Введение

В этой статье я хочу немного рассказать про Docker. Здесь я покажу, как начать с ним работать, самые основы. Большая проблема, с которой я сам столкнулся, когда изучал эту технологию, была в огромном количестве избыточной информации. Когда же я разобрался, всё оказалось намного проще.

Эту статью я писал, используя Debian 9 на VirtualBox. Конечно, это не обязательное условие, но докер сам по себе, на мой взгляд, ставит много лишнего в систему, поэтому я предпочитаю держать его в виртуальной машине.

Из чего состоит Docker

Сам по себе докер, это виртуальная машина. И нужно рассматривать его именно как *полноценную* виртуальную машину, в которой есть свой жёсткий диск, оперативная память и сетевая карта. Если вы работали с виртуальными машинами, напримкер, VirtualBox, то многие вещи будут уже знакомы, правда, с той разницей, что докер не имеет красивую GUI-оболочку, а управляется в основном из консоли.

Основные компоненты докера:

* *Образ* (**image**) - образ диска, который будет запускаться в контейнере.
* *Контейнер* (**container**) - виртуальная машина, в которой запущен *образ*.
* *Репозиторий образов* (**image repository**) - удалённое или локальное хранилище для образов.

Нужно помнить, что в контейнере всегда запускается образ. При этом, если докер не нашёл образ в локальном хранилище, то он будет искать его в удалённом репозитории (как правило, на [Dockerhub](https://hub.docker.com/" \t "_blank)).

Как запустить образ

Как бы странно это не казалось, но, если вам хочется уже что-то запустить, то образ даже и собирать-то не надо. Достаточно [установить docker](https://docs.docker.com/install/) и выполнить:

docker run hello-world

Это приведёт к следующим действиям:

1. Докер скачает образ hello-world в локальное хранилище.
2. Создаст новый контейнер.
3. Запустит контейнер, в котором выполнится команда /hello.

Если запустить команду несколько раз, то будет создано несколько контейнеров. Убедиться в этом можно, если выполнить docker ps -a:

% docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9bc9586b6531 hello-world "/hello" 9 seconds ago Exited (0) 9 seconds ago upbeat\_ellis

3db4dc859aaf hello-world "/hello" 11 seconds ago Exited (0) 10 seconds ago suspicious\_chebyshev

db61a2a26199 hello-world "/hello" About a minute ago Exited (0) About a minute ago recursing\_babbage

Чтобы удалить ненужные контейнеры и образы, выполните:

docker container prune -f

docker image prune -fa

Поздравляю! Теперь вы стали почти докер-мастером.

Немного теории

Основная идея докера, это запуск приложений в изолированном пространстве. Виртуальные контейнеры нужны, чтобы окружение одного процесса не мешало другому. Каждый контейнер по умолчанию *изолирован* от других контейнеров и от машины, на которой он работает. Данные внутри контейнера существуют, пока жив сам контейнер. Если контейнер удалить, то удалятся и данные, которые в нём были. При этом образ, из которого был запущен контейнер, никак не изменится.

Именно на идеи изоляции *процесса* построена основная часть логики докера. Контейнер работает, пока работает начальный процесс (его называют *entry point* или *command*). Как только процесс завершается, контейнер останавливается. Это ключевое отличие докера от привычных виртуальных машин (вроде VirtualBox).

Благодаря изоляции процессов, на одной машине можно запускать много версий приложения, которое в обычных условиях будет конфликтовать. Например, можно запустить сразу MySQL 8 и MySQL 5, nodejs 8 и nodejs 10. Без докера сделать это тоже можно, но проблематичней.

Ещё одним важным отличием докера от обычных виртуальных машин является то, что он *не эмулирует* аппаратную часть. Он использует ресурсы системы, но изолирует сам процесс. Поэтому не рекомендуют запускать контейнеры из образов, которые были собраны под платформы, отличные от той, где запущен докер.

Как использовать ресурсы контейнера

По умолчанию контейнер закрыт от любых контактов извне. Вы не можете ни скопировать в него файл, ни подключиться к сокету. Он закрыт. Но при запуске контейнера можно пробросить порт или папку. Тогда к нему можно будет подключиться, добавить файл или что-то скачать. Всё это делается *при создании (запуске)* контейнера через аргументы.

Проброс портов

Для примера запустим контейнер с Debian 9 и пробросим локальный порт 3132 на 80 порт контейнера:

% docker run --tty --detach --entrypoint /bin/cat --name my\_container --publish=3132:80 debian:stretch-slim

1b1d50ead8fff5eecd24029f9f70ae4ad40838d7fe6675e54ba0cd0fe93c83af

% docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1b1d50ead8ff debian:stretch-slim "/bin/cat" 5 seconds ago Up 5 seconds 0.0.0.0:3132->80/tcp my\_container

Пояснения к команде:

* docker run - создаёт и запускает новый контейнер.
* --tty - подключает виртуальную консоль. Это нужно, чтобы команда cat не завершала работу, иначе контейнер остановится.
* --detach - запускает выполнение контейнера в фоне. Без этого аргумента консоль будет ждать, когда контейнер остановится (для остановки придётся использовать другую консоль).
* --entrypoint /bin/cat- использовать в качестве процесса системную утилиту cat. Просто потому, что она не завершится пока не закроется stdin, а значит и контейнер будет работать.
* --name my\_container - уникальное имя, которое используется для управления контейнером. Если его не указывать, то докер сам придумает какое-нибудь имя.
* --publish 3132:80 - проброс портов. Сначала надо указать порт машины (можно указать вместе с IP), потом порт в контейнере.

