Лабораторная работа 10
Docker

1.1 Введение в Docker

Docker is an open platform for developing, shipping and running applications.

(c) Документация

Docker – это платформа, которая помогает разработчику тестировать, доставлять, разрабатывать, запускать и эксплуатировать приложение. Иными словами, платформа, созданная для разработки, доставки и развертывания приложений с использованием технологий контейнеризации.

Контейнер – в отличие от виртуальной машины, которая виртуализирует аппаратное обеспечение, использует виртуализацию на уровне операционной системы, реализуемую за счет использования специфичных функций ядра Linux, с помощью чего достигается приемлемая изоляция, », изолирование окружения. Все контейнеры на хосте используют одно, общее с другими контейнерами ядро системы. Контейнер, который запускается внутри операционной системы хоста, — это стандартный программный модуль, который развертывается из стандартизованного образа (по сути по аналогии с процессом и программой, контейнер -запущенный экземпляр образа), в который упаковывается код и все его зависимости так, чтобы приложение могло быть быстро запущено и надежно работать в любой среде, где установлена среда выполнения контейнеров. К среде контейнеризации предъявляются следующие требования:

- Следует принципу единственной ответственности
- Self-contained
- Стандартизированная сборка и запуск контейнеров
- Поддержка легковесных “контейнерных” образов
- Эфемерность.

Движок Docker – программное обеспечение с открытым кодом, установленное на хосте и обеспечивающее создание и запуск контейнеров. Движок Docker действует как часть платформы Docker, клиент-серверного приложения, обеспечивающего запуск контейнеров на различных ОС, в Windows и в дистрибутивах Linux.

Образы Docker – легковесный, независимый от внешних источников, запускаемый в среде контейнеризации пакет, содержащий приложение, вместе со всем необходимым для его запуска и работы (набором из инструментов, библиотек, зависимостей), которое должно запускаться как процесс в контейнере. Образы также содержат набор метаданных для запуска контейнера, который может быть запущен на платформе Docker. Образы состоят из слоев, которые являются неизменяемыми, поэтому, если Вы хотите внести изменения, необходимо будет создать новый образ.

Плюсы использования Docker:

1. Легкое и быстрое развертывание
2. Экономичность потребления ресурсов (особенно в сравнении с ВМ)
3. Переносимость между разными системами
4. Легкость масштабирования
5. Хорошо подходит для микросервисной архитектуры
6. Доставка ПО в стандартизированной «коробке»

7. Изолированность от других процессов, что означает большую безопасность при одновременной работе множества процессов на хосте

Представим ситуацию: Вы работаете над большим проектом. У Вас есть несколько программ с кодом, написанным на Python. В первой программе у Вас стоит одна версия NumPy (библиотека языка программирования Python), во второй программе другая версия этой же библиотеки, а на компьютере установлена вообще последняя версия NumPy 2.1.1. В какой-то момент Вы можете обнаружить, что первая программа перестала работать, так как использует какую-то другую версию библиотеки. Docker заворачивает нашу программу в некий контейнер. Таким образом, программа становится изолированной, помогая тем самым избежать проблемы рассогласования версий.

1.2 Сущности Docker

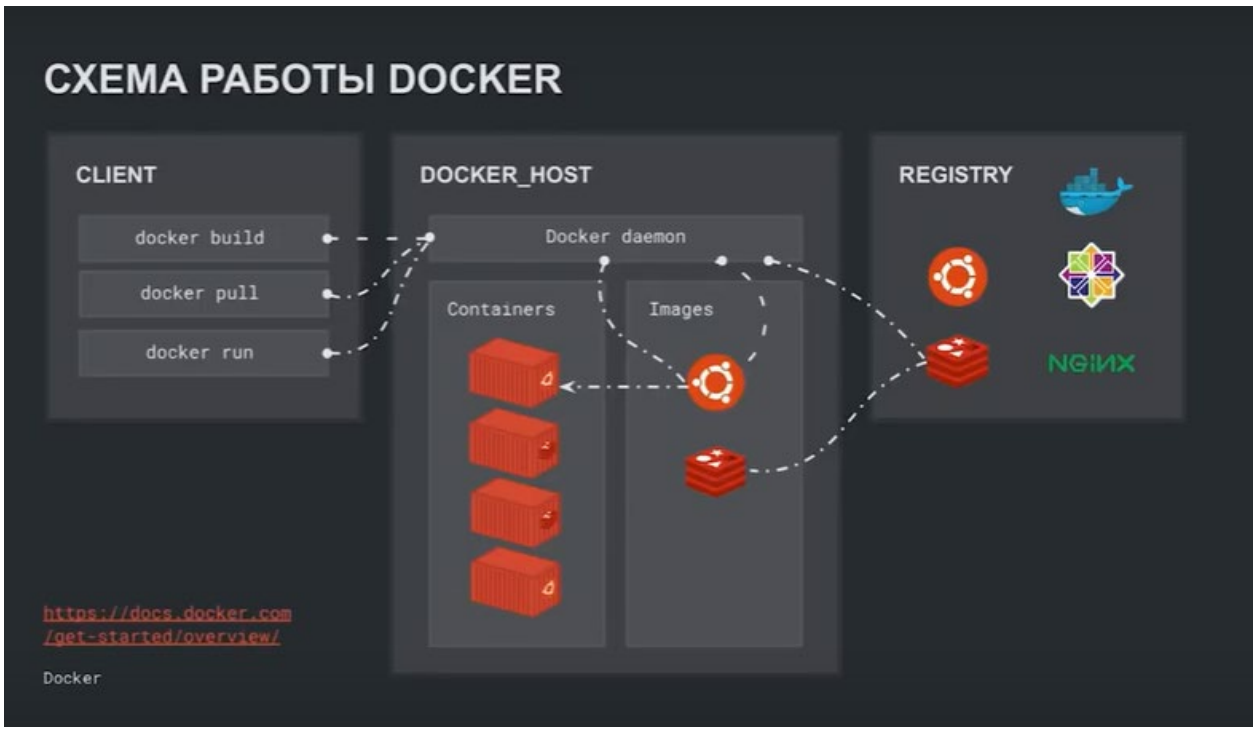
- Демон (docker daemon)
- Образ (docker image)
- Репозиторий / Реестр (docker registry)
- Контейнер (docker container)

На рисунке 1 приведена схема работы Docker. Основным элементом является **Docker host** –компьютер или виртуальная машина на котором установлен сам Docker.

Client (Docker CLI) – это пользовательский интерфейс, с помощью которого возможно писать команды в интерфейсе командной строки и взаимодействовать с API Docker.

