GeekBrains

Dockerfiles и слои

Контейнеризация

Урок 4

Оглавление

Введение

Всем привет! На этом уроке мы познакомимся с docker-файлами: научимся создавать собственные образы, запускать из них контейнеры, публиковать образы. Также изучим ряд новых понятий. В общем, будет интересно! :)

На этом уроке

1. Понятие “Слои” в образах и контейнерах
2. Архитектура образа
3. Взаимосвязь размера образа и его компонентов
4. Best practice создания образов

Вступление

Помимо возможности использования готовых образов в Docker предусмотрен вариант создания собственных с целью дальнейшего использования, однако, не стоит торопиться с созданием чего-то собственного. Велика вероятность того, что необходимое вам приложение уже имеется в репозитории и его можно запускать, прочитав необходимую документацию по использованию.

Сегодня мы научимся создавать собственные образы и запускать из них контейнеры. Также разберемся в общей архитектуре образов и изучим синтаксис создания докерфайлов.

Образы Docker являются некоторым результатом процесса их сборки. Контейнеры же - это образы, которые запущены пользователем с целью выполнения каких-либо действий. Помимо этих двух компонентов, также имеются и Dockerfile, которые являются инструкцией по сборке образов, на основе которых, в дальнейшем, будут выполняться контейнеры.

Dockerfile

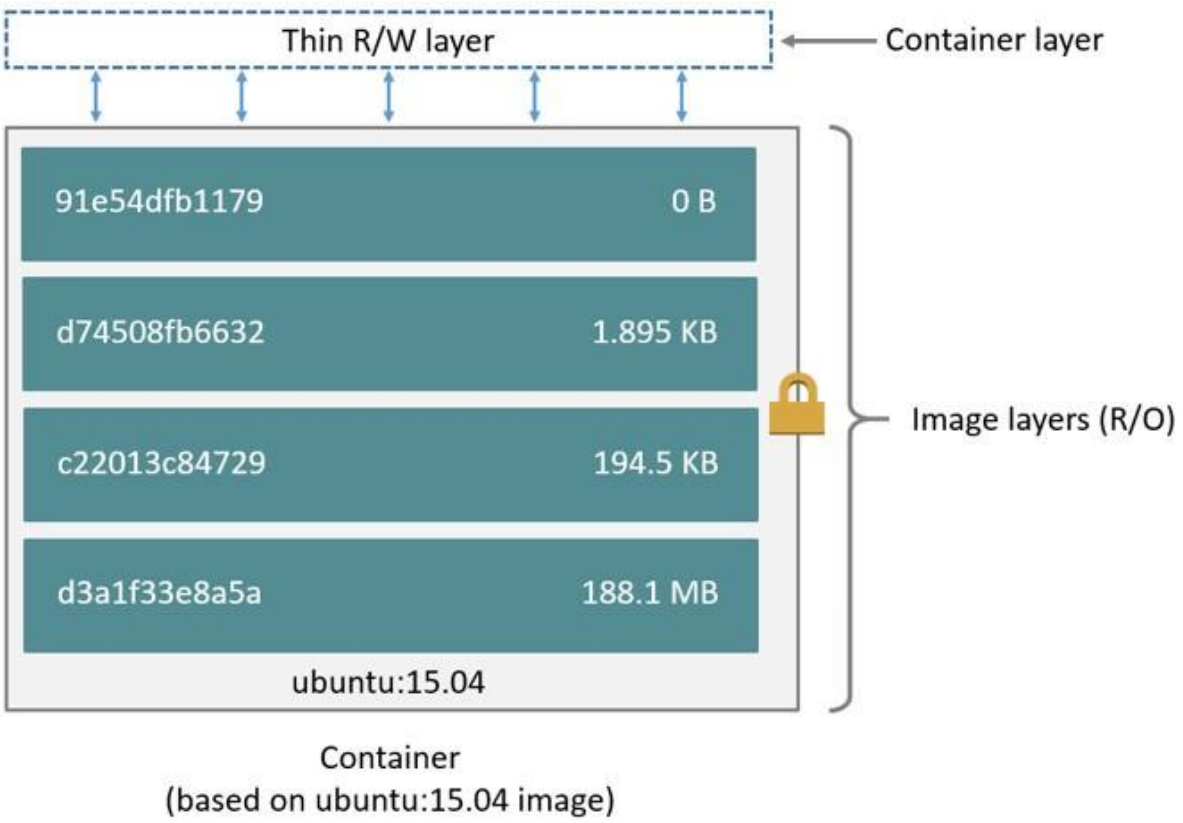
Каждому образу соответствует свой собственный Dockerfile. Название полностью соответствует произношению (никаких расширений файла).

Как уже упоминалось ранее, контейнеры и образы состоят из слоев. Каждый отдельный слой, кроме верхнего (он же - последний), имеет режим доступа “только для чтения”. Также в Dockerfile описывается не только информация о том, что должно быть в образе, но и в какой именно последовательности это “что-то” добавлять.

Стоит обратить ваше внимание на то, что один и тот же образ можно собрать разными способами. А еще точнее так: образы при различных параметрах сборки будут различными технически, но, при этом, выполнять будут одну и ту же функцию. Что именно использовать и как - это выбор каждого.

Когда образ загружается из любого удаленного репозитория локально, физически происходит загрузка лишь слоев, которых локально еще нет. Это сделано с целью экономии дискового пространства и сетевого трафика.

Подойдем немного иначе: каждый слой - это лишь файл, в котором описано очередное изменение состояния образа по сравнению с тем, которое было на предыдущей итерации.



Таким образом выглядит самый обычный образ. Как можно видеть, основой образа выступает ОС ubuntu, а далее идут остальные инструкции, которые образуют слой. Также стоит отметить, что все слои, кроме последнего, подключаются к образу в режиме “только чтение”. Верхний (он же - последний) - доступен в режиме “чтение-запись”. Понятнее станет в процессе разбора инструкций и, как обычно, на практике.

Также хочется затронуть еще одну особенность слоев. Образы необходимо создавать так, чтобы изменения, которые вы, возможно, планируете вносить, содержались на верхних слоях. В чем идея: в самом начале предлагается вставлять аргументы, которые в дальнейшем не планируется изменять, либо изменения необходимы крайне редко. Это позволит сэкономить немного времени в процессе пересборки контейнера. На практике мы обратим дополнительное внимание на эту особенность. В случае, если слой не менялся, образ просто не будет пересобирать этот слой. Чем выше измененный слой, тем меньше будет дополнительных пересборок.

Для чего это важно? Чем выше располагается измененный слой, тем меньше слоев система будет пересобирать при подготовке образа.

Прежде чем мы начнем сопоставлять теорию и практику, предлагается описать каждую команду (аргумент, если хотите), которая может быть использована при

создании образа. Их не так уж и много, поэтому дадим определение каждой, чтобы в дальнейшем использовать их на практике:

- FROM. Эта инструкция задает родительский (базовый) образ, на основе которого будет формироваться наш собственный.
- LABEL - метка. С ее помощью можно описать метаданные образа. Пример: сведения о том, кто создал данный образ и занимается его поддержкой. Также можно указать контакты при необходимости.
- ENV - с помощью этой команды можно указать постоянные переменные для среды. С ее помощью можно задавать глобальные переменные, которые будут использоваться в образе и, впоследствии, в запущенном контейнере.
- ARG - эта инструкция также задает переменные в образе, однако с ее помощью можно задать переменные во время сборки образа.
- RUN - с помощью этого аргумента можно выполнить команду Linux, создав при этом очередной слой.
- COPY - с ее помощью можно скопировать внутрь нашего образа файлы и папки, необходимые для работы.
- ADD - она выполняет похожую процедуру: с ее помощью также можно скопировать файлы и папки, а еще распаковать архивы - tar-файлы.
- WORKDIR - таким образом можно задать рабочую директорию для последующих инструкций, описанных в Dockerfile.
- CMD - с помощью этого аргумента можно описать команду, которая будет выполняться при запуске контейнера. Эти аргументы можно переопределить при запуске. Об этом моменте мы поговорим дополнительно на практике, так как это излюбленный вопрос на технических собеседованиях.
- ENTRYPOINT - предоставляет команду с набором аргументов для вызова во время выполнения контейнера. Эти аргументы нельзя переопределить. На практике мы повторно рассмотрим этот момент и все станет намного понятнее.
- EXPOSE - об этой инструкции нам уже известно и мы рассматривали ее на предыдущем уроке и семинарах. Она указывает на необходимость открытия порта.
- VOLUME - позволяет создать точку монтирования для работы с каким-либо внешним хранилищем.

