ChatGPT

Введение

Перед нами задача регулярно переносить обновления Django-проекта и связанной базы данных PostgreSQL из локальной сети разработки (с доступом в Интернет) в изолированную локальную сеть (без доступа в Интернет). В исходной сети проект (включая приложение Django и СУБД PostgreSQL) работает в контейнерах Docker, причем данные базы хранятся в Docker volume. Предполагается, что данные в базе не критичны – при переносе можно ограничиться структурой базы (схемой таблиц, возможно с минимальным наполнением). Для передачи артефактов из одной сети в другую допускается использование физического носителя (например, USB-накопителя).

Наша цель – выработать **повторяемый процесс деплоя**, позволяющий регулярно переносить новые версии Django-приложения и актуальную структуру базы данных с среды разработки в изолированную продакшен-среду. На стороне продакшена доступны любые необходимые ресурсы: можно использовать Docker-контейнеры, виртуальные машины или разворачивать проект на любой ОС вручную. Ниже представлены инструкции и рекомендации, разбитые по этапам, с рассмотрением двух основных подходов к переносу: **с использованием Docker** и **без использования Docker**. Отчет структурирован по ключевым аспектам задачи, включая подготовку пакета для переноса, перенос/обновление структуры базы данных, настройку окружения в целевой сети и обеспечение простоты последующих обновлений.

Опции переноса: через Docker и без Docker

Перенос с помощью Docker. Этот подход предполагает упаковку приложения и всех его зависимостей в Docker-образы, перенос этих образов в целевую сеть и запуск контейнеров там. Преимущества данного подхода: среда выполнения полностью контейнеризована, что исключает проблемы с несовместимыми версиями пакетов или отсутствием зависимостей на продакшене. Фактически, вы переносите готовые контейнеры, протестированные в сети разработки, и повторно запускаете их в изолированной сети. Docker изначально поддерживает работу в офлайн-режиме – образы можно **экспортировать командой** `docker save` **и импортировать** командой `docker load` [1]. Например, на машине разработки можно выполнить:

```
docker save --output project_image.tar <имя_образа>
```

для сохранения образа в tar-файл, а на целевой машине выполнить:

```
docker load --input project_image.tar
```

для загрузки этого образа [2]. При использовании Docker рекомендуется также перенести файлы оркестрации (например, `docker-compose.yml`, файл окружения `.env` и прочие необходимые конфиги), чтобы воспроизвести контейнерную инфраструктуру на продакшен-стороне. В офлайн-среде в файле `docker-compose.yml` нужно настроить использование локальных образов вместо попытки их заново собрать или загрузить из Registry. Как показывают практические

примеры, после импорта образов достаточно изменить настройки Compose-файла, чтобы сервисы запускались из уже загруженных образов, а не пытались билдиться из Dockerfile [3].

Стоит отметить, что в рамках Docker-подхода возможен и альтернативный путь переноса: **через виртуальную машину**. Некоторые специалисты предлагают собирать всё приложение внутри готовой виртуальной машины с установленным Docker и необходимыми контейнерами, а затем переносить **саму VM целиком** на изолированный сегмент [4]. При этом на продакшене достаточно запустить эту виртуальную машину, уже содержащую запакованный проект. Такой метод может быть оправдан, если удобно переносить крупные образы целиком или требуется полностью идентичное окружение; однако обычно достаточно переноса отдельных Docker-образов, как описано выше.

Перенос без Docker (традиционное развёртывание). Данный подход предполагает передачу исходного кода проекта, необходимых зависимостей и миграций базы данных, с последующей установкой/настройкой всего окружения вручную на целевой машине (или виртуальной машине) без контейнеризации. Этот вариант может понадобиться, если на продакшен-сервере решено не использовать Docker. Преимущества: меньше абстракций – приложение работает напрямую на ОС, что может упростить отладку определенных проблем и использование системных ресурсов. Недостатки: нужно убедиться, что на целевой системе установлены нужные версии Python, Django, библиотек, а также сама PostgreSQL, причем **все зависимости должны быть доставлены офлайн**. Придётся позаботиться о переносе Python-пакетов (модулей) и, возможно, системных библиотек. Для Python-зависимостей удобно подготовить «колесо» пакетов: на машине с доступом в интернет можно **выгрузить все требуемые пакеты** командой `pip download` [5], а затем установить их на целевой машине из локального хранилища, без обращения к PyPI (`pip install --no-index`) [6] – этот процесс детально рассмотрен ниже. Также нужно учитывать системы управления процессами (например, настройка systemd-сервиса для Django или запуск через uWSGI/Gunicorn), конфигурацию веб-сервера для статических файлов, и т.д. Всё это требует более тщательной настройки, зато не зависит от Docker.

Выбор между Docker и традиционным деплоем зависит от требований продакшена. Если допускается использование Docker, то контейнерный перенос обычно проще и надежнее, так как обеспечивает идентичность сред разработки и эксплуатации. Далее мы рассмотрим оба варианта более подробно – от подготовки пакета обновления до его развертывания в изолированной сети.

Подготовка переносимого пакета

Правильная подготовка переносимого пакета (набора файлов/образов, которые мы физически переместим через USB-накопитель) – ключ к успешному и быстрому обновлению продакшена. Пакет должен включать все необходимое для запуска обновленной версии приложения и обновления схемы базы данных. Рассмотрим состав пакета в двух случаях:

Подготовка пакета при использовании Docker

1. **Сборка новых образов приложения.** В среде разработки, после внесения изменений в код Django-проекта, необходимо собрать Docker-образ приложения с новой версией. Обычно используется Dockerfile, в котором прописаны инструкции по установке зависимостей и копированию кода. Убедитесь, что в образ включены **все необходимые зависимости** (например, пакеты Python из `requirements.txt`, компиляция статических файлов и пр.). Рекомендуется запускать сборку на машине с той же архитектурой, что и целевая (например, x86_64), чтобы избежать несовместимости образов. В Docker Compose

сценарии обычно достаточно выполнить `docker-compose build` (либо `docker build`) для сервисов приложения, чтобы собрать обновленные образы. Если используются базовые образы (например, `python:3.11` или `node:...`), они будут скачаны в процессе сборки; **убедитесь, что эти базовые образы сохранены** для офлайн-переноса (Docker автоматически включает все слои образа при сохранении). После успешной сборки и тестирования контейнеров (в сети разработки) можно пометить образы тегами версии (например, `myproject:v2.3`) для удобства версионирования.

2. **Экспорт Docker-образов.** Все собранные образ(ы) приложения, а также необходимые сторонние образы (например, образ PostgreSQL, если СУБД тоже будет запускаться в контейнере на продакшене) нужно сохранить в файлы. Для этого на машине разработки используйте команду `docker save`. Можно сохранить образы по отдельности или все сразу в один архив. Например, чтобы сохранить один образ:

```
docker save --output myapp_v2.3.tar myproject:v2.3
```

(где `myproject:v2.3` – имя и тег вашего образа приложения). Если нужно одновременно экспортировать несколько образов (например, сам проект и образ БД), укажите их через пробел, либо выполните `docker save` для каждого. Экспорт поместит все слои образа в `.tar` файл, готовый для переноса. Аналогично можно сохранить образ PostgreSQL (например, `postgres:15`) или любой другой, если планируется использовать конкретную версию на продакшене.

