Публикация на тему

Docker и Docker-compose

Программное обеспечение контейнерной виртуализации Docker предназначено для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему.

Анотация
Использование docker в разработке web-приложений
Used docker in web-development
Автор
Михалькевич Александр Викторович

Публикация

Наименование Docker и Docker-compose
Автор А.В.Михалькевич
Специальность Программное обеспечение контейнерной виртуализации Docker предназначено для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему.,
Анотация
Использование docker в разработке web-приложений
Anotation in English
Used docker in web-development
Ключевые слова docker-compose
Количество символов 59002

Содержание

Введение
1 Что это такое и для чего нужно
2 Docker и виртуальные машины
3 Установка Docker в Linux
    3.1 Установка с помощью apt-get
    3.2 Установка с помощью snap

3.3 Установка с помощью pacman
4 Ключевые концепции
5 Основные компоненты Docker
6 Запуск первого образа
7 Просмотр текущих контейнеров
8 Запуск и остановка контейнера
9 Установка сервера Nginx
10 Докерфайлы и синтаксис для их создания
11 Шпаргалка docker
12 Docker Compose
13 Использование docker-compose в разработке
13.1 MySQL и PHPMyAdmin
13.2 Nginx и PHP
14 Использование Docker в Laravel
Заключение
Список использованных источников
Приложения

Введение

Docker - это программное обеспечение с открытым кодом, принцип работы которого проще всего сравнить с транспортными контейнерами.

1 Что это такое и для чего нужно

Docker это платформа для разработчиков и системных администраторов для развертывания и запуска приложений с помощью контейнеров. Использование контейнеров Linux для развертывания приложений называется контейнеризацией.

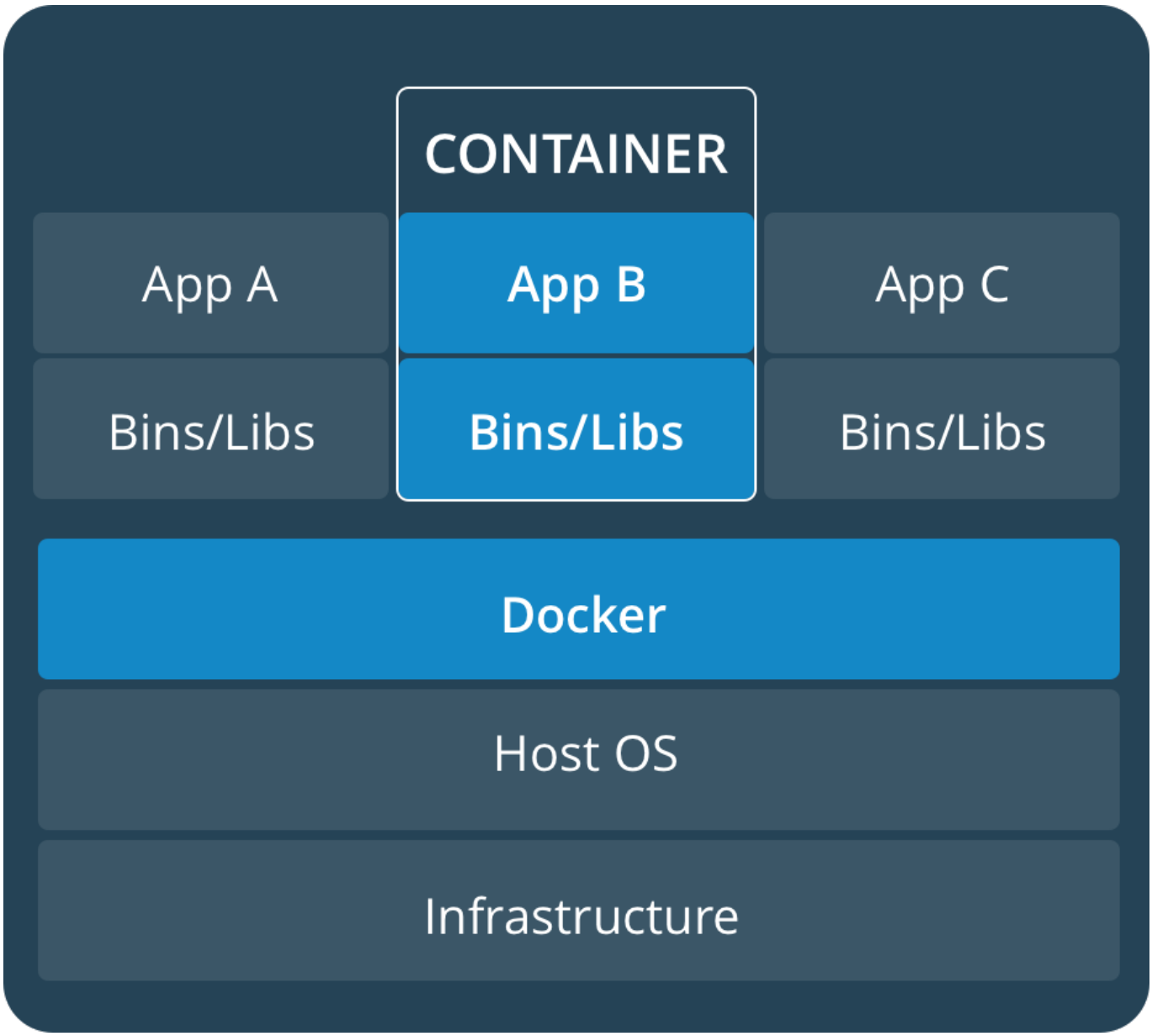
Преимещества конейнеризации:

Гибкость: даже самые сложные приложения могут быть упакованы в контейнеры.
Легкость: используются встроенные возможности ядра операционной системы. Пользователи могут загружать и запускать сложные приложения без возни с конфигурированием.
Взаимозаменяемость: развертывание и обновление контейнеров на лету. Одновременно на одном хосте могут быть запущены десятки контейнеров.
Переносимость: имеется возможность создавать и развертывать взаимозаменяемые локальные и облачные контейнеры. Переносимость контейнеров обеспечивает потенциальную возможность устранения программных ошибок, вызываемыми незначительными изменениями рабочей среды.
Масштабируемость: имеется возможность увеличивать и автоматически публиковать контейнеры в специальных репозиториях.
Наращиваемость: контейнеры поддерживают возможность наращивания сервисов.

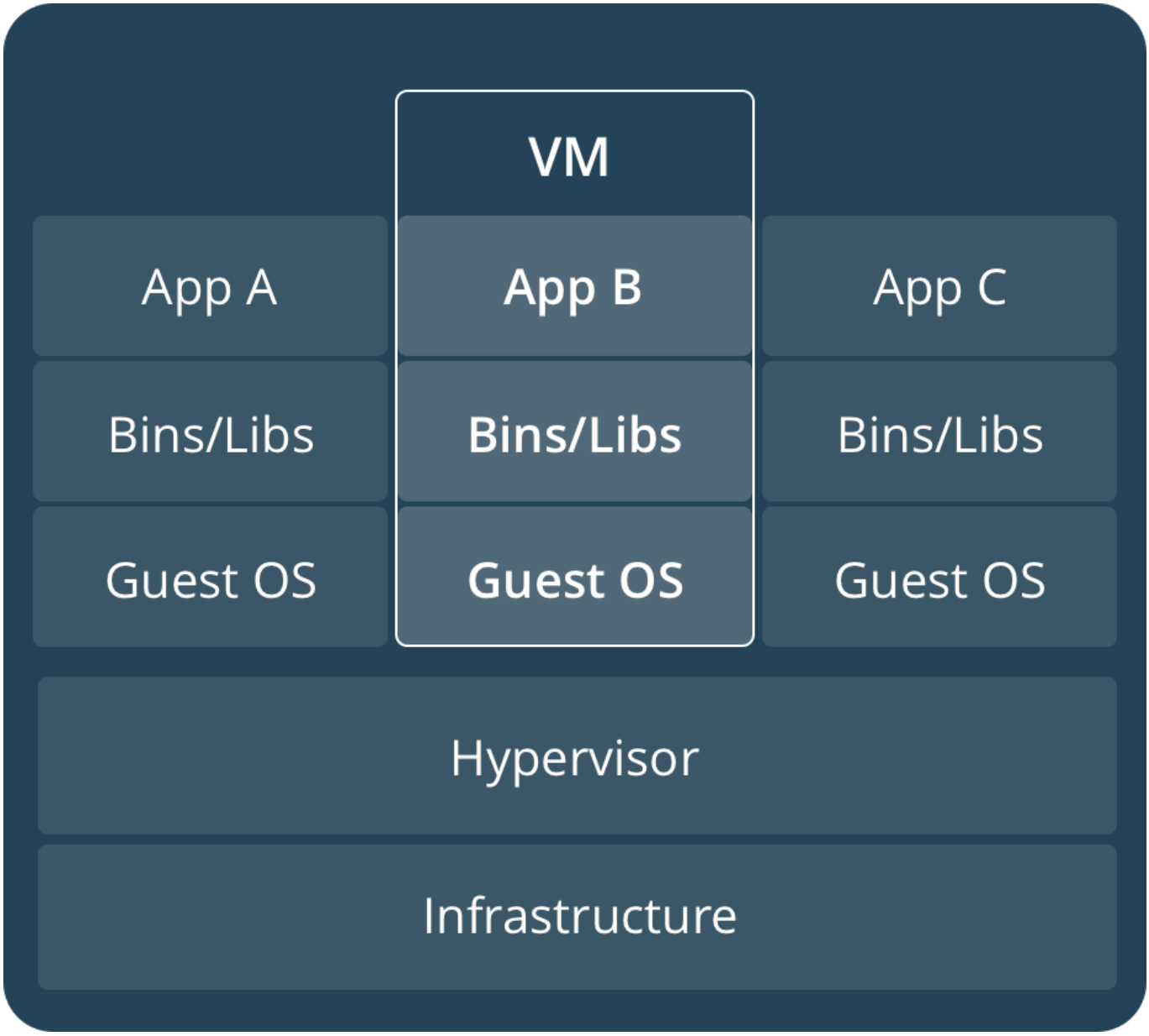
2 Docker и виртуальные машины

Контейнер изначально работает в Linux и разделяет ядро хост-машины с другими контейнерами. Он запускает отдельный процесс, занимая не больше памяти, чем любой другой

исполняемый файл, что делает его легким.



На виртуальной машине работает полноценная «гостевая» операционная система с виртуальным доступом к ресурсам хоста через гипервизор. Виртуальные машины предоставляют среде больше ресурсов, которые в основном уходят на запуск самой "гостевой" операционной системы.



