МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»  
(ФГБОУ ВО «ВГТУ», ВГТУ)

Факультет информационных технологий и компьютерной безопасности

Кафедра компьютерных интеллектуальных технологий проектирования

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

По дисциплине: «Системы хранения и обработки данных»

Тема: «Развертывание СУБД Postgres с использованием средств автоматизации развертывания и управления приложениями»

Выполнил работу студент группы змИИВТ-241: Фролов Д.С.

подпись, дата

Принял: Короленко В.В.

подпись, дата

Воронеж 2024

Лабораторная работа №6

«Развертывание СУБД Postgres с использованием средств автоматизации развертывания и управления приложениями»

1. Цель лабораторной работы: изучить основы работы программного средства для автоматизации развертывания и управления приложениями в средах с поддержкой контейнеризации Docker на примере решения задачи развертывания СУБД Postgres.

2. Задачи лабораторной работы:

- установка приложения для работы с Docker-контейнерами;

- установка программного средства для работы с СУБД DBeaver;

- скачивание docker-образа с СУБД Postgres;

- создание Dockerfile;

- создание образа на основе Dockerfile;

- запуск контейнера для развертывания СУБД Postgres;

- подключение к работающему контейнеру и запуск интерфейса psql;

- создание контейнера с томом (volume);

- создание контейнера с использованием файла docker-compose.yml.

3. Результаты лабораторной работы

Зарегистрируемся на Docker Hub. Скачаем и установим Docker Desktop. Скачаем и установим средство для работы с СУБД DBeaver. Скачаем docker-образ с СУБД postgres с помощью команды для консоли «docker pull postgres» (рисунок 1).

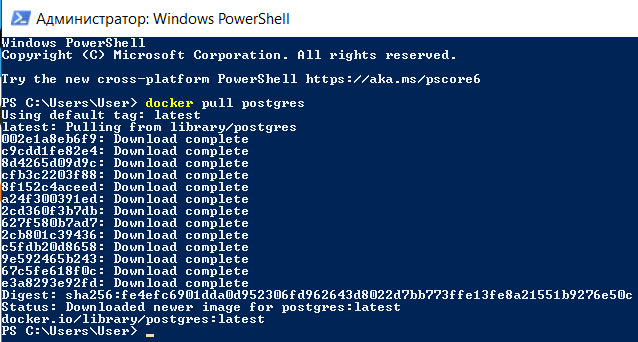


Рисунок 1 – Установка docker-образа с СУБД postgres

Создадим Dockerfile в Visual Studio Code со следующим содержанием (рисунок 2).

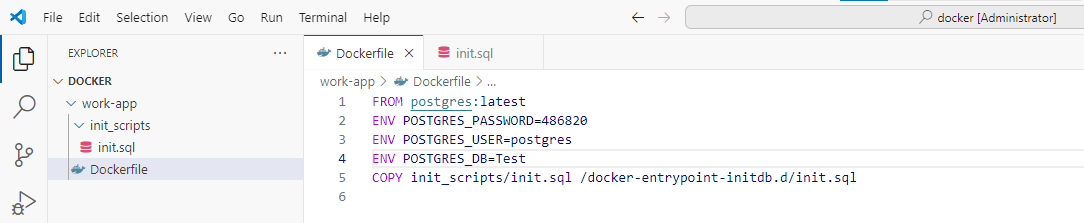


Рисунок 2 – Создание DockerFile

Опишем каждую строку Dockerfile:

* FROM postgres:latest – указывает, что базовым образом для нашего Dockerfile будет последняя версия образа PostgreSQL. Это значит, что наш образ, будет основан на официальном образе PostgreSQL;
* ENV POSTGRES\_PASSWORD = 486820 – устанавливает переменную окружения POSTGRES\_PASSWORD, задавая пароль для пользователя базы данных PostgreSQL;
* ENV POSTGRES\_USER = postgres – устанавливает переменную окружения POSTGRES\_USER, задавая имя пользователя базы данных PostgreSQL;
* ENV POSTGRES\_DB = Test – устанавливает переменную окружения POSTGRES\_DB, задавая имя база данных;
* COPY init\_scripts/init.sql/docker-entrypoint-initdb.d/init.sql – копирует файл init.sql из локальной директории init\_scripts в директорию /docker-entrypoint-initdb.d внутри контейнера.

Создадим файл init.sql (рисунок 3).

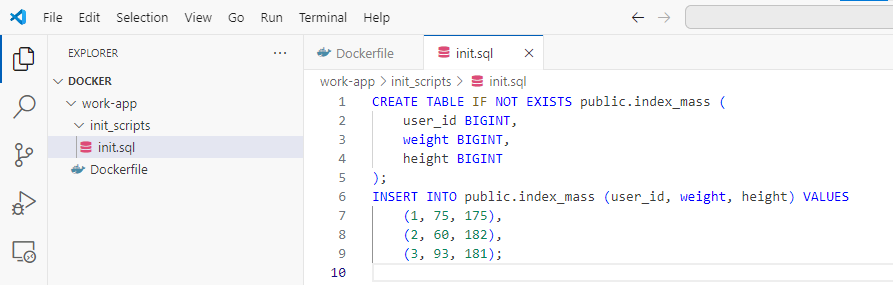


Рисунок 3 – Создание файла init.sql

Опишем каждую строку файла init.sql:

* CREATE TABLE IF NOT EXISTS public.index\_mass – эта команда создает новую таблицу «index\_mass» в схеме public, если она еще не существует. Это предотвращает ошибку при попытке создать таблицу, которая уже существует;
* user\_id BIGINT – определяет столбец user\_id типа BIGINT;
* weight BIGINT – определяет столбец weight типа BIGINT;
* height BIGINT – определяет столбец height типа BIGINT;
* INSERT INTO public.index\_mass (user\_id, weight, height) VALUES – эта команда вставляет данные в таблицу index\_mass;
* (1, 75, 175) – добавляет запись с user\_id = 1, weight = 75, height = 175;
* (2, 60, 182) – добавляет запись с user\_id = 2, weight = 60, height = 182;
* (3, 93, 181) – добавляет запись с user\_id = 3, weight = 93, height = 181.

