Неделя 5. Лекция 3. Данные и Docker Compose: PostgreSQL, тома, сети

Часть 1 из 2.

После лекции: вы понимаете, где хранить данные контейнерных приложений и что с ними происходит на протяжении жизни тома, как PostgreSQL в контейнере инициализирует базу, как сервис подключается к ней и держит пул соединений, как контейнеры находят друг друга в сети, и умеете описать всё окружение одним файлом Docker Compose. **Инструменты:** Docker Desktop, Visual Studio 2026, PowerShell 7.

1. Проблема: данные пропадают

В лабе 2 проекты, созданные в контейнере, исчезли после `docker stop`. Причины две:

1. **Хранилище `InMemory`** держит данные в памяти процесса. Процесс завершился — данных нет.
2. **Даже файлы внутри контейнера временные.** Всё, что процесс записал на диск, попадает в **записываемый слой** контейнера. `docker rm` — и слой удалён.

Отсюда главное правило: **контейнер — одноразовый, данные — отдельно.** Контейнер сервиса можно в любой момент удалить и создать заново из образа (так и работает деплой в теме 5). Данные должны жить там, где удаление контейнера их не заденет.

Stateless и stateful. Наш API после перехода на PostgreSQL становится *stateless*: всё его состояние лежит в БД, а сам процесс можно убить, пересоздать из нового образа или запустить в трёх экземплярах, и ни один запрос не заметит разницы. БД — *stateful*: её ценность и есть данные на диске. Поэтому к двум контейнерам в одном `compose.yaml` относятся по-разному. Контейнер API пересоздают на каждый деплой без раздумий, а с контейнером БД так можно поступать только потому, что его файлы вынесены в том.

В продакшене БД часто вообще не держат в контейнерах на той же машине, что и сервисы: берут управляемую базу у облачного провайдера или отдельный сервер, где за бэкапы, репликацию и обновления отвечает DBA или провайдер. Для разработки, тестов и учебного стенда PostgreSQL в контейнере — стандартный выбор: за минуту получаем ту же версию, что в проде, и можем в любой момент выбросить её и начать с чистого листа.

2. Где хранить данные контейнера

Способ	Где физически	Живёт после `docker rm`	Для чего	У нас
Записываемый слой	Внутри контейнера	Нет	Временные файлы	—
Том (volume)	Каталог, которым управляет Docker	**Да**	Данные БД	`demo-pgdata` — файлы PostgreSQL
Bind mount	Папка на вашем диске	Да (это ваша папка)	Отдать контейнеру файлы с хоста: конфиги, скрипты	`db/` с учебными данными

Том монтируется в путь внутри контейнера: `-v demo-pgdata:/var/lib/postgresql`. PostgreSQL пишет свои файлы в `/var/lib/postgresql`, а на самом деле они лежат в томе. Новый контейнер с тем же томом видит те же данные.

Bind mount подключает папку хоста: `-v "${PWD}\db:/docker-entrypoint-initdb.d:ro"`. `:ro` — только чтение: контейнер не может испортить ваши файлы.

Где физически лежит том в Docker Desktop. Контейнеры Linux на Windows работают внутри служебной виртуальной машины на WSL 2. Тома живут в её файловой системе, а сама она хранится в виртуальном диске (`.vhdx`) в профиле пользователя. `docker volume inspect` показывает путь, но это путь внутри VM, а не на диске `C:`:

```
PS> docker volume inspect demo_pgdata --format '{{ .Mountpoint }}'
/var/lib/docker/volumes/demo_pgdata/_data
```

Отсюда два следствия. Том работает на родной для Linux файловой системе, поэтому быстрый. И он исчезает вместе с VM: сброс Docker Desktop к заводским настройкам или удаление его данных уносит все

тома.

Почему bind mount с Windows медленнее. Папка `C:\dev\demo-api\db` лежит на NTFS, и каждое обращение к ней из контейнера пересекает границу между VM и Windows через механизм общего доступа к файлам. Для одного `seed.sql`, который читается один раз, это незаметно. Для каталога данных PostgreSQL, который постоянно пишет мелкие блоки и вызывает `fsync`, — заметно медленнее, к тому же NTFS не хранит владельцев и права Linux, а PostgreSQL строго их проверяет. Правило простое: **данные БД — в именованном томе, с хоста — только то, что вы сами редактируете.**

Жизненный цикл тома не связан с жизненным циклом контейнера:

Действие	Что с томом
`docker volume create`, `-v имя:...` при первом `run`, `compose up`	Создаётся пустым
`docker stop`, `docker rm`, `compose down`	Остаётся нетронутым
Новый контейнер с тем же именем тома	Видит все прежние данные
`docker volume rm`, `compose down -v`	Удаляется вместе с данными
`docker volume prune`	Удаляются анонимные тома, не подключённые ни к одному контейнеру; с `-a` — и именованные

Сколько места занимает каждый том, показывает `docker system df -v`. Тома — первое, что растёт на машине разработчика, и первое, что хочется почистить, поэтому полезно заранее знать, какие из них можно удалять.

Есть и **анонимные тома**. В образе `postgres` объявлена инструкция `VOLUME /var/lib/postgresql`, поэтому контейнер, запущенный вообще без `-v`, всё равно получает том — со случайным именем. В п. 1 лабы данные именно в нём, но новый контейнер создаёт себе новый анонимный том и старого не видит. Старый остаётся лежать мусором:

```
PS> docker volume ls
DRIVER    VOLUME NAME
local     8c1f0e7a3b...        # анонимный том от контейнера без -v (имя сокращено)
local     demo-pgdata          # ручной стенд, части 1–2 лабы
local     demo_pgdata          # том compose-проекта demo
```

3. PostgreSQL в контейнере

Официальный образ `postgres` настраивается переменными окружения и при первом старте сам создаёт БД:

```
docker run -d --name demo-db `
  -e POSTGRES_DB=demo -e POSTGRES_USER=demo -e POSTGRES_PASSWORD=demo `
  -v demo-pgdata:/var/lib/postgresql `
  -v "${PWD}\db:/docker-entrypoint-initdb.d:ro" `
  -p 127.0.0.1:5432:5432 `
  postgres:18-alpine
```

Часть	Зачем
`-e POSTGRES_DB/USER/PASSWORD`	При первом старте создаются БД `demo` и пользователь `demo` с паролем
`-v demo-pgdata:/var/lib/postgresql`	Файлы БД на томе. **В PostgreSQL 18** путь именно такой; в версиях до 17 был `/var/lib/postgresql/data`
`-v ...db:/docker-entrypoint-initdb.d:ro`	Образ выполняет скрипты из этой папки при **первой** инициализации: наш `db/seed.sql` создаёт таблицы и учебные записи
`-p 127.0.0.1:5432:5432`	Порт БД доступен с вашего ПК (для Visual Studio), но не из сети

Что делает образ при старте. Точка входа образа — скрипт `docker-entrypoint.sh`. Он смотрит в каталог данных: в PostgreSQL 18 это `/var/lib/postgresql/18/docker`, подкаталог того самого тома. Версия в пути нужна, чтобы при переходе на следующую мажорную версию старые и новые файлы могли лежать в одном томе рядом. Дальше два сценария:

1. **Каталог пуст.** Скрипт запускает `initdb` (создаёт системные таблицы), поднимает временный сервер, доступный **только через локальный сокет**, создаёт пользователя и БД из `POSTGRES_*`, по алфавиту выполняет файлы `*.sql` и `*.sh` из `/docker-entrypoint-initdb.d`, останавливает временный сервер и запускает

настоящий, уже слушающий сеть.
2. **В каталоге уже есть БД.** Всё перечисленное пропускается, сразу стартует сервер.

В логах это выглядит так (фрагменты):

```
# первый старт на пустом томе
/usr/local/bin/docker-entrypoint.sh: running /docker-entrypoint-initdb.d/seed.sql
CREATE TABLE
INSERT 0 2
PostgreSQL init process complete; ready for start up.
... LOG:  database system is ready to accept connections

# следующий старт на том же томе
PostgreSQL Database directory appears to contain a database; Skipping initialization
... LOG:  database system is ready to accept connections
```

Строка `ready to accept connections` при первом старте встречается дважды: первый раз её пишет временный сервер. Это важно для проверки готовности (п. 7).

Первая инициализация происходит только на пустом томе. Если том уже содержит БД, образ пишет `Skipping initialization`: seed-скрипты и новый `POSTGRES_PASSWORD` **не применяются**. Отсюда частая ловушка: поменяли пароль в переменной, а БД ждёт старый. Пароль в работающей БД меняют SQL-командой `ALTER USER`, а на учебном стенде проще пересоздать том.