3 Установка Docker в Linux

Установка Docker в Linux достаточна проста, однако мелкие детали установки могут значительно различаться в разных дистрибьютивах Linux. На странице официальной документации Docker - https://docs.docker.com/installation/ можно ознакомиться с различными вариантами установки.
Здесь же мы рассмотрим установку Docker в Ubutnu и ArchLinux

3 .1 Установка с помощью apt-get

Устанавливаем всегда последнюю версию, и информацию по установке берем из официальной документации, по ссылке https://docs.docker.com/install/linux/docker-ce/ubuntu

Процесс установки можно разбить на несколько шагов

1. Обновляем пакеты Ubuntu

```
sudo apt-get update
```

2. Устанавливаем необходимые зависимости

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

3. Добавляем официальный ключ

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

И сравниваем отпечаток:

```
sudo apt-key fingerprint 0EBFCD88
```

4. Добавляем репозиторий где расположен Docker

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

5. Обновляем индекс пакетов apt

```
sudo apt-get update
```

6. Установка Docker

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

7. Убеждаемся, что Docker CE установлен правильно, запустив образ hello-world.

```
sudo docker run hello-world
```

3 .2 Установка с помощью snap

`Snap` - это пакет приложения для Linux систем, который легко устанавливать без дополнительных зависимостей.

Сперва убеждаемся в том, что snap установлен, либо устанавливаем его с помощью `apt`.

```
sudo apt update
sudo apt install snapd
```

После чего устанавливаем Docker

```
sudo snap install docker
```

3 .3 Установка с помощью pacman

Установить Docker можно с помощью различных пакетных менеджеров.

Воспользуемся менеджером пакетов Pacman, который установлен по умолчанию в таких операционных системах как Arch Linux и Manjaro. Сперва обновим pacman.

```
sudo pacman -Syu
```

Далее установим модуль loop

```
sudo tee /etc/modules-load.d/loop.conf <<< "loop"
```

```
modprobe loop
```

Теперь установим Docker с официального репозитория

```
sudo pacman -S docker
```

После установки добавим пользователя в группу Docker

```
sudo usermod -aG docker
```

4 Ключевые концепции

Прежде чем запускать Docker, лучше всего разобраться с понятиями образов, контейнеров, слоёв (набором различий) и методанными Docker.

Лучший способ понять образы и контейнеры - это рассматривать их как програмы. **Образы** - это классы, **контейнеры** - объекты. Точто так же, как объекты представляют собой экземпляры классов, контейнеры являются экземплярами образов. Вы можете создать несколько контейнеров из одного образа, и все они будут изолированы друг от друга, так же, как и объекты. Чтобы вы ни изменили в объекте, это не повлияет на определение класса.

Образы устроены таким образом, что они состоят из **слоев** и **методанных**. Методанные содержат информацию о переменных среды, пробросе портов и других деталей. Файлы образов занимают большую часть пространства. Они содержат ядра систем и копии необходимых инструментов, включая языковые среды и библиотеки. Например, есть такой образ - Ubuntu.

Контейнеры создаются из образов, наследуют свои файловые системы и используют методанные для определения своих конфигураций запуска. Контейнеры являются отдельными, но могут быть связаны друг с другом.

5 Основные компоненты Docker

Dockerfile. Текстовый файл с последовательно расположенными инструкциями для создания образа Docker. Файл создаётся по принципу «одна строка — одна команда».

Daemon. Фоновая служба на хосте, которая отвечает за создание, запуск и уничтожение контейнеров.

Image. Неизменяемый файл (образ), из которого можно неограниченное количество раз развернуть контейнер.

Client. Утилита командной строки в Docker для управления демоном. Любое взаимодействие с контейнером проходит через Daemon.

Container. Запущенное приложение, которое развернули из образа.

Registry. Служба в Docker, выполняющая функции репозитория (хранилища). Позволяет следить за версиями образов, создавать приватные репозитории.

Docker Hub. Популярный публичный репозиторий, используемый по умолчанию в Docker. Обеспечивает интеграцию с GitHub и BitBucket.

Docker Desktop. Приложение, позволяющее локально собирать, выполнять и тестировать контейнеры. Работает на Windows и macOS.

Docker volumes. Тома для постоянного хранения информации. По умолчанию в Docker папки хранилищ создаются на хост-машине, но предусмотрена и возможность подключения удаленных хранилищ. Использование томов позволяет лучшим образом настроить хранение данных.

6 Запуск первого образа

Запуск образа возможен после запуска самого docker. Для запуска образа Hello world воспользуемся следующей командой

```
docker run ubuntu /bin/echo 'Hello world'
```

> docker run – это команда запуска контейнера.
> ubuntu – образ, который вы запускаете (например, образ операционной системы Ubuntu). Когда вы его указываете, Докер сначала анализирует элемент в разрезе хоста.
> /bin/echo 'Hello world' – команда, которая будет запускаться внутри нового контейнера. Данный контейнер просто выводит «Hello world» и останавливает выполнение.

7 Просмотр текущих контейнеров

Давайте посмотрим, какие контейнеры у нас есть на данный момент:

```
docker ps
```

docker ps – команда для перечисления контейнеров.

```
docker ps -a
```

-a показывает все контейнеры (без -a ps покажет только запущенные контейнеры).

8 Запуск и остановка контейнера

Давайте остановим контейнер-демон:

```
docker stop daemon
```

А теперь запустим контейнер:

```
docker start my_container
```

9 Установка сервера Nginx

Начиная с этого примера, вам понадобятся дополнительные файлы, которые вы можете найти в репозитории GitHub. Как вариант, загрузите образцы файлов по ссылке.

Теперь давайте создадим контейнер Nginx.

Измените каталог на тот, куда был скачен образ:

```
docker run -d --name test-nginx -p 80:80 -v $(pwd):/usr/share/nginx/html:ro
nginx:latest
```

-p – отображение портов HOST PORT: CONTAINER PORT.
-v отвечает за HOST DIRECTORY:CONTAINER DIRECTORY.

Теперь проверьте этот URL-адрес в своем веб-браузере.

10 Докерфайлы и синтаксис для их создания

Dockerfile — скрипт, который позволяет автоматизировать процесс построения контейнеров — шаг за шагом, используя при этом **base** образ.

Докерфайл автоматически выполняет определенные действия или команды в base образе, для формирования нового образа.

Все подобные файлы начинаются с обозначения `FROM`, также как и процесс построения нового контейнера, далее следуют различные методы, команды, аргументы или условия, после применения которых получится Docker контейнер.

В Докерфайлах содержится два типа основных блоков: комментарии и команды с аргументами. Причем для всех команд подразумевается определенный порядок.

Рассмотрим типичный пример синтаксиса, где первая строка является комментарием, а вторая - командой.

```
# Print «Hello from Merionet!»
RUN echo «Hello from Merionet!!»
```

Перед тем, как переходить к собственно написанию собственно Докерфайла, сначала разберем все возможные команды.

> Все команды в Докерфайлах принято указывать заглавными буквами — к примеру **RUN, CMD** и т.д.

Команда **ADD** — данная команда берет два аргумента, путь откуда скопировать файл и путь куда скопировать файлы в собственную файловую систему контейнера. Если же source путем является **URL** (т.е адрес веб-страницы) — то вся страница будет скачена и помещена в контейнер.

```
# Синтаксис команды: ADD [исходный путь или URL] [путь назначения]
ADD /my_merionet_app /my_merionet_app
```

Команда **CMD** — довольно таки похожая на команду RUN, используется для выполнения определенных программ, но, в отличие от RUN данная команда обычно применяется для запуска/инициации приложений или команд уже после их установки с помощью RUN в момент построения контейнера.

```
# Синтаксис команды: CMD %приложение%  «аргумент», «аргумент», ..
CMD «echo» «Hello from Merionet!».
```

Команда **ENTRYPOINT** устанавливает конкретное приложение по умолчанию, которое используется каждый раз в момент построения контейнера с помощью образа. К примеру, если вы установили определенное приложение внутри образа и вы собираетесь использовать данный образ только для этого приложения, вы можете указать это с помощью ENTRYPOINT, и каждый раз, после создания контейнера из образа, ваше приложение будет воспринимать команду CMD, к примеру. То есть не будет нужды указывать конкретное приложение, необходимо будет только указать аргументы.

```
#Синтаксис команды: ENTRYPOINT %приложение% «аргумент»
# Учтите, что аргументы опциональны — они могут быть предоставлены
командой CMD или #во время создания контейнера.
ENTRYPOINT echo

#Синтаксис команды совместно с CMD:
CMD «Hello from Merionet!»
ENTRYPOINT echo
```

Команда **ENV** используется для установки переменных среды (одной или многих). Данные переменные выглядят следующим образом «ключ = значение» и они доступны внутри контейнера скриптам и различным приложениям. Данный функционал Докера, по сути, очень сильно увеличивает гибкость в плане различных сценариев запуска приложений.

```
# Синтаксис команды:  ENV %ключ% %значение%
ENV BASH /bin/bash
```

Команда **EXPOSE** используется для привязки определенного порта для реализации сетевой связности между процессом внутри контейнера и внешним миром — хостом.

```
# Синтаксис команды:  EXPOSE %номер_порта%
EXPOSE 8080
```

Команда **FROM** — данную команду можно назвать одной из самых необходимых при создании Докерфайла. Она определяет базовый образ для начала процесса построения контейнера. Это может быть любой образ, в том числе и созданные вами до этого. Если указанный вами образ не найден на хосте, Докер попытается найти и скачать его. Данная команда в Докерфайле всегда должна быть указана первой. # Синтаксис команды: FROM

%название_образа% FROM centos
Команда **MAINTAINER** — данная команда не является исполняемой, и просто определяет значение поля автора образа. Лучше всего ее указывать сразу после команды FROM.

```
# Синтаксис команды:  MAINTAINER %ваше_имя%
MAINTAINER MerionetNetworks
```

Команда **RUN** - является основной командой для исполнения команд при написании Докерфайла. Она берет команду как аргумент и запускает ее из образа. В отличие от CMD данная команда используется для построения образа (можно запустить несколько RUN подряд, в отличие от CMD).

```
# Синтаксис команды:  RUN %имя_команды%
RUN yum install -y wget
```

Команда **USER** — используется для установки UID или имени пользователя, которое будет использоваться в контейнере.

```
# Синтаксис команды:  USER %ID_пользователя%
USER 751
```

Команда **VOLUME** — данная команда используется для организации доступа вашего контейнера к директории на хосте (тоже самое, что и монтирование директории)

```
# Синтаксис команды:  VOLUME [«/dir_1», «/dir2» ...]
VOLUME [«/home»]
```

Команда **WORKDIR** указывает директорию, из которой будет выполняться команда CMD.

```
# Синтаксис команды:  WORKDIR /путь
WORKDIR ~/
```

Создание своего собственного образа для установки MongoDB

Для начала создадим пустой файл и откроем его с помощью **vim**:

```
vim Dockerfile
```

Затем мы можем указать комментариями для чего данный Докерфайл будет использоваться и все такое — это не обязательно, но может быть полезно в дальнейшем. На всякий случай напомню — все комментарии начинаются с символа **#**.

```
########
# Dockerfile to build MongoDB container images
# Based on Ubuntu
########
```

Далее, укажем базовый образ:

```
FROM ubuntu
```

Затем, укажем автора:

```
MAINTAINER Merionet_Translation
```

После чего обновим репозитории(данный шаг совершенно необязателен, учитывая, что мы не будем их использовать) :

```
RUN apt-get update
```

После укажем команды и аргументы для скачивания **MongoDB** (установку проводим в соответствии с гайдом на официальном сайте):

```
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
10gen' | tee /etc/apt/sources.list.d/mongodb.list

RUN apt-get update

RUN apt-get install -y mongodb-10gen

RUN mkdir -p /data/db
```

После чего укажем дефолтный порт для MongoDB:

```
EXPOSE 27017

CMD [«--port 27017»]

ENTRYPOINT usr/bin/mongod
```

Вот как должен выглядеть у вас финальный файл - проверьте и, затем, можно сохранить изменения и закрыть файл:

```
#########
# Dockerfile to build MongoDB container images
# Based on Ubuntu
#########

FROM ubuntu

MAINTAINER Merionet_Translation

RUN apt-get update

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
10gen' | tee /etc/apt/sources.list.d/mongodb.list

RUN apt-get update
```

```
RUN apt-get install -y mongodb-10gen

RUN mkdir -p /data/db


EXPOSE 27017

CMD ["--port 27017"]

ENTRYPOINT usr/bin/mongod
```

Запуск контейнера Docker

Итак, мы готовы создать наш первый MongoDB образ с помощью Docker!

```
sudo docker build -t merionet_mongodb .
```

> -t и имя здесь используется для присваивания тэга образу. Для вывода всех возможных ключей введите
>
> ```
> sudo docker build –help
> ```
>
> , а **точка** в конце означает что Докерфайл находится в той же категории, из которой выполняется команда.

Далее запускаем наш новый MongoDB в контейнере!

```
sudo docker run -name MerionetMongoDB -t -i merionet_mongodb
```

> Ключ -name используется для присвоения простого имени контейнеру, в противном случае это будет довольно длинная цифро-буквенная комбинация. После запуска контейнера для того, чтобы вернуться в систему хоста нажмите **CTRL+P**, а затем **CTRL+Q**.

11 Шпаргалка docker

Создание образа

> `docker pull ОБРАЗ` - загружает образ из Docker Hub (аналог GitHub для Docker)
>
> `docker build ПУТЬ | URL` - создает образ с помощью Dockerfile
>
> **Параметры:**
>
> `-t | --tag=""` - помечает созданный образ переданным названием (и, тэгом, если он будет передан)
>
> `--rm` - Удаляет промежуточные контейнеры после успешной сборки (по умолчанию == true)

Управление образами

> `docker rmi` - Удаляет образ, образ не может быть удален, если существуют

контейнеры (даже незапущенные), которые основаны

на данном образе

Параметры:

`-f` - позволяет удалить образ даже если на нём основаны контейнеры

`docker images` - Отображает список всех существующих образов

Параметры:

`-a` | `--all` - отображает все образы (по умолчанию не отображает промежуточные контейнеры)

`-q` - отображает только id образов, вместо таблицы

Запуск и остановка контейнеров

`docker run ОБРАЗ [КОМАНДА + АРГУМЕНТЫ]` - Запускает выбранный образ в новом контейнере

Параметры:

`-d` | `--detach` - запускает контейнер в фоновом режиме и выводит только id свежесозданного контейнера. (по умолчанию == `false`)

`-i` | `--interactive` - запускает контейнер в интерактивном режиме (оставляет STDIN открытым, даже если контейнер запущен в неприкрепленном режиме)

`-t` | `--tty` - запускает псевдотерминал, часто используется с `-i`

`-p` | `--publish=[]` - пробрасывает порты контейнера в хост. Формат: ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | containerPort

`-e` | `--env=[]` - пробрасывает переменные окружения внутрь контейнера.

`-v` | `--volume=[]` - пробрасывает директорию файловой системы внутрь контейнера

`docker stop КОНТЕЙНЕР` - останавливает контейнер, передавая внутрь SIGTERM, а по истечении таймаута - SIGKILL

`docker start КОНТЕЙНЕР` - запускает остановленный контейнер.

Параметры:

`-i` | `--interactive` - аналогично docker run -i

`docker restart КОНТЕЙНЕР` - Перезапускает выбранный контейнер с помощью docker stop и docker start

`docker kill КОНТЕЙНЕР` - Убивает контейнер, передавая внутрь SIGKILL

Управление контейнерами

`docker port КОНТЕЙНЕР` - отображает маппинг портов между хостом и контейнером

`docker ps` - отображает список запущенных контейнеров

Параметры:

`-a` | `--all=(true|false)` - отображать ли все контейнеры. По умолчанию == `false`, т.е. отображаются только запущенные контейнеры

`-q` - отображает только ID контейнеров вместо таблицы

`docker rm КОНТЕЙНЕР` - удаляет контейнер. По умолчанию можно удалить только запущенный контейнер.

Параметры:

`-f` | `--force=(true|false)` - позволяет удалить запущенный контейнер. Используется передача SIGKILL внутрь.

`docker diff` - отображает изменения относительно образа.

Синтаксис Dockerfile

Dockerfile служит скриптом сборки для команды `docker build`. Перед началом сборки docker передает сборщику всё содержимое папки с Dockerfile'ом,
поэтому располагать его в корневой директории системы будет не лучшей идеей.

Формат файла:

```
# Комментарий
ИНСТРУКЦИЯ аргументы
```

Первая инструкция обязательно должна быть инструкцией FROM.

Инструкции:

`FROM ОБРАЗ` | `FROM ОБРАЗ:ТЭГ` - Задает базовый образ для последующих инструкций. Может встречаться несколько раз в одном Dockerfile,
если необходимо собрать несколько образов за раз.

`MAINTAINER имя` - Позволяет задать поле *Author* сгенерированного образа

`RUN команда` | `RUN ["исполняемый файл", "параметр1", "параметр2", ..]` - Запускает команду на основе текущего образа и фиксирует изменения в новом образе. Новый образ будет использован для исполнения последующих инструкций. Первый синтаксис подразумевает запуск команд в стандартной оболочке (bin\sh -c)

`CMD ["исполняемый файл", "параметр1", "параметр2"]` | `CMD ["параметр1", "параметр2"]` | `CMD команда параметр1 параметр2` - Предоставляет значения по умолчанию для запуска контейнера. Эти значения могут как включать исполняемый файл (варианты 1, 3), так и не включать его (вариант 2). В последнем случае запускаемая команда
должна быть задана с помощью инструкции `ENTRYPOINT`.

`EXPOSE порт <порт...>` - Информирует Docker, что контейнер будет прослушивать указанные порты во время исполнения. Docker может использовать эту информацию, чтобы соединять контейнеры между собой используя связи. `EXPOSE` сам по себе не делает порт доступным из хостовой системы. Для того, чтобы открыть порты в хостовую систему следует использовать флаг `-p`.

`ENV ключ значение` - Позволяет задавать переменные окружения. Эти переменные будут использованы при запуске контейнера из собранного образа. Могут быть просмотрены с помощью команды `docker inspect`, а также переопределены с помощью флага `--env` команды `docker run`.

`ADD ОТКУДА <ОТКУДА...> КУДА` - Используется для добавления новых файлов, директорий или ссылок на удалённые файлы в файловую систему контейнера. Несколько ОТКУДА может быть передано одновременно, но в этом случае все адреса должны быть относительны для директории, из которой происходит сборка. Каждый вхождение ОТКУДА может содержатьодин или несколько символов подстановки, которые будут разрешены с использование функции языка Go filepath.Match. КУДА должен быть абсолютным путем внутри контейнера.

`ENTRYPOINT ["исполняемый файл", "параметр1", "параметр2"]` | `ENTRYPOINT команда параметр1 параметр2` - позволяет сконфигурировать контейнер так, чтобы он запускался как исполняемый файл. В отличии от команды `CMD` значение не будет переопределено аргументами, переданными в команду `docker run`. Таким образом, аргументы из команды `docker run` будут переданы внутрь контейнера, т.е. `docker run ОБРАЗ -d` передаст -d исполняемому файлу.

`VOLUME [ПУТЬ]` - создает точку монтирования с указанным именем и помечает её как содержащую подмонтированные разделы из хостовой системы или других контейнеров. Значение может быть задано как массив JSON, например, `VOLUME ["/var/log/"]`, так и как обычной строкой с одним или несколькими аргументами, например `VOLUME /var/log` или `VOLUME /var/log /var/db`

`USER имя` - позволяет задавать имя пользователя или UID, который будет использован для запуска образа, а так же для любой из инструкций `RUN`

`WORKDIR ПУТЬ` - задает рабочую директорию для команд `RUN`, `CMD` и `ENTRYPOINT`. Инструкция может быть использована несколько раз. Если ПУТЬ относителен, то он будет относительным для ПУТИ, заданным предыдущей инструкцией `WORKDIR`.

Образ - image
Контейнер - container

12 Docker Compose

При работе со сложными приложениями, зависящими от нескольких сервисов, управлять контейнерами и синхронизировать их работу – задача довольно непростая. Docker Compose – это инструмент, разработанный сообществом, который позволяет вам запускать мультиконтейнерные приложения, основанные на определениях из фала YAML. Определения сервисов позволяют собирать гибкие пользовательские среды с большим количеством контейнеров, которые могут совместно использовать сети и тома данных.

Данный мануал поможет установить Docker Compose в Ubuntu 20.04 и научит использовать его.

Требования

Сервер Ubuntu 20.04, настроенный по этому мануалу.
Предустановленная система Docker (инструкции по установке можно найти в разделах 1-2 этого мануала).

1: Установка Docker Compose

Установить Docker Compose можно из официального репозитория Ubuntu, однако тогда вы получите не самую свежую версию, потому лучше установить программу из GitHub-репозитория Docker.

Найдите ссылку на свежий релиз на этой странице. На данный момент это версия 1.26.0.

