# 4. Создание образов Docker. Команды Dockerfile

В предыдущей главе мы рассмотрели основные концепции и базовое устройство контейнеров (containers). Контейнеры, поддерживаемые возможностями изоляции операционной системы Linux или, чуть реже, Windows Server, способы обеспечить легкую, скоростную виртуализацию с использованием общего ядра операционной системы и крайне эффективно разделить и изолировать вычислительные ресурсы мощного сервера или кластера, работающего в удаленном облачном центре данных. Главный инструмент для организации и запуска стандартных контейнеров — Docker.

Все зависимости контейнеров, их файлы и библиотеки упакованы в так называемый образ (image) контейнера. Образ с определенной меткой (tag) является неизменным (immutable) и гарантирует одинаковую работу контейнера и логики приложения или сервиса внутри него при переносе и перезапуске на любых кластерах и серверах. Образы хранятся в репозитории образов, самый популярный — это официальный репозиторий Docker Hub. Все это делает контейнеры идеальным способом переноса функциональности и зависимостей сложной распределенной системы между серверами, кластерами и провайдерами облачных вычислительных ресурсов.

Нам, как разработчикам, прежде всего интересно, как создавать новые образы контейнеров, в которых мы будем размещать свои приложения или микросервисы, а затем запускать контейнеры из этих образов в облаке. Управлять ими, как правило, мы станем с помощью Kubernetes. Давайте займемся этим.

В этой главе большое количество файлов и особенно много команд, выполняемых из терминала — все они доступны в репозитории с примерами книги ivanporty/cloud-docker-k8s, а все команды работают как часть непрерывной интеграции (CI) c помощью GitHub Actions — вы сможете увидеть их ежедневный запуск и результаты и сравнить со своими.

## Структура Dockerfile. Основные команды. Базовый образ

Создать свои собственные образы для запуска своих команд, приложений или микросервисов с помощью Docker чрезвычайно просто. Необходимо указать перечень зависимостей, файлов, библиотек и основного, базового образа в так называемом файле Dockerfile. Формат так прост и популярен, что его поддержку вы найдете в любых предпочитаемых вами редакторах и IDE, иногда с использованием расширений (plugins, или extensions для VS Code).

Вот основа любого файла Dockerfile:

# Это комментарий  
# Каждый файл Dockerfile должен начинаться с FROM  
FROM [базовый\_образ]  
  
[Команда|Инструкция] [аргументы]

Любой новый образ должен на чем-то основываться — как мы помним, контейнер работает в общей операционной системе, имея доступ лишь к ядру, и даже простейшие консольные приложения требуют базовых библиотек для вывода данных на консоль и работы с терминалом. Базовый образ — это обычно или некий набор файлов, отвечающий дистрибутиву Linux, или чуть более расширенный набор библиотек, инструментов и зависимостей для компиляции и запуска приложений для выбранного языка. Стоит еще раз вспомнить, что все версии и названия Linux, используемые для создания образов — это просто файлы с инструментами и библиотеками. Ядро операционной системы будет общим, доступным через систему выполнения контейнеров Docker.

Именно базовый образ указывает команда FROM, правила выбора образа такие же, как и при запуске образа командой docker run. Если не указывать версию вместе с меткой tag явно, это будет latest — обычно последняя, самая свежая версия образа.

Следующая распространенная команда — RUN. Она запускает команду уже внутри контейнера. Этих двух команд вполне достаточно, чтобы создать первый собственный образ (image):

# Используем полную версию Ubuntu как базовый образ  
FROM ubuntu  
# Любые команды Ubuntu теперь доступны для запуска RUN  
# Но они запускаются при построении образа, не для запуска контейнера  
RUN echo "привет мир!" > hello\_world

Мы сохраним этот файл в директории dockerfile/helloworld и попробуем построить на его основе образ контейнера. Сделаем это командой docker build:

$ docker build . -t helloworld  
Sending build context to Docker daemon 2.048kB  
Step 1/2 : FROM ubuntu  
 ---> 775349758637  
Step 2/2 : RUN echo "привет мир!" > hello\_world  
 ---> Running in 98b510ad11b3  
Removing intermediate container 98b510ad11b3  
 ---> eb795c9beae9  
Successfully built eb795c9beae9  
Successfully tagged helloworld:latest

Для запуска этой команды мы перешли непосредственно в директорию, где находится наш Dockerfile, указали расположение этого файла (текущая директория .) и, самое главное, название образа нашего контейнера (- t helloworld). Дополнительную версию после названия образа мы для простоты не включили, и по умолчанию такая команда всегда будет строить образ с версией helloworld:latest.

