Docker-образа. Микросервисы.

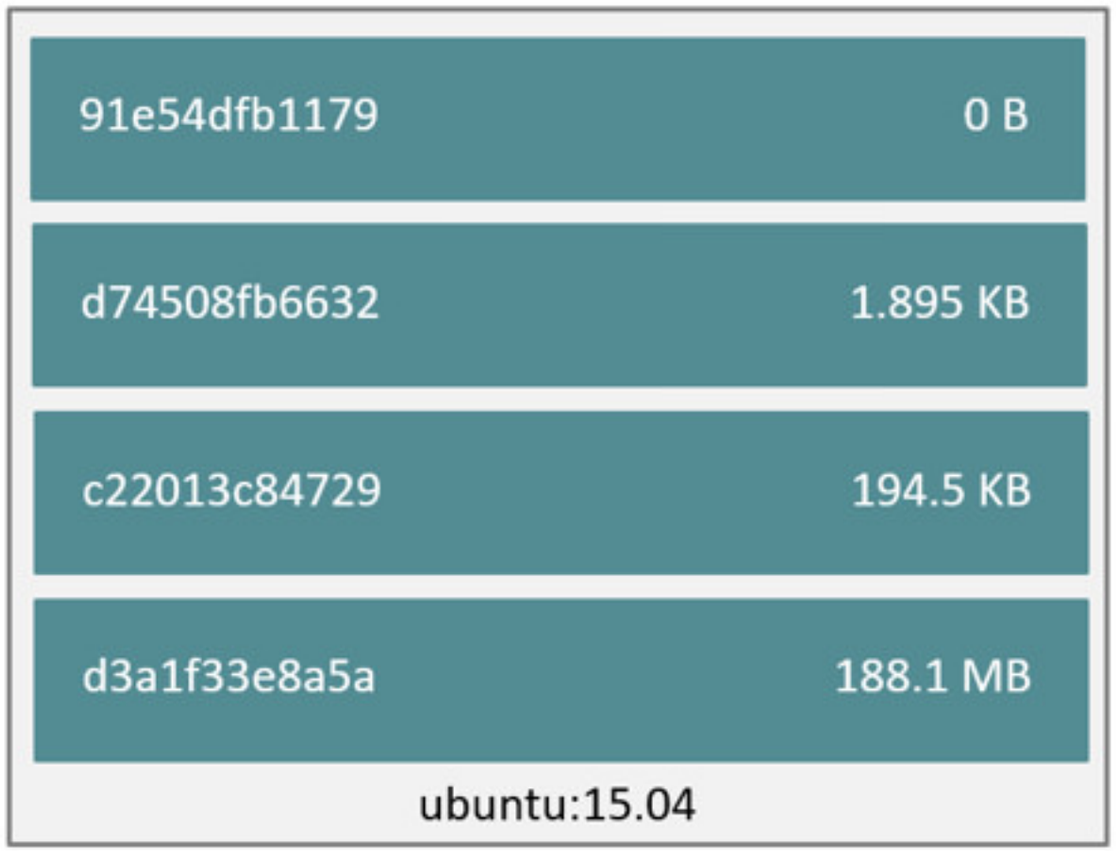
План

- Dockerfile
- Еще немного Dockerfile
- Базовые Образа
- Микросервисы

Dockerfile

- Текстовый файл с build инструкциями
- Инструкции декларативно описывают image
- каждая инструкция – промежуточный image
- сам build делает docker daemon

Docker images



Image

CMD

RUN

RUN

ADD/COPY

Dockerfile

Dockerfile:

```
# Комментарий
# Комментарии - это строки начинающиеся со знака #
 # Строка начинающаяся с пробела и решетки вызовет ошибку

INSTRUCTION arguments
INSTRUCTION multiline \
                arguments
```

FROM

FROM <image>[:<tag>]

<image> - имя базового образа
<tag> - опциональный атрибут указывающий на версию образа

Примеры:

```
FROM ubuntu:16.04
FROM quay.io/vektorlab/ctop
```

LABEL

LABEL <key>=<value> [<key>=<value> ...]

