Лекция 14: Управляем многоконтейнерными приложениями с помощью Docker Compose

Введение: От одного контейнера к целой системе

Здравствуйте! На прошлой лекции мы совершили важный прорыв: научились упаковывать наше Node.js-приложение в Docker-контейнер. Теперь у нас есть гарантия, что сам код будет работать одинаково в любой среде.

Но давайте посмотрим на наше приложение целиком. Оно состоит из нескольких частей:

1. **API-сервер** (наш Node.js код).
2. **База данных** (PostgreSQL).
3. **Кэш** (Redis).

Запускать каждый из этих компонентов вручную отдельными docker run командами — очень неудобно. Нам пришлось бы вручную создавать для них общую сеть, прописывать адреса и управлять их жизненным циклом по отдельности. Это сложно, долго и чревато ошибками.

Нам нужен инструмент, который позволит описать **всю нашу систему как единое целое** и управлять ей одной командой. Этот инструмент — **Docker Compose**.

1. Что такое Docker Compose?

Docker Compose — это инструмент для определения и запуска многоконтейнерных Docker-приложений.

Ключевая идея: Вы описываете все сервисы вашего приложения, их конфигурацию, сети и тома в **одном-единственном файле** — docker-compose.yml. После этого вы можете одной командой docker-compose up создать и запустить всю систему целиком.

Аналогия:

- **Dockerfile** — это **чертеж одной детали** (например, двигателя).
- **docker-compose.yml** — это **сборочный чертеж всего автомобиля**. Он показывает, как соединить двигатель, шасси, колеса (другие сервисы) и какие между ними должны быть "трубки" и "провода" (сети).

2. Анатомия файла docker-compose.yml

docker-compose.yml — это файл в формате YAML, который имеет простую и понятную структуру. Давайте разберем его ключевые секции.

```
version: '3.8' # Версия синтаксиса Docker Compose

services: # Главная секция, где описываются все наши контейнеры

  # Имя первого сервиса (нашего приложения)
  app:
    build: . # Указываем, что образ нужно собрать из Dockerfile в текущей папке
    ports:
      - "3000:3000" # Проброс портов [хост]:[контейнер]
    volumes:
```

```
      - .:/app # Монтирование тома (об этом позже)
    environment:
      - DATABASE_URL=... # Переменные окружения
    depends_on:
      - db # Указываем, что этот сервис зависит от сервиса 'db'

  # Имя второго сервиса (базы данных)
  db:
    image: postgres:13 # Используем готовый образ с Docker Hub
    environment:
      - POSTGRES_PASSWORD=password
    volumes:
      - pgdata:/var/lib/postgresql/data # Именованный том для хранения данных

volumes: # Секция для объявления именованных томов
  pgdata:
```

Давайте разберем самые важные директивы.

- services: Главный блок, где каждый ключ (app, db) — это имя одного из контейнеров в нашей системе.
- build vs image:
    - build: . говорит Compose, что для этого сервиса нужно **собрать образ** из Dockerfile в текущей директории.
    - image: postgres:13 говорит, что нужно **скачать готовый образ** postgres с тегом 13 из Docker Hub.
- ports: Пробрасывает порты из контейнера на нашу локальную машину, чтобы мы могли к ним обратиться.
- environment: Позволяет передать переменные окружения внутрь контейнера. Это основной способ конфигурации.
- depends_on: Управляет порядком запуска. app не будет запущен, пока не запустится сервис db.
- volumes: Самая важная директива для работы с данными.

3. Управление данными: Volumes (Тома)

Проблема: Контейнеры по своей природе **эфемерны**. Если мы удалим и пересоздадим контейнер с PostgreSQL, **все данные в базе данных будут потеряны**, потому что они хранились внутри записываемого слоя этого контейнера.

Решение: Volumes (Тома)

Том — это специальный механизм Docker для **хранения данных за пределами жизненного цикла контейнера**. Это как "внешний жесткий диск", который мы "подключаем" к нашему контейнеру.

- **Как работает:** Мы говорим Docker: "Содержимое папки /var/lib/postgresql/data внутри контейнера db на самом деле должно храниться вот здесь, на моем хост-компьютере, в специальной области, управляемой Docker (pgdata)".

- **Результат:** Теперь, если мы остановим, удалим и заново запустим контейнер с базой данных (docker-compose down и docker-compose up), он "подцепит" тот же самый том pgdata, и все наши данные окажутся на месте.

Типы томов:

- **Именованные тома (Named Volumes):** pgdata:/var/lib/postgresql/data. Docker сам управляет этим томом. **Это предпочтительный способ для хранения данных (БД, кэш).**
- **Bind Mounts:** .:/app. Мы "пробрасываем" папку с нашего компьютера (.) внутрь контейнера (/app). Любое изменение в файлах на нашем компьютере **мгновенно** отражается внутри контейнера. **Это идеальный способ для локальной разработки**, так как нам не нужно пересобирать образ после каждого изменения кода.

4. Сетевое взаимодействие (Networking)

Когда мы запускаем docker-compose up, Docker Compose автоматически создает **виртуальную сеть** для всех сервисов, описанных в файле.

- Каждый сервис получает свое DNS-имя, **равное имени сервиса** в docker-compose.yml.
- Это значит, что наш сервис app может обратиться к базе данных просто по адресу db, а к кэшу — по адресу redis.

В нашем Dockerfile мы передавали переменную окружения:
DATABASE_URL="postgresql://user:password@postgres-service:5432/todo_db"
Именно так наше приложение, работая внутри контейнера app, сможет найти контейнер db в общей сети.

5. Практика: Собираем наше приложение с помощью Docker Compose

Давайте создадим docker-compose.yml для нашего To-Do приложения, которое состоит из трех сервисов: app, db, redis.

docker-compose.yml:

```
version: '3.8'

services:
  # Сервис нашего приложения
  app:
    build: .
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - /app/node_modules # Исключаем node_modules из монтирования, чтобы
использовать те, что внутри образа
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/todo_db
      - REDIS_URL=redis://redis:6379
    depends_on:
      - db
      - redis

  # Сервис базы данных PostgreSQL
```

```
  db:
    image: postgres:14-alpine
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=todo_db
    volumes:
      - pgdata:/var/lib/postgresql/data
    ports: # Пробрасываем порт для удобства подключения извне (например, через
DBeaver)
      - "5432:5432"

  # Сервис кэша Redis
  redis:
    image: redis:7-alpine
    volumes:
      - redisdata:/data

volumes:
  pgdata:
  redisdata:
```

Теперь, находясь в корне проекта, мы можем выполнить всего две команды:

```
# Собрать образы и запустить все сервисы в фоновом режиме
docker-compose up --build -d

# Остановить и удалить все контейнеры и сети
docker-compose down
```

Заключение

Сегодня мы сделали следующий большой шаг в мире контейнеризации. Мы научились:

- Описывать сложные, многокомпонентные приложения в виде простого YAML-файла.
- Управлять жизненным циклом всей системы одной командой.
- Обеспечивать сохранность данных с помощью volumes.
- Настраивать сетевое взаимодействие между контейнерами.

Docker Compose — это ваш незаменимый инструмент для **локальной разработки и тестирования**. Он позволяет любому новому разработчику в команде поднять полную копию рабочего окружения за считанные минуты.

На следующей лекции мы перейдем от локальной машины к "облакам" и поговорим о том, как автоматизировать процесс сборки и тестирования наших Docker-контейнеров с помощью **CI/CD**.