Теперь, зная определения инструкциям, можно разобраться с каждой в отдельности, дав необходимые пояснения.

Инструкция FROM

Итак, мы начинаем работу с образом с создания базового слоя. Каждый докерфайл должен начинаться либо с инструкции FROM, либо с инструкции ARG. Следом за ARG должна идти инструкция FROM. Это правила создания докерфайлов.

Ключевое слово FROM сообщает системе о том, что необходимо начинать сборку нашего проекта с базового образа, указанного в самом начале докерфайла. Пример выглядит следующим образом:

```
FROM ubuntu:22.10
```

Хотелось бы в этом месте обратить ваше внимание на описание базового образа. Мы будем использовать образ ubuntu с тегом 22.10. Это то, о чем мы и говорили на прошлом уроке: необходимо указывать конкретные версии в процессе работы, чтобы наверняка ничего не ломалось и работало так, как мы хотим. Именно образ, указанной версии будет использоваться в процессе создания образа. Опять же, если никакой тег не включен, то тогда система будет исходить из предположения, что нам необходима последняя версия образа.

Если на локальной машине отсутствует образ, необходимый для сборки, система автоматически выкачает слои, необходимые для работы образа.

Инструкция LABEL

Довольно простая инструкция, которая позволяет добавлять в образ метаданные разного рода: информацию о создателе, контакты и так далее. Важный момент: объявление меток в докерфайле не замедляет процесс сборки образа и не увеличивает его размер. Добавление метки сулит лишь добавление полезной информации об образе Docker.

Пример использования инструкции:

```
LABEL maintainer="hack-me@list.ru"
```

Инструкция ENV

Эта инструкция позволяет создавать постоянные переменные среды, которые будут доступны в контейнере во все время его выполнения.

Простой пример использования выглядит следующим образом:

```
ENV ADMIN="mazzahaker"
```

Эта инструкция замечательно подходит при необходимости использования констант внутри контейнера. Например, можно использовать эти переменные, если у нас, при описании команд, выполняющихся в контейнере, многоразово используется та или иная переменная. Ее стоит использовать, во-первых, с точки зрения удобства, во-вторых, с точки зрения переиспользования кода. Рано или поздно, возможно, эту переменную придется изменить. Куда удобнее изменить значение одной переменной, чем много раз одно и то же в различных участках кода.

Инструкция RUN

Она позволяет выполнять Linux-команду во время сборки образа. После выполнения этой инструкции, в образ добавляется слой и его состояние фиксируется. Простейший пример использования выглядит следующим образом:

```
RUN apt update

RUN apt upgrade

RUN apt install openssh
```

Выполнение этой инструкции говорит системе, что на базовый образ необходимо найти обновленные пакеты для дальнейшего использования и установить при этом обновленный openssh.

Инструкция COPY

Она необходима для копирования файлов и папок из локальной подсистемы в рабочую директорию образа. Важный момент: если вдруг директория, в которую мы копируем данные, отсутствует, то инструкция ее создаст.

Пример использования инструкции выглядит следующим образом:

```
COPY . ./test123
```

Важное пояснение: копирование будет производиться из текущей папки, в которой находится докерфайл. При этом скопировано будет все, что имеется внутри папки.

