cookiecutter-django Documentation

Release 2021.27.2

cookiecutter-django

Jul 06, 2021

Contents

A Cookiecutter template for Django.

Contents:

CHAPTER 1

Project Generation Options

project_name: Your project's human-readable name, capitals and spaces allowed.

project_slug: Your project's slug without dashes or spaces. Used to name your repo and in other places where a Python-importable version of your project name is needed.

description: Describes your project and gets used in places like `README.rst` and such.

author_name: This is you! The value goes into places like `LICENSE` and such.

email: The email address you want to identify yourself in the project.

domain_name: The domain name you plan to use for your project once it goes live. Note that it can be safely changed later on whenever you need to.

version: The version of the project at its inception.

open_source_license: 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

timezone: The value to be used for the `TIME_ZONE` setting of the project.

windows: Indicates whether the project should be configured for development on Windows.

use_pycharm: Indicates whether the project should be configured for development with PyCharm.

use_docker: Indicates whether the project should be configured to use Docker and Docker Compose.

postgresql_version: Select a PostgreSQL version to use. The choices are:

1. 12.3
2. 11.8

3. 10.8
4. 9.6
5. 9.5

js_task_runner: Select a JavaScript task runner. The choices are:

1. None
2. Gulp

cloud_provider: Select a cloud provider for static & media files. The choices are:

1. AWS
2. GCP
3. None

Note that if you choose no cloud provider, the media files will be served by a local nginx instance.

mail_service: Select an email service that Django-Anymail provides

1. Mailgun
2. Amazon SES
3. Mailjet
4. Mandrill
5. Postmark
6. SendGrid
7. SendinBlue
8. SparkPost
9. Other SMTP

use_async: Indicates whether the project should use web sockets with Uvicorn + Gunicorn.

use_drf: Indicates whether the project should be configured to use Django Rest Framework.

custom_bootstrap_compilation: Indicates whether the project should support Bootstrap recompilation via the selected JavaScript task runner's task. This can be useful for real-time Bootstrap variable alteration.

use_compressor: Indicates whether the project should be configured to use Django Compressor.

use_celery: Indicates whether the project should be configured to use Celery.

use_mailhog: Indicates whether the project should be configured to use MailHog.

use_sentry: Indicates whether the project should be configured to use Sentry.

use_whitenoise: Indicates whether the project should be configured to use WhiteNoise.

use_heroku: Indicates whether the project should be configured so as to be deployable to Heroku.

ci_tool: Select a CI tool for running tests. The choices are:

1. Travis CI
2. Gitlab CI
3. Github Actions
4. None

use_activecollab_digger: Indicates whether the project should install the ActiveCollab Digger app for the ActiveCollab project management tool.

use_elasticsearch: Indicates whether the project should be configured to use the Elasticsearch search engine and the Kibana platform.

use_ldap_authentication: Indicates whether the project should be configured to use LDAP authentication via the django-auth-ldap app.

use_wagtail: Indicates whether the project should be configure to use the Wagtail CMS with the django-kdl-wagtail app.

use_wagtail_search: Indicates whether the project should be configured to use the Wagtail CMS search.

keep_local_envs_in_vcs: Indicates whether the project's `.envs/.local/` should be kept in VCS (comes in handy when working in teams where local environment reproducibility is strongly encouraged). Note: .env(s) are only utilized when Docker Compose and/or Heroku support is enabled.

debug: Indicates whether the project should be configured for debugging. This option is relevant for Cookiecutter Django developers only.

CHAPTER 2

Getting Up and Running Locally

2.1 Setting Up Development Environment

Make sure to have the following on your host:

- Python 3.8
- PostgreSQL.
- Redis, if using Celery
- Cookiecutter

First things first.

1. Create a virtualenv:

```
$ python3.8 -m venv <virtual env path>
```

2. Activate the virtualenv you have just created:

```
$ source <virtual env path>/bin/activate
```

3. Install cookiecutter-django:

```
$ cookiecutter gh:pydanny/cookiecutter-django
```

4. Install development requirements:

```
$ pip install -r requirements/local.txt
$ git init # A git repo is required for pre-commit to install
$ pre-commit install
```

> **Note:** the *pre-commit* exists in the generated project as default. for the details of *pre-commit*, follow the [site of pre-commit](https://pre-commit.com/).

5. Create a new PostgreSQL database using createdb:

```
$ createdb <what you have entered as the project_slug at setup stage> -U postgres␣
↪--password <password>
```

Note: if this is the first time a database is created on your machine you might need an initial PostgreSQL set up to allow local connections & set a password for the `postgres` user. The postgres documentation explains the syntax of the config file that you need to change.

6. Set the environment variables for your database(s):

```
$ export DATABASE_URL=postgres://postgres:<password>@127.0.0.1:5432/<DB name␣
↪given to createdb>
# Optional: set broker URL if using Celery
$ export CELERY_BROKER_URL=redis://localhost:6379/0
```

Note: Check out the *Settings* page for a comprehensive list of the environments variables.

See also:

To help setting up your environment variables, you have a few options:

- create an `.env` file in the root of your project and define all the variables you need in it. Then you just need to have `DJANGO_READ_DOT_ENV_FILE=True` in your machine and all the variables will be read.
- Use a local environment manager like direnv

7. Apply migrations:

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

8. If you're running synchronously, see the application being served through Django development server:

```
$ python manage.py runserver 0.0.0.0:8000
```

or if you're running asynchronously:

```
$ uvicorn config.asgi:application --host 0.0.0.0 --reload
```

2.2 Setup Email Backend

2.2.1 MailHog

Note: In order for the project to support *MailHog* it must have been bootstrapped with `use_mailhog` set to `y`.

MailHog is used to receive emails during development, it is written in Go and has no external dependencies.

For instance, one of the packages we depend upon, `django-allauth` sends verification emails to new users signing up as well as to the existing ones who have not yet verified themselves.

