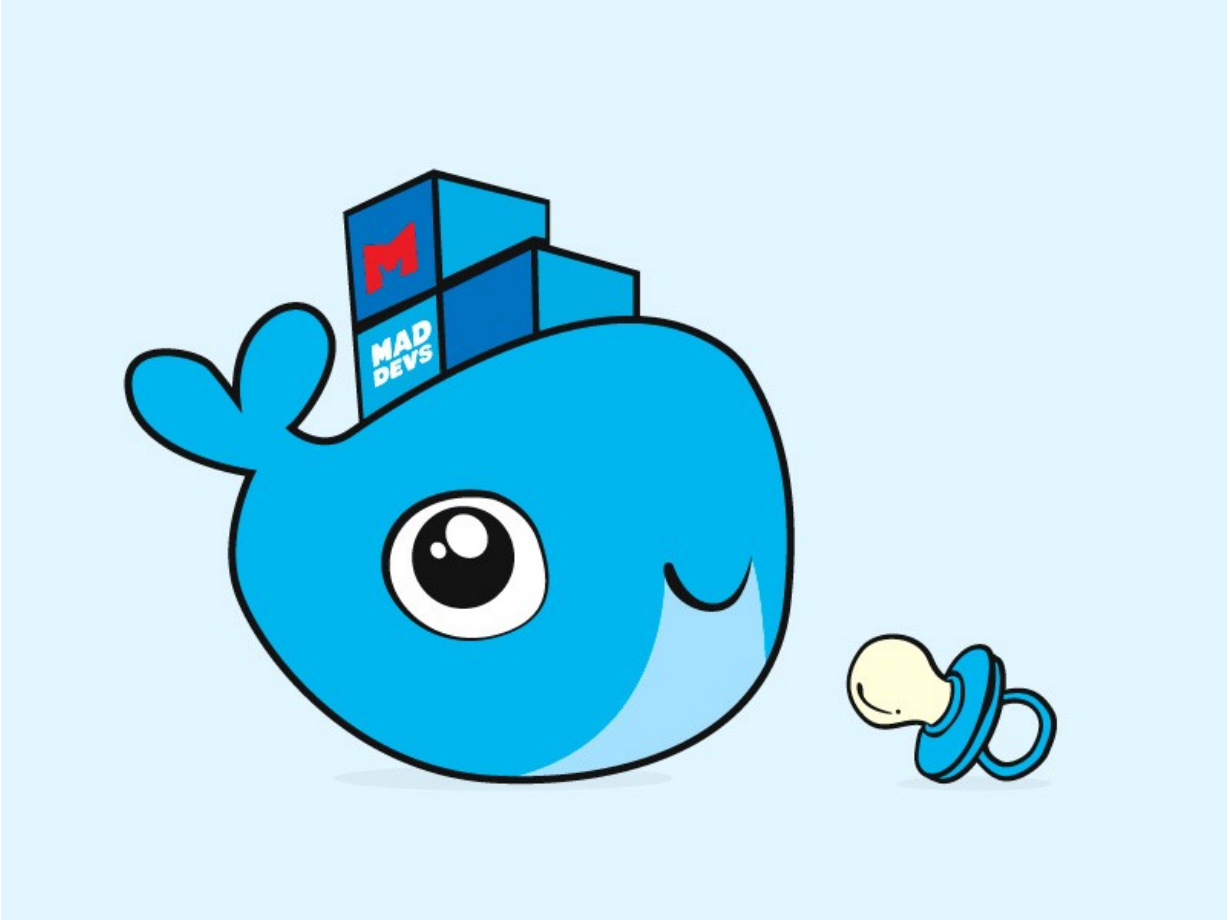
Docker для начинающего разработчика

Andrew S. Follow
May 8, 2017 · 13 min read



Docker стремительно ворвался в мир контейнеров и за пару лет из надстройки над LXC развился в систему запуска, окрестрирования, кластеризации, конфигурирования, поставки и создания контейнеров с программным обеспечением. Он так-же имеет простой интерфейс для контроля над ограничением ресурсов. Основой основ для докера служат linux namespaces, которые позволяют изолировать и виртуализировать ресурсы системы, такие как сеть, процессы, точки монтирования и пользователи. В итоге мы можем запустить любой процесс совершенно изолированно, как от самой системы так и от других контейнеров, в своем уникальном программном окружении, со своей сетью, деревом процессов, файловой системой и сетью. Все это дело работает исключительно поверх линукса, docker for windows, docker for osx — это велосипеды с линуксовской виртуалкой. Поэтому некоторые вещи работать там не будут, а некоторые ток с бубном. Имея опыт работы с джейлами во FreeBSD, zfs-зонами Solaris, openvz и lxc системами

линукс можно с уверенностью заявить, что докер вылечил большинство болячек предшественников, и значительную часть головников админов/девопсов о том, как запустить перевел в плоскость, где запустить). Имея докерфайл, дженкинсфайл, вагрантфайл и анзибл с паппетом дев, опс, девопс теперь имеет в наличии everything as a code. Что позволяет ему воспроизвести с нуля весь стек разработки и поставки в любое время в любой среде.

Звучит круто, с чего мне начать?

Процесс установки бессмысленно описывать, так как разработчики докера слова “просто” и “понятно” понимают буквально, что в итоге выливается в хорошие доки и простые шаги установки. Здесь можно найти доки для любой популярной платформы. Итак, докер у нас есть и введя docker в консоли мы ужасаемся от списка команд вываленных на экран. Стоит отметить, что с версии 1.13 появился раздел Mangement Commands, куда со временем перетекут все команды, а они в свою очередь будут удалены. А пока например команды docker container run и docker run идентичны. Большинство опенсорсных проектов сейчас имеют докерфайлы и уже готовые докер-имейджи. Разработчику нет необходимости устанавливать на свою машину mysql/postgres/redis/mongo/apache/python/nodejs/php итд — все необходимые версии софта упакованы разрабами в официальне образы. Для примера запустим nginx:

```
$ docker run nginx
```

Данная команда скачает latest имейдж с офиц репо nginx на докерхаб, создаст и запустит контейнер с nginx(по сути docker run — это комбинация команд docker pull, docker create, docker start и docker attach). В соседней консоли наберем команду docker ps

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5d1e75edaf10 nginx “nginx -g ‘daemon …” 24 seconds ago Up 22 seconds
80/tcp, 443/tcp silly_bassi
```

Здесь мы видим, что у нас запущен один контейнер с обазом nginx. И по сути от нашей команды docker run больше никакой пользы и нет. Чтобы убить контейнер нужно ввести команду docker kill <container id | container name>. Чтобы увидеть помимо запущенных и потушенные контейнеры — можно набрать docker ps -a:

```
$ docker kill silly_bassi
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
$ docker ps -a
```

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5d1e75edaf10 nginx “nginx -g ‘daemon …” 8 minutes ago Exited (0) 0
minutes ago silly_bassi
$ docker start silly_bassi
silly_bassi
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5d1e75edaf10 nginx “nginx -g ‘daemon …” 11 minutes ago Up 16 seconds
80/tcp, 443/tcp silly_bassi
$ docker rm -f 5d1e75edaf10
5d1e75edaf10
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Наигрались и удалили контейнер совсем.
Немного добавим параметров в запуск контейнера:

```
$ docker run -d — name nginx -p 8080:80 nginx
a22f05d60db2484bcdc026996e3c7562a9131c2f2cb86a42b7e52b9490310e71
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
a22f05d60db2 nginx “nginx -g ‘daemon …” 5 minutes ago Up 5 minutes
0.0.0.0:8080->80/tcp, 443/tcp nginx
```

Из непонятных параметров я думаю тут -d и -p. Первый флаг запускает контейнер в бэкграунде, второй паблишит порт(форвардит порт из системы) в контейнер. Теперь перейдя по адресу http://localhost:8080 мы окажемся на приветственной страничке nginx. В систему можно прокидывать любой произвольный порт, но только на тот, что экспозится в контейнере, -p можно задать несколько раз, можно так-же задать диапазон. Команда ниже покажет stdout и stderr контейнера:

```
$ docker logs nginx
172.17.0.1 — — [28/Feb/2017:08:16:41 +0000] “GET / HTTP/1.1” 304 0
“-” “Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/56.0.2924.87 Safari/537.36” “-”
172.17.0.1 — — [28/Feb/2017:08:16:41 +0000] “GET / HTTP/1.1” 304 0
“-” “Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/56.0.2924.87 Safari/537.36” “-”
172.17.0.1 — — [28/Feb/2017:08:16:41 +0000] “GET / HTTP/1.1” 304 0
“-” “Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/56.0.2924.87 Safari/537.36” “-”
172.17.0.1 — — [28/Feb/2017:08:16:41 +0000] “GET / HTTP/1.1” 304 0
“-” “Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/56.0.2924.87 Safari/537.36” “-”
```

Иногда приходится цепляться к уже работающему контейнеру, проверять наличие необходимого контента или доустанавливать какие-либо тулзы, для этих целей мы юзаем команду:

```
$ docker exec -ti nginx bash
root@a22f05d60db2:/# ps auwwx
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 31872 4988 ? Ss 08:10 0:00 nginx: master process nginx
-g daemon off;
nginx 6 0.0 0.0 32260 3564 ? S 08:10 0:00 nginx: worker process
root 7 0.0 0.0 20244 3252 ? Ss 08:30 0:00 bash
root 13 0.0 0.0 17500 2104 ? R+ 08:31 0:00 ps auwwx
```

Так мы открываем интерактивную псевдо-tty консоль(-ti) а запускаем там bash. И можем установить необходимый софт, который будет доступен на время жизни контейнера:

```
root@a22f05d60db2:/# apt-get update && apt-get install -y curl
root@a22f05d60db2:/# curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
 body {
 width: 35em;
 margin: 0 auto;
 font-family: Tahoma, Verdana, Arial, sans-serif;
 }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href=”http://nginx.org/”>nginx.org</a>.<br/>
Commercial support is available at
<a href=”http://nginx.com/”>nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Чтобы контейнер с nginx стал еще полезнее, пусть он отдает нашу статику:

```
$ docker rm -f nginx
$ docker run -d --name nginx -p 80:80 -v
/static/content:/usr/share/nginx/html nginx
```

Таким образом мы монтируем папку /static/content в контейнер с nginx. Аналогично можно примонтировать в контейнер кастомные конфиги в папку /etc/nginx контейнера. В качестве источника можно указать другой контейнер.

Полный список команд докера и их параметров можно посмотреть тут.

Как я уже писал выше, по дефолту docker качает latest таг имейджа, если необходимо указать конкретную версию, это делается через двоеточие например latest версия nginx на базе alpine-linux будет выглядеть как nginx:alpine. Доступные и поддерживаемые версии лучше всего смотреть на официальных страничках докерхаба и стора.

Увлекательно, но мало!

Попробуем разобрать более сложный вариант — мы разрабатываем тему для wordpress и нам необходимо его быстро развернуть у себя локально. Инсталлить пых как модуль апача или фпм, инсталлить мускуль, это все настраивать, а если несколько тем — носиться с вхостами, отдельными бд итд уныло и утомительно. Поэтому идем на офиц странички wordpress и mysql, изучаем, выбираем версии и вперед.
Работать мы будем из корня проекта, тема наша лежит в папке themes/mytheme.
Запускаем базу данных:

```
$ docker run -d --name=db -e MYSQL_ROOT_PASSWORD=password mysql:5.7
```

По умолчанию пользователь будет root, а через переменную окружения мы задаем пароль
Запускаем Вордпресс, коннектим его к бд и монтируем в контейнер нашу тему:

```
$ docker run -d --name=wp --link=db -p 80:80 -v
$(pwd)/themes:/var/www/html/wp-content/themes -e WORDPRESS_DB_HOST=db
-e WORDPRESS_DB_PASSWORD=password wordpress:4.7
```

По дефолту контейнеры друг друга не видят, здесь мы использовали параметр link, чтобы законнектить контейнер с Вордпрессом к контейнеру с бд. Помимо этого включается докеровский резолвер и ипка контейнера с базой данных будет резолвиться по имени db. Этот хостнейм мы используем в настройке Вордпресса через переменную окружения WORDPRESS_DB_HOST. Перейдя на http://localhost мы закончим установку и можем применить нашу тему.
Как видим, две команды и Вордпресс готов. Однако и это можно улучшить — команда докера написала отличный скрипт, который очень полезен для разворачивания комплексных сервисов на рабочих станциях разработчиков.

docker-compose

После простой инсталляции перейдем сразу к предыдущему примеру и перенесем его в docker-compose.yml. Данный файл описывает наши сервисы и контейнеры, параметры запуска итд. На текущий момент существует уже 3.1 версия формата. каждый ямл файл должен начинаться с версии. По умолчанию docker-compose ищет docker-compose.yml в текущей дире, через параметр -f можно указать любой другой файл или перечислить все файл, если их несколько.

Касательно версий, то если грубо — version: “1” — это простое описание команд запуска докер контейнеров в ямл формате. В version: “2” добавилось понятие сервисов, теперь контейнеры можно скейлить, так-же добавилась секция с описанием вольюмов и сетей и их драйверов. По умолчанию, если сеть не задана, docker-compose при запуске создает новый бридж и коннектит все сервисы к нему, включает резолвер (резовлятся как сервисы, так и отдельные контейнеры). Не нужны линки для взаимосвязи между контейнерами. Version: “3” работает только с docker-engine 1.13+. Данная версия учла кривой опыт doker bundle и реализовала простой и прозрачный деплой в swarm-кластер, в 3.1 запилили поддержку docker secrets. Для сингл хоста за глаза второй версии.
docker-compose.yml:

```
version: '2'
services:
 wp:
  image: wordpress:4.7
  volumes:
   - ./themes:/var/www/html/wp-content/themes
  ports:
   - 80:80
  environment:
   - WORDPRESS_DB_PASSWORD=password
   - WORDPRESS_DB_HOST=db

 db:
  image: mysql:5.7
  environment:
   - MYSQL_ROOT_PASSWORD=password
```

Данный файл положим в корень проекта(как мы видим docker-compose понимает относительные пути, поэтому мы указали ./themes в описание вольюма с темой). Чтобы запустить наш Вордпресс достаточно набрать команду docker-compose up или docker-compose up -d чтобы запустить в бэкграунде:

```
$ docker-compose up -d
# trimmed output
$ docker-compose ps
  Name                Command               State         Ports
------------------------------------------------------------------------
-
```

```
tmp_db_1   docker-entrypoint.sh mysqld      Up      3306/tcp
tmp_wp_1   docker-entrypoint.sh apach ...   Up      0.0.0.0:80-
>80/tcp
```

Откроем в браузере http://localhost

Данный файл удобно держать с проектом. Таким образом мы или кто-либо еще всегда смогут поднять проект(в нашем случае проект с темой Вордпресса) на любом компе с докером.
Подчистим за собой:

```
$ docker-compose down
Stopping tmp_wp_1 … done
Stopping tmp_db_1 … done
Removing tmp_wp_1 … done
Removing tmp_db_1 … done
Removing network tmp_default
```

Напишем более сложный ямл, для Вордпреса docker-compose-staging.yml:

```
version: '2'

networks:
  wp-proxy:
    driver: bridge
  wp-db:
    driver: bridge

volumes:
  db-mysql:
  wp-uploads:
  wp-plugins:

services:

  nginx:
    image: jwilder/nginx-proxy
    ports:
      - 80:80
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
    networks:
      - wp-proxy

  wp:
    image: wordpress:4.7
    volumes:
      - ./themes:/var/www/html/wp-content/themes
      - wp-uploads:/var/www/html/wp-content/uploads
      - wp-plugins:/var/www/html/wp-content/plugins
    environment:
      WORDPRESS_DB_PASSWORD : password
      WORDPRESS_DB_HOST     : db
```

```
      WORDPRESS_DB_USER     : wp
      WORDPRESS_DB_NAME     : wp
      VIRTUAL_HOST          : wp.local
    networks:
      - wp-proxy
      - wp-db

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD   : root_password
      MYSQL_PASSWORD        : password
      MYSQL_USER            : wp
      MYSQL_DATABASE        : wp
    volumes:
      - db-mysql:/var/lib/mysql
    networks:
      - wp-db
```

В примере выше мы создали вольюм для базы данных db-mysql и примонтировали его в контейнер с мускулем в /var/lib/myql. Сделали мы это для того, чтобы при пересоздании, рестарте итд — данные сохранялись. Так-же создали две сети: внешняя wp-proxy для соединения сервисов nginx и wordpress и внутреннюю wp-db для коннекта сервиса водпресс с базой данных. Для сервиса Вордпресс помимо темы, мы создали два вольюма, для того чтобы заперсистить плагины и аплоад. Теперь при пересоздании контейнеров бд и Вордпресса все наши данные будут сохранятся, так как мы вынесли их из контейнеров.
В качестве прокси мы заюзали имейдж jwilder/nginx-proxy, который слушает сокет докера, и если находит переменную окружения VIRTUAL_HOST — создает указанный виртуалхост у себя в конфигах с проксированием на контейнер, где прописана эта переменная. Если контейнер экспозит несколько портов, надо указать так-же и VIRTUAL_PORT, куда будет проксироваться трафик.

Запустим наше творение:

```
$ docker-compose -f docker-compose-staging.yml up -d
Creating network “tmp_wp-db” with driver “bridge”
Creating network “tmp_wp-proxy” with driver “bridge”
Creating volume “tmp_wp-plugins” with default driver
Creating volume “tmp_wp-uploads” with default driver
Creating tmp_wp_1
Creating tmp_nginx_1
Creating tmp_db_1

$ docker-compose -f docker-compose-staging.yml ps
   Name                 Command               State
Ports
-------------------------------------------------------------------------
-------------
tmp_db_1      docker-entrypoint.sh mysqld      Up      3306/tcp
tmp_nginx_1   /app/docker-entrypoint.sh  ...   Up      443/tcp,
0.0.0.0:80->80/tcp
tmp_wp_1      docker-entrypoint.sh apach ...   Up      80/tcp
```

В /etc/hosts укажем

```
127.0.1.1 wp.local
```

При переходе в браузере по ссылке http://wp.local откроется привычный визард настроки вордпресса. Внутри контейнера nginx-proxy в конфиг /etc/nginx/conf.d/default автоматом добавится следующая запись:

```
# wp.local
upstream wp.local {
  ## Can be connect with "tmp_wp-proxy" network
  # tmp_wp_1
  server 172.21.0.3:80;
}
server {
  server_name wp.local;
  listen 80 ;
  access_log /var/log/nginx/access.log vhost;
  location / {
   proxy_pass http://wp.local;
  }
}
```

Теперь самое интересное, т.к. у нас все изменяемые данные Вордпресса вынесены наружу, мы можем без боязни его скейлить:

```
$ docker-compose -f docker-compose-staging.yml scale wp=2

Creating and starting tmp_wp_2 ... done

docker-compose -f docker-compose-staging.yml ps

Name                   Command               State              Ports

------------------------------------------------------------------------
-------------

tmp_db_1      docker-entrypoint.sh mysqld      Up      3306/tcp

tmp_nginx_1   /app/docker-entrypoint.sh  ...   Up      443/tcp,
0.0.0.0:80->80/tcp

tmp_wp_1      docker-entrypoint.sh apach ...   Up      80/tcp

tmp_wp_2      docker-entrypoint.sh apach ...   Up      80/tcp
```

А секция upstream в nginx примет вид:

```
upstream wp.local {
  ## Can be connect with "tmp_wp-proxy" network
  # tmp_wp_2
  server 172.21.0.4:80;
  ## Can be connect with "tmp_wp-proxy" network
  # tmp_wp_1
  server 172.21.0.3:80;
}
```

Помимо этого, в пределах сети wp-proxy оба контейнера доступны по названию сервиса wp. А в версии 3 докер-композа мы можем провернуть тоже самое в swarm-кластере.

В итоге без какого-либо напряга мы создали масштабируемый сервис на Вордпрессе.

Убедил, хочу заиметь свой контейнер!

Для начала сделаем шаг назад — а что такое контейнеры? Обычному человеку при слове контейнер на ум может придти 20и тонник. Это прочный объект, который выдерживает нагрузку во время хранения, погрузки и транспортировки и защищают содержимое, находящееся внутри. В голове так-же может возникнуть картинка с портовым доком, где стоят тысячи контейнеров в рядах, составленные друг на друга. Большая часть мерчанта поставляется в таких контейнерах: они прочны, стандартны, их легко хранить и транспортировать. Основная часть людей, задействованная в поставке не имеет представления что находится внутри, для поставки это не имеет значения. Идея софтварных контейнеров аналогична — это неизменные изолированные образы с ПО, функционал которых доступен чаще всего через вызовы api. Это современное решение для надежного запуска ПО (почти) в любом окружении, будь то комп разработчика, тестовые сервера или продакшн кластер. Не важно где — результат запуска всегда будет неизменным.
Поскольку поведение предсказуемо и неизменно независимо от среды, то самый упоротый диалог на свете наконец уходит в небытие:

> *QA: не работает п'ный функционал на проде.*
> *Dev: но у меня на компе все ок!*

В терминологии докер: контейнер — это запущенный образ. Что из себя представляет изнутри сам образ, советую прочитать эту статью. Если грубо, то докеровский имейдж — это стопка более мелких имеджей, каждый из которых содержит файлы, команды, результат их выполнения и другую мета-инфу. Потом драйвер overlay при запуске все это собирает в заданном порядке и из контейнера это выглядит как цельная единая система. Помимо этого, при старте контейнера поверх всего создается новый слой/имейдж. При удалении контейнера по сути только этот слой и удаляется со всеми изменениями

произошедшими во время жизни контейнера, однако их можно “закоммитить” создав новый образ.
Текстовым представлением докеровского образа является Dockerfile. При команде docker build этот файл считывается, каждая строка-команда запускает новый контейнер, а ее результат коммитится в новый слой/имейдж. Далее рассмотрим простой пример и все станет понятно. Для этого склонируем проект:

```
$ git clone git@github.com:halfb00t/bamboo-build-tools.git
$ cd bamboo-build-tools
```

Проект не мой, я просто его форкнул, немного подправил и добавил Dockerfile:

```
FROM python:2.7-alpine
MAINTAINER Andrew S. <halfb00t@gmail.com>

RUN apk add --no-cache libxslt libxml2
RUN apk add --no-cache --virtual .build-deps build-base git libxslt-
dev libxml2-dev \
    && cd /tmp && git clone https://github.com/halfb00t/bamboo-build-
tools.git . \
    && python setup.py build && python setup.py install \
    && cd / && rm -rf /tmp/* \
    && apk del .build-deps

ENV APP_HOME /app
WORKDIR $APP_HOME
```

Каждый Dockefile начинается с команды FROM, так мы объявляем, какой базовый имейдж мы будем использовать. Далее может быть указан MAINTAINER, это просто служебная инфа о том, кто во этом всем виноват. В моем случае за базовый имейдж я взял python:2.7-alpine. после чего идут команды RUN. По сути это инструкция, что нужно выполнить внутри имейджа и закоммитить в новый слой. Если с первой командой все более-менее понятно, то второй RUN выглядит мягко говоря по уродски, ее можно было бы переписать так:

```
RUN apk add --no-cache --virtual .build-deps build-base git libxslt-
dev libxml2-dev
WORKDIR /tmp
RUN git clone https://github.com/halfb00t/bamboo-build-tools.git .
RUN python setup.py build
RUN python setup.py install
RUN rm -rf /tmp/*
RUN apk del .build-deps
```