1. **Подготовка файлов оркестрации и конфигурации.** В пакет обновления следует включить файлы, описывающие запуск контейнеров и настройки окружения:
2. Файл `docker-compose.yml` (или несколько, если используете разделение на несколько Compose-файлов) – он описывает состав сервисов (приложение, база данных, и пр.), сети, тома. **Перед переносом убедитесь, что в этом файле сервис приложения не пытается собрать образ из Dockerfile** (директива `build:`), иначе в офлайн-среде сборка не пройдет из-за отсутствия интернета. Вместо этого укажите использование готового образа (директива `image:` с именем вашего образа). Вы можете изначально настроить Compose-файл под развёртывание, указав там имя загруженного образа и нужный тег.
3. Файл окружения `.env` (если используется) или другие конфигурационные файлы, в которых заданы критичные параметры: секретный ключ Django, DEBUG=False, ALLOWED_HOSTS, настройки подключения к базе (имя БД, пользователь, пароль), и др. Эти файлы нужно обновить в соответствии с продакшен-окружением (например, убедиться, что доступы к БД корректны для целевой среды). **Не включайте чувствительные данные в открытом виде**, если перенос осуществляется через несекьюрный носитель – при необходимости файл `.env` можно привезти отдельно и удалить копию из переносимого архива.
4. При необходимости – скрипты initial migration/seed для БД или dump-файлы (о них подробнее в следующем разделе). Если база данных будет разворачиваться пустая с нуля, можно использовать механизм инициализации PostgreSQL-контейнера: поместите SQL-скрипт с созданием схемы в директорию, которая монтируется в контейнер по пути `/docker-entrypoint-initdb.d/` – тогда при первом старте контейнера PostgreSQL этот скрипт выполнится автоматически. Однако более универсальный способ – просто перенести дамп схемы и самостоятельно выполнить его на продакшен-стороне.
5. **Статические файлы**: если в Django-проекте используются статические файлы (CSS, JS, изображения и пр.), убедитесь, что они собраны командой `manage.py collectstatic`. В

Docker-образе приложения имеет смысл выполнить `collectstatic` на этапе сборки (чтобы при запуске не требовать Node.js или других инструментов). Если статические файлы собираются вне контейнера, включите папку со статикой (например, `static/` или другую, указанную как `STATIC_ROOT`) в переносимый пакет, чтобы на продакшене сразу иметь необходимые файлы для обслуживания. В противном случае, нужно будет запускать collectstatic уже в изолированной сети (что возможно, если все ресурсы локальны, но может быть неудобно).

6. **Документация или скрипты деплоя:** полезно приложить README или набор команд, описывающих, как запустить обновление на продакшене. Это позволит любому администратору выполнить процедуру, следуя инструкциям, и снизит вероятность ошибок.

7. **Формирование архива для переноса.** Соберите все необходимые файлы в одну структуру (например, в директорию `release_v2.3/`), куда поместите:

8. Экспортированные образы (файлы `.tar`).
9. Compose-файл и окружение.
10. SQL-дамп структуры БД (если используете).
11. Прочие скрипты/файлы, о которых говорилось выше.

12. При желании – архив с исходным кодом проекта (это не обязательно, так как код уже включен в Docker-образ, но для прозрачности можно приложить и его). Затем заархивируйте эту папку (например, в `release_v2.3.zip` или `tar.gz`) для удобства транспортировки. Убедитесь, что объем получившегося архива соответствует возможности вашего физического носителя.

13. **Проверка перед переносом.** Желательно протестировать восстановление из подготовленного пакета в условиях, максимально приближенных к изолированной сети. Например, на отдельной машине отключить интернет и проверить, что:

14. Образы успешно загружаются командой `docker load`.
15. `docker-compose.yml` поднимает все сервисы без попыток доступа в интернет.
16. Приложение стартует и может подключиться к базе.
17. Миграции применяются корректно (например, запуск `docker-compose run web python manage.py migrate` проходит успешно) – подробнее о миграциях в разделе про базу данных.
18. Статические файлы доступны (например, если используете WhiteNoise или другой способ их отдачи, убедитесь, что всё работает). Такая «генеральная репетиция» поможет убедиться, что ничего не забыто (например, какой-то базовый образ или зависимость) и сэкономит время при непосредственном обновлении продакшена.

Подготовка пакета для развёртывания без Docker

1. **Подготовка исходного кода.** Сохраните актуальную версию кода Django-проекта. Обычно достаточно скопировать весь проект (включая `manage.py`, директорию с приложениями, файлы настроек, и т.д.). Рекомендуется удалить из папки проекта все не нужные для продакшена файлы (например, кеши, временные файлы, тестовые данные). Получившуюся папку можно сжать в архив (zip или tar) для переноски. **Важно:** убедитесь, что в проекте присутствуют все миграции Django (файлы в папках `migrations/` ваших приложений). Если вы добавляли новые модели/поля, выполните `python manage.py`

`makemigrations` на стадии разработки, чтобы сгенерировать миграционные файлы, и включите их в пакет. В изолированной сети мы будем применять эти миграции, чтобы создать/обновить структуру базы.

2. **Подготовка Python-зависимостей.** Продакшен-сервер, не имея выхода в интернет, не сможет напрямую установить пакеты через `pip`. Поэтому на стадии подготовки необходимо **выгрузить все требуемые зависимости** локально:

3. **Составьте список зависимостей.** Обычно это файл `requirements.txt`. Убедитесь, что он актуален (можно сгенерировать через `pip freeze > requirements.txt` в виртуальном окружении разработки).

4. **Скачайте все пакеты из PyPI.** На машине с доступом к интернету выполните команду:

   ```
   pip download -r requirements.txt --dest ./packages --only-binary :all:
   ```

   Эта команда скачает все указанные в requirements пакеты (в виде файлов `.whl` или `.tar.gz`) в локальную директорию `packages`. Ключ `--only-binary :all:` заставляет предпочесть бинарные колеса (wheel), чтобы не тянуть исходники и не компилировать при установке [5]. Если целевая платформа (ОС, версия Python) отличается от текущей, дополнительно укажите параметры `--platform`, `--python-version` и т.д., чтобы получить совместимые сборки. Например, для скачивания пакетов под Windows 64-bit с Python 3.8: `pip download -r requirements.txt -d packages --platform win_amd64 --python-version 38 --only-binary=:all:` (не забудьте также `--implementation cp` для указания CPython). В нашем случае, скорее всего, и разработка, и продакшен используют Linux x86_64 и одинаковую версию Python, тогда подобные флаги не требуются. - **Сохраните и упакуйте папку с пакетами.** После выполнения `pip download` у вас в папке (например, `packages/`) будут все необходимые дистрибутивы. Эту папку включаем в переносимый пакет. Также добавляем туда файл `requirements.txt` (он понадобится для установки). - **Системные зависимости.** Проверьте, нет ли у вашего проекта зависимостей, требующих установки системных библиотек или утилит. Например, пакет `psycopg2` (PostgreSQL драйвер) обычно требует клиентскую библиотеку Postgres (libpq). Если вы скачали wheel для `psycopg2`, он уже включает бинарный модуль и внешних libs не потребуется. Но в общем случае, если необходимы какие-то .deb-пакеты или установочные файлы для ОС (например, JDK для поиска, libcairo для отчетов, etc.), их тоже нужно положить в пакет и быть готовым установить вручную на продакшен-сервере. Альтернативно, можно собрать такие зависимости статически или использовать облегченные аналоги (как в случае psycopg2-binary wheel). В нашем сценарии, скорее всего, достаточно стандартного набора (Python, Django, psycopg2) – все они могут быть установлены из wheel, без дополнительных шагов.

