

Практика использования и наиболее полезные конструкции Docker

Подвойский А.О.

Содержание

1	Общие сведения о системе Docker	1
1.1	Термины и определения	1
1.2	Высокоуровневая концепция	2
1.3	Установка	5
1.4	Анализ manifest-файлов Docker	5
1.5	Контейнеры	7
1.6	Создание образов из Dockerfile	7
1.7	Пример файла docker-compose.yml	8
1.8	Порядок работы с docker-compose	9
1.9	Работа с ключами и сертификатами	9
1.10	Организация собственного реестра	12
1.11	Сокращение размера образа	13
2	Базовые концепции, связанные с системой Docker	14
2.1	Структура стека протоколов TCP/IP	14
2.2	Формат IP-адреса	15
2.3	Виртуальный сетевой интерфейс	15
3	Пример создания простого web-приложения	15
4	Пример работы с базой данных Redis	18
5	Наиболее полезные конструкции	19
5.1	Манипуляции с контейнерами	19
5.2	Информация о контейнере	19
5.3	Удаление контейнеров и образов	20
6	Docker для ML	20
	Список литературы	21

1. Общие сведения о системе Docker

1.1. Термины и определения

Контейнер – это НЕ виртуальная машина, а *приложение и его зависимости*, упакованные в стандартизированное, изолированное, легковесное окружение. Контейнеры используют виртуа-

лизацию на уровне ядра операционной системы и не содержат своей собственной операционной системы, поэтому они такие легкие.

Образ – неизменяемая сущность, снимок контейнера. Образ состоит из слоев.

localhost (так называемый, «локальный хост», по смыслу «этот компьютер») – стандартное, официально зарезервированное доменное имя для *частых* (или что то же самое *локальных*) IP-адресов¹ *петлевого интерфейса*² (диапазон 127.0.0.1 – 127.255.255.255). Использование IP-адреса 127.0.0.1 позволяет устанавливать соединение и передавать информацию для программ-серверов, работающих на том же компьютере, что и программа-клиент. Примером может быть запущенный на компьютере веб-сервер приложений, обращение к которому выполняется с этого же компьютера для веб-разработки на данном компьютере без необходимости выкладывать веб-программу в сеть Интернет, пока ее разработка не закончена. Традиционно IP-адресу 127.0.0.1 однозначно сопоставляется имя хоста **localhost**.

порт – целое неотрицательное число, записываемое в заголовках *протоколов транспортного уровня* модели OSI (TCP, UDP, SCTP, DCCP). Используется для определения процесса-получателя пакета в пределах одного хоста (локального компьютера).

1.2. Высокоуровневая концепция

На одной физической машине может находиться несколько виртуальных машин (например, ОС Windows, ОС Linux и ОС MacOS X), которые не будут знать ничего друг о друге (рис. 1). На этих виртуальных машинах, разумеется, могут быть установлены различные приложения, специфичные для конкретной операционной системы и т.д. Но каждая такая виртуальная машина не знает, что она виртуальная.

В случае с Docker внутри одного контейнера может существовать несколько приложений, которые взаимодействуют с одной операционной системой (ОС физической машины).

Внутри контейнера может быть:

- Приложение (Java/Ruby/Go etc.),
- Библиотеки (libxml, wkhtmltopdf etc.),
- Сервисы (PostgreSQL, Redis etc.),
- Утилиты (sbt, ant, gems, eggs etc.),
- Фреймворки (jre, ruby etc.)

В разных контейнерах могут стоять различные версии перечисленных объектов.

Docker состоит из Daemon, Client и Registry. *Docker-демон* запускается на машине (локальная машина, сервер или что-то еще), на которой планируется запускать контейнеры. Также предоставляет API и управляет docker-объектами (Dockerfile, ...).

Docker-клиент принимает команды пользователя и общается по API с docker-демоном.

В сущности Docker-pipeline представляет собой следующую цепочку:

- На базе файла **Dockerfile** собирается *образ* с помощью **docker build** (на локальной машине),

¹Уникальный сетевой адрес узла в компьютерной сети, построенной на базе стека протоколов TCP/IP

²Обычно используется термин *loopback*, который описывает методы или процедуры маршрутизации электронных сигналов, цифровых потоков данных, или других движущихся сущностей от их источника и обратно к тому же источнику без специальной обработки или модификации

Containers vs VM

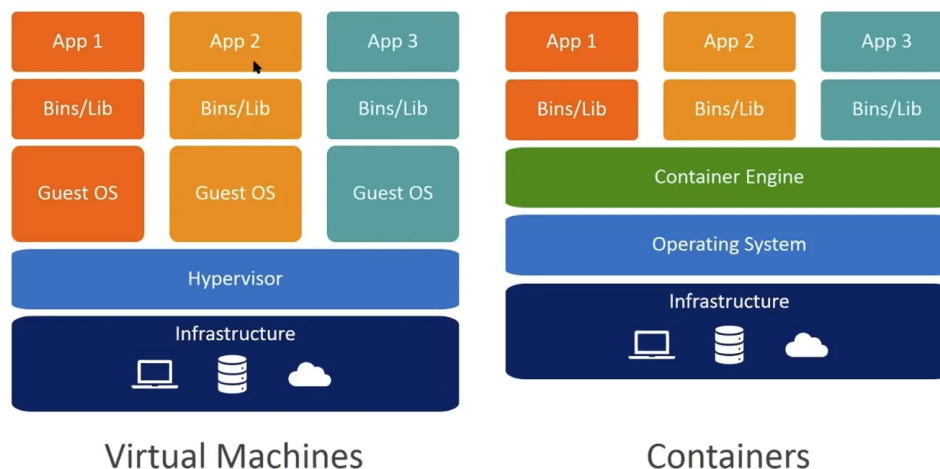


Рис. 1. Сравнение архитектуры docker-контейнеров и виртуальных машин

- На базе образа запускается *контейнер* с помощью `docker run`.