Создадим образ с помощью команды «docker build -t d\_img:latest .». Команду запустим из каталога, в котором находится Dockerfile – C:\Users\User\Desktop\docker\postgres-app>. В результате запустится Dockerfile и будет создан docker-образ «d\_img». Результат создания образа представлен на рисунках 4-5.



Рисунок 4 – Создание образа

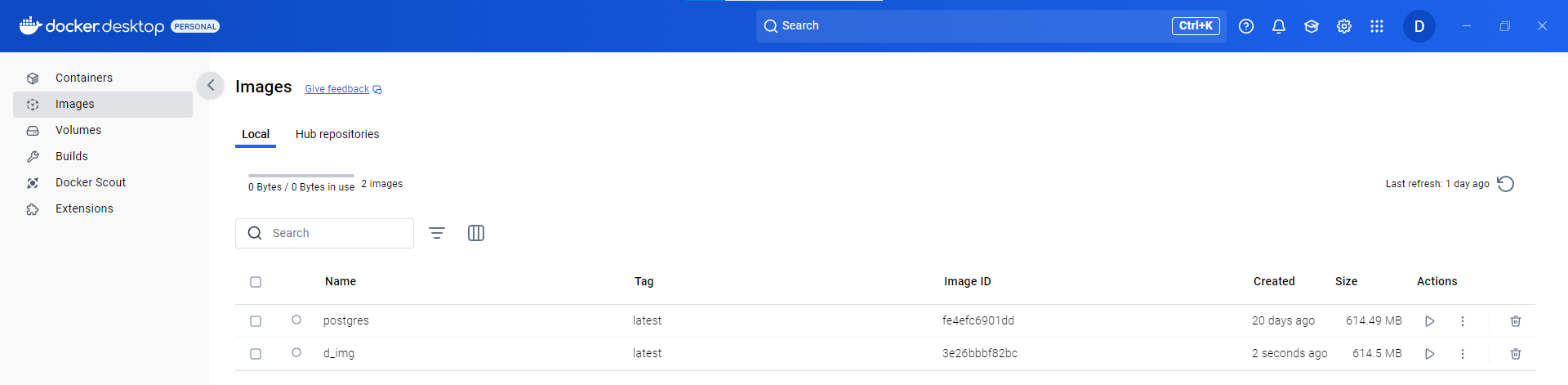


Рисунок 5 – Результат создания образа

Чтобы воспользоваться созданной СУБД запустим контейнер с именем «d\_cont» на основе образа «d\_img:latest» с помощью команды: «docker run -d -p 5432:5432 --name d\_cont d\_img:latest». Результат представлен на рисунках 6-7.

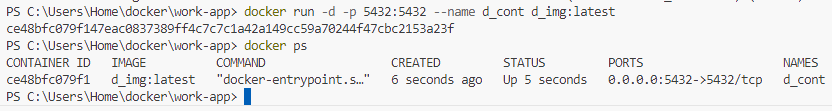


Рисунок 6 – Запуск контейнера «d\_cont»

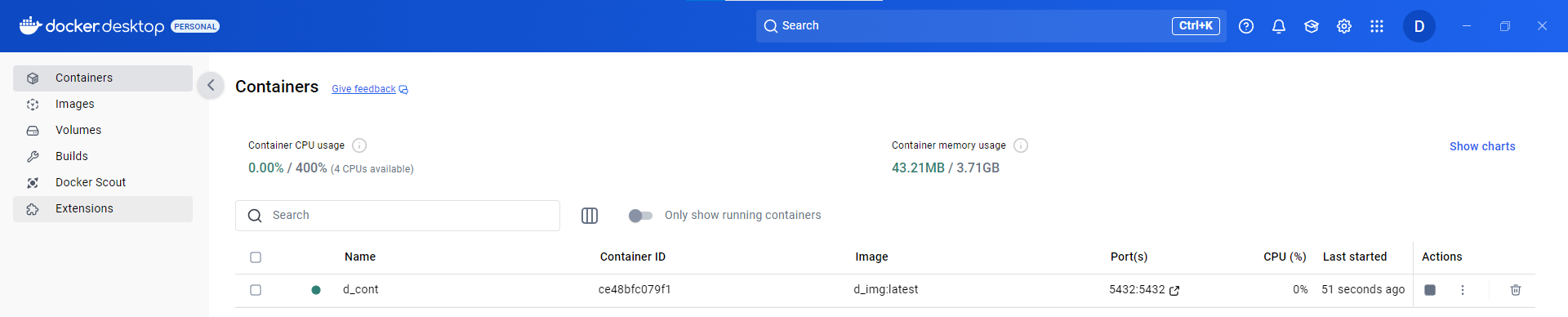


Рисунок 7 – Результат запуска контейнера

Опишем использованную команду:

* docker run – команда для создания и запуска нового контейнера на основе указанного образа;
* -d (--detach) – запускает контейнер в фоновом режиме. Контейнер будет запущен, но вы вернетесь к командной строке и сможете выполнять другие команды;
* -p 5432:5432 – позволяет явно сопоставить один порт или диапазон портов в контейнере с хостом. Синтаксис: -p<порт\_хоста>:<порт\_контейнера>. Порт 5432 на вашей локальной машине (хосте) будет связан с портом 5432 внутри контейнера. Порт 5432 является стандартным портом для PostgreSQL. Он позволяет приложениям на вашей локальной машине подключаться к PostgreSQL, работающему внутри контейнера, как если бы он был установлен локально;
* --name d\_cont – присваивает контейнеру имя d\_cont. Вместо автоматического присвоения случайного имени, контейнер будет иметь заданное вами имя;
* d\_img:latest – указывает образ Docker, на основе которого будет создан и запущен контейнер. Docker ищет локально образ d\_img с тегом latest. Если образ не найден локально, Docker попытался бы загрузить его из реестра (Docker Hub).

При запуске команды Docker проверяет наличие локального образа с именем «d\_img» и тегом «latest». На основе образа «d\_img:latest» запускается новый контейнер. Благодаря флагу -d контейнер запускается в фоновом режиме. Флаг –p связывает порт 5432 вашей локальной машины с портом 5432 внутри контейнера. Это необходимо для доступа к PostgreSQL, который по умолчанию работает на порту 5432. Теперь приложения на локальной машине могут подключаться к базе данных внутри контейнера через порт 5432. Контейнер получает имя «d\_cont» благодаря флагу «--name d\_cont». При первом запуске контейнера выполняется скрипт «init.sql», который мы скопировали в директорию «/docker-entrypoint-initdb.d/» в Dockerfile. PostgreSQL автоматически выполняет все скрипты «.sql» из этой директории при инициализации. В результате создается таблица «public.index\_mass» и в нее вставляются начальные данные.

Проверим через DBeaver развернутую СУБД и подключим базу данных с параметрами, указанными в Dockerfile. Для этого запустим DBeaver и создадим новое подключение. В списке доступных баз данных выберем PostgreSQL (рисунок 8).

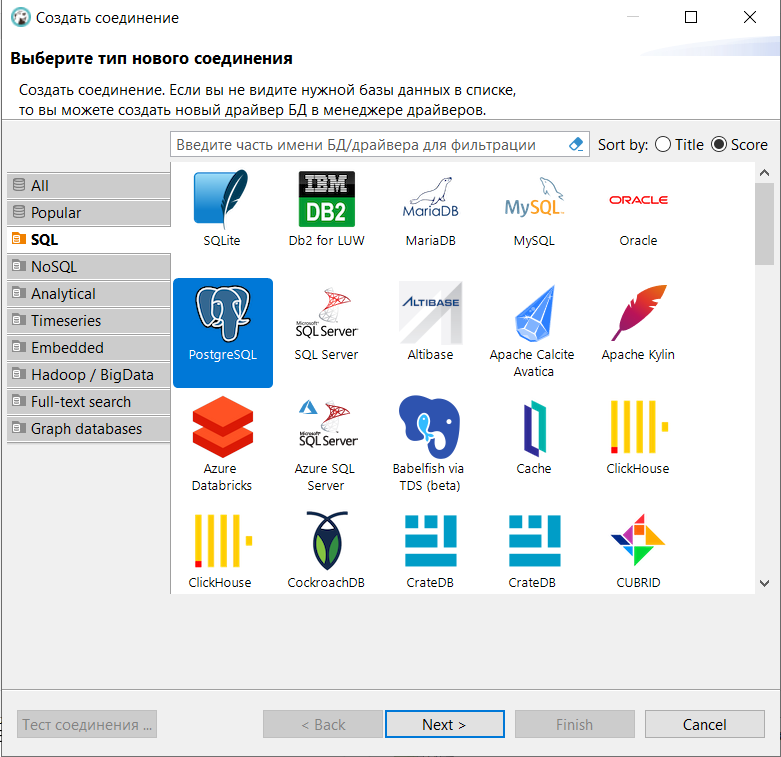


Рисунок 8 – Выбор PostgreSQL в качестве базы данных

Далее нажимаем «Next», выполняем настройки соединения и нажимаем на кнопку «Тест соединения» (рисунок 9).

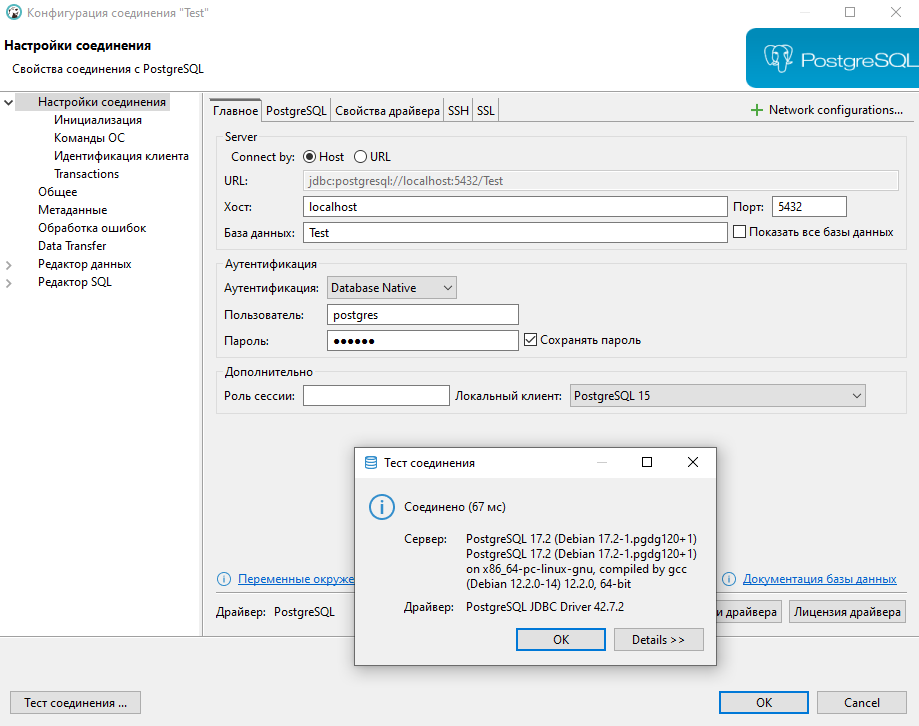


Рисунок 9 – Настройки и тест соединения

Результат проверки развернутой СУБД через DBeaver представлен на рисунке 10.

