Лабораторная работа 1. Виртуализация и контейнеризация

Виртуализация обычно подразумевает логическое объединение вычислительных ресурсов и абстрагирование их от физического оборудования, например, сервера или компьютера. У виртуальной машины есть свой пул логических ресурсов: CPU, RAM и дисковое пространство. Так, благодаря виртуализации, на одном физическом сервере можно запустить несколько независимых виртуальных машин.

Другой пример виртуализации – одновременный запуск нескольких операционных систем (ОС) на одном компьютере (контейнерная виртуализация). Каждая система работает со своим набором ресурсов, которые предоставляются из общего пула на уровне устройства. Этим пулом управляет хостовая система – гипервизор.

В отличие от виртуализации, когда аппаратное окружение эмулируется и запускается множеством гостевых операционных систем, в случае с контейнеризацией запускается только один экземпляр ОС, имеющий общее ядро с хостовой ОС. Таким образом достигается снижение накладных расходов для запуска приложений. На рисунке 1 представлена организация окружения в контейнерах.

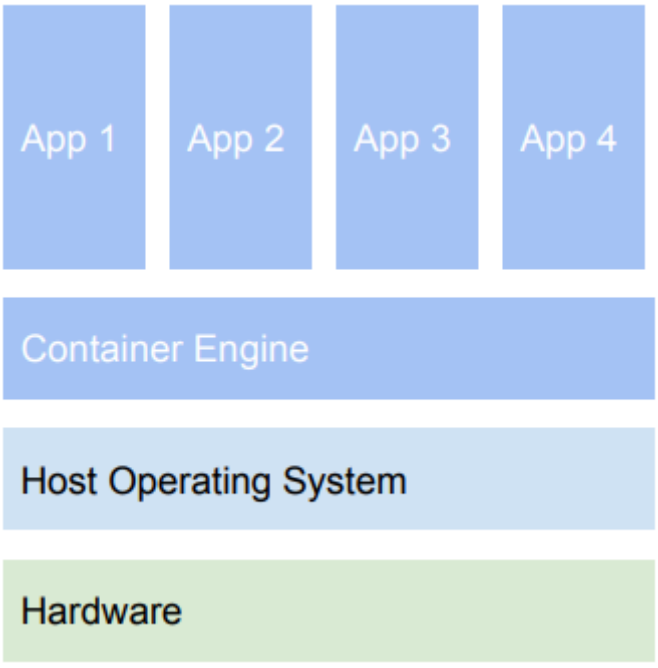


Рисунок 1 – Приложения в контейнерах

Контейнер – пакет, который содержит приложение, все зависимости и конфигурацию, необходимую для его запуска. Контейнеры легко распространять, передавать, использовать для разработки и эксплуатации ПО.

1.1 Преимущества Docker и контейнеризации

Использование контейнеров имеет ряд преимуществ по сравнению с использованием обычных средств разработки и доставки приложений:

- **сохранение окружения** при разработке, тестировании и эксплуатации;
- **небольшой размер** (контейнеры занимают мало места, что позволяет разворачивать большое количество приложений без больших затрат ресурсов);
- **стандартный формат** (контейнеры стандартизированы, что позволяет легко использовать их в различных средах);
- **изоляция** контейнеров друг от друга и от ОС (повышает уровень безопасности);
- **возможность быстрого прототипирования** (при необходимости поэкспериментировать с различным ПО можно очень быстро получить среду разработки, так как необходимые компоненты в контейнерах легко установить, и они готовы к использованию);
- **поддержка микросервисной архитектуры (**каждый отдельный микросервис может поставляться в виде отдельного контейнера);
- **непрерывная доставка** (возможность быстрого развертывания и сохранение среды внутри контейнера позволяет в автоматическом режиме выпускать новые версии ПО).

1.2 Docker: основные концепции

Самые важные концепции Docker – образ, контейнер и слой.

Образ – файл с всей конфигурацией запуска программы в Docker-контейнере.

Контейнер – экземпляр образа. Контейнер действует как виртуальная машина, но не имеет отдельной ОС. Каждый образ состоит из нескольких слоев, или наборов различий. Основные концепции представлены на рисунках 2 и 3.

