



docker

Docker самый простой и понятный турориал. Изучаем докер, так, если бы он был игровой приставкой



Fomenko Alexander

23 мая 2019 г. • 26 min read



Добро пожаловать в гайд по изучению Docker, в котором я проиллюстрирую вам совершенно иной подход при разработке ваших приложений с его помощью. Эту статью вы можете считать как **быстрый старт, введение в Docker**.

Когда вы полностью прочитаете эту статью, уверен, вы поймёте, что такое Docker, для чего нужен, и где и

Subscribe

Но основным ключом к его освоению - это полное повторение процесса написания кода, как демонстрируется в этой статье. Это **гайд по работе с Docker для новичков**, потому, повторение процесса у себя на компьютере - **обязательное условие к его пониманию**. Одного чтения недостаточно, важно - повторение процесса и много практики. Так же чтобы наконец-то научиться работать с ним, даже при условии плохого понимания Docker-а, начните его уже применять в своей разработке. Начните, и увидите, как стали продвинутым его пользователем.

Когда я наконец-то понял все тонкости работы с Docker (на полное изучение которого ушло несколько месяцев), и начал правильно применять его при разработке (а он как раз и нужен для разработки, в большей степени), то почувствовал, как будто обрёл какую-то сверхспособность. Смотря на свой опыт изучения Докера, я понял, что мне есть что рассказать, и чем поделиться. В этой статье я постарался создать максимально понятную для новичков инструкцию, благодаря которой вы сможете **полностью изучить Docker за 30 минут**. Я долго думал о том, чтобы написать турориал, и наконец-то осилил эту задачу, и, как мне кажется, получилось неплохо :)

Эта инструкция так же подходит для тех, кто не имеет никаких знаний, или опыта работы с докером, или аналогичным программным обеспечением. Вы получите все важные знания, необходимые для работы. Статья построена по принципу от простого к сложному. В итоге статьи вы будете чётко понимать, **что такое Docker**, зачем нужен, как с ним работать, и применять его для разработки: создавать окружение, необходимое для создания вашего приложения.

Эта статья, в большей мере, нацелена на получение практических знаний, и только немного теории, построенной на аналогиях из жизни. Потому, эта статья имеет окрас веселья и лайтовости, в большей мере, чем супер-конкетики и теоретических нюансов.

*В этом турориале я показываю всё на примере ОС Windows 10, делая все команды из консоли винды, и демонстрируя **процесс установки Docker на Windows 10**. Но, все команды будут работать аналогично и на Linux и Mac. Эта статья - это продолжение ряда статей, посвященных настройке рабочего окружения. В прошлой статье мы рассматривали работу с Vagrant, что не менее интересно, чем Docker. И Docker и Vagrant преследуют цель - упростить жизнь разработчикам, но с Докером открывается больше возможностей.*

Так же, прошу заметить, если вы используете Vagrant, и решите установить Docker, то Vagrant перестанет работать. Такая жизнь, но с этим можно смириться, тем более, субъективно, Docker круче ^^.

Что вы узнаете из этой статьи

- Как работает Docker?
- Как установить Docker на Windows?
- Что такое Docker Image (образ)?
- Что такое Docker контейнер?
- Что такое Docker Volumes?
- Что такое Dockerfile?
- Как пробрасывать локальную папку в контейнер Докера (монтирование папки)?

- Как работают и прорасываются Docker порты?
- Слои Docker образа, особенности создания образов.
- Что такое Docker-compose?
- Что такое микросервисы и микросервисная архитектура?

Что такое Docker

О том, как появился Docker:

Docker - это программное обеспечение, которое начинало с того, что зародилось в одной компании, как внутренний проект **platform-as-a-service** в компании **dotCloud**.

В процессе развития Докера, он вырос из масштабов внутреннего проекта, стал доступен для широких масс, и затмил своей популярностью своего родителя **dotCloud**, из-за чего было принято решение создать новую отдельную компанию под названием *Docker Incorporated*. Направление новосозданной компании было только в разработке Докера, и развитию его экосистемы.

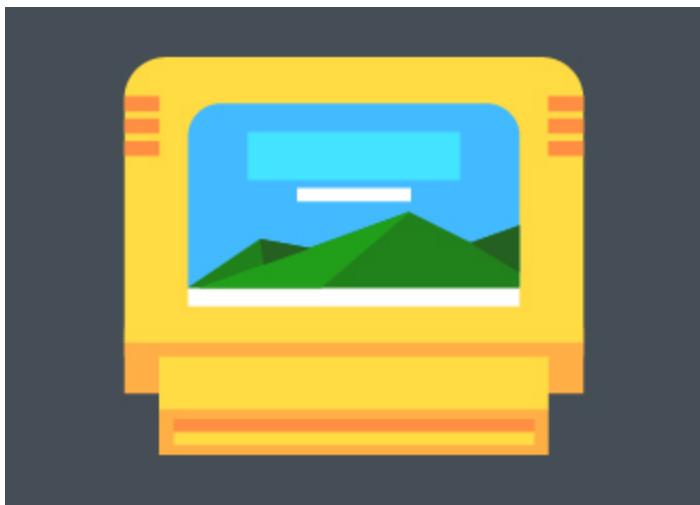
На сайте Докера можно найти статью, в которой подробно рассказывается, **что такое Docker**. Из их слов - это стандартизированное ПО для разработки и развёртывания проектов.

Но, что это на самом деле значит?

Давайте на секунду забудем про Докер, и вспомним про такую ностальгическую штуку, как **GameBoy Color**:



Если вы помните, игры для этой приставки поставлялись в виде картриджей:



И я уверен в том, что производители видео игр пользуются успехом из-за своей простоты:

1. Когда ты хочешь поиграть, ты просто вставляешь картридж в приставку, и игра сразу же работает.
2. Ты можешь поделиться своей игрой с друзьями, передав всего лишь картридж, который они вставлят в приставку, и сразу же смогут играть.

Docker следует похожему принципу - позволяет запускать своё ПО настолько просто, что это соизмеримо с вставкой картриджа и нажатием кнопки *ON* на приставке.

Это основная суть, почему Docker настолько полезен - теперь кто угодно, у кого установлен Docker может запустить ваше приложение, выполнив для этого всего несколько команд.

Раньше, вы, создавая приложения, к примеру на PHP, устанавливали локально *PHP*, *MySQL*, возможно, *NodeJs*, при этом устанавливая зависимости в виде нужных расширений и библиотек. И, в случае передачи вашего скрипта какому-то знакомому, ему требовалось настраивать аналогичное окружение, аналогичных версий, иметь аналогичные расширения и конфигурацию, чтобы успешно запустить ваше приложение.

Сейчас же, при использовании Докера, такой проблемы не возникнет впринципе. Теперь вам достаточно иметь установленную программу Docker, которая по одной вашей команде установит окружение, описанное в конфиге для запуска вашего приложения.

Какое программное обеспечение можно запустить с помощью докера? В техническом плане, Docker чем-то похож на виртуальную машину:

Докер - это движок, который запускает виртуальную операционную систему, имеющую чрезвычайно маленький вес (в отличие от Vagrant-а, который создаёт полноценную виртуальную ОС, Докер, имеет особые образы ПО, запускающиеся в виртуальной среде, не создавая полную копию ОС).

Docker позволяет запустить ОС Linux в изолированной среде очень быстро, в течение нескольких минут.

Зачем использовать Docker?

Кошмар при установке ПО, с которым приходится сталкиваться. У вас когда-нибудь было такое, что вы пытаетесь установить ПО на ваш компьютер, а оно отказывается работать? Вы получаете несколько непонятных вам ошибок, из-за которых ничего не запускается. И после нескольких часов гугления, на десятой странице гугла...и на каком-то форуме, до этого неизвестного вам, вы наконец-то находите случайный комментарий, который помогает исправить вашу проблему.

Аналогично, что делает написание РС игр более сложным, чем написание *Game Boy игр* - это то, что приходится проектировать систему с учётом большого множества существующих РС девайсов и спецификаций. Так как разные компьютеры имеют различные операционные системы, драйвера, процессоры, графические карты, и т.д.

И потому задача разработчика - написать приложение

совместимое со всеми популярными системами, является достаточно затруднительной и трудоёмкой.

Docker спасёт нас. Docker, как и Game Boy приставка, берёт стандартизованные части программного обеспечения и запускает их так, как Game Boy запускал бы игру.

В этом случае вы не должны беспокоиться об операционной системе, на которой пользователь будет запускать ваше приложение. Теперь, когда пользователи будут запускать приложение через Docker - конфигурация будет собрана автоматически, и код будет выполняться ВСЕГДА.

Как разработчик, теперь вы не должны волноваться о том, на какой системе будет запущено ваше приложение.

Как пользователь, вам не нужно волноваться о том, что вы скачаете неподходящую версию ПО (нужного для работы программы). В Докере эта программа будет запущена в аналогичных условиях, при которых это приложение было разработано, потому, исключается факт получить какую-то новую, непредвиденную ошибку.

Для пользователя все действия сводятся к принципу **подключи и играй**.

Установка Docker

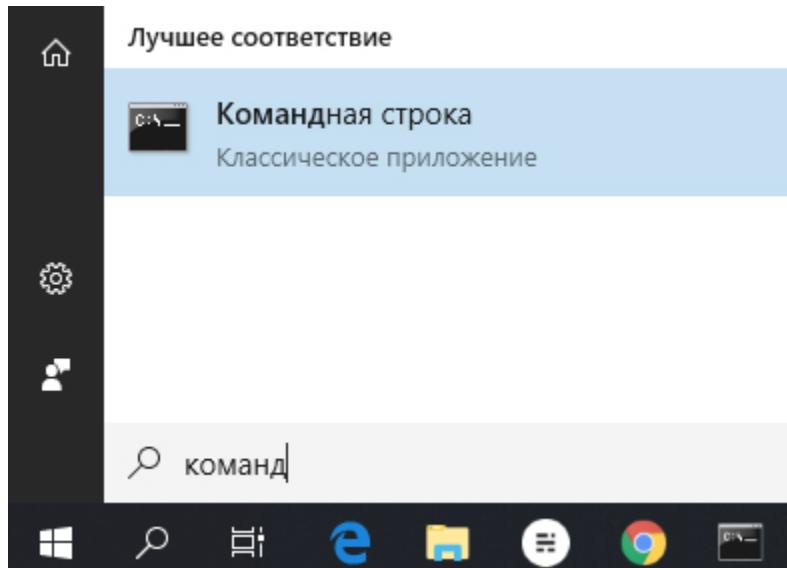
Docker доступен для любой из операционных систем: Windows, Linux, Mac. Для скачивания установочного файла - перейдите по ссылке и выберите подходящую вам версию. Я же, как и писал ранее, выбираю версию docker для Windows 10.

Docker предоставляет 2 сборки:

- **Community Edition** (полностью бесплатная версия)
- **Enterprise Edition** (платно). *Enterprise Edition* содержит в себе дополнительные систелки-перделки функции, которые, на данном этапе, точно не нужны.
Функциональность, которую мы будем использовать совершенно не отличается в этих двух сборках.

После установки потребуется перезагрузка системы, и уже можно начинать полноценно работать с Докером.

Для того, чтобы проверить, запущен ли **Docker**, откроем командную строку (на **Windows 10** - Нажмите кнопку windows, и начните писать **командная строка**)



Где, напишем команду **docker**, и в случае успешно работающего докера, получим ответ

```
Usage: docker [OPTIONS] COMMAND
A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default "C:\\\\Users\\\\Alexa\\\\.docker")
  -D, --debug          Enable debug mode
  -H, --host list      Daemon socket(s) to connect to
  -l, --log-level string    Set the logging level
                            ("debug"|"info"|"warn"|"error"|"fatal")
                            (default "info")
  --tls                Use TLS; implied by --tlsverify
  --tlscacert string  Trust certs signed only by this CA (default "C:\\\\Users\\\\Alexa\\\\.docker\\\\ca.pem")
  --tlscert string    Path to TLS certificate file (default "C:\\\\Users\\\\Alexa\\\\.docker\\\\cert.pem")
  --tlskey string     Path to TLS key file (default "C:\\\\Users\\\\Alexa\\\\.docker\\\\key.pem")
  --tlsverify         Use TLS and verify the remote
  -v, --version        Print version information and quit
```

Дальше, нужно удостовериться, что вместе с докером, доступен так же, **docker-compose**, для этого, выполним команду:

docker-compose (вывод обеих команд будет примерно одинакового содержания).

Если вы используете Linux, то, docker-compose нужно будет устанавливать отдельно по инструкции.

Что такое Docker Image?

Docker образ (он же *Docker Image*), похож на Game Boy картридж - это просто программное обеспечение. Это стандартизированное программное обеспечение, которое запускается на любой приставке Game Boy. Вы можете дать игру вашему другу, и он сможет просто вставить картридж в приставку, и играть.

Как в случае с картриджами, бывают различные игры, так и Docker имеет различные образы ПО: **ubuntu**, **php** (который

наследуется от оригинального образа Ubuntu), `nodejs`, и т.д.

Рассмотрим пример скачивания нашего первого образа.

Для этого, существует команда:

`docker pull <IMAGE_NAME>`, где `<IMAGE_NAME>` - имя скачиваемого образа

Зная эту команду, скачаем образ `Ubuntu 18.10`:

```
docker pull ubuntu:18.10
```

```
C:\Users\Alexa>docker pull ubuntu:18.10
18.10: Pulling from library/ubuntu
5940862bcfcfd: Pull complete
a496d03c4a24: Pull complete
5d5e0cccd5d0c: Pull complete
ba24b170ddf1: Pull complete
Digest: sha256:20b5d52b03712e2ba8819eb53be07612c67bb87560f121cc195af27208da10e0
Status: Downloaded newer image for ubuntu:18.10
```

Эта команда сообщает Докеру о том, что нужно скачать образ `Ubuntu 18.10` с Dockerhub.com - основной репозиторий Docker-образов, на котором вы и можете посмотреть весь их список и подобрать нужный образ для вашей программы.

Это как поездка за новым картриджем в магазин, только намного быстрее :).

Теперь, для того, чтобы посмотреть список всех загруженных образов, нужно выполнить:

```
docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|-------|--------------|--------------|--------|
| ubuntu | 18.10 | 0bfd76efee03 | 20 hours ago | 73.7MB |

У вас, как и на скрине, должен появиться только что скачанный образ **Ubuntu 18.10**.

Как и обсуждалось выше, по поводу маленького размера образов, чистый образ Ubuntu при установке из Docker-образа, весит всего **74 МБ**. Не чудо ли?

Проводя аналогии, команда **docker images** выглядит как коллекция картриджей от приставки, которые у вас есть сейчас:



Что такое Docker контейнер?

Теперь представьте, что мы обновили нашу приставку с **Game Boy** на **GameCube**. Игры хранятся на диске, который предназначен только для чтения самого образа игры. А прочие файлы (сохранения, кеш и т.д.) сохраняются внутри самой приставки, локально.

Так же, как и игра на диске, исходный Docker Image (образ) - неизменяемый.

Docker контейнер - это экземпляр запущенного образа.

Аналогично тому, что вы вставляете диск в приставку, после чего игра начинается.

А сам образ игры никак не модифицируется, все файлы, содержащие изменения хранятся где-то локально на приставке.



Запуск Docker контейнера соответствует тому, что вы играете в свою Gamecube игру. Docker запускает ваш образ в своей среде, аналогично тому, как Gamecube запускает игру с диска, не модифицируя оригинальный образ, а лишь сохраняя изменения и весь прогресс в какой-то песочнице.

Для запуска контейнера существует команда:

```
docker run <image> <опциональная команда, которая выполнится внут
```

Давайте запустим наш первый контейнер Ubuntu:

```
docker run ubuntu:18.10 echo 'hello from ubuntu'
```

```
C:\Users\Alexa>docker run ubuntu:18.10 echo 'hello from ubuntu'
'hello from ubuntu'
```

Команда `echo 'hello from ubuntu'` была выполнена внутри среды *Ubuntu*. Другими словами, эта команда была выполнена в **контейнере *ubuntu:18.10***.

Теперь выполним команду для проверки списка запущенных контейнеров:

```
docker ps
```

| C:\Users\Alexa>docker ps | CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------------------|--------------|-------|---------|---------|--------|-------|-------|
|--------------------------|--------------|-------|---------|---------|--------|-------|-------|

Здесь пустота... это потому что `docker ps` показывает только список контейнеров, которые запущены в данный момент (наш же контейнер выполнил одну команду `echo 'hello from ubuntu'` и завершил свою работу).

А для того, чтобы посмотреть список всех контейнеров без исключения, нужно добавить флаг `-a`, выполним:

```
docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|--------------|--------------------------|---------------|--------------------------|-------|------------------|
| bb597feb7fbe | ubuntu:18.10 | "echo 'hello from ub..." | 5 minutes ago | Exited (0) 5 minutes ago | | relaxed_robinson |

После выполнения нужных операций внутри контейнера, то **Docker-контейнер завершает работу**. Это похоже на режим сохранения энергии в новых игровых консолях - если вы не совершаете действий какое-то время, то система выключается автоматически.

Каждый раз, когда вы будете выполнять команду `docker run`, будет создаваться новый контейнер, на каждую из выполненных команд.

Выполнение неограниченное количество команд внутри контейнера

Давайте добавим немного интерактивности в наше обучение. Мы можем подключиться к консоли виртуальной ОС (`Ubuntu 18.10`), и выполнять любое количество команд без завершения работы контейнера, для этого, запустим команду:

```
docker run -it ubuntu:18.10 /bin/bash
```

Опция `-it` вместе с `/bin/bash` даёт доступ к выполнению команд в терминале внутри контейнера *Ubuntu*.

Теперь, внутри этого контейнера можно выполнять любые команды, применимые к `Ubuntu`. Вы же можете представлять это как мини виртуальную машину, условно, к консоли которой мы подключились по *SSH*.

В результате, теперь мы знаем возможные способы, как подключиться к контейнеру, и **как выполнить команду в контейнере Docker-a**.

Узнаём ID контейнера

Иногда является очень полезным узнать ID контейнера, с которым мы работаем. И как раз-таки, при выполнении команды `docker run -it <IMAGE> /bin/bash`, мы окажемся в терминале, где все команды будут выполняться от имени пользователя `root@<containerid>`.

Теперь, все команды буду выполняться внутри операционной системы `Ubuntu`. Попробуем, например, выполнить команду `ls`, и посмотрим, список директорий, внутри этого образа `Ubuntu`.

```
root@7579c85c8b7e:/# ls  
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

Docker контейнер является полностью независимым от системы хоста, из которой он запускался. Как изолированная виртуальная машина. И в ней вы можете производить любые изменения, которые никак не повлияют на основную операционную систему.

Это аналогично тому, как, если бы вы играли в `Mario Kart` на приставке `Gamecube`, и неважно, что вы делаете в игре, вы никак не сможете изменить само ядро игры, или изменить информацию, записанную на диске.

Контейнер является полностью независимым и изолированным от основной операционной системы, аналогично виртуальной операционной системе. Вы можете вносить любые изменения внутри виртуалки, и никакие из

этих изменений не повлияют на основную операционную систему.

Теперь откройте **новое окно терминала** (не закрывая и не отключаясь от текущего), и выполните команду `docker ps`

| C:\Users\Alexa>docker ps | CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------------------|--------------|--------------|-------------|----------------|---------------|-------|----------------|
| | 7579c85c8b7e | ubuntu:18.10 | "/bin/bash" | 19 minutes ago | Up 19 minutes | | naughty_turing |

Только на этот раз вы можете увидеть, что контейнер с **Ubuntu 18.10** в текущий момент запущен.

Теперь вернёмся назад к первому окну терминала (который находится внутри контейнера), и выполним:

```
mkdir /truedir #создаст папку truedir
exit #выйдет из контейнера, и вернётся в основную ОС
```

Выполнив команду `exit`, контейнер будет остановлен (чтобы убедиться, можете проверить командой `docker ps`). Теперь, вы так же знаете, **как выйти из Docker контейнера**.

Теперь, попробуем ещё раз просмотреть список всех контейнеров, и убедимся, что новый контейнер был создан

| C:\Users\Alexa>docker ps -a | CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS |
|-----------------------------|--------------|--------------|--------------------------|-------------------|------------------------------|
| | 7579c85c8b7e | ubuntu:18.10 | "/bin/bash" | 45 minutes ago | Exited (0) 23 minutes ago |
| | bb597feb7fbe | ubuntu:18.10 | "echo 'hello from ub..." | About an hour ago | Exited (0) About an hour ago |

Так же, для того, чтобы запустить ранее созданный контейнер, можно выполнить команду `docker start <CONTAINER_ID>`,

где **CONTAINER_ID** - id контейнера, который можно посмотреть, выполнив команду `docker ps -a` (и увидеть в столбце **CONTAINER_ID**)

В моём случае, `CONTAINER_ID` последнего контейнера = `7579c85c8b7e` (у вас же, он будет отличаться)

Запустим контейнер командой:

```
docker start 7579c85c8b7e      #ваш CONTAINER_ID
docker ps
docker exec -it 7579c85c8b7e /bin/bash  #ваш CONTAINER_ID
```

И теперь, если внутри контейнера выполнить команду `ls`, то можно увидеть, что ранее созданная папка `truedir` существует в этом контейнере

```
C:\Users\Alexa>docker exec -it 7579c85c8b7e /bin/bash
root@7579c85c8b7e:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  truedir
root@7579c85c8b7e:/#
```

Команда `exec` позволяет выполнить команду внутри запущенного контейнера. В нашем случае, мы выполнили `/bin/bash`, что позволило нам подключиться к терминалу внутри контейнера.

Для выхода, как обычно, выполним `exit`.

Теперь остановим и удалим Docker контейнеры командами:

```
docker stop <CONTAINER_ID>
docker rm <CONTAINER_ID>
```

```
docker ps a  # просмотрим список активных контейнеров
docker stop aa1463167766  # остановим активный контейнер
docker rm aa1463167766    # удалим контейнер
docker rm bb597feb7fbe    # удалим второй контейнер
```

В основном, нам не нужно, чтобы в системе плодилось большое количество контейнеров. Потому, команду `docker run` очень часто запускают с дополнительным флагом `--rm`, который удаляет запущенный контейнер после работы:

```
docker run -it --rm ubuntu:18.10 /bin/bash
```

Что такое DockerFile?

Docker позволяет вам делиться с другими средой, в которой ваш код запускался и помогает в её простом воссоздании на других машинах.

Dockerfile - это обычный конфигурационный файл, описывающий пошаговое создание среды вашего приложения. В этом файле подробно описывается, какие команды будут выполнены, какие образы задействованы, и какие настройки будут применены. А движок Docker-а при запуске уже распарсит этот файл (именуемый как **Dockerfile**), и создаст из него соответствующий образ (*Image*), который был описан. К примеру, если вы разрабатывали приложение на *php7.2*, и использовали *ElasticSearch 9* версии, и сохранили это в *Dockerfile*-е, то другие пользователи, которые запустят образ используя ваш *Dockerfile*, получат ту же среду с *php7.2* и *ElasticSearch 9*.

С *Dockerfile* вы сможете подробно описать инструкцию, по которой будет воссоздано конкретное состояние. И делается это довольно-таки просто и интуитивно понятно.

Представьте, что вы играете в покемонов

Вы пытаетесь пройти первый уровень, но безрезультатно. И я, как ваш друг, хочу с этим помочь. У меня есть 2 таблетки варианта:

1. Я дам вам файл сохранений, в котором игра ничинается со второго уровня. Всё что вам нужно - это загрузить файл.
2. Я могу написать инструкцию, в которой опишу шаг за шагом процесс прохождения уровня. Это как рецепт, которому нужно будет следовать в точности, как описано. Эта инструкция могла бы выглядеть как-то так:

Инструкция прохождения первого уровня

- Выбрать покемона *Squirtle*
- Направляйтесь к лесу, к северу от города
- Тренируйтесь, пока покемон не достигнет 10 уровня
- Направляйтесь в Оловянный город
- Подойдите к боссу, и победите его заклинанием *Watergun*

Что является более полезным? Я склоняюсь, что это второй вариант. Потому что он демонстрирует, как добиться желаемого состояния. Это не просто чёрный ящик, который переносит игру на второй уровень.

С докером вы так же имеете два варианта при создании образа:

1. Вы можете запаковать ваш контейнер, создать из него образ (аналогично тому, что вы записали на диск новую

игру с собственными модификациями). Это похоже на способ, когда вы делитесь сохранениями напрямую.

- Или же, можно описать *Dockerfile* - подробную инструкцию, которая приведёт среду к нужному состоянию.

Я склоняюсь ко второму варианту, потому что он более подробный, гибкий, и редактируемый (вы можете переписать *Dockerfile*, но не можете перемотать состояние образа в случае прямых изменений).

Пришло время попрактиковаться на реальном примере. Для начала, создадим файл `cli.php` в корне проекта с содержимым:

```
<?php
$n = $i = 5;

while ($i--) {
    echo str_repeat(' ', $i).str_repeat('* ', $n - $i)."\n";
}
```

И файл под названием **Dockerfile**, с содержимым:

```
FROM php:7.2-cli
COPY cli.php /cli.php
RUN chmod +x /cli.php
CMD php /cli.php
```

Имена команд в *Dockerfile* (выделенные красным) - это синтаксис разметки *Dockerfile*. Эти команды означают:

- FROM** - это как будто вы выбираете движок для вашей игры (*Unity*, *Unreal*, *CryEngine*). Хоть вы и могли бы начать писать движок с нуля, но больше смысла было бы в использовании готового. Можно было бы

использовать, к примеру, *ubuntu:18.10*, в нашем коде используется образ *php:7.2-cli*, потому весь код будет запускаться внутри образа с предустановленным php 7.2-*cli*.

- **COPY** - Копирует файл с основной системы в контейнер (копируем файл *cli.php* внутрь контейнера, с одноимённым названием)
- **RUN** - Выполнение shell-команды из терминала контейнера (в текущем случае, присвоим права на выполнение скрипта */cli.php*)
- **CMD** - Выполняет эту команду каждый раз, при новом запуске контейнера

Для просмотра полного списка команд можете перейти по ссылке

При написании Dockerfile, начинать следует с наиболее актуального существующего образа, дополняя его в соответствии с потребностями вашего приложения.

*К примеру, мы могли не использовать образ *php:7.2-cli*, а могли взять *ubuntu:18.10*, последовательно выполняя команды в RUN одна за одной, устанавливая нужное ПО. Однако, в этом мало смысла, когда уже есть готовые сборки.*

Для создания образа из *Dockerfile* нужно выполнить:

```
docker build <DOCKERFILE_PATH> --tag <IMAGE_NAME>
```

<DOCKERFILE_PATH> - путь к файлу *Dockerfile* (- текущая директория),

<IMAGE_NAME> - имя, под которым образ будет создан

Выполним:

```
docker build . --tag pyramid
```

При том, что имя файла **Dockerfile** при указывании пути упускается, нужно указывать только директорию, в которой этот файл находится (а **.** означает, что файл находится в той директории, из которой была запущена консоль)

```
C:\projects\docker-example\cli>docker build . --tag pyramid
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM php:7.2-cli
--> 66ee26afcc64
Step 2/4 : COPY cli.php /cli.php
--> Using cache
--> aaf3c2ee31b6
Step 3/4 : RUN chmod +x /cli.php
--> Using cache
--> ec568d0494b8
Step 4/4 : CMD php /cli.php
--> Running in 03a7374e4716
Removing intermediate container 03a7374e4716
--> d168acd128cl
Successfully built d168acd128cl
Successfully tagged pyramid:latest
```

После того, как команда выполнилась, мы можем обращаться к образу по его имени, которое было указано в <IMAGE_NAME>, проверим список образов: **docker images**

```
C:\docker\tutorial\1-pyramid>docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
pyramid        latest       62811c036539   23 minutes ago  73.7MB
```

Теперь, запустим контейнер из нашего образа командой **docker run pyramid**

```
C:\docker\tutorial\1-pyramid>docker run pyramid
*
 ***
 ****
 *****
```

Круто! Shell скрипт был успешно скопирован, и выполнен благодаря указанному в *Dockerfile* параметру **CMD**.

Сначала мы скопировали файл *cli.php* в Docker образ, который создался с помощью *Dockerfile*. Для того, чтобы удостовериться в том, что файл действительно был проброшен внутрь контейнера, можно выполнить команду `docker run pyramid ls`, которая в списке файлов покажет и *cli.php*.

Однако, сейчас этот контейнер недостаточно гибкий. Нам бы хотелось, чтобы можно было удобно изменять количество строк, из скольки состоит пирамида.

Для этого, отредактируем файл *cli.php*, и изменим, чтобы количество аргументов принималось из командной строки. Отредактируем вторую строку на:

```
$n = $i = $argv[1] ?? 5; //а было $n = $i = 5  
// это значит, что мы принимаем аргумент из консоли, а если он не
```

После чего, пересоберём образ: `docker build . --tag pyramid`
И запустим контейнер: `docker run pyramid php /cli.php 9`, получив вывод ёлки пирамиды в 9 строк

