ChatGPT

Развёртывание Django-приложения в Docker (Production)

Введение

В этом руководстве описывается пошаговое развёртывание Django-приложения в продуктивной среде с использованием **Docker** и **Docker Compose**. Проект будет состоять из двух контейнеров: один для веб-приложения Django, второй – для базы данных **PostgreSQL**. Статические файлы будут обслуживаться самим Django через пакет **WhiteNoise**, что упрощает деплой за счёт отсутствия внешнего веб-сервера для статики [1]. Мы будем использовать актуальные стабильные версии Python и PostgreSQL (на момент написания – Python 3.x и PostgreSQL 15). В качестве примера настроек базы данных используется БД **Advisor** с пользователем/паролем *postgres/postgres* (в реальном продакшене следует задать надёжный пароль). Ниже представлены все необходимые конфигурации: **Dockerfile**, **docker-compose.yml**, настройки Django (включая WhiteNoise), переменные окружения, а также команды для запуска и первоначальной настройки приложения.

Dockerfile для Django-приложения

Начнём с создания файла `Dockerfile` для сборки образа Django-приложения. В образ будут включены исходный код проекта, зависимости, а также статические файлы, собранные командой `collectstatic`. Ниже приведён пример Dockerfile:

```
# Используем официальный базовый образ Python (slim-версия для меньшего размера)
FROM python:3.12-slim

# Устанавливаем рабочую директорию внутри контейнера
WORKDIR /app

# Устанавливаем переменные окружения для Python
ENV PYTHONDONTWRITEBYTECODE=1  # не создавать pyc-файлы
ENV PYTHONUNBUFFERED=1        # вывод Python не буферизуется (сразу в лог)

# Устанавливаем зависимости Python
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Копируем исходный код проекта в образ
COPY . .

# Сбор статических файлов (для WhiteNoise)
RUN python manage.py collectstatic --noinput

# Открываем порт 8000 для входящих подключений
```

```
EXPOSE 8000

# Запуск приложения с Gunicorn (WSGI-сервер) на порту 8000
CMD ["gunicorn", "myproject.wsgi:application", "--bind", "0.0.0.0:8000"]
```

Пояснения к Dockerfile:
- Используется легковесный базовый образ `python:3.12-slim` (Debian-based) с последней версией Python. Это минимизирует размер конечного образа.
- Рабочая директория устанавливается в `/app` – все последующие команды будут выполняться там.
- Переменные окружения `PYTHONDONTWRITEBYTECODE` и `PYTHONUNBUFFERED` улучшают работу интерпретатора: отключается создание файлов `.pyc` и вывод логируется сразу без буферизации.
- Файл `requirements.txt` копируется и все зависимости (включая Django, WhiteNoise и Gunicorn) устанавливаются командой `pip install`. **Важно:** убедитесь, что в `requirements.txt` перечислены `whitenoise` и `gunicorn` [2], а также драйвер БД `psycopg2-binary` для PostgreSQL.
- Затем копируется весь код проекта. Предполагается, что рядом с Dockerfile находится ваш Django-проект (управляющий скрипт `manage.py` и т.д.).
- Команда `python manage.py collectstatic --noinput` собирает все статические файлы проекта в одну директорию (определённую как `STATIC_ROOT` в настройках Django). Это необходимо для того, чтобы WhiteNoise мог эффективно раздавать статику [3]. Все статические файлы будут включены в образ на этапе сборки.
- Команда `EXPOSE 8000` документирует, что контейнер прослушивает порт 8000.
- Наконец, задаётся команда запуска контейнера: здесь используется **Gunicorn** – надёжный WSGI-сервер для Python. Django-приложение запускается через Gunicorn, обращаясь к WSGI-приложению `myproject.wsgi:application` (замените `myproject` на имя вашего проекта). Gunicorn привязывается ко всем интерфейсам контейнера на порту 8000. В production-среде **не следует** использовать встроенный `runserver` Django; Gunicorn обеспечит лучшее управление подключениями и производительность.

