Неделя 6. Лекция 3. Данные и Docker Compose: PostgreSQL, тома, сети

Часть 2 из 2. Начало лекции — в материалах предыдущей недели.

6. Docker Compose: всё окружение в одном файле

Чтобы вручную запустить сервис с БД, нужны сеть, том, bind mount, два контейнера и десяток флагов. Их легко забыть и нельзя отревьюить. **Compose** описывает то же самое декларативно в `compose.yaml`, который лежит в git рядом с кодом. Любой участник команды поднимает окружение одной командой.

Декларативность здесь главное. Вы описываете желаемое состояние, а `docker compose up` сравнивает его с тем, что уже запущено, и делает минимум действий: чего нет — создаёт, что изменилось в описании — пересоздаёт, что совпадает — не трогает. Поэтому `up` можно запускать повторно сколько угодно. Свои контейнеры, сети и тома Compose находит по меткам с именем проекта.

Разберём `compose.yaml` нашего проекта по частям. Начало и сервис БД:

```yaml
name: demo                                  # имя проекта: demo-db-1, demo_default, demo_pgdata

services:
  db:
    image: postgres:18-alpine
    environment:
      POSTGRES_DB: demo
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: ${DB_PASSWORD:-demo}
    ports:
      - "127.0.0.1:${DB_PORT:-5432}:5432"   # только для локальных тестов с хоста
    volumes:
      - pgdata:/var/lib/postgresql
      - ./db:/docker-entrypoint-initdb.d:ro # учебные данные при первом запуске
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -h 127.0.0.1 -U demo -d demo"]
      interval: 5s
      timeout: 3s
      retries: 10
    restart: unless-stopped

volumes:
  pgdata:
```

Здесь те же флаги, что в `docker run` из п. 3, но с именами. `name: demo` фиксирует имя проекта: без него Compose взял бы имя папки, и у двух студентов с разными папками получились бы разные тома. `pgdata` в секции `volumes` верхнего уровня объявляет именованный том, Docker создаст его как `demo_pgdata`. Путь `./db` отсчитывается от папки с `compose.yaml`, а не от текущей папки в терминале. `DB_PORT` меняет только порт на хосте, внутренний `5432` остаётся прежним. О `healthcheck` — в п. 7.

Сервис API:

```yaml
  api:
    image: ${API_IMAGE:-demo-api}:${API_TAG:-local}
    build:
      context: .
      args:
        APP_VERSION: ${API_TAG:-local}
    ports:
      - "8080:8080"
    environment:
      Storage: Postgres
      ConnectionStrings__Db: "Host=db;Database=demo;Username=demo;Password=${DB_PASSWORD:-demo}"
      Chaos__Enabled: "true"
    depends_on:
      db:
        condition: service_healthy
    restart: unless-stopped
```

Одновременно `image` и `build` означают: собрать из Dockerfile в текущей папке и назвать результат `demo-`

`api:local`. В теме 5 пайплайн подставит через `API_IMAGE` и `API_TAG` готовый образ из GHCR, и тот же файл будет разворачивать релиз. В строке подключения `Host=db` — имя сервиса, которое встроенный DNS разрешит в IP контейнера БД. Пароль один и тот же, `${DB_PASSWORD:-demo}`, в обоих сервисах, поэтому БД и сервис не могут разойтись.

Хвост сервиса API — лимиты и ротация логов, подробно о них в лекции 6:

```
    memswap_limit: 256M      # без этого контейнер уйдёт в swap вместо OOMKilled
    deploy:
      resources:
        limits:
          cpus: "1.0"
          memory: 256M
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
```

Ключ	Суть
`services`	Контейнеры приложения; имя сервиса = DNS-имя в сети (`Host=db`)
`build` / `image`	Собрать из Dockerfile / взять готовый образ; вместе — собрать и назвать
`environment`	Переменные окружения контейнера
`${VAR:-default}`	Подстановка из окружения или файла `.env` рядом с compose-файлом
`depends_on.condition`	`service_started` (по умолчанию) или `service_healthy`
`healthcheck`	Как понять, что сервис готов
`volumes`	Именованные тома и bind mounts
`restart`	Политика перезапуска (лекция 6)
`deploy.resources.limits`, `logging`	Лимиты ресурсов и ротация логов (лекция 6)

Сеть Compose создаёт сам — `<проект>_default` — и подключает к ней все сервисы. Это пользовательская сеть, поэтому имена работают. Контейнеры называются `<проект>-<сервис>-<номер>`: `demo-api-1`, `demo-db-1`.

Переменные и `.env`. Перед запуском Compose подставляет значения во все `${...}` в файле. Значение ищется сначала в переменных окружения вашего терминала, потом в файле `.env` рядом с `compose.yaml`; если нигде нет, берётся значение после `:-`. Значит, `$env:DB_PORT = "15432"` в PowerShell перебьёт строку из `.env`. Вариант `${VAR:?текст}` вместо значения по умолчанию остановит запуск с ошибкой, если переменная не задана, — так делают для обязательных секретов. Не путайте `.env` с ключом `env_file:` в описании сервиса: `.env` используется только для подстановки в сам файл, а `env_file:` передаёт переменные внутрь контейнера.

Секреты. Пароль не пишут в `compose.yaml`. Он лежит в `.env` рядом с файлом (`.env` в `.gitignore`), а в git коммитят шаблон `.env.example`. Итоговый файл с подставленными значениями показывает `docker compose config` — первое, что стоит запустить, когда «переменная не подхватилась».

Compose как документация окружения. `compose.yaml` отвечает на вопросы, которые раньше жили в устной традиции команды и в устаревшей вики: какая версия PostgreSQL, на каком порту, откуда учебные данные, какие переменные нужны сервису. Этот файл невозможно «забыть обновить»: если он неверен, окружение не поднимется. Новый участник команды клонирует репозиторий, копирует `.env.example` и через пару минут работает с тем же стендом, что и все.

7. Порядок старта и готовность

Сервису нужна работающая БД: при старте он сразу выполняет `EnsureSchemaAsync`, и если БД недоступна, процесс падает с исключением. Compose по умолчанию стартует сервисы параллельно, а `depends_on` задаёт порядок. Вопрос в том, чего именно ждать:

- `condition: service_started` — ждать, пока контейнер БД **запустится**. Процесс PostgreSQL существует, но может ещё выполнять `initdb` и seed.
- `condition: service_healthy` — ждать, пока **healthcheck** БД не станет успешным.

Healthcheck — команда, которую Docker периодически выполняет внутри контейнера. Код выхода 0 — проверка прошла, иначе нет. После `retries` неудач подряд контейнер помечается `unhealthy`. Для PostgreSQL штатная проверка — утилита `pg_isready`.

Почему `-h 127.0.0.1`. Помните временный сервер из п. 3, который во время инициализации слушает только локальный сокет? `pg_isready` без `-h` подключается как раз через сокет и получает «готово» в тот момент, когда seed ещё выполняется, а через сеть к БД подключиться нельзя. С `-h 127.0.0.1` проверка идёт по TCP, как у настоящего клиента, и пройдёт только после запуска основного сервера.

```
sequenceDiagram
  participant C as docker compose up --wait
  participant DB as db
  participant API as api
  C->>DB: создать и запустить
  DB->>DB: initdb, временный сервер<br/>только сокет, seed.sql
  C->>DB: pg_isready -h 127.0.0.1
  DB-->>C: нет ответа по TCP
  DB->>DB: основной сервер слушает сеть
  C->>DB: pg_isready -h 127.0.0.1
  DB-->>C: accepting connections, healthy
  C->>API: запустить
  API->>DB: EnsureSchemaAsync
  API-->>C: HEALTHCHECK /health/live, healthy
  C-->>C: --wait: все сервисы healthy, выход
```

Чего `depends_on` не делает. Он работает только в момент `docker compose up`. Если БД перезапустилась через час, Compose об этом не узнает, и сервис должен сам пережить разрыв соединения. У нас это обеспечено двумя вещами. Пул Npgsql выбрасывает сломанные соединения и открывает новые, поэтому после возврата БД запросы снова проходят. А если БД недоступна при старте сервиса, процесс падает, и `restart: unless-stopped` поднимает его заново, пока БД не появится. Надёжные сервисы вообще не рассчитывают на порядок старта и повторяют подключение сами, потому что в Kubernetes и других оркестраторах `depends_on` нет.

Healthcheck контейнера и готовность сервиса — разные вещи. У API есть два эндпоинта: `/health/live` («процесс жив») и `/health/ready` («готов обслуживать запросы», включая доступность БД). `HEALTHCHECK` в Dockerfile (лекция 2) проверяет `/health/live`. Что из этого следует, когда БД падает, вы увидите в п. 10 лабы, а почему выбран именно этот эндпоинт, разберём в лекции 6.

8. Команды Compose

Команда	Что делает
`docker compose up -d --build --wait`	Собрать, создать, запустить и дождаться `healthy`
`docker compose ps` / `logs -f api`	Состояние / логи
`docker compose exec db psql -U demo -d demo`	Команда внутри сервиса
`docker compose stop` / `start`	Остановить / запустить без удаления
`docker compose down`	Удалить контейнеры и сеть. **Тома остаются**
`docker compose down -v`	**Удалить и тома: данные пропадут**
`docker compose config`	Итоговый файл с подставленными переменными

Все команды работают с проектом из текущей папки: Compose ищет `compose.yaml` в ней, определяет имя проекта и по меткам находит свои объекты. Запущенная из другой папки `docker compose ps` покажет пустой список, хотя контейнеры работают.

Из практики. Разработчик переименовал папку проекта или скопировал репозиторий в новое место, а в `compose.yaml` не было `name:`. После `up` база оказалась пустой, «данные пропали». На самом деле имя проекта поменялось вместе с папкой, Compose создал новый том с новым префиксом, а старый спокойно лежит в `docker volume ls` со всеми данными. Отсюда строка `name: demo` в нашем файле.

9. Бэкап и обслуживание

Том защищает данные от удаления контейнера, но не от `down -v`, ошибочного `DELETE` без `WHERE`, сброса Docker Desktop или поломки диска. Для этого нужен **бэкап** — копия, лежащая в другом месте.

