres-address>:<postgres-
# or
export DATABASE_URL='mysql://<pythonanywhere-username>:<mysql-password>@<mysql-address>/<database-nar
# or
export DATABASE_URL='sqlite:////home/yourusername/path/to/db.sqlite'
```

If you're using MySQL, you may need to run `pip install mysqlclient`, and maybe add `mysqlclient` to *requirements/production.txt* too.

Now run the migration, and collectstatic:

```
source $VIRTUAL_ENV/bin/postactivate
python manage.py migrate
python manage.py collectstatic
# and, optionally
python manage.py createsuperuser
```

7.5 Configure the PythonAnywhere Web Tab

Go to the PythonAnywhere **Web tab**, hit **Add new web app**, and choose **Manual Config**, and then the version of Python you used for your virtualenv.

NOTE: *If you're using a custom domain (not on *.pythonanywhere.com), then you'll need to set up a CNAME with your domain registrar.*

When you're redirected back to the web app config screen, set the **path to your virtualenv**. If you used virtualenvwrapper as above, you can just enter its name.

Click through to the **WSGI configuration file** link (near the top) and edit the wsgi file. Make it look something like this, repeating the environment variables you used earlier:

```
import os
import sys
path = '/home/<your-username>/<your-project-directory>'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'config.settings.production'
os.environ['DJANGO_SECRET_KEY'] = '<as above>'
os.environ['DJANGO_ALLOWED_HOSTS'] = '<as above>'
os.environ['DJANGO_ADMIN_URL'] = '<as above>'
os.environ['DJANGO_MAILGUN_API_KEY'] = '<as above>'
os.environ['DJANGO_MAILGUN_SERVER_NAME'] = '<as above>'
os.environ['DJANGO_AWS_ACCESS_KEY_ID'] = ''
os.environ['DJANGO_AWS_SECRET_ACCESS_KEY'] = ''
os.environ['DJANGO_AWS_STORAGE_BUCKET_NAME'] = ''
os.environ['DATABASE_URL'] = '<as above>'

from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

Back on the Web tab, hit **Reload**, and your app should be live!

NOTE: *you may see security warnings until you set up your SSL certificates. If you want to supress them temporarily, set DJANGO_SECURE_SSL_REDIRECT to blank. Follow the instructions here to get SSL set up: https://help.pythonanywhere.com/pages/SSLOwnDomains/*

7.6 Optional: static files

If you want to use the PythonAnywhere static files service instead of using whitenoise or S3, you'll find its configuration section on the Web tab. Essentially you'll need an entry to match your `STATIC_URL` and `STATIC_ROOT` settings. There's more info here: https://help.pythonanywhere.com/pages/DjangoStaticFiles

7.7 Future deployments

For subsequent deployments, the procedure is much simpler. In a Bash console:

```
workon my-virtualenv-name
cd project-directory
git pull
python manage.py migrate
python manage.py collectstatic
```

And then go to the Web tab and hit **Reload**

TIP: *if you're really keen, you can set up git-push based deployments: https://blog.pythonanywhere.com/87/*

CHAPTER 8

Deployment on Heroku

You can either push the 'deploy' button in your generated README.rst or run these commands to deploy the project to Heroku:

```
heroku create --buildpack https://github.com/heroku/heroku-buildpack-python

heroku addons:create heroku-postgresql:hobby-dev
heroku pg:backups schedule --at '02:00 America/Los_Angeles' DATABASE_URL
heroku pg:promote DATABASE_URL

heroku addons:create heroku-redis:hobby-dev
heroku addons:create mailgun

heroku config:set DJANGO_ADMIN_URL="$(openssl rand -base64 32)"
heroku config:set DJANGO_SECRET_KEY="$(openssl rand -base64 64)"
heroku config:set DJANGO_SETTINGS_MODULE='config.settings.production'
heroku config:set DJANGO_ALLOWED_HOSTS='.herokuapp.com'

heroku config:set DJANGO_AWS_ACCESS_KEY_ID=YOUR_AWS_ID_HERE
heroku config:set DJANGO_AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_ACCESS_KEY_HERE
heroku config:set DJANGO_AWS_STORAGE_BUCKET_NAME=YOUR_AWS_S3_BUCKET_NAME_HERE

heroku config:set DJANGO_MAILGUN_SERVER_NAME=YOUR_MALGUN_SERVER
heroku config:set DJANGO_MAILGUN_API_KEY=YOUR_MAILGUN_API_KEY
heroku config:set MAILGUN_SENDER_DOMAIN=YOUR_MAILGUN_SENDER_DOMAIN

heroku config:set PYTHONHASHSEED=random
heroku config:set DJANGO_ADMIN_URL=\^somelocation/

git push heroku master
heroku run python manage.py migrate
heroku run python manage.py check --deploy
heroku run python manage.py createsuperuser
heroku open
```

CHAPTER 9

Deployment with Docker

9.1 Prerequisites

- Docker (at least 1.10)
- Docker Compose (at least 1.6)

9.2 Understand the Compose Setup

Before you start, check out the *docker-compose.yml* file in the root of this project. This is where each component of this application gets its configuration from. Notice how it provides configuration for these services:

- *postgres* service that runs the database
- *redis* for caching
- *nginx* as reverse proxy
- *django* is the Django project run by gunicorn

If you chose the *use_celery* option, there are two more services:

- *celeryworker* which runs the celery worker process
- *celerybeat* which runs the celery beat process

If you chose the *use_letsencrypt* option, you also have:

- *certbot* which keeps your certs from letsencrypt up-to-date

9.3 Populate .env With Your Environment Variables

Some of these services rely on environment variables set by you. There is an *env.example* file in the root directory of this project as a starting point. Add your own variables to the file and rename it to *.env*. This file won't be tracked by git by default so you'll have to make sure to use some other mechanism to copy your secret if you are relying solely on git.

9.4 Optional: nginx-proxy Setup

By default, the application is configured to listen on all interfaces on port 80. If you want to change that, open the *docker-compose.yml* file and replace *0.0.0.0* with your own ip.

If you are using nginx-proxy to run multiple application stacks on one host, remove the port setting entirely and add *VIRTUAL_HOST=example.com* to your env file. Here, replace example.com with the value you entered for *domain_name*.

This pass all incoming requests on nginx-proxy to the nginx service your application is using.

9.5 Optional: Postgres Data Volume Modifications

Postgres is saving its database files to the *postgres_data* volume by default. Change that if you wan't something else and make sure to make backups since this is not done automatically.

9.6 Optional: Certbot and Let's Encrypt Setup

If you chose *use_letsencrypt* and will be using certbot for https, you must do the following before running anything with docker-compose:

Replace dhparam.pem.example with a generated dhparams.pem file before running anything with docker-compose. You can generate this on ubuntu or OS X by running the following in the project root:

```
$ openssl dhparam -out /path/to/project/compose/nginx/dhparams.pem 2048
```

If you would like to add additional subdomains to your certificate, you must add additional parameters to the certbot command in the *docker-compose.yml* file:

Replace:

```
command: bash -c "sleep 6 && certbot certonly -n --standalone -d {{ cookiecutter.domain_name }} --tex
```

With:

```
command: bash -c "sleep 6 && certbot certonly -n --standalone -d {{ cookiecutter.domain_name }} -d ww
```

Please be cognizant of Certbot/Letsencrypt certificate requests limits when getting this set up. The provide a test server that does not count against the limit while you are getting set up.

The certbot certificates expire after 3 months. If you would like to set up autorenewal of your certificates, the following commands can be put into a bash script:

```
#!/bin/bash
cd <project directory>
docker-compose run --rm --name certbot certbot bash -c "sleep 6 && certbot certonly --standalone -d
docker exec pearl_nginx_1 nginx -s reload
```

And then set a cronjob by running *crontab -e* and placing in it (period can be adjusted as desired):

```
0 4 * * 1 /path/to/bashscript/renew_certbot.sh
```

9.7 Run your app with docker-compose

To get started, pull your code from source control (don't forget the *.env* file) and change to your projects root directory.

You'll need to build the stack first. To do that, run:

```
docker-compose build
```

Once this is ready, you can run it with:

```
docker-compose up
```

To run a migration, open up a second terminal and run:

```
docker-compose run django python manage.py migrate
```

To create a superuser, run:

```
docker-compose run django python manage.py createsuperuser
```

If you need a shell, run:

```
docker-compose run django python manage.py shell
```

To get an output of all running containers.

To check your logs, run:

```
docker-compose logs
```

If you want to scale your application, run:

```
docker-compose scale django=4
docker-compose scale celeryworker=2
```

Warning: Don't run the scale command on postgres, celerybeat, certbot, or nginx.

If you have errors, you can always check your stack with *docker-compose*. Switch to your projects root directory and run:

```
docker-compose ps
```

9.8 Supervisor Example

Once you are ready with your initial setup, you wan't to make sure that your application is run by a process manager to survive reboots and auto restarts in case of an error. You can use the process manager you are most familiar with. All it needs to do is to run *docker-compose up* in your projects root directory.

If you are using *supervisor*, you can use this file as a starting point:

```
[program:{{cookiecutter.project_slug}}]
command=docker-compose up
directory=/path/to/{{cookiecutter.project_slug}}
redirect_stderr=true
autostart=true
autorestart=true
priority=10
```

Place it in */etc/supervisor/conf.d/{{cookiecutter.project_slug}}.conf* and run:

```
supervisorctl reread
supervisorctl start {{cookiecutter.project_slug}}
```

To get the status, run:

```
supervisorctl status
```

CHAPTER 10

Database Backups with Docker

The database has to be running to create/restore a backup. These examples show local examples. If you want to use it on a remote server, remove `-f dev.yml` from each example.

10.1 Running Backups

Run the app with *docker-compose -f dev.yml up*.

To create a backup, run:

```
docker-compose -f dev.yml run postgres backup
```

To list backups, run:

```
docker-compose -f dev.yml run postgres list-backups
```

To restore a backup, run:

```
docker-compose -f dev.yml run postgres restore filename.sql
```

Where <containerId> is the ID of the Postgres container. To get it, run:

```
docker ps
```

To copy the files from the running Postgres container to the host system:

```
docker cp <containerId>:/backups /host/path/target
```

10.2 Restoring From Backups

To restore the production database to a local PostgreSQL database:

```
createdb NAME_OF_DATABASE
psql NAME_OF_DATABASE < NAME_OF_BACKUP_FILE
```

CHAPTER 11

FAQ

11.1 Why is there a django.contrib.sites directory in Cookiecutter Django?

It is there to add a migration so you don't have to manually change the `sites.Site` record from `example.com` to whatever your domain is. Instead, your `{{cookiecutter.domain_name}}` and {{cookiecutter.project_name}} value is placed by **Cookiecutter** in the domain and name fields respectively.

See 0003_set_site_domain_and_name.py.

11.2 Why aren't you using just one configuration file (12-Factor App)

TODO

11.3 Why doesn't this follow the layout from Two Scoops of Django 1.8?

You may notice that some elements of this project do not exactly match what we describe in chapter 3 of Two Scoops of Django. The reason for that is this project, amongst other things, serves as a test bed for trying out new ideas and concepts. Sometimes they work, sometimes they don't, but the end result is that it won't necessarily match precisely what is described in the book I co-authored.

CHAPTER 12

Troubleshooting

This page contains some advice about errors and problems commonly encountered during the development of Cookiecutter Django applications.

1. If you get the error `jinja2.exceptions.TemplateSyntaxError: Encountered unknown tag 'now'.` , please upgrade your cookiecutter version to >= 1.4 (see issue # 528)
2. `project_slug` must be a valid Python module name or you will have issues on imports.

CHAPTER 13

Creating your first app with Cookiecutter-Django

This tutorial will show you how to build a simple app using the Cookiecutter Django templating system. We'll be building a cookie polling app to determine the most popular flavor of cookie.

Developers who have never used Django will learn the basics of creating a Django app; developers who are experienced with Django will learn how to set up a project within the Cookiecutter system. While many Django tutorials use the default SQLite database, Cookiecutter Django uses PostGres only, so we'll have you install and use that.

13.1 Dependencies

This tutorial was written on Windows 10 using git bash; alternate instructions for Mac OS and Linux will be provided when needed. Any Linux-style shell should work for the following commands.

You should have your preferred versions of Python and Django installed. Use the latest stable versions if you have no preference.

You should have Virtualenv and Cookiecutter installed:

```
$ pip install virtualenv
$ pip install cookiecutter
```

You should also have PostgreSQL installed on your machine–just download and run the installer for your OS. The install menu will prompt you for a password, which you'll use when creating the project's database.

13.2 Instructions

1. **Setup** – how to set up a virtual environment
2. **Cookiecutter** – use Cookiecutter to initialize a project with your own customized information.
3. **Building the App** – creating the My Favorite Cookie application.

