# Django Blog

We use the django all auth for authentication/authorization, also allowing users to use the system with their social accounts. Articles can be rated. The blog has two groups, writer and traveller. Any user who sign up become travelers automatically while being promoted to a writer affords more priviledges.

<https://www.addwebsolution.com/blog/digitalocean-droplets-vs-app-platform>

Deployed on digital ocean app platform, a PaaS. Performs seamless code deployment direct from Git repositories such as GitHub

`

Databased used is postgresql

For production, our app is connected to render postgresql database and rediss.

In development we would configure them to run locally.

Features

* Ckeditor creates ability to insert images along with rich text content
* 2 groups, writers and travelers. Permissions are automatically created by a migration and are managed in admin. Signals are used to create profiles of every user that joins
* A search function
* Ability to sort by author names
* Can tweet author profiles/articles/etc.
* Use of social apps to perform authentication
* With mcert certificate files, the django app can also be served on HTTPS
* Use of X web intent to make tweets. It doesn’t require any API

The libraries and frameworks that characterize the blog:

* App is dockerised. All the services exist in their own containers.
* Bootstrap. Great to work with for UI.
* MPTT: Handles hierarchical data which means its great to build a nested comment section for the blog
* Pagination keeps number of posts in the index page at 5 or at any desired number
* Django-widget-tweaks for making customizable forms
* Sendrid backend as email backend is used to handle emails
* Requires integration with sendgrid account
* While sendgrid takes care of sending the email, the act of sending the email should be queued so it doesn’t block the request response cycle
* Celery and redis as a broker to take on the asynchronous task execution of sending emails for all auth outside the HTTP request-response cycle. This is helpful as sending email via SMTP or other third party packages can be slow. Check adapter.py and tasks.py. celery basically looks for the tasks.py and pushes a message to redis and celery pulls out it out and perform it
* A send email function that takes in a emailmultipart object as argument is define in the tasks.py and celery discovers it. The send\_mail function in django all auth adapter is overridden to use the send email function from tasks.py. when triggered, the task is queued to redis and celery pulls it out to perform it.
* Redis is also used to maintain cache. Either locally or in cloud.
* Whitenoise serves static files for the time being, basically lets gunicorn serve files

We have two ways to make the app serve through https, by using certs and using the runserver plus command, or by using a nginx server and making it perform a reverse proxy

Besides handling secure connection, we have to carefully separate the project into being configurable in development and production mode

In this project, we need SSL certificates. We have used mcert self signed certs

# Run-server plus using mcert files

Steps to Run with Elevated Privileges:

1. **Open an Elevated Command Prompt**:
   * Click on the **Start** menu, type cmd or powershell.
   * Right-click on **Command Prompt** or **PowerShell** and select **Run as administrator**.
2. **Install mkcert**:
   * Once the elevated Command Prompt or PowerShell window is open, run the following command:

bash

CopyEdit

choco install mkcert

then in the terminal

mkcert -install

mkcert example.com "\*.example.com" example.test localhost 127.0.0.1 ::1

we will see creation of two files

1.example.com+5.pem

2.example.com+5-key

Then to run the app, we type this

python manage.py runserver\_plus --cert-file example.com+5.pem --key-file example.com+5-key.pem

<https://github.com/FiloSottile/mkcert?tab=readme-ov-file>

Making your app comply with X’s requirements to use its API needs some careful work.

# Dockerizing the app

For best performance and ease, use wsl to run docker.

* We use docker-compose
* Docker runs as a non root user
* Docker must run in a linux environment or if on Windows, inside a WSL environment
* Containers are created for services, django-web, celery, redis, database, nginx. For production, nginx and database are not required
* Nginx and gunicorn must be set up inside docker
* We will run the application with gunicorn because that’s what we do for production. It also supports multithreading among other benefits

There are two conditions, one, we don’t need nginx when we use PaaS, digital oceans app because it handles TLS and SSL termination.? Two, we use nginx and keep it configured in our yaml file. In that case we need the ssl certs if we want our app to be served over https

* For https, we need to use ssl certs. We have volume mounted the certs inside a volume we created in docker. Docker runs in a linux environment and therefore store the certs in etc/nginx/ssl/
* In rare cases of issues, you can have both django and nginx handling HTTPS, which causes a redirect loop

The code is designed to run in two modes, production and development. Both modes happen with the app serving over HTTPS.