Инструкция ADD

Эта инструкция позволяет решать те же задачи, что и предыдущая инструкция. Но с ней связаны еще несколько вариантов исполнения. С этой функцией, например, можно добавлять файлы, скачанные с удаленных источников и, как ранее говорилось, распаковывать локальные .tar-файлы.

Пример использования выглядит следующим образом:

```
ADD
https://github.com/mazzahaker/test-lesson-jenkins/blob/main/pipel
ine /testdir123
```

В этом варианте использования, инструкция скачивала файл, расположенный удаленно и добавляла его в директорию внутри контейнера.

Тут стоит отметить, что использовать эту инструкцию вы можете и, формально, вам никто и ничего не запрещает, однако, официальная документация гласит об обратном. В ней настоятельно рекомендуется вместо инструкции ADD использовать инструкцию COPY. Сделано это для того, чтобы докерфайлы были понятнее и читабельнее.

Инструкция WORKDIR

Она позволяет изменять рабочую директорию контейнера. С этой директорией работают следующие ранее описанные функции: COPY, ADD, RUN, CMD, ENTRYPOINT. При использовании этих инструкций также есть ряд особенностей:

- Рекомендация: в аргументах инструкции лучше указывать абсолютные пути к папкам, а не относительные. И уж тем более не перемещаться по файловой системе с помощью команд cd в DOCKERFILE.
- В случае отсутствия директории, указанной в инструкции, система автоматически создаст ее.
- При необходимости в одном докерфайле можно использовать несколько инструкций WORKDIR.

Инструкция ARG

Эта инструкция позволяет задавать переменную, значение которой передается из командной строки в образ во время его сборки. Также можно указать и значение

по умолчанию, которое будет меняться, если указывать в аргументах командной строки соответствующее значение.

Пример использования:

```
ARG test_var=script.py
```

Важное отличие ARG от ENV состоит в том, что переменные, переданные в ARG, недоступны во время выполнения контейнера, но их, при этом, можно использовать для задания значений по умолчанию для переменных ENV (переопределять) в процессе сборки образа.

Инструкция CMD

Эта инструкция предоставляет Docker команду, которую необходимо выполнить непосредственно при запуске контейнера. Важное замечание: результаты выполнения данной команды не добавляются в образ во время его сборки.

Также стоит отметить, что в одном докерфайле может присутствовать только одна инструкция CMD. Если вдруг в файле будет содержаться несколько таких инструкций, система проигнорирует все кроме последней.

Также эту инструкцию принято располагать в самом конце файла, что логично, так как ею и заканчивается описание нашего докерфайла. По сути, заканчиваем его мы инструкцией по запуску нашего ПО.

Еще одно интересное замечание: аргументы, введенные в командной строке, которые передаются при запуске **docker run**, переопределяют все аргументы запуска, которые предоставляются в докерфайле.

Пример использования:

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

Инструкция ENTRYPOINT

Она же позволяет задавать команду вместе с аргументами, которую необходимо выполнить сразу при запуске контейнера. По сути, она похожа на CMD, однако, параметры, задаваемые в entrypoint, нельзя заменить какими-либо параметрами командной строки. То есть в любом случае будет выполнена команда, записанная в докерфайл.

Итак давайте внесем немного ясности - когда же использовать CMD, а когда - ENTRYPOINT:

- Если при каждом запуске контейнера необходимо выполнять одну и ту же команду раз за разом, то необходимо использовать ENTRYPOINT.
- В случае, если контейнер используется в роли приложения (запуск БД, к примеру), также необходимо использовать ENTRYPOINT.
- Если мы знаем, что во время запуска контейнера необходимо передавать ему аргументы, которые могут перезаписывать аргументы по умолчанию (изменять параметры запуска, к примеру), используйте CMD.

