Контейнеризация Django-проекта с помощью Docker

Введение

Контейнеризация позволяет упаковать приложение вместе со всеми зависимостями в легковесные изолированные контейнеры. Docker описывает контейнер как «изолированный процесс с набором всех нужных файлов» (What is a container? | Docker Docs), благодаря чему каждый компонент приложения (веб-сервер, база данных и т.д.) работает независимо друг от друга. Это обеспечивает **изоляцию** и **повторяемость** среды: контейнер, запускаемый на одной машине, будет вести себя так же на любой другой (портативность) (What is a container? | Docker Docs). К тому же каждый контейнер делает одну конкретную задачу (принцип «один контейнер – одна задача») (Multi-container applications | Docker Docs), что упрощает масштабирование и поддержку.

Для сложных приложений с несколькими сервисами (например, Django-приложение и СУБД) используется Docker Compose. Он позволяет описать **многоконтейнерное приложение одним YAML-файлом** (обычно `docker-compose.yml`) (Multi-container applications | Docker Docs): в нём задаются все сервисы, их зависимости, настройки сети и тома для хранения данных. При помощи Docker и Docker Compose разработчики получают единое воспроизводимое окружение для разработки и продакшена, избавляясь от проблемы «работает на моей машине».

Разбор файлов проекта

Ниже рассмотрим каждый файл, необходимый для контейнеризации Django-проекта. Исходный код файла приводится полностью, после чего идет подробный разбор фрагмент за фрагментом.

Файл `create_admin.py`

```
#!/usr/bin/env python
import os
import django
from django.contrib.auth import get_user_model

# Указываем модуль настроек проекта (замените "myproject" на имя вашего проекта)
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "myproject.settings")
django.setup()

User = get_user_model()
# Создаем суперпользователя, если он еще не существует
if not User.objects.filter(username="admin").exists():
    User.objects.create_superuser("admin", "admin@example.com", "password")
    print("Superuser created.")
else:
    print("Superuser already exists.")
```

Объяснение: Этот скрипт запускается отдельно от `manage.py` и создает суперпользователя Django. Сначала устанавливаем переменную окружения `DJANGO_SETTINGS_MODULE`, чтобы Django знал, где искать настройки проекта, и вызываем `django.setup()` для инициализации конфигурации. Далее получаем модель пользователя через `get_user_model()`. Если пользователь с логином `admin` не найден, создаем его с заданными логином, email и паролем. Таким образом при старте контейнера автоматически обеспечивается наличие суперпользователя. Печатаем сообщения об успехе или о том, что суперпользователь уже есть. Этот файл не вызывается вручную — он запускается из `entrypoint.sh` после применения миграций.

Файл `wait-for-db.sh`

```
#!/bin/bash

# Ждем, пока база данных станет доступна.
# Предполагается, что заданы переменные окружения PG_HOST и PG_PORT.
while ! nc -z "$PG_HOST" "$PG_PORT"; do
  echo "Waiting for database connection at $PG_HOST:$PG_PORT..."
  sleep 1
done

echo "Database is up - continuing."
```

Объяснение: Этот скрипт проверяет доступность сервиса базы данных перед выполнением миграций и запуска Django. Он зациклено пытается установить TCP-соединение (команда `nc -z`) на хост и порт из переменных окружения `PG_HOST` и `PG_PORT`. Если соединение не устанавливается, скрипт ждет по секунде и пробует снова. Как только база данных начинает отвечать, выводится сообщение и скрипт завершается. Такой механизм ожидания помогает избежать ошибок вида “django.db.utils.OperationalError: could not connect to server” — похожее решение с «мониторингом запуска БД» рекомендуется использовать для отложенного запуска миграций (cannot set up docker it gives this error. Postgres and Django - Mystery Errors - Django Forum). Скрипт `wait-for-db.sh` вызывается из `entrypoint.sh` (см. ниже).

Файл `.dockerignore`

```
# Игнорируем окружение Python и кеш
venv/
env/
*.pyc
__pycache__/

# Игнорируем файлы базы данных и медиа (если не нужны в образе)
*.sqlite3
media/
staticfiles/

# Игнорируем настройки разработки и логи
.env
*.log
```

Объяснение: Файл `.dockerignore` определяет, какие файлы **не** будут отправляться в контекст сборки Docker (аналогично `.gitignore`) (Build context | Docker Docs). Здесь обычно исключают большие или ненужные файлы: виртуальное окружение (`venv/`, `env/`), скомпилированные файлы Python (`*.pyc`, `__pycache__/`), файлы локальной БД (`*.sqlite3`), а также файлы, содержащие секреты и локальные настройки (`.env`, логи и т.д.). Благодаря этому Docker не копирует в образ ненужные данные, а сборка проходит быстрее и итоговый образ меньше. Обратите внимание, что мы игнорируем `.env` — файл с секретными настройками будет использоваться только Docker Compose на этапе запуска, но не попадет внутрь образа.