1. Download the latest MailHog release for your OS.

2. Rename the build to `MailHog`.
3. Copy the file to the project root.
4. Make it executable:

```
$ chmod +x MailHog
```

5. Spin up another terminal window and start it there:

```
./MailHog
```

6. Check out http://127.0.0.1:8025/ to see how it goes.

Now you have your own mail server running locally, ready to receive whatever you send it.

2.2.2 Console

Note: If you have generated your project with `use_mailhog` set to `n` this will be a default setup.

Alternatively, deliver emails over console via `EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'`.

In production, we have Mailgun configured to have your back!

2.3 Celery

If the project is configured to use Celery as a task scheduler then by default tasks are set to run on the main thread when developing locally. If you have the appropriate setup on your local machine then set the following in `config/settings/local.py`:

```
CELERY_TASK_ALWAYS_EAGER = False
```

To run Celery locally, make sure redis-server is installed (instructions are available at https://redis.io/topics/quickstart), run the server in one terminal with *redis-server*, and then start celery in another terminal with the following command:

```
celery -A config.celery_app worker --loglevel=info
```

2.4 Sass Compilation & Live Reloading

If you'd like to take advantage of live reloading and Sass compilation you can do so with a little bit of preparation, see *Sass Compilation & Live Reloading*.

2.5 Summary

Congratulations, you have made it! Keep on reading to unleash full potential of Cookiecutter Django.

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.

Note: If you're new to Docker, please be aware that some resources are cached system-wide and might reappear if you generate a project multiple times with the same name (e.g. *this issue with Postgres*).

3.1 Prerequisites

- Docker; if you don't have it yet, follow the installation instructions;
- Docker Compose; refer to the official documentation for the installation guide.

3.2 Build the Stack

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

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

Generally, if you want to emulate production environment use `production.yml` instead. And this is true for any other actions you might need to perform: whenever a switch is required, just do it!

3.3 Run the Stack

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 local.yml up
```

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

```
$ export COMPOSE_FILE=local.yml
```

And then run:

```
$ docker-compose up
```

To run in a detached (background) mode, just:

```
$ docker-compose up -d
```

3.4 Execute Management Commands

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

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

Here, `django` is the target service we are executing the commands against.

3.5 (Optionally) Designate your Docker Development Server IP

When `DEBUG` is set to `True`, the host is validated against `['localhost', '127.0.0.1', '[::1]']`. This is adequate when running a `virtualenv`. For Docker, in the `config.settings.local`, add your host development server IP to `INTERNAL_IPS` or `ALLOWED_HOSTS` if the variable exists.

3.6 Configuring the Environment

This is the excerpt from your project's `local.yml`:

```
# ...

postgres:
  build:
    context: .
    dockerfile: ./compose/production/postgres/Dockerfile
  volumes:
    - local_postgres_data:/var/lib/postgresql/data
    - local_postgres_data_backups:/backups
  env_file:
    - ./.envs/.local/.postgres

# ...
```

The most important thing for us here now is `env_file` section enlisting `./.envs/.local/.postgres`. Generally, the stack's behavior is governed by a number of environment variables (*env(s)*, for short) residing in `envs/`, for instance, this is what we generate for you:

```
.envs
├── .local
│   ├── .django
│   └── .postgres
└── .production
    ├── .django
    └── .postgres
```

By convention, for any service `sI` in environment `e` (you know `someenv` is an environment when there is a `someenv.yml` file in the project root), given `sI` requires configuration, a `.envs/.e/.sI` *service configuration* file exists.

Consider the aforementioned `.envs/.local/.postgres`:

```
# PostgreSQL
# --------------------------------------------------------------------------------
POSTGRES_HOST=postgres
POSTGRES_DB=<your project slug>
POSTGRES_USER=XgOWtQtJecsAbaIyslwGvFvPawftNaqO
POSTGRES_PASSWORD=jSljDz4whHuwO3aJIgVBrqEml5Ycbghorep4uVJ4xjDYQu0LfuTZdctj7y0YcCLu
```

The three envs we are presented with here are `POSTGRES_DB`, `POSTGRES_USER`, and `POSTGRES_PASSWORD` (by the way, their values have also been generated for you). You might have figured out already where these definitions will end up; it's all the same with `django` service container envs.

One final touch: should you ever need to merge `.envs/.production/*` in a single `.env` run the `merge_production_dotenvs_in_dotenv.py`:

```
$ python merge_production_dotenvs_in_dotenv.py
```

The `.env` file will then be created, with all your production envs residing beside each other.

3.7 Tips & Tricks

3.7.1 Fabric script

The Fabric script `fabfile.py` can be used as a shortcut for interactions with the Docker stack and for remote task automation. To get a list of the available tasks:

```
$ fab --list
Available tasks:

  backup    Create a database backup.
  clone     Clone the project repository into a host instance.
  compose   Run a raw compose command.
  deploy    Deploy the project. By default it creates a database backup before
            updating from source control and rebuilding the docker stack.
  django    Run a Django management command.
  down      Stop and remove stack components.
  restart   Restart one or more services.
  restore   Restore a database backup.
  shell     Connect to a running service.
  start     Start one or more services.
  stop      Stop one or more services.
  test      Run tests with pytest.
```

(continues on next page)

(continued from previous page)

```
  up          Build the stack for the host instance.
  update      Update the host instance from source control.
```

And for more details on how to use a task:

```
$ fab --help TASK_NAME
```

For project specific configuration edit the `[fabric]` section in the `setup.cfg` file.

Note: By default, when no options are passed to the task, the task will run in the local machine. The *clone* and *deploy* tasks only run in the remote host.

3.7.2 Activate a Docker Machine

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)"
```

3.7.3 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 -f local.yml run --rm --service-ports django
```

django-debug-toolbar

In order for `django-debug-toolbar` to work designate your Docker Machine IP with `INTERNAL_IPS` in `local.py`.

docker

The `container_name` from the yml file can be used to check on containers with docker commands, for example:

```
$ docker logs worker
$ docker top worker
```

3.7.4 Mailhog

When developing locally you can go with MailHog for email testing provided `use_mailhog` was set to `y` on setup. To proceed,

1. make sure `mailhog` container is up and running;
2. open up `http://127.0.0.1:8025`.

3.7.5 Celery tasks in local development

When not using docker Celery tasks are set to run in Eager mode, so that a full stack is not needed. When using docker the task scheduler will be used by default.

If you need tasks to be executed on the main thread during development set CELERY_TASK_ALWAYS_EAGER = True in config/settings/local.py.

Possible uses could be for testing, or ease of profiling with DJDT.

3.7.6 Celery Flower

Flower is a "real-time monitor and web admin for Celery distributed task queue".

Prerequisites:

- `use_docker` was set to `y` on project initialization;
- `use_celery` was set to `y` on project initialization.

By default, it's enabled both in local and production environments (`local.yml` and `production.yml` Docker Compose configs, respectively) through a `flower` service. For added security, `flower` requires its clients to provide authentication credentials specified as the corresponding environments' `.envs/.local/.django` and `.envs/.production/.django` `CELERY_FLOWER_USER` and `CELERY_FLOWER_PASSWORD` environment variables. Check out `localhost:5555` and see for yourself.

3.8 Developing locally with HTTPS

Increasingly it is becoming necessary to develop software in a secure environment in order that there are very few changes when deploying to production. Recently Facebook changed their policies for apps/sites that use Facebook login which requires the use of an HTTPS URL for the OAuth redirect URL. So if you want to use the `users` application with a OAuth provider such as Facebook, securing your communication to the local development environment will be necessary.

On order to create a secure environment, we need to have a trusted SSL certficate installed in our Docker application.

1. **Let's Encrypt**

   The official line from Let's Encrypt is:

   [For local development section] ... The best option: Generate your own certificate, either self-signed or signed by a local root, and trust it in your operating system's trust store. Then use that certificate in your local web server. See below for details.

   See letsencrypt.org - certificates-for-localhost

2. **mkcert: Valid Https Certificates For Localhost**

   mkcert is a simple by design tool that hides all the arcane knowledge required to generate valid TLS certificates. It works for any hostname or IP, including localhost. It supports macOS, Linux, and Windows, and Firefox, Chrome and Java. It even works on mobile devices with a couple manual steps.

   See https://blog.filippo.io/mkcert-valid-https-certificates-for-localhost/

After installing a trusted TLS certificate, configure your docker installation. We are going to configure an `nginx` reverse-proxy server. This makes sure that it does not interfere with our `traefik` configuration that is reserved for production environements.

These are the places that you should configure to secure your local environment.

3.8.1 certs

Take the certificates that you generated and place them in a folder called `certs` on the projects root folder. Assuming that you registered your local hostname as `my-dev-env.local`, the certificates you will put in the folder should have the names `my-dev-env.local.crt` and `my-dev-env.local.key`.

3.8.2 local.yml

1. Add the `nginx-proxy` service.

```
...

nginx-proxy:
  image: jwilder/nginx-proxy:alpine
  container_name: nginx-proxy
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - /var/run/docker.sock:/tmp/docker.sock:ro
    - ./certs:/etc/nginx/certs
  restart: always
  depends_on:
    - django

...
```

2. Link the `nginx-proxy` to `django` through environmental variables.

   `django` already has an `.env` file connected to it. Add the following variables. You should do this especially if you are working with a team and you want to keep your local environment details to yourself.

```
# HTTPS
# ------------------------------------------------------------------------------
VIRTUAL_HOST=my-dev-env.local
VIRTUAL_PORT=8000
```

   The services run behind the reverse proxy.

3.8.3 config/settings/local.py

You should allow the new hostname.

```
ALLOWED_HOSTS = ["localhost", "0.0.0.0", "127.0.0.1", "my-dev-env.local"]
```

Rebuild your `docker` application.