Конфигурация docker-compose.yml

Создадим файл `docker-compose.yml`, описывающий два сервиса: **web** (контейнер с нашим Django-приложением) и **db** (контейнер PostgreSQL). Compose позволит запустить их одной командой и обеспечить сеть между ними. Ниже пример конфигурации:

```
version: '3.9'

services:
  db:
    image: postgres:15
    environment:
      POSTGRES_DB: Advisor           # Имя создаваемой БД
      POSTGRES_USER: postgres        # Имя пользователя БД
      POSTGRES_PASSWORD: postgres    # Пароль пользователя
    volumes:
      - postgres_data:/var/lib/postgresql/data
```

```
  web:
    build: .                        # Собираем образ из Dockerfile в текущей
директории
    command: gunicorn myproject.wsgi:application --bind 0.0.0.0:8000
    env_file: .env                  # Файл с переменными окружения для Django
    depends_on:
      - db
    ports:
      - "8000:8000"
    # restart: always               # (Опционально) автоматически
перезапускать контейнер при сбоях

volumes:
  postgres_data:
```

Что делает этот docker-compose.yml:
- **db:** использует официальный образ PostgreSQL (версии 15). Через переменные окружения задаются имя БД, пользователь и пароль. При первом запуске контейнер инициализирует базу данных *Advisor* и пользователя *postgres* с указанным паролем. Данные БД сохраняются на volume `postgres_data`, чтобы не потеряться при пересоздании контейнера. (Volume монтируется в `/var/lib/postgresql/data`, где Postgres хранит данные.)
- **web:** билдит образ нашего Django-приложения на основе Dockerfile (директива `build: .`). Команда запуска переопределяется на использование Gunicorn (аналогично CMD в Dockerfile) – это пример, можно оставить как в Dockerfile. С помощью `depends_on` гарантируется, что контейнер *db* будет запущен перед *web*. Порт 8000 контейнера проксируется на порт 8000 хоста, чтобы приложение было доступно по адресу `http://localhost:8000`. Файл `.env` подключается через `env_file` – в нём мы определим необходимые переменные окружения для Django (SECRET_KEY, настройки БД и др.).

Обратите внимание, что **порт 5432 для Postgres не проброшен наружу** (директивы `ports` для db нет). Это сделано специально: база данных будет доступна только изнутри docker-сети (сервиса web), что повышает безопасность. Django-приложение будет подключаться к БД по внутреннему хосту `db` (имя сервиса выступает как hostname). Внешний доступ к базе не требуется в данном сценарии. Если понадобится подключаться к Postgres извне (например, для отладки), можно добавить `"5432:5432"` в секцию ports сервиса db, но в продакшене этого лучше избегать.

Настройка WhiteNoise для статических файлов

WhiteNoise позволит нашему Django-контейнеру эффективно раздавать статические файлы прямо через Gunicorn, без отдельного веб-сервера. Настроим его следующим образом:

- **Установка WhiteNoise:** пакет должен быть добавлен в зависимости (в `requirements.txt`) и установлен при сборке образа (`pip install whitenoise`). Если вы ещё этого не сделали, убедитесь, что WhiteNoise установлен (наряду с `gunicorn`) [2].
- **Настройка STATIC_ROOT:** В файле `settings.py` укажите переменную `STATIC_ROOT` – путь, куда командой `collectstatic` будут собраны статические файлы. Как правило, это директория внутри проекта, например:

```
STATIC_ROOT = BASE_DIR / "staticfiles"
```

(здесь `BASE_DIR` – корневая директория проекта). Django при выполнении `collectstatic` скопирует туда все статические файлы из ваших приложений и из папки `STATICFILES_DIRS` [3]. Мы уже включили выполнение `collectstatic` на этапе сборки Docker-образа, поэтому при запуске контейнера все статические файлы будут готовы к раздаче.

- **Middleware WhiteNoise:** Добавьте WhiteNoise в список посредников (`MIDDLEWARE`) в `settings.py`. Он должен быть размещён сразу после `SecurityMiddleware`:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    # ... остальные middleware ...
]
```

Такой порядок обязателен – WhiteNoise должен идти перед другими промежуточными слоями, которые могут обрабатывать запросы к статике [4]. После этого WhiteNoise автоматически перехватит запросы к файлам в `/static/` и будет обслуживать их напрямую из собранной папки `STATIC_ROOT`.

- **Настройка WhiteNoise для лучшей производительности:** Рекомендуется включить сжатие и хеширование имен статических файлов, что позволит эффективно кэшировать их в браузерах. Для этого используем класс хранения статики от WhiteNoise. В Django 4.x можно указать в `settings.py`:

```
STATICFILES_STORAGE =
'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

Этот бекенд статических файлов будет автоматически сохранять сжатые версии файлов (gzip, Brotli) и давать им уникальные имена на основе содержимого (хеш) [5]. Благодаря этому, статические файлы могут кэшироваться браузером на длительное время (так как имя меняется при изменении файла). *Примечание:* В Django 4.2+ вместо `STATICFILES_STORAGE` может использоваться настройка `STORAGES` – с аналогичным указанием бекенда WhiteNoise.