<key> - ключ
<value> - значение

Примеры:

```
LABEL maintainer="user@example.com"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

COPY

COPY <src> [<src> ...] <dst>

<src> - файл или директория внутри build контекста
<dst> - файл или директория внутри контейнера

Примеры:

```
COPY start* /startup/
COPY httpd.conf magic /etc/httpd/conf/
```

ADD

ADD <src> [<src> ...] <dst>

<src> - файл или директория внутри build контекста
<dst> - файл или директория внутри контейнера

Примеры:

```
ADD web-page-config.tar /
ADD http://example.com/foobar /
```

ENV

ENV <key> <value>

<key> - имя переменной окружения

<value> - присваиваемое значение

Примеры:

```
ENV LOG_LEVEL debug
ENV DB_HOST 127.0.0.1:3389
```

WORKDIR

WORKDIR <path>

<path> - путь внутри контейнера

Примеры:

WORKDIR /app

VOLUME

VOLUME <dst> [<dst> ...]

<dst> - директория монтирования для volume'a

Примеры:

```
VOLUME /app /db /data
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
```

EXPOSE

EXPOSE <port>[/<proto>] [<port>[/<proto>] ...]

<port> - порт по которому контейнер будет слушать
<proto> - tcp или udp

Примеры:

EXPOSE 5000
EXPOSE 8080/tcp 3389/udp

RUN <command>

<command> - команда которая будет выполнена при создании образа

Примеры:

```
RUN apt-get update && apt-get install nginx
RUN ["bash", "-c", "rm", "-rf", "/tmp/abc"]
```

CMD <command>

<command> - команда которая будет выполнена при старте контейнера

Примеры:

```
CMD /start.sh
CMD ["echo", "Dockerfile CMD demo"]
```

CMD варианты

- CMD script param param <-- shell form
- CMD ["script", "param", "param"] <-- exec form

ENTRYPOINT

ENTRYPOINT <command>

<command> - команда которая будет выполнена при старте контейнера

Примеры:

```
ENTRYPOINT exec top -b
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

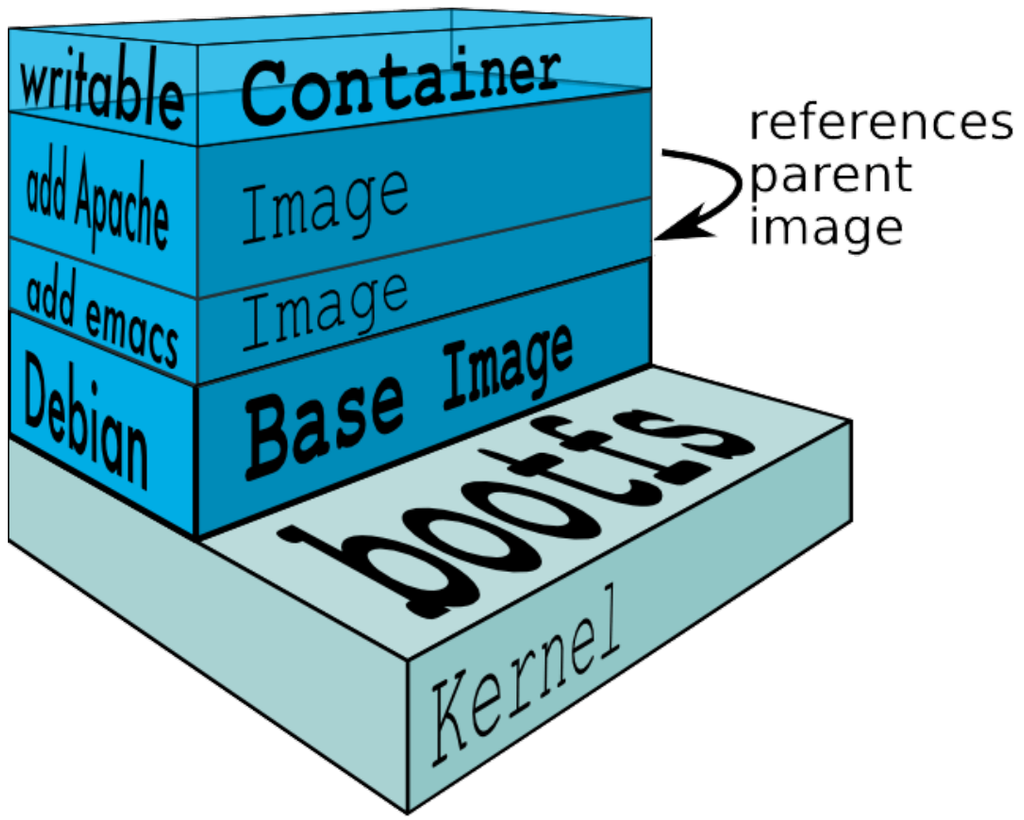
Dockerfile reference(partial)

Dockerfile:

ONBUILD	<- Информирует Docker, какую команду запустить при наследуемом биде
STOPSIGNAL	<- Указывает какой сигнал послать приложению при выходе
USER	<- Пользователь для RUN CMD итд команд
ARG	<- Похоже на ENV, но существует только в пределах docker build
HEALTHCHECK	<- Объявляет какой командой внутри контейнера можно проверить состояние

Подробная документация
https://docs.docker.com/engine/reference/builder/

Оптимизация сборки Docker Image



Оптимизация сборки Docker Image

- Каждая инструкция в Dockerfile это отдельный image
- Инструкции кешируются с помощью images

Оптимизация сборки Docker Image

- Каждая команда в Dockerfile, новый image

```
ENV myvar true                                          <- image!!!

RUN apt-get install -y nginx                            <- image!
RUN apt-get install -y php-fpm                          <- image!
RUN apt-get install -y imagemagick                      <- image!

ADD https://some-site.com/soft/master.tar.gz /bin/      <- image!!!!

CMD ["/bin/cool-soft"]                                  <- image!
```

Оптимизация сборки Docker Image

- Каждая команда в Dockerfile, новый image

```
ENV myvar true                                          <- image

RUN apt-get install -y nginx && \                       <- image
    apt-get install -y php-fpm && \
    apt-get install -y imagemagick

ADD https://some-site.com/soft/master.tar.gz /bin/      <- image

CMD ["/bin/cool-soft"]                                  <- image
```

Поменяем порядок инструкций

Есть ли разница?

```
ENV myvar false
ADD https://some-site.com/soft/master.tar.gz /bin/

RUN apt-get install -y nginx && \
    apt-get install -y php-fpm && \
    apt-get install -y imagemagick

CMD ["/bin/cool-soft"]
```

Кеширование сборки

Как работала упаковка новой версии кода в образ до изменений:

```
ENV myvar true                                                    <- image, cache hit

RUN apt-get install -y nginx && \
    apt-get install -y php-fpm && \
    apt-get install -y imagemagick                                <- image, cache hit

ADD https://some-site.com/soft/master.tar.gz /bin/                <- image, cache miss

CMD ["/bin/cool-soft"]                                            <- image, cache miss
```

Кеширование сборки

Как будеть работать после изменений?

```
ENV myvar false                                          <- image, cache hit
ADD https://some-site.com/soft/master.tar.gz /bin/       <- image, cache miss

RUN apt-get install -y nginx && \
    apt-get install -y php-fpm && \                      <- image, cache miss
    apt-get install -y imagemagick

                                                         <- image, cache miss
CMD ["/bin/cool-soft"]
```

Вывод: порядок инструкций важен!

Работа с кешем

- Кеширование очень важна для реализации быстрых сборок

- ADD, COPY - файлы кешируется, в случае изменений файлов, кеш сбрасывается

- Для остальных команда(втч RUN) проверяется только изменение команды.

- Например: RUN apt-get -y update, не проверяет обновления постоянно, только первый раз

- docker build --no-cache

.dockerignore

- Содержимое директории указанной при docker build попадает в build контекст
- Лучшая практика - держать в директории только необходимое
- Иначе используйте .dockerignore

Общие рекомендации

- Избегайте установки лишних пакетов
- Уменьшайте количество слоев
- Один контейнер - одна задача
- Чистите за собой
- Разные контейнеры для сборки и запуска

Пример Dockerfile

Имеем скомпилированное приложение, которое не имеет зависимостей

```
FROM ubuntu:14.04
COPY ./hello-world .
EXPOSE 8080
CMD [ "./hello-world" ]
```

Пример Dockerfile

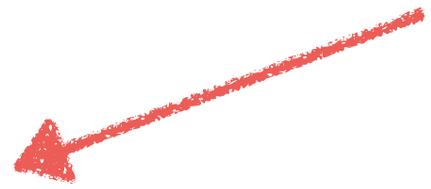
Проверим размер образа созданного из такого Dockerfile:

```
$ docker images
```

REPOSITORY	TAG	CREATED	SIZE
artemkin/hello-app	1.0	15 seconds ago	**194MB**
ubuntu	14.04	8 days ago	**188MB**

Пример Dockerfile

А нужен ли нам образ ОС?



```
FROM ubuntu:14.04
COPY ./hello-world .
EXPOSE 8080
CMD [ "./hello-world" ]
```