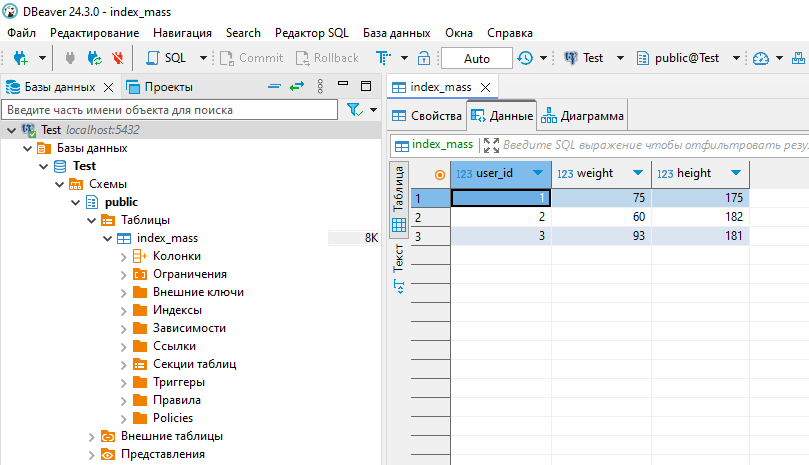


Рисунок 10 – Результат проверки развернутой СУБД через DBeaver

Напишем специальную команду docker, которая позволит подключаться к работающему контейнеру, запускать интерфейс psql и вносить новые данные «на лету»: «docker exec -it d\_cont psql -U postgres -d Test». Теперь есть возможность работать с базой данных из консоли. Выведем в консоль список имеющихся баз данных. Результат выполнения данной команды и вывод в консоль баз данных представлен на рисунке 11.

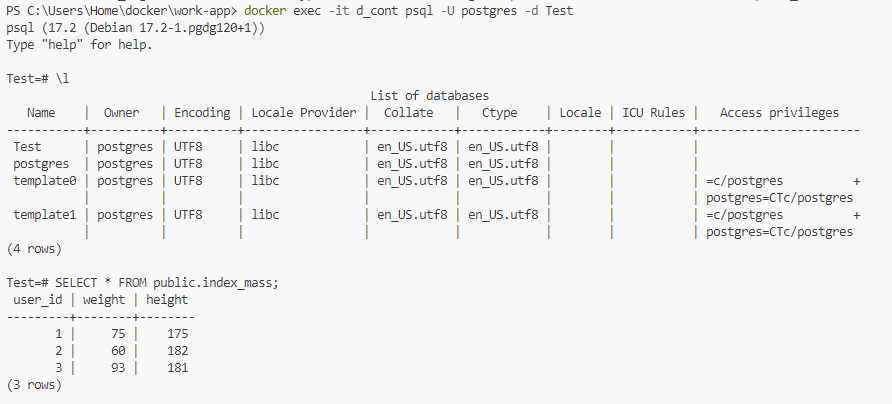


Рисунок 11 – Результат выполнения данной команды и вывод в консоль баз данных

Опишем использованную команду:

* docker exec – команда для выполнения команд внутри работающего контейнера;
* флаг -it – для того чтобы работала интерактивная среда прямо в терминале;
* d\_cont – имя вашего Docker-контейнера, внутри которого будет выполнена команда;
* psql – утилита командной строки PostgreSQL для взаимодействия с базой данных;
* -U postgres – указывает имя пользователя для подключения к базе данных. В данном случае – postgres;
* -d Test – указывает имя базы данных для подключения. В данном случае – Test.

При выполнении данной команды Docker запускает команду «psql –U postgres –d Test» внутри контейнера d\_cont. Благодаря флагам -it мы получаем интерактивный терминал и можем вводить команды в psql. Мы подключаемся к базе данных Test под пользователем postgres. Теперь мы можем управлять базой данных из консоли. После подключения к интерфейсу psql мы вывели список всех баз данных, используя команду «\1». Далее мы сделали запрос к таблице «index\_mass».

Чтобы данные, измененные в процессе работы контейнера, были доступны после удаления контейнера (во вновь созданном контейнере) необходимо создать том (volume). Для добавления в Dockerfile инструкции для вызова создания контейнера с volume необходимо в Dockerfile добавить строку «VOLUME /d\_data:/var/lib/postgresql/data». При этом локально будет создан том (volume) в папке «/d\_data», в котором будут сохраняться изменения, аналогичные тем, которые происходят в контейнере в папке «/var/lib/postgresql/data». Добавление строки в Dockerfile представлено на рисунке 12.

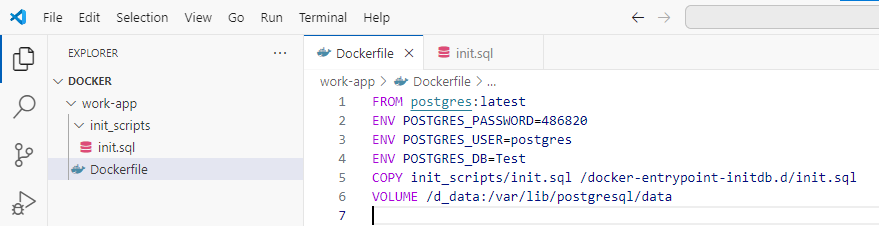


Рисунок 12 – Добавление в Dockerfile инструкции для создания тома

Далее необходимо удалить созданный контейнер. Для этого выполним команды «docker stop 65b89770df40» и «docker container prune». Результат удаления контейнера представлен на рисунках 13-14.