```
$ docker-compose -f local.yml up -d --build
```

Go to your browser and type in your URL bar `https://my-dev-env.local`

See https with nginx for more information on this configuration.

3.8.4 .gitignore

Add `certs/*` to the `.gitignore` file. This allows the folder to be included in the repo but its contents to be ignored.

This configuration is for local development environments only. Do not use this for production since you might expose your local `rootCA-key.pem`.

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 and project settings:

Environment Variable	Django Setting	Development Default	Production Default
DJANGO_READ_DOT_ENV_FILE	READ_DOT_ENV_FILE	False	False

Environment Variable	Django Setting	Development Default	Production Default
DATABASE_URL	DATABASES	auto w/ Docker; postgres://project_slug w/o	raises error
DJANGO_ADMIN_URL	n/a	'admin/'	raises error
DJANGO_DEBUG	DEBUG	True	False
DJANGO_SECRET_KEY	SECRET_KEY	auto-generated	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	Produ
CELERY_BROKER_URL	CELERY_BROKER_URL	auto w/ Docker; raises error w/o	raises
DJANGO_AWS_ACCESS_KEY_ID	AWS_ACCESS_KEY_ID	n/a	raises
DJANGO_AWS_SECRET_ACCESS_KEY	AWS_SECRET_ACCESS_KEY	n/a	raises
DJANGO_AWS_STORAGE_BUCKET_NAME	AWS_STORAGE_BUCKET_NAME	n/a	raises
DJANGO_AWS_S3_REGION_NAME	AWS_S3_REGION_NAME	n/a	None
DJANGO_AWS_S3_CUSTOM_DOMAIN	AWS_S3_CUSTOM_DOMAIN	n/a	None
DJANGO_GCP_STORAGE_BUCKET_NAME	GS_BUCKET_NAME	n/a	raises
GOOGLE_APPLICATION_CREDENTIALS	n/a	n/a	raises
SENTRY_DSN	SENTRY_DSN	n/a	raises
SENTRY_ENVIRONMENT	n/a	n/a	produ
SENTRY_TRACES_SAMPLE_RATE	n/a	n/a	0.0
DJANGO_SENTRY_LOG_LEVEL	SENTRY_LOG_LEVEL	n/a	loggin
MAILGUN_API_KEY	MAILGUN_API_KEY	n/a	raises
MAILGUN_DOMAIN	MAILGUN_SENDER_DOMAIN	n/a	raises
MAILGUN_API_URL	n/a	n/a	"https
MAILJET_API_KEY	MAILJET_API_KEY	n/a	raises
MAILJET_SECRET_KEY	MAILJET_SECRET_KEY	n/a	raises
MAILJET_API_URL	n/a	n/a	"https
MANDRILL_API_KEY	MANDRILL_API_KEY	n/a	raises
MANDRILL_API_URL	n/a	n/a	"https
POSTMARK_SERVER_TOKEN	POSTMARK_SERVER_TOKEN	n/a	raises

Table 1 – continued from previous page

Environment Variable	Django Setting	Development Default	Produ
POSTMARK_API_URL	n/a	n/a	"https
SENDGRID_API_KEY	SENDGRID_API_KEY	n/a	raises
SENDGRID_GENERATE_MESSAGE_ID	True	n/a	raises
SENDGRID_MERGE_FIELD_FORMAT	None	n/a	raises
SENDGRID_API_URL	n/a	n/a	"https
SENDINBLUE_API_KEY	SENDINBLUE_API_KEY	n/a	raises
SENDINBLUE_API_URL	n/a	n/a	"https
SPARKPOST_API_KEY	SPARKPOST_API_KEY	n/a	raises
SPARKPOST_API_URL	n/a	n/a	"https

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)

4.2 KDL Settings

4.2.1 ActiveCollab Digger

Environment Variable	Django Setting	Development Default	Production Default
AC_DIGGER_COMPANY_ID	AC_COMPANY_ID	n/a	raises error
AC_DIGGER_PROJECT_ID	AC_PROJECT_ID	n/a	raises error
AC_DIGGER_USER_ID	AC_USER	n/a	raises error
AC_DIGGER_API_TOKEN	AC_TOKEN	n/a	raises error

4.2.2 LDAP Authentication

These settings are only used in a production environment.

Environment Variable	Django Setting	Development Default	Production Default
LDAP_SERVER_URI	AUTH_LDAP_SERVER_URI	n/a	"ldap://ldap1.cch.kcl.ac.uk"
LDAP_BIND_DN	AUTH_LDAP_BIND_DN	n/a	""
LDAP_BIND_PASSWORD	AUTH_LDAP_BIND_PASSWORD	n/a	""
LDAP_BASE_DC	LDAP_BASE_DC	n/a	"dc=dighum,dc=kcl,dc=ac,dc=uk"
LDAP_BASE_GROUP	LDAP_BASE_GROUP	n/a	"kdl-staff"
LDAP_FIRST_NAME_FIELD	AUTH_LDAP_USER_ATTR_MAP	n/a	"givenName"
LDAP_LAST_NAME_FIELD	AUTH_LDAP_USER_ATTR_MAP	n/a	"sn"
LDAP_EMAIL_FIELD	AUTH_LDAP_USER_ATTR_MAP	n/a	"mail"

4.2.3 Elasticsearch

Environment Variable	Django Setting	Development Default	Production Default
DISCOVERY_TYPE	n/a	single-node	single-node

Kibana

Environment Variable	Django Setting	Development Default	Production Default
SERVER_NAME	n/a	kibana	kibana
SERVER_HOST	n/a	0	0
ELASTICSEARCH_HOSTS	n/a	http://elasticsearch:9200	http://elasticsearch:9200

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 88 chars
- Exclude `.tox,.git,*/migrations/*,*/static/CACHE/*,docs,node_modules`

5.2 pylint

To run pylint:

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

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

- Use the pylint_django plugin. If using Celery, also use pylint_celery.
- Set max line length to 88 chars
- Disable linting messages for missing docstring and invalid name
- max-parents=13

5.3 pycodestyle

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

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

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

- Set max line length to 88 chars
- Exclude `.tox,.git,*/migrations/*,*/static/CACHE/*,docs,node_modules`

CHAPTER 6

Testing

We encourage users to build application tests. As best practice, this should be done immediately after documentation of the application being built, before starting on any coding.

6.1 Pytest

This project uses the Pytest, a framework for easily building simple and scalable tests. After you have set up to develop locally, run the following commands to make sure the testing environment is ready:

```
$ pytest
```

You will get a readout of the *users* app that has already been set up with tests. If you do not want to run the *pytest* on the entire project, you can target a particular app by typing in its location:

```
$ pytest <path-to-app-in-project/app>
```

If you set up your project to develop locally with docker, run the following command:

```
$ docker-compose -f local.yml run --rm django pytest
```

Targeting particular apps for testing in `docker` follows a similar pattern as previously shown above.

6.2 Coverage

You should build your tests to provide the highest level of **code coverage**. You can run the `pytest` with code `coverage` by typing in the following command:

```
$ docker-compose -f local.yml run --rm django coverage run -m pytest
```

Once the tests are complete, in order to see the code coverage, run the following command:

```
$ docker-compose -f local.yml run --rm django coverage report
```

Note: At the root of the project folder, you will find the *pytest.ini* file. You can use this to customize the `pytest` to your liking.

There is also the *.coveragerc*. This is the configuration file for the `coverage` tool. You can find out more about configuring `coverage`.

See also:

For unit tests, run:

```
$ python manage.py test
```

Since this is a fresh install, and there are no tests built using the Python unittest library yet, you should get feedback that says there were no tests carried out.

CHAPTER 7

Document

This project uses Sphinx documentation generator.

After you have set up to develop locally, run the following command from the project directory to build and serve HTML documentation:

```
$ make -C docs livehtml
```

If you set up your project to develop locally with docker, run the following command:

```
$ docker-compose -f local.yml up docs
```

Navigate to port 7000 on your host to see the documentation. This will be opened automatically at localhost for local, non-docker development.

Note: using Docker for documentation sets up a temporary SQLite file by setting the environment variable `DATABASE_URL=sqlite:///readthedocs.db` in `docs/conf.py` to avoid a dependency on PostgreSQL.

7.1 Generate API documentation

Edit the `docs` files and project application docstrings to create your documentation.

Sphinx can automatically include class and function signatures and docstrings in generated documentation. See the generated project documentation for more examples.

7.2 Setting up ReadTheDocs

To setup your documentation on ReadTheDocs, you must

1. Go to ReadTheDocs and login/create an account
2. Add your GitHub repository

3. Trigger a build

Additionally, you can auto-build Pull Request previews, but you must enable it.

CHAPTER 8

Versioning

This project uses bumpversion to manage the version strings related to releases. The `history.rst` file should also be updated with the release notes for each version.

To configure `bumpversion` edit the `setup.cfg` file. Running the `bumpversion` command will run a `git commit` and `git tag` by default. Also by default, the version is updated in the project's `__init__.py` and `history.rst` files.

Before running `bumpversion`, make sure all the changes are commited, and run:

```
$ bump2version [major|minor|patch]
```

Examples:

```
$ cat setup.cfg | grep current_version
  current_version = 0.1.0

$ bump2version patch

$ cat setup.cfg | grep current_version
  current_version = 0.1.1

$ bump2version minor

$ cat setup.cfg | grep current_version
  current_version = 0.2.0

$ bump2version minor

$ cat setup.cfg | grep current_version
  current_version = 1.0.0
```

For more examples and configuration options see the bumpversion documentation.

8.1 Git Commit Messages

For the Git commit messages, it is recommend to use the Emoji-Log spec. Sample `.gitconfig` configuration:

```
[alias]
  # https://opensource.com/article/19/2/emoji-log-git-commit-messages
  ac = "!f() { git add ${@:1:$(($# - 1))}; git commit -m \"${@:$#}\"; }; f"
  new = "!f() { git ac ${@:1:$(($# - 1))} \" New: ${@:$#}\";  }; f"
  imp = "!f() { git ac ${@:1:$(($# - 1))} \" Improve: ${@:$#}\";  }; f"
  fix = "!f() { git ac ${@:1:$(($# - 1))} \" Fix: ${@:$#}\";  }; f"
  rlz = "!f() { git ac ${@:1:$(($# - 1))} \" Release: ${@:$#}\";  }; f"
  doc = "!f() { git ac ${@:1:$(($# - 1))} \" Doc: ${@:$#}\";  }; f"
  tst = "!f() { git ac ${@:1:$(($# - 1))} \" Test: ${@:$#}\";  }; f"
```

CHAPTER 9

Deployment on PythonAnywhere

9.1 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` {%- if cookiecutter.use_compressor == "y" %}and `compress` {%- endif %}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 :)

9.2 Getting your code and dependencies installed on PythonAnywhere

Make sure your project is fully committed 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/python3.8 my-project-name
pip install -r requirements/production.txt  # may take a few minutes
```

9.3 Setting environment variables in the console

Generate a secret key for yourself, eg like this:

```
python -c 'import random;import string; print("".join(random.SystemRandom().
↪choice(string.digits + string.ascii_letters + string.punctuation) for _ in␣
↪range(50)))'
```

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 WEB_CONCURRENCY=4
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 MAILGUN_API_KEY='<mailgun key>'
export MAILGUN_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.*

9.4 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-port>/<database-name>'
# or
export DATABASE_URL='mysql://<pythonanywhere-username>:<mysql-password>@<mysql-
↪address>/<database-name>'
# 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
{%- if cookiecutter.use_compressor == "y" %}python manage.py compress {%- endif %}
# and, optionally
python manage.py createsuperuser
```

9.5 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['MAILGUN_API_KEY'] = '<as above>'
os.environ['MAILGUN_DOMAIN'] = '<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 suppress them temporarily, set DJANGO_SECURE_SSL_REDIRECT to blank. Follow the instructions here to get SSL set up: https://help.pythonanywhere.com/pages/SSLOwnDomains/*

9.6 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

9.7 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
{%- if cookiecutter.use_compressor == "y" %}python manage.py compress {%- endif %}
```

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 10

Deployment on Heroku

10.1 Commands to run

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
# On Windows use double quotes for the time zone, e.g.
# heroku pg:backups schedule --at "02:00 America/Los_Angeles" DATABASE_URL
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:starter

heroku config:set PYTHONHASHSEED=random

heroku config:set WEB_CONCURRENCY=4

heroku config:set DJANGO_DEBUG=False
heroku config:set DJANGO_SETTINGS_MODULE=config.settings.production
heroku config:set DJANGO_SECRET_KEY="$(openssl rand -base64 64)"

# Generating a 32 character-long random string without any of the visually similar␣
↪characters "IOl01":
heroku config:set DJANGO_ADMIN_URL="$(openssl rand -base64 4096 | tr -dc 'A-HJ-NP-Za-
↪km-z2-9' | head -c 32)/"

# Set this to your Heroku app url, e.g. 'bionic-beaver-28392.herokuapp.com'
heroku config:set DJANGO_ALLOWED_HOSTS=

# Assign with AWS_ACCESS_KEY_ID
```

(continues on next page)

(continued from previous page)

```
heroku config:set DJANGO_AWS_ACCESS_KEY_ID=

# Assign with AWS_SECRET_ACCESS_KEY
heroku config:set DJANGO_AWS_SECRET_ACCESS_KEY=

# Assign with AWS_STORAGE_BUCKET_NAME
heroku config:set DJANGO_AWS_STORAGE_BUCKET_NAME=

git push heroku master

heroku run python manage.py createsuperuser

heroku run python manage.py check --deploy

heroku open
```

Warning: If your email server used to send email isn't configured properly (Mailgun by default), attempting to send an email will cause an Internal Server Error.

By default, django-allauth is setup to have emails verifications mandatory, which means it'll send a verification email when an unverified user tries to log-in or when someone tries to sign-up.

This may happen just after you've setup your Mailgun account, which is running in a sandbox subdomain by default. Either add your email to the list of authorized recipients or verify your domain.

10.2 Optional actions

10.2.1 Celery

Celery requires a few extra environment variables to be ready operational. Also, the worker is created, it's in the `Procfile`, but is turned off by default:

```
# Set the broker URL to Redis
heroku config:set CELERY_BROKER_URL=`heroku config:get REDIS_URL`
# Scale dyno to 1 instance
heroku ps:scale worker=1
```

10.2.2 Sentry

If you're opted for Sentry error tracking, you can either install it through the Sentry add-on:

```
heroku addons:create sentry:f1
```

Or add the DSN for your account, if you already have one:

```
heroku config:set SENTRY_DSN=https://xxxx@sentry.io/12345
```

10.2.3 Gulp & Bootstrap compilation

If you've opted for a custom bootstrap build, you'll most likely need to setup your app to use multiple buildpacks: one for Python & one for Node.js:

```
heroku buildpacks:add --index 1 heroku/nodejs
```

At time of writing, this should do the trick: during deployment, the Heroku should run `npm install` and then `npm build`, which runs Gulp in cookiecutter-django.

If things don't work, please refer to the Heroku docs.

10.3 About Heroku & Docker

Although Heroku has some sort of Docker support, it's not supported by cookiecutter-django. We invite you to follow Heroku documentation about it.

CHAPTER 11

Deployment with Docker

11.1 Prerequisites

- Docker 17.05+.
- Docker Compose 1.17+

11.2 Understanding the Docker Compose Setup

Before you begin, check out the `production.yml` file in the root of this project. Keep note of how it provides configuration for the following services:

- `django`: your application running behind `Gunicorn`;
- `postgres`: PostgreSQL database with the application's relational data;
- `redis`: Redis instance for caching;
- `traefik`: Traefik reverse proxy with HTTPS on by default.

Provided you have opted for Celery (via setting `use_celery` to `y`) there are three more services:

- `celeryworker` running a Celery worker process;
- `celerybeat` running a Celery beat process;
- `flower` running Flower.

The `flower` service is served by Traefik over HTTPS, through the port `5555`. For more information about Flower and its login credentials, check out *Celery Flower* instructions for local environment.

11.3 Configuring the Stack

The majority of services above are configured through the use of environment variables. Just check out *Configuring the Environment* and you will know the drill.

To obtain logs and information about crashes in a production setup, make sure that you have access to an external Sentry instance (e.g. by creating an account with sentry.io), and set the `SENTRY_DSN` variable. Logs of level *logging.ERROR* are sent as Sentry events. Therefore, in order to send a Sentry event use:

```
import logging
logging.error("This event is sent to Sentry", extra={"<example_key>": "<example_value>
↪"})
```

The *extra* parameter allows you to send additional information about the context of this error.

You will probably also need to setup the Mail backend, for example by adding a Mailgun API key and a Mailgun sender domain, otherwise, the account creation view will crash and result in a 500 error when the backend attempts to send an email to the account owner.

Warning: If your email server used to send email isn't configured properly (Mailgun by default), attempting to send an email will cause an Internal Server Error.

By default, django-allauth is setup to have emails verifications mandatory, which means it'll send a verification email when an unverified user tries to log-in or when someone tries to sign-up.

This may happen just after you've setup your Mailgun account, which is running in a sandbox subdomain by default. Either add your email to the list of authorized recipients or verify your domain.

11.4 Optional: Use AWS IAM Role for EC2 instance

If you are deploying to AWS, you can use the IAM role to substitute AWS credentials, after which it's safe to remove the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` from `.envs/.production/.django`. To do it, create an IAM role and attach it to the existing EC2 instance or create a new EC2 instance with that role. The role should assume, at minimum, the `AmazonS3FullAccess` permission.

11.5 HTTPS is On by Default

SSL (Secure Sockets Layer) is a standard security technology for establishing an encrypted link between a server and a client, typically in this case, a web server (website) and a browser. Not having HTTPS means that malicious network users can sniff authentication credentials between your website and end users' browser.

It is always better to deploy a site behind HTTPS and will become crucial as the web services extend to the IoT (Internet of Things). For this reason, we have set up a number of security defaults to help make your website secure:

- If you are not using a subdomain of the domain name set in the project, then remember to put your staging/production IP address in the `DJANGO_ALLOWED_HOSTS` environment variable (see *Settings*) before you deploy your website. Failure to do this will mean you will not have access to your website through the HTTP protocol.
- Access to the Django admin is set up by default to require HTTPS in production or once *live*.

The Traefik reverse proxy used in the default configuration will get you a valid certificate from Lets Encrypt and update it automatically. All you need to do to enable this is to make sure that your DNS records are pointing to the server Traefik runs on.

You can read more about this feature and how to configure it, at Automatic HTTPS in the Traefik docs.

11.6 (Optional) Postgres Data Volume Modifications

Postgres is saving its database files to the `production_postgres_data` volume by default. Change that if you want something else and make sure to make backups since this is not done automatically.

11.7 Building & Running Production Stack

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

```
docker-compose -f production.yml build
```

Once this is ready, you can run it with:

```
docker-compose -f production.yml up
```

To run the stack and detach the containers, run:

```
docker-compose -f production.yml up -d
```

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

```
docker-compose -f production.yml run --rm django python manage.py migrate
```

To create a superuser, run:

```
docker-compose -f production.yml run --rm django python manage.py createsuperuser
```

If you need a shell, run:

```
docker-compose -f production.yml run --rm django python manage.py shell
```

To check the logs out, run:

```
docker-compose -f production.yml logs
```

If you want to scale your application, run:

```
docker-compose -f production.yml up --scale django=4
docker-compose -f production.yml up --scale celeryworker=2
```

> **Warning:** don't try to scale `postgres`, `celerybeat`, or `traefik`.

To see how your containers are doing run:

```
docker-compose -f production.yml ps
```

11.8 Example: Supervisor

Once you are ready with your initial setup, you want 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 -f production.yml 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 -f production.yml up
directory=/path/to/{{cookiecutter.project_slug}}
redirect_stderr=true
autostart=true
autorestart=true
priority=10
```

Move it to `/etc/supervisor/conf.d/{{cookiecutter.project_slug}}.conf` and run:

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

For status check, run:

```
supervisorctl status
```

11.9 Docker Security

This section contains a list of security issues identified by the Docker Bench for Security tool, after a deployment in an *Ubuntu 16.04* machine using the instructions in *Building & Running Production Stack*, and possible fixes.

Warning: After applying some of the fixes you might need to rebuild the stack, otherwise the issues might still be reported when re-running Docker Bench for Security.

11.9.1 Issues

The numbers in the headings correspond to the Docker Bench for Security test number.

1.2.1 - Ensure a separate partition for containers has been created

1.2.3 - Ensure auditing is configured for the Docker daemon and files/directories

Install auditd:

```
$ sudo apt-get install auditd
```

Edit the auditing system rules:

```
$ sudo vim /etc/audit/audit.rules
```

These rules instruct auditd to watch (-w) the specified file or directory and log any writes or attribute changes (-p wa) to those files:

```
-w /etc/default/docker -p wa
-w /etc/docker -p wa
-w /etc/docker/daemon.json -p wa
-w /etc/sysconfig/docker -p wa
-w /lib/systemd/system/docker.service -p wa
-w /lib/systemd/system/docker.socket -p wa
-w /project/docker -p wa
-w /usr/bin/docker -p wa
-w /usr/bin/containerd -p wa
-w /usr/bin/runc -p wa
-w /var/lib/docker -p wa
```

Restart autditd:

```
$ sudo systemctl restart auditd
```

2 - Docker daemon configuration

Listing 1: /etc/docker/daemon.json

```
{
    "data-root": "/project/docker",
    "icc": false,
    "live-restore": true,
    "log-driver": "syslog",
    "no-new-privileges": true,
    "userland-proxy": false,
    "userns-remap": "default"
}
```

For more information on how to configure the Docker daemon see the official Docker daemon documentation. Below is a short explanation for each of the configuration options in `daemon.json`.

`data_root` Root directory of persistent Docker state (default "/var/lib/docker")

`icc` 2.1 - Ensure network traffic is restricted between containers on the default bridge

`live-restore` 2.13 - Ensure live restore is Enabled

`log-driver` 2.12 - Ensure centralized and remote logging is configured

`no-new-privileges` 2.18 - Ensure containers are restricted from acquiring new privileges

`userland-proxy` 2.15 - Ensure Userland Proxy is Disabled

`userns-remap` 2.8 - Enable user namespace support

4.5 - Ensure Content trust for Docker is Enabled

To enable content trust for all users and sessions:

```
$ echo "DOCKER_CONTENT_TRUST=1" | sudo tee -a /etc/environment
```

For more information see the Docker content trust documentation.

4.6 - Ensure that HEALTHCHECK instructions have been added to container images

This should also cover the issue with *5.26 - Ensure that container health is checked at runtime*.

For example, to check every five minutes or so that a web-server is able to serve the site's main page within three seconds:

```
HEALTHCHECK --interval=5m --timeout=3s \
CMD curl -f http://localhost/ || exit 1
```

5.2 - Ensure that, if applicable, SELinux security options are set

5.7 - Ensure privileged ports are not mapped within containers

Mapping http port 80 and https port 443 is necessary for traefik/webserver. All the other ports in the stack are not privileged ports.

5.10 - Ensure that the memory usage for containers is limited

Runtime options with Memory.

5.11 - Ensure CPU priority is set appropriately on the container

Runtime options with CPUs.

5.12 - Ensure that the container's root filesystem is mounted as read only

Mount host-sensitive directories as read-only. In the default cookiecutter configuration no host-sensitive directories are shared with the containers.

5.13 - Ensure that incoming container traffic is bound to a specific host interface

5.14 - Ensure that the 'on-failure' container restart policy is set to '5'

Restart policy.

5.25 - Ensure that the container is restricted from acquiring additional privileges

Set in */etc/docker/daemon.json*.

5.27 - Ensure that Docker commands always make use of the latest version of their image

5.28 - Ensure that the PIDs cgroup limit is used

Useful Resources

- Top 20 Docker Security Tips
- 10 Docker Image Security Best Practices

- 10+ top open-source tools for Docker security
- How To Audit Docker Host Security with Docker Bench for Security on Ubuntu 16.04
- Securing Docker Containers on AWS
- Hardening Docker containers, images, and host - security toolkit
- Building Docker Images using Docker Compose and Gitlab CI/CD

CHAPTER 12

PostgreSQL Backups with Docker

Note: For brevity it is assumed that you will be running the below commands against local environment, however, this is by no means mandatory so feel free to switch to `production.yml` when needed.

12.1 Prerequisites

1. the project was generated with `use_docker` set to `y`;
2. the stack is up and running: `docker-compose -f local.yml up -d postgres`.

12.2 Creating a Backup

To create a backup, run:

```
$ docker-compose -f local.yml exec postgres backup
```

Assuming your project's database is named `my_project` here is what you will see:

```
Backing up the 'my_project' database...
SUCCESS: 'my_project' database backup 'backup_2018_03_13T09_05_07.sql.gz' has been␣
↪created and placed in '/backups'.
```

Keep in mind that `/backups` is the `postgres` container directory.