5. **Подготовка базы данных (дамп или скрипты миграции).** Поскольку данные в БД «не критичны», акцент делается на переносе схемы базы. Здесь есть два возможных подхода (можно использовать любой, но **не одновременно**, чтобы избежать конфликта миграций):

6. **Через миграции Django.** Предполагается, что вы перенесёте миграционные файлы вместе с кодом и на продакшене просто выполните `python manage.py migrate`, чтобы Django сама создала или обновила структуру БД до актуального состояния. Это наиболее

«естественный» способ: он сохраняет любые существующие данные (если таковые есть) и применяет только новые изменения. Убедитесь, что на продакшене есть доступ к базе (см. раздел настройки окружения) и корректные настройки подключения в `settings.py` / `.env`. Django будет опираться на таблицу `django_migrations` внутри базы, чтобы понять, какие миграции уже применены, и применить недостающие. **Если продакшен-база разворачивается с нуля (пустая)** – миграции создадут всю структуру заново. Если база уже существует от предыдущей версии – миграции выполнит только изменения. Важно: если в новой версии были удалены модели/поля, убедитесь, что миграции для этого также созданы (Django по умолчанию создает миграции удаления).

7. **Через SQL-дамп структуры.** Этот вариант – вручную перенести *схему* PostgreSQL из разработки в продакшен с помощью инструментов СУБД. На стороне разработки выполните экспорт схемы командой pg_dump с флагом схемы:

```
pg_dump -U <user> -s -d <dbname> -f db_schema.sql
```

Ключ `-s` (`--schema-only`) заставляет `pg_dump` выгрузить **только структуру базы без данных** [7]. В результате получится SQL-файл с командами `CREATE TABLE`, `CREATE INDEX`, etc., для всех таблиц и объектов БД. Обратите внимание, что пользователи/роли БД по умолчанию не входят в такой дамп – их можно сохранить отдельно через `pg_dumpall --globals-only`, хотя в простом случае достаточно вручную создать пользователя на продакшене. После получения файла `db_schema.sql` включите его в переносимый пакет. При желании, можно также выгрузить какую-то справочную информацию или минимальные данные (например, контент таблиц-справочников) – для этого можно либо создать отдельный дамп с данными конкретных таблиц, либо впоследствии загрузить данные фикстурами Django. Но в условии задачи сказано, что пользовательские данные несущественны, так что будем переносить только структуру.

Второй шаг – **подготовить скрипт восстановления**. Он может быть простым: на продакшен-сервере достаточно выполнить команду создания базы и залить дамп:

```
CREATE DATABASE <dbname>;
```

(если база еще не создана) – эту команду можно выполнить из-под пользователя Postgres. Затем:

```
psql -U <user> -d <dbname> -f db_schema.sql
```

или эквивалентно `psql -U <user> <dbname> < db_schema.sql` [8]. Эта команда применит все SQL инструкции и создаст схему базы данных, аналогичную разработке. Перед этим убедитесь, что: - На сервере создан пользователь БД (`<user>`), от имени которого вы будете восстанавливать схему, и у него есть права (например, CREATEDB или вы подключаетесь под суперпользователем PostgreSQL). - Имя базы данных в продакшене совпадает с тем, что прописано в настройках Django (или вы скорректируете настройки). Например, можно использовать одно имя во всех средах, чтобы не менять конфиги. После выполнения импорта можно (необязательно) прогнать `ANALYZE` для обновления статистики, если база большая [9] – однако при отсутствии данных это не критично.

Обратите внимание: **не рекомендуется одновременно использовать оба подхода**. То есть, если вы решаете пойти путем SQL-дампа, то на продакшене не следует запускать `manage.py migrate` для создания тех же таблиц – это приведет к конфликтам (Django обнаружит, что таблицы уже существуют, и может выдать ошибки, либо придется помечать миграции как выполненные). В первом же подходе (с миграциями) нет нужды вручную импортировать SQL, Django сама всё сделает. Выберите тот метод, который удобнее. Часто миграции предпочтительнее, так как позволяют обновлять схему без потери данных и контролировать изменения, но если вы планируете **каждый раз переносить чистую схему и не сохранять продакшен-данные**, то можно и просто перезаливать структуру целиком. В дальнейшем, когда проект стабилизируется, вы можете комбинировать: например, первый импорт сделать через SQL (создав начальную структуру), а последующие изменения накатывать миграциями – Django поддерживает применение миграций поверх существующей базы, если отметите начальные миграции как выполненные (`--fake-initial`). Но для простоты, в рамках данной задачи можно всегда пересоздавать базу на продакшене заново, раз критичных данных нет.

1. **Иные компоненты проекта.** Помимо кода и базы, убедитесь, что учитываете:
2. **Статические файлы**: как и в Docker-подходе, либо перенесите заранее собранные статические файлы, либо будьте готовы собрать их на продакшен-сервере. Проще включить папку со статикой (после `collectstatic`) в архив, чтобы потом просто распаковать ее в нужное место (например, в папку, откуда веб-сервер раздает статику, или настроить `STATIC_ROOT` на уже существующую папку).
3. **Конфигурация веб-сервера**: если планируется использовать Nginx/Apache для обслуживания Django (например, Nginx + Gunicorn/UWSGI), то подготовьте конфигурационные файлы для них (виртуальный хост, настройка прокси на Gunicorn, раздача статики из директории). Их тоже можно включить в пакет или иметь отдельные шаблоны. В изолированной сети установка веб-сервера и настройка происходит вручную, но если она выполнена один раз, обновление Django-проекта не должно потребовать изменений веб-сервера (если только не меняется путь к статике или имена сервисов).

4. **Скрипты запуска приложения**: если Django будет работать под управляющим процессом (например, systemd unit файл для Gunicorn), то такой unit-файл тоже стоит заготовить и перенести. Например, `myproject.service` для systemd, где указано, как запускать Gunicorn (путь к сокету/порту, пользователь, PATH до виртуального окружения и т.д.). При первом развёртывании на продакшене вы установите этот сервис и включите (enable) его, а при следующих обновлениях – просто перезапустите сервис после замены кода.

5. **Формирование и проверка архива.** Как и для Docker-варианта, собрать все файлы:

6. Архив с кодом (`myproject.zip` или tar.gz).
7. Папка с пакетами Python (`packages/` + `requirements.txt`).
8. Дамп базы (`db_schema.sql`) или альтернативно данные миграций (но миграции уже в коде).
9. Скрипты разворачивания (shell-скрипт, batch или инструкции).
10. Конфиги сервиса/web-сервера (если применимо).
11. Любые другие необходимые артефакты (ключи, сертификаты, если, например, SSL на веб-сервере — хотя это выходит за рамки чисто Django-проекта). Убедитесь, что архив можно распаковать и установить без доступа к интернету. Хорошая практика – протестировать установку в изолированной виртуальной машине: развернуть там чистую ОС (той же версии, что на продакшен-сервере), скопировать архив и пошагово пройти процесс установки (см. следующий раздел). Так вы убедитесь, что, например, **все зависимости на месте** (pip не пытается лезть в сеть), **версия Python подходит**, **база данных**

подключается и т.д. При обнаружении проблем (нехватающих библиотек, неправильных путей) можно исправить пакет до реального переноса.

Экспорт и импорт базы данных (структуры)

Как было отмечено, перенос данных в PostgreSQL можно осуществить либо на уровне приложения (миграциями Django), либо на уровне СУБД (дамп/импорт схемы). Здесь обобщим необходимые команды и шаги, чтобы схема БД на продакшене соответствовала версии проекта:

- **Экспорт схемы на стороне разработки:** В среде разработки, где есть доступ к PostgreSQL, выполните резервное копирование структуры БД. Если база развернута в Docker-контейнере, можно сделать дамп либо изнутри контейнера (через `docker exec`), либо с хоста (если хост может подключиться к контейнеру по порту). Например: `docker exec -t <postgres_container> pg_dump -U <user> -s <dbname> > db_schema.sql`. Либо, если Postgres опубликован на порт хоста: `pg_dump -U <user> -h <host> -p <port> -s <dbname> > db_schema.sql`. В итоге файл `db_schema.sql` содержит DDL команды (CREATE TABLE и др.) без данных. Проверьте, что дамп не содержит ничего лишнего (например, данные или артефакты, если вы по ошибке не указали `-s`). Размер файла схемы обычно небольшой.

- **Перенос и импорт схемы на продакшен:** Скопируйте файл схемы на целевую машину (он будет внутри вашего переносного пакета). Если вы используете Docker на продакшене для базы, то для импорта можно либо скопировать файл внутрь контейнера, либо выполнить команду psql через `docker exec` с перенаправлением ввода. Например: `docker cp db_schema.sql <postgres_container>:/db_schema.sql` и затем `docker exec -u postgres <postgres_container> psql -d <dbname> -f /db_schema.sql`. Либо проще: `docker exec -i <postgres_container> psql -U <user> -d <dbname> < db_schema.sql` (это сразу читает локальный файл и подает в STDIN psql). Если же база на продакшене установлена как служба на хосте (без Docker), то используйте команду psql напрямую: `psql -U <user> -d <dbname> -f db_schema.sql` (предварительно скопировав sql-файл на сервер). **Не забудьте создать саму базу и роль пользователя перед импортом**, если они отсутствуют [8]. Обычно это делается из-под суперпользователя Postgres:

```
CREATE DATABASE <dbname> OWNER <user>;
```

(эта команда назначит существующего пользователя владельцем БД; если пользователя `<user>` нет, сначала создайте его через `CREATE ROLE <user> LOGIN PASSWORD '...'` или используйте универсального postgres). Далее выполните импорт, как описано. После успешного выполнения SQL-скрипта структура БД готова.

- **Применение миграций Django:** Если выбран вариант миграций, то на продакшене (уже после настройки окружения и запуска приложения) выполните команду:

```
python manage.py migrate
```

(или эквивалент через `docker-compose run` / `docker exec` внутри контейнера приложения). Django просканирует файлы миграций и применит все неоплаченные миграции к базе данных. Убедитесь, что приложение сконфигурировано на подключение к нужной БД (например, через переменные окружения в Docker или в settings.py). При первом деплое на пустую базу миграции сами создадут все таблицы (аналогично импортированному SQL). Если база уже содержала старые данные, миграции обновят структуру без потери данных (для данных, помещающихся в новые структуры). После применения миграций рекомендуется запустить встроенные команды инициализации, если они есть (например, `createsuperuser` для создания администратора, или загрузку фикстур, если вы подготовили их с исходными данными). Однако, если вы переносили структуру через SQL-дамп, **не запускайте migrate на чистую базу**, иначе Django может попытаться заново создать таблицы. В случае полного переноса схемы SQL-дампом, лучше **отключить автоматический запуск миграций** и пометить начальные миграции как выполненные (например, командой `manage.py migrate --fake-initial`, которая помечает initial миграции примененными, если таблицы уже существуют).

- **Восстановление данных (при необходимости):** Хотя мы предполагаем, что критичных данных нет, возможно, вам понадобится перенести часть данных (например, справочники или первоначального пользователя). Проще всего это сделать через Django фикстуры или скрипты:

- Фикстуры: на разработке можно создать фикстуру (`manage.py dumpdata app.Model --indent 2 > fixture.json`) с нужными моделями, а на продакшене загрузить через `manage.py loaddata fixture.json`.
- Скрипты инициализации: либо написать management command, которая заполнит начальными данными.
- SQL: либо включить SQL-INSERT команды в ваш дамп (pg_dump позволяет делать дамп с данными, но тогда не забудьте флаг `--inserts` или `--data-only` для конкретных таблиц). Например, можно дампом выгрузить справочные таблицы с данными.

Если ничего из этого не критично, можно эти шаги опустить. Для создания суперпользователя часто достаточно выполнить `manage.py createsuperuser` уже на продакшене вручную.

Итог: после выполнения либо SQL импорта, либо миграций, ваша база данных в изолированной сети должна иметь актуальную структуру (и, опционально, данные) для работы нового выпуска Django-приложения.

Настройка целевого окружения в изолированной сети

На данном этапе переносимый пакет с новым выпуском доставлен в изолированную сеть. Теперь нужно правильно настроить окружение и запустить обновленное приложение. Разделим рекомендации для двух подходов.

Настройка на стороне продакшена при использовании Docker

1. **Установка Docker (если еще не установлен).** В изолированной сети может потребоваться установка Docker Engine на целевой машине. Если интернет недоступен, нужно иметь офлайн-установщик. Для Linux можно заранее скачать .deb/.rpm пакеты Docker CE и установить их вручную. Для Windows/Mac – установщик Docker Desktop. Предположим, что Docker уже установлен и настроен (так как условия допускают

использование Docker). Также убедитесь, что версия Docker не ниже той, на которой образы были собраны, во избежание проблем совместимости.

2. **Загрузка образов.** Скопируйте файлы с Docker-образами (tar-файлы, полученные через `docker save`) на целевую машину. Выполните импорт каждого образа:

```
docker load --input myapp_v2.3.tar
docker load --input postgres15.tar
```

   и т.д. (в зависимости от того, какие файлы вы привезли). Docker загрузит все слои, и после этого команды `docker images` должны показать новые образы. **Примечание:** если образы занимают много места, убедитесь, что на диске целевой машины достаточно свободного пространства для их распаковки.

3. **Обновление/развёртывание контейнеров.** Далее, используйте ваш `docker-compose.yml`. Скопируйте (распакуйте) его в нужное место на сервере. Убедитесь, что файл `.env` (с переменными окружения) тоже присутствует рядом, если он нужен. Теперь запускайте (или обновляйте) сервисы:

4. Если на продакшен-сервере **впервые** разворачивается этот проект, просто выполните команду:

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

   (или `docker compose up -d` для новой версии CLI) в директории с `docker-compose.yml`. Compose прочитает конфиг, обнаружит, что нужные образы уже есть локально (после `docker load`), создаст контейнеры сети, тома и запустит все сервисы (Django-приложение, Postgres и др.).
5. Если на сервере **уже работала предыдущая версия** проекта (например, запущена старая версия контейнеров), то процесс обновления такой:
   - Остановить и удалить старые контейнеры: `docker-compose down` (при этом, если использовался именованный volume для БД, он **не удалится** по умолчанию, что позволит сохранить старые данные; если данные не нужны – можно потом удалить volume вручную).
   - Загрузить новые образы (мы это сделали).
   - Обновить конфигурацию (если, например, изменился `docker-compose.yml` — заменить на новый).
   - Запустить новый состав: `docker-compose up -d --remove-orphans`. Флаг `--remove-orphans` на случай, если какие-то старые сервисы удалены в новой версии – он уберет лишние контейнеры.
   - Docker Compose сам сопоставит имена сервисов и использует новые образы, создаст новые контейнеры. Если имя volume для БД не поменялось и volume не удаляли, то новый контейнер БД подхватит существующие данные (но структура может быть старая, поэтому всё равно надо применить миграции!). Если же мы решили развернуть **с чистой базы** (не сохраняя volume), можно перед `up` удалить volume (например, `docker volume rm <name>`), Docker затем создаст пустой.
6. Убедитесь, что в Compose-файле заданы правильные настройки: например, переменные окружения для подключения Django к БД (если БД контейнер именуется `db`, то обычно

`DB_HOST=db`, `DB_NAME=...` и т.д. – они должны быть определены либо в `.env`, либо прямо в compose). Также, возможно, нужно прописать `restart: always` для сервисов, чтобы они автоматически перезапускались после ребута сервера.

7. **Применение миграций/импорт схемы.** После запуска контейнеров, в зависимости от выбранного подхода для БД:

8. Если вы полагаетесь на Django миграции: выполните миграции внутри контейнера приложения. Команда: `docker-compose exec web python manage.py migrate` (где `web` – имя сервиса Django в compose). Либо `docker-compose run --rm web python manage.py migrate` – что запустит временный контейнер приложения и выполнит команду. Удостоверьтесь, что база данных (`db` сервис) уже запущена к этому моменту.
9. Если вы импортируете SQL-дамп: как описано ранее, выполните `docker exec` в контейнер БД или другие эквивалентные шаги, чтобы залить дамп. Это можно сделать до запуска контейнера приложения (т.е. поднять сначала только БД, залить схему, затем поднять приложение), либо после – не суть важно, главное чтобы до того, как приложение начнет обслуживать запросы.

10. После миграций/импорта проверьте, не требуется ли выполнить дополнительные команды: собрать статику (если не сделали заранее) – например `docker-compose exec web python manage.py collectstatic --noinput` (но если вы включили это в Dockerfile, то уже не нужно), создать суперпользователя и т.п.

11. **Проверка работы приложения.** Когда контейнеры запущены, убедитесь, что:

12. Django-приложение *работает*: контейнер не падает, в логах (`docker-compose logs web`) нет ошибок импортов или миграций. Если есть ошибки — возможно, забыта какая-то переменная окружения или зависимость.
13. Контейнер БД работает: `docker-compose logs db` не содержит ошибок, БД слушает порт.
14. Выполните пробный HTTP-запрос к приложению. Это может быть через браузер (если инфраструктура позволяет, например, приложение доступно по определенному URL/IP внутри сети) или командой `curl` внутри той же сети. Если вы не используете отдельный веб-сервер, а просто запускаете Django через встроенный dev-сервер или Gunicorn, то вероятно приложение доступно на порту 8000. Docker-Compose пример, приведенный ниже, публикует порт 8000 наружу (для простоты). В реальности, для продакшена лучше поставить прокси-сервер (Nginx) или хотя бы использовать Gunicorn. В любом случае, проверка ответа (например, HTTP 200 на страницу логина админки `/admin/`) покажет, что развертывание прошло успешно.

15. **Статические файлы:** зайдите на страницу админки и убедитесь, что стили применяются (значит, статика раздается). Если вы используете WhiteNoise внутри Django, то статика будет обслуживаться самим приложением (убедитесь, что `STATIC_ROOT` и т.д. настроены, и файлы действительно на месте в образе). Если используете отдельный контейнер nginx для статики – проверьте его тоже. Наш пример упрощенно предполагает, что WhiteNoise или аналог используется, чтобы не заводить лишний контейнер в офлайн-сети.

16. **Очистка и завершение.** Если всё проверено, деплой можно считать завершенным. Старые образы Docker на сервере при желании можно удалить (`docker image rm <id>`), чтобы не занимали место, особенно если они больше не будут использоваться. Однако, возможно, имеет смысл оставить предыдущий образ как резерв на случай быстрого отката

(если обновление вдруг оказалось с проблемами, вы могли бы загрузить старый контейнер назад). Решайте по обстоятельствам.

Настройка окружения на продакшене без Docker

1. **Подготовка системы.** Разворачивание приложения вручную требует, чтобы на целевой машине были установлены:
2. **Python нужной версии.** Убедитесь, что версия Python соответствует той, под которую готовились зависимости (например, Python 3.11). Если нет – установите необходимую версию. В офлайн-среде установка может быть выполнена с дистрибутивного носителя (например, если ОС – Ubuntu, то иметь .deb пакеты python3.11 и pip) или путем переноса сборки Python. В крайнем случае, Python можно собрать из исходников на месте, но лучше позаботиться о бинарном установщике.
3. **PostgreSQL сервер.** Если планируется, что база будет крутиться на том же сервере, установите PostgreSQL (версии, совместимой с дампом – обычно можно ту же версию, что на dev, либо близкую). Без интернета это значит иметь заранее скачанный установщик (например, RPM/DEB пакеты) или установить с ISO-образа ОС, если он включает Postgres. Настройте PostgreSQL: запустите службу, откройте порт (если нужно внешнее подключение, либо сокет локально). Создайте пользователя и пустую базу, как обсуждалось. Альтернативный путь – запустить саму СУБД также в контейнере, даже если приложение без Docker. Иногда так делают: т.к. Postgres легко завернуть в Docker, можно его там и оставить. Но вопрос подразумевает вариант "без Docker" как полностью без контейнеров, поэтому рассмотрим классическую установку.
4. **Веб-сервер и WSGI.** Продакшен-стек Django обычно включает Gunicorn (WSGI-сервер) + Nginx (реверс-прокси). Установите Gunicorn (`pip install gunicorn`, можно включить в requirements). Установите Nginx из пакета (если требуется раздавать статику и проксировать запросы). Заметьте, для офлайн-установки Nginx/Apache вам тоже нужно иметь их пакеты, либо, если интернет недоступен, использовать репозиторий, находящийся внутри этой сети (вопрос не описывает, есть ли внутри изолированной ЛВС локальный репозиторий, но предположим, что нет). Если очень сложно ставить веб-сервер, Django-приложение *может работать и без него* – например, Gunicorn можно настроить слушать 80 порт и отдавать статические файлы через WhiteNoise. Это не идеально для производительности, но в маленькой изолированной сети может быть приемлемо. Мы отметим конфигурацию с Gunicorn + WhiteNoise для упрощения.

5. **Виртуальное окружение (virtualenv).** Рекомендуется создать отдельное виртуальное окружение Python для вашего проекта, чтобы не смешивать с системными пакетами. Например: `python3 -m venv /opt/myproject/venv`. Это не обязательно, но good practice. После создания активируйте его (`source venv/bin/activate`) и далее выполняйте установки пакетов и запуск приложения внутри него.

6. **Развертывание кода и установка зависимостей.** Распакуйте архив с кодом проекта (например, скопируйте `myproject.zip` и разархивируйте в `/opt/myproject/`). Проверьте, что файлы на месте, структура корректна.

7. Скопируйте папку с подготовленными пакетами Python (директория `packages/` с wheels) на сервер, либо монтируйте USB и работайте оттуда.
8. Установите зависимости из локального источника. Для этого активируйте виртуальное окружение и выполните:

```
pip install --no-index -f ./packages -r requirements.txt
```

Эта команда возьмет каждый пакет из `requirements.txt` и установит его, **не** обращаясь в интернет, а используя файлы из папки (`-f` указывает путь к папке с пакетами) [6]. Если все зависимости присутствуют, установка пройдет успешно. Следите за выводом: pip сообщит, если какого-то пакета не находит или ему не хватает подходящего колеса (например, если версия/платформа не совпала). В таком случае нужно либо перекачать нужный пакет правильно, либо установить вручную.

9. Дополнительно: установите любые системные зависимости, если нужно. Например, если в вашем проекте используется GDAL, а его wheel нет, нужно установить GDAL через ОС. Такие случаи зависят от проекта, в нашем примере, вероятно, ничего дополнительного не надо.

10. **Конфигурация проекта.** Перед запуском приложения убедитесь, что настройки продакшена заданы правильно:

11. В Django `settings.py` (или лучше в отдельном `production.py` или через переменные окружения) должен быть `DEBUG = False` (для безопасности), настроены `ALLOWED_HOSTS` (добавьте адреса/имена, по которым будете обращаться к приложению в этой сети).
12. Настройки подключения к БД (DATABASES) – укажите имя хоста (например, localhost или ip, если БД на отдельном сервере), порт (5432 по умолч.), имя базы, пользователя, пароль. Эти параметры можно задать через переменные окружения, используя механизм `django-environ` или просто `os.environ` в настройках, чтобы не хранить пароли в коде. Поскольку сеть изолирована, угроза утечки меньше, но все же секреты лучше не хардкодить. Если вы перенесли `.env` файл, загрузите его.
13. Настройки статических/медийных файлов: `STATIC_ROOT` должен указывать на директорию, где лежат статические файлы (например, `/opt/myproject/static/`). Если вы привезли статические файлы отдельно, поместите их туда. Если нет – создайте пустую директорию и убедитесь, что у веб-сервера или WhiteNoise есть к ней доступ.
14. Файл `manage.py` и wsgi/uvicorn: убедитесь, что виртуальное окружение активно при запуске. Например, если будете делать systemd unit, пропишите там `ExecStart=/opt/myproject/venv/bin/gunicorn myproject.wsgi:application ...`.

15. Логирование: настройте, куда пишутся логи (в файл, в syslog, консоль). В Docker это было неактуально, т.к. логи шли в stdout, а здесь полезно писать в файл.

16. **Инициализация базы данных.** В зависимости от выбранного подхода:

17. Если используем **миграции Django**: убедитесь, что PostgreSQL сервер запущен и доступен. Затем выполните:

```
python manage.py migrate
```

(из директории проекта, с активированным venv). Это создаст все таблицы. Если нужно, создайте суперпользователя: `python manage.py createsuperuser` и введите данные.

18. Если используем **SQL-дамп**: сначала создайте новую базу (как описано ранее), затем выполните команду импорта:

```
psql -U <user> -d <dbname> -f db_schema.sql
```

После этого, если вы планируете использовать Django миграции дальше, можно пометить исходные миграции как выполненные:

```
python manage.py migrate --fake-initial
```

(эта команда помечает начальные миграции как примененные, если таблицы из них уже существуют в базе). Это позволит Django не пытаться снова создать таблицы. После этого, дальнейшие миграции (из новых версий) можно будет применять обычным способом.

19. **Запуск приложения.** Настало время запустить Django-приложение на продакшен-сервере:

20. **Gunicorn (WSGI-сервер):** Установленный gunicorn можно запустить командой:

```
gunicorn myproject.wsgi:application --bind 0.0.0.0:8000 --workers 4
```

(запустит 4 worker-процесса, слушающих на порт 8000). Попробуйте запустить в ручном режиме для теста. Если всё настроено правильно, вы должны увидеть, что сервер стартовал без ошибок. Попробуйте зайти по URL (например, http://<server_ip>:8000/) — возможно, нужно сделать туннель или использовать текстовый браузер на самом сервере, если внешней машины в сети нет. Но хотя бы убедитесь, что gunicorn не падает.

21. **Настройка сервиса:** Чтобы приложение автоматически стартовало и работало постоянно, обычно на Linux создается unit-файл systemd. Пример (`/etc/systemd/system/myproject.service`):

```
[Unit]
Description=Gunicorn instance for myproject
After=network.target
[Service]
User=www-data
Group=www-data
WorkingDirectory=/opt/myproject
Environment="PATH=/opt/myproject/venv/bin"
ExecStart=/opt/myproject/venv/bin/gunicorn myproject.wsgi:application --
bind 0.0.0.0:8000
[Install]
WantedBy=multi-user.target
```

После создания файла:
`sudo systemctl enable myproject && sudo systemctl start myproject`. Это запустит Gunicorn в фоне. (Настройте `User` / `Group` под подходящую системную учетную запись, права на файлы проекта должны ей соответствовать).

22. **Веб-сервер (опционально):** Если требуется, настройте Nginx как реверс-прокси, который слушает 80/443 порты и форвардит на Gunicorn (127.0.0.1:8000), а также раздает статику. Конфигурация стандартна (location /static/ -> корень на STATIC_ROOT, location / -> proxy_pass к gunicorn). Если Nginx не используется, убедитесь, что Gunicorn слушает на интерфейсе 0.0.0.0 (как выше), чтобы его можно было достичь извне, и, возможно, откройте firewall для порта 8000.

23. **Проверка приложения:** Как и в Docker случае, проверьте работоспособность. Зайдите на основную страницу или `/admin/`. Убедитесь, что всё функционирует: страницы открываются, статика грузится (если WhiteNoise, то Gunicorn сам отдаст статику; если Nginx, то он). Проверьте базовые действия в приложении (логин, переходы), чтобы убедиться, что миграции успешно применены и ошибок нет.

24. **Очистка и последующие обновления.** Старая версия приложения (если была) может быть остановлена и файлы удалены. Если вы используете новый путь для кода, аккуратно удалите старые файлы, чтобы не мешались. При переходе на новую версию убедитесь, что нет конфликтов портов (новый Gunicorn должен слушать тот же порт, что и старый, иначе users might connect to old instance). Обычно при обновлении делают так: запускают новый код на другом порту для проверки, затем переключают (но это избыточно для офлайн, т.к. пользователя, возможно, вообще нет в момент обновления).