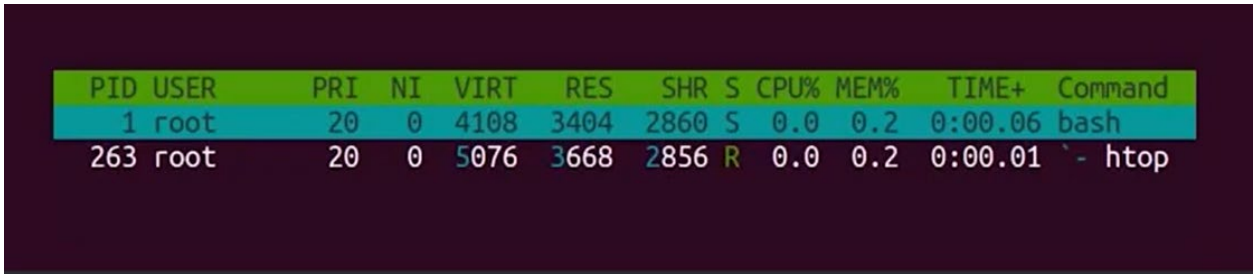
Docker daemon – это служба, которая управляет всеми контейнерами Docker, скачивает образы из репозитория, запускает контейнеры, обеспечивает для контейнеров сетевое подключение и постоянное хранилище (тома), собирает с них логи. Она постоянно работает в фоновом режиме на хосте.

Image – неизменяемый файл, представляющий в большинстве случаев слепок файловой системы (содержит в себе файлы, директории, программы, с которыми взаимодействует основное приложение). Характерная особенность Image заключается в том, что они строятся слоями.



Рассмотрим пример запущенного контейнера, содержащего в себе образ интерпретатора командной строки.

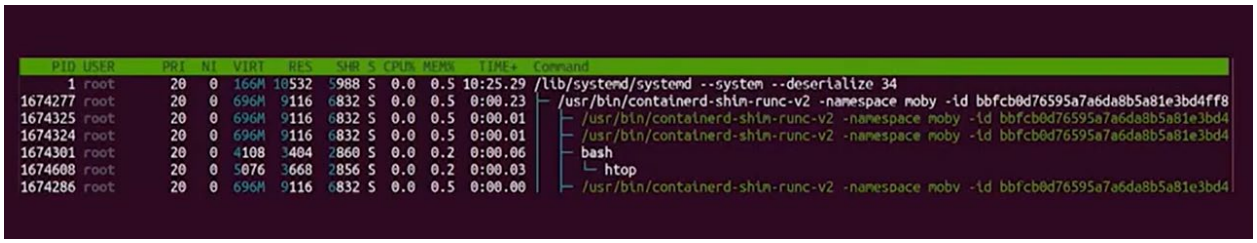
Воспользуемся утилитой htop, чтобы посмотреть процессы, запущенные в контейнере. Как видно из последнего столбца Command, в контейнере запущен bash, вызвавший саму программу htop.



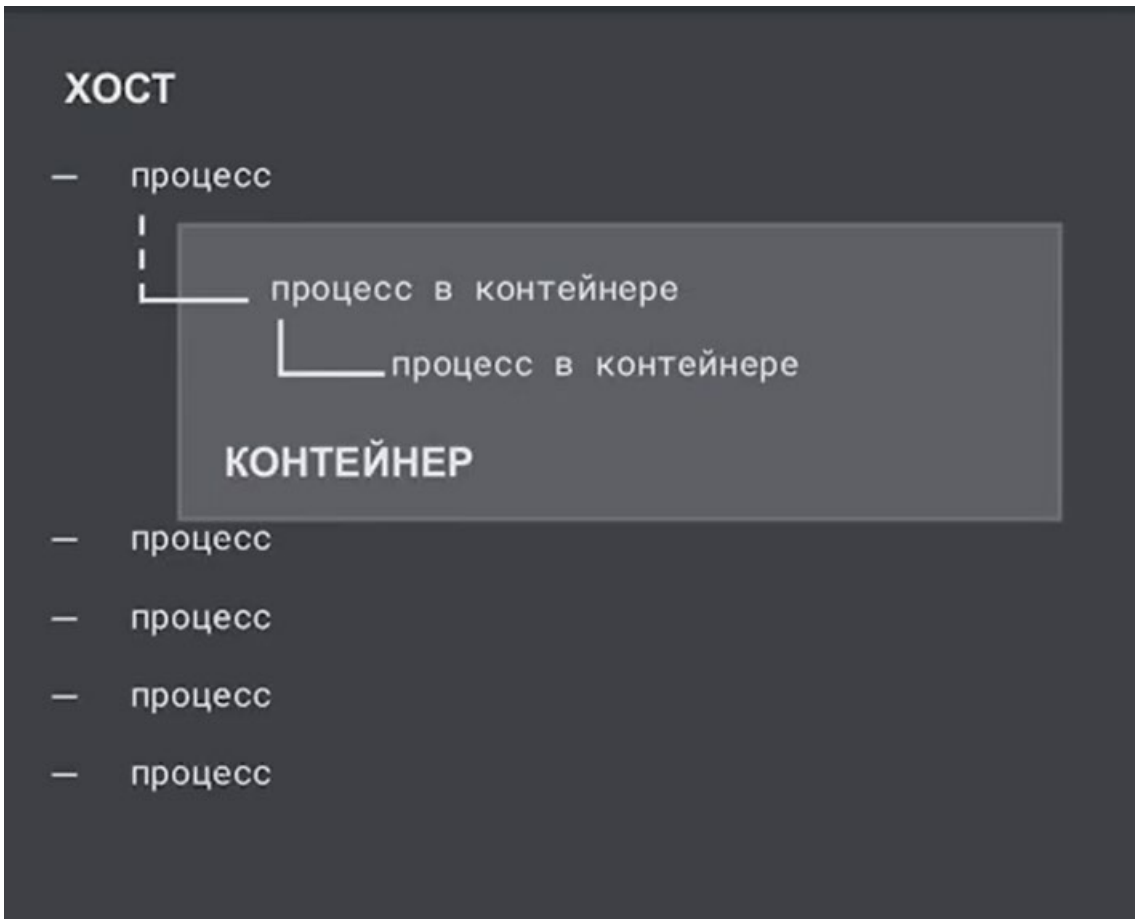
Другая обстановка будет на хосте. Здесь можно увидеть не только процессы на нашем компьютере, но и то, что находится внутри контейнера. Как раз тот самый bash, который вызвал htop.

В контейнере *Docker* процессы обычно запускаются под пользователем, указанным в *Dockerfile* с помощью директивы *USER* или по умолчанию под пользователем root, если это не указано. Если Вы запускаете утилиту *htop* внутри контейнера, то в списке процессов Вы увидите процессы, запущенные под этим пользователем. Если это *root*, то все процессы будут отображаться с *UID* 0, иначе процессы будут запускаться под созданным пользователем с его *UID*.

При этом, если контейнер запускается с параметром ***--user***, Вы можете явно указать, под каким пользователем и с каким UID запускать процессы внутри контейнера.



Таким образом система видит то, что происходит в контейнерах, но контейнеры не видят то, что происходят снаружи. Это хорошо иллюстрирует рисунок.



Registry (репозиторий или реестр) — это сервер, хранилище готовых образов, которые можно использовать. Стоит отметить, что есть публичные репозитории и частные. Так большинство компаний предоставляют собственные репозитории для образов, такие есть у Yandex, Google, Amazon и т.д. Это важно, поскольку процесс доставки и поднятия контейнеров, зачастую связан именно со скачиванием образа. Самый известный публичный репозиторий — DockerHub. Здесь вы можете найти практически все базовые образы(Ubuntu, Alpine, Python, Postgres), а также зарегистрироваться и выложить свой.

Соберём всю картину целиком:

1) После запуска компьютера включается Docker daemon
2) Далее через команду говорим Docker daemon, чтобы он поднял контейнер на основе образа
3) Если образ есть на хосте – поднимается контейнер, если локально образа нет, то daemon обращается в репозиторий и узнает есть ли у него указанный в спецификации образ
4) Если образ есть в репозитории, то он скачивается на наш компьютер локально и происходит запуск контейнера

1.3 Образы и контейнеры

Рассмотрим основные способы получения образа контейнера:

1) Загрузка из репозитория (Docker Hub) — docker pull <образ>

2) Создание образа на основе уже запущенного ранее контейнера — docker commit <контейнер>

3) Создание своего образа на основе Dockerfile, существующего образа или с нуля, указав в качестве основы пустой образ «scratch»

Tag — версия образа (на рисунке приведён пример с разными версиями образа node). Если не указывать его явно в команде docker pull , то по умолчанию будет скачиваться образ , имеющий tag версии - latest.

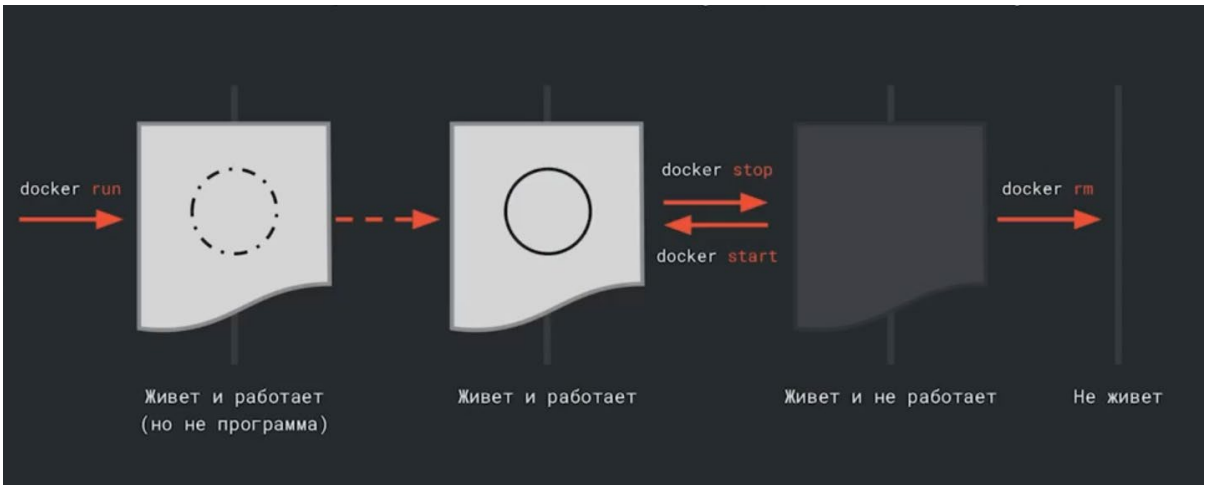
Image ID — id с помощью, которого можем обратиться к образу

Created — дата создания образа

Size — размер образа. Обратите внимание, что при загрузки новой версии образа старая автоматически не удаляется, поэтому необходимо делать это вручную.

```
onik110@onik110-pc:~$ sudo docker pull node:17-alpine3.14
[sudo] password for onik110:
17-alpine3.14: Pulling from library/node
8663204ce13b: Pull complete
3380d21d2e9d: Pull complete
81f27490e553: Pull complete
b82819fd644b: Pull complete
Digest: sha256:0d8276c8e82fa717a9a88b8734bbad60ac29a0f15f9d04acbe8dd16a850f783c
Status: Downloaded newer image for node:17-alpine3.14
docker.io/library/node:17-alpine3.14
onik110@onik110-pc:~$ sudo docker images
REPOSITORY               TAG             IMAGE ID       CREATED       SIZE
kcoursedocker/task-1.1   latest          a7c163a4b2cd   2 years ago   72.8MB
node                     17-alpine3.14   b20b24e39dda   2 years ago   168MB
onik110@onik110-pc:~$
```

Жизненный цикл контейнеров



Встречается ситуация, когда контейнер уже запущен, но программа в нем ещё не готова к работе. Такое может случиться, например, с базой данных. Вы подняли контейнер с базой и сразу же делаете какой-нибудь запрос, но при этом сама база внутри контейнера ещё не развернулась и поэтому ваш запрос может выдать ошибку.

Варианты решения проблемы являются:

1. **Поэтапный запуск –** один из способов избежать проблем с зависимостями. Например, если есть приложение, зависящее от базы данных, то возможно сначала запустить контейнер с ней, а затем контейнер с приложением.

2. **Проверка готовности контейнера (Health Checks):** Docker предоставляет возможность использовать проверки состояния (health checks) для контейнеров. Вы можете определить, как Docker должен проверять, готов ли контейнер к работе. Например, для базы данных это может быть SQL-запрос, который проверяет, доступна ли база данных. Если контейнер не готов, Docker не будет считать его "живым", и вы сможете настроить другие контейнеры на ожидание.

3. **Docker Compose**: Docker Compose — это инструмент для определения и запуска многоконтейнерных приложений на базе Docker, имеет встроенные механизмы для управления зависимостями между сервисами.

4. **Пробные запросы (Retries)**: В вашем приложении можно реализовать логику повторных попыток (retries) при выполнении запросов к зависимым сервисам. Например, если приложение не может подключиться к базе данных, вы можете настроить его на повторные попытки подключения через определенные интервалы времени, пока база данных не станет доступной.

5. **Скрипты и ожидание**: Вы можете использовать скрипты, которые будут выполнять ожидание (wait-for-it или аналогичные) перед запуском вашего приложения. Эти скрипты могут проверять доступность сервиса перед тем, как запустить основное приложение.

1.4 Некоторые полезные команды для работы с Docker

Работа с образами:

Посмотреть список образов

```
sudo docker images
```

Удалить образы

```
sudo docker rmi <образ> [образ…]
sudo docker image rm <образ> [образ…]
```

Удаление неиспользуемых образов

```
sudo docker image prune
```

Удаление всех неиспользуемых образов (все неиспользуемые образы, включая те, которые имеют теги)

```
sudo docker image prune -a
```

Удаление неиспользуемых слоев

```
sudo docker system prune
```

Работа с контейнерами:

Создание контейнера с именем

```
sudo docker run --name <имя> <образ>
```

Удаление контейнера после завершения его работы

```
sudo docker run --rm <образ>
```

Вывести список всех контейнеров

```
sudo docker ps -a
sudo docker ps --all
```

Вывести только ID контейнеров

```
sudo docker ps -q
sudo docker ps --quiet
```

Отслеживание запущенных контейнеров

```
sudo docker ps
```

Запуск и подключение к контейнеру в интерактивном режиме

```
sudo docker run -it <образ>
```

Удаление контейнера

```
sudo docker rm <NAMES or CONTAINER ID>
```

Подключение к работающему контейнеру

```
sudo docker exec -it <NAMES or CONTAINER ID> bash
```

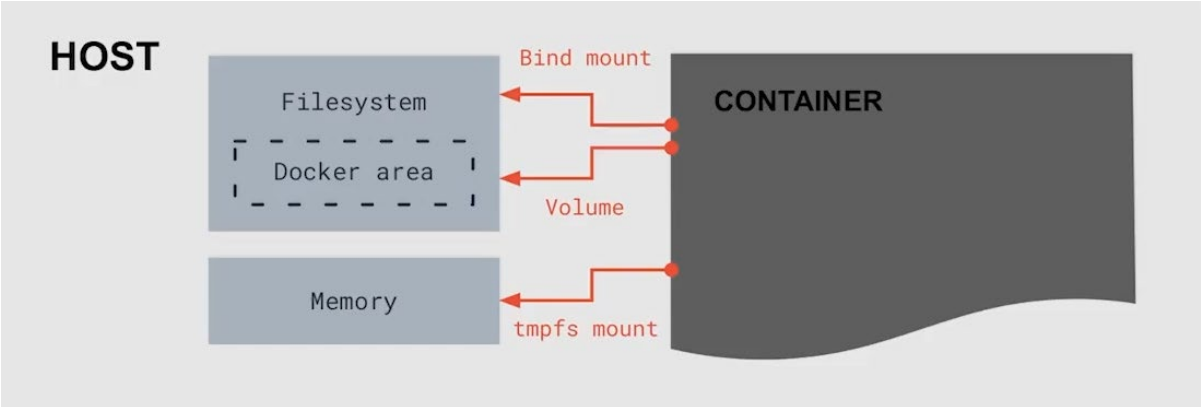
Запуск интерактивной оболочки в контейнере, не имеющем внутри себя Bash, например Alpine.

```
sudo docker exec -it <NAMES or CONTAINER ID> sh
```

1.5 Volume, bind mount

Бывает нужно сделать так, чтобы файл оказался в контейнере не на этапе сборки образа, а в момент запуска контейнера. Кроме того, нам, также, нужно сохранять данные из контейнера, к примеру, если в контейнере запускается база данных. Если мы запустим ее в контейнере как есть, зальем туда данные, а потом по каким-то причинам контейнер умрет или будет перезапущен, мы все эти данные потеряем, поскольку контейнеры эфемерны. Чтобы этого не произошло, требуется как-то связать хост и контейнер, чтобы иметь возможность сохранить данные из контейнера на хосте.

Получается необходим механизм, который позволил построить «мостик», соединяющий контейнер и хост, общее хранилище, доступное и из контейнера, и с хоста. У докера условно, есть две его реализации. Это bind mount и volume.

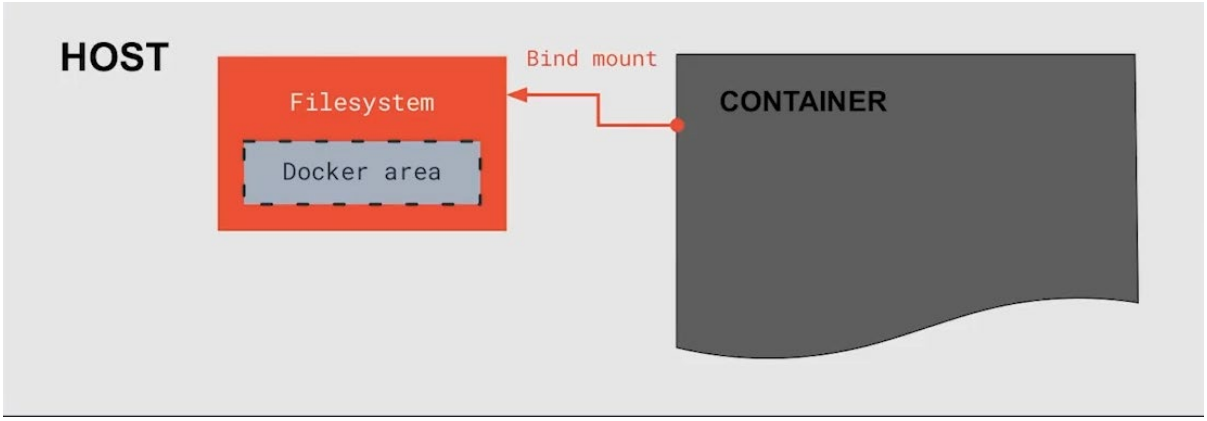


По умолчанию контейнеры изолированы от хоста, в том числе в контексте файловой системы. При помощи и bind mount(монтирование директории с хоста в контейнер), и volume (создание отдельной директории внутри /var/lib/docker/volumes/ с поддержкой дополнительного функционала) мы можем связать файловую систему контейнера с файловой системой нашего компьютера.

<u>В случае с bind mount:</u>

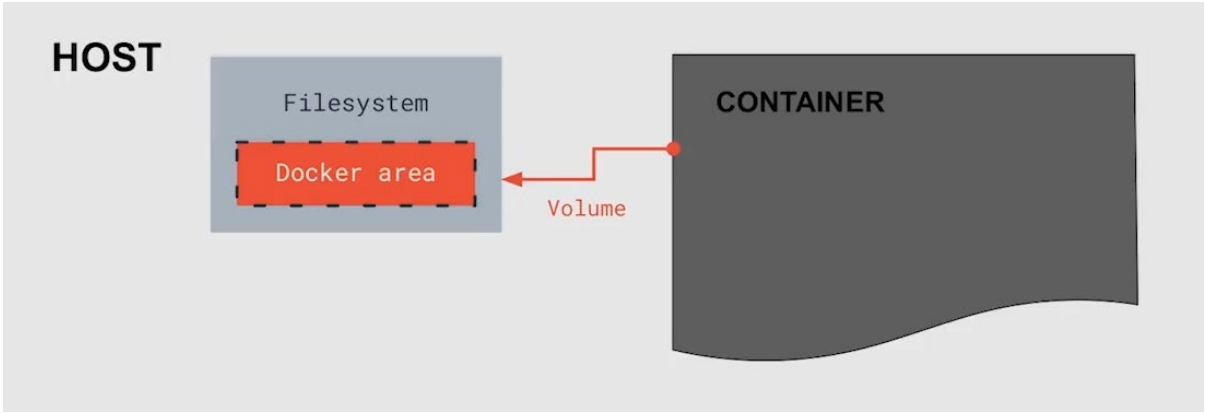
- Монтирование файлов / папок, которые находятся в любом месте на хосте