12.3 Viewing the Existing Backups

To list existing backups,

```
$ docker-compose -f local.yml exec postgres backups
```

These are the sample contents of `/backups`:

```
These are the backups you have got:
total 24K
-rw-r--r-- 1 root root 5.2K Mar 13 09:05 backup_2018_03_13T09_05_07.sql.gz
-rw-r--r-- 1 root root 5.2K Mar 12 21:13 backup_2018_03_12T21_13_03.sql.gz
-rw-r--r-- 1 root root 5.2K Mar 12 21:12 backup_2018_03_12T21_12_58.sql.gz
```

12.4 Copying Backups Locally

If you want to copy backups from your `postgres` container locally, `docker cp` command will help you on that.

For example, given `9c5c3f055843` is the container ID copying all the backups over to a local directory is as simple as

```
$ docker cp 9c5c3f055843:/backups ./backups
```

With a single backup file copied to `.` that would be

```
$ docker cp 9c5c3f055843:/backups/backup_2018_03_13T09_05_07.sql.gz .
```

12.5 Restoring from the Existing Backup

To restore from one of the backups you have already got (take the `backup_2018_03_13T09_05_07.sql.gz` for example),

```
$ docker-compose -f local.yml exec postgres restore backup_2018_03_13T09_05_07.sql.gz
```

You will see something like

```
Restoring the 'my_project' database from the '/backups/backup_2018_03_13T09_05_07.sql.
↪gz' backup...
INFO: Dropping the database...
INFO: Creating a new database...
INFO: Applying the backup to the new database...
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
# ...
ALTER TABLE
SUCCESS: The 'my_project' database has been restored from the '/backups/backup_2018_
↪03_13T09_05_07.sql.gz' backup.
```

12.6 Backup to Amazon S3

For uploading your backups to Amazon S3 you can use the aws cli container. There is an upload command for uploading the postgres /backups directory recursively and there is a download command for downloading a specific backup. The default S3 environment variables are used.

```
$ docker-compose -f production.yml run --rm awscli upload
$ docker-compose -f production.yml run --rm awscli download backup_2018_03_13T09_05_
↪07.sql.gz
```

CHAPTER 13

Sass Compilation & Live Reloading

If you'd like to take advantage of live reload and Sass compilation:

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

```
$ npm install
```

- Now you just need:

```
$ npm run dev
```

The base app will now run as it would with the usual `manage.py runserver` but with live reloading and Sass compilation enabled. When changing your Sass files, they will be automatically recompiled and change will be reflected in your browser without refreshing.

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

CHAPTER 14

Websocket

You can enable web sockets if you select `use_async` option when creating a project. That indicates whether the project can use web sockets with Uvicorn + Gunicorn.

14.1 Usage

JavaScript example:

```
> ws = new WebSocket('ws://localhost:8000/') // or 'wss://<mydomain.com>/' in prod
WebSocket {url: "ws://localhost:8000/", readyState: 0, bufferedAmount: 0, onopen:␣
↪null, onerror: null, ...}
> ws.onmessage = event => console.log(event.data)
event => console.log(event.data)
> ws.send("ping")
undefined
pong!
```

If you don't use Traefik, you might have to configure your reverse proxy accordingly (example with Nginx).

CHAPTER 15

FAQ

15.1 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.

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

TODO .. TODO

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

You may notice that some elements of this project do not exactly match what we describe in chapter 3 of Two Scoops of Django 1.11. 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 16

Troubleshooting

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

16.1 Server Error on sign-up/log-in

Make sure you have configured the mail backend (e.g. Mailgun) by adding the API key and sender domain

If your email server used to send email isn't configured properly (Mailgun by default), attempting to send an email will cause an Internal Server Error.

By default, django-allauth is setup to have emails verifications mandatory, which means it'll send a verification email when an unverified user tries to log-in or when someone tries to sign-up.

This may happen just after you've setup your Mailgun account, which is running in a sandbox subdomain by default. Either add your email to the list of authorized recipients or verify your domain.

16.2 Docker: Postgres authentication failed

Examples of logs:

```
postgres_1      | 2018-06-07 19:11:23.963 UTC [81] FATAL:  password authentication␣
↪failed for user "pydanny"
postgres_1      | 2018-06-07 19:11:23.963 UTC [81] DETAIL:  Password does not match␣
↪for user "pydanny".
postgres_1      |   Connection matched pg_hba.conf line 95: "host all all all md5"
```

If you recreate the project multiple times with the same name, Docker would preserve the volumes for the postgres container between projects. Here is what happens:

1. You generate the project the first time. The .env postgres file is populated with the random password

2. You run the docker-compose and the containers are created. The postgres container creates the database based on the .env file credentials
3. You "regenerate" the project with the same name, so the postgres .env file is populated with a new random password
4. You run docker-compose. Since the names of the containers are the same, docker will try to start them (not create them from scratch i.e. it won't execute the Dockerfile to recreate the database). When this happens, it tries to start the database based on the new credentials which do not match the ones that the database was created with, and you get the error message above.

To fix this, you can either:

- Clear your project-related Docker cache with `docker-compose -f local.yml down --volumes --rmi all`.
- Use the Docker volume sub-commands to find volumes (ls) and remove them (rm).
- Use the prune command to clear system-wide (use with care!).

16.3 Others

1. `project_slug` must be a valid Python module name or you will have issues on imports.
2. `jinja2.exceptions.TemplateSyntaxError: Encountered unknown tag 'now'.`: please upgrade your cookiecutter version to >= 1.4 (see #528)
3. New apps not getting created in project root: This is the expected behavior, because cookiecutter-django does not change the way that django startapp works, you'll have to fix this manually (see #1725)

CHAPTER 17

Indices and tables

- genindex
- search

Index

Symbols

C

D

F

H

L

P

V