Для следующих переносов процесс будет аналогичным: принести обновленный код, остановить `myproject.service`, обновить файлы, установить новые зависимости (можно оптимизировать, установив только разницу: `pip install` новых версий), применить миграции, перезапустить сервис. Чтобы облегчить повторяемость, можно написать сценарий (например, shell-скрипт) на продакшен-стороне, который выполняет эти шаги, чтобы каждый раз не делать вручную. Однако его написание требует понимания всех команд — по сути, он будет следовать тем же пунктам, что мы описали.

Обеспечение простоты и воспроизводимости дальнейших переносов

Настроив базовый процесс миграции проекта в изолированную сеть, важно сделать так, чтобы повторные переносы (новых версий) были максимально простыми и надежными. Вот несколько рекомендаций:

- **Документируйте процесс:** Хотя в рамках этой инструкции уже описаны шаги, полезно иметь внутреннюю документацию именно под ваш проект. Обновляйте ее, если процесс меняется. Например, сохраняйте список команд, которые нужно выполнить на продакшене для обновления. Хорошо, если эта документация путешествует вместе с переносимым пакетом (например, текстовый файл README или скрипт с комментариями).

- **Автоматизация:** Стремитесь автоматизировать как можно больше. В идеале, подготовьте скрипты:

- На стороне разработки: скрипт `make_release.sh`, который собирает образ, делает `docker save`, пакует все нужные файлы в архив релиза, готовый для копирования. Этот скрипт будет включать команды, описанные в разделе подготовки (сборка, дамп, копирование файлов).

- На стороне продакшена: скрипт `deploy_update.sh`, который разворачивает обновление. Например, для Docker: загружает образы (`docker load`), перезапускает Compose. Для варианта без Docker: останавливает сервис, распаковывает архив, устанавливает зависимости, применяет миграции, запускает сервис. Конечно, запускаться этот скрипт будет все равно вручную, но он минимизирует ручной труд и вероятность ошибки. Скрипт может запрашивать подтверждение опасных действий (например, удаление старой базы).

- **Версионирование и консистентность:** Ясно помечайте версии приложений и соответствующей схемы БД. Например, можно синхронизировать версию релиза с номером миграции. Тогда при переносе вы будете уверены, что обновляете правильной миграцией. В Docker подходе используйте теги образов (v1, v2 и т.п.) и прописывайте тот же тег в docker-compose. Таким образом, одновременно на сервере могут существовать образы разных версий (не конфликтуя), и вы всегда можете откатиться, запустив предыдущий тег. В бездокерном подходе можно хранить прошлую версию кода в резервной папке, чтобы быстро вернуться на старую версию, если новая дала сбой (не забудьте иметь и копию старой БД, если надо).

- **Проверка и тестирование обновлений:** Перед тем как перенести обновление в «боевую» (пусть и изолированную) среду, протестируйте новую версию в максимально приближенных условиях. Это включает не только сам функционал приложения, но и процедуру переноса. Например, можно поднять временную виртуальную машину, где нет интернета, и пройти весь процесс деплоя как учебную тревогу. Если команда, выполняющая обновление, меняется, позаботьтесь о том, чтобы knowledge transfer был – возможно, менее опытные коллеги смогут просто запустить подготовленные вами скрипты.

- **Минимизация передаваемых данных:** Для ускорения и удобства вы можете оптимизировать пакет. Например, если статические файлы очень объемны и редко меняются, можно передавать только дифф или не передавать их каждый раз (держать на продакшене копию и обновлять при изменениях). Если Docker-образы получаются большими, рассмотрите multi-stage build (сборка только нужного, а не полного OS) или чистку кешей/слоев перед `docker save`. Однако не переусердствуйте – простота важнее: лучше передать один большой файл, но автоматически, чем пытаться вручную отбирать, что обновилось.

- **Безопасность и доступ:** Изолированная сеть – это хорошо для безопасности, но подумайте о механизме доставки. Физический перенос через USB должен соответствовать политике безопасности вашей организации (например, проверка носителя на вирусы и т.д.). Удостоверьтесь, что на продакшен-сервере есть порты/устройства для чтения носителей или предусмотрите другой канал (например, временное прямое подключение к ноутбуку). После переноса убедитесь, что нигде не остались конфиденциальные данные (например, если на носителе были пароли в файлах, удалите их после установки или шифруйте такие файлы на перенос).

- **Консистентность баз данных:** Поскольку вы переносите не сами данные, а только структуру, важно поддерживать некоторую согласованность между средами. Например, если в продакшен-среде все-таки будут накоплены какие-то данные (пусть и не критичные), при очередном переносе новой версии убедитесь, что либо эти данные сохранены (в случае миграций), либо вы предупредили пользователей, что данные будут сброшены. Во втором случае, возможно, удобнее всегда пересоздавать БД и заново

загружать какие-то основы (тогда можно, например, встроить создание начального суперпользователя и пары записей прямо в SQL-дамп или миграцию). Главное – зафиксируйте эту политику, чтобы не было неожиданностей.

- **Хранение резервных копий:** Несмотря на не критичность данных, храните резервные копии того, что развернуто на продакшене: и код, и структура БД, и (если есть) пользовательские данные. Это поможет, если придется восстановить систему или отладить проблему. Делайте бэкап перед применением обновления (например, pg_dump старой схемы/данных, копию папки проекта). Это можно встроить в сценарий обновления.

Следуя этим рекомендациям, вы создадите предсказуемый и удобный процесс регулярного обновления изолированной продакшен-среды. В идеале, со временем перенос новой версии сведется к нескольким понятным действиям, занимая минимум времени и не вызывая простоев.

Примеры Dockerfile и docker-compose.yml

Ниже приведены упрощенные примеры конфигурационных файлов для контейнеризации Django-приложения с PostgreSQL. Их можно взять за основу и адаптировать под свой проект.

Dockerfile для Django-приложения

Этот Dockerfile предполагает, что у вас есть файл `requirements.txt` и Django-проект (с `manage.py` в корне и приложениями в папке, например, `myproject/`).

```
# Базовый образ с Python (выберите нужную версию, например 3.11)
FROM python:3.11-slim

# Устанавливаем рабочую директорию внутри контейнера
WORKDIR /app

# (Опционально) установка системных зависимостей, если нужны (пример: libpq
для psycopg2)
# RUN apt-get update && apt-get install -y libpq-dev && rm -rf /var/lib/apt/
lists/*

# Копируем файл зависимостей и устанавливаем Python-пакеты
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Копируем весь код проекта в контейнер
COPY . .

# Собираем статические файлы (если используете Django collectstatic)
RUN python manage.py collectstatic --noinput

# Открываем порт (если планируем обращаться напрямую, иначе через gunicorn -
не обязательно)
EXPOSE 8000
```

```
# Определяем команду запуска контейнера: запускаем Gunicorn WSGI-сервер
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "myproject.wsgi:application"]
```

