Production Deployment Guide

# 1. Local Setup

The first step is to get things ready locally so they are deployable. This means the container, data setup and environment settings should be set up.

## 1.1 Single Docker Image

Create a single docker container for both backend & frontend. Use a [multistage docker build](https://docs.docker.com/build/building/multi-stage/) and [Whitenoise](https://whitenoise.readthedocs.io/en/latest/) to bundle and serve the frontend along with the backend.

Whitenoise is for Python. Express (and other node backends) would have a similar solution.

| **Configuring a Single Container**  You can create a single container with the following [docker-file](https://github.com/Boot41/sample-fse/blob/main/Dockerfile). It first builds the frontend in the first stage and then copies the code into the second container (Django server).  The Django server is set up to use [Whitenoise](https://whitenoise.readthedocs.io/en/latest/) in [settings.py](https://github.com/Boot41/sample-fse/blob/main/server/core/settings.py#L46C50-L47C1) to render the frontend files.  **# Stage 1 - Builds Frontend**  from node:22 as client\_build  WORKDIR /code  COPY ./client /code  RUN npm install  RUN npm run build  **# Stage 2 - Uses Frontend Files**  from python:3.12.3  WORKDIR /code  COPY server/requirements.txt /code/requirements.txt  RUN pip install gunicorn  RUN pip install -r requirements.txt  COPY --from=client\_build /code/build/static/ /code/static/  COPY --from=client\_build /code/build/ /code/static/  COPY ./server /code  CMD ["gunicorn" , "-b", "0.0.0.0:8000", "core.wsgi:application"] |
| --- |

## 1.2 Configuration & Settings

All config and secrets should be externalised into env properties. Make sure your docker container uses settings from env properties using [dotenv](https://pypi.org/project/python-dotenv/).

Here are some of the expected ones (you may have less or more).

* GOOGLE\_OAUTH\_CLIENT\_ID
* GOOGLE\_OAUTH\_CLIENT\_SECRET
* JWT\_SECRET\_KEY
* DJANGO\_SECRET\_KEY
* OPENAI\_BASE\_URL
* OPENAI\_API\_KEY
* DB\_ENGINE
* DB\_NAME
* DB\_PASSWORD
* DB\_HOST
* DB\_PORT

## 1.3 Database & Demo Data

For databases, just use sqlite, in a specific folder to avoid any external dependencies. Your docker command should also support a migrate command for any schema migrations.

For Django, your build step can run the following command to ensure the container is ready with all initial data and schema needed for the demo already set up. Make sure all data you need for the demo is already stored as fixtures and committed to github.

# Can be part of build Dockerfile

python manage.py migrate

python manage.py loaddata <fixture1> <fixture2>

## 1.4 Testing & Verification

For local readiness, test the image with the following.

docker run --env-file .env -p 8000:8000 <image>:<version>

Check your React UI, APIs and Django Admin are all working.

PS: For Django Admin, you would need to ensure createsuperuser is run (or a superuser is created as part of a fixture).

# 2. Cloud Deployment

We should begin with installing all the required libraries, then configuration and finally deployment.

## 2.1 Installing CLI tools

Docker should already be installed and running. Following needs to be installed

* **Google Cloud CLI**  
  <https://cloud.google.com/sdk/docs/install-sdk>
* **Terraform**<https://developer.hashicorp.com/terraform/install>

## 2.2 Authentication and Permissions

# select boot41 project and asia-south1 region

gcloud init

gcloud auth configure-docker asia-south1-docker.pkg.dev

gcloud auth application-default login

terraform init

## 2.3 Building the Docker Image

docker build . -t asia-south1-docker.pkg.dev/boot41/a3/**<project-name>**

## 2.4 Push the Docker Image to Google Cloud

docker push asia-south1-docker.pkg.dev/boot41/a3/**<project-name>**

## 2.5 Setup “Infra as code” for Google Cloud Run

* Create a cloudrun.tf at the etc/deploy folder. For a sample, look at  
  <https://github.com/Boot41/sample-fse/tree/main/etc/deploy>
* Create terraform.tfvars for your settings. Update it with project and docker image as well as environment variables to be used.

## 2.6 Apply and Deploy the latest container

* Initialise and apply the script  
  terraform apply
* If it succeeds, it would output the url e.g.  
  service\_url = "https://sample-fse-mha4s7stfa-el.a.run.app

Your deployment should now be working.

## 2.7 Update & Deploy Latest

Everytime, new code needs to be deployed, just

**# Re build the image**  
docker build . -t asia-south1-docker.pkg.dev/boot41/a3/**<project-name>**

**# Push the image to Container Repo**  
docker push asia-south1-docker.pkg.dev/boot41/a3/**<project-name>**

**# Apply the latest version again**

terraform apply

Terraform engine applies whatever is configured in .tfvars. So if you want to deploy a specific version of the image, not the latest, update tfvars with that and run apply.

# 3. Production Checklist

Once in production, here are some best practices to ensure.

1. **Django DEBUG**Django DEBUG=True is a huge security risk. It's good for debugging, but hackers can compromise your production servers. Ensure its set to false on production.
2. **Strong Passwords**Use strong passwords (not admin/password) for production.
3. **Data Management**We are using an embedded database. So, when you deploy again, the old one would be lost. Make sure your demo related data is part of fixtures which run on every build.
4. **No Open URLs**Your APIs should have authentication. Otherwise data çan be accessed and leaked by anyone.
5. **URL & API settings**For production, your URLs would be slightly different as they will all be running on a single port. Keep that in mind and use settings to manage that. One way to do it is use two different settings in frontend based on window.host (which would be either localhost, 127.0.0.1 or something like sample-fse-mha4s7stfa-el.a.run.app