```
C:\projects\docker-example\cli>docker run pyramid php /cli.php 9
```

```
*  
 * *  
 * * *  
 * * * *  
 * * * * *  
 * * * * * *  
 * * * * * * *  
 * * * * * * * *
```

Почему это работает?

Когда контейнер запускается, вы можете переопределить команду записанную в *Dockerfile* в поле **CMD**.

Наша оригинальная **CMD** команда, записанная в *Dockerfile* `php`

`/cli.php` - будет переопределена новой `php /cli.php 9`.

Но, было бы неплохо передавать этот аргумент самому контейнеру, вместо переписывания всей команды. Перепишем так, чтобы вместо команды `php /cli.php 7` можно было передавать просто аргумент-число.

Для этого, дополним *Dockerfile*:

```
FROM php:7.2-cli
COPY cli.php /cli.php
RUN chmod +x /cli.php
ENTRYPOINT ["php", "/cli.php"]
## аргумент, который передаётся в командную строку
CMD ["9"]
```

Мы немного поменяли формат записи. В таком случае, **CMD** будет добавлена к тому, что выполнится в **ENTRYPOINT**.

`["php", "/cli.php"]` на самом деле запускается, как `php /cli.php`. И, учитывая то, что **CMD** будет добавлена после выполнения текущей, то итоговая команда будет выглядеть как: `php /cli.php 9` - и пользователь сможет переопределить этот аргумент, передавая его в командную строку, во время запуска контейнера.

Теперь, заново пересоберём образ

```
docker build . --tag pyramid
```

И запустим контейнер с желаемым аргументом

```
docker run pyramid 3
```

```
C:\projects\docker-example\cli>docker run pyramid 3
 *
 *
 *
 * * *
```

Монтирование локальной директории в Docker-контейнер

Монтирование директории в Docker контейнер - это предоставление доступа контейнеру на чтение содержимого вашей папки из основной операционной системы. Помимо чтения из этой папки, так же, контейнер может её изменять, и такая связь является двусторонней: при изменении файлов в основной ОС изменения будут видны в контейнере, и наоборот.

Когда игра читает файлы сохранений, файловая система *Game Cube* внедряет их в текущий сеанс игры (представим это, даже если это не так). Игра может изменять файл сохранений, и это изменение отразится на файловой системе *Game Cube*, т.е. возникает двусторонняя связь.

Монтирование директории в контейнер позволяет ему читать и писать данные в эту директорию, изменяя её состояние.

Для того, чтобы смонтировать папку из основной системы в контейнер, можно воспользоваться командой

```
docker run -v <DIRECTORY>:<CONTAINER_DIRECTORY> ... ,
```

где **DIRECTORY** - это путь к папке, которую нужно смонтировать, **CONTAINER_DIRECTORY** - путь внутри контейнера.

Только путь к монтируемой папке должен быть прописан полностью: `C:\projects\docker-example`, или на *nix-системах можно воспользоваться конструкцией `$(pwd)`

Выполним команду:

```
docker run -it -v C:\projects\docker-example\cli:/mounted ubuntu
ls
ls mounted
touch mounted/testfile
```

При выполнении этой команды, указанная папка смонтируется в папку **/mounted**, внутри файловой системы контейнера, а команда `touch mounted/testfile` создаст новый файл под названием *testfile*, который вы можете увидеть из основной ОС.

Теперь вы можете увидеть, что после выполнения этой команды в текущей директории появился новый файл *testfile*. И это говорит о том, что двусторонняя связь работает - при изменении директории на основной ОС всё отразится на смонтированную папку внутри контейнера, а при изменениях изнутри контейнера всё отразится на основную ОС.

Монтирование папки позволяет вам изменять файлы вашей основной системы прямо во время работы внутри Docker контейнера.

Это удобная особенность, которая позволяет нам редактировать код в редакторе на основной ОС, а изменения

будут сразу же применяться внутри контейнера.

Что такое Docker Volumes?

Docker Volumes - что-то похоже на карты памяти для *Game Cube*. Эта карта памяти содержит данные для игры. Эти карты съемные, и могу работать, когда *Gamecube* приставка выключается. Вы так же можете подключить различные карты памяти, содержащие разные данные, а так же, подключать к разным приставкам.

Вы можете вставить вашу карту внутрь приставки, точно так же, как и Docker Volume может быть прикреплён к любому из контейнеров.

С *Docker Volum-ами* мы имеем контейнер, который хранит постоянные данные где-то на нашем компьютере (это актуально, потому что после завершения работы контейнер удаляет все пользовательские данные, не входящие в образ). Вы можете прикрепить Volume-данные к любому из запущенных контейнеров.

Вместо того, чтобы каждый раз, при запуске контейнера, писать, какие из папок вы хотите смонтировать, вы просто можете создать один контейнер с общими данными, который потом будете прикреплять.

Лично я, не использую это очень часто на практике, потому что есть много других методов по управлению данными. Однако, это может быть очень полезно для контейнеров, которые должны сохранять какие-то важные данные, или данные, которыми нужно поделиться между несколькими контейнерами.

Порты контейнеров

Docker позволяет нам получить доступ к какому-то из портов контейнера, пробросив его наружу (в основную операционную систему). По умолчанию, мы не можем достучаться к каким-либо из портов контейнера. Однако, в *Dockerfile* опция `EXPOSE` позволяет нам объявить, к какому из портов мы можем обратиться из основной ОС.

Для этого, на по-быстрому, запустим Docker-образ *php-apache*, который работает на 80 порту.

Для начала, создадим новую папку `apache` (перейдём в неё `cd apache`), в которой создадим файл `index.php`, на основе которого мы и поймём, что всё работает.

```
<?php  
echo 'Hello from apache. We have PHP version = ' . phpversion()
```

А также, в этой папке создадим файл `Dockerfile`:

```
FROM php:7.2-apache  
# Указываем рабочую папку  
WORKDIR /var/www/html  
# Копируем все файлы проекта в контейнер  
COPY . /var/www/html  
EXPOSE 80
```

Пробежимся по командам:

FROM: это вам уже знакомо, это образ с уже установленным `php` и `apache`

WORKDIR: создаст папку если она не создана, и перейдёт в неё.

Аналогично выполнению команд `mkdir /var/www/html && cd /var/www/html`

EXPOSE: Apache по-умолчанию запускается на 80 порту, попробуем "прокинуть" его в нашу основную ОС (посмотрим как это работает через несколько секунд)

Для работы с сетью в Docker, нужно проделать 2 шага:

- Прокинуть системный порт (*Expose*).
- Привязать порт основной ОС к порту контейнера (выполнить соответствие).

Это что-то похоже на подключение вашей PS4 приставки к телевизору по HDMI кабелю. При подключении кабеля, вы явно указываете, какой HDMI-канал будет отображать видео.

В этой аналогии наша основная ОС будет как телевизор, а контейнер - это игровая консоль. Мы должны явно указать, какой порт основной операционной системы будет соответствовать порту контейнера.

EXPOSE в Докерфайле разрешает подключение к 80 порту контейнера - как разрешение HDMI подключения к PS4.

Выполним первый шаг прокидывания порта. Сбилидим контейнер:

```
docker build . --tag own_php_apache
```

И после этого, запустим контейнер:

```
docker run own_php_apache
```

```
C:\projects\docker-example\apache>docker run own_php_apache
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive global!
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive global!
[Wed May 22 15:27:36.180644 2019] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.25 (Debian) PHP/7.2.18 configured -- resuming normal operation
[Wed May 22 15:27:36.180712 2019] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
```

После чего, попробуем перейти по адресу localhost:80

Но, **это не сработало**, потому что мы ещё не выполнили 2 шаг по маппингу портов.

Выйдите из контейнера, нажав *CTRL+C*.

Если у вас проблемы с остановкой контейнера, в новом окне откройте терминал, выполните `docker ps`, найдите ID контейнера, который сейчас запущен, и выполните `docker stop {CONTAINER_ID}` (указав ваш ID контейнера)

Теперь, осталось сообщить нашему компьютеру, какой порт контейнера ему нужно слушать, и для этого формат записи будет такой:

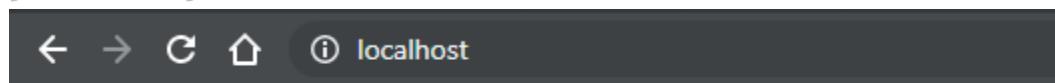
```
docker run -p <HOST_PORT>:<CONTAINER_PORT>
```

И мы можем указать любое соответствие портов, но сейчас просто укажем, что порт системы 80 будет слушать 80 порт контейнера:

```
docker run -p 80:80 own_php_apache
```

Здесь, вы уже наверное заметили, что добавился новый параметр `-p 80:80`, который говорит *Docker*-у: я хочу, чтобы порт 80 из apache был привязан к моему локальному порту 80.

И теперь, если перейти по адресу localhost:80, то должны увидеть успешный ответ:



Оказывается, это даже легче, чем подключение HDMI-кабеля. Сейчас, можем попробовать выполнить запуск на разных портах:

```
docker run -p 8080:80 own_php_apache
```



Теперь, немного подчистим за собой: нужно остановить и удалить контейнеры, которые в данный момент мы запустили:

```
docker ps
docker stop <CONTAINER_ID> ...
docker rm <CONTAINER_ID> ...
```

Для *nix пользователей есть небольшой хак, который позволит остановить и удалить все контейнеры Docker:

```
docker stop $(docker ps -a -q)      # Остановит все контейнеры
docker rm $(docker ps -a -q)        # Удалит все остановленные
```

Запомните, что любой, кто будет запускать этот код на своём компьютере, не должен иметь установленный PHP, всё что ему нужно - один только Docker.

Docker образ: прослойка данных и кеширование

Docker умнее, чем вы могли бы подумать :).

Каждый раз, когда вы собираете образ, он кешируется в отдельный слой. Ввиду того, что образы являются неизменяемыми, их ядро никогда не модифицируются, потому применяется система кеширования, которая нужна для увеличения скорости билдинга.

Каждая команда в Dockerfile сохраняется как отельный слой образа.

Рассмотрим это на примере нашего прошлого Dockerfile-а:

```
FROM php:7.2-apache
# Копирует код ядра
COPY . /var/www/html
WORKDIR /var/www/html
EXPOSE 80
```

Когда вы пишите свой Dockerfile, вы добавляете слои поверх существующего основного образа (указанного в FROM), и создаёте свой собственный образ (Image).

FROM: говорит Докеру взять за основу этот существующий образ. А все новые команды будут добавлены слоями поверх

этого основного образа.

COPY: копирует файлы с основной ОС в образ

WORKDIR: устанавливает текущую папку образа в

`/var/www/html`

Слой Образа Докера это как точка сохранения в игре Super Mario. Если вы хотите изменить какие-то вещи, произошедшие до этой точки сохранения, то вам придётся перезапустить этот уровень полностью. Если вы хотите продолжить прогресс прохождения, вы можете начать с того места, где остановились.

Docker начинает кешировать с "того места, где остановился" во время билдинга Dockerfile. Если в Докерфайле не было никаких изменений с момента последнего билдинга, то образ будет взят полностью из кеша. Если же вы измените какую-то строку в Dockerfile - кеш будет взят только тех слоёв команд, которые находятся выше изменённой команды.

Для иллюстрации этого, добавим новые строки в Dockerfile:

```
FROM php:7.2-apache
WORKDIR /var/www/html
# Copy the app code
COPY . /var/www/html
RUN apt-get update && apt-get upgrade -y && apt-get install -y cu
RUN echo "Hello, Docker Tutorial"
EXPOSE 80
```

После чего, пересоберём образ:

```
docker build . --tag own_php_apache
```

```
C:\projects\docker-example\apache>docker build . --tag own_php_apache
Sending build context to Docker daemon    5.12kB
Step 1/6 : FROM php:7.2-apache
--> ea0b3a98a03e
Step 2/6 : WORKDIR /var/www/html
--> Using cache
--> 20a05485d9c0
Step 3/6 : COPY . /var/www/html
--> 15577df271aa
Step 4/6 : RUN apt-get update && apt-get install -y wget curl
--> Running in 30d0a30cd7e5
Preparing to unpack .../wget_1.18-5+deb9u3_amd64.deb ...
Unpacking wget (1.18-5+deb9u3) ...
Setting up wget (1.18-5+deb9u3) ...
Removing intermediate container 30d0a30cd7e5
--> e9ac05136323
Step 5/6 : RUN echo "Hello, Docker Tutorial"
--> Running in 74fe71c961c1
Hello, Docker Tutorial
Removing intermediate container 74fe71c961c1
--> 323c6c16cf45
Step 6/6 : EXPOSE 80
--> Running in 015cdalf25b6
Removing intermediate container 015cdalf25b6
--> 0656c83815f1
Successfully built 0656c83815f1
Successfully tagged own_php_apache:latest
```

Выполнив эту команду, из вывода в консоль можете увидеть, что некоторые слои были взяты из кеша. Это как раз те команды, выше которых в Dockerfile не было добавлено/изменено содержимого.

И можно заметить, что в случае изменения Dockerfile, билдинг занимает больше времени, потому что не используется кеш. Где бы вы не написали команду, все закешированные команды, которые находятся ниже в Dockerfile, будут перебиложены заново. А те, что находятся выше, будут по-прежнему браться из кеша.

*Когда вы используете команду **COPY**, она копирует указанную директорию в контейнер. И, в случае изменения содержимого любого из файлов этой директории, кеш команды **COPY** будет сброшен. Docker сверяет изменения во время билдинга в каждом из файлов. Если они были изменены, кеш будет сброшен, как и для всех последующих слоёв.*

Если честно, то это действительно крутая функция. Docker следит за изменениями в файлах и использует кеш всегда, когда это нужно (когда были произведены изменения в каких-то из файлов). Изменение ваших файлов потенциально может затрагивать будущие команды, из-за чего, и все последующие слои билдятся заново, а не берутся из кеша.

Какие выводы из этого можно сделать:

1. Команды, которые вероятнее всего не будут меняться в будущем, нужно помещать как можно выше в Dockerfile.
2. Команды копирования данных нужно помещать ниже, потому что файлы при разработке изменяются довольно часто.
3. Команды, которые требуют много времени на билдинг, нужно помещать выше.

В заключение, так же хочу сказать, **как можно уменьшить размер слоёв Docker образов.**

В Dockerfile вы можете иметь несколько команд (RUN) на выполнение:

```
RUN apt-get update  
RUN apt-get install -y wget  
RUN apt-get install -y curl
```

В результате выполнения этой команды, будет создано 3 разных слоя в образе. Вместо этого, все команды стараются объединить в одну строку:

```
RUN apt-get update && apt-get install -y wget curl
```

Если команда становится длинной, и нечитаемой, то для переноса на следующую строку делаем так:

```
RUN apt-get update && apt-get install -y wget curl && \
&& apt-get clean -y \
&& docker-php-ext-install soap mcrypt pdo_mysql zip bcmath
```

*Если же команда становится слишком большой, и неудобной для чтения, то можно создать новый shell скрипт, в который поместить длинную команду, и запускать этот скрипт одной простой командой **RUN**.*

*Технически, только команды **ADD**, **COPY**, и **RUN** создают новый слой в Docker образе, остальные команды кешируются по-другому*

Что такое Docker-Compose?

Docker-compose это как дирижёр оркестра, где ваш оркестр - это набор контейнеров, которыми нужно управлять. Каждый контейнер имеет отдельную задачу, как и музыкальные инструменты в разных частях песни.

Docker Compose управляет контейнерами, запускает их вместе, в нужной последовательности, необходимой для вашего приложения.

Его можно назвать дирижёром в мире Docker-a.

Docker-compose организовывает совместных запуск контейнеров, как инструменты в групповой игре в определённых участках песни.

Каждый инструмент имеет конкретную задачу в группе оркестра. Труба создаёт основную мелодию песни; фортепиано, гитара дополняют основную мелодию; барабаны задают ритм; саксофоны добавляют больше гармонии и т.д.

Каждый из инструментов имеет свою конкретную работу и задачу, как и наши контейнеры.

Docker-compose написан в формате YAML который по своей сути похож на JSON или XML. Но YAML имеет более удобный формат для его чтения, чем вышеперечисленные. В формате YAML имеют значения пробелы и табуляции, именно пробелами отделяются названия параметров от их значений.

Создадим новый файл `docker-compose.yml`, для рассмотрения синтаксиса Docker Compose:

```
version: '3'

services:
  app:
    build:
      context: .
    ports:
      - 8080:80
```

Теперь, построчно разберёмся с заданными параметрами, и что они значат:

version: какая версия docker-compose используется (3 версия - самая последняя на данный момент).

services: контейнеры которые мы хотим запустить.

app: имя сервиса, может быть любым, но желательно, чтобы оно описывало суть этого контейнера.

build: шаги, описывающие процесс билдинга.

context: где находится Dockerfile, из которого будем билдить образ для контейнера.

ports: маппинг портов основной ОС к контейнеру.

Мы можем использовать этот файл для билдинга нашего предыдущего образа apache:

```
docker-compose build
```

После выполнения этой команды, Docker спарсит файл docker-compose и создаст описанные сервисы на основе инструкций во вкладке **build**.

А **context** говорит о том, из какой директории мы берём Dockerfile для создания образа сервиса (в текущем случае - это означает текущую директорию `.`, но могло быть и `/php-cli`, `/nginx`, и т.д.).

И теперь, запустим эти сервисы, которые создали:

```
docker-compose up
```

В результате чего, сервер должен был запуститься, и стать доступным по адресу `localhost:8080`.

Теперь, отключитесь от консоли, нажав **CTRL+C**.

Когда контейнер под названием **app** запускается, docker-compose автоматически связывает указанные порты во вкладке **ports**. Вместо того, как мы делали ранее, выполняя **-v 8080:80**, docker-compose делает это за нас, получив информацию параметра **ports**. Docker-compose избавляет нас боли, связанной с указанием параметров в командной строке напрямую.

С docker-compose.yml мы переносим все параметры, ранее записываемые в командной строке при запуске контейнера в конфигурационный YAML файл.

В этом примере мы записали **BUILD** и **RUN** шаги для нашего сервиса в **docker-compose.yml**. И преимущество такого подхода ещё в том, что теперь для билдинга и для запуска этих сервисов нам нужно запомнить только 2 команды: **docker-compose build** и **docker-compose up**. При таком подходе не нужно помнить, какие аргументы нужно указывать, какие опции нужно задавать при запуске контейнера.

Теперь давайте сделаем нашу разработку немного легче. Сделаем, чтобы вместо постоянного перестроения образа при каждом изменении файлов, мы можем примонтировать нашу рабочую папку в контейнер.

Для этого, удалите строку **COPY . /var/www/html** с **Dockerfile** - теперь все файлы будут прокинуты из основной ОС. Новый **Dockerfile** будет иметь вид:

```
FROM php:7.2-apache  
WORKDIR /var/www/html
```

```
RUN apt-get update && apt-get install -y wget
EXPOSE 80
```

Ранее мы рассматривали, как примонтировать папку в контейнер, для этого мы запускали контейнер с аргументом `-v <HOST_DIRECTORY>:<CONTAINER_DIRECTORY>`. С Docker-compose мы можем указать напрямую в `docker-compose.yml`:

```
version: '3'
services:
  app:
    build:
      context: .
    ports:
      - 8080:80
    volumes:
      - .:/var/www/html
```

Добавленная строка примонтирует текущую директорию основной операционной системы к директории `/var/www/html` контейнера.

В отличие от указания путь в консоли, здесь можно указывать относительный путь (`.`), не обязательно указывать полный путь (`C:\projects\docker-example\apache`), как было ранее при ручном запуске контейнера.

Теперь, выполните по очереди команды:

```
docker-compose stop
docker-compose rm
```

При удалении, вас спросят, действительно ли удалять, напишите `y` и нажмите кнопку enter. Эти команды остановят

и удалят все контейнеры, описанные в файле *docker-compose.yml* (то же самое, как мы ранее запускали `docker stop <CONTAINER_ID>` и `docker rm <CONTAINER_ID>`)

Теперь перебилдим сервисы, потому что мы изменили Dockerfile:

```
docker-compose build
```

И заново запустим:

```
docker-compose up
```

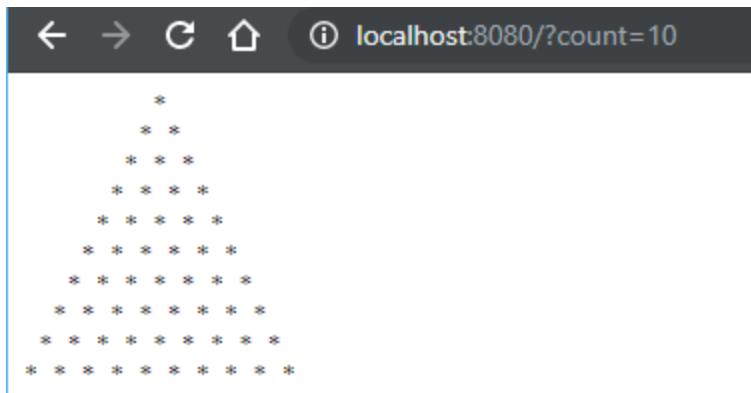
И опять, по адресу `localhost:8080` поднимется наш сервер.

Вместо того, чтобы копировать каждый раз файлы в образ, мы просто примонтировали папку, содержащую исходный код приложения. И теперь, каждый раз не придётся делать ребилд образа, когда файлы изменяются, теперь изменения происходят в лайв режиме, и будут доступны без перестройки образа.

Чтобы в этом убедиться, изменим файл *index.php*, добавим в него скрипт нашей любимой пирамиды:

```
<?php
$n = $i = $_GET['count'] ?? 4;
echo '<pre>';
while ($i--) {
    echo str_repeat(' ', $i).str_repeat('* ', $n - $i)."\n";
}
echo '</pre>';
```

И теперь, если перейти по адресу `localhost:8080?count=10`, то увидим, что пирамида выводится:



```
*  
* *  
* * *  
* * * *  
* * * * *  
* * * * * *  
* * * * * * *  
* * * * * * * *  
* * * * * * * * *
```

Монтирование вашей локальной папки как Docker Volume это основной метод как разрабатывать приложения в контейнере.

Так же, как мы ранее выполняли команды внутри контейнера, указывая аргументы `-it ... /bin/bash`. Docker-compose так же предоставляет интерфейс по удобному выполнению команд внутри конкретного контейнера. Для этого нужно выполнить команду:

```
docker-compose exec {CONTAINER_NAME} {COMMAND}
```

где, вместо `{CONTAINER_NAME}` нужно записать имя контейнера, под которым он записан в сервисах;
а вместо `{COMMAND}` - желаемую команду.

К примеру, эта команда может выглядеть так:

```
docker-compose exec php-cli php -v
```

Но, сделаем это на основе текущего Dockerfile:

```
docker-compose down # остановим контейнеры  
docker-compose up -d # здесь используется опция -d которая сообща  
docker-compose exec app apache2 -v
```

И в результате должны получить

```
C:\projects\docker-example\apache>docker-compose exec app apache2 -v
Server version: Apache/2.4.25 (Debian)
Server built:   2018-11-03T18:46:19
```

Пока что, в `docker-compose.yml` описан только один сервис, потому разворачиваем мы только один контейнер. Но реальный файл `docker-compose` выглядит больше. К примеру, для Laravel он такой:

```
version: '3'
services:
  nginx:
    build:
      context: ./nginx
      dockerfile: docker/nginx.docker
    volumes:
      - ./:/var/www
    ports:
      - "8080:80"
    depends_on:
      - php-fpm
  php-fpm:
    build:
      context: ./php-fpm
      dockerfile: docker/php-fpm.docker
    volumes:
      - ./:/var/www
    depends_on:
      - mysql
      - redis
    environment:
      - "DB_PORT=3306"
      - "DB_HOST=mysql"
      - "REDIS_PORT=6379"
      - "REDIS_HOST=redis"
  php-cli:
    build:
      context: ./php-cli
      dockerfile: docker/php-cli.docker
    volumes:
      - ./:/var/www
    depends_on:
```

```

        - mysql
        - redis

environment:
    - "DB_PORT=3306"
    - "DB_HOST=mysql"
    - "REDIS_PORT=6379"
    - "REDIS_HOST=redis"

tty: true

mysql:
    image: mysql:5.7
    volumes:
        - ./storage/docker/mysql:/var/lib/mysql
environment:
    - "MYSQL_ROOT_PASSWORD=secret"
    - "MYSQL_USER=app"
    - "MYSQL_PASSWORD=secret"
    - "MYSQL_DATABASE=app"
ports:
    - "33061:3306"

redis:
    image: redis:3.0
    ports:
        - "6379:6379"

```

И при выполнении одной только команды `docker-compose up`, поднимутся 5 сервисов. В сравнении с тем, что мы бы вручную выполняли 5 раз команду `docker run ...`. Так что, использование docker-compose в этом случае - очевидно. Так что, теперь, ещё один шаг позади, теперь вы знаете, **что такое docker и docker compose**, и для чего нужен каждый из них.

Как писать Микро Сервисы с Docker? Что такое микросервисы?

Docker наиболее часто используемый инструмент для написания Микросервисов. Микросервисы - это архитектурный шаблон проектирования который следует философии "разделения ответственности".

Ниже я постараюсь кратко описать, что такое микросервисы:
Одиночный миросервис **выполняет одну конкретную задачу**.
Он никак не связан с другими существующими
микросервисами. Вместе же, микросервисы образуют
приложение.

Микросервис должен выполнять свою задачу в изолированной среде, управлять своей собственной локальной информацией, и быть независимым от общей системы настолько, насколько это возможно. В основном, каждый из микросервисов **имеет собственную, отдельную базу данных** (если она нужна). Это позволяет создавать максимально гибкие и легко масштабируемые приложения.

Идея разделения по ответственности предоставляет преимущество в том, что команда разработчиков может работать параллельно, фокусируясь над разными компонентами.

В основном, микросервисы имеют канал коммуникации между собой, в виде REST API, который возвращает данные в JSON, или что-то типа того.

Иногда бывает неясно, насколько большой, или маленькой должна быть задача микросервиса, которую он должен решать. Это решение лежит на плечах разработчика, здесь нет чёткого правила.

Использование микросервисов позволяет быстро масштабироваться под большой нагрузкой, без масштабирования остальных частей вашего приложения. То есть, такая архитектура позволяет вам **масштабировать систему**

компонентно, там, где это требуется. В случае монолитной, единой системы, вам придётся масшабировать всё приложение.

Ввиду того, что статья и так получилась достаточно большой, то пример реализации микросервисной архитектуры я покажу в следующей статье на эту тему. Я принял решение подготовить более качественный материал на эту тему вместе с примерами кода.

Резюме

Я попытался написать эту статью максимально просто, построив всё объяснение на аналогиях и примерах их жизни. Эта статью можно считать **простой инструкцией по работе с Docker**. Помимо того, что я описал, как пользоваться Docker-ом, добавив несколько рабочих примеров, которые вы можете попробовать у себя на компьютере, я добавил много дополнительной информации, и некоторых тонкостей работы с Docker-ом. Надеюсь, что эта статья показала вам, **что такое Docker, и с чем его едят**. В следующей статья я затрону более продвинутые темы и приведу примеры.

Discussion

2 comments

**Bighamster**

48 weeks ago

Спасибо! Отличное введение в докер.

0

L LuchnikKek

Junior Python developer · 19 weeks ago

Статья от 2019 и всё-таки напишу:)

Простой и понятный язык, фактологическая точность и отличное сравнение с приставкой. Читал, тыкал примеры, в результате закрепил знания, узнал много нового.

Единственное, что хочется поправить:

"EXPOSE в Докерфайле разрешает подключение к 80 порту контейнера", и пара предложений вытекающих из этого. Не совсем.

Цитата из доки: "The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.". В общем, Expose только документирует. А флаг -r, о котором речь идёт чуть дальше в этой же главе, как раз пробрасывает.

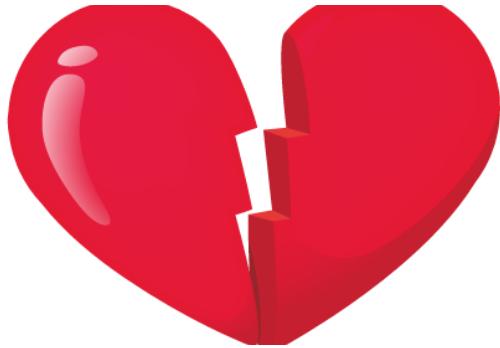
В любом случае, спасибо за Ваш труд.

0

Join the discussion

Become a member of Блог php программиста: статьи по PHP, JavaScript, MySql to start commenting.

[Sign up now](#)[Already a member? Sign in](#)



В серці. Назавжди.

Вчора у мене помер однокласник. А сьогодні бабуся. І хто б міг уявити, що...

20 мая 2022 г. 1 min read 2 comments

Ось такий він, русський мир

"Русський мир" - звучить дуже сильно та виправдовуюче. Гарна обгортка...

16 апр. 2022 г. 3 min read 2 comments

Блог php програмиста: статьи по PHP, JavaScript, MySql © 2024

[Sign up](#) [Блог про кіно](#) [badseller.net - порівняння цін на алкоголь](#)

Powered by Ghost