Как мы видим из напечатанного в консоли, наш новый образ создается в два этапа — сначала скачивается и используется базовый образ Ubuntu (он, конечно же, будет скачан только один раз и после этого сохраняется в кэше вашей машины для ускорения процесса), а затем запускается команда, которая создает простой файл с эпохальной фразой «привет мир!».

Давайте запустим созданный собственными руками новый образ! Начнем с интерактивного режима с терминалом (-it) и посмотрим, что у нас есть в нашей файловой системе:

$ docker run -it helloworld  
root@68ec78485349:/# ll  
total 76  
drwxr-xr-x 1 root root 4096 Nov 7 17:53 ./  
drwxr-xr-x 1 root root 4096 Nov 7 17:53 ../  
-rwxr-xr-x 1 root root 0 Nov 7 17:53 .dockerenv\*  
drwxr-xr-x 2 root root 4096 Oct 29 21:25 bin/  
drwxr-xr-x 2 root root 4096 Apr 24 2018 boot/  
drwxr-xr-x 5 root root 360 Nov 7 17:53 dev/  
drwxr-xr-x 1 root root 4096 Nov 7 17:53 etc/  
-rw-r--r-- 1 root root 21 Nov 7 17:51 hello\_world  
…  
  
root@68ec78485349:/# cat hello\_world   
привет мир!  
root@68ec78485349:/# exit  
exit

Как мы видим, наш новый образ успешно запущен, контейнер работает, файловая система взята из базового образа Ubuntu и созданный в процессе построения образа файл hello\_world на месте и содержит именно то, что мы хотели.

Однако если мы попробуем просто запустить контейнер, он тут же закончит свою работу:

$ docker run helloworld  
$

Дело в том, что команда RUN просто исполняет указанные ей инструкции при построении образа, в нашем случае создавая файл или запуская любые другие команды, однако после построения образа она вызываться уже не будет.

Чтобы указать команду, которая будет выполняться после запуска контейнера из образа image, используется команда CMD или ENTRYPOINT. Добавим их и создадим новый файл Dockerfile в папке helloworld-loop. Вместо создания файла в процессе построения образа скопируем файл и скрипт для его печати командой COPY.

Вот наш скрипт для печати содержимого файла в цикле:

#!/bin/bash  
while true; do date; cat hello\_world; sleep $1; done

Скрипт будет печатать содержимое нашего файла в цикле, добавляя текущее время, в промежутке делая паузу. Размер паузы передается в параметре. Перенесем все нужные нам зависимости внутрь нового образа:

# Используем полную версию Ubuntu как базовый образ  
FROM ubuntu  
  
# Поменяем рабочую директорию на более удобную  
WORKDIR /opt/helloworld  
  
# Скопируем нужные нам для работы контейнера файлы в образ  
# Обратите внимание — ./ отвечает директории, указанной командой WORKDIR  
COPY hello\_world ./  
COPY print\_loop.sh ./  
  
# Команда CMD или ENTRYPOINT исполняется при запуске контейнера  
# Сначала идет команда, затем список аргументов. У нас — длина паузы.  
CMD ["/opt/helloworld/print\_loop.sh", “2”]

Здесь у нас целая гроздь новых команд Dockerfile, все они тем не менее чрезвычайно просты и логичны:

* WORKDIR — меняет рабочую директорию в файловой системе контейнера. Для образности представьте эту команду в виде обычной mkdir.
* COPY — копирует файл из директории, в который вы запустили команду docker build, в файловую систему контейнера. Обычно самая полезная и часто используемая команда для переноса исходного кода и библиотек в контейнер. Обратите внимание, что по умолчанию COPY переносит файлы в корень файловой системы, после команды WORKDIR — в эту новую директорию, а еще вы можете указать ей абсолютный путь файловой системы, куда следует поместить файлы.
* CMD (или ENTRYPOINT) — команда, которая будет выполняться после запуска контейнера. Основная форма — массив в квадратных скобках, где указывается команда и ее аргументы. Мы просто запустим свой shell-скрипт. Ему требуется параметр, мы его передаем в том же массиве. Разница между командами CMD и ENