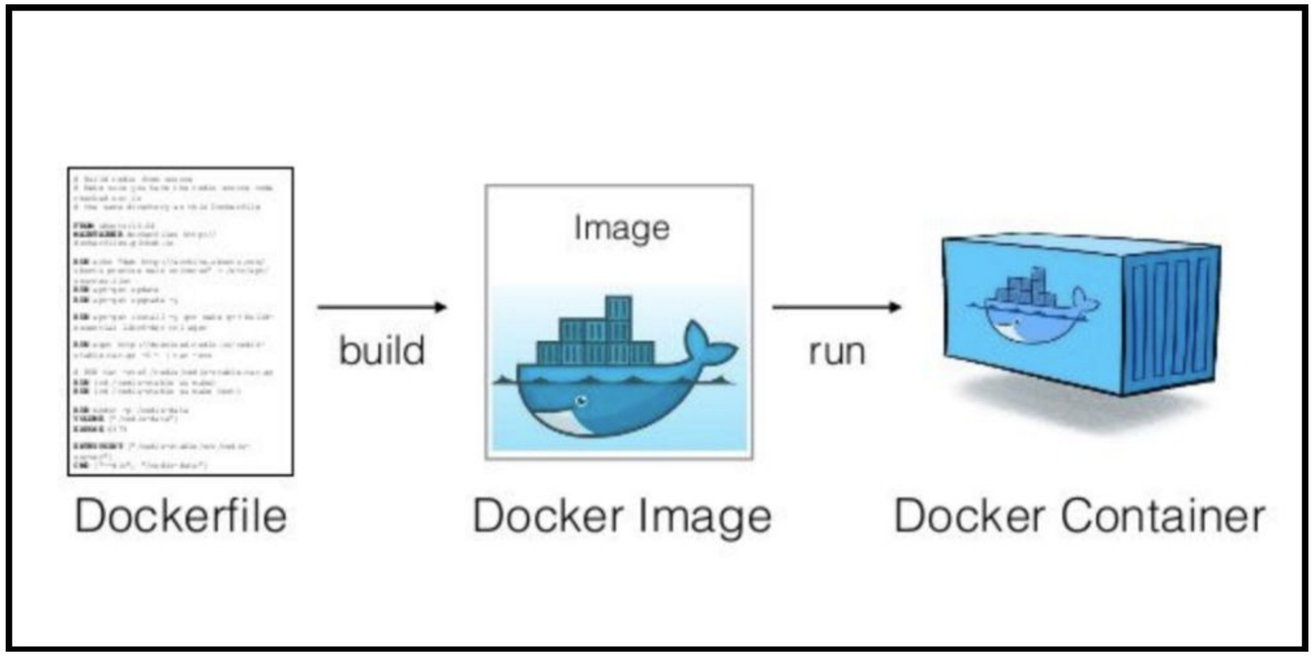
Skillbox

Основные инструкции Dockerfile

Основные инструкции Dockerfile

Dockerfile — это файл, содержащий набор инструкций, следуя которым Docker будет собирать образ контейнера.
Этот файл содержит описание базового образа, который будет представлять собой исходный слой образа



Best practice создания Dockerfile

Образ Docker состоит из слоёв, каждый из которых представляет инструкцию Dockerfile. Слои уложены друг на друга, и каждый из них представляет собой дельту изменений от предыдущего слоя
Например, Dockerfile:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

- FROM создаёт слой из ubuntu:18.04 образа Docker
- COPY добавляет файлы из текущего каталога вашего Docker-клиента
- RUN строит ваше приложение с помощью make
- CMD указывает, какую команду запускать в контейнере

Какие ещё бывают опции?

- LABEL — описывает метаданные. Например, сведения о том, кто создал и поддерживает образ
- ENV — устанавливает постоянные переменные среды
- ADD — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы
- WORKDIR — задаёт рабочую директорию для следующей инструкции
- ARG — задаёт переменные для передачи Docker во время сборки образа
- ENTRYPOINT — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются
- EXPOSE — указывает на необходимость открыть порт
- VOLUME — создаёт точку монтирования для работы с постоянным хранилищем

Основные правила создания Dockerfile

1. Create ephemeral containers
2. Understand build context
3. Pipe Dockerfile through
4. Exclude with .dockerignore
5. Use multi-stage builds
6. Don’t install unnecessary packages
7. Decouple applications
8. Minimize the number of layers
9. Sort multi-line arguments
10. Leverage build cache

Create ephemeral containers — создавайте эфемерные контейнеры

Сгенерированные вами image через dockerfile, должны соблюдать свойство эфемерности

В данном случае под «эфемерным» мы подразумеваем, что контейнер может быть остановлен и уничтожен, затем перестроен и заменён с минимальными настройками

Для сохранения преимущества контейнеров:

1. Не надо хранить данные внутри контейнера
2. Не надо дробить доставку приложений
3. Не надо создавать большие образы
4. Не надо использовать однослойные контейнеры
5. Не надо создавать образы из запущенных контейнеров — не применяйте команду docker commit
6. Не надо использовать только тег latest
7. Не надо выполнять в контейнере более одного процесса
8. Не надо хранить учетные данные в образе
9. Не надо запускать процессы от имени root
10. Не надо полагаться на IP-адреса

Understand build context — понимание контекста сборки

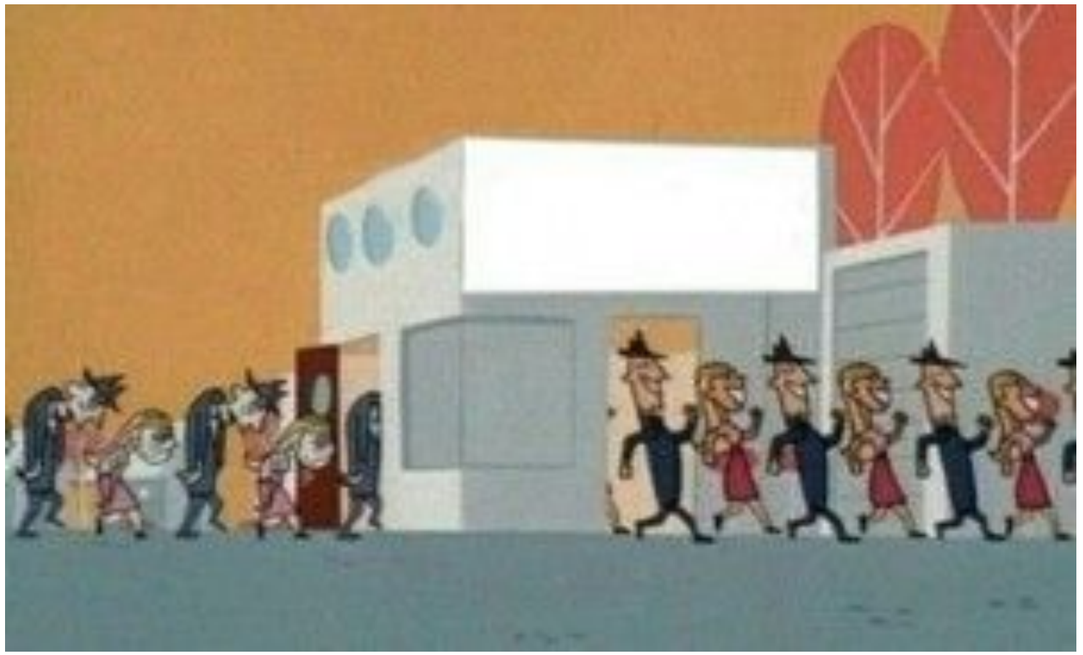
Когда вы вводите docker build команду, текущий рабочий каталог называется контекстом сборки

```
mkdir myproject && cd myproject
echo "hello" > hello
echo -e "FROM busybox\nCOPY /hello /\nRUN cat /hello" > Dockerfile
docker build -t helloapp:v1 .
```