Файл `docker-compose.yml`

```
version: "3.8"

services:
  db:
    image: postgres:13
    env_file:
      - .env
    environment:
      - POSTGRES_DB=${PG_NAME}
      - POSTGRES_USER=${PG_USER}
      - POSTGRES_PASSWORD=${PG_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data

  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db
    env_file:
      - .env

volumes:
  postgres_data:
```

Объяснение: Этот файл описывает два сервиса – базу данных (`db`) и веб-приложение (`web`) – и их параметры.

- Сервис **`db`** использует официальный образ PostgreSQL версии 13. Через секцию `env_file: - .env` подгружаются переменные из файла `.env` (ниже). Мы задаем переменные окружения PostgreSQL (`POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD`) через значения из `.env` (используются синтаксис `${...}`). Это гарантирует, что СУБД создаст базу данных и пользователя с нужными именами и паролем. Подключается том `postgres_data`, чтобы данные БД сохранялись на хосте и не терялись при перезапуске контейнера.

- Сервис **`web`** строится из локального `Dockerfile` (опция `build: .`), то есть образ собирается из исходников проекта. Команда запускает встроенный сервер Django на всех интерфейсах порта 8000. Папка проекта монтируется внутрь контейнера (опция `volumes: - .:/app`) — это удобно для разработки (можно править код на хосте без пересборки). Порт 8000 на хосте пробрасывается на контейнер. Опция `depends_on` гарантирует, что контейнеры запустятся в нужном порядке (сначала БД). Файл `.env` также подключается к контейнеру `web`, чтобы Django взял настройки БД и секреты из переменных окружения.

Важно: в настройках Django нужно указывать имя хоста БД (`PG_HOST`) совпадающим с именем сервиса (здесь `db`), а не `localhost`. Как отмечено в сообществе Django, неверный `PG_HOST` (например, `localhost` вместо `db`) приведет к ошибке подключения (cannot set up docker it gives this error. Postgres and Django - Mystery Errors - Django Forum). Поэтому в `.env` следует писать `PG_HOST=db`.

Файл `Dockerfile`

```
FROM python:3.12-slim

WORKDIR /app

RUN apt-get update && apt-get install -y netcat-openbsd gcc libpq-dev && apt-get
clean

COPY requirements.txt /app/
RUN pip install --upgrade pip && pip install -r requirements.txt

# Явно указываем файлы
COPY entrypoint.sh /app/
COPY wait-for-db.sh /app/
RUN chmod +x /app/entrypoint.sh /app/wait-for-db.sh

COPY . /app/

ENTRYPOINT ["/app/entrypoint.sh"]
```

Объяснение: Этот `Dockerfile` описывает процесс сборки образа для сервиса `web`.

- **FROM python:3.12-slim**
  За основу берётся официальный облегчённый образ Python версии 3.12 (на базе Debian). Slim-образ содержит минимальный набор пакетов, что уменьшает размер итогового контейнера.

- **WORKDIR /app**
  Создаёт директорию `/app` внутри контейнера и устанавливает её как текущую рабочую директорию. Все дальнейшие команды будут выполняться внутри этой папки.

- **RUN apt-get update && apt-get install -y netcat-openbsd gcc libpq-dev && apt-get clean**
  Выполняет установку системных зависимостей:
  - `netcat-openbsd` — необходимая утилита (`nc`) для проверки доступности базы данных в скрипте `wait-for-db.sh`.
  - `gcc` и `libpq-dev` — компилятор и библиотеки, необходимые для корректной установки Python-библиотек, таких как `psycopg` (клиент PostgreSQL). Команда `apt-get clean` используется для очистки кэша после установки, уменьшая размер образа.

- **COPY requirements.txt /app/**
  Копирует файл зависимостей Python (`requirements.txt`) в контейнер. Это позволяет сначала установить зависимости отдельно, что ускоряет пересборку образа при изменениях в коде.

- **RUN pip install --upgrade pip && pip install -r requirements.txt**
  Обновляет менеджер пакетов `pip` до последней версии и устанавливает все зависимости проекта, указанные в `requirements.txt`.

- **COPY entrypoint.sh /app/** и **COPY wait-for-db.sh /app/**
  Явно копирует скрипты запуска (`entrypoint.sh`) и ожидания базы данных (`wait-for-db.sh`) внутрь контейнера.