Пример Dockerfile

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"

RUN apk update && apk upgrade && apk add bash # форма оболочки

COPY . ./app

ADD https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample_vids/vid1.mp4 \
/my_app_directory

RUN ["mkdir", "/a_directory"] # JSON-форма

CMD ["python", "./my_script.py"]
```

FROM: в роли базового образа выступает официальный Python-образ с тегом 3.7.2-alpine3.8.

LABEL: добавляет метаданные к образу, предоставляет контактную информацию. Она не замедляет процесс запуска и не занимает много места, наоборот, обеспечивает образ полезной информацией.

ENV: создает переменную окружения, которая становится доступной во время запуска контейнера.

RUN: запускает заданную инструкцию внутри контейнера. Docker фиксирует состояние образа после каждой инструкции RUN. Чаще всего используется для установки нужных пакетов внутри контейнера. RUN и ее родственные конструкции (CMD, ENTRYPOINT) – могут быть как в форме оболочки, так и в JSON-форме. Во втором случае используют JSON-синтаксис

```
RUN ["my_executable", "my_first_param1", "my_second_param2"] |
```

А в примере выше использовалась форма оболочки

```
RUN apk update && apk upgrade && apk add bash
```

COPY: инструкция `COPY . ./app` говорит Docker'у, что нужно скопировать файлы и папки из вашей локальной сборки в рабочую директорию образа. `COPY` создаст все нужные папки, если они отсутствуют.

ADD: делает то же самое, что и `COPY`, но с двумя отличиями. `ADD` может загружать файл по URL, а также извлекать локальные tar-файлы.

В примере выше `ADD` копировала файлы по URL внутрь директории контейнера. Но официальной документации не рекомендуется использовать `ADD`. Однако, в общем случае лучше воспользоваться более простой инструкцией `COPY` для копирования файлов и каталогов в локальном контексте создания или инструкцией `RUN` с запуском `curl` или `wget` для загрузки удаленных ресурсов.

CMD: инструкция для запуска чего-либо во время запуска самого контейнера, то есть инструкция запускается во время инициализации контейнера. По ходу сборки она не фиксирует никакого результата. В примере выше во время сборки запускался сценарий `my_script.py`. Если была определена инструкция `ENTRYPOINT`, то заданная здесь инструкция будет интерпретироваться как аргумент для `ENTRYPOINT`. Инструкция `CMD` замещается любыми аргументами, указанными в команде `docker run` после имени образа. В действительности выполняется только самая последняя инструкция `CMD`, а все предыдущие инструкции `CMD` будут отменены.

WORKDIR: меняет текущую рабочую директорию для инструкций: `COPY`, `ADD`, `RUN` и `ENTRYPOINT`. Предпочтительно задавать абсолютный путь. `WORKDIR` автоматически создает директорию, если ее еще нет. Можно использовать несколько `WORKDIR`-инструкций.

ARG: определяет переменную для передачи из командной строки в образ. Для `ARG` можно указать значение по умолчанию `ARG my_var=my_default_value`. Например

```
...
ARG my_var=my_default_value

ENTRYPOINT ["python", "./app/my_script.py", "my_var"] # будет прочитано значение my_var
...
```

В отличие от `ENV`-переменных `ARG`-переменные не доступны для запущенных контейнеров. Другими словами, нельзя напрямую установить значение переменных `ENV` при запуске сборки образа с помощью `docker build`. Но можно использовать `ARG` для передачи значений в инструкции `ENV`.

Например <https://vsupalov.com/articles/>

Dockerfile

```
...
# переменная buildtime_variable со значением по умолчанию
ARG buildtime_variable=default_value # существует только для передачи значения в переменную окружения
ENV env_var_name=$buildtime_variable
...
```

Теперь при запуске сборки образа можно переопределять значение переменной `buildtime_variable` «на лету»

```
docker build --build-arg buildtime_variable=a_value ...
```

То есть можно изменять значение переменной `buildtime_variable` без необходимости вносить изменения в `Dockerfile`.

ENTRYPOINT: также позволяет задавать дефолтные команды и аргументы во время запуска контейнера. Аргументы командной строки, передаваемые `docker run my_image_name`, добавляются к аргументам инструкции **ENTRYPOINT**. Например,

```
docker run my_image_name bash
```

добавляет аргумент `bash` в конец, ко всем другим аргументам **ENTRYPOINT**.

Dockerfile должен обязательно содержать либо **CMD**-инструкцию, либо **ENTRYPOINT**-инструкцию. В официальной документации есть несколько советов, которые помогут сделать выбор между **CMD** и **ENTRYPOINT**

- Если вам нужно запускать одну и ту же команду несколько раз, выбирайте **ENTRYPOINT**,
- Используйте **ENTRYPOINT**, когда ваш контейнер выступает в роли исполняющей программы,
- При наличии дополнительных дефолтных аргументов, которые могут быть изменены через командную строку, лучше подойдет **CMD**.

EXPOSE: показывает какой порт пробрасывать из контейнера. Используйте команду `docker run` с флагом `-p` для пробрасывания и сопоставления нескольких портов во время запуска. Флаг в верхнем регистре `-P` будет пробрасывать все открытые порты.

VOLUME: определяет, где контейнер будет хранить постоянные данные и получать к ним доступ.

1.3. Установка

Установить **Docker** можно с помощью менеджера пакетов `conda`

```
conda install -c conda-forge docker-py
```

На текущий момент без серьезных проблем **Docker** работает только на 64-битовом **Linux**.

Для нормальной работы на **MacOS X** или **Windows** потребуется дополнительно установить какую-либо виртуальную машину в полной комплектации или пакет **Docker Toolbox**:

- для **MacOS X**: https://docs.docker.com/toolbox/toolbox_install_mac/,
- для **Windows**³: https://docs.docker.com/toolbox/toolbox_install_windows/; после установки **Docker Toolbox** останется только запустить **Docker Quick Start Terminal**.

То есть на операционных системах **Windows** и **MacOS X** для того, чтобы можно было работать с **Docker**, требуется поднять виртуальную машину **Linux VM**, внутри которой будут существовать `docker`-демон и контейнеры.

1.4. Анализ manifest-файлов Docker

Чтобы извлечь данные с удаленного репозитория нужно воспользоваться конструкцией

```
docker pull leorfinkelberg/app_name:tag_name
```

или что то же самое (по умолчанию, если хост не указан, используется **DockerHub**)

```
$ docker pull registry.hub.docker.com/leorfinkelberg/app_name:tag_name
```

Пытаемся скачать `manifest`-файл

³Поддерживается даже **Windows 7**

```
$ curl -D - -s https://registry.hub.docker.com/v2/leorfinkelberg/manifests/latest
# вернет
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Docker-Distribution-API-Version: registry/2.0
Www-Authenticate: Bearer realm="https://auth.docker.io/token",service="registry.docker.io",scope
    ="repository:leorfinkelberg/myapp:pull"
Date: Thu, 04 Jun 2020 17:57:01 GMT
Content-Length: 163
Strict-Transport-Security: max-age=31536000

{"errors":[{"code":"UNAUTHORIZED","message":"authentication required","detail":[{"Type":"
    repository","Class":"","Name":"leorfinkelberg/myapp","Action":"pull"}]}]}
```

Здесь флаг `-D` сохраняет заголовки, возвращенные сервером, `-s` заставляет выводить минимум информации.

Возникли сложности с авторизацией. Смотрим на строку `Www-Authenticate` и на основании информации, приведенной в этой строке, строим запрос

```
$ curl -D - -s 'https://auth.docker.io/token?service=registry.docker.io&scope=repository:
    leorfinkelberg/myapp:pull'
# вернет
HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 04 Jun 2020 13:46:22 GMT
Transfer-Encoding: chunked
Strict-Transport-Security: max-age=31536000

{"token":"eyJhbGciOiJSUzI1NiIsInR5cCI6I..}"
```

Параметр `scope` описывает запрашиваемые права. Затем создаем переменную окружения `REGISTRY_TOKEN`

```
export REGISTRY_TOKEN="eyJhbGciOiJSUzI1NiIsInR5cCI6I.."
```

А затем конструируем такой запрос и получаем `manifest`-файл в формате `json`

```
$ curl -D - -s -H "Authorization: Bearer $REGISTRY_TOKEN" https://registry.hub.docker.com/v2/
    leorfinkelberg/myapp/manifests/latest
# вернет
HTTP/1.1 200 OK
Content-Length: 5778
Content-Type: application/vnd.docker.distribution.manifest.v1+prettyjws
Docker-Content-Digest: sha256:cb0c53e4c8471a2deb8d22df00646ee2abe81cb11818ed12d0fe4e697ccec93f
Docker-Distribution-API-Version: registry/2.0
Etag: "sha256:cb0c53e4c8471a2deb8d22df00646ee2abe81cb11818ed12d0fe4e697ccec93f"
Date: Thu, 04 Jun 2020 18:11:59 GMT
Strict-Transport-Security: max-age=31536000

{
  "schemaVersion": 1,
  "name": "leorfinkelberg/myapp",
  "tag": "latest",
  "architecture": "amd64",
  "fsLayers": [
    {
      "blobSum": "sha256:bd1a014b71c4b5b06a7f57486a8be2d690fc6034ede049ab63e9e827c9814e5a"
    },
    ...
  ]
}
```

```

    "signatures": [
      {
        "header": {
          "jwk": {
            "crv": "P-256",
            "kid": "PDNB:TMYN:5AZL:Y3UQ:6ACR:VJER:ON6K:H5XV:AIUX:7ZBZ:2PUD:GUI4",
            "kty": "EC",
            "x": "o_2HgAeBYjFsEhtDbFsB0afzrwkODIgIsDg7Tslk33o",
            "y": "04Wp7S22uBJYbca2CABSBI18ZkJUksit8dLQDvshhro"
          },
          "alg": "ES256"
        },
        "signature": "eWlDs1YAVkLG1t1SfB9-Pboa3aWIuKEkaMomNx07xH...",
        "protected": "eyJmb3JtYXRmZW5ndGgiOiJ1eMzEsImZvcmlhdFRhaWwiOi..."
      }
    ]
  }
}

```

Сохраняем manifest-файл

```
curl -o manifest.json -s -H "Authorization: Bearer $REGISTRY_TOKEN" https://registry.hub.docker.com/v2/leorfinkelberg/myapp/manifests/latest
```

Контрольные суммы слоев образа можно посмотреть так

```
$ $ docker inspect --format {{.RootFS.Layers}} leorfinkelberg/myapp
```

1.5. Контейнеры

Контейнеры представляют собой средства инкапсуляции приложения вместе со всеми его зависимостями.

Поскольку **Docker** сам по себе не обеспечивает реализацию любого типа виртуализации, контейнеры всегда должны соответствовать ядру хоста – контейнер на **Windows Server** может работать только на хосте под управлением операционной системы **Windows Server**, а 64-битный **Linux**-контейнер работает только на хосте с установленной 64-битной версией операционной системы **Linux** [1].

1.6. Создание образов из Dockerfile

Dockerfile – это обычный текстовый файл, содержащий набор операций, которые могут быть использованы для создания **Docker**-образа.

Пример. Для начала создадим новый каталог и собственно **Dockerfile**

```

$ mkdir cowsay
$ cd cowsay
$ touch Dockerfile

```

Затем в созданный **Dockerfile** добавим следующее

Dockerfile

```

FROM debian:wheezy
MAINTAINER John Smith <john@smith.com>
RUN apt-get update && apt-get install -y cowsay fortune
COPY entrypoint.sh /
ENTRYPOINT ["/entrypoint.sh"]

```

Инструкция **FROM** определяет базовый образ ОС (это в данном случае **debian** с уточненной версией «wheezy»). Инструкция **FROM** является строго обязательной для всех файлов **Dockerfile** как самая первая незакомментированная инструкция.

Инструкция **MAINTAINER** просто определяет информацию, позволяющую связаться с автором образа.

Инструкция **COPY** копирует файл из файловой системы хоста в файловую систему образа, где первый аргумент определяет файл хост, а второй – целевой путь.

Инструкция **RUN** определяет команды, выполняемые в командной оболочке внутри данного образа.

Комментарии к скрипту **entrypoint.sh**. Файл **entrypoint.sh** должен лежать в той же директории, что и файл **Dockerfile** и иметь содержание на подобие следующего

entrypoint.sh

```
if [ $# -eq 0 ]; then
    /usr/games/fortune | /usr/games/cowsay
else
    /usr/games/cowsay "$@"
fi
```

Здесь конструкция [...] – это форма⁴ команды **test** для проверки различных условий. Последовательность символов **##** – встроенная переменная, обозначающая количество аргументов в командной строке. Последовательность символов **\$\$** – это все аргументы командной строки, а **"\$@"** – все аргументы командной строки, заключенные по отдельности в кавычки [2, стр. 44].

После сохранения необходимо сделать этот файл исполняемым при помощи команды **chmod +x entrypoint.sh**.

Теперь можно создать образ на основе файла **Dockerfile**

```
docker build -t test/cowsay-dockerfile .
```

Здесь **test** – имя репозитория, а **cowsay-dockerfile** – имя образа.

После этого можно запускать контейнер, который строится на основе образа **test/cowsay-dockerfile**

```
docker run test/cowsay-dockerfile Moo
```

1.7. Пример файла **docker-compose.yml**

Рассмотрим в качестве примера файл **docker-compose.yml** для автоматизации процесса настройки и запуска контейнера

docker-compose.yml

```
registry:
  restart: always
  image: registry:2
  ports:
    - 443:5000
  environment:
    REGISTRY_HTTP_ADDR: 0.0.0.0:5000
    REGISTRY_HTTP_HOST: https://registry.kis.im
    REGISTRY_HTTP_TLS_LETSENCRYPT_CACHEFILE: /tmp/le.cache
    REGISTRY_HTTP_TLS_LETSENCRYPT_EMAIL: ov@rebrain.com
```

⁴Есть еще вариант **test** выражение, но форма [выражение] более популярна


```
REGISTRY_HTTP_TLS_LETSENCRYPT_HOSTS: [registry.kis.im]
volumes:
  - /path/data:/var/lib/registry
  - /path/cers:/certs
```

Описание. Создаем сервис **registry** с политикой постоянной перезагрузки. Образ используется официальный **registry-2**. Пробрасываем 443 порт на 5000 внутрь контейнера.

Для запуска контейнера достаточно набрать

```
$ docker-compose up -d
```

1.8. Порядок работы с docker-compose

Docker-compose придумали для того, чтобы создавать себе на локальном компьютере (или окружении для *тестов*) *эмуляцию* продакшн среды. В действительности речь идет о распределенном приложении, которое будет работать на нескольких машинах, но для того чтобы посмотреть связку этих раздельных блоков на локальной машине (например, для проведения интеграционных тестов) можно использовать docker-compose.

Обычный порядок работы начинается с выполнения команды для запуска приложения

```
docker-compose up -d
```

Команды **docker-compose logs** и **docker-compose ps** могут использоваться для проверки состояния приложения и как вспомогательное средство при отладке.

После внесения изменений в исходный код нужно выполнить

```
docker-compose build
```

а затем

```
docker-compose up -d
```

При этом будет создан новый образ и заменен работающий контейнер.

Замечание

Compose сохраняет все ранее существовавшие тома из старых контейнеров, таким образом, базы данных и кэши остаются неизменными при переходе к новым версиям контейнеров (это может привести к беспорядку, поэтому будьте осторожны при замене контейнеров)

Если создание нового образа не требуется, но внесены изменения в **docker-compose.yml**, то выполните команду **docker-compose up -d**, чтобы заменить контейнер на точно такой же, но с новыми настройками.

После завершения сеанса работы с приложением выполните команду **docker-compose stop** для его остановки. Тот же самый комплект контейнеров будет повторно запущен при выполнении команды **docker-compose start** или **docker-compose up -d**, если не был изменен исходный код.

Для окончательного удаления набора контейнеров приложения используйте команду **docker-compose rm**.

1.9. Работа с ключами и сертификатами

Создать ключ и сертификаты можно с помощью утилиты **openssl**. Создадим корневой ключ

```
$ openssl genrsa -out rootCA.key 2048
```

Теперь на основе ключа можно сгенерировать сертификат

```
$ openssl req -x509 -new -key rootCA.key -days 10000 -out rootCA.crt
```

Ключ

```
$ cat rootCA.key
# выведет
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAlAK30s5wgbTFNGzfqbVTbqjv2ExRI0jvS/wHwIWrPuR19K4
4TPegK8dM5DaNbWLCs0Lynp4OuWt56MLyH1Ewu9xynyc0zRP40otmUGtain8HL
TqPMgA2vvVUA3JbFuN1wsyI9mCdxkSbLH5jwKslkWw9tUPP9k+Mi6NAFF3/RNMPw
5agJqp05M+9AnlgNchqosMxomdQpkXLZuTr7zWrk5vjXksrvszM4nJABGrbrGTnz
ZK51aW7brt023fnxU3HwhmkKthvE8oYhyM23c6G+Ti2zpdI8ILlCMfC/rdKJUmeX
vjuM98zUudxTH6MFVvX8Vq4OUxaVmisXWviGAwIDAQABAoIBAQCkbuY3HnUtGPGg
qu/G/1zyF1X55D6e7S+wWJugnZDCdEyx021MTzm666f78gWceLTiyNxQ0paZklPv
HaoCe//Xln7I9hKS3wZ+VLqFhQpwJXjdYoq4ZdL5PZudGVbtNHPxFOlI27QbKoOW
4RMxfqBPtBwueqLdb4WhDH402H7XRsb9t+bTVtb7agtCjPAvZoV8x5EHlj3LErO
...
-----END RSA PRIVATE KEY-----
```

Сертификат

```
$ cat rootCA.crt
# выведет
-----BEGIN CERTIFICATE-----
MIID9zCCAtgAwIBAgIUB/wdkhR8j3McjC1vNr/OelcPLoMwDQYJKoZIhvcNAQEL
BQAwYoxCzAUBgNVBAYTA1JVMQ8wDQYDVQQIDAZNb3Njb3cxZzANBgNVBACMBk1v
c2NvdzEQMA4GA1UECgWRHR2F6cHJvbTEQMA4GA1UECwwHT2lsJkdhcjELMAkGA1UE
AwwCQOEKdAmBgkqhkiG9w0BCQEwGw1b3IuZmlua2VsYmVyZ0B5Yw5kZXgucnUw
HhcNMjAwNjAOMTk1NDZWhcNNDcxMDIxMTk1NDZwCjBiZELMAkGA1UEBhMCU1Ux
DzANBgNVBAGMBk1vc2NvdzEPMA0GA1UEBwwGTW9zY293MRAwDgYDVQQKDAdHYXpw
cm9tMRAwDgYDVQQQLDAdPaWwmR2FzMQswCQYDVQQDDAJDQTEoMCYGCsqsIb3DQEJ
ARYZbGVVci5maW5rZWxiZXJnQlhlbmRleC5ydTCCASIdQYJKoZIhvcNAQEBBQAD
ggEPADCCAQoCggEBANSwCtzr0cIG0xTRs36m1U26o79hMUSNI70v8B8CFqz7kdfS
u0Ez3oCvHTOQ2jW1iwrNC8p6eDrliFleejC8h9RMLvccp8nDsOT+NKLZ1BrWop/B
y06jzIANr71VANYwxbjdcLMiPZgncZEmyx+Y8CrJZFsPbVDz/ZPjIujQBRd/OTTD
80WoCaqdOTPVqJ5YDXIaqLDMaJnUKZFy2bk6+81q50b415LK77Mz0JyQARq26xk5
82SudWlu267dNt358VNx8IZpCrYbXPKGIcJNt30hvk4ts6XSPCC5QjHwv63SiVJh
...
-----END CERTIFICATE-----
```

Прочитать сертификат