Общий принцип запуска контейнеров довольно простой:

docker run [аргументы и параметры] название\_образа[:тег]

Аргументы, параметры и тег необязательны, их можно опускать. Но нужно помнить, что без аргументов образ сам по себе не пробрасывает порты и папки. Это всегда делается через аргументы при создании контейнера.

Теперь давайте рассмотрим, как выполнять команды внутри контейнера. Для примера установим и запустим консольный сервер php 7:

*# Обновляем дерево пакетов*

docker exec --tty --interactive my\_container apt update

*# Устанавливаем консольный php*

docker exec --tty --interactive my\_container apt install php7.0-cli

*# Создаём index.php в контейнере*

echo '<?php echo PHP\_VERSION, PHP\_EOL;' | docker exec --interactive my\_container tee index.php

*# Запускаем php-сервер на 80 порту контейнера*

docker exec --tty --interactive my\_container php -S 0.0.0.0:80 index.php

Пояснения к команде:

* docker exec - выполняет команду внутри запущенного контейнера.
* --tty - подключает виртуальную консоль. Без этого аргумента вывод будет неправильным.
* --interactive - подключает ввод. Без него не будет работать клавиатура.
* my\_container - имя контейнера, в котором выполняется команда.
* команда и аргументы - команда, которая будет выполнена внутри контейнера.

Чтобы проверить, работает сервер или нет, нужно подключиться на 3132 порт основной машины, например так:

% curl -s https://127.0.0.1:3132

7.0.33-0+deb9u3

В этом примере я использовал две разные консоли. На одной я запускал сервер, а на другой curl. Ещё можно использовать браузер, если докер установлен у вас в системе.

Проброс папки

В предыдущем примере я создал index.php прямо в контейнере. Это не самый удобный способ разработки проектов в через докер. Во-первых, много файлов так не создашь, во-вторых ими сложно управлять, а, в-третьих, если удалить контейнер, они тоже удалятся. Чтобы решить эти проблемы, можно *пробросить* (*примонтировать*) папку из реальной машины в виртуальный контейнер. Делается это, как всегда, при создании контейнера, через аргумент --volume исходная\_папка:папка\_контейнера.

Прежде, чем начать что-то менять, надо удалить старый контейнер:

docker container rm -f my\_container

Теперь подготовим наш "проект":

mkdir ${HOME}/project

echo '<?php echo "PHP ", PHP\_VERSION, " is the best!\n";' > ${HOME}/project/index.php

А теперь запускаем контейнер с пробросом папки проекта:

docker run --tty --detach --entrypoint /bin/cat --name my\_container --volume ${HOME}/project:/project --publish=3132:80 debian:stretch-slim

Если вы помните, я удалил контейнер, в котором был установлен php, а это значит, что мне заново придётся установить пакет php7.0-cli:

*# Устанавливаем php*

docker exec --tty --interactive my\_container apt update

docker exec --tty --interactive my\_container apt install php7.0-cli

Теперь в контейнере есть и проект, и php, можно запускать сервер:

*# Запускаем php-сервер на 80 порту контейнера*

docker exec --tty --interactive my\_container php -S 0.0.0.0:80 -t /project/

Теперь проверяем, как работает наш проект:

% curl -s https://127.0.0.1:3132

PHP 7.0.33-0+deb9u3 is the best!

Для наглядности давайте создадим ещё один файл в "проекте", чтобы удостовериться, что всё работает как надо:

*# Добавляем файл*

echo '<?php print\_r($\_SERVER);' > ${HOME}/project/test.php

*# Проверяем*

curl -s https://127.0.0.1:3132/test.php

Должно вывести что-то вроде этого:

Array

(

[DOCUMENT\_ROOT] => /project

[REMOTE\_ADDR] => 172.17.0.1

[REMOTE\_PORT] => 38694

[SERVER\_SOFTWARE] => PHP 7.0.33-0+deb9u3 Development Server

[SERVER\_PROTOCOL] => HTTP/1.1

[SERVER\_NAME] => 0.0.0.0

[SERVER\_PORT] => 80

[REQUEST\_URI] => /test.php

[REQUEST\_METHOD] => GET

[SCRIPT\_NAME] => /test.php

[SCRIPT\_FILENAME] => /project/test.php

[PHP\_SELF] => /test.php

[HTTP\_HOST] => 127.0.0.1:3132

[HTTP\_USER\_AGENT] => curl/7.52.1

[HTTP\_ACCEPT] => \*/\*

[REQUEST\_TIME\_FLOAT] => 1557839721.1989

[REQUEST\_TIME] => 1557839721

)

Если у вас получилось, смело пишите в резюме, что владеете докером!