- Используется, чтобы прокинуть файлы/папки в контейнер
- Нужно использовать полный путь



Соответственно, Volumes также позволяют примонтировать папку к контейнеру, однако они отличаются тем, что на диске, то есть на хосте, они лежат в специальном месте и контролируются докером (напрямую с хоста «залезать» в директорию с данными тома строго не рекомендуется разработчиками):

- Монтирование папок, находящихся в специально отведенном месте (Например, в системе Linux в директории /var/lib/docker/volumes/)



- Используется, чтобы хранить данные в контейнере

Таким образом, несмотря на то, что и bind mount, и Volume предназначены примерно для одного и того же, у них есть ряд различий.

- Volumes, в отличие от bind mount, не зависят от хоста (например, Mac или Windows). Как мы сказали, это некая папка, которой управляет сам Docker. В случае с bind mount нам же нужно учитывать абсолютные пути в различных системах.
- Volumes, в отличие от bind mount, более безопасны. Это в частности связано с тем, что через bind mount мы можем монтировать абсолютно любой файл или папку хоста.
- Volumes находятся в специально отведенном месте. То есть в случае volumes мы просто задаем для контейнера имя нужного volumes. Если же мы используем bind

mount, то нам самим нужно помнить, где находятся папки. Особенно это становится неудобно, когда у нас много контейнеров и у каждого своя папка.

- Volumes позволяют связывать только папки. Volumes это именно привязывание папок, а через bind mount можно прокидывать и папки, и файлы.

Отсюда вытекает следующий момент. Если это все предназначено для сохранения некого state приложения, сохранения данных, связи нескольких контейнеров через эти данные, возникает вопрос, а что будет, если контейнер упадет? Как в таком случае будут вести себя эти сущности?

Если мы используем bind mount, тут в целом все понятно. То есть у нас на хосте есть какая-нибудь папка Data. Мы связали ее с контейнером. Внутри контейнера наша программа либо что-то взяла из этой папки, либо положила в нее данные. Так или иначе, содержимое папки Data у нас на компьютере и в контейнере синхронизировано. Если же контейнер упадет, то все то, что было у нас, так и останется. Данная папка не удалится.

С Volumes все примерно так же. Было сказано, что образы физически хранятся на нашем компьютере и занимают место, аналогично устроены Volumes. То есть это реальные папки в нашей системе, куда записываются данные.

С одной стороны, это здорово, потому что мы данные не теряем. Более того, мы можем их переиспользовать. Но, с другой стороны, у нас может нагенерироваться сотни volumes, каждый из которых также будет занимать место на диске. Поэтому volumes, которые больше не нужны, имеет смысл удалять.

Особенно это критично в следующей ситуации. Дело в том, что volumes могут создаваться как бы по умолчанию. Таким volumes дается некий id, поэтому они часто называются неименованные. Или еще можно встретить названия анонимные. То есть до этого мы явно создавали собственный volume и монтировали его к контейнеру. А может быть так, что при поднятии контейнера volume создается автоматически.

1.6 Dockerfile

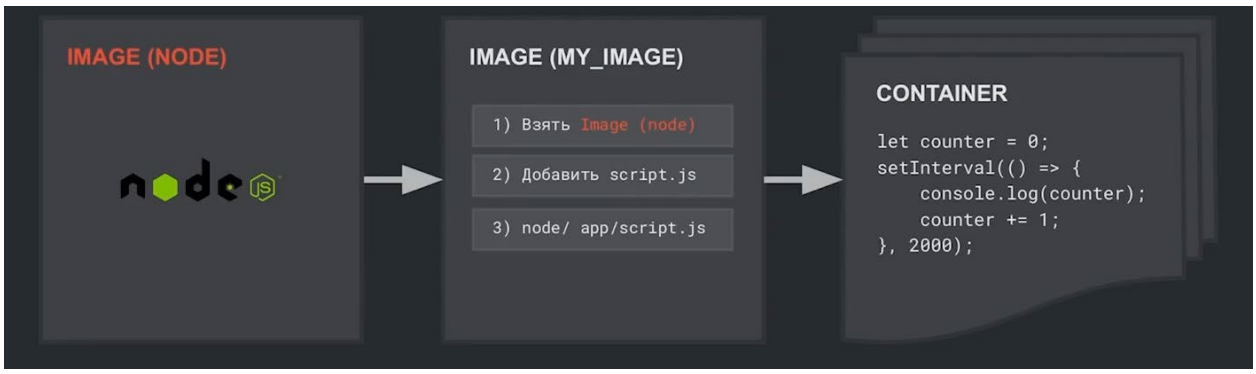
Как получить образ:

- Команда docker pull <образ> - скачать из репозитория (Docker Hub)

- На основе готового образа создать свой через ***Dockerfile*** — представляет из себя файл с инструкциями, как правильно собирать контейнеры

- Команда docker commit <контейнер> - создать образ на основе контейнера

Смысл заключается в следующем: мы берем за основу готовый образ (node) и описываем, что нужно добавить к node



Вы можете на основе node сделать абсолютно разные образы и соответственно разные контейнеры. Например, у нас имеется папка в которой находятся 2 файла: docker-файл и script.js. В docker-файле записаны 3 строки, в script.js - некоторый код.

```
onik110@onik110-pc:~/docker$ tree
.
├── Dockerfile
└── script.js

0 directories, 2 files
onik110@onik110-pc:~/docker$ cat Dockerfile
FROM node:17

COPY ./script.js /app/script.js

CMD ["node", "/app/script.js"]
onik110@onik110-pc:~/docker$ cat script.js
let counter = 0

setInterval(() => {
        console.log(counter);
        counter += 1;
}, 2000);
onik110@onik110-pc:~/docker$
```

Задача сделать так, чтобы при поднятии контейнера у нас запускался этот скрипт, а не просто node. Необходимо сообщить Docker на основе какого образа мы будем строить свой — в нашем примере взята node 17 версия. После этого нам нужно поместить наш скрипт в контейнер, для этого мы берем и копируем его. Обращаю ваше внимание, что мы физически копируем файл с нашей файловой системы в файловую систему контейнера. Далее запускаем программу при поднятии контейнера (последняя строчка в Dockerfile).

Dockerfile:

```
FROM node:17

COPY ./script.js /app/script.js

CMD ["node", "/app/script.js"]
```

Так Dockerfile является обычным текстовым файлом, где можно при помощи специальных инструкций описать каким должен быть наш образ. То есть мы расширяем возможности уже готовых образов.