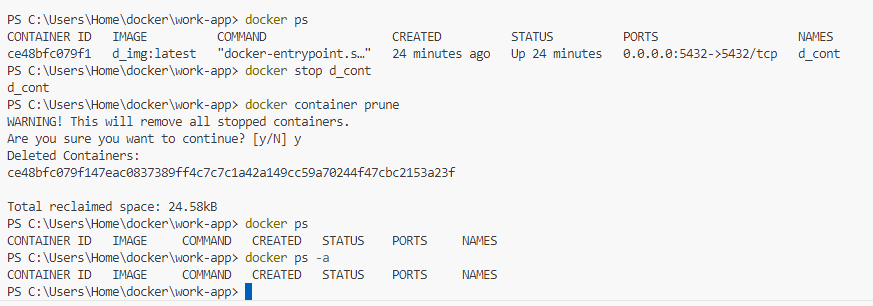


Рисунок 13 – Удаление контейнера

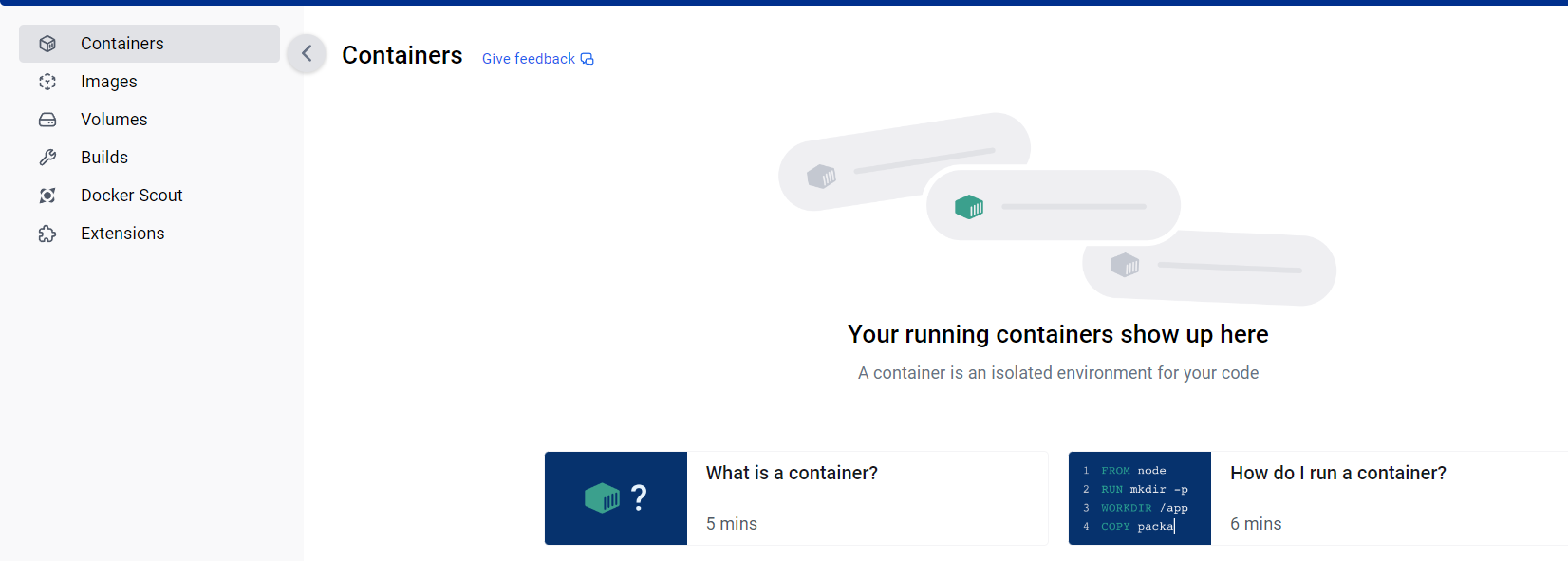


Рисунок 14 – Результат удаления контейнера

Далее создадим контейнер с помощью команды «docker run -d -p 5432:5432 --name d\_count d\_img:latest -v /d\_data:/var/lib/postgresql/data». Результат выполнения данной команды представлен на рисунках 15-16.

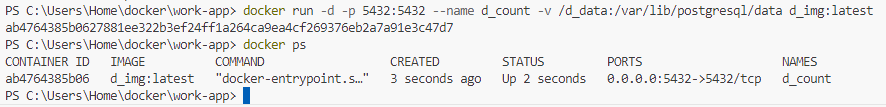


Рисунок 15 – Создание контейнера

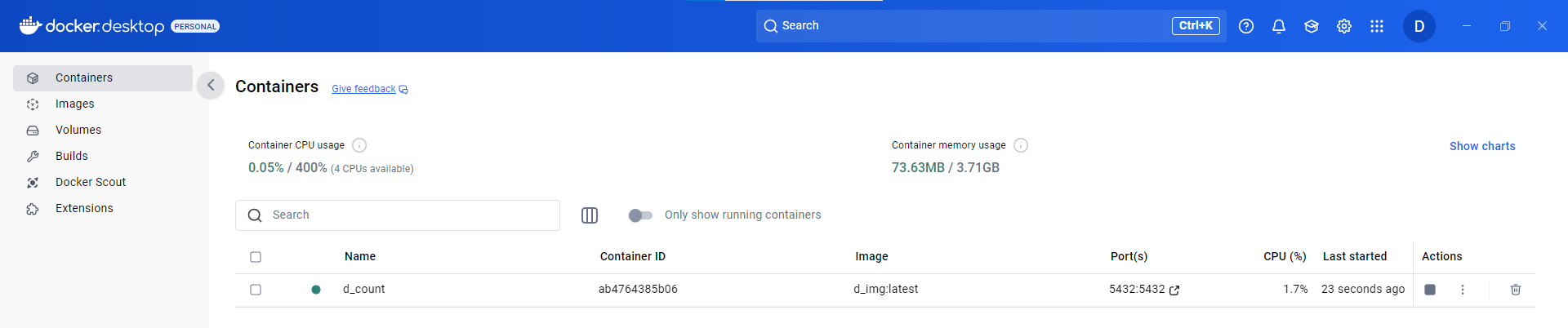


Рисунок 16 – Результат создания контейнера

Опишем использованную команду:

* docker run – команда для создания и запуска нового контейнера на основе указанного образа;
* -d (--detach) – запускает контейнер в фоновом режиме. Контейнер будет запущен, но вы вернетесь к командной строке и сможете выполнять другие команды;
* -p 5432:5432 – позволяет явно сопоставить один порт или диапазон портов в контейнере с хостом. Синтаксис: -p<порт\_хоста>:<порт\_контейнера>. Порт 5432 на вашей локальной машине (хосте) будет связан с портом 5432 внутри контейнера. Порт 5432 является стандартным портом для PostgreSQL. Он позволяет приложениям на вашей локальной машине подключаться к PostgreSQL, работающему внутри контейнера, как если бы он был установлен локально;
* --name d\_cont – присваивает контейнеру имя d\_cont. Вместо автоматического присвоения случайного имени, контейнер будет иметь заданное вами имя;
* d\_img:latest – указывает образ Docker, на основе которого будет создан и запущен контейнер. Docker ищет локально образ d\_img с тегом latest. Если образ не найден локально, Docker попытался бы загрузить его из реестра (Docker Hub);
* флаг -v – монтирует том (volume) между хостовой машиной и контейнером;
* /d\_data – путь на хост-машине, где будут сохраняться данные базы;
* /var/lib/postgresql/data – путь внутри контейнера, где PostgreSQL хранит данные;
* Между этими папками устанавливается синхронизация, то есть новые данные будут сохраняться в папку на компьютере (в нашем случае /data).