13.2.1 1. Setup

Virtual Environment

Create a virtual environment for your project. Cookiecutter will install a bunch of dependencies for you automatically; using a virtualenv will prevent this from interfering with your other work.

```
$ virtualenv c:/.virtualenvs/cookie_polls
```

Replace `c:/.virtualenvs` with the path to your own `.virtualenvs` folder.

Activate the virtual environment by calling `source` on the `activate` shell script . On Windows you'll call this from the virtualenv's `scripts` folder:

```
$  source /path/to/.virtualenvs/cookie_polls/scripts/activate
```

On other operating systems, it'll be found in the `bin` folder.

```
$  source /path/to/.virtualenvs/cookie_polls/bin/activate
```

You'll know the virtual environment is active because its name will appear in parentheses before the command prompt. When you're done with this project, you can leave the virtual environment with the `deactivate` command.

```
(cookie_polls)
$  deactivate
```

Now you're ready to create your project using Cookiecutter.

13.2.2 2. Cookiecutter

Django developers may be familiar with the `startproject` command, which initializes the directory structure and required files for a bare-bones Django project. While this is fine when you're just learning Django for the first time, it's not great for a real production app. Cookiecutter takes care of a lot of standard tasks for you, including installing software dependencies, setting up testing files, and including and organizing common libraries like Bootstrap and AngularJS. It also generates a software license and a README.

Change directories into the folder where you want your project to live, and run `cookiecutter` followed by the URL of Cookiecutter's Github repo.

```
$ cd /my/project/folder
(cookie_polls)
my/project/folder
$ cookiecutter https://github.com/pydanny/cookiecutter-django
```

This will prompt you for a bunch of values specific to your project. Press "enter" without typing anything to use the default values, which are shown in [brackets] after the question. You can learn about all the different options here, but for now we'll use the defaults for everything but your name, your email, the project's name, and the project's description.

```
project_name [project_name]: My Favorite Cookie
project_slug [My_Favorite_Cookie]:
author_name [Your Name]: Emily Cain
email [Your email]: contact@emcain.net
description [A short description of the project.]: Poll your friends to determine the most popular co
```

Then hit "enter" to use the default values for everything else.

CHAPTER 14

Deployment with Elastic Beanstalk

14.1 Warning: Experimental

This is experimental. For the time being there will be bugs and issues. If you've never used Elastic Beanstalk before, please hold off before trying this option.

On the other hand, we need help cleaning this up. If you do have knowledge of Elastic Beanstalk, we would appreciate the help. :)

14.2 Prerequisites

- awsebcli

14.3 Instructions

If you haven't done so, create a directory of environments:

```
eb init -p python3.4 MY_PROJECT_SLUG
```

Replace *MY_PROJECT_SLUG* with the value you entered for *project_slug*.

Once that is done, create the environment (server) where the app will run:

```
eb create MY_PROJECT_SLUG
# Note: This will eventually fail on a postgres error, because postgres doesn't exist yet
```

Now make sure you are in the right environment:

```
eb list
```

If you are not in the right environment, then put yourself in the correct one:

```
eb use MY_PROJECT_SLUG
```

Set the environment variables. Notes: You will be prompted if the *.env* file is missing. The script will ignore any PostgreSQL values, as RDS uses it's own system:

```
# Set the environment variables
python ebsetenv.py
```

Speaking of PostgreSQL, go to the Elasting Beanstalk configuration panel for RDS. Create new RDS database, with these attributes:

- PostgreSQL
- Version 9.4.9
- Size db.t2.micro (You can upgrade later)

(Get some coffee, this is going to take a while)

Once you have a database specified, deploy again so your instance can pick up the new PostgreSQL values:

```
eb deploy
```

Take a look:

```
eb open
```

14.4 FAQ

14.4.1 Why Not Use Docker on Elastic Beanstalk?

Because I didn't want to add an abstraction (Docker) on top of an abstraction (Elastic Beanstalk) on top of an abstraction (Cookiecutter Django).

14.4.2 Why Can't I Use Both Docker/Heroku with Elastic Beanstalk?

Because the environment variables that our Docker and Heroku setups use for PostgreSQL access is different then how Amazon RDS handles this access. At this time we're just trying to get things to work reliably with Elastic Beanstalk, and full integration will come later.

CHAPTER 15

Indices and tables

- genindex
- search

we are doing. Then we can * modindex back in.

Symbols

D

E

F

H

L

P

V