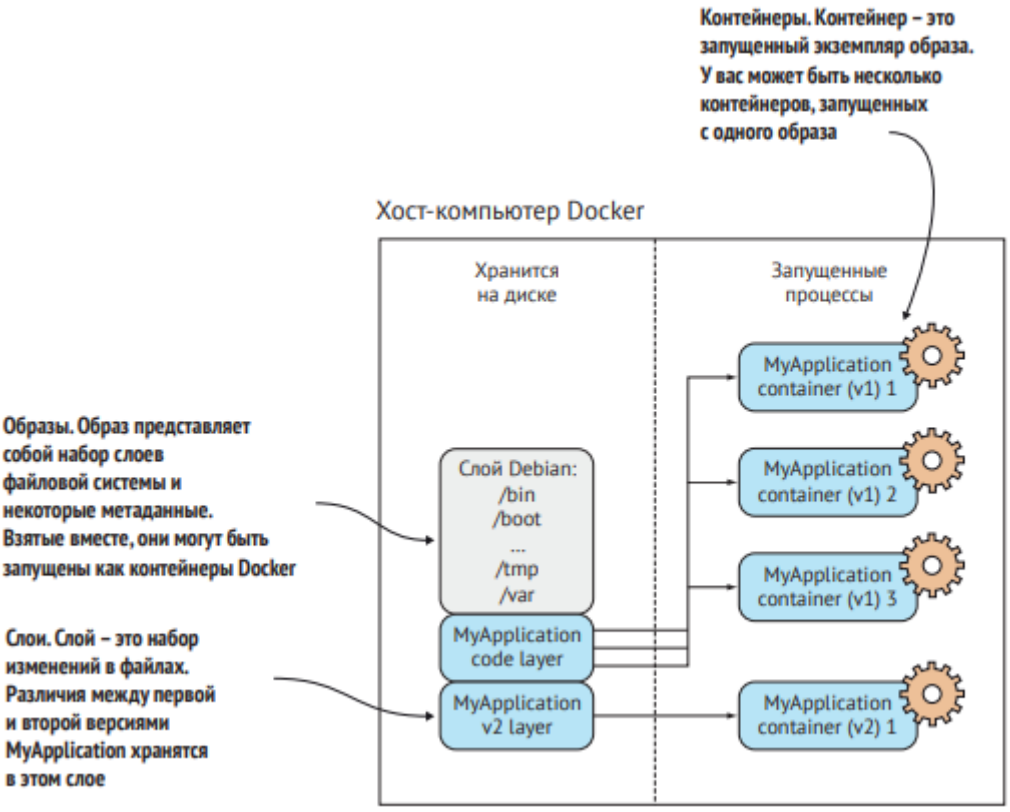


Рисунок 2 – Базовые концепции Docker

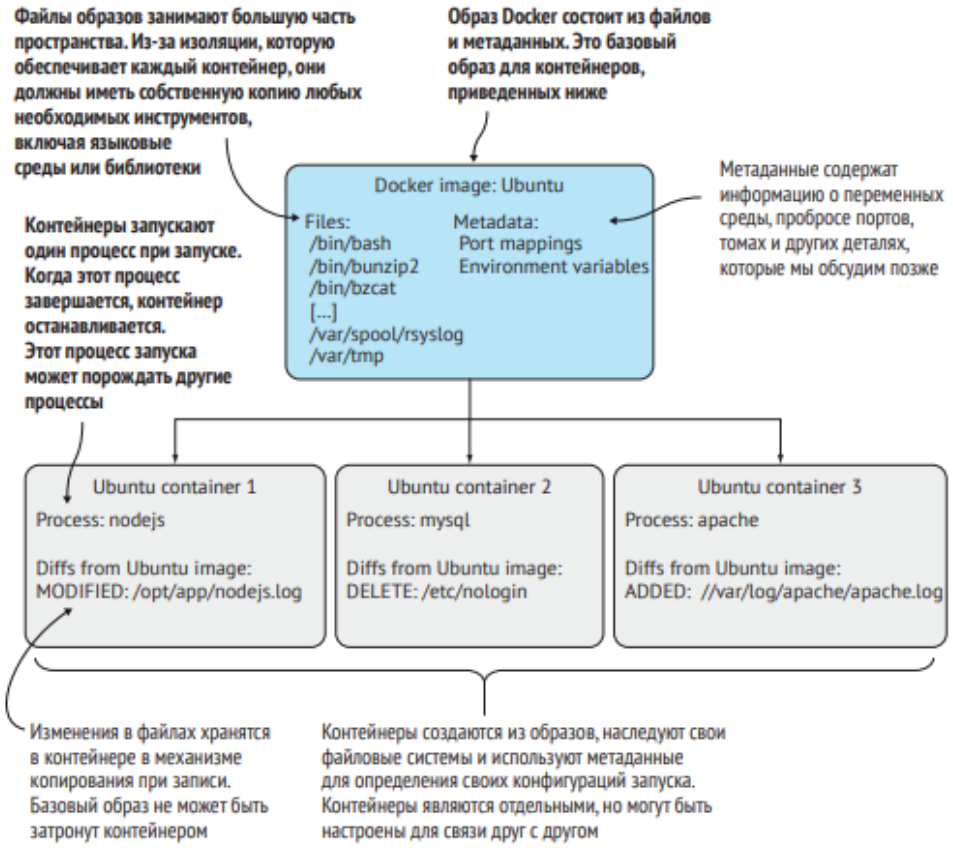


Рисунок 3 – Образы и контейнеры

Основные команды Docker:

- *pull* – загрузка образа;
- *rmi* – удаление образа;
- *run* – создание и запуск контейнера на основе образа;
- *ps* – просмотр списка контейнеров;
- *stop* – остановка запущенного контейнера;
- *start* – запуск существующего контейнера;
- *exec* – запуск команды в работающем контейнере;
- *rm* – удаление контейнера;
- *logs* – просмотр логов контейнера;
- *build* – создание нового образа на основе Dockerfile.

Другие команды, а также формат их использования и подробное описание можно получить, вызвав команду docker help.

1.3 Создание приложения Docker

Попробуем создать простое приложение и запустить его с помощью Docker. Для примера будем использовать запуск простого nodejs-сервера. Для этого понадобится создать скрипт с сервером и файл Dockerfile для описания процесса сборки приложения.

Для начала создадим новый проект, вызвав команду npm init –y, создадим папку src, а в ней файл *app.js* и поместим в него следующий код:

```
const http = require('http');