Билдим образ с нуля

Зарезервированное имя образа scratch, которое Docker считает как отсутсвие операции:

```
FROM scratch
COPY ./hello-world .
EXPOSE 8080
CMD [ "./hello-world" ]
```

Разница в размере образа

```
$ docker images

REPOSITORY              TAG                 CREATED             SIZE
artemkin/hello-app      2.0                 51 seconds ago      5.85MB
artemkin/hello-app      1.0                 15 minutes ago      194MB
```

Выбор базового образа

- Должен содержать необходимое ПО для сборки образа и запуска приложения
- Полезно иметь утилиты для дебага: telnet, ping
- Популярный базовый образ alpine

Уменьшение размера образа

- Удаляйте за собой архивы и временные файлы, которые остались во время билда

Уменьшение размера образа

- В результирующем image остается zip архив

```
COPY <filename>.zip <copy_directory>          <- image
RUN unzip <filename>.zip                      <- image
```

Уменьшение размера образа

- Остаются наследуемые images с zip архивом

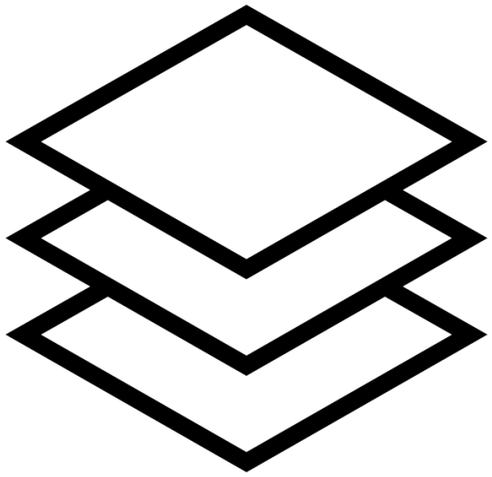
```
COPY <filename>.zip <copy_directory>          <- image
RUN unzip <filename>.zip                      <- image
RUN rm <filename>.zip                         <- image
```

Уменьшение размера образа

- Создается только один image с данными архива

```
RUN curl <file_download_url> -O <copy_directory> \                    <- image
&& unzip <copy_directory>/<filename>.zip -d <copy_directory> \
&& rm <copy_directory>/<filename>.zip
```

Типичный проект

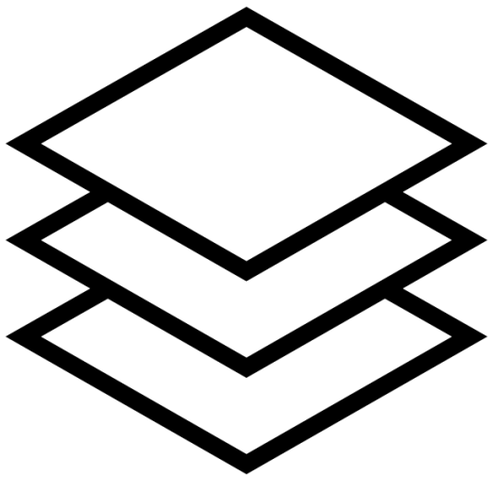


1/2/3 приложения

и/или монолитное приложение

хранилище данных

Изменения

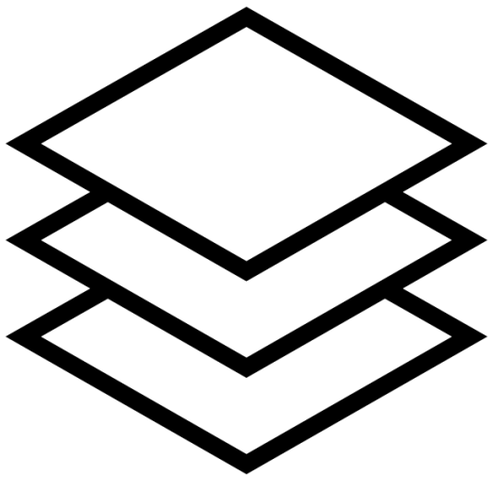


— конфигурация ПО: puppet/Chef/Ansible/Salt/...

— деплой: rpm/Fabric/...

— configuration drift

Поддержка



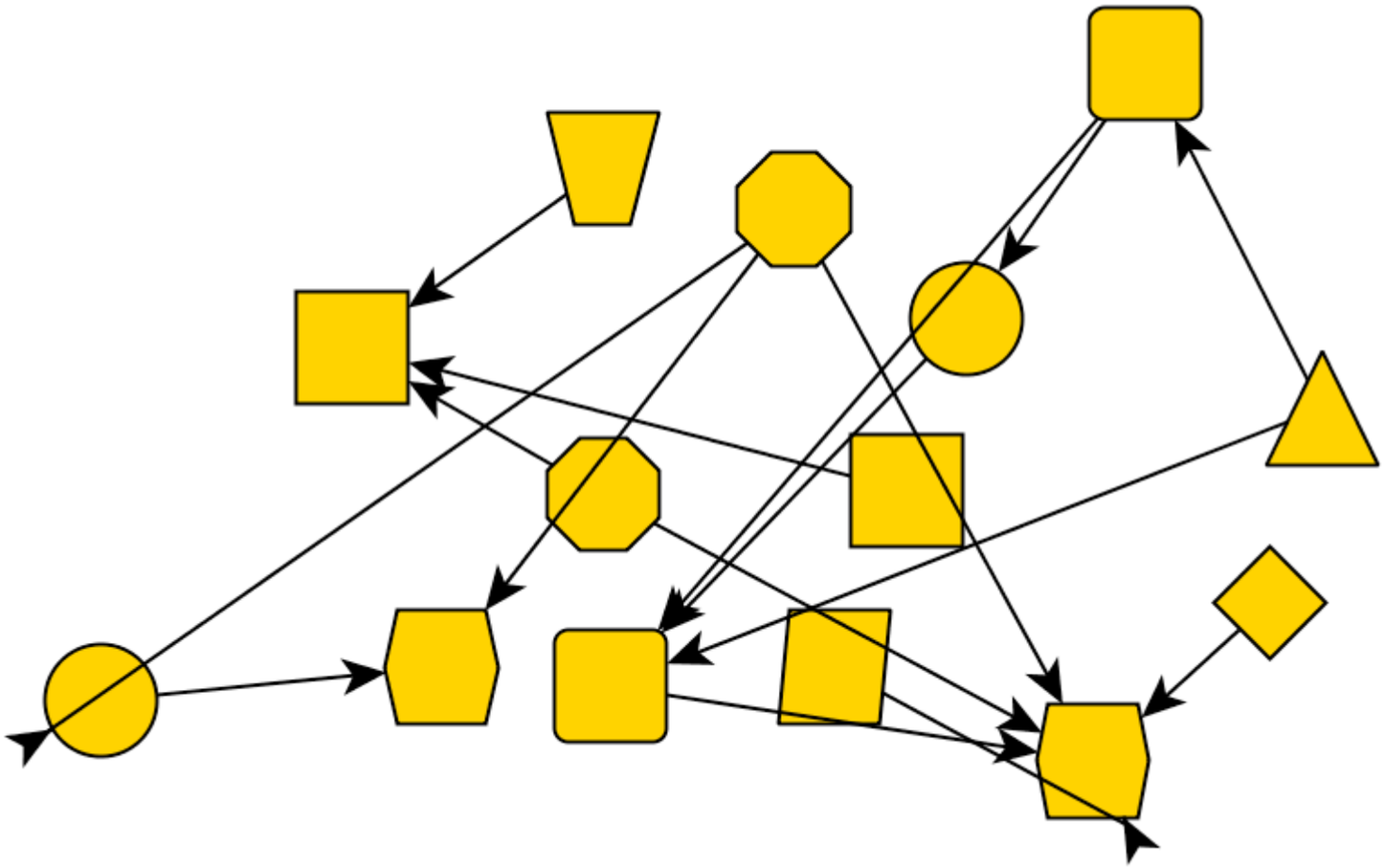
— рассчитать потребление ресурсов просто
— понятно какой компонент где запускать

(Микро)сервисы

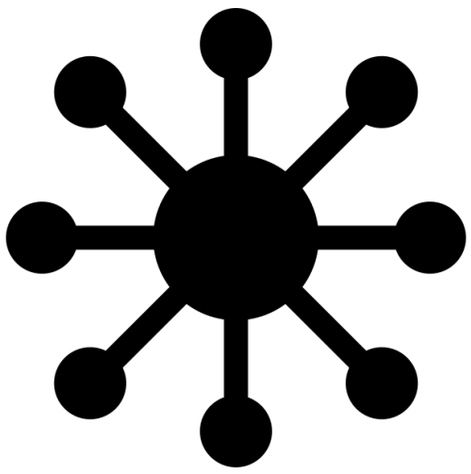
DB/back/front

Микросервисы

Микросервисы



Микросервисы



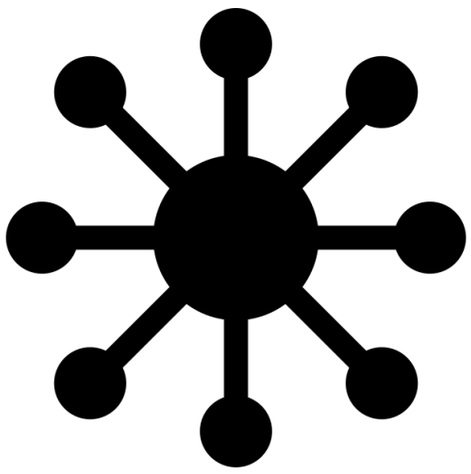
— 10/20/30 компонентов

— могут иметь связи any-to-any

— компоненты появляются и исчезают

— реестр сервисов(static, dynamic, service discovery)

Изменения

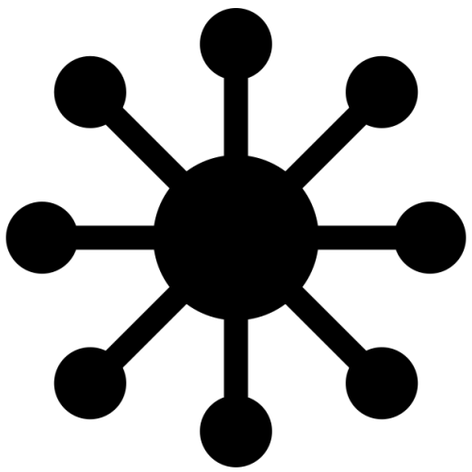


— конфигурация ПО: Chef/ Ansible

— rpm/Fabric/... – неудобно

— configuration drift!!!

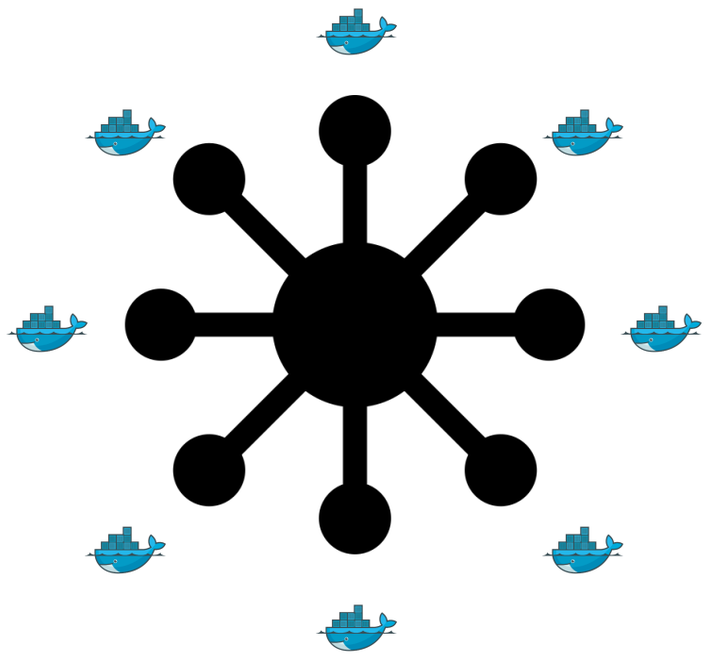
Поддержка



— рассчитать потребление ресурсов сложно

— ручная аллокация приложений

Микросервисы с Docker



— стандартизация

— изоляция

— декларативное описание среды

— большая плотность на один хост

Особенности

- Приходится адаптировать приложения
- Для эффективного использования нужно много других сущностей(service discovery, CI/CD, оркестраторы, registry)

Особенности запуска приложение в Docker среде

1. Один репозиторий -> одно приложение -> много развертываний

2. Четко объявляйте свои зависимости и прописывайте их в Dockerfile

3. Задавайте всю конфигурацию через переменные окружения

4. Считайте все сторонние сервисы внешними ресурсами, которые могут вести себя непредсказуемо

5. Запускайте приложение как один или несколько процессов не сохраняющих внутреннее состояние (stateless)

Особенности запуска приложение в Docker среде

6. Взаимодействуйте с приложениями по сети
7. Масштабируйте приложение с помощью процессов
8. Доверьте обработку логов Docker-у, не пишите логи самостоятельно, пишите в stdout

Особенности запуска приложение в Docker среде

https://12factor.net/ru/