При запуске данной команды Docker запускает контейнер в фоновом режиме. Пробрасывается порт 5432 для связи между хостовой машиной и контейнером (для доступа к PostgreSQL). Задается имя контейнера «d\_cont». PostgreSQL в контейнере работает с привязанным хранилищем данных «d\_data» (обеспечивается сохранение данных после остановки/удаления контейнера). Контейнер использует созданный ранее Docker-образ «d\_img:latest». После запуска контейнера PostgreSQL готова к работе, а база данных с таблицей «index\_mass» создана и заполнена начальными данными, указанными в «init.sql».

Создадим текстовый файл cmd.txt, в котором опишем все команды, которые необходимо использовать для развертывания базы данных Postgres с помощью Dockerfile (рисунок 17).

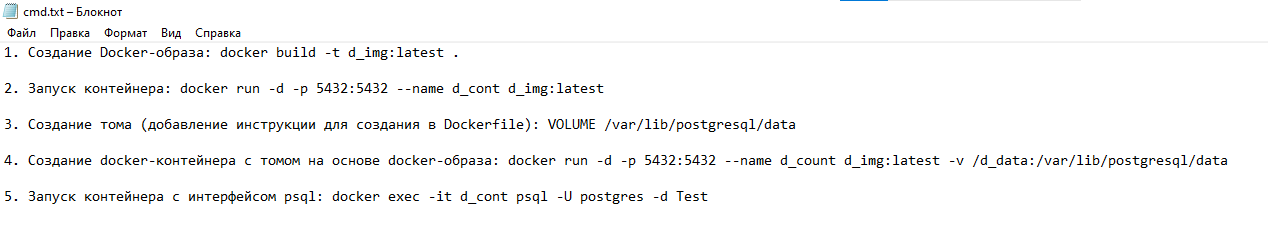


Рисунок 17 – Команды для развертывания базы данных Postgres с помощью Dockerfile

Создадим контейнер с использованием файла docker-compose.yml (рисунок 18).

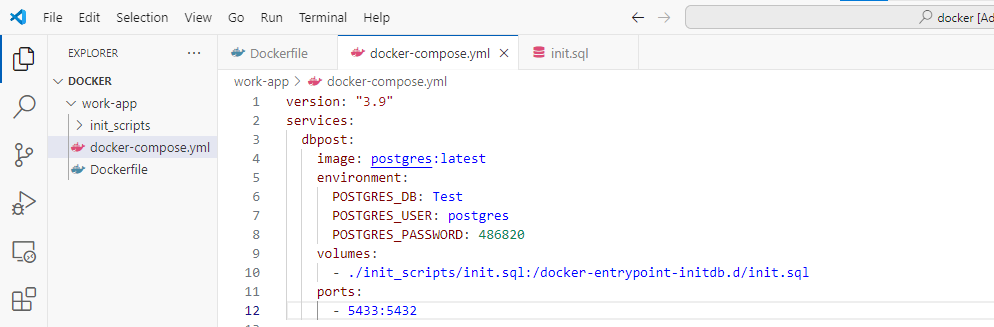


Рисунок 18 – Создание файла docker-compose.yml

Опишем docker-compose.yml:

* version: «3.9» – указывает версию файла конфигурации Docker Compose;
* services – ключевое слово, определяющее список сервисов (контейнеров), которые будут запущены Docker Compose;
* dbpost – имя сервиса;
* image: postgres: latest – указывает, какой образ Docker использовать для этого сервиса. Здесь используется официальный образ PostgreSQL с тегом latest, означающий последнюю доступную версию;
* environment – раздел, в котором определяются переменные окружения, которые будут переданы контейнеру при запуске;
* POSTGRES\_DB: Test – устанавливает переменную окружения POSTGRES\_DB со значением Test. Это означает, что при инициализации PostgreSQL будет создана база данных с именем Test;
* POSTGRES\_PASSWORD: 486820 – устанавливает переменную окружения POSTGRES\_PASSWORD со значением 486820. Это задает пароль для пользователя postgres;
* POSTGRES\_USER: postgres – устанавливает переменную окружения POSTGRES\_USER со значением postgres. Это задает имя пользователя базы данных;
* volumes – раздел для определения томов. Тома позволяют сохранять данные вне контейнера или монтировать файлы из вашей файловой системы внутрь контейнера;
* -./init\_scripts/init.sql:/docker-entrypoint-initdb.d/init.sql – определяет монтирование файла init.sql из локальной директории ./init\_scripts в контейнер по пути / docker-entrypoint-initdb.d/init.sql. Все SQL-скрипты в директории /docker-entrypoint-initdb.d/ автоматически выполняются при инициализации контейнера PostgreSQL;
* ports – раздел для настройки перенаправления портов между хост-машиной и контейнером;
* 5433:5432 – пробрасывает порт 5432 контейнера на порт 5433 хост-машины. Это позволяет получать доступ к базе данных PostgreSQL, работающей в контейнере, через порт 5433 с вашей локальной машины.

Запустим Docker Compose командой docker-compose up –d. Результат представлен на рисунках 19-20.