Из практики. В seed-скрипт добавили строку с опечаткой. На первом запуске скрипт упал посреди выполнения, контейнер остановился, но `initdb` к этому моменту уже отработал, и в томе осталась «полуготовая» БД: пользователь есть, половины таблиц нет. Опечатку исправили, запустили снова — в логе `Skipping initialization`, таблиц по-прежнему нет, и полчаса уходит на поиск ошибки в исправленном файле. Если seed упал, первым делом удаляют том и инициализируют заново.

Мажорная версия закреплена в теге (`18-alpine`), и это не формальность. Файлы данных PostgreSQL 17 не читаются сервером 18, поэтому переход между мажорными версиями — это перенос данных (`pg_upgrade` или выгрузка и загрузка), а не замена тега. Тег `latest` на БД однажды превратит обычный `docker compose pull` в неработающий стенд.

> **Демонстрация для преподавателя.** Жизнь тома и два сценария точки входа:
>
> ```
>         docker run -d --name pg-demo -e POSTGRES_PASSWORD=demo -v pg-demo:/var/lib/postgresql postgres:18-alpine
>         docker logs -f pg-demo                  # initdb, init process complete, ready — дважды; Ctrl+C
>         docker rm -f pg-demo; docker run -d --name pg-demo -e POSTGRES_PASSWORD=demo -v pg-demo:/var/lib/postgresql
> postgres:18-alpine
>         docker logs pg-demo 2>&1 | Select-String 'Skipping'
>         docker volume inspect pg-demo --format '{{ .Mountpoint }}'   # путь внутри VM, не на C:
> ```
>
> Уборка: `docker rm -f pg-demo; docker volume rm pg-demo`.

4. Переключаем сервис на PostgreSQL

Код сервиса уже умеет работать с PostgreSQL (лекция 1, п. 9). Нужны две настройки:

Настройка	Значение	Зачем
`Storage`	`Postgres`	Выбрать реализацию репозиториев с SQL
`ConnectionStrings:Db`	`Host=...;Port=5432;Database=demo;Username=demo;Password=demo`	Куда подключаться

Из Visual Studio — в `appsettings.Development.json` или переменными окружения в профиле запуска (`Properties/launchSettings.json` → `environmentVariables`). Сервис работает на вашем ПК и подключается к опубликованному порту: `Host=localhost;Port=5432`.

В контейнере — переменными окружения, не пересобирая образ:

```
        -e Storage=Postgres -e "ConnectionStrings__Db=Host=demo-db;Database=demo;Username=demo;Password=demo"
```

Строка подключения по частям. Это набор пар `ключ=значение`, который разбирает драйвер Npgsql:

Ключ	Смысл	Если не указан

`Host`	DNS-имя или IP сервера БД	Обязателен
`Port`	TCP-порт	`5432`
`Database`	Имя БД на сервере	Совпадает с именем пользователя
`Username`, `Password`	Учётные данные	—
`Timeout`	Сколько секунд ждать установки соединения	15
`Maximum Pool Size`	Предел пула соединений	100

Обратите внимание, что в строке для контейнера `Port` не указан. Сервис-контейнер ходит к БД напрямую по сети Docker, на внутренний порт `5432`, и опубликованный порт хоста (в том числе `15432`, если 5432 занят) его не касается.

Пул соединений. Установить соединение с PostgreSQL дорого: TCP-рукопожатие, аутентификация, а на сервере под каждое соединение запускается отдельный процесс. Открывать соединение на каждый HTTP-запрос и закрывать после него — значит тратить на это больше времени, чем на сам запрос. Поэтому в `Program.cs` регистрируется один на всё приложение `NpgsqlDataSource`:

```
builder.Services.AddNpgsqlDataSource(connectionString);
```

Он держит пул открытых соединений. Репозиторий вызывает `db.CreateCommand(...)`, выполняет запрос, и соединение возвращается в пул, а следующий запрос берёт его готовым. Сам код соединения не открывает и не закрывает. У пула есть и обратная сторона: у PostgreSQL по умолчанию `max_connections = 100`, а каждый экземпляр сервиса может открыть до `Maximum Pool Size` соединений. Три реплики с настройками по умолчанию теоретически способны исчерпать лимит сервера, поэтому при масштабировании размер пула считают.