Собираем образ (image)

В предыдущих примерах я два раза установил один и тот же пакет в два разных контейнера. Это, мягко говоря, неудобно. Докер предлагает более гибкое и простое решения - собрать образ с уже установленными пакетами и некоторыми настройками.

Работа с образом не представляет ничего сложного. Нужно создать файл [Dockerfile](https://docs.docker.com/engine/reference/builder/" \t "_blank), указать в нём образ-источник, добавить свои команды и собрать.

Прежде, чем писать файл, я бы хотел определить проблемы, которые нужно решить:

1. Установить пакет php7.0-cli.
2. Сделать точку входа (чтобы не мучить cat)

Указать проброс портов или папки в Dockerfile файле нельзя (несмотря на директивы EXPOSE и VOLUME, проброс и монтирование можно делать только при запуске контейнера). Сделано это специально по причинам безопасности, иначе владелец образа в репозитории мог бы подключать любую системную папку в контейнер без ведома пользователя и красть данные.

Создаём Dockerfile в любой папке со следующим содержимым:

*# Указываем образ-источник*

**FROM** debian:stretch-slim

*# Устанавливаем php + очистка системы*

**RUN** apt-get update; apt-get install -y php7.0-cli; \

apt-get clean && rm -rf /tmp/\* /var/lib/apt/lists/\*

*# Указываем точкой входа консольный сервер php*

**ENTRYPOINT** ["php", "-S", "0.0.0.0:80", "-t", "/project"]

Несколько слов об установке пакетов. Изначально в сборках \*-slim отсутствует дерево пакетов (для уменьшения размера образа), поэтому надо обязательно выполнять apt-get update. После установки желательно удалить ненужные файлы и папки, поэтому выполняются команды очистки.

Небольшое отступление про точку входа. Как я уже говорил, основная специализация докера, это изолирование процесса. Поэтому в одном контейнере запускается всегда один процесс, и точка входа тоже всегда одна. Если нужно запустить вместе несколько процессов (например, php-frm и nginx), то пишется скрипт, который становится точкой входа и запускает нужные приложения. Но объявить несколько точек входа, на данный момент, нельзя.

Теперь нужно собрать образ командой:

docker build -t my\_image:1.0.1 -f Dockerfile ${HOME}/project

Пояснения к команде:

* docker build - запускает сборку образа.
* -t my\_image:1.0.1 - задаёт имя и тэг образа. Если тэг не указан, используется значение lastest.
* -f Dockerfile - явно указывает расположение Dockerfile. Можно опустить, если файл называется Dockerfile и находится в текущей директории.
* ${HOME}/project - рабочая директория для сборки образа. Относительные пути в Dockerfile будут строиться от этой папки.

Готово! Теперь можно запустить контейнер из нашего образа:

*# Сначала удаляем старый контейнер*

docker rm -f my\_container

*# Запускаем новый контейнер*

docker run --detach --name my\_container --volume ${HOME}/project:/project --publish=3132:80 my\_image:1.0.1

Если всё сделано правильно, то список запущенных контейнеров должен быть таким:

% docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

07c72af15d34 my\_image:1.0.1 "php -S 0.0.0.0:80 -…" About a minute ago Up 59 seconds 0.0.0.0:3132->80/tcp my\_container

Ура! Теперь вы гуру :) Конечно, это далеко не все возможности докера, но это то, на чём строятся большие проекты. В основе каждого проекта стоят контейнеры, которые запускаются из образов. Комбинации контейнеров образуют кластеры, которые обеспечивают безотказную работу крупных приложений.

Что дальше

Теперь, когда вы научились создавать образы и поднимать контейнеры, надо понять, как организовать их бесперебойную работу и межконтейнерное взаимодействие.

Попробуйте выполнить следующие шаги:

* Протестировать холодную перезагрузку сервера (лучше в VirtualBox). В реальной жизни это нечастое явление, но случается, поэтому лучше заранее знать, как поведёт себя докер.
* Поднять несколько контейнеров и настроить между ними сеть. Пожалуй, это самая частая задача, с которой вы будете сталкиваться.
* Научиться передавать переменные окружения в контейнер.
* Научиться работать с документацией докера. Конечно, в интернетах есть много статей, но уметь найти нужную информацию в первоисточнике - бесценно.
* Почитать про docker-compose. Рано или поздно это вам точно пригодится.

Шпаргалка по командам докера

# Запустить контейнер с автоматическим удалением (после остановки)

docker run --rm -ti my\_container my\_image

# Войти в командную оболочку в контейнере

docker exec -ti my\_container sh

# Показать список всех контейнеров

docker ps -a

# Показать список образов

docker images

# Удалить нерабочие контейнеры

docker container prune -f

# Удалить лишние образы

docker image prune -fa

# Показать логи контейнера

docker logs my\_container -f