const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Если все сделано верно, должна получиться структура файлов, представленная на рисунке 4.

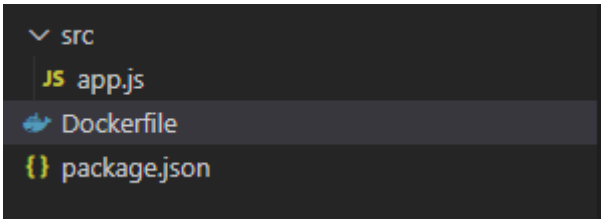


Рисунок 4 – Образы и контейнеры

Для запуска приложения нужно вызвать команду `node src/app`.

Если все сделано правильно, в консоль будет выведено следующее:

```
Server running at http://localhost:3000/
```

Теперь упакуем это приложение в контейнер, чтобы запустить через Docker. Для этого нужно поместить следующей код в файл Dockerfile (в нём будет храниться инструкция по сборке и запуску приложения):

```
# FROM - образ, на основе которого создается приложение.
FROM node:alpine
# WORKDIR - рабочая директория приложения
WORKDIR /app
# копирование package.json файла в рабочую директорию
COPY package.json ./
# запуск процесса установки зависимостей
RUN npm i
# копирование файлов с исходным кодом в папку src
COPY src ./src

# Указание порта, который будет слушать сервис
EXPOSE 3000
# Команда запуска сервиса
CMD ["node", "src/app.js"]
```

После описания файла Dockerfile выполним следующую команду в терминале в директории проекта:

```
docker build --tag test:latest .
```

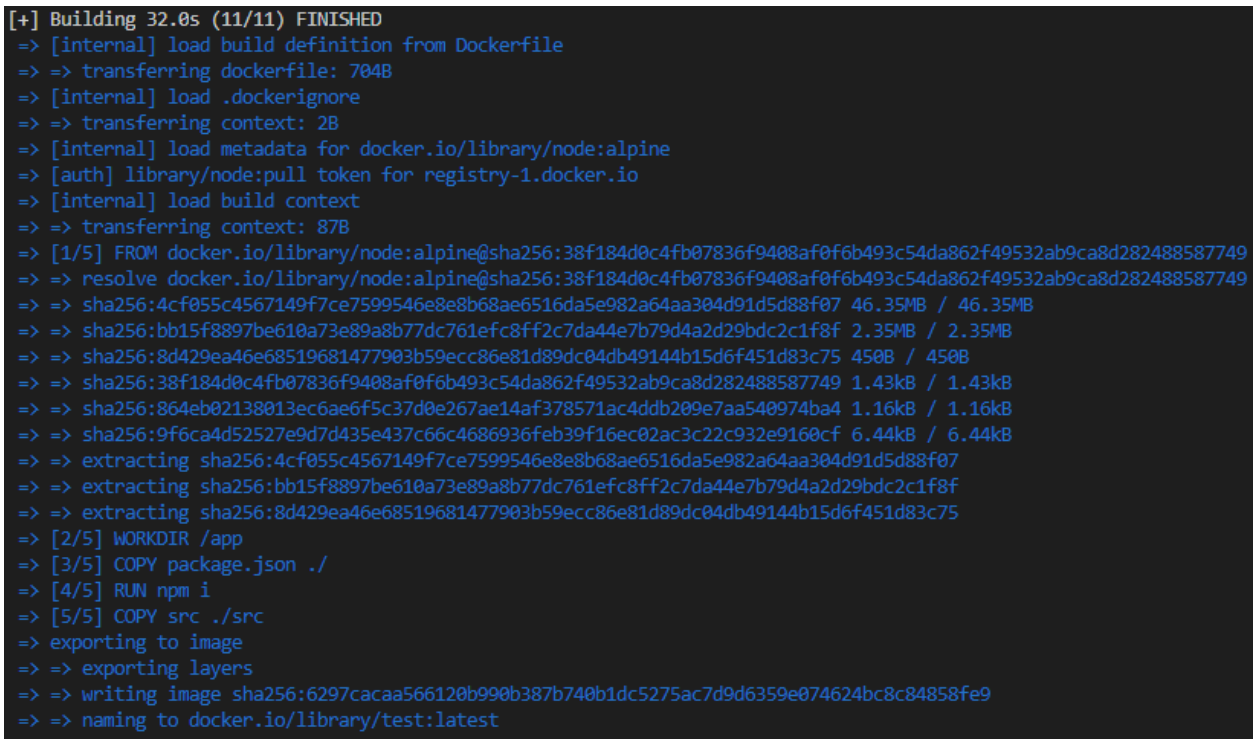
Важно: перед тем, как выполнять любые Docker-команды, необходимо его установить. Инструкции по установке на вашу платформу можно найти здесь: https://docs.docker.com/engine/install/

Разберем, что делает каждая из частей этой команды:

- docker – вызов консольной утилиты docker;
- build – команда для сборки образа;
- tag test:latest – пометка собранного образа (в дальнейшем это позволит работать с образом по его тегу, вместо использования генерируемого SHA256 хэша).

Точка в конце вызова – указание на расположение Dockerfile.

При успешном выполнении команды в консоли будет подобный вывод:



Для просмотра списка доступных образов, можно выполнить команду `docker images`. В списке должен появиться образ с REPOSITORY: test, TAG: latest и некоторым Image ID. На этом сборка образа завершена и можно переходить к запуску сервиса. Для этого выполним команду:

```
docker run --name test-container -p 3000:3000 -d test
```

Снова разберем команду:

- docker – как уже известно, обращение к утилите docker;
- run – команда запуска контейнера;
- флаг – –name test-container – название контейнера, как и в случае с сборкой образа, позволит обращаться к контейнеру по его имени. В данном случае было указано имя test-container;
- флаг –p 3000:3000 – указание какой внешний порт в системе будет соответствовать порту в контейнере. Первый аргумент – в системе, второй – в контейнере. Если указать, например, 3001:3000, то порт 3001 на вашей машине пойдет в порт 3000 внутри контейнера;

- флаг –d – запуск в фоновом режиме. В таком случае Вы получите только хэш контейнера после запуска, и не увидите вывода из контейнера в консоль. Если хотите видеть, что выводит Ваша программа, можете не использовать этот флаг;
- test – название образа для запуска. Его указали ранее, когда выполняли docker build.

Если все указано верно, то если открыть в браузере http://localhost:3000, Вы увидите сообщение «Hello world».

Для просмотра запущенных контейнеров нужно вызвать команду `docker ps`

Если нужно увидеть все контейнеры, нужно добавить флаг –a: `docker ps -a`

Для остановки приложения, нужно вызвать команду `docker stop <container name>`

Для удаления контейнера нужно использовать команду `docker rm <container name>`

Для удаления образа нужно использовать команду ниже, при этом tag указывать не обязательно, если был указан latest: `docker rmi <image name:tag>`

Если при работе с образами и контейнерами имена не были указаны, то вместо имен нужно использовать image ID и container ID.

Задания:

1) собрать тестовый вариант приложения;
2) выполнить сборку образа без имени и тега, с указанием любого имени и тега latest, с указанием любого другого тега;
3) удалить собранные образы;
4) запустить контейнер из собранного образа без указания имени, после остановить и удалить;
5) запустить контейнер с указанием имени, после остановить и удалить;

6) запустить контейнер с указанием порта в системе, отличающегося от порта внутри контейнера.

Дополнительные задания:

1) загрузить командой `docker pull` образ `busybox`;
2) запустить контейнер в фоновом режиме с командой ping ya.ru и 10 пингами, назвать контейнер pinger;
3) вывести в консоль список запущенных и остановленных контейнеров;
4) вывести в консоль логи контейнера `pinger`;
5) удалить контейнер `pinger`;
6) удалить образ `busybox`.

1.4 Обновление приложений в контейнерах

При разработке ПО часто нужно выполнять запуск новых версий. Так как приложения поставляются в контейнерах, необходимо выполнять сборку и запуск новых версий контейнеров. Рассмотрим, как это можно сделать.

Для начала внесем изменения в код внутри приложения. Например, пусть теперь сервер при выдаче фразы Hello world, выведет еще и время его запуска. Обновленный код будет выглядеть следующим образом:

```
const http = require('http');

const port = 3000;
const createdAt = new Date();

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Hello World, started at ${createdAt.toISOString()}`);
});


server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Теперь, нужно заново собрать контейнер и запустить его. Можно просто повторно собрать образ, вызвав команду `docker build`, однако это создаст новый образ, а это приведет к расходу места на диске. Поэтому вместе с созданием нового образа нужно будет удалить существующий контейнер и образ, а после собрать и запустить новый. Чтобы сделать это, используйте следующие команды:

- `docker stop;`
- `docker rm;`
- `docker rmi.`

Стоит учесть, что команда `docker rmi` снимает тег с образа, и образ удаляется, когда снимается последний тег. Таким образом, можно оставлять образы старых версий, добавляя к ним сначала тег нужной версии, а затем снимая тег `latest`.

В итоге, если все сделано правильно, то запустится новая версия сервера, который в ответе будет отправлять дату запуска.

Познакомимся с еще одной важной частью работы с контейнерами – переменными средами. Переменные среды позволяют выполнять настройку сервиса. Например, с помощью переменных среды можно:

- указать порт работы сервиса;
- указывать токены для работы с внешними системами;
- указывать директории для работы с файлами (осторожно, внутри контейнера своя файловая система, о работе с хостовой системой будет рассказано в следующем подразделе);
- любые прочие настройки, которые зависят от среды исполнения.