Несколько BEST PRACTICES:

- Используйте .dockerignore файл
- Устанавливайте только то, что действительно нужно
- Старайтесь уменьшить количество слоев

Если Вы знакомы с git, то dockerignore, то же самое что и gitignore. Так можно указать название файла в dockerignore и daemon файл перестанет существовать.

В качестве примера давайте попробуем сделать свою кастомную Ubuntu. Начнем мы с двух слоев, дальше постепенно будем добавлять новые и на каждом этапе будем собирать образы и смотреть, что получается. Для начала нам нужно создать docker-файл:

```
onik110@onik110-pc:~/docker$ cat Dockerfile
FROM ubuntu:20.04

WORKDIR /home/onik110

onik110@onik110-pc:~/docker$
```

Итак, на первом этапе нам нужно указать базовый образ нашей сборки. Поскольку мы хотим кастомизировать Ubuntu, логично взять базовый образ Ubuntu. Соответственно, инструкция для того, чтобы задать базовый образ, называется ***FROM***. Укажем явно тег, что это Ubuntu 20.04. В целом это уже валидный docker-файл, то есть если мы сейчас попробуем запустить сборку, мы получим новый образ. То есть точно так же мы могли поднимать контейнеры на основе образа Ubuntu и ничего бы не поменялось. Поэтому давайте наш первый такой вот образ будет содержать хотя бы две инструкции. В качестве второй инструкции давайте попробуем задать, например, рабочую директорию. Делается это при помощи инструкции ***Workdir***. Это будет директория /home/onik110. Итак, вот у нас уже некоторый маленький образ. И что у нас еще раз тут происходит? Мы берем Ubuntu и говорим, что папка, в которой будет происходить дальнейшая работа, будет, соответственно, иметь путь /home/onik110. Теперь пробуем собрать образ:

```
onik110@onik110-pc:~/docker$ docker build -t custom_ubuntu:1 .
[+] Building 1.4s (6/6) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 79B
 => [internal] load metadata for docker.io/library/ubuntu:20.04
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/2] FROM docker.io/library/ubuntu:20.04@sha256:6d8d9799fe6ab3221965efac00b4c34a2bcc102c086a58dff9e19a08b913c7ef
 => CACHED [2/2] WORKDIR /home/onik110
 => exporting to image
 => => exporting layers
 => => writing image sha256:c0924e7280017b77d54609e118509f9a2c59d9809bd001d07c4fbbc11d8c59b3
 => => naming to docker.io/library/custom_ubuntu:1
onik110@onik110-pc:~/docker$
```

Теперь входим в образ в интерактивном режиме. И внимание, видите, мы оказались в директории /home/onik110. То есть если сейчас вывести команду pwd, то по умолчанию будет директория /home/onik110. В целом мы уже сделали свой собственный образ, просто переназначив рабочую папку, то есть в обычной ubuntu это корень, а у нас это, соответственно, две папки.

```
onik110@onik110-pc:~/docker$ docker run --rm -it custom_ubuntu:1
root@84dd46efe5c8:/home/onik110# pwd
/home/onik110
root@84dd46efe5c8:/home/onik110#
```

Давайте, например, скопируем какой-нибудь файл. И в этом случае мы могли бы использовать две инструкции. Первая — это ***ADD***, вторая — это ***COPY***. В примере мы будем пользоваться ***COPY***. ***COPY*** инструкция позволяющая явно скопировать файл откуда куда-то. Первым аргументом будет название файла - путь до этого файла на нашем компьютере, а вторым путь в контейнере. У нас уже три инструкции. Повторяем сборку. Собрался второй образ, смотрите, второй слой уже взялся из кэша, оставшегося после сборки первого образа. Посмотрим, что у нас произошло в контейнере. Видно, что здесь появился скрипт script.sh. Давайте попробуем вывести его содержимое. То есть в целом у нас уже в контейнере есть какая-то программа, которую мы можем запустить и работать как-то с ней:

```
onik110@onik110-pc:~/docker$ docker build -t custom_ubuntu:2 .
[+] Building 1.8s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 108B
 => [internal] load metadata for docker.io/library/ubuntu:20.04
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/3] FROM docker.io/library/ubuntu:20.04@sha256:6d8d9799fe6ab3221965efac00b4c34a2bcc102c086a58dff9e19a08b913c7ef
 => [internal] load build context
 => => transferring context: 73B
 => CACHED [2/3] WORKDIR /home/onik110
 => [3/3] COPY ./script.sh ./script.sh
 => exporting to image
 => => exporting layers
 => => writing image sha256:06a72e20c5270ebbdf58e3c38ff975c5a157b1a2b36f1cf7e10d5094e806bd46
 => => naming to docker.io/library/custom_ubuntu:2
onik110@onik110-pc:~/docker$ docker run --rm -it custom_ubuntu:2
root@72a2134087cf:/home/onik110# ls
script.sh
root@72a2134087cf:/home/onik110# cat script.sh
for i in 1 2 3 4 5
do
        exho $i
done
```

Давайте установим теперь некоторые программы. Итак, как мы это будем делать? В docker-файле есть специальная инструкция, которая называется RUN. Это инструкция позволяет запустить, произвольный скрипт. Установим vim, git, curl. Инструкция RUN запустит эти команды. Более того, RUN можно использовать, для запуска некой команду. Теперь давайте посмотрим, что у нас получилось:

```
onik110@onik110-pc:~/docker$ cat Dockerfile
FROM ubuntu:20.04

WORKDIR /home/onik110

COPY ./script.sh ./script.sh

RUN apt-get update && apt install -y vim curl git
onik110@onik110-pc:~/docker$ docker build -t custom_ubuntu:3 .
[+] Building 47.6s (9/9) FINISHED                                                                          docker:default
 => [internal] load build definition from Dockerfile                                                                 0.0s
 => => transferring dockerfile: 159B                                                                                 0.0s
 => [internal] load metadata for docker.io/library/ubuntu:20.04                                                      0.6s
 => [internal] load .dockerignore                                                                                    0.0s
 => => transferring context: 2B                                                                                      0.0s
 => [internal] load build context                                                                                    0.0s
 => => transferring context: 30B                                                                                     0.0s
 => [1/4] FROM docker.io/library/ubuntu:20.04@sha256:6d8d9799fe6ab3221965efac00b4c34a2bcc102c086a58dff9e19a08b913c7ef    0.0s
 => CACHED [2/4] WORKDIR /home/onik110                                                                               0.0s
 => CACHED [3/4] COPY ./script.sh ./script.sh                                                                        0.0s
 => [4/4] RUN apt-get update && apt install -y vim curl git                                                         45.2s
 => exporting to image                                                                                               1.7s
 => => exporting layers                                                                                              1.7s
 => => writing image sha256:9652a0201816018860f7e560f9194969195fab5eb9fa763a7a9642e00b395f1e                         0.0s
 => => naming to docker.io/library/custom_ubuntu:3                                                                   0.0s
```