Примечания к Dockerfile: - Используется официальный образ Python slim для минимизации размера. - Перед копированием кода копируется только `requirements.txt` для лучшего кеширования слоев: это значит, что при неизменном списке зависимостей Docker будет брать кешированный слой установки пакетов. - Командой `collectstatic` мы собираем статику в STATIC_ROOT (убедитесь, что в настройках Django `STATIC_ROOT` указывает на какую-то папку внутри `/app`, которая будет содержать файлы; можно внести правку, например, `STATIC_ROOT = "/app/static"`). После этого статические файлы можно обслуживать напрямую. Здесь мы не настраиваем Nginx, вместо этого Gunicorn можно дополнить middleware WhiteNoise для отдачи статики, или же раздавать статику через сам Django (нежелательно на продакшене без WhiteNoise, но возможно). - Команда запуска (CMD) использует Gunicorn. Убедитесь, что Gunicorn добавлен в `requirements.txt`. Также можно настроить число воркеров Gunicorn через переменные окружения (например, `--workers 4` для 4 воркеров, в CMD или через ENV). - Если требуется, можно добавить создание пользователя внутри контейнера и запуск от не root (для безопасности), но чтобы не усложнять, опустим. - Если нужно выполнять миграции при стартуе контейнера автоматически, это можно сделать, добавив entrypoint script, который вызывает `python manage.py migrate` перед запуском сервера. Однако, некоторые предпочитают управлять миграциями вручную, чтобы контролировать процесс. В нашем случае, т.к. среда офлайн, безопаснее мигрировать вручную.

docker-compose.yml для Django + PostgreSQL

Этот Compose-файл описывает два сервиса: веб-приложение и базу данных. Предполагается, что образы уже будут загружены (или доступны для сборки).

```
version: '3.8'

services:
  web:
    image: myproject:v2.3             # имя образа Django-приложения
(используйте актуальный тег версии)
    container_name: myproject_web     # удобочитаемое имя контейнера
(опционально)
    env_file: .env                    # файл с переменными окружения (напр.,
настройки БД, секретный ключ и т.д.)
    ports:
      - "8000:8000"                   # публикуем порт 8000 на хост (можно
изменить внешний порт при конфликте)
    depends_on:
      - db                            # контейнер web запускается после db
    restart: unless-stopped           # политика перезапуска: перезапускать
при сбоях
    # volumes:                        # (опционально) монтирование томов
    #   - static_data:/app/static     # пример: если хотим, чтобы статика
сохранялась вне контейнера
    #   - media_data:/app/media       # если есть медиа-файлы загружаемые
пользователями
```

```
  db:
    image: postgres:15-alpine         # образ PostgreSQL (должен совпадать с
тем, что использовался в разработке)
    container_name: myproject_db
    environment:
      - POSTGRES_DB=mydatabase        # имя создаваемой БД
      - POSTGRES_USER=dbuser          # пользователь БД
      - POSTGRES_PASSWORD=dbpass      # пароль для пользователя
    volumes:
      - db_data:/var/lib/postgresql/data   # хранилище данных БД
    restart: unless-stopped

volumes:
  db_data:
  # static_data:
  # media_data:
```

Примечания к Compose-файлу: - Директива `image` указывает использовать образ `myproject:v2.3`. В офлайн-среде этот образ должен быть загружен (`docker load`) до выполнения `docker-compose up`. Если образа нет, Compose будет пытаться его скачать или собрать – что нам не подходит. Поэтому важно обновлять значение тега и убедиться, что нужный образ присутствует. - `env_file: .env` предполагает, что рядом с docker-compose.yml лежит файл `.env`, где вы можете хранить переменные `DJANGO_SECRET_KEY`, `DJANGO_DEBUG=False`, `DJANGO_ALLOWED_HOSTS=...` и настройки подключения к базе (например, если не хотим хардкодить в settings.py, можно читать из env). В примере для базы мы сразу задали ENV для контейнера Postgres, а приложение `web` могло бы, например, ожидать `DATABASE_URL=postgres://dbuser:dbpass@db:5432/mydatabase` в .env – это возможно при использовании библиотек вроде dj-database-url. - Сервис `db` использует официальный образ Postgres 15 на Alpine. Учтите: если этот образ не был ранее загружен на продакшен, его тоже надо доставить (либо через `docker save` аналогично). Либо вы можете собрать свой образ Postgres, но обычно достаточно взять официальный. - Настройки среды `POSTGRES_*` и volume `db_data` обеспечивают, что при первом старте контейнера Postgres создаст базу `mydatabase` с пользователем `dbuser`/`dbpass` и сохранит данные в volume. При рестарте контейнера (например, при обновлении приложения) этот volume монтируется заново и данные (таблицы) сохраняются. **Внимание:** если вы меняете схему вне миграций (т.е. пересоздаете БД), вы можете удалить volume или задать другой volume name для новой версии, чтобы развернуть БД с нуля. - Проброс порта 8000 наружу позволяет заходить на веб из хоста (например, для проверки). В продакшене, если есть Nginx на хосте, можно не экспонировать порт, а линковать через сеть. Но для простоты демонстрации оставлено. - Политика `restart: unless-stopped` перезапустит контейнеры при сбоях или перезагрузке хоста, что удобно для продакшена. - Закомментированы volume для статики/медиа: это вариант, если хотим, чтобы они были доступны/постоянны на хосте. Но в нашем случае, статику можно запекать в образ, а медиа (если мало или не используются) – хранить в том же контейнере. Для боевого приложения, скорее всего, media-файлы пользователей стоит вынести на volume (чтобы обновление образа не затронуло загруженные файлы).

Compose-файл можно расширять: добавлять, например, сервис nginx, если нужно (который монтирует volume static_data и проксирует на web:8000). В офлайн-сети это возможно, но потребует образа nginx тоже. В нашем сценарии мы ограничились двумя контейнерами для компактности.

Заключение: Представленные инструкции и примеры предоставляют детальный план переноса Django-проекта с PostgreSQL из одной сети в другую в условиях отсутствия прямого соединения. Следуя этому плану – независимо от того, выберете вы Docker-контейнеризацию или традиционную установку – вы сможете регулярно и с минимальными трудозатратами обновлять продакшен-окружение, сохраняя синхронизацию с разработкой. Главное – тщательно подготовить переносимые артефакты, автоматизировать повторяющиеся шаги и тестировать процесс обновления в условиях, близких к боевым, чтобы избежать неожиданностей. При должной организации, офлайн-деплой перестанет быть проблемой, и ваша изолированная ЛВС будет своевременно получать все обновления проекта.

---

1 2 3 4 Как перенести и запустить Docker образы на автономной машине? — Хабр Q&A
https://qna.habr.com/q/1307766

5 6 Скачать пакеты python для оффлайн установки - Stack Overflow на русском
https://ru.stackoverflow.com/questions/693137/%D0%A1%D0%BA%D0%B0%D1%87%D0%B0%D1%82%D1%8C-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%8B-python-%D0%B4%D0%BB%D1%8F-%D0%BE%D1%84%D1%84%D0%BB%D0%B0%D0%B9%D0%BD-%D1%83%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8

7 8 9 Резервное копирование и восстановление PostgreSQL
https://wiki.dieg.info/rezervnoe_kopirovanie_i_vosstanovlenie_v_postgresql