Для примера добавим в наш сервис использование переменной среды, которая будет отвечать за порт, на котором работает сервис. Обычно при определенной конфигурации контейнеров нет необходимости переопределять порт. Однако, часто такая необходимость может присутствовать. Например, запуск сервиса вне контейнера.

Обновим код приложения, добавив получение переменной PORT:

```
const http = require('http');

const port = process.env.PORT;
if (!port) {
  throw new Error('PORT variable not set!');
}
const createdAt = new Date();

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Hello World, started at ${createdAt.toISOString()}`);
});


server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Обновим приложение, удалив старый контейнер и образ и заменив их новыми. Но при выполнении команды `docker run` теперь нужно добавить флаг `-e PORT=1234`.

Команда запуска будет выглядеть следующим образом:

```
docker run -dp 3000:1234 -e PORT=1234 --name test-container test
```

Для просмотра логов и проверки, корректно ли запустился сервер, можно использовать команду `docker logs test-container`. Если все выполнено правильно, будет выведено:

```
Server running at http://localhost:1234/
```

При этом снаружи сервис должен остаться доступен через порт 3000, так как в параметре –p было указано, что 3000 порт с хост-машины адресуется в порт 1234 контейнера.

Задания:

- написать скрипт для обновления контейнера;
- выполнить обновление контейнера;
- выполнить обновление контейнера, сохранив старую версию образа;
- запустить два контейнера: один с новой версией, другой со старой.

Дополнительное задание:

- передать в контейнер с сервером несколько переменных среды, в ответе сервера вывести значения этих переменных.

1.5 Сохранение данных, docker volume

Попробуем еще расширить функционал сервера, добавив к нему лог запросов. Доработаем код сервера, добавив в него логирование запросов:

```
const http = require('http');
const path = require('path');
const fs = require('fs');
const logsDir = 'logs';
const logsPath = path.resolve('./logs');
if (!fs.existsSync(logsPath)) {
  fs.mkdirSync(logsDir, {recursive: true});
}

const file = 'access-log.log';
const logFilePath = path.resolve(logsPath, file);


const port = process.env.PORT || 3000;
if (!port) {
  throw new Error('PORT variable not set!');
}
const createdAt = new Date();

const server = http.createServer((req, res) => {
  fs.appendFileSync(logFilePath, `${new Date().toISOString()}:request\n`);
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Hello World, started at ${createdAt.toISOString()}`);
});


server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Теперь при запросе к серверу в директории logs в файл access-log.log будут дописываться строки запроса к серверу. Сделайте несколько запросов к серверу, чтобы логи появились. Однако, чтобы убедиться, что все работает, придется научиться присоединяться к контейнеру.

Чтобы просмотреть содержимое файла внутри контейнера, придется заглянуть внутрь его файловой системы. Самый простой способ – вызвать оболочку командной строки. Для этого выполните следующую команду:

```
docker exec -it test-container sh
```

После выполнения команды откроется командная строка, но внутри контейнера. Дальше потребуется выполнение нескольких bash-команд:

- `ls` – просмотр содержимого папки;
- `cat` – просмотр содержимого файла;
- `cd` – переход в директорию.

После корректного выполнения команды `docker exec` в терминале отобразится `/app #`

Можно выполнить команду `ls`, в числе отображенных папок должна появиться папка `logs`.

Перейдем в эту папку, выполнив `cd logs`. Если Вы уже сделали несколько запросов к серверу, то вызов команды `ls` должен отобразить лог-файл. Просмотреть его содержимое можно, вызвав команду `cat <filename>`. Если все сделано верно, в консоли будет выведено несколько строк со временем запроса и строкой `request`.

Теперь внесите любое изменение в исходный код, пересоберите и перезапустите контейнер, после чего попробуйте найти лог-файл, не выполняя запросов к серверу. Если все сделано верно, лог файл **не появится**.

Это происходит из-за того, что содержимое контейнера не обязано сохраняться между сборками. В случае, если сервис генерирует какие-то данные, которые нужно хранить, то к контейнеру необходимо подключить хранилище.