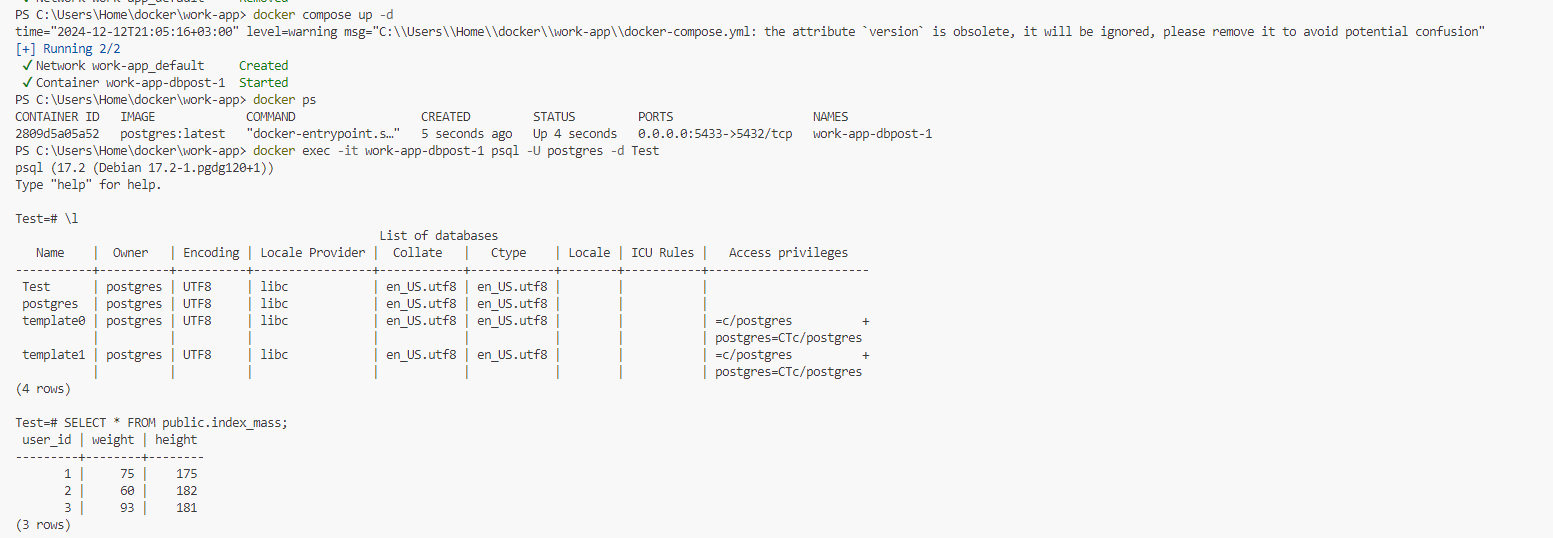


Рисунок 19 – Запуск Docker Compose

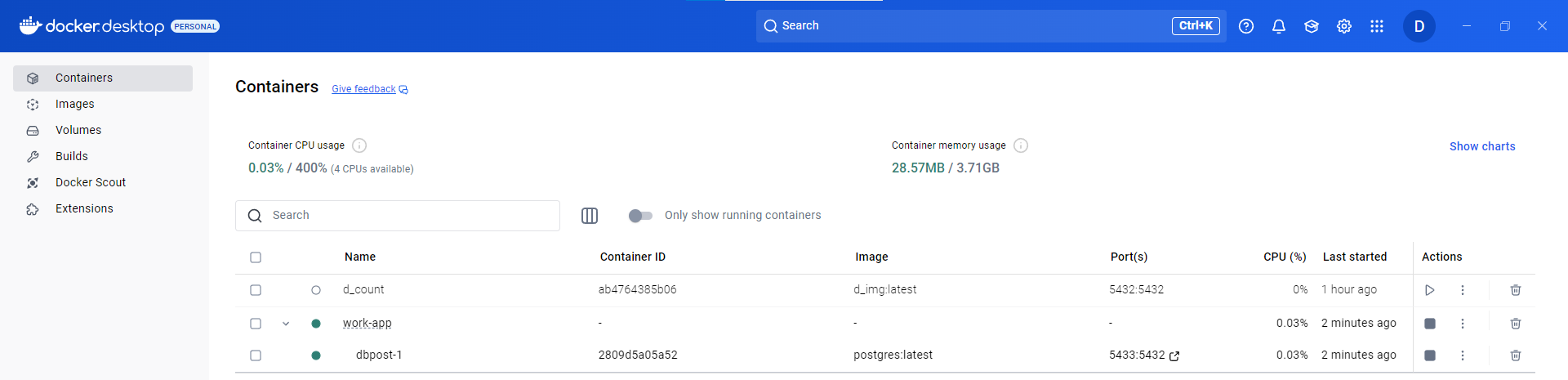


Рисунок 20 – Результат запуска Docker Compose

При запуске Docker Compose читает файл docker-compose.yml и создает сервис dbpost, используя образ postgres:latest. Контейнер запускается с переменными окружениями, указанными в секции environment. Это приводит к созданию пользователя «postgres» с паролем 486820 и базы данных «Test». Файл init.sql из локальной директории ./init\_scripts монтируется в контейнер в директорию /docker-entrypoint-initdb.d/. PostgreSQL при инициализации базы данных выполняет все SQL-скрипты из этой директории, поэтому таблица «index\_mass» будет создана и данные будут вставлены. Порт 5432 контейнера публикуется на порт 5433 хост-машины, что позволяет подключаться к базе данных с вашего компьютера, используя инструменты DBeaver или psql.

В результате запуска Docker Compose создается сеть с именем «work-app\_default». Это изолированная виртуальная сеть, внутри которой будут взаимодействовать контейнеры. Также запущен контейнер с именем «work-app-dbpost-1».

4. Выводы по результатам лабораторной работы

В ходе выполнения данной лабораторной работы были изучены основы работы программного средства для автоматизации развертывания и управления приложениями в средах с поддержкой контейнеризации Docker на примере решения задачи развертывания СУБД Postgres.

5. Ответы на контрольные вопросы

1. Docker – это проект с открытым исходным кодом для автоматизации развертывания приложений в виде переносимых автономных контейнеров, выполняемых в облаке или локальной среде. Одновременно с этим, Docker — это компания, которая разрабатывает и продвигает эту технологию в сотрудничестве с поставщиками облачных служб, а также решений Linux и Windows, включая корпорацию Майкрософт.

2. Задачи, которые решает Docker:

- развертывание среды или приложения. Docker позволяет перенести приложение со всеми зависимостями на другую систему с помощью пары команд в терминале. Настройка зависимостей вручную занимает больше времени. Также с помощью Docker можно быстро развернуть рабочую среду с определенными настройками. Существуют «системные контейнеры», которые содержат дистрибутивы ОС.