Пример использования:

```
ENTRYPOINT ["python", "test_var"]
```

А теперь немного занимательного! С одной стороны, в нашем примере есть ENTRYPOINT. То есть всегда будет запускаться одно и то же. Формально, изменить это нельзя. И это так! При каждом запуске будет выполняться python, и он будет запускать исполняемый файл, заданный в переменной **test_var**. Она у нас фигурировала ранее. И по умолчанию эта переменная имеет значение - имя файла, который содержится локально. Однако, в случае, если мы передадим соответствующее значение для переменной, формально, выполняться в том же контейнере будет уже другой файл.

Инструкция EXPOSE

Использование этой инструкции указывает на необходимость открытия портов для связи с уже работающим контейнером. Она, формально, не открывает порты самостоятельно. Она скорее будет чем-то в роли документации к имеющемуся образу и руководством к действию инженеру, который запускает образ.

Для того, чтобы открыть какой-либо порт (ну или группу портов) и настроить перенаправление портов необходимо как и раньше использовать команду **docker run** с ключом **-p**, который и указывает на необходимость открытия портов у контейнера.

Пример использования:

```
EXPOSE 8080
```

Инструкция VOLUME

Данная инструкция позволяет указывать место в локальной системе, которое контейнер будет использовать для хранения данных.

На практике мы подробнее рассмотрим: как лучше хранить данные при работе с контейнерами и получим большой практический опыт, где будет видна разница хранения данных.

Создание своего первого контейнера

Теперь же, когда мы познакомились со всеми инструкциями и теорией, которая необходима для работы, предлагается закрепить все на практике, создав несколько докерфайлов и, вместе с этим, разобрать еще несколько интересных моментов.

Однако, прежде чем мы начнем, нужно установить в систему одно приложение - cowsay. Это приложение поможет нам довольно просто разобраться с нашим собственным образом и с его помощью на текущий момент будут проходить различного рода эксперименты и сравнения. Приложение просто выводит в консоль изображение коровы с фразой, которую мы ей введем.

Установить его довольно просто:

```
sudo apt install cowsay
```

И давайте его запустим **в системе**, то есть без контейнера:

```
/usr/games/cowsay "GeekBrains"
```

Как можем видеть, приложение установлено и готово к работе. Теперь давайте перейдем непосредственно к созданию нашего докерфайла.

Выглядеть он будет вот так:

```
FROM ubuntu:22.10

RUN apt-get update

RUN apt-get install -y cowsay

RUN ln -s /usr/games/cowsay /usr/bin/cowsay

CMD [“cowsay”]
```

Самая первая строка (FROM...) - это то, что является первым слоем. Он же - исходный слой. Необходимо еще раз запомнить: в терминах Docker - это базовый образ (БО). Его же иногда называют родительским. БО - это то, с чего и начинается сборка образа докера. Его можно выбрать абсолютно любым, пусть для начала будет ubuntu.

Вторая строка приводит к установке новых программных пакетов, обновляя существующие.

Третья - устанавливает приложение **cowsay**.

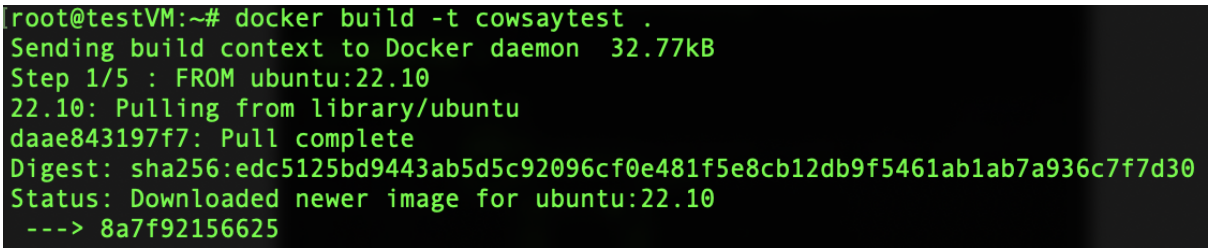
Четвертая - создает символьную ссылку для того, чтобы можно было запускать приложение не через вызов полного пути, а напрямую. Вот так:

Итак, наш докерфайл готов. Давайте “скомпилируем” его! Для этого необходимо перейти в директорию с созданным файлом (показать ls -l) и запустить команду:

```
docker build -t cowsaytest .
```

(Подождать вывода консоли!)