Однако мы на самом деле допустили «небрежность» при написании docker-файла. Инструкция, которая отвечает за установку некоторых зависимостей ***RUN apt-get update && apt install -y vim curl git***, не изменяется. То есть, когда вы, будете работать с вашим образом Ubuntu, вы, скорее всего, будете писать базовый слой, потом обновлять

```
onik110@onik110-pc:~/docker$ docker run --rm -it custom_ubuntu:3
root@59fd7f732d70:/home/onik110# vim script.sh

[1]+  Stopped                 vim script.sh
root@59fd7f732d70:/home/onik110# git init
Initialized empty Git repository in /home/onik110/.git/
root@59fd7f732d70:/home/onik110# git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .script.sh.swp
        script.sh

nothing added to commit but untracked files present (use "git add" to track)
root@59fd7f732d70:/home/onik110#
```

инструкции, копировать какие-то файлы, но при этом инструкция ***RUN apt-get update && apt install -y vim curl git*** у вас будет везде повторяться. Поэтому, по-хорошему, нам бы нужно было вынести эту инструкцию:

Dockerfile с допущенной «небрежностью»:

```
1 FROM ubuntu:20.04
2
3 WORKDIR /home/onik110
4
5 COPY ./script.sh ./script.sh
6
7 RUN apt-get update && apt install -y vim curl git
```

Dockerfile изменённый:

```
1 FROM ubuntu:20.04
2
3 RUN apt-get update && apt install -y vim curl git
4
5 WORKDIR /home/onik110
6
7 COPY ./script.sh ./script.sh
```

Смотрите, какое здесь у нас стало преимущество. Дело в том, что часть ***WORKDIR /home/onik110*** могла бы измениться. ***COPY ./script.sh ./script.sh*** тоже может измениться. То есть мы, например, переписали script.sh или мы решили копировать другой файл, но обновление и установка библиотек при этом бы у нас закэшировалась. То есть, во-первых, это стало просто быстрее, а во-вторых, что на данном этапе для нас более важно, это размеры образов, то есть то, что у нас физически лежит на диске.

1.7 Инструкции Dockerfile

1. **FROM** — задаёт базовый (родительский) образ.
2. **LABEL** — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.
3. **ENV** — устанавливает постоянные переменные среды.
4. **RUN** — выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
5. **COPY** — копирует в контейнер файлы и папки.

6. **ADD** — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы.
7. **CMD** — описывает команду с аргументами, которую нужно выполнить когда контейнер будет запущен. Аргументы могут быть переопределены при запуске контейнера. В файле может присутствовать лишь одна инструкция CMD.
8. **WORKDIR** — задаёт рабочую директорию для следующей инструкции.
9. **ARG** — задаёт переменные для передачи Docker во время сборки образа.
10. **ENTRYPOINT** — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.
11. **EXPOSE** — указывает на необходимость открыть порт.
12. **VOLUME** — создаёт точку монтирования для работы с постоянным хранилищем.

Команды для сборки:

Создать образ на основе Dockerfile

docker build <путь, где лежит Dockerfile>

Создать образ с именем и тегом (ссылка)

docker build -t <имя_образа:тег> <путь>

Разница в использовании ADD и COPY:

Рассмотрим разницу в использовании на двух примерах — добавление архива и файла по ссылке.

Dockerfile для ADD:

```
FROM ubuntu:22.04

WORKDIR /url
ADD https://airflow.apache.org/docs/apache-airflow/2.4.0/docker-compose.yaml .
```

При использовании ADD файл (в данном случае docker-compose.yaml) по ссылке скачался:

```
onik110@onik110-pc:~/docker$ docker build -t add -f Dockerfile2 .
[+] Building 6.2s (8/8) FINISHED                                                              docker:default
 => [internal] load build definition from Dockerfile2                                                   0.0s
 => => transferring dockerfile: 149B                                                                    0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04                                         1.3s
 => [internal] load .dockerignore                                                                       0.0s
 => => transferring context: 2B                                                                         0.0s
 => [1/3] FROM docker.io/library/ubuntu:22.04@sha256:58b87898e82351c6cf9cf5b9f3c20257bb9e2dcf33af051e12ce532d7f94e3fe  4.7s
 => => resolve docker.io/library/ubuntu:22.04@sha256:58b87898e82351c6cf9cf5b9f3c20257bb9e2dcf33af051e12ce532d7f94e3fe  0.0s
 => => sha256:58b87898e82351c6cf9cf5b9f3c20257bb9e2dcf33af051e12ce532d7f94e3fe 1.34kB / 1.34kB          0.0s
 => => sha256:3d1556a8a18cf5307b121e0a98e93f1ddf1f3f8e092f1fddfd941254785b95d7 424B / 424B              0.0s
 => => sha256:97271d29cb7956f0908cfb1449610a2cd9cb46b004ac8af25f0255663eb364ba 2.30kB / 2.30kB          0.0s
 => => sha256:6414378b647780fee8fd903ddb9541d134a1947ce092d08bdeb23a54cb3684ac 29.54MB / 29.54MB        2.4s
 => => extracting sha256:6414378b647780fee8fd903ddb9541d134a1947ce092d08bdeb23a54cb3684ac               2.1s
 => [3/3] ADD https://airflow.apache.org/docs/apache-airflow/2.4.0/docker-compose.yaml .                0.1s
 => [2/3] WORKDIR /url                                                                                  0.1s
 => [3/3] ADD https://airflow.apache.org/docs/apache-airflow/2.4.0/docker-compose.yaml .                0.0s
 => exporting to image                                                                                  0.0s
 => => exporting layers                                                                                 0.0s
 => => writing image sha256:32a27bfae2475a3a6a23dc2f1e8e386c0f06c38e61708f76e093241600da89ef            0.0s
 => => naming to docker.io/library/add                                                                  0.0s
onik110@onik110-pc:~/docker$
```

Dockerfile для COPY:

```
1 FROM ubuntu:22.04
2
3 WORKDIR /url
4 COPY https://airflow.apache.org/docs/apache-airflow/2.4.0/docker-compose.yaml .
```

При использовании COPY файл по ссылке скачивать нельзя, поэтому упали с ошибкой:

```
onik110@onik110-pc:~/docker$ docker build -t copy -f Dockerfile2 .
[+] Building 1.4s (3/3) FINISHED                                                docker:default
 => [internal] load build definition from Dockerfile2                                   0.0s
 => => transferring dockerfile: 150B                                                    0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04                         1.3s
 => [internal] load .dockerignore                                                       0.0s
 => => transferring context: 2B                                                         0.0s
Dockerfile2:4
--------------------
   2 |
   3 |     WORKDIR /url
   4 | >>> COPY https://airflow.apache.org/docs/apache-airflow/2.4.0/docker-compose.yaml .
   5 |
--------------------
ERROR: failed to solve: source can't be a URL for COPY
onik110@onik110-pc:~/docker$
```

Чтобы более подробно ознакомиться с разницей между COPY и ADD, полезно будет заглянуть в документацию (COPY, ADD).