- **STATIC_URL:** Убедитесь, что переменная `STATIC_URL` установлена, например: `STATIC_URL = '/static/'`. Это URL-префикс для всех статических файлов. WhiteNoise будет отвечать на запросы по этому префиксу. В шаблонах Django обращайтесь к статике через `{% static 'path/to/file.css' %}`.

После этих настроек ваш Django-приложение в режиме DEBUG=False сможет самостоятельно раздавать собранные статические файлы. WhiteNoise также автоматически устанавливает правильные заголовки кэширования и поддерживает сжатие контента, что улучшает производительность отдачи статики [6] [7].

Конфигурация настроек Django (settings.py)

Для корректной работы в Docker-продакшене нужно внести изменения в файл `settings.py` вашего Django-проекта. Основные моменты: отключение режима отладки, настройка допустимых хостов, секретный ключ, параметры подключения к базе данных из переменных окружения, а также параметры для статических файлов. Рассмотрим по порядку.

1. Режим отладки и ключ безопасности:
В продакшене **DEBUG** должен быть выключен, а **SECRET_KEY** храниться вне кода. Зададим их через переменные окружения, чтобы не хардкодить значения в репозитории [8]. В начале `settings.py` импортируйте модуль os:

```
import os
```

Далее настройте:

```
SECRET_KEY = os.environ.get('SECRET_KEY', 'unsafe-secret-key')
DEBUG = os.environ.get('DEBUG', 'False').lower() == 'true'
```

Здесь мы пытаемся получить значения из переменных окружения. Если переменная `SECRET_KEY` не задана, используется небезопасный дефолт (не забудьте поменять на реальный ключ). Для `DEBUG` читаем строку и сравниваем с `"true"` – так значение `'False'` превратится в `False`. Теперь поведение приложения управляется внешними параметрами: не нужно менять код, чтобы включить/отключить отладку или сменить ключ.

2. Разрешённые хосты (ALLOWED_HOSTS):
Когда DEBUG=False, Django **требует** явного указания допустимых хостов, иначе запросы не будут обслуживаться (ошибка Bad Request 400) [9]. Настройте:

```
ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')
```

Переменная окружения `ALLOWED_HOSTS` должна содержать список ваших доменов/хостов через запятую. Например: `"mydomain.com,localhost,127.0.0.1"`. В контейнере для простоты можно разрешить локальные адреса. **Важно:** в продакшене не оставляйте ALLOWED_HOSTS пустым и не используйте `['*']` без необходимости – укажите конкретный домен или IP вашего сервера.

3. Настройки базы данных:
Подключение к PostgreSQL в Docker-среде настраивается через host `db` (имя сервиса). Мы уже задали переменные в `.env` /Compose для имени БД, пользователя и пароля. Используем их в настройках:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB', 'Advisor'),
```

```
        'USER': os.environ.get('POSTGRES_USER', 'postgres'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD', 'postgres'),
        'HOST': os.environ.get('DB_HOST', 'db'),
        'PORT': os.environ.get('DB_PORT', '5432'),
    }
}
```

Здесь вместо фиксированных значений используются `os.environ.get(...)` – это позволяет задать конфигурацию через окружение контейнера [10]. Значения по умолчанию (`Advisor`, `postgres` и т.д.) указаны для удобства, но в реальной среде они будут переопределены вашими переменными. Обратите внимание: `HOST` по умолчанию `db` – то самое имя сервиса базы данных в Docker Compose. Django внутри контейнера «видит» сервис Postgres по этому хостнейму.

4. Статические файлы:
Эти настройки мы отчасти уже рассмотрели:

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / 'staticfiles'
```