Схема при старте: `DatabaseInitializer`. С `Storage=Postgres` сервис при старте вызывает `DatabaseInitializer.EnsureSchemaAsync` и пишет в лог `Хранилище: PostgreSQL`. Инициализатор в одной транзакции выполняет `CREATE TABLE IF NOT EXISTS` для `projects` и `tasks` и создаёт индекс. Та же схема есть в `db/seed.sql`, поэтому порядок не важен: кто пришёл первым, тот и создал таблицы. Первой строкой транзакции стоит

```
SELECT pg_advisory_xact_lock(4242);
```

Это **advisory lock** — «рекомендательная» блокировка по числу, которая ничего в БД сама не запрещает, но второй вызов с тем же числом ждёт, пока первая транзакция не завершится. Зачем она: если одновременно стартуют две реплики сервиса, их параллельные `CREATE TABLE IF NOT EXISTS` могут столкнуться в системном каталоге PostgreSQL, и одна из них упадёт с ошибкой уникальности, несмотря на `IF NOT EXISTS`. С блокировкой вторая реплика дождётся первой и увидит, что таблицы уже есть.

Миграции. `IF NOT EXISTS` хватает, пока схема только создаётся. Как только нужно добавить столбец в таблицу с данными, нужны **миграции**: пронумерованные скрипты изменения схемы, которые применяются по порядку, а БД помнит, какие уже выполнены (EF Core Migrations, Flyway, Liquibase, DbUp). В проде их обычно запускают отдельным шагом деплоя, до выкладки новой версии, а не при старте каждой реплики: так видно, прошла ли миграция, и её сбой не превращается в падение всех экземпляров сервиса. В курсе миграций нет, но знать, где проходит эта граница, полезно.

5. Сети: как контейнеры находят друг друга

`localhost` внутри контейнера — это сам контейнер. Сервис-контейнер со строкой `Host=localhost` ищет PostgreSQL у себя внутри и не находит.

Контейнеры в одной **пользовательской сети** находят друг друга **по имени** — Docker держит для них встроенный DNS:

```
flowchart LR
  H["Ваш ПК<br/>Visual Studio, браузер"] -->|localhost:8080| API
  H -->|localhost:5432| DB
  subgraph N["сеть demo-net"]
    API["контейнер api"] -->|demo-db:5432| DB["контейнер demo-db"]
  end
  DB --- V[("том demo-pgdata")]
```

```
docker network create demo-net
docker network connect demo-net demo-db
docker run -d --name api --network demo-net -p 8080:8080 -e ... demo-api:2.0
```

Кто подключается	Адрес БД
Сервис в Visual Studio на вашем ПК	`localhost:5432` — через опубликованный порт
Сервис в контейнере в той же сети	`demo-db:5432` — по имени контейнера

Как это устроено. Сеть типа `bridge` — виртуальный коммутатор внутри VM Docker: каждый подключённый контейнер получает свой IP в подсети сети. Контейнер, запущенный без `--network`, попадает в сеть по умолчанию, которая так и называется — `bridge`. У неё есть ограничение, из-за которого мы создавали `demo-net`: **в сети по умолчанию имена контейнеров не разрешаются**, общаться можно только по IP, а IP меняется при каждом пересоздании. В пользовательской сети Docker подставляет контейнерам свой DNS-сервер по адресу `127.0.0.11`. Он знает имена и IP всех контейнеров сети и пересылает остальные запросы внешнему DNS.

Публикация порта (`-p`) — отдельный механизм, он нужен только для доступа снаружи сети: с вашего ПК. Запись `127.0.0.1:5432:5432` читается как «адрес и порт хоста : порт контейнера». Контейнерам между собой публикация не нужна. В проде порт БД обычно не публикуют вовсе, у нас он открыт только на `127.0.0.1` ради Visual Studio и тестов. Без `127.0.0.1` порт слушал бы на всех интерфейсах ПК, и к учебной БД с паролем `demo` мог бы подключиться любой в той же Wi-Fi-сети.

В сети Compose у контейнера БД два DNS-имени: имя сервиса `db` и имя контейнера `demo-db-1`. В строках подключения используют имя сервиса: оно не зависит ни от имени проекта, ни от номера экземпляра. Проверить, во что имя разрешается, можно изнутри контейнера: `docker compose exec api getent hosts db` (п. 10 лабы).

> **Демонстрация для преподавателя.** Имена в пользовательской сети и в сети по умолчанию:
>
> ```
> docker network create net-demo
> docker run -d --name web1 --network net-demo nginx:alpine
> docker run --rm --network net-demo alpine nslookup web1   # Server: 127.0.0.11, адрес найден
> docker run -d --name web2 nginx:alpine                     # сеть по умолчанию
> docker run --rm alpine nslookup web2                       # имя не найдено
> ```
>
> Уборка: `docker rm -f web1 web2; docker network rm net-demo`.