Pipe Dockerfile through — конвейер Dockerfile через стандартный ввод

Docker может создавать образы, передавая Dockerfile по конвейеру через стандартный ввод с локальным или удалённым контекстом сборки

```
echo -e 'FROM busybox\nRUN echo "hello world"' | docker build -
```



Exclude with .dockerignore — исключить с помощью .dockerignore

Чтобы исключить файлы, не относящиеся к сборке (без реструктуризации исходного репозитория), используйте .dockerignore файл

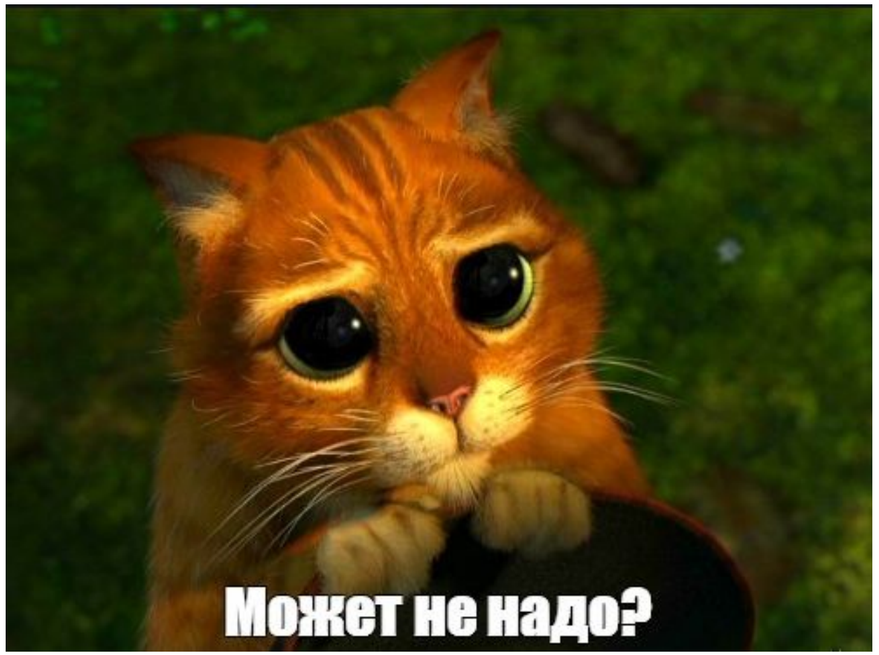
Этот файл поддерживает шаблоны исключения, аналогичные .gitignore файлам

Use multi-stage builds — используйте многоступенчатые сборки

Многоступенчатые сборки позволяют значительно уменьшить размер конечного изображения, не пытаясь уменьшить количество промежуточных слоёв и файлов

```
FROM golang:1.11-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

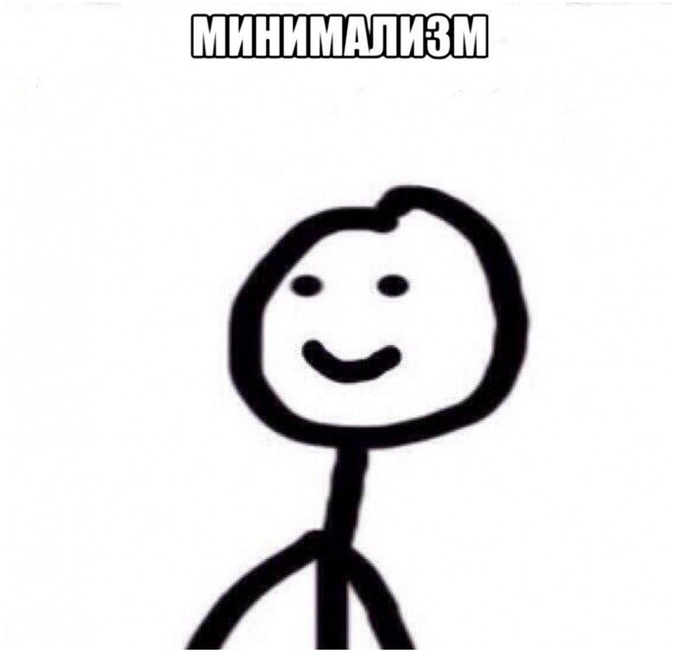
Don’t install unnecessary packages — не устанавливайте ненужные пакеты



Decouple applications — разделяйте приложения

У каждого контейнера должна быть только одна проблема!

Minimize the number of layers — минимизируйте количество слоёв



Sort multi-line arguments — сортируйте многострочные аргументы

Вот пример с buildpack-deps изображением:

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion \
  && rm -rf /var/lib/apt/lists/*
```