Lets start with the dev mode which is actually a semi dev mode. We have

* Docker-compose
* Fully configured nginx handling SSL termination and serves static and media
* Gunicorn
* Local database in container
* Local redis in container
* Celery in container
* Smtp for email backend
* Local storage for media
* Local storage for ckeditor media

In full prod mode, the database will be connected to a cloud and so will be the redis. Hence the compose file will reflect

The env file contains all the values required to make the right configurations for running the app in dev mode. It can also be set up for production like dev mode. To keep it at development mode, ENIVRONMENT\_DEVELOPMENT must be True

# environment variables for mode

ENVIRONMENT\_DEVELOPMENT=True

USE\_SPACES=False

# django secret

SECRET\_KEY=your-secret-key

# local postgresql database variables

DB\_NAME=name-of-your-database

DB\_USER=postgres

DB\_PASSWORD=your-password

DB\_HOST=db #we change to db  from localhost so it would match the database name in docker-compose

DB\_PORT=5432

# Docker postgresql database variables

DJANGO\_LOGLEVEL=info

DJANGO\_ALLOWED\_HOSTS=localhost

DATABASE\_ENGINE=postgresql\_psycopg2

DATABASE\_HOST=db

DATABASE=postgres

# Google secrets for django-allauth

GOOGLE\_CLIENT\_ID=your-secret

GOOGLE\_CLIENT\_SECRET=your-secret

TWITTER\_API\_KEY=your-secret

TWITTER\_API\_KEY\_SECRET=your-secret

TWITTER\_BEARER\_TOKEN=your-secret

TWITTER\_ACCESS\_TOKEN=your-secret

TWITTER\_ACCESS\_TOKEN\_SECRET=your-secret

# Twitter OAuth2 secrets for django-allauth

TWITTER\_0AUTH2\_CLIENT\_ID=your-secret

TWITTER\_OAUTH2\_CLIENT\_SECRET=your-secret

# email variables

GMAIL\_PASSWORD\_SMTP=your-secret

# sendgrid

SENDGRID\_API\_KEY=your-secret

SENDGRID\_FROM\_EMAIL=your-email

DEFAULT\_FROM\_EMAIL=your-email

# Online database configuration

DATABASE\_RENDER\_PASSWORD=online-db-password

DATABASE\_RENDER\_NAME=online-db-name

DATABASE\_RENDER\_USER=online-db-secret

DATABASE\_RENDER\_HOST=online-db-secret

DATABASE\_RENDER\_PORT=5432

DATABASE\_RENDER\_URL=online-db-url

# Redis configuration. redis in render. we use rediss. emphasis on secure

REDISS\_URL=online-rediss-url

# settings for DO SPACES for storage

AWS\_ACCESS\_KEY\_ID=secret

AWS\_SECRET\_ACCESS\_KEY=secret

AWS\_STORAGE\_BUCKET\_NAME=secret

AWS\_S3\_ENDPOINT\_URL=secret-url

DO\_SPACES\_CDN\_END\_POINT=secret-url

Lets check on the development mode.

To begin start with these commands

1. = docker volume create nginx\_ssl
2. = docker run --rm -v nginx\_ssl:/mnt -v "/$(pwd):/host" busybox cp /host/example.com+5.pem /host/example.com+5-key.pem /mnt/
3. = docker-compose -f docker\_new\_compose.dev.yml up --build
4. = docker-compose exec django-web python manage.py migrate
5. = docker-compose exec --user root django-web chown -R appuser:appuser /app/staticfiles
6. = docker-compose exec --user root django-web chown -R appuser:appuser /app/media
7. = docker-compose exec django-web python manage.py collectstatic

Step 1. Creates a persistent volume in docker

Step2. Mounts the volume in the working directory in docker and copies the SSL certs. These certs will persist even after the containers are spun down

Step3. Spin up your containers with the specific development compose files

Step4. Migrate your models

Step5/6. Give permissions to your non root user to manage these subdirectories

Step7. Collects your static

To wind your docker containers down

-docker rm $(docker ps -aq)

to remove all your images

-docker rmi $(docker images -q)

This is to remove your persistent volumes

-docker volume rm $(docker volume ls -q)

If your volumes are still running, then you can begin with 4 and skip 6 and 7.

Set up your postgresql. Have pgadmin greatly aids managing your database