- изолированный запуск. Docker позволяет запустить приложение отдельно от всей системы без конфликтов с другими программами. Программа становится практически автономной и не вызывает ошибок зависимости.

- контроль ресурсов. Еще одна возможность Docker — распределение ресурсов между разными приложениями. Неизолированные процессы могут конкурировать за память и вычислительные мощности процессора. Изолированные друг от друга программы не делают этого. Docker позволяет эффективнее использовать ресурсы и не допускать конфликтов.

- повышение безопасности. Если код контейнерного приложения окажется небезопасным, это не навредит серверу-хосту. При правильной настройке контейнера деятельность кода не затронет основную систему. Даже фатальная ошибка не повлияет на работоспособность остальных служб и программ.

- работа с микросервисами. Микросервисная архитектура — такой тип организации ПО, при котором функции большого приложения разделяются на маленькие независимые программные модули. Они общаются друг с другом с помощью протоколов, но в целом работают автономно друг от друга. Docker подходит для реализации архитектуры этого типа: каждый микросервис упаковывается в отдельный контейнер, который можно настроить и протестировать, запустить или остановить отдельно от других.

- ускорение цикла разработки. Технологии контейнеризации помогают программировать быстрее. На настройку среды, разворачивание приложений под разными платформами тратится меньше времени. В результате повышается производительность всей команды.

- управление сложными системами. Для автоматизации большинства процессов со сложными контейнерными приложениями используются платформы оркестрации. Многие возможности специального ПО завязаны на контейнеризации и функциях Docker.

Например, платформы автоматизируют разворачивание контейнеров, их настройку и масштабирование. Это нужно, так как программная архитектура становится более сложной. Приложения могут состоять из сотен отдельных контейнеров, каждый из которых нужно развернуть и настроить. Поддержка таких приложений вручную занимает много времени.

- масштабирование. Это еще одна задача для платформ оркестрации. Во многих из них поддерживается автоматическое масштабирование систем под разные площадки и условия. Пример такой платформы — Kubernetes, которая часто используется в связке с Docker.

Автомасштабирование помогает быстро оптимизировать сектор под повышенную нагрузку. Если сайт неожиданно получит больше трафика, чем обычно, система перераспределит ресурсы и сервисы адаптируются.

3. Docker-образ — шаблон для создания Docker-контейнеров. Представляет собой исполняемый пакет, содержащий все необходимое для запуска приложения: код, среду выполнения, библиотеки, переменные окружения и файлы конфигурации.

Docker-образ состоит из слоев. Каждое изменение записывается в новый слой.

* При загрузке или скачивании Docker-образа, операции производятся только с теми слоями, которые были изменены.
* Слои исходного Docker-образа являются общими между всеми его версиями и не дублируются.

4. Docker-контейнер — это изолированная среда выполнения, в которой пользователи могут запускать приложения, изолируя их от хостовой системы. Контейнер включает всё необходимое для работы приложения: код, библиотеки, зависимости и минимальную файловую систему. Он использует ядро хост-системы, что делает его лёгким и эффективным. Обычно в одном контейнере запускают одно приложение, отдельный компонент системы или микросервис.

Контейнеры работают автономно, изолированы друг от друга и от хостовой системы, поэтому ошибка в одном контейнере обычно не влияет на работу других или на саму хостовую систему. Благодаря этому подходу приложения становятся более устойчивыми и легко масштабируются.

5. Docker Volumes предоставляет разработчикам постоянное хранилище для контейнеров. Этот инструмент отключает привязку данных к жизненному циклу контейнера, позволяя получить доступ к контейнерным данным в любой момент. Таким образом, сделанные в контейнерах записи остаются доступными после уничтожения содержавшего их контейнера и могут повторно использоваться в других. Это полезное решение для обмена данными между контейнерами Docker, которое также дает возможность подключения новых контейнеров к созданному хранилищу.

Docker Volumes устроен следующим образом: на сервере создается каталог, который затем монтируется в один или несколько контейнеров. При этом каталог является независимым, поскольку не включается в структуру слоев образа Docker. Это и позволяет обойти ограничение read only для создаваемых контейнеров с таким каталогом внутри.

6. Docker-compose — это надстройка над докером, которая позволяет запускать множество контейнеров одновременно и маршрутизировать потоки данных между ними.

Для каждого проекта (кластера контейнеров) Docker создаёт свою сеть, где контейнеры могут обращаться друг к другу по именам, которые мы укажем в docker-compose.yml. Все настройки запуска кластера контейнеров находятся в этом же файле, который располагается в корневой директории проекта.

7. Dockerfile фокусируется на создании отдельных образов Docker с подробными параметрами настройки, в то время как Docker Compose оптимизирует управление многоконтейнерными приложениями, предоставляя декларативный формат конфигурации. По сути, Dockerfile используется для создания образов Docker, тогда как Docker Compose используется для оркестрации и управления многоконтейнерными приложениями Docker.

Dockerfile используется, когда нужно определить один контейнер. Он отлично подходит для настройки среды именно так, как вам нужно для вашего приложения.

Docker Compose используется, когда вам нужно скоординировать несколько контейнеров, которые должны работать вместе. Например, приложение, которому требуется веб-сервер, база данных и кэш, выиграет от Docker Compose.

Подводя итог, можно сказать, что основное различие между Dockerfile и файлом «docker-compose.yml» заключается в их области применения и назначении. Файл Dockerfile используется для определения процесса сборки для одного образа Docker, а файл docker-compose.yml используется для определения нескольких контейнеров Docker и управления ими в качестве одного приложения.

8. Команда docker exec позволяет отправлять различные задания в запущенный Docker контейнер. С ее помощью вы можете выполнять команды или запускать новые процессы внутри уже работающего контейнера без необходимости останавливать его или создавать новый.

9. Для сохранения важных данных после аварийного отключения контейнера в Docker можно использовать инструмент Docker Volumes (тома Docker).

Он предоставляет постоянное хранилище для контейнеров, отключая привязку данных к жизненному циклу контейнера и позволяя получить доступ к контейнерным данным в любой момент. Таким образом, сделанные в контейнерах записи остаются доступными после уничтожения содержавшего их контейнера и могут повторно использоваться в других.