Убедитесь, что `INSTALLED_APPS` содержит `'django.contrib.staticfiles'` (по умолчанию есть в свежем проекте). Благодаря ранее добавленному WhiteNoiseMiddleware, при DEBUG=False статика будет обслуживаться из `STATIC_ROOT`. Если вы применили рекомендацию по WhiteNoise Storage, добавьте:

```
STATICFILES_STORAGE =
'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

(либо эквивалент через `STORAGES` в новых версиях Django).

Приведённые изменения в `settings.py` помогут сделать проект конфигурируемым и безопасным для запуска в Docker. Настройки больше не содержат чувствительных данных (они приходят из окружения), и параметры, зависящие от среды, можно менять без изменения кода [8] [11].

Переменные окружения (.env)

Создайте файл **.env** в корне проекта (рядом с docker-compose.yml). В этот файл запишем все секретные и средовые переменные, которые будут загружены в контейнеры. Docker Compose автоматически может подхватывать .env или можно явно указать его, как мы сделали в настройках сервиса web. Вот пример содержания `.env`:

```
# .env - примеры переменных окружения

# Настройки Django
SECRET_KEY=django-insecure-0123456789abcdefghijk   # секретный ключ Django
DEBUG=False                                        # отключаем режим отладки
```

```
ALLOWED_HOSTS=127.0.0.1,localhost
# разрешённые хосты (через запятую)

