Docker images
Микросервисы

Не забудь включить запись!

План

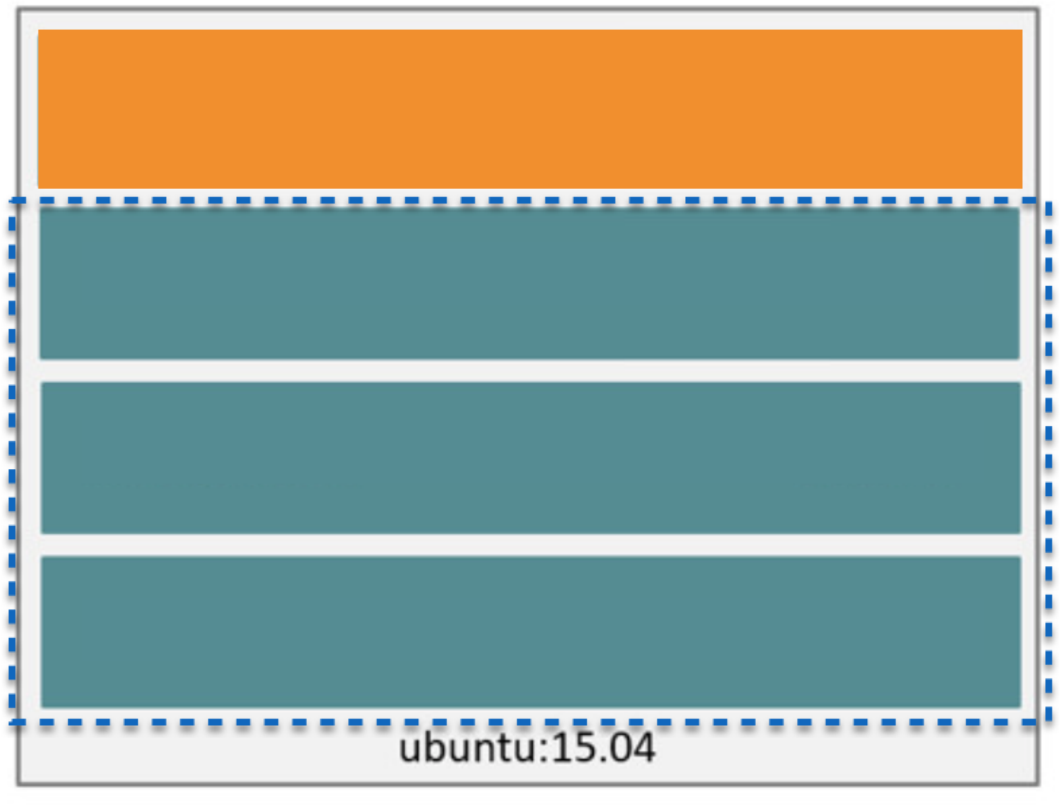
- Основы написания `Dockerfile`
- Готовим `Dockerfile` правильно
- Базовые образы
- Микросервисы

Работа с Dockerfile

Dockerfile

- Текстовый файл с последовательным описанием инструкций для сборки образа
- Каждая инструкция создает промежуточный слой образа
- Сам сборку делает демон Docker, а не Docker CLI

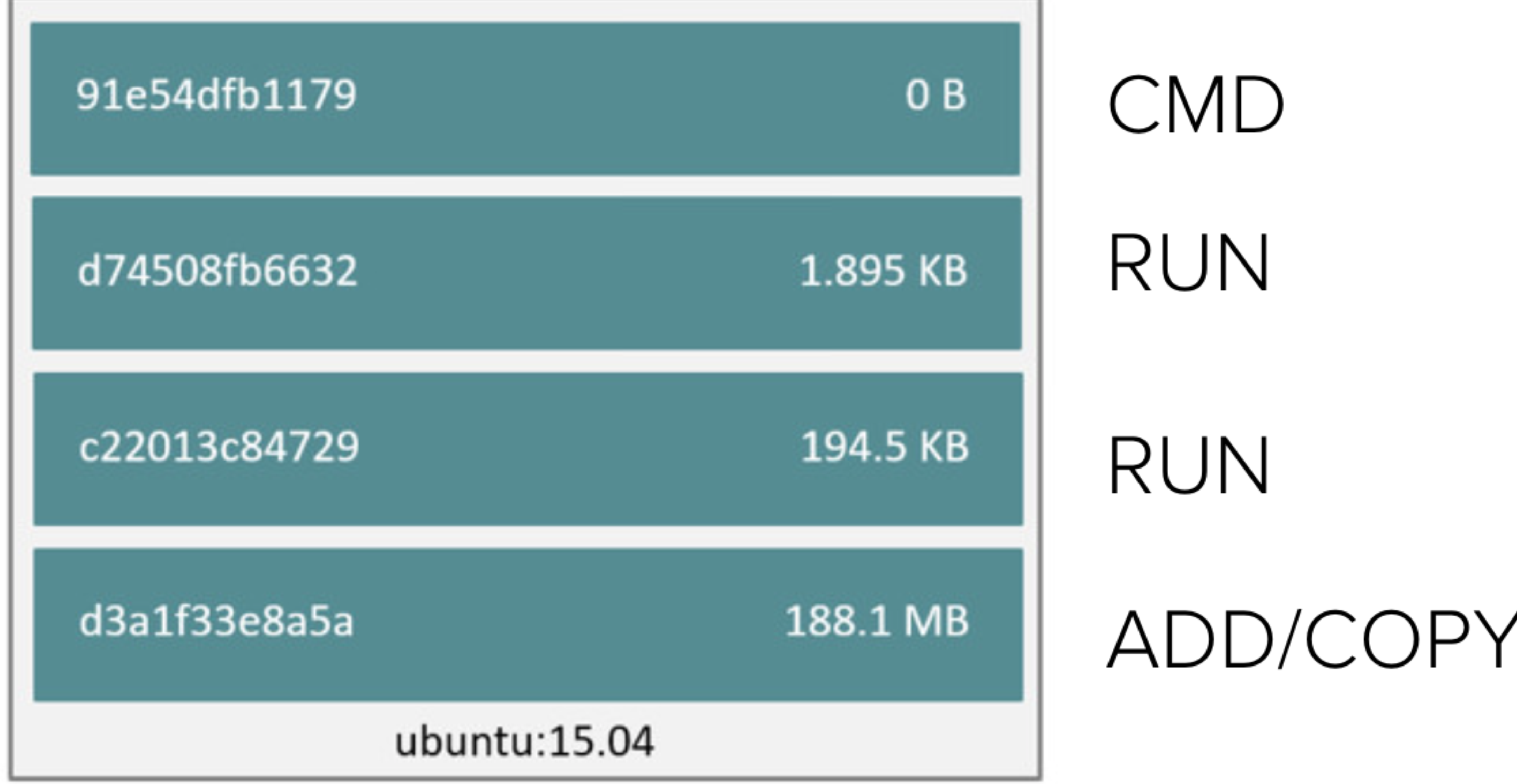
Docker Images



Итоговый слой, доступный для чтения и записи. Монтируется при запуске контейнера.

Слои образа доступны только для чтения

Docker Images



Image

Dockerfile

Абстрактный Dockerfile:

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

INSTRUCTION arguments
INSTRUCTION multiline \
            arguments
```

Инструкция: FROM

Инструкция FROM указывает базовый образ, на основе которого мы строим свою сборку:

```
FROM <image>[:<tag>]
# <image> - имя базового образа
# <tag> - опциональный атрибут указывающий на версию образа
```

Примеры:

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

Инструкция: LABEL

Инструкция `LABEL` задает метаданные для нашего образа:

```
LABEL <key>=<value> [<key>=<value> …]
# <key> - ключ
# <value> - значение
```

Примеры:

```
LABEL maintainer="user@example.org" version="0.2.1-8d2095e3ce"
```

Инструкция: COPY

Инструкция `COPY` копирует файлы из контекста в образ:

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

# или

COPY ["<src>",... "<dest>"]

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

Примеры:

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

ADD или COPY?

ADD:

- Больше магии
- Умеет работать с Remote URL
- Не рекомендуется использовать

Примеры:

```
ADD http://example.org/app.tar.xz /src/app/
RUN tar -xJf /usr/src/things/big.tar.xz -C /src/app
RUN make -C /usr/src/app all
RUN rm -f /usr/src/things/big.tar.xz
```

ADD или COPY?

COPY:

- Простота и лаконичность
- Нет дополнительных возможностей по копированию (URL)

```
COPY app.tar.xz /src/app/
RUN tar -xJf /usr/src/things/big.tar.xz -C /src/app
RUN make -C /usr/src/app all
```

```
RUN mkdir -p /src/app \
    && curl -SL http://example.org/app.tar.xz \
    | tar -xJC /src/app \
    && make -C /src/tapp all
```

Инструкция: ENV

Инструкция `ENV` задает переменные окружения при сборке:

```
ENV <key> <value>

# <key> - имя переменной окружения
# <value> - присваиваемое значение
```

Примеры:

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

Инструкция: WORKDIR

Инструкция `WORKDIR` задает рабочую директорию при сборке:

```
WORKDIR <path>

# <path> - путь внутри контейнера
```

Примеры:

```
WORKDIR /app
```

Инструкция: VOLUME

Инструкция VOLUME позволяет указать точки для монтирования томов внутри образа:

```
VOLUME <dst> [<dst> …]

# <dst> - директория монтирования для volume'a
```

Примеры:

```
VOLUME /app /db /data

# или

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

Инструкция: EXPOSE

Инструкция EXPOSE позволяет указать порты, которые слушает сервис в запущенном контейнере:

```
EXPOSE <port>[/<proto>] [<port>[/<proto>] …]

# <port> - порт сервиса внутри контейнера
# <proto> - tcp или udp
```

Примеры:

```
EXPOSE 5000
EXPOSE 8080/tcp 3389/udp
```

Инструкция: RUN

Инструкция `RUN` задает команды, которые выполняются при **сборке** контейнера:

```
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"]
RUN ["myscript.py", "argument1","argument2"]
```

Инструкция: CMD

Инструкция `CMD` задает команду, которая выполняется при **старте** контейнера:

```
CMD <command>

# <command> - команда которая будет выполнена при старте контейнера
```

Примеры:

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

Инструкция: ENTRYPOINT

Инструкция ENTRYPOINT задает команду, которая (почти обязательно) выполняется при **старте** контейнера:

```
ENTRYPOINT <command>

# <command> - команда которая будет выполнена при старте контейнера
```

Примеры:

```
ENTRYPOINT exec top -b
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
ENTRYPOINT ["/bin/sh", "/docker-entrypoint.sh"]
```

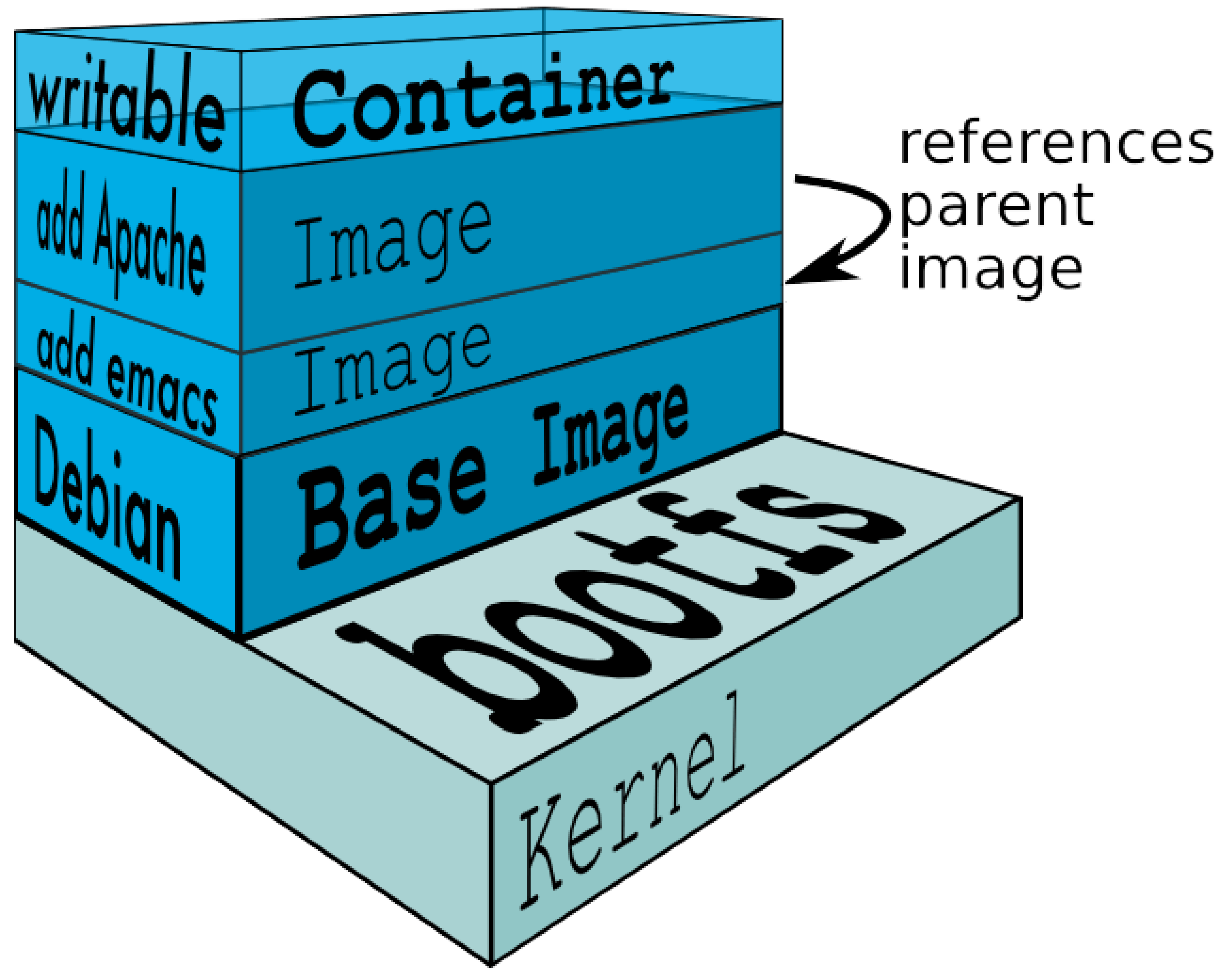
Dockerfile Reference (Partial)

```
ONBUILD <cmd>            # Задает команду, которая запускается
                         # при сборке образа на базе текущего

STOPSIGNAL <sig>         # Указывает сигнал, который посылается
                         # процессу при остановке контейнера

USER <username>          # Имя (ID) пользователя, от которого
                         # выполняются директивы RUN, CMD, ENTRYPOINT

ARG <string>             # Почти как ENV, но задает параметры
                         # только для docker build

HEALTHCHECK <cmd>        # Указывает команду, которой можно
                         # проверить состояние сервиса
```

Подробная документация на официальном сайте

Сборка образов Docker

Оптимизация сборки образов Docker



Оптимизация сборки образов Docker

- Каждая инструкция в `Dockerfile` это отдельный образ (слой)
- Инструкции кешируются в образах

Оптимизация сборки образов Docker

Заблуждение на счет лаконичного `Dockerfile`:

```
ENV myvar true                            # New image! 🐳

RUN apt-get install -y nginx              # New image! 🐳
RUN apt-get install -y php-fpm            # New image! 🐳
RUN apt-get install -y imagemagick        # New image! 🐳

COPY app /src/app                         # New image! 🐳

CMD ["/bin/cool-soft"]                    # New image! 🐳
```

Оптимизация сборки образов Docker

Оптимизируем количество слоев, ускоряя таким образом сборку:

```
ENV myvar true                                      # New image! 🐳
RUN apt-get install -y nginx php-fpm imagemagick    # New image! 🐳
COPY app /src/app                                   # New image! 🐳
CMD ["/bin/cool-soft"]                              # New image! 🐳
```

Порядок описания инструкций

Есть ли разница?

```
ENV myvar false
RUN apt-get install -y nginx php-fpm && imagemagick
COPY app /src/app
CMD ["/bin/cool-soft"]
```

или

```
ENV myvar false
COPY app /src/app
RUN apt-get install -y nginx php-fpm && imagemagick
CMD ["/bin/cool-soft"]
```

Кеширование сборки

Как работала упаковка новой версии кода в образ до изменений:

```
ENV myvar false                                         # Cache hit! ✅
RUN apt-get install -y nginx php-fpm && imagemagick     # Cache hit! ✅
COPY app /src/app                                       # Cache miss! ❌
CMD ["/bin/cool-soft"]                                  # Cache miss! ❌
```

Кеширование сборки

Как работала упаковка новой версии кода в образ до изменений:

```
ENV myvar false                                       # Cache hit! ✅
COPY app /src/app                                     # Cache miss! ❌
RUN apt-get install -y nginx php-fpm && imagemagick   # Cache miss! ❌
CMD ["/bin/cool-soft"]                                # Cache miss! ❌
```

Вывод: порядок инструкций важен!

Работа с кешем в образах

- Кеширование очень важно для реализации быстрых сборок
- `ADD`, `COPY` - файлы по умолчанию кешируются (в случае изменений файлов, кеш сбрасывается)
- Для остальных инструкций, включая `RUN`, проверяется только изменение параметров самой инструкции
  - `RUN apt-get -y update` не проверяет обновления постоянно, только в первый раз
- Пропустить и перестроить кэш можно командой

```
docker build --no-cache
```

Уменьшение размера образа

Все просто: удаляйте за собой архивы и временные файлы, которые остались во время билда.

```
COPY <filename>.zip <copy_directory>      # New image! 🐳
RUN unzip <filename>.zip                  # New image! 🐳
```

> ❗ В результирующем образе останется ZIP-архив

Уменьшение размера образа

```
COPY <filename>.zip <copy_directory>      # New image! 🐳
RUN unzip <filename>.zip                  # New image! 🐳
RUN rm <filename>.zip                     # New image! 🐳
```

❗ Остаются наследуемые образы. И конечно же, с ZIP-архивом

Уменьшение размера образа

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

👍 Создается только один образ, с распакованными данными из архива

Файл .dockerignore

- Содержимое директории указанной при запуске `docker build` попадает в build-контекст
- Лучшая практика - держать в директории минимум лишних файлов
- Если иначе никак, используйте `.dockerignore`, но это усложнит рабочее пространство проекта

Общие рекомендации

- Избегайте установки лишних пакетов и упаковки лишних данных в образы при сборке (build context bloating)
- Используйте связанные команды для `RUN`-инструкций
- Следим за последовательностью описания `Dockerfile`, избегаем cache miss
- Уменьшайте количество слоев
- Один контейнер - одна задача
- Чистите за собой
- Используйте multi-stage сборки (для компилируемых языков)

Секретный слайд

У `docker build` есть параметр `—squash`, который в финале собирает все слои в один (как будто инструкции `Dockerfile` были выполнены в одном слое), так что можно сильно не заморачиваться.

Packer собирает контейнеры Docker именно с этим параметром

Секретный слайд

Но есть свои тонкости:

1. Промежуточные слои не делятся между образами, только базовый образ (тот, который в `FROM`). В итоге можем тратить место
2. Поскольку слой один - он может быть большим и долго распаковываться и передаваться по сети (много слоев качаются параллельно)
3. При сборке будет потрачено гораздо больше места: все промежуточные слои в кэше + итоговый слой.
4. Как ни крути, итоговый образ не будет меньше базового :)

Секретный слайд

Когда использовать:

- Если мы создаем файлы в контейнере в процессе сборки (например, кэши PIP/NPM/Apt) и потом их удаляем.
- Если уже выстроили хорошую иерархию образов Docker:
    - Base
    - Runtime
    - Common dependencies
    - App

В остальном, лучше использовать Multi-stage

Пример Multi-stage сборки

```
# Создаем throw-away контейнер для сборки бинарного файла
FROM express42/ubuntu:16.04 as build-binary
RUN apt-get update -y && apt-get install make
ADD . /src
RUN cd src && make

# Создаем финальный образ контейнера с полученным бинарным файлом
FROM express42/main
COPY --from=build-binary /src/build/binary-app /usr/local/bin/binary-app
EXPOSE 9100
ENTRYPOINT /usr/local/bin/binary-app
```

Выбор базового образа

Пример Dockerfile

Дано:

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
express42/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" ]
```

Он же в *30 раз больше* самого приложения!

Пример Dockerfile: создаем образ с нуля

`scratch` - один из зарезервированных образов Docker, в котором ничего нет (пустой `Dockerfile`) В случае со `scratch`-образом, следующая инструкция создаст первый слой с файловой системой

```
FROM scratch

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

Пример Dockerfile: разница в размере образа

```
$ docker images

REPOSITORY            TAG                CREATED              SIZE
express42/hello-app   2.0                51 seconds ago       5.85MB
express42/hello-app   1.0                15 minutes ago       194MB
```

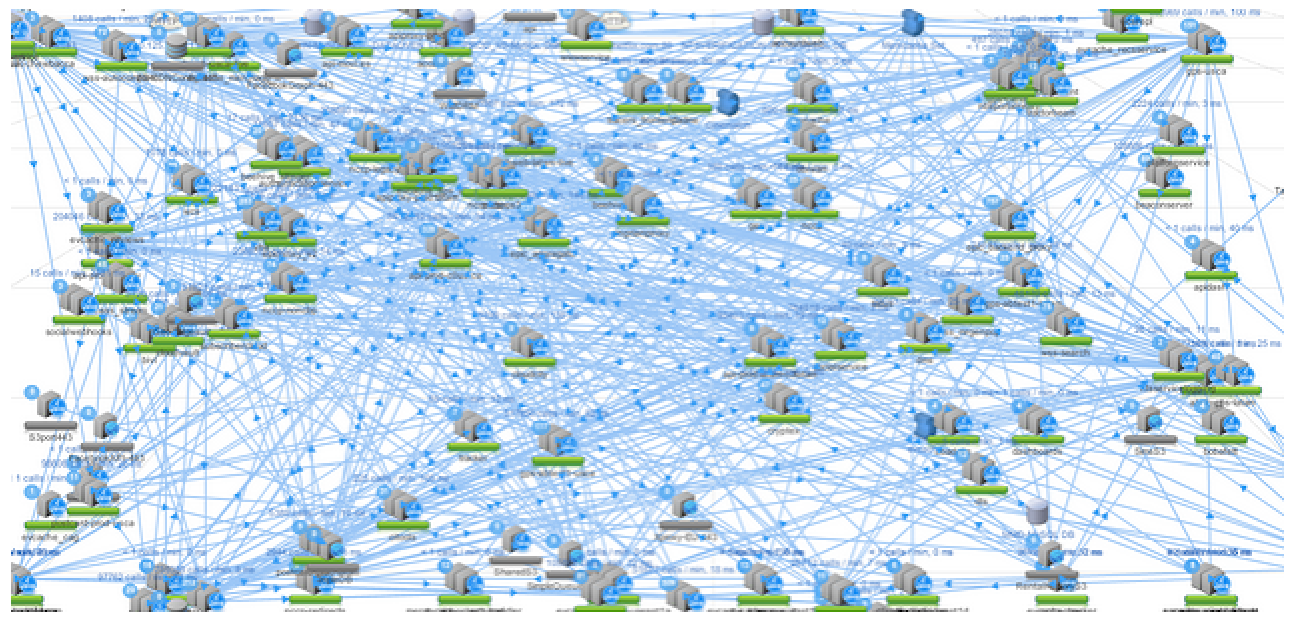
Итого

- Используем минимальные образы для запуска нашего кода:
  - Образ содержит только необходимое ПО для запуска приложения
  - Популярный базовый образ - `alpine`
  - Полезно иметь утилиты для дебага: `telnet`, `ping`
- Оптимизируем образы так:
  - выполняем очистку мусора при сборке
  - разделяем на стадии сборки и запуска (Multi-stage)
  - следим за порядком команд

Микросервисы

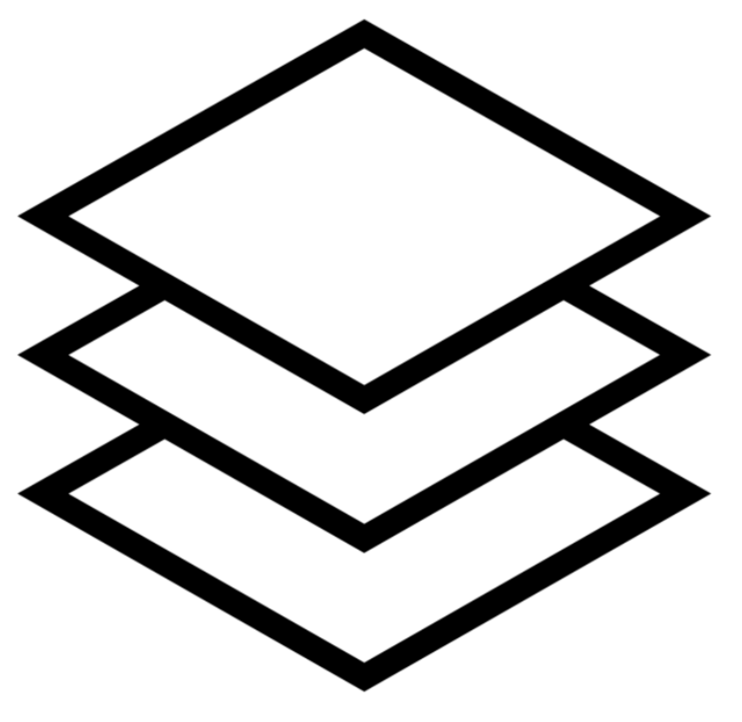
Микросервисы

Это - Netflix ...



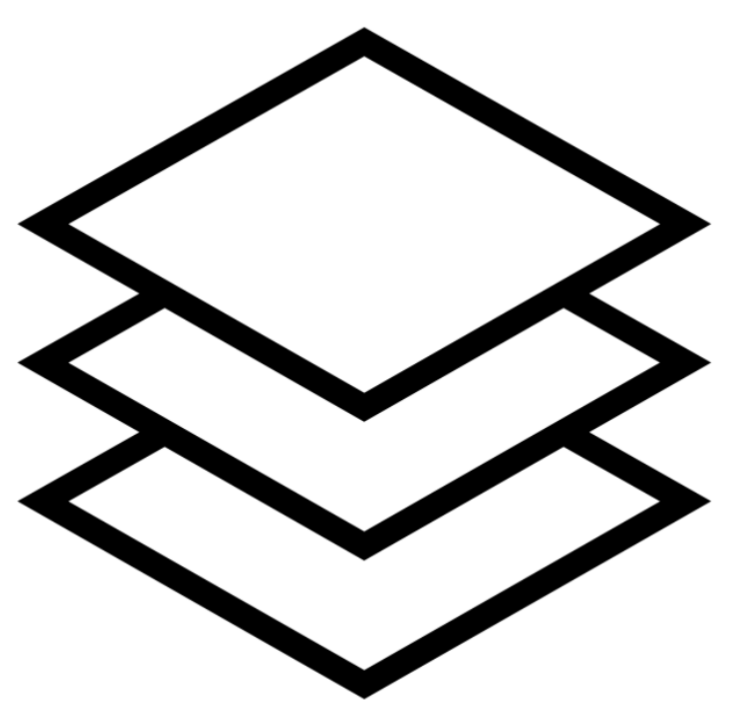
... А мы переходим к следующему слайду

Типичный проект



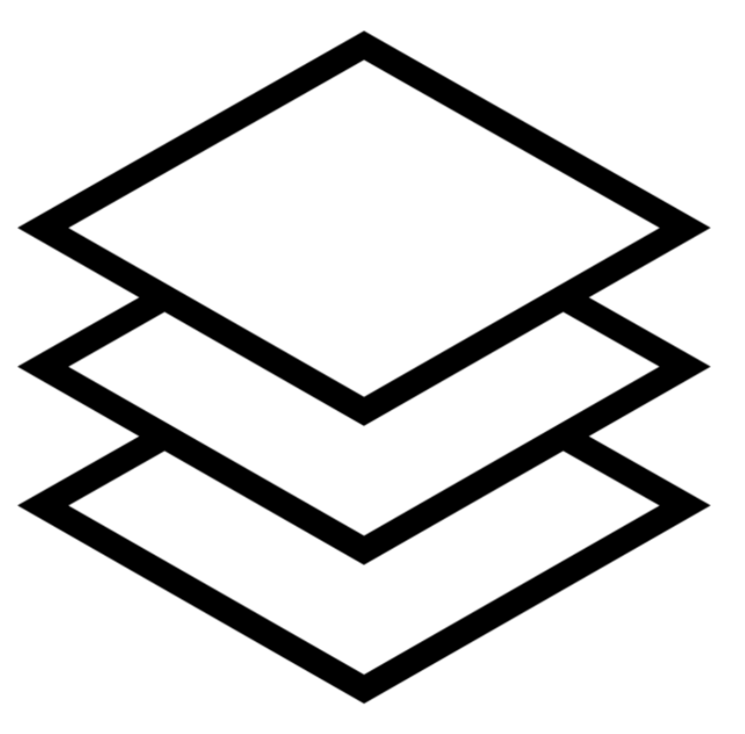
- 1/2/3 приложения
- И/или монолитное приложение
- Уровни:
  - Интерфейс пользователя aka front-end
  - Бизнес-логика aka back-end
  - Хранилище данных

Типичный проект: Внесение изменений



- Конфигурация ПО: Puppet/Chef/Ansible/Salt/...
- Деплой: RPM/Fabric/...
- Умеренный Configuration drift

Типичный проект: Поддержка

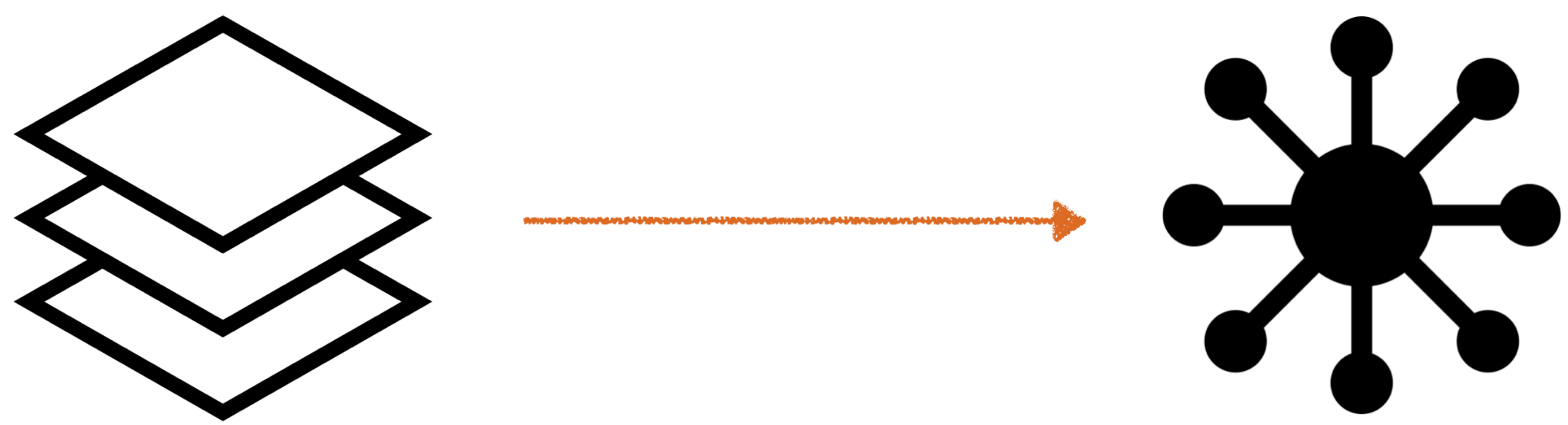


- Рассчитать потребление ресурсов просто
- Понятно, какой компонент и где запускать
- Простой мониторинг
- Сложно масштабировать отдельные модули

Типичный проект: Разработка

- *Долгие сборка, тестирование, развертывание*
- Высокая сложность (LOCs) и тесно связанные компоненты:
  - "Эффект бабочки" - изменение кода функции (или обновление библиотеки) может сломать приложение в неочевидном месте
  - "Эффект домино" - memory leak и подобные неприятности роняют все приложение
  - Невозможно сменить язык программирования или фреймворк
- Простое end-to-end тестирование (Запуск -> Selenium -> Profit)

