django-naqsh Documentation

Release 2018.47.5

django-naqsh

Nov 23, 2018

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. 10.3
2. 10.2

3. 10.1
4. 9.6
5. 9.5
6. 9.4
7. 9.3

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.

use_travisci: Indicates whether the project should be configured to use Travis CI.

use_gitlabci: Indicates whether the project should be configured to use Gitlab CI.

use_grappelli: Indicates whether the project should be configured to use Django Grappelli.

use_cors_headers: Indicates whether the project should be configured to use Django CORS Headers.

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).

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 3.6
- PostgreSQL.
- Redis, if using Celery

First things first.

1. Create a virtualenv:

```
$ python3.6 -m venv <virtual env path>
```

2. Activate the virtualenv you have just created:

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

3. Install development requirements:

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

4. Create a new PostgreSQL database using createdb:

```
$ createdb <what you've entered as the project_slug at setup stage>
```

> **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.

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

```
$ export DJANGO_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

6. Apply migrations:

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

7. See the application being served through Django development server:

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

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 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.

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 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 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!

3.4 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
```

3.5 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.

3.6 (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.

3.7 Configuring the Environment

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

```
# ...

postgres:
  build:
    context: .
```

(continues on next page)

(continued from previous page)

```
    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
    ├── .caddy
    ├── .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` and `caddy` service container envs.

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

```
$ python .envs/merge.py
```

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

3.8 Tips & Tricks

3.8.1 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)"
```

3.8.2 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`.

3.8.3 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`.

3.8.4 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.

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
DJANGO_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	Production Default
CELERY_BROKER_URL	CELERY_BROKER_URL	auto w/ Docker; raises error w/o	raises error
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
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
MAILGUN_API_KEY	MAILGUN_ACCESS_KEY	n/a	raises error
MAILGUN_DOMAIN	MAILGUN_SENDER_DOMAIN	n/a	raises error

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

CHAPTER 6

Deployment on PythonAnywhere

6.1 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 :)

6.2 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/python3.6 my-project-name
pip install -r requirements/production.txt  # may take a few minutes
```

6.3 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.*

6.4 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
# and, optionally
python manage.py createsuperuser
```

6.5 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 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/*

6.6 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

6.7 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 7

Deployment on Heroku

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

# If using mailgun:
heroku addons:create mailgun:starter

heroku addons:create sentry:f1

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 similiar␣
↪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
heroku config:set DJANGO_AWS_ACCESS_KEY_ID=
```

(continues on next page)

(continued from previous page)

```
# 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 migrate
heroku run python manage.py createsuperuser
heroku run python manage.py collectstatic --no-input

heroku run python manage.py check --deploy

heroku open
```

CHAPTER 8

Deployment with Docker

8.1 Prerequisites

- Docker 1.10+.
- Docker Compose 1.6+

8.2 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;
- `caddy`: Caddy web server 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 (for more info, check out *Celery Flower* instructions for local environment).

8.3 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.

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.

8.4 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.

8.5 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 the 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 Caddy web server 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 Caddy runs on.

You can read more about this here at Automatic HTTPS in the Caddy docs.

8.6 (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.

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

Warning: don't try to scale `postgres`, `celerybeat`, or `caddy`.

To see how your containers are doing run:

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

8.8 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 start {{cookiecutter.project_slug}}
```

For status check, run:

```
supervisorctl status
```

CHAPTER 9

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.

9.1 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`.

9.2 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.

9.3 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
```

9.4 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 .
```

9.5 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.
```

CHAPTER 10

FAQ

10.1 Why is there a django.contrib.sites directory in Django Naqsh?

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.

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

TODO .. TODO

10.3 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 11

Troubleshooting

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

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. Internal server error on user registration: make sure you have configured the mail backend (e.g. Mailgun) by adding the API key and sender domain

CHAPTER 12

Indices and tables

- genindex
- search

Index

Symbols

C

D

F

H

L

P

V