- **RUN chmod +x /app/entrypoint.sh /app/wait-for-db.sh**
  Делает эти скрипты исполняемыми, чтобы их можно было запустить напрямую.

- **COPY . /app/**
  Копирует весь исходный код проекта в контейнер. Благодаря файлу `.dockerignore`, ненужные файлы (например, виртуальное окружение, `.env`, файлы `.pyc`) будут проигнорированы и не попадут в образ.

- **ENTRYPOINT ["/app/entrypoint.sh"]**
  Устанавливает точкой входа (entrypoint) скрипт `entrypoint.sh`. Это означает, что при запуске контейнера первым будет выполняться именно этот скрипт. Внутри него уже происходит ожидание базы данных, миграции и запуск Django-приложения.

Таким образом, `Dockerfile` чётко определяет структуру образа, порядок установки зависимостей и последовательность запуска скриптов, обеспечивая стабильный запуск Django-приложения в контейнере.

Файл `entrypoint.sh`

```
#!/bin/sh
set -e

# Ожидаем доступности БД
/app/wait-for-db.sh db 5432

# Выполняем миграции
python manage.py migrate

# Создаем суперпользователя, если его нет
python /app/create_admin.py

# Запускаем Django (используя Gunicorn или runserver)
# exec gunicorn itg.wsgi:application --bind 0.0.0.0:8000
# или просто:
exec python manage.py runserver 0.0.0.0:8000
```

Объяснение: файл `entrypoint.sh` является **«точкой входа»** (entrypoint) для контейнера с Django-приложением. Это значит, что именно он будет выполнен при старте контейнера Docker. Рассмотрим каждую строку подробно:

- `#!/bin/sh`
  Обозначает, что скрипт выполняется через оболочку `sh` (shell).
- `set -e`
  Это важная настройка: скрипт будет немедленно остановлен, если любая из команд завершится ошибкой. Это предотвращает дальнейшее выполнение, если, например, не запустилась база данных или не применились миграции.
- `/app/wait-for-db.sh db 5432`
  Запускает скрипт `wait-for-db.sh`, передавая ему два параметра: хост (`db`) и порт (`5432`). Этот скрипт ждёт, пока не станет доступной база данных PostgreSQL (пока не появится возможность подключиться к указанному хосту и порту). Это необходимо, так как база данных может запускаться дольше, чем Django-приложение, и попытка применить миграции слишком рано приведёт к ошибке.
- `python manage.py migrate`
  Выполняет миграции Django, подготавливая структуру базы данных. После этого шага все таблицы базы данных, описанные в моделях Django, будут созданы и готовы к использованию.
- `python /app/create_admin.py`
  Запускает скрипт `create_admin.py`, который проверяет, существует ли суперпользователь, и если нет — создаёт его с указанными параметрами (имя пользователя, email, пароль). Это удобно для автоматического развёртывания, чтобы сразу иметь готового администратора для входа в админку Django.

- `exec python manage.py runserver 0.0.0.0:8000`
  Запускает встроенный веб-сервер Django, привязанный к адресу `0.0.0.0` на порту `8000`. Использование команды `exec` заменяет текущий процесс скрипта процессом веб-сервера Django, таким образом Django-приложение становится главным процессом контейнера. Это важно для корректной работы Docker, так как Docker отслеживает состояние именно этого главного процесса.

Также есть закомментированная строка:

```
# exec gunicorn itg.wsgi:application --bind 0.0.0.0:8000
```

Это альтернативный способ запуска приложения через сервер Gunicorn (обычно используемый в production-окружениях).

Таким образом при поднятии контейнера все подготовительные действия выполняются автоматически, и приложение стартует уже с примененными миграциями и готовым администратором.

Файл `.env`

```
# Режим отладки и секретный ключ Django
DEBUG=1
SECRET_KEY=your-secret-key

# Настройки базы данных
DB_NAME=postgres
DB_USER=postgres
DB_PASSWORD=postgres
DB_HOST=db
DB_PORT=5432

# Дополнительные настройки Django
ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
```

Объяснение: Файл `.env` хранит конфиденциальные переменные окружения. Здесь указаны настройки Django и БД:

- `DEBUG=1` включает режим отладки (для разработки). В продакшене нужно ставить `0`.
- `SECRET_KEY` – секретный ключ Django, его следует сгенерировать и хранить конфиденциально.
- Параметры `DB_NAME`, `DB_USER`, `DB_PASSWORD`, `DB_HOST`, `DB_PORT` определяют подключение к БД. В данном примере мы используем стандартные настройки PostgreSQL. Важно, что `DB_HOST=db` – имя сервиса БД в Docker Compose.
- `ALLOWED_HOSTS` задает список разрешенных доменов/хостов для Django.