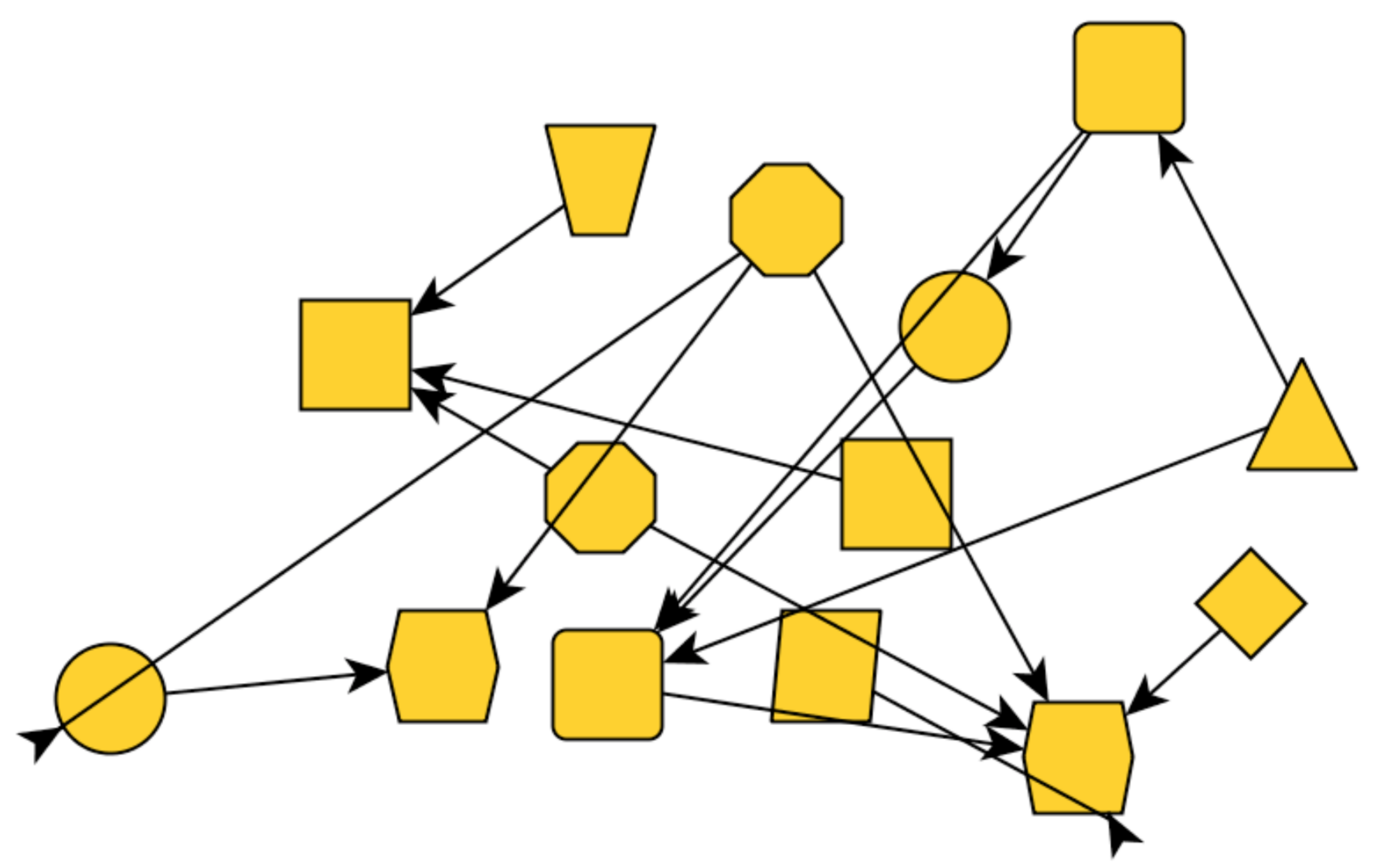
(Микро)сервисы



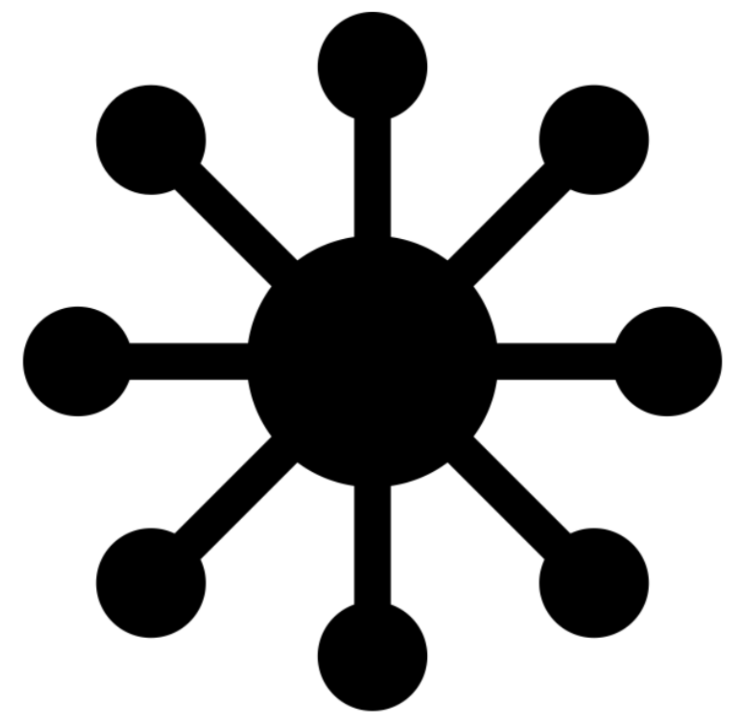
DB/back/front

Микросервисы

Микросервисы

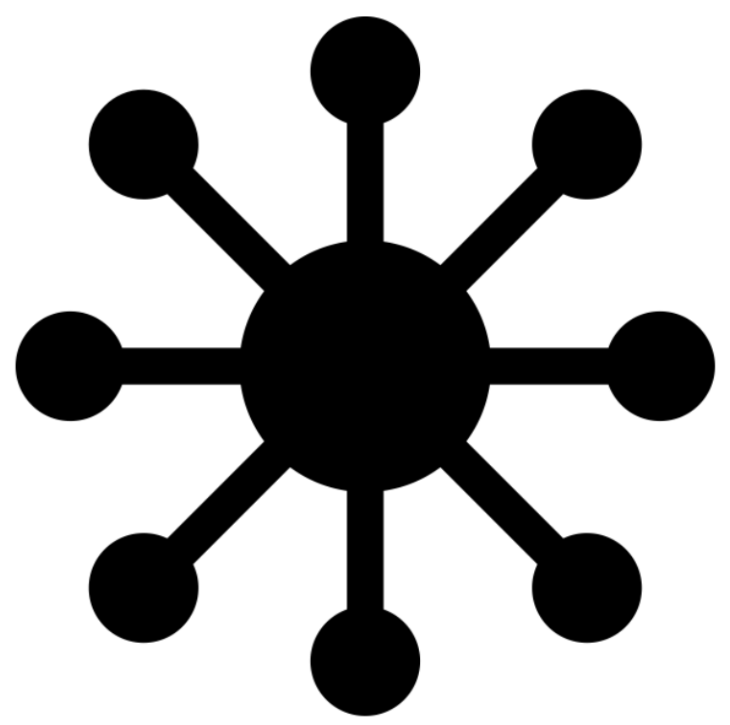


Микросервисы



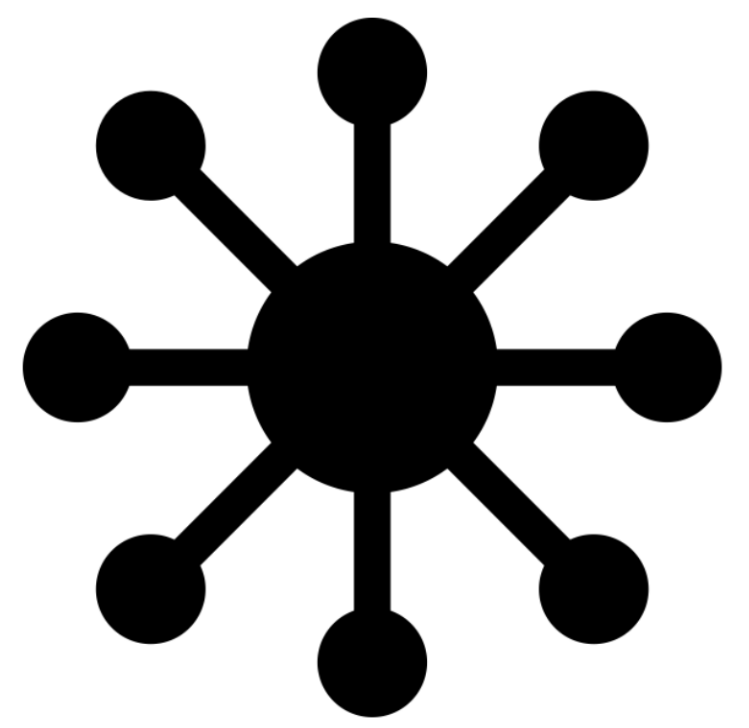
- 10/20/30 компонентов (со своими хранилищами данных)
- Могут иметь связи any-to-any
- Компоненты появляются и исчезают
- Нужен реестр сервисов (static, dynamic, service discovery)
- Для клиентов желательны единые точки входа (API Gateway)

Микросервисы: Внесение изменений



- Конфигурация ПО: Chef/Ansible
- RPM/Fabric/... – неудобно
- Configuration drift 🙀

Микросервисы: Поддержка

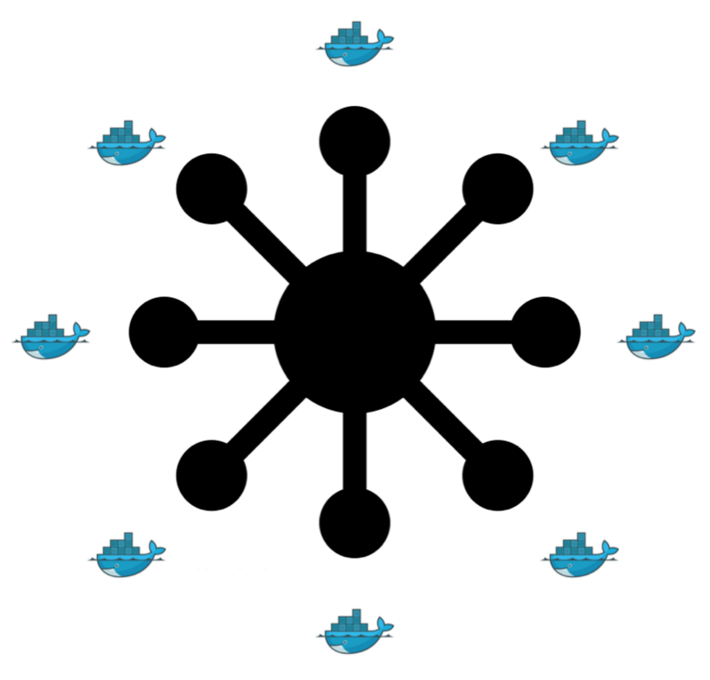


- Рассчитать потребление ресурсов сложно
- Вручную размещать приложения тяжело
- Управлять зависимостями - почти невозможно
- Поиск проблем и мониторинг ... тема отдельных нескольких лекций :)