Однако, необходимо помнить, что каждое успешное завершение команды ведет к созданию нового слоя, который записывается поверх предыдущих. Соответственно мы установили пачку пакетов, в тмп склонировали проект, а после установки почистили /tmp и удалили ненужные пакеты. Последние два слоя пометят файлы как удаленные, но сами слои не удалят, а запишутся поверх. В итоге у нас и место не очистится, и файлов в контейнере не будет. Собрав первый Dockerfile мы получим имейдж размером 94Mb, собрав второй, работающий точно так-же — 261Mb. Соответственно при написании докерфайлов всегда следует держать баланс между читаемостью и экономией места, особенно для прода, и по возможности объединять в oneliners команды инсталляции, деинсталляции и удаления, чтобы все это выполнялось в рамках одного слоя.

Помимо RUN есть еще команды COPY ADD CMD и другие. COPY копирует файлы из текущего контекста системы в имейдж по указанному пути. Следует помнить, что команда COPY /etc /etc не скопирует /etc/ системы в имейдж, корнем для команды будет директория, которая указана в контексте билда имейджа. Команда ADD аналогична COPY за тем исключением, что понимает урлы и архивы. CMD — команда, которую следует выполнить при запуске контейнера(ее можно определить/переопределить и при старте).

Как уже писал выше, за основу моего имейджа был взят python:2.7-alpine. Это официальный имейдж питона версии 2.7 на базе легковесного дистрибутива linux alpine. В качестве сборки мелких имеджей alpine — лучший выбор, однако следует помнить, что за основу здесь взята сишная либа musl и пока не каждый софт скомпиллится и заработает в alpine.

На проде мы используем debian, как базовый образ, несмотря на то, что весит он больше 120Mb против 4Mb у alpine. Это не проблема, т.к. этот слой копируется только один раз, а все остальные имейджи будут использовать локальную копию. Помимо этого, дебиан надежен и предсказуем, и по умолчанию в нем есть все необходимое.

Но для мелких тулз alpine подходит лучше всего. Самый первый исходный образ — scratch. По сути не содержит ничего. Его можно использовать для гошных бинарников. однако следует не забывать, что необходимо скопировать туда корневые ссл сертификаты и необходимые либы, которые покажет ldd. Советую почитать так-же, что пишут сами докеровцы о докерфайлах. Так-же отличным примером различных техник служат официальные докерфайлы Вордпресс, мускуль, пхп, го, питон, редис итд.

Чтобы собрать наш имейдж необходимо в директории проекта выполнить команду

```
$ docker build -t bamboo-build-tools .
```

В качестве имени имейджа мы указали bamboo-build-tools (можно указать несколько), а в качестве контекста для команд COPY и ADD точкой текущую диру. По умолчанию docker build ищет Dockerfile, однако через -f можно указать любой другой файл, аналогично и с контекстом — можно указать любой другой путь.

Еще один головняк для девелопера?

Пробежав галопом и не углубляясь в детали, наверняка сложилось двоякое впечатление. С одной стороны, это все интересно и местами очень полезно, а с другой, возможно остались сомнения — зачем это надо разработчику, ведь он может просто продолжать разрабатывать так, как привык и не тратить время на изучение докера, им пусть балуются опсы. Однако при всех возможных издержках, которые можно насочинять плюсов от использования докера неоспоримо больше, вот некоторые:

- разработчик перестает писать код для своего компа, а с первых коммитов начинает участвовать в процессе поставки на стейджинг и прод
- код будет работать одинаково в любой среде и популярной ОС
- окружение в виде докерфайлов и композов хранится в репо и всегда может быть воссоздано, будь то комп нового разработчика, интеграционка и т.д.
- рабочий комп не захламляется, любая версия необходимого софта может быть запущена в докере и так-же спокойно удалена
- с использованием докера, меняется и парадигма разработки. логику теперь можно разбить на множество микросервисов, каждый из которых может быть написан на любом фреймворке и языке программирования.

)