В `docker-compose.yml` (`env_file`) и внутри контейнера (через `os.environ`) эти переменные подхватываются и используются в `settings.py` вашего проекта. Например, в коде проекта в `settings.py` нужно читать эти переменные (с помощью `os.getenv` или библиотеки `django-environ`).

Шаги по сборке и запуску

1. Перейдите в корневую папку проекта (где лежат `Dockerfile` и `docker-compose.yml`).
2. **Сборка образов:** выполните в терминале

   ```
   docker-compose build
   ```

   Это соберет образ для Django-приложения на основе Dockerfile.
3. **Запуск контейнеров:** затем запустите

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

   Флаг `-d` (detached) запустит контейнеры в фоновом режиме. Docker Compose автоматически поднимет сначала базу данных, затем веб-приложение. Благодаря скриптам внутри контейнера все миграции применятся, и суперпользователь будет создан.
4. **Проверка работы:** убедитесь, что контейнеры запущены командой

   ```
   docker-compose ps
   ```

   или

   ```
   docker ps
   ```

   После этого откройте браузер по адресу http://localhost:8000. Вы должны увидеть ваше Django-приложение. Зайдите на `/admin`, чтобы проверить доступность административной панели. Введите логин `admin` и пароль `password` (как в `create_admin.py`), чтобы убедиться, что суперпользователь создан.

> **Совет:** Если вы вносите изменения в код Django во время разработки, можно делать `docker-compose up --build`, чтобы пересобрать образ и перезапустить контейнеры с обновлённым кодом.

Типичные ошибки и их решения

- **Ошибка подключения к базе данных:** Чаще всего возникает, если Django не может достучаться до контейнера с БД. Проверьте, что в файле `.env` переменная `PG_HOST` соответствует имени сервиса базы (`db` в нашем примере). Если там стоит `localhost` или другое имя, соединение будет невозможным (cannot set up docker it gives this error. Postgres and Django - Mystery Errors - Django Forum). Убедитесь, что сервис `db` действительно запущен (через `docker-compose ps`) и что `wait-for-db.sh` успешно дождался порта. Иногда помогает перезапуск `docker-compose down && docker-compose up --build`.

- **Ошибка при создании суперпользователя:** Если при запуске вы не видите учетную запись `admin`, возможно, скрипт `create_admin.py` не выполнился или в нем указано не то имя проекта. Убедитесь, что в `create_admin.py` `os.environ.setdefault("DJANGO_SETTINGS_MODULE", "...")` указывает на правильный модуль настроек (имя вашего проекта вместо `myproject`). Также проверьте, что после первого запуска скрипт **не падает**, а в случае дубликата просто пишет, что пользователь уже существует. Не забывайте, что этот скрипт запускается автоматически из `entrypoint.sh` и не требует ручного вмешательства.

- **Проблемы с миграциями:** Если при старте контейнера видите ошибки вида «error checking a consistent migration history» или подобные, возможно, миграции пытаются примениться **до** того, как база поднялась. Это случается, если `wait-for-db.sh` не сработал (например, нет `nc`) или `depends_on` не успевает дождаться готовности. Решение – убедиться, что в `Dockerfile` установлен `netcat` (или другой инструмент проверки) и что `entrypoint.sh` выполняется после поднятия БД. В целом, как указал один из экспертов, стоит «ввести задержку или мониторинг, чтобы команда миграции ждала инициализации базы данных» (cannot set up docker it gives this error. Postgres and Django - Mystery Errors - Django Forum). Если же история миграций повреждена, можно попробовать удалить локальные файлы миграций (кроме `__init__.py`), заново выполнить `makemigrations` и `migrate`.

- **Другие проблемы:** Иногда пригодится посмотреть логи контейнера: `docker-compose logs web` и `docker-compose logs db`, чтобы найти подсказки. Если вы меняли версию Python или библиотеки, может потребоваться пересобрать образ. Всегда проверяйте, что файлы проекта актуальны и `.env` правильно настроен.

Таким образом, описанная структура файлов и команд позволяет полностью подготовить Django-приложение к работе в контейнерах Docker, устранив большинство проблем на старте. Убедитесь, что внимательно настроены пути и имена сервисов, и тогда развертывание пройдет гладко.

Источники: Официальная документация Docker по контейнерам и Compose (What is a container? | Docker Docs) (Multi-container applications | Docker Docs), а также обсуждения практик запуска Django в Docker (Build context | Docker Docs) (cannot set up docker it gives this error. Postgres and Django - Mystery Errors - Django Forum) (cannot set up docker it gives this error. Postgres and Django - Mystery Errors - Django Forum) помогают лучше понять логику и ошибки на каждом этапе.