Следующая команда загрузит эту версию и сохранит исполняемый файл в /usr/local/bin/docker-compose, что сделает его глобально доступным как docker-compose:

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.26.0/docker-compose-$(
uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Теперь установите права и сделайте файл исполняемым:

```
sudo chmod +x /usr/local/bin/docker-compose
```

Запросите версию программы, чтобы убедиться, что установка прошла успешно:

```
docker-compose --version
```

Команда должна вернуть примерно следующее:

```
docker-compose version 1.26.0, build 8a1c60f6
```

Инструмент Docker Compose успешно установлен. Теперь можно настроить файл docker-compose.yml, чтобы инструмент смог запускать контейнеризованные системы.

2: Настройка docker-compose.yml

Чтобы продемонстрировать, как настроить файл docker-compose.yml и работать с Docker Compose, мы создадим тестовую среду для веб-сервера, используя официальный образ Nginx из Docker Hub, публичного реестра Docker. Эта контейнерная среда будет обслуживать один статический HTML-файл.

Создайте в домашнем каталоге новый каталог, а затем перейдите в него:

```
mkdir ~/compose-demo
cd ~/compose-demo
```

В этом каталоге создайте папку приложения, которая будет корневым каталогом для вашей среды Nginx:

```
mkdir app
```

Используя любой удобный текстовый редактор, создайте новый файл index.html в папке app:

```
nano app/index.html
```

В этом файле помещаем html-код.

Теперь создайте файл docker-compose.yml:

```
nano docker-compose.yml
```

Вставьте в файл такие строки:

```
version: '3.7'
```

```
services:
web:
image: nginx:alpine
ports:
- "8000:80"
volumes:
- ./app:/usr/share/nginx/html
```

Файл docker-compose.yml обычно начинается с определения версии. Так Docker Compose узнает, какую версию конфигурации мы используем.

Затем идет блок services, где можно настроить сервисы, которые являются частью этой среды. У нас есть только один сервис, который называется web. Этот сервис использует образ nginx:alpine и устанавливает перенаправление портов с помощью директивы ports. Все запросы на порт 8000 хоста (системы, в которой вы запускаете Docker Compose) будут перенаправлены на порт 80 контейнера web, где будет работать Nginx.

Директива volume создаст общий том между хостом и контейнером. Это поделит локальную папку app с контейнером, а том будет расположен в папке /usr/share/nginx/html внутри контейнера, что затем перезапишет корневой каталог Nginx по умолчанию.

Сохраните и закройте файл.

Мы создали тестовую страницу и файл docker-compose.yml, который запустит контейнерную среду веб-сервера для ее обслуживания.

3: Запуск Docker Compose

Подготовив файл docker-compose.yml, мы можем запустить Docker Compose, чтобы активировать среду. Следующая команда загрузит необходимые образы Docker, создаст контейнер для сервиса web и запустит контейнерную среду в фоновом режиме:

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

Сначала Docker Compose ищет определенный образ в вашей локальной системе, и, если он не может найти его, он загружает его из Docker Hub. Вы увидите такой вывод:

```
Creating network "compose-demo_default" with the default driver
Pulling web (nginx:alpine)...
alpine: Pulling from library/nginx
cbdbe7a5bc2a: Pull complete
10c113fb0c77: Pull complete
9ba64393807b: Pull complete
c829a9c40ab2: Pull complete
61d685417b2f: Pull complete
Digest:
sha256:57254039c6313fe8c53f1acbf15657ec9616a813397b74b063e32443427c5502
Status: Downloaded newer image for nginx:alpine
Creating compose-demo_web_1 ... done
```

Теперь среда работает в фоновом режиме. Чтобы убедиться, что контейнер активен, вы

можете запустить эту команду:

```
docker-compose ps
```

Она покажет вам информацию о запущенных контейнерах и их состоянии, а также о всех перенаправлениях портов, которые происходят в настоящее время:

```
Name                      Command               State          Ports
------------------------------------------------------------------
compose-demo_web_1   /docker-entrypoint.sh ngin ...   Up
0.0.0.0:8000->80/tcp
```

Теперь вы можете получить доступ к тестовому приложению, указав в браузере localhost:8000, если вы запускаете его на локальном компьютере, или your_server_domain_or_IP:8000, если вы запускаете его на удаленном сервере. Вы должны увидеть такую страницу:

```
This is a Docker Compose Demo Page.
This content is being served by an Nginx container.
```

Общий том, который вы настроили в файле docker-compose.yml, синхронизирует файлы из папки app с корневым каталогом контейнера. Если вы внесете какие-либо изменения в файл index.html, они будут автоматически собраны контейнером и отражены в вашем браузере после обновления страницы.

Далее мы расскажем, как управлять вашей контейнерной средой с помощью команд Docker Compose.

4: Команды Docker Compose

Теперь вы знаете, как настроить файл docker-compose.yml и запустить вашу среду. Давайте посмотрим, как использовать другие команды Docker Compose, предназначенные для управления и взаимодействия с вашей контейнерной средой.

Чтобы проверить логи, созданные контейнером Nginx, вы можете использовать команду logs:

```
docker-compose logs
Attaching to compose-demo_web_1
web_1  | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will
attempt to perform configuration
web_1  | /docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/
web_1  | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-
ipv6-by-default.sh
web_1  | 10-listen-on-ipv6-by-default.sh: Getting the checksum of
/etc/nginx/conf.d/default.conf
web_1  | 10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in
/etc/nginx/conf.d/default.conf
web_1  | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-
```

```
on-templates.sh
web_1  | /docker-entrypoint.sh: Configuration complete; ready for start up
web_1  | 172.22.0.1 - - [02/Jun/2020:10:47:13 +0000] "GET / HTTP/1.1" 200 353
"-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/83.0.4103.61 Safari/537.36" "-"
```

Чтобы поставить выполнение среды на паузу, не изменяя при этом текущего состояния контейнеров, используйте команду:

```
docker-compose pause
Pausing compose-demo_web_1 ... done
```

Чтобы восстановить работу среды, используйте:

```
docker-compose unpause
Unpausing compose-demo_web_1 ... done
```

Команда stop прервет работу контейнера, но сохранит все данные, связанные с ним.

```
docker-compose stop
Stopping compose-demo_web_1 ... done
```

Если вы хотите удалить контейнеры, сети и тома, связанные с этой контейнерной средой, используйте команду down:

```
docker-compose down
Removing compose-demo_web_1 ... done
Removing network compose-demo_default
```

Обратите внимание, это не удалит базовый образ, используемый Docker Compose для запуска вашей среды (в нашем случае nginx: alpine). Таким образом, всякий раз, когда вы снова запускаете свою среду с помощью docker-compose up, процесс будет запускаться намного быстрее, поскольку образ уже находится в вашей системе.

Если вы хотите удалить базовый образ из вашей системы, вы можете использовать эту команду:

```
docker image rm nginx:alpine
Untagged: nginx:alpine
Untagged:
nginx@sha256:b89a6ccbda39576ad23fd079978c967cecc6b170db6e7ff8a769bf2259a71912
Deleted:
sha256:7d0cdcc60a96a5124763fddf5d534d058ad7d0d8d4c3b8be2aefedf4267d0270
Deleted:
sha256:05a0eaca15d731e0029a7604ef54f0dda3b736d4e987e6ac87b91ac7aac03ab1
Deleted:
sha256:c6bbc4bdac396583641cb44cd35126b2c195be8fe1ac5e6c577c14752bbe9157
Deleted:
sha256:35789b1e1a362b0da8392ca7d5759ef08b9a6b7141cc1521570f984dc7905eb6
Deleted:
sha256:a3efaa65ec344c882fe5d543a392a54c4ceacd1efd91662d06964211b1be4c08
```

```
Deleted:
sha256:3e207b409db364b595ba862cdc12be96dcdad8e36c59a03b7b3b61c946a5741a
```

Заключение

Теперь вы умеете устанавливать Docker Compose и управлять контейнерами с помощью этого инструмента.

Данное руководство охватывает только основы работы с Docker Compose. Больше информации об этом инструменте вы найдете в его официальной документации.

13 Использование docker-compose в разработке

Docker-compose — это система сборки, запуска и управления множеством контейнеров. Docker-compose не входит в единый пакет поставки Docker и устанавливается отдельно.

Для сборки кластера контейнеров используется docker-compose.yml.

Docker-compose.yml — конфигурационный файл в YAML-формате, описывающий логику запуска и взаимодействия контейнеров между собой и сервером. В сущности инструкции заложенные в docker-compose.yml по логике работы идентичны команде docker run.

Далее рассмотрим примеры docker-compose.yml файлов и контейнерные структуры docker, используемые в web-разработке.

13 .1 MySQL и PHPMyAdmin

Создадайте файл docker-compose.yml со следующим содержимым

```
version: '3.2'

services:
  db:
    image: mysql:8.0
    container_name: appsDB
    restart: always
    ports:
      - '6603:3306'
    environment:
      MYSQL_ROOT_PASSWORD: helloworld
  app:
    depends_on:
      - db
    image: phpmyadmin/phpmyadmin
    container_name: phpmyadmin
    restart: always
    ports:
      - '82:80'
    environment:
      PMA_HOST: db
```

Запустим этот файл с помощью следующей команды

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

13 .2 Nginx и PHP

Создайте файл `docker-compose.yml` со следующим содержимым:

```
version: '3.9'

services:
    web:
      image: nginx:latest
      ports:
        - "8001:80"
      volumes:
        - ./:/var/www/html
        - ./docker/nginx/:/etc/nginx/conf.d/
      links:
        - php-fpm
    php-fpm:
      image: php:8-fpm
      volumes:
        - ./:/var/www/html
```

Далее необходимо создать папку `docker/nginx` в ней файл `default.conf` с настройками `Nginx`.

Содержимое `default.conf` по умолчанию

```
server {
    index index.php index.html;
    server_name phpfpm.local;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /var/www/html;
    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass php-fpm:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
}
```

14 Использование Docker в Laravel

Используя Docker мы можем запустить приложение Laravel

Для этого, в папке с проектом необходимо создать файл `Dockerfile`:

Пример для проектов на Laravel 8 с PHP 7.2:

```
FROM php:7.2-cli

RUN apt-get update -y && apt-get install -y libmcrypt-dev

RUN curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer
RUN docker-php-ext-install pdo mbstring

WORKDIR /app
COPY . /app

RUN composer install

EXPOSE 8000
CMD php artisan serve --host=0.0.0.0 --port=8000
```

После создания Dockerfile нужно создать реальный образ Docker. В терминале перейдите в корневую папку вашего проекта и запустите:

```
docker build -t my_php7 .
```

После сборки можете использовать образ

```
docker run -p 8000:8000 my_php7
```

В дополнение к Docker файлу создайте файл `docker-compose.yml`

```
web:
    image: my_php7
    ports:
        - 8000:8000
    volumes:
        - ./:/app
```

Запускаем docker-compose:

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

Заключение

Список использованных источников

Приложения