Разница в использовании ENTRYPOINT и CMD:

Инструкция ENTRYPOINT в Dockerfile используется для указания исполняемого файла или команды по умолчанию, которая должна быть запущена при старте контейнера. Она позволяет определить основную цель контейнера и задать команду, которая будет выполняться по умолчанию.

Инструкция CMD используется для предоставления аргументов по умолчанию инструкции ENTRYPOINT или для определения команды по умолчанию, которая будет выполняться при запуске контейнера, если ENTRYPOINT не задан.

Различия:

- Назначение: ENTRYPOINT используется для определения основной цели контейнера, в то время как CMD обеспечивает поведение по умолчанию, которое может быть переопределено.

- Переопределение: При запуске контейнера вы можете легко переопределить инструкцию CMD, предоставив дополнительные аргументы командной строки, в то время как инструкцию ENTRYPOINT нельзя переопределить, если она явно не указана с помощью флага -entrypoint.

- Взаимодействие: Если в Dockerfile указаны и ENTRYPOINT, и CMD, инструкция CMD предоставляет аргументы по умолчанию для инструкции ENTRYPOINT.

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT [“exec_entry”, “p1_entry”]
No CMD	error, not allowed	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

1.8 YAML и Docker-Compose

Docker-Compose — это программа позволяющая удобно работать с многоконтейнерными приложениями. Работает это следующим образом — в специальном файле docker-compose.yaml пишем инструкции, которые необходимы под поставленную задачу. Получается, мы в удобном формате описываем структуру нашего приложения, а docker-compose на основе уже этого файла будет запускать контейнеры.

Формат для которого описываются инструкции для docker-compose, называется YAML. YAML — это формат сериализации данных, который используется для представления структурированных данных в удобочитаемом текстовом виде. Он часто применяется в конфигурационных файлах, обмене данными между приложениями и в других сценариях, где требуется простота и читаемость.

Основные характеристики YAML:

a) *Читаемость:*

YAML разработан с акцентом на удобочитаемость для человека. Структура данных представляется в виде отступов, что делает его более понятным по сравнению с другими форматами, такими как JSON или XML.

b) *Структура:*

YAML поддерживает различные структуры данных, такие как скалярные значения (строки, числа), списки и ассоциативные массивы (словари).

c) *Отступы:*

В YAML используется отступ для обозначения уровней вложенности, что делает его визуально понятным.

d) *Типы данных:*

YAML поддерживает различные типы данных, включая строки, числа, булевы значения, списки и карты (ассоциативные массивы).

Ниже приведён пример Yaml-файла:

```yaml
name: Ivan Ivanov
age: 21
is_student: true
languages:
  - Russian
  - English
  - German
address:
  street: Pushkin St, 10
  city: Moscow
  country: Russia
```

В этом примере представлены данные о человеке, включая его имя, возраст, статус студента, языки и адрес.

Основные функции Docker Compose:

- *Определение сервисов*

В файле docker-compose.yml можно определить несколько сервисов (контейнеров), которые составляют ваше приложение. Каждый сервис может иметь свои собственные настройки, такие как образ, порты, переменные окружения и зависимости.

- *Управление зависимостями*

Docker Compose позволяет указать, какие сервисы зависят от других. Это упрощает настройку и запуск приложений, состоящих из нескольких компонентов.

- *Упрощение команд*

Вместо того чтобы вручную запускать и управлять каждым контейнером, вы можете использовать одну команду (docker-compose up), чтобы запустить все сервисы, определенные в вашем файле.

- *Сетевые настройки*

Docker Compose автоматически создает сеть для ваших контейнеров, что позволяет им легко взаимодействовать друг с другом.

- *Масштабирование*

Вы можете легко масштабировать сервисы, увеличивая или уменьшая количество экземпляров контейнеров с помощью одной команды.

Пример файла docker-compose.yaml:

```yaml
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./html:/usr/share/nginx/html

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: mydatabase
    volumes:
      - db_data:/var/lib/mysql

volumes:
  db_data:
```

Использование Docker-Compose:

1) Создайте файл ***docker-compose.yaml***: Определите ваши сервисы, их настройки и зависимости.

2) Запустите приложение: Выполните команду ***docker-compose up*** в каталоге с вашим файлом ***docker-compose.yaml***. Это создаст и запустит все указанные контейнеры.

3) Остановите приложение: Используйте ***docker-compose down*** для остановки и удаления всех контейнеров, сетей и томов, созданных с помощью ***docker-compose up***.

Команды:

docker-compose ps — список контейнеров
docker-compose up — поднять приложение
docker-compose stop — остановить поднятые контейнеры
docker-compose start — запустить остановленные контейнеры
docker-compose down — остановить и удалить контейнеры

Практическое задание

Задание 1

Скачайте с интернет-ресурса Docker Hub образ node, postgres и yandex/clickhouse-server. Для node возьмите tag — latest, 16 и alpine. Выведите список текущих образов,

убедитесь что их ровно 5 (лишние удалите). Сравните размер node latest, node 16 и node alpine. Результаты продемонстрируйте преподавателю.

Задание 2

Запустите контейнер на основе образа с node, проверьте список всех контейнеров. Что интересного можно заметить в столбце STATUS? Проделайте тоже самое с образом yandex/clickhouse-server. Сравните результаты.

Подсказка: запустить или остановить контейнер можно с помощью команд — $ sudo docker start(stop) <NAMES or CONTAINER ID>

Задание 3

Зайдите в работающий контейнер yandex/clickhouse-server в директорию home, выведите содержание файла users.xml. В файле необходимо найти информацию об IP-address or network mask и Hostname.

Скачайте образ nginx'a с тегом 1.23. Поднимите контейнер на основе этого образа. В самом контейнере вам нужно найти файл, который лежит по пути /etc/nginx/conf.d/default.conf. В этом файле вам нужно найти настройку server_name (при помощи, например, утилит cat или grep). Она находится в секции server.

Задание 4

Поднимите контейнеры на основе разных образов, указываете опцию --rm и фоновый режим (-d):

- python
- nginx
- postgres
- redis
- golang
- node

Сколько у вас будет работающих (поднятых) контейнеров? Сколько всего будет контейнеров (включая остановленные)? Остановите сначала все работающие контейнеры, а затем удалите все контейнеры одной командой.

Задание 5

Создайте контейнер с PostgreSQL, дайте ему имя и порт, откройте в фоновом режиме с возможностью быстрого удаления. Создайте локальный контейнер с программой на языке C с приложением клиентом для СУБД (см. ЛР 8 по Базам данных). Используйте Docker Compose для их общего подключения. Проверьте правильность работы контейнеров.