Сделать это можно двумя способами, но итоговый процесс будет идентичен:

1) создать новый `volume`, вызвав команду `docker volume create <name>`. После, вызвав команду `docker volume list`, можно увидеть список созданных томов, при желании можно выполнить удаление через `docker volume rm <name>`;
2) ничего не создавать, и использовать хранилище системы.

Пойдем простым путем и используем хранилище системы. При запуске сервиса командой `docker run` необходимо будет указать путь в файловой системе, который будет присоединен к контейнеру. В таком случае всё, что контейнер будет писать в примонтированную директорию, попадет в файловую систему, а при следующем запуске содержимое директории не будет очищено, так как ее содержимое не является частью контейнера. Команда запуска будет выглядеть следующим образом:

```
docker run -dp 3000:1234 -e PORT=1234 -v <path>:/app/logs --name test-container test
```

Вместо `<path>` необходимо подставить путь к директории, в которую будет записываться содержимое `logs` директории внутри контейнера. При указании пути нужно использовать абсолютный путь.

Откройте указанную директорию и сделайте несколько запросов к серверу. Если все сделано верно, появится лог-файл. После этого внесите изменения в исходный код сервера и пересоберите контейнер. Содержимое папки не должно очиститься.

`Docker volumes` обычно используются, когда у сервиса есть какие-либо данные, которые нужно сохранять независимо от версии сервиса. Например, это могут быть базы данных для СУБД. При использовании `volume` можно иметь доступ к данным, а также обновлять контейнер без потери данных.

Задания:

- добавить в запуск сервиса переменную **INSTANCE_ID.** Запустить одновременно два контейнера с сервером, в один передать **INSTANCE_ID=1**, в другой **INSTANCE_ID=2**. Модифицировать вывод лог-файла, в имени файла теперь должен присутствовать **INSTANCE_ID** сервиса. Используя `volumes`, подключить оба сервиса к папке `logs` как было описано в задании. На хост-машине создать файл с произвольным именем в папке `logs`. Сделать несколько запросов к каждому сервису. Используя команду `docker exec`,

подключиться к первому контейнеру и отобразить содержимое папки `logs`, затем выйти, подключиться ко второму и отобразить содержимое папки `logs`;

- с помощью `docker volume create` создать `volume` для сервисов. Подключить каждый `volume` к отдельному сервису для записи логов. Запустить контейнер `busybox` в интерактивном режиме и подключить к нему созданный `volume`. Просмотреть логи, которые записывает сервис;
- Реализовать сервис, отдающий через эндпоинт последние несколько строк логов других сервисов, которые пишут логи каждый в свой `volume`.

1.6 Работа с docker-compose

При использовании приложений, состоящих из нескольких контейнеров, а также для упрощения процесса описания конфигурации, можно использовать инструмент `docker-compose`. Это средство позволяет по описанной заранее конфигурации собрать и запустить один или множество контейнеров.

Для описания конфигурации используется язык YAML. Полное описание можно найти на официальном сайте https://docs.docker.com/compose/

Перепишем конфигурацию запуска сервиса на `docker-compose`, и попробуем запустить обновленный сервис. Рядом с файлом Dockerfile необходимо создать файл docker-compose.yaml со следующим содержимым:

```
version: '2.1'

services:
  test-container:
    container_name: test-container
    build:
      context: .
    environment:
      - PORT=1234
    ports:
      - "3000:1234"
    volumes:
      -<path>:/app/logs
```

Важно: `<path>` в секции `volumes` нужно заменить на свой путь, как было указано в предыдущем разделе.

Теперь вся конфигурация контейнера описана в одном файле. Для запуска сервиса нужно запустить команду `docker-compose up`

Для удаления контейнера и созданной для него сети нужно запустить команду `docker-compose down`

Стоит учитывать, что собранный образ не будет автоматически удален для возможности запуска в следующий раз без сборки. Если нужно будет удалить образ, используйте команду `docker images` и найдите созданный образ.