# Настройки базы данных (PostgreSQL)
POSTGRES_DB=Advisor
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
DB_HOST=db
DB_PORT=5432
```

Что означают эти переменные:
- `SECRET_KEY` – секретный ключ приложения Django. В продакшене **обязательно** должен быть уникальным и секретным. Никогда не храните его в публичном репозитории. В примере указан сгенерированный шаблон, замените на свой.
- `DEBUG` – флаг отладки. Значение `False` (или `0`) означает, что отладка выключена. Мы обрабатываем строковые значения в настройках, поэтому можно использовать `False/True` или `0/1`. В production всегда должно быть False!
- `ALLOWED_HOSTS` – список адресов, по которым ваше приложение доступно. Здесь указаны localhost и 127.0.0.1 (например, для тестирования локально). В боевой среде впишите сюда доменное имя вашего сайта или IP сервера. Это защищает от HTTP Host header атак и является требованием Django при DEBUG=False [9].
- `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD` – параметры для инициализации базы данных PostgreSQL. Они же будут использованы приложением для подключения к БД. **Замечание:** Пароль `postgres` здесь приведён для простоты настройки, но в реальной системе используйте сложный пароль для БД. Добавьте при необходимости другие переменные (например, отдельное имя пользователя для приложения).
- `DB_HOST` и `DB_PORT` – адрес и порт для подключения к базе данных. В контейнерной сети хостом является имя сервиса `db` (что Docker Compose пропишет в DNS), порт стандартный 5432. Эти значения совпадают с настройками по умолчанию в нашем `settings.py`, так что их можно и не дублировать, но явное указание не повредит.

Файл `.env` должен быть добавлен в `.gitignore`, чтобы не попасть в систему контроля версий (особенно из-за `SECRET_KEY` и паролей). Для коллег можно создать файл-шаблон `.env.example` с перечислением переменных без реальных секретов [12]. Использование `.env` и `env_file` в Compose делает конфигурацию гибкой: вы можете иметь разные файлы окружения для dev/stage/prod, не меняя код приложения и compose-файлы [13] [14].

Запуск и первоначальная настройка контейнеров

После того как Dockerfile, docker-compose.yml, настройки Django и .env-файл подготовлены, можно запускать приложение. Ниже приведены основные команды для развёртывания и первоначальной настройки:

1. **Сборка и запуск контейнеров:**
   Выполните команду:

   ```
   docker-compose up -d --build
   ```

Опция `--build` принудительно соберёт образ по Dockerfile (если вы не делали этого ранее). Ключ `-d` запускает контейнеры в фоновом режиме. Docker Compose прочитает переменные из `.env`, создаст сеть, контейнер с Postgres и контейнер с Django.

> *Примечание:* Первый запуск Postgres может занять несколько секунд (инициализация БД). Контейнер *web* имеет `depends_on: db`, поэтому он стартует почти сразу, но может попытаться подключиться к БД, которая ещё не готова. Как правило, Docker Compose ожидает успешный запуск сервиса db. Тем не менее, если приложение Django не сразу подключилось, оно может выдать ошибки в логах о подключении – в таком случае перезапустите контейнер web или используйте механизм healthcheck для надёжности.

2. **Применение миграций:**
   Когда оба контейнера запущены, нужно выполнить миграции базы данных (создать таблицы в новой БД *Advisor*). Используем команду:

   ```
   docker-compose exec web python manage.py migrate --noinput
   ```

   Эта команда выполнит `manage.py migrate` внутри запущенного контейнера *web*. Флаг `--noinput` отключает запросы интерактивного ввода (на случай подтверждений). После успешного выполнения в базе данных появятся необходимые таблицы. Если миграции прошли без ошибок, значит связь с базой установлена правильно. Если произошла ошибка подключения, убедитесь, что параметры в `settings.py` совпадают с окружением, и что контейнер *db* работает.

3. **Создание суперпользователя:**
   Чтобы попасть в админ-панель Django, потребуется суперпользователь. Его можно создать командой:

   ```
   docker-compose exec web python manage.py createsuperuser
   ```

   Вам будет предложено ввести имя пользователя, email и пароль для администратора. После этого учетная запись будет сохранена в базе. (Если предпочитаете, можно автоматизировать создание суперпользователя через скрипт или переменные окружения, но ручной способ через `createsuperuser` – самый простой.)

4. **Сбор статики (при необходимости):**
   Поскольку мы включили сбор статических файлов на этапе сборки образа (через Dockerfile), повторно выполнять `collectstatic` не требуется при каждом запуске. Однако при обновлении фронтенд-ресурсов (CSS/JS) не забудьте либо пересобрать образ, либо запустить миграцию статики вручную:

   ```
   docker-compose exec web python manage.py collectstatic --noinput
   ```

   В нашем случае это обычно не нужно до изменения кода, но стоит помнить о команде. WhiteNoise сразу начинает раздавать новые статические файлы после их появления в STATIC_ROOT.

После выполнения этих шагов ваше приложение должно работать на `http://localhost:8000/` (или на указанном в ALLOWED_HOSTS хосте). Проверьте, что вы можете войти в админ-панель Django (по адресу `/admin/`) с учётными данными суперпользователя. Логи контейнеров можно просмотреть командой `docker-compose logs -f` для отладки. При развёртывании на реальный сервер, вы можете настроить прокси-сервер (например, Nginx) или сетевой балансировщик перед контейнером web, но даже без него приложение будет обслуживать запросы (Gunicorn + WhiteNoise достаточно для начала).

Советы по безопасности и завершающие штрихи

Развёртывая Django-приложение в боевой (production) среде, убедитесь, что учитываете следующие моменты безопасности и оптимизации:

- **DEBUG=False обязательно:** Никогда не оставляйте `DEBUG = True` в production – это может раскрыть конфиденциальную информацию при ошибках и даже дать доступ к отладочной консоли. Как отмечает документация, при DEBUG=False необходимо правильно установить ALLOWED_HOSTS, иначе Django отвергнет все запросы [9]. Мы уже сделали DEBUG управляемым через переменные окружения – на продакшене установите его в False (что мы и сделали в .env).

- **SECRET_KEY:** Храните секретный ключ вне репозитория (в переменной окружения, либо секрет-хранилище). Значение по умолчанию в коде должно использоваться **только** для разработки. В нашем примере мы подставляем `unsafe-secret-key` если переменная не задана – это подсказка, что нужно определить SECRET_KEY снаружи. Настоящий ключ должен быть длинной случайной строкой. Не используйте один и тот же SECRET_KEY на разных проектах.

- **Пароли и доступы БД:** Хотя в примере используется пароль "postgres" для простоты, в реальной среде задайте сложный пароль для пользователя базы данных. Также, можно создать отдельного БД-пользователя с минимальными правами, специально для вашего приложения (чтобы не использовать суперпользователя postgres). Ограничьте сеть – мы не публиковали порт Postgres наружу, и это правильно: только ваше приложение должно обращаться к БД. При необходимости внешнего доступа используйте VPN или специализированные средства, а не открытый порт.

- **ALLOWED_HOSTS:** Настройте допустимые хосты максимально конкретно. Временно для отладки можно поставить `ALLOWED_HOSTS = ['*']`, но перед выпуском в продакшн обязательно указать реальные доменные имена или IP. Это предотвратит злоупотребления заголовком Host и обеспечит корректную работу Django на вашем домене.

- **Обработка статических и медиа файлов:** WhiteNoise подходит для раздачи статических файлов при относительно невысокой и средней нагрузке, упрощая конфигурацию деплоя [1]. Он поддерживает сжатие и кэширование, как мы включили. Однако, если объём трафика статических файлов большой, рассмотрите использование CDN или прокси-сервера для отдачи статики [15]. Аналогично, для пользовательских медиа-файлов (например, загруженных изображений) можно настроить отдельное хранилище (S3, облако) или отдавать их через Nginx. В нашем примере медиа не рассматривались, но учтите этот момент при необходимости.

- **Безопасность контейнеров:** Убедитесь, что используете актуальные образы (мы указали конкретную версию `postgres:15` – обновляйте при выходе новых релизов). Следите за обновлениями безопасности в ваших зависимостях (`pip install --no-cache-dir -r requirements.txt` всегда берёт свежие версии, но фиксируйте версии в requirements.txt для предсказуемости). Можно запускать контейнер приложения от непривилегированного пользователя (в Dockerfile можно создать пользователя и переключиться на него с помощью USER), чтобы уменьшить риски – хотя по умолчанию Gunicorn запускается от root внутри контейнера, лучше изолировать права. Также не забудьте про **SECRET_KEY** и другие секреты: в логах приложения (и Docker) они не должны светиться.

- **Логирование и мониторинг:** В продакшене настройте более подробное логирование (например, вывод Gunicorn логов, настройки `LOGGING` в Django) и систему мониторинга контейнеров. Это выходит за рамки базовой инструкции, но важно для поддержания приложения. Docker Compose позволяет добавлять рестарт-политику (`restart: always`) для авто-перезапуска, а также healthcheck-команды, чтобы отслеживать живость сервисов.

Следуя этой инструкции, вы получите изолированную и переносимую инфраструктуру для Django-приложения. Использование Docker Compose и .env-файлов делает конфигурацию гибкой – можно легко менять параметры без перекомпиляции кода [11]. Контейнеризация с разделением на веб-приложение и базу данных упрощает масштабирование и обслуживание. **WhiteNoise** обеспечивает удобную раздачу статических файлов прямо приложением, что сокращает количество движущихся частей в вашей архитектуре [16].

Теперь у вас есть базовый шаблон для развёртывания Django в Docker. В дальнейшем вы можете улучшать его – например, добавить сервис для фоновых задач (Celery + Redis), подключить Nginx как реверс-прокси для управления SSL, или использовать оркестрацию (Docker Swarm, Kubernetes) для более сложных сценариев. Но даже в простом варианте, описанном здесь, ваше приложение будет работать в докере стабильно и безопасно. Успехов в деплое!

Ссылки на документацию и материалы:
- Официальное руководство по WhiteNoise [3] [4] – настройка статических файлов в Django для production.
- Документация WhiteNoise о сжатии и хешировании статики [5].
- Статья *"Making Your Django Project Production-Ready with WhiteNoise and Gunicorn"* [16] [2] – преимущества WhiteNoise и рекомендация установки Gunicorn.
- Рекомендации по использованию переменных окружения в Django-проектах [8] (BetterStack, 2025) – хранение секретов вне кода.
- Официальная документация Django: настройка DEBUG и ALLOWED_HOSTS [9] (требование ALLOWED_HOSTS при DEBUG=False).
- Пример конфигурации Django + Postgres в Docker (TestDriven.io) [10] – иллюстрация использования переменных окружения для параметров базы данных.

---

1 2 6 7 15 16 Make a Django project production-ready, create a Docker Image and use GitHub CI/CD to automate the push of the Docker image - DEV Community
https://dev.to/doridoro/make-a-django-project-production-ready-create-a-docker-image-and-use-github-cicd-to-automate-the-push-of-the-docker-image-fm3

3 4 5 Using WhiteNoise with Django - WhiteNoise 6.9.0 documentation
https://whitenoise.readthedocs.io/en/stable/django.html

8 11 12 13 14 Django Docker Best Practices: 7 Dos and Don'ts | Better Stack Community
https://betterstack.com/community/guides/scaling-python/django-docker-best-practices/

9 Settings | Django documentation | Django
https://docs.djangoproject.com/en/5.2/ref/settings/

10 Dockerizing Django with Postgres, Gunicorn, and Nginx | TestDriven.io
https://testdriven.io/blog/dockerizing-django-with-postgres-gunicorn-and-nginx/