```
$ openssl x509 -in rootCA.crt -text
# выведет
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            07:fc:1d:92:14:7c:8f:73:1c:8c:2d:6f:36:bf:f4:7a:57:0f:2e:83
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = RU, ST = Moscow, L = Moscow, O = Gazprom, OU = Oil&Gas, CN = CA,
        emailAddress = leor.finkelberg@yandex.ru
        Validity
            Not Before: Jun  4 19:54:03 2020 GMT
            Not After : Oct 21 19:54:03 2047 GMT
        Subject: C = RU, ST = Moscow, L = Moscow, O = Gazprom, OU = Oil&Gas, CN = CA,
        emailAddress = leor.finkelberg@yandex.ru
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public-Key: (2048 bit)
```

Modulus:

```
00:d4:b0:0a:dc:eb:39:c2:06:d3:14:d1:b3:7e:a6:
d5:4d:ba:a3:bf:61:31:44:8d:23:bd:2f:f0:1f:02:
16:ac:fb:91:d7:d2:b8:e1:33:de:80:af:1d:33:90:
da:35:b5:8b:0a:cd:0b:ca:7a:78:3a:e5:88:52:de:
7a:30:bc:87:d4:4c:2e:f7:1c:a7:c9:c3:b3:44:fe:
```

...

Генерируем приватный ключ для registry

```
$ openssl genrsa -out registry.key 2048
```

И по ключу создаем запрос сертификата с помощью

```
$ openssl req -new -key registry.key -out registry.csr
```

Подписываем корневым сертификатом ключ, который потом надо будет отдать серверу

```
openssl x509 -req -in registry.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out
registry.crt -days 5000
# будет создан registry.crt
```

Файл docker-compose.yml будет выглядеть так

```
registry:
  restart: always
  image: registry:2
  ports:
    - 443:5000
  environment:
    REGISTRY_HTTP_ADDR: 0.0.0.0:5000
    REGISTRY_HTTP_HOST: https://registry.kis.im
    REGISTRY_HTTP_TLS_CERTIFICATE: /path/to/registry.crt
    REGISTRY_HTTP_TLS_KEY: /path/to/registry.key
  volumes:
    - /path/data:/var/lib/registry
    - /etc/cers:/etc/certs
```

Скачаем образ и поместим его в registry

```
$ docker pull alpine:latest
# перетегиваем
$ docker tag alpine:latest registry.kis.im/leorfinkelberg/alpine:dev
```

Теперь можно отправить перетегированный образ в репозиторий

```
$ docker push registry.kis.im/leorfinkelberg/alpine:dev
```

Docker может не доверять самоподписанным сертификатам. Чтобы Docker доверял самоподписанным сертификатам нужно создать в /etc/ приведенную ниже цепочку каталогов и поместить туда сертификат, т.е.

```
$ cat /etc/docker/certs.d/registry.kis.im/ca.crt
...
```

Теперь можно скачивать

```
$ docker pull registry.kis.im/leorfinkelberg/alpine:dev
```

1.10. Организация собственного реестра

Простейшим способом создания *локального реестра* является использование официального образа

```
docker run -d -p 5000:5000 registry:2
```

Теперь у нас есть работающий реестр, и можем присваивать образам соответствующие теги и выгружать их в этот реестр. При использовании механизма `docker-machine` остается возможность указания адреса `localhost`.

```
# создаем псевдоним для образа amouat/identidock:0.1 в пространстве имен localhost:5000
$ docker tag amouat/identidock:0.1 localhost:5000/identidock:0.1
$ docker push localhost:5000/identidock:0.1
```

Если сейчас удалить локальную версию, то в любой момент можно извлечь ее из локального реестра

```
$ docker rmi localhost:5000/identidock:0.1
$ docker pull localhost:5000/identidock:0.1
```

Образы можно извлекать по дайджесту⁵

```
$ docker pull localhost:5000/identidock@sha256:d20...45345
```

Главным преимуществом использования дайджеста является абсолютная гарантия того, что извлекается в точности тот образ, который нужен пользователю. При извлечении (загрузке) по тегу можно оказаться в ситуации, когда имя тегированного образа было изменено, а пользователь об этом не знает.

Главное обоснование использования частного (локального) реестра – необходимость организации централизованного хранилища для группы разработчиков или всей организации. Это означает, что потребуется возможность загрузки образов из реестра, выполняемая удаленным демоном Docker. Но при попытке обращения к локальному реестру из вне, будет получена ошибка «Error response from daemon: unable to ping registry endpoint».

Попробуем разобраться в том, что произошло. Демон Docker запретил соединение с удаленным хостом, так как этот хост не имеет действительного сертификата TLS (Transport Layer Security). До этого установление соединения разрешалось только потому, что в механизме Docker предусмотрено особое исключение для загрузки с серверов, расположенных на локальном хосте (то есть по адресу `localhost`).

Возникшую проблему можно решить одним из трех способов:

- перезапустить каждый демон Docker, которому требуется доступ к нашему реестру с аргументом `--insecure-registry 192.168.1.100:5000`⁶,
- установить на хосте реестра подписанный сертификат от аккредитованного центра сертификации,
- установить на хосте реестра самоподписанный сертификат и скопировать его на все хосты демонов Docker, которым должен быть предоставлен доступ к этому реестру.

Для создания собственного *самоподписанного сертификата* можно воспользоваться утилитой `OpenSSL`. Все операции должны быть выполнены на компьютере, который предполагается использовать в качестве сервера в течение длительного времени

⁵Уникальное хэш-значение на основании содержимого образа и его метаданных

⁶Разумеется нужно указывать адрес и номер порта, который вы выбрали для своего сервера

```
~$ mkdir registry_certs
~$ openssl req -newkey rsa:4096 -nodes -sha256 -keyout registry_certs/domain.key -x509 \
-days 365 -out registry_certs/domain.crt
```

Здесь создается самоподписанный сертификат x509 и 4096-битный закрытый ключ по алгоритму RSA. Сертификат подписан с помощью дайджеста SHA256 и действителен в течение 365 дней. После завершения процесса мы получаем файл сертификата `domain.crt`, который будет совместно использоваться всеми клиентами, и закрытый ключ `domain.key`, который следует хранить в безопасном месте, исключив возможность постороннего доступа к нему.

Теперь необходимо скопировать сертификат в каждую систему демона Docker, которому потребуется доступ к реестру. Копирование должно быть выполнено в файл

```
/etc/docker/certs.d/<адрес_реестра>/ca.crt
```

где `<адрес_реестра>` – это адрес (имя) и номер порта конкретного сервера реестра. Также потребуется перезапустить демон Docker. Например

```
~$ sudo mkdir -p /etc/docker/certs.d/reginald:5000
~$ sudo cp registry_certs/domain.crt /etc/docker/certs.d/reginald:5000/ca.crt
~$ sudo service docker restart
```

Теперь можно запустить реестр

```
$ docker run -d -p 5000:5000 -v $(pwd)/registry_certs:/certs \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
--restart=always --name registry registry:2
```

Здесь подкаталог `registry_certs` файловой системы хоста монтируется на каталог `/certs` внутри контейнера. Таким образом из-под каталога `/certs` внутри контейнера будут доступны `*.crt` и `*.key`.

Выполним операции извлечения образа, замены его тега и возврата (выгрузки) обратно в реестр, чтобы убедиться в работоспособности нового реестра

```
root@reginald:~$ docker pull debian:wheezy
root@reginald:~$ docker tag debian:wheezy reginald:5000/debian:local
root@reginald:~$ docker push reginald:5000/debian:local
```

Итак, мы получили реестр с возможностью удаленного доступа к нему с обеспечением безопасной работы и безопасного хранения образов. При тестировании реестра с удаленных компьютеров не забудьте скопировать сертификаты в файл `/etc/docker/certs.d/<адрес_реестра>/ca.crt` на компьютерах с работающими механизмами Docker.

Хранилище По умолчанию образ реестра использует драйвер файловой системы, который вполне ожидаемо сохраняет все данные и образы в соответствующей файловой системе.

1.11. Сокращение размера образа

Образ формируется из нескольких уровней, причем каждый уровень создается отдельной командой из соответствующего файла `Dockerfile` и его родительских файлов `Dockerfile`. Общий конечный размер образа представляет собой сумму размеров всех его уровней.

Если файл удаляется на том же уровне, на котором он создается, то такой файл не включается в образ. Поэтому часто встречаются файлы `Dockerfile`, которые загружают tar-архивы или архивы

других форматов, распаковывают их и сразу же удаляют архивный файл в одной инструкции RUN. Например, в официальный образ MongoDB включена следующая инструкция

```
RUN curl -SL "https://${MONGO_VERSION}.tgz" -o mongo.tgz \
  && curl -SL "https://${MONGO_VERSION}.tgz.sig" -o mongo.tgz.sig \
  && gpg --verify mongo.tgz.sig && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
  && rm mongo.tgz*
```

2. Базовые концепции, связанные с системой Docker

2.1. Структура стека протоколов TCP/IP

Сегодня стек протоколов TCP/IP используется как в глобальных, так и в локальных сетях. Стек имеет иерархическую, четырехуровневую структуру (см табл. 1).

Прикладной уровень стека TCP/IP соответствует трем верхним уровням модели OSI: прикладному, представления и сеансовому [3].

Таблица 1. Иерархическая структура стека протоколов TCP/IP

Прикладной уровень	FTP, Telnet, HTTP, SMTP, SNMP, TFTP
Транспортный уровень	TCP, UDP
Сетевой уровень	IP, ICMP, RIP, OSPF
Уровень сетевых интерфейсов	не регламентируется

Протоколы прикладного уровня развертываются на хостах.

Транспортный уровень стека TCP/IP может предоставлять вышележащему уровню два типа сервиса:

- гарантированную доставку обеспечивает *протокол управления передачей* (Transmission Control Protocol, TCP),
- доставку по возможности, или с максимальными усилиями, обеспечивает *протокол пользовательских дейтаграмм* (User Datagram Protocol, UDP).

Чтобы обеспечить надежную доставку данных, протокол TCP предусматривает установление *логического соединения*. Это позволяет нумеровать пакеты, подтверждать их прием квитанциями, организовать в случае потери повторные передачи, распознавать и уничтожать дубликаты, доставлять прикладному уровню пакеты в том порядке, в котором они были отправлены. Благодаря этому протоколу объекты на *хосте-отправителе* и *хосте-получателе* могут поддерживать обмен данными в дуплексном режиме. TCP дает возможность без ошибок доставить сформированный на одном из компьютеров поток байтов на любой другой компьютер, входящий в составную сеть.

Протокол UDP является простейшим *дейтаграммным* протоколом, используемым, если задача надежного обмена данными либо вообще не ставится, либо решается средствами более высокого уровня – прикладным уровнем или пользовательским приложением.

В функции протоколов TCP и UDP входит также исполнение роли связующего звена между прилегающими к транспортному уровню прикладным и сетевым уровням. От прикладного протокола (например, от HTTP) транспортный уровень принимает задание на передачу данных с тем или иным качеством прикладному уровню-получателю.

Сетевой уровень, называемый также уровнем Интернета, является стреем всей архитектуры TCP/IP. Протоколы сетевого уровня поддерживают интерфейс с вышележащим транспортным

уровнем, получая от него запросы на передачу данных по составной сети, а также с нижележащим уровнем сетевых интерфейсов.

Основным протоколом сетевого уровня является межсетевой протокол (Internet Protocol, IP). В его задачу входит продвижение пакета между сетями – от одного маршрутизатора к другому до тех пор, пока пакет не попадет в сеть назначения. В отличие от протоколов прикладного уровня и транспортного уровней, протокол IP разворачивается не только на хостах, но и на всех маршрутизаторах. Протокол IP – это дейтаграммный протокол, работающий без установления соединения по принципу доставки с максимальными усилиями. Такой тип сетевого сервиса называют также «ненадежным».

2.2. Формат IP-адреса

В заголовке IP-пакета предусмотрены поля для хранения *IP-адреса отправителя* и *IP-адреса получателя*. Каждое из этих полей имеет фиксированную длину 4 байта (32 бита).

IP-адрес состоит из двух логических частей – номера сети и номера узла в сети. Наиболее распространенная форма представления IP-адреса – запись в виде четырех чисел, представляющих значения каждого байта в десятичной форме и разделенных точками, например: 128.10.2.30.

Границу в IP-адресе между номером сети и номером узла в сети можно найти с помощью *маски*. Маска – это число, применяемое в паре с IP-адресом, причем двоичная запись маски содержит непрерывную последовательность единиц в тех разрядах, которые должны в IP-адресе интерпретироваться как номер сети. Граница между последовательностями единиц и нулей в маске соответствует границе между номером сети и номером узла в IP-адресе.

2.3. Виртуальный сетевой интерфейс

Все TCP/IP-реализации поддерживают loopback-механизмы, которые реализуют виртуальный сетевой интерфейс исключительно программно и не связаны с каким-либо оборудованием, но при этом полностью интегрированы во внутреннюю сетевую инфраструктуру компьютерной системы. Пожалуй самый распространенным IP-адресом в механизмах loopback является 127.0.0.1. В IPv4 в него также отображается любой адрес из диапазона 127.0.0.0 – 127.255.255.255. IPv6 определяет единственный адрес для этой функции – 0:0:0:0:0:0:0:1/128 (так же записывается как ::1/128). Стандартное, официально зарезервированное доменное имя для этих адресов – `localhost`.

Интерфейс loopback имеет несколько путей применения. Он может быть использован сетевым клиентским программным обеспечением, чтобы общаться с серверным приложением, расположенным на том компьютере. То есть если на компьютере, на котором запущен веб-сервер, указать в веб-браузере URL `http://127.0.0.1/` или `http://localhost/`, то он попадает на веб-сайт этого компьютера.

3. Пример создания простого web-приложения

Структура проекта [1, стр. 99]

```
indentidock/  
  -- Dockerfile  
  -- app/  
    -- indentidock.py
```



```
-- cmd.sh
-- docker-compose.yml
```

Python-приложение будет выглядеть так

app/identidock.py

```
from flask import Flask, Response, request
import requests
import hashlib
import redis

app = Flask(__name__)
cache = redis.StrictRedis(host='redis', port=6379, db=0)
salt = 'UNIQUE_SALT'
default_name = 'Leor Finkelberg'

@app.route('/', methods=['GET', 'POST'])
def mainpage():
    name = default_name
    if request.method == 'POST':
        name = request.form['name']

    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method='POST'>
        Hell <input type='text' name='name' value='{0}'>
        <input type='submit' value='submit'>
    </form>
    <p>You look like a:
    <img src='/monster/{1}'/>
    '''.format(name, name_hash)
    footer = '</body></html>'

    return header + body + footer

@app.route('/monster/<name>')
def get_identicon(name):
    image = cache.get(name)
    if image is None:
        print('Cache miss', flush=True)
        r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
        image = r.content
        cache.set(name, image)

    return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug = True, port = 5000)
```

bash-сценарий управляет режимами запуска приложения

./cmd.sh

```
#!/bin/bash

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
```



```

    exit 0  # << для того чтобы прекратить выполнение программы!
}

# будет реагировать на прерывание
trap exit_on_signal_SIGINT SIGINT

if [[ "$ENV" == 'DEV' ]]; then
    echo 'Running Development Server'
    exec python 'identidock.py'
elif [[ "$ENV" == 'UNIT' ]]; then
    echo 'Running Unit Tests'
    exec python 'test.py'  # файл test.py должен размещаться в app/
else
    echo 'Running Production Server'
    ...
fi

```

./Dockerfile

```

FROM python:3.5

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==1.1.1 gunicorn==20.0.0 requests==2.22.0 redis==3.5.0
WORKDIR /app
COPY app /app
COPY cmd.sh /

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"]

```

Запустить приложение в режиме тестирования можно так

```

$ docker build -t identidock .  # создать образ на основе Dockerfile
$ docker run -e ENV=UNIT identidock

```

./docker-compose.yml

```

identidock:
  build: .
  ports:
    - "5000:5000"
  environment:
    ENV: DEV
  volumes:
    - "./app:/app"
  links:
    - dnmonster
    - redis

dnmonster:
  image: amouat/dnmonster:1.0

redis:
  image: redis:alpine3.12

```

4. Пример работы с базой данных Redis

Redis – *резидентная* система управления базами данных⁷ класса NoSQL, работающая со структурами данных типа «ключ-значение».

Рассмотрим пример взаимодействия с СУБД Redis на базе концепции Docker-контейнеров. Для начала скачаем образ Redis

```
$ docker pull redis
$ docker images
# выведет
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
redis	latest	36304d3b4540	9 days ago	104MB

Для того чтобы подготовить «основу» для работы с СУБД Redis, запустим контейнер на базе образа Redis в фоновом режиме и присвоим этому образу имя «myredis»

```
$ docker run -d --name myredis redis
$ docker ps -a # 0671bf41b781180da2c6341875fae93d484f8df3f54bb25569220f83936fd032
# выведет
```

CONTAINER ID	IMAGE	STATUS	... PORTS	NAMES
0671bf41b781	redis	Up 3 seconds	... 6379/tcp	myredis

Теперь для того чтобы получить доступ к контейнеру **myredis** нужно создать новый временный контейнер (контейнер, который будет уничтожен после завершения его процесса⁸), который будет выступать в роли интерфейса к СУБД Redis. С помощью этого контейнера мы сможем записать данные в базу, ассоциированную с контейнером **myredis**

```
# обязательно нужно указать /bin/bash
$ docker run -it --rm --link myredis:redis redis /bin/bash
root@5cab704887ff:/data# redis-cli -h redis -p 6379
redis:6379> set "Ansys" "MAPDL" # OK
redis:6379> set "Nastran" "Python" # OK
...
redis:6379> save # OK
redis:6379> exit
root@5cab704887ff:/data# exit
```

Здесь: флаг **-it** означает создать интерактивный сеанс работы на подключаемом терминальном устройстве; ключ **--rm** означает удалить контейнер как только завершится его процесс; ключ **--link myredis:redis** означает создать связь между текущим контейнером и контейнером с именем **myredis** (**redis** – это просто псевдоним, через который можно обращаться к контейнеру **myredis** из-под текущего контейнера).

С помощью терминального клиента **redis-cli** подключаемся к СУБД Redis, указывая хост и порт, и создаем несколько пар «ключ-значение». Затем сохраняем наработки и завершаем сеанс.

Остается только перенести базу данных из контейнера на локальную машину. Для этого запускаем еще один контейнер. Теперь на базе образа какой-нибудь UNIX-подобной операционной системы, например, Ubuntu и копируем файл базы данных **dump.rdb** в каталог файловой системы хоста **backup_host**

```
$ docker run -it --rm --volumes-from myredis -v $(pwd)/backup_host:/backup_cont ubuntu
```

В контейнере будет создан каталог **backup_cont**, который связан с каталогом файловой системы хоста **backup_host**. Говорят, что каталог **backup_host** файловой системы хоста смонтирован

⁷Организованная в оперативной памяти

⁸Контейнер существует пока существует его основной процесс [1]

на каталог `backup_cont` внутри контейнера. Файлы, который попадают в каталог `backup_cont` внутри контейнера оказываются в каталоге `backup_host`. Ключ `--volumes-from` позволяет получить доступ к каталогу базы данных Redis.

Если теперь вывести содержание корневой директории контейнера, то среди прочих каталогов и файлов мы увидим каталог `data` и `backup_cont`

```
root@3cc5e757c42f:/# ls -l
# выведет
total 52
drwxr-xr-x  2 root root   80 Jun  7 20:18 backup_cont
lrwxrwxrwx  1 root root    7 Apr 23 11:06 bin -> usr/bin
drwxr-xr-x  2 root root 4096 Apr 15 11:09 boot
drwxr-xr-x  2 999 999 4096 Jun  7 22:42 data
drwxr-xr-x  5 root root  360 Jun  7 22:54 dev
drwxr-xr-x  1 root root 4096 Jun  7 22:54 etc
drwxr-xr-x  2 root root 4096 Apr 15 11:09 home
...
```

Скопируем файл базы данных в `backup_cont`

```
root@3cc5e757c42f:/# cp -ip data/dump.rdb backup_cont/
```

После копирования этот файл можно будет найти в каталоге `backup_host`. Когда работа с контейнером Redis будет закончена, его можно остановить и удалить

```
$ docker stop myredis
$ docker rm -v myredis # '-v' означает удалить контейнер вместе с томами
```

Вспомогательные контейнеры можно удалить с помощью команды

```
$ docker rm $(docker ps -aq)
```

5. Наиболее полезные конструкции

5.1. Манипуляции с контейнерами

Запустить контейнер с именем `leorcont`, создав сеанс интерактивной работы (`-i`) на подключаемом терминальном устройстве (`-t`) `tty`, и вызывать командную оболочку `bash` из-под ОС Ubuntu Linux

```
docker run -it --name leorcont ubuntu bash
```

Запустить контейнер, а после остановки удалить сам контейнер и созданную на время его существования файловую систему

```
docker run --rm -it ubuntu bash
```

Перезапустить остановленный контейнер

```
docker start quizzical_wright
```

5.2. Информация о контейнере

Получить информацию о контейнере

```
docker inspect quizzical_wright
```

Вывести информацию о контейнере с использованием утилиты `grep`

```
docker inspect quizzical_wright | grep SandboxID
```

Вывести информацию о контейнере с использованием шаблона языка Go <https://metanit.com/go/web/2.2.php>

```
docker inspect --format {{.NetworkSettings.SandboxID}} quizzical_wright
```

Вывести список файлов в работающем контейнере. Для контейнеров `Docker` использует файловую систему `UnionFS`, которая позволяет монтировать несколько файловых систем в общую иерархию, которая выглядит как *единая файловая система*. Файловая система конкретного образа смонтирована как уровень *только для чтения*, а любые изменения в работающем контейнере происходят на уровне с разрешенной записью, монтируемого поверх основной файловой системы образа. Поэтому `Docker` при поиске изменений в работающей системе должен рассматривать только самый верхний уровень, на котором возможна запись [1]

```
docker diff quizzical_wright
```

Вывести список работающих контейнеров

```
docker ps
```

Вывести список всех контейнеров, включая остановленные (stopped)⁹. Такие контейнеры могут быть перезапущены с помощью `docker start`

```
docker ps -a
```

5.3. Удаление контейнеров и образов

Удалить контейнер

```
docker rm quizzical_wright
```

Удалить несколько остановленных контейнеров можно следующим способом. Значение флагов: `-a` (все контейнеры), `-q` (вывести только числовой идентификатор контейнера), `-f` (фильтр), `-v` (все тома, на которые не ссылаются какие-либо другие контейнеры)

```
docker rm -v $(docker ps -aq -f status=exited)
```

Удалить все образы

```
docker rmi $(docker images | sed '1d' | awk -F ' ' '{ print $3 }')
```

6. Docker для ML

Данных не должно быть в docker-образе. Данные должны подгружаться при запуске контейнера. Docker-образ можно запускать с помощью `Airflow`. Оркестратор `Airflow` скачает docker-образ и запустит его. Можно настроить запуск по расписанию с помощью `Kubernetes`.

⁹Формально их называют контейнерами, из которых был совершен выход (exited containers)

Список литературы

1. *Моуэт Э.* Использование Docker. – М.: ДМК Пресс, 2017. – 354 с.
2. *Роббинс А. Bash.* Карманный справочник системного администратора, 2-е изд.: Пер. с англ. – СПб.: ООО «Альфа-книга», 2017. – 152 с.
3. *Олифер В., Олифер Н.* Компьютерные сети. Принципы, технологии, протоколы. – СПб.: Питер, 2020. – 1008 с.