Файлы `docker-compose` можно использовать не только для того, чтобы не писать длинные команды по запуску одного контейнера, но в них можно описывать и запуск всей среды разработки целиком. Например, так можно поднять свой сервис, базу данных и `pgadmin`:

```
version: '2.1'

services:
  test-container:
    container_name: test-container
    build:
      context: .
    environment:
      - PORT=1234
    ports:
      - "3000:1234"
    volumes:
      - D:/code/playground/dockerize/volume:/app/logs
  db:
    container_name: postgres
    image: postgres:alpine
    restart: always
    environment:
      - POSTGRES_PASSWORD: password
    ports:
      - 5432:5432
    volumes:
      - <path>:/var/lib/postgresql/data
  pgadmin:
    container_name: pgadmin
    image: dpage/pgadmin4
    restart: always
    depends_on:
      - db
    ports:
      - 80:80
    environment:
      - PGADMIN_DEFAULT_EMAIL=admin@example.com
      - PGADMIN_DEFAULT_PASSWORD=password
      - PGADMIN_CONFIG_STORAGE_DIR='/tmp-mount'
```

При работе с `postgres` можно через подключение `volume` указать, где на хост-машине хранить базы данных. Однако при использовании `ntfs` могут возникать трудности, связанные с необходимостью обеспечить права доступа контейнера к файловой системе. В таком случае рекомендуется сначала создать `volume` через `docker volume create`, а затем указать его в `docker-compose.yaml` вместо `<path>`.

Задания:

- изучить указание переменных среды по умолчанию и файлов переменных среды из документации `docker-compose` Environment variables in Compose | Docker Documentation;
- запустить `postgres` и `pgadmin` через `docker-compose` с использованием переменных среды и значений по умолчанию. Подключить `pgadmin` к запущенному `postgres`;
- подключить приложение к `postgres` и сделать несколько тестовых запросов;
- поднять контейнер `mongo` и `mongo-express`, подключиться через веб-интерфейс к `mongo`;
- создать отдельный `volume` для `postgres` и создать контейнер `postgres`, подключив `data` к нему. Создать и настроить контейнер `portainer`. Используя `portainer`, создать контейнер `busybox`, подключить его к `volume postgres` контейнера. Подключиться к контейнеру `busybox` и продемонстрировать содержимое `data postgres` контейнера.

1.7 Разворачивание контейнеров при разработке

В настоящем подразделе будут приведены теоретические сведения по автоматизации развертывания контейнеров. Данная часть работы выполняется самостоятельно. Для этого необходимо ознакомиться с написанием `powershell`- или `bat`-файлов для Windows, либо `shell`-скриптов для Linux.

Рассмотрим два схожих сценария, разницей будет источник данных и место сборки контейнера.

Вариант 1: разворачивание через github

Распространенный вариант разворачивания сервисов, который часто используют при разработке для непрерывного выпуска новых версий. Обычно

для достижения автоматического развертывания используют веб-хуки от репозитория к машине, которая выполняет сборку, и, возможно, запускает сервис.

Веб хук – это запрос от репозитория к нашему серверу, когда внутри репозитория происходят изменения. Например, появляется новый коммит в develop-ветке.

Для реализации развертывания через git в простом случае понадобится выполнение трех команд:

- `git pull` для загрузки свежей версии репозитория;
- `docker-compose down` для остановки и удаления уже запущенного контейнера;
- `docker-compose up` для сборки и запуска новой версии.

Вариант 2: разворачивание через docker hub

Данный способ можно использовать, когда уже есть готовый собранный образ, и нужно только развернуть его на целевой машине. Для этого необходимо собрать приложение и опубликовать его в docker hub. Для обновления приложения в таком случае потребуется:

- описать файл `docker-compose.yaml`, где необходимо указать все сведения о конфигурации запускаемого контейнера;
- выполнить команду `docker-compose up` для запуска. При необходимости обновить контейнер при помощи `docker-compose pull`.

Если не использовать файл `docker-compose`, можно обойтись командой `docker run`, в которой необходимо перечислить все нужные для запуска параметры.

Задания:

- Выгрузить проект в git-репозиторий, написать скрипт для обновления контейнера из git-репозитория;

- собрать контейнер локально и загрузить его в docker hub. Обновить приложение, используя образ, загруженный из docker hub.