Копировать файлы из тома работающей БД нельзя: PostgreSQL пишет их постоянно, и копия окажется несогласованной. Штатный инструмент — `pg_dump`, он делает согласованный снимок одной БД, не останавливая её:

```
docker compose exec db pg_dump -U demo -d demo -Fc -f /tmp/demo.dump   # снимок внутри контейнера
docker compose cp db:/tmp/demo.dump .\demo.dump                        # забрать на хост
```

Восстанавливают его утилитой `pg_restore`, предварительно вернув файл в контейнер:

```
docker compose cp .\demo.dump db:/tmp/demo.dump
docker compose exec db pg_restore -U demo -d demo --clean --if-exists /tmp/demo.dump
```

`--clean` удаляет существующие объекты перед созданием, иначе восстановление в непустую БД упрётся в уже существующие таблицы. Формат `-Fc` — сжатый архив PostgreSQL; без флага получится обычный SQL-скрипт. Бэкап, который ни разу не восстанавливали, бэкапом считать рано: проверка восстановления — такая же часть процесса, как сам снимок. В проде к этому добавляются расписание, хранение копий вне машины и непрерывное архивирование журнала, но для стенда разработчика достаточно `pg_dump` перед рискованным экспериментом.

10. Compose в разработке

Обычный рабочий режим разработчика: **зависимости — в контейнерах, свой код — в IDE с отладчиком.**

```
docker compose up -d --wait db     # только БД
# F5 в Visual Studio — сервис на вашем ПК подключается к localhost:5432
```

Если API из compose уже запущен, он занимает порт 8080: остановите его (`docker compose stop api`), и Visual Studio запустит сервис на том же порту. Перед проверкой «как в проде» поднимают весь стек: `docker compose up -d --build --wait`. Флаг `--build` здесь обязателен: без него Compose возьмёт уже существующий образ `demo-api:local`, и вы будете проверять вчерашний код.

Типичные ошибки

- **БД без тома:** `docker rm` — и все проекты и задачи исчезли. Данные остались в анонимном томе, но новый контейнер его не видит.
- **Каталог данных PostgreSQL в bind mount с Windows:** медленно, и возможны ошибки прав доступа. Данные БД — в именованном томе.
- **`Host=localhost`** в строке подключения сервиса, работающего в контейнере.
- **Контейнеры в сети по умолчанию** и обращение по имени: в сети `bridge` имена не разрешаются.
- **Порт занят** (на ПК уже стоит PostgreSQL): `ports are not available`. Решение — другой порт хоста: `127.0.0.1:15432:5432`, в compose — `${DB_PORT:-5432}`. Строку подключения сервиса-контейнера это не меняет.
- **Сменили `POSTGRES_PASSWORD` (или `DB_PASSWORD` в `.env`) или seed, а ничего не изменилось:** на непустом томе инициализация не выполняется. Сервис получает новый пароль, БД ждёт старый — `28P01: password authentication failed`.
- **Seed упал на первом запуске:** том остался полуинициализированным, повторного seed не будет. Удалить том и начать заново.
- **`depends_on` без `condition: service_healthy`:** сервис стартует раньше, чем БД готова, и падает.
- **Healthcheck `pg_isready` без `-h 127.0.0.1`:** «готово» наступает во время инициализации.
- **`down -v` «на всякий случай»** или `docker volume prune -a` для освобождения места: тестовые данные потеряны.
- **Тег `latest` у образа БД:** следующая мажорная версия не прочитает старые файлы.
- **Пароль в `compose.yaml` вместо `.env`.**

Вопросы для самопроверки

1. Почему данные пропали в лабе 2 и почему том решает проблему?
2. Чем том `demo-pgdata` отличается от папки `db/`, подключённой через bind mount? Почему каталог данных БД не держат в bind mount?
3. Почему `seed.sql` не выполнится повторно при пересоздании контейнера? Что делает точка входа образа на пустом и на непустом томе?
4. Какой адрес БД у сервиса в Visual Studio, а какой — у сервиса в контейнере? Почему разный?
5. Зачем сервису пул соединений и чем грозит слишком большой пул при нескольких репликах?
6. Зачем в `DatabaseInitializer` advisory lock, если таблицы создаются с `IF NOT EXISTS`?
7. Почему контейнеры в сети по умолчанию не находят друг друга по имени, а в `demo_default` находят?
8. Что переживёт `docker compose down`, а что — `docker compose down -v`?

9. Чем `service_started` отличается от `service_healthy` и зачем в healthcheck `-h 127.0.0.1`?
10. Что случится с сервисом, если БД перезапустится через час после `compose up`? Поможет ли `depends_on`?
11. В `.env` написано `DB_PORT=5432`, а в терминале задано `$env:DB_PORT = "15432"`. Какой порт получит БД и как это проверить?

Практика: лаба 3

Запустите PostgreSQL в контейнере с томом и учебными данными, переключите сервис на него — сначала в Visual Studio, потом в контейнере. Разберёте две типичные ошибки подключения из контейнера, соедините контейнеры сетью, а затем опишете всё окружение в `compose.yaml`.