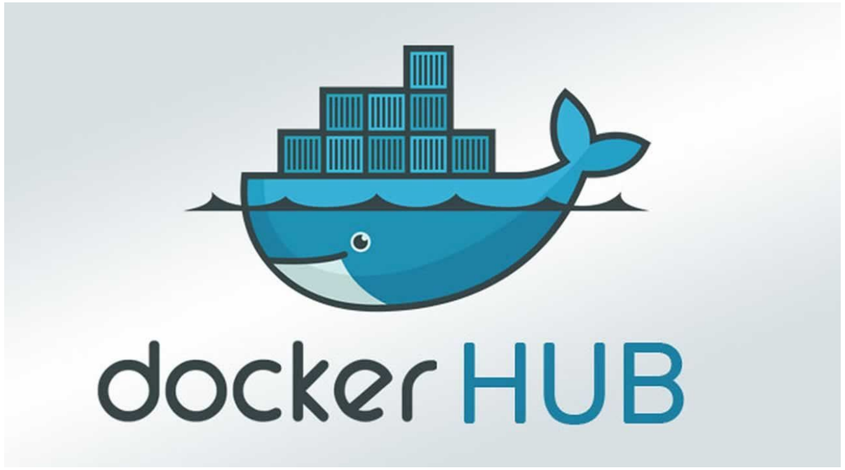
Leverage build cache — используйте кеш сборки

Основные правила:

- начиная с родительского образа, который уже находится в кэше, следующая инструкция сравнивается со всеми дочерними образами, полученными из этого базового образа, чтобы увидеть, было ли одно из них построено с использованием той же самой инструкции. В противном случае кеш становится недействительным
- в большинстве случаев достаточно просто сравнить инструкцию в Dockerfile с одним из дочерних изображений. Однако некоторые инструкции требуют дополнительного изучения и объяснения
- для инструкций ADD и COPY проверяется содержимое файла (ов) в изображении, и для каждого файла вычисляется контрольная сумма. Время последнего изменения и последнего доступа к файлу (ам) не учитывается в этих контрольных суммах. Во время поиска в кеше контрольная сумма сравнивается с контрольной суммой в существующих образах. Если что-либо изменилось в файле (ах), например, содержимое и метаданные, то кеш становится недействительным
- помимо ADD и COPY команд, проверка кеша не смотрит на файлы в контейнере , чтобы определить соответствие кеша. Например, при обработке RUN apt-get -y update команды, файлы, обновлённые в контейнере, не проверяются, чтобы определить, существует ли попадание в кеш. В этом случае для поиска совпадения используется только сама командная строка

Создание образов с помощью BuildKit

Управление image



Создадим базовый image

Создадим файл dockerfile:

```
FROM scratch
ADD hello /
CMD ["/hello"]
```

Присвоим ему tag:

```
docker build --tag hello .
```

Запустим:

```
docker run --rm hello
```

Итоги

- Узнали, что такое Dockerfile
- Изучили опции Dockerfile
- Узнали, как создавать контейнер через Dockerfile

Спасибо
за внимание!