Давайте на этом моменте сделаем паузу и разберем каждый этап.



Здесь система говорит о том, что на первом этапе происходит скачивание базового образа, который до текущего момента отсутствовал локально. Как видим, название образа и его версия полностью соответствуют тому, что мы указывали в Dockerfile.

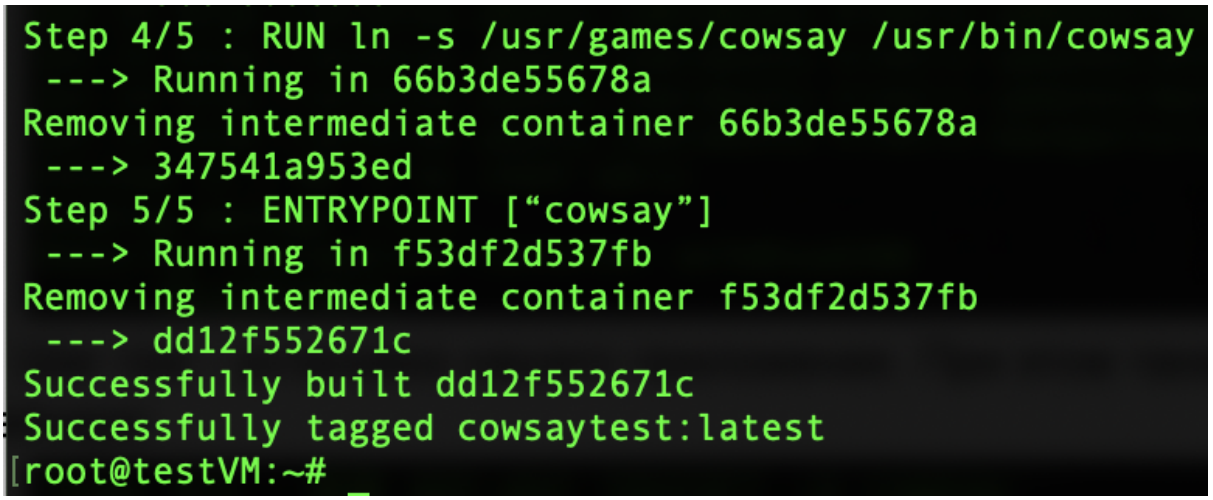
Второй этап. На этом этапе происходит добавление нового слоя к базовому образу - обновление существующих пакетов. Немного позднее мы заострим на этом свое внимание отдельно.

```
Step 2/5 : RUN apt-get update
 ---> Running in ee7b85aa6308
Get:1 http://archive.ubuntu.com/ubuntu kinetic InRelease [267 kB]
Get:2 http://security.ubuntu.com/ubuntu kinetic-security InRelease [109 kB]
Get:3 http://archive.ubuntu.com/ubuntu kinetic-updates InRelease [108 kB]
Get:4 http://security.ubuntu.com/ubuntu kinetic-security/universe amd64 Packages [28.5 kB]
Get:5 http://archive.ubuntu.com/ubuntu kinetic-backports InRelease [99.9 kB]
Get:6 http://security.ubuntu.com/ubuntu kinetic-security/main amd64 Packages [92.4 kB]
Get:7 http://security.ubuntu.com/ubuntu kinetic-security/restricted amd64 Packages [76.3 kB]
Get:8 http://archive.ubuntu.com/ubuntu kinetic/restricted amd64 Packages [151 kB]
Get:9 http://archive.ubuntu.com/ubuntu kinetic/universe amd64 Packages [17.9 MB]
Get:10 http://archive.ubuntu.com/ubuntu kinetic/multiverse amd64 Packages [289 kB]
Get:11 http://archive.ubuntu.com/ubuntu kinetic/main amd64 Packages [1778 kB]
Get:12 http://archive.ubuntu.com/ubuntu kinetic-updates/restricted amd64 Packages [76.3 kB]
Get:13 http://archive.ubuntu.com/ubuntu kinetic-updates/universe amd64 Packages [28.8 kB]
Get:14 http://archive.ubuntu.com/ubuntu kinetic-updates/main amd64 Packages [94.2 kB]
Get:15 http://archive.ubuntu.com/ubuntu kinetic-backports/universe amd64 Packages [1032 B]
Fetched 21.1 MB in 4s (5647 kB/s)
Reading package lists...
Removing intermediate container ee7b85aa6308
 ---> 96cb687f8171
```

Шаг три - установка нашего приложения. При этом также создается новый слой в контейнере:

```
Step 3/5 : RUN apt-get install -y cowsay
 ---> Running in 76653ac1b832
Reading package lists...
Building dependency tree...
Reading state information...
The following additional packages will be installed:
  libgdbm-compat4 libgdbm6 libperl5.34 libtext-charwidth-perl netbase perl
  perl-modules-5.34
Suggested packages:
  filters cowsay-off gdbm-l10n perl-doc libterm-readline-gnu-perl
  | libterm-readline-perl-perl make libtap-harness-archive-perl
The following NEW packages will be installed:
  cowsay libgdbm-compat4 libgdbm6 libperl5.34 libtext-charwidth-perl netbase
  perl perl-modules-5.34
0 upgraded, 8 newly installed, 0 to remove and 0 not upgraded.
```

На четвертом шаге создается символьная ссылка (также с добавлением нового слоя). И, наконец, пятый шаг - добавление точки входа. Это значит, что при вызове контейнера, будет запускаться наше приложение. А аргументы ему передавать мы будем при запуске.



Теперь давайте проверим корректность сборки. Для этого давайте зайдем внутрь контейнера (мы это уже умеем):

```
docker run -it cowsaytest bash
```

И проверим ряд файлов:

1. Что мы находимся внутри контейнера:

```
ps -aux

hostname
```

Как видим, мы внутри контейнера. Давайте попробуем запустить наше приложение вручную (cowsay “GeekBrains in container”). После выполнения команды, все должно запуститься!

Выходим из контейнера в хостовую ОС и запустим наш контейнер из созданного образа:

```
docker run cowsaytest cowsay “GeekBrains”
```

Результат вы можете видеть в консоли. Все запускается и отрабатывает.

Важное замечание! На данном этапе, возможно, вы скажете, что мы все усложняем. Данное приложение можно запустить из системы и все будет замечательно работать. А мы сами себе усложнили жизнь. Но нет! Это приложение лишь показательный пример работы с контейнерами. Оно было выбрано в связи с тем, что его легко контейнеризировать и можно быстро получить результат.

Теперь же давайте посмотрим на наши образы:

```
root@testVM:~# docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
cowsaytest   latest    bd5b2cacc57b   6 seconds ago   152MB
```

Мы видим, что размер нашего текущего образа составляет 152 Мб. Давайте его уменьшать! Проще всего начать с уменьшения количества слоев. Давайте поправим наш докерфайл:

```
FROM ubuntu:22.10
RUN apt-get update && \
    apt-get install -y cowsay && \
    ln -s /usr/games/cowsay /usr/bin/cowsay
RUN rm -rf /var/lib/apt/lists/*
CMD [“cowsay”]
```

Как видим, теперь мы объединили несколько команд, используя символ &&. Это позволило сократить количество слоев с пяти до трех. Вместе с этим мы специально ОТДЕЛЬНО добавили команду очистки скачанных кэшей, которые более нам не нужны. Давайте проверим результат, запустив сборку контейнера:

```
docker build -t cowsaytest .
```

И правда! Теперь система показывает, что у нас 3 шага вместо пяти. Давайте взглянем на размер:

```
root@testVM:~# docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
cowsaytest   latest    7afc9495be91   4 seconds ago   152MB
```

Ого, казалось бы, мы сделали улучшения, мы попытались удалить кэши, но размер не меняется. Тут можно подумать, что все было сделано зря! Однако, не совсем так. Теперь давайте еще раз исправим наш докерфайл:

```
FROM ubuntu:22.10
RUN apt-get update && \
    apt-get install -y cowsay && \
```

```
    ln -s /usr/games/cowsay /usr/bin/cowsay && \
    rm -rf /var/lib/apt/lists/*
CMD [“cowsay”]
```

Теперь мы переместили очистку кэшей на другой слой. Эта команда не создает новый слой, а работает на старом, в котором система и создавала эти кэши. Из-за нахождения в разных слоях ранее, очистка кэшей и не происходила. Это было связано с тем, что каждая из команд работала на разных слоях образа. А теперь давайте проверим!

Пересоберем наш контейнер и посмотрим на размер:

```
docker build -t cowsaytest .
root@testVM:~# docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
cowsaytest   latest    6067ef1e4752   7 seconds ago    118MB
<none>       <none>    7afc9495be91   3 minutes ago    152MB
```

Ура! Как можно заметить, размер изменился! Предыдущий контейнер весил 152 Мб, а наш текущий - 118 Мб. Это прогресс! Однако, все равно чересчур много для простого приложения, которое изначально весит около 5 Мб.

Наш докерфайл идеален либо близок к идеалу. Тут улучшать нечего. Однако, следующий шаг для возможного изменения размера (а точнее - уменьшения) - смена базового образа. Изначально это большая ошибка новичков: мы использовали базовый образ той системы, которая у всех на слуху, однако, для сборки необходимо использовать более легковесные образы. Например, **alpine**. Давайте проверим:

```
FROM alpine
RUN apt-get update && \
    apt-get install -y cowsay && \
    ln -s /usr/games/cowsay /usr/bin/cowsay && \
    rm -rf /var/lib/apt/lists/*
```

```
CMD [“cowsay”]
```

Соберем образ:

```
docker build -t cowsaytest

./bin/sh: apt-get: not found
```

Ага! Вот и снова мы попались! Для каждого базового образа используются различные пакетные менеджеры. Это зависит от дистрибутива операционной системы, которая содержится в БО. Обычно в описании содержится информация о БО. Если нет, ее можно получить самостоятельно. Например, с помощью команды uname. Давайте еще раз поправим наш докерфайл:

```
FROM alpine

RUN apk update && apk add cowsay \

        --update-cache \

--repository
https://alpine.global.ssl.fastly.net/alpine/edge/community \

                                                --repository
https://alpine.global.ssl.fastly.net/alpine/edge/main \

                                                --repository
https://dl-3.alpinelinux.org/alpine/edge/testing && \

        rm -rf /var/cache/apk/*

CMD [“cowsay”]
```

Соберем образ и проверим размер:

```
root@testVM:~# docker images

REPOSITORY      TAG              IMAGE  ID             CREATED
SIZE

cowsaytest      latest       0eb4784817d6      4  seconds  ago
40.9MB
```

Идеально! В нашем примере размер уменьшился в 4 раза.

Заключение

Итак, могу вас поздравить! На этом уроке мы изучили и разобрались в термине “слои”, изучили архитектуру образа и на примерах разобрались с зависимостями вводимых команд и слоями. Также изучили несколько базовых способов по уменьшению размера образа. Изученный материал - часть основных практик по созданию образов. Далее, на семинарах, вы получите больше практики с различными образами и контейнерами. Вместе с этим, изучите на практике набор оставшихся ключей, которые не были охвачены на практической части.

Термины, используемые в лекции

Слои — это результат того, как создаются образы Docker. Каждый этап Dockerfile создает новый «уровень», который, по сути, представляет собой разницу изменений файловой системы с момента последнего этапа.