Микросервисы: Разработка

- Быстрая и независимая разработка и развертывание
- Простое и быстрое тестирование изменений в коде
- Сложное тестирование end-to-end (оказывается, тестирование в Prod - не такая уж плохая идея)
- Необходимо взаимодействовать с реальным миром

Микросервисы и Docker



- Стандартизация
- Изоляция (в основном, это про зависимости, а не про безопасность)
- Декларативное описание среды
- Большая плотность на один хост (в сравнении с ВМ)

Особенности

- Приходится адаптировать приложения
- Работа с legacy
- Для эффективного использования нужно много других сущностей:
  - Service discovery (Consul, ZooKeeper, ...)
  - Балансировщики нагрузки или API Gateway
  - CI/CD (Jenkins, GitLab, ...)
  - Оркестраторы (Kubernetes, Nomad, ...)
  - Registry (Docker Registry, Nexus, Artifactory)
  - Системы отладки и мониторинга (Zipkin, Prometheus, ELK, ...)

Особенности запуска приложений в Docker-среде

- Один репозиторий `->` одно приложение `->` много развертываний
- Четко объявляйте свои зависимости и прописывайте их в `Dockerfile`
- Управляем базовой конфигурацией через переменные окружения
- Считайте все сторонние сервисы внешними ресурсами, которые могут вести себя непредсказуемо

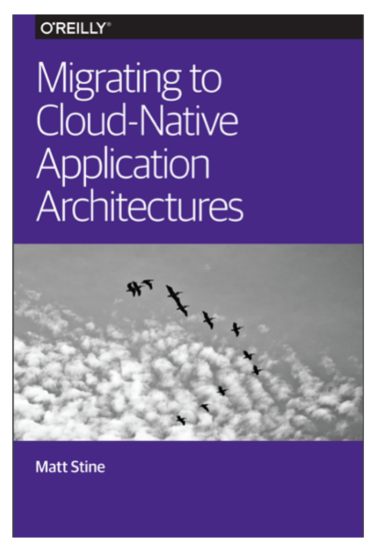
Особенности запуска приложений в Docker-среде

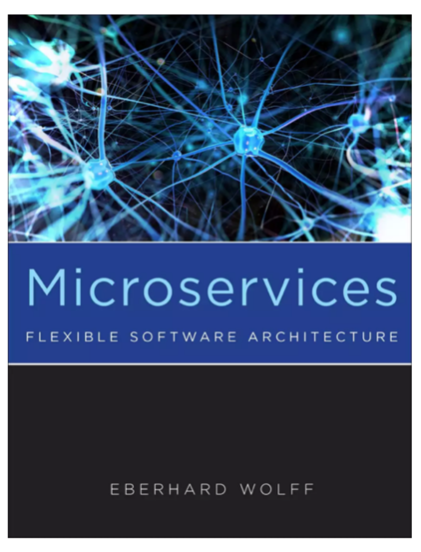
- Проектируйте приложение как один или несколько процессов не сохраняющих внутреннее состояние (stateless)
- Учитывайте взаимодействие с приложениями по сети (при разработке, автоматическом тестировании и развертывании)
- Доверьте обработку логов Docker-у, не пишите лог-файлы самостоятельно, пишите в `stdout`
- Постарайтесь сделать так, чтобы не использовать `root` для запуска и работы приложения.

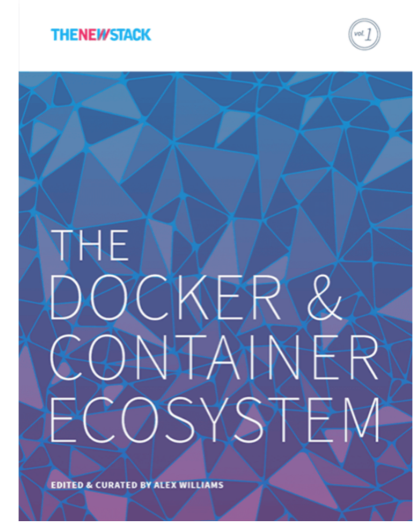
Дополнительные материалы

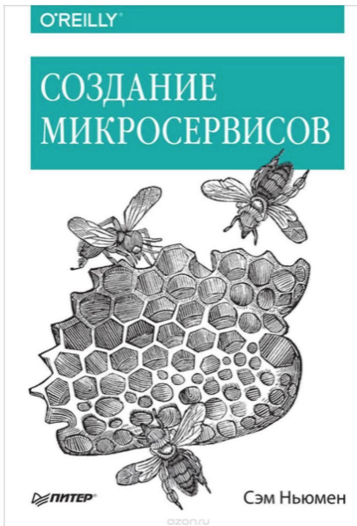
- Статья Testing in Production, the Safe way
- Статья Testing Microservices, the Sane way
- Основные паттерны для разработки микросервисов на microservices.io
- Описание паттерна DB per Service
- Методология для создания сервисных приложений
- NGINX Blog: Introduction to microservices

Дополнительные материалы









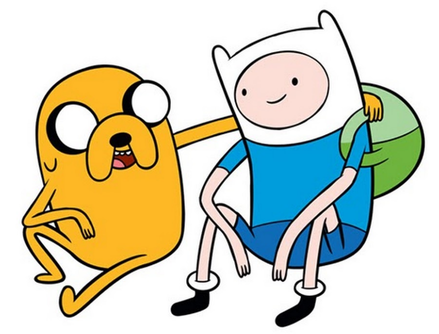
Дополнительные материалы

Бесплатные книги от O'Reilly:

- Для инженеров
- Для разработчиков
- Для безопасников, инженеров и разработчиков

Дополнительные материалы: Образы Docker

- Best practices от Docker
- Отчуждаемый от IDE линтер
- Test framework для Dockerfile



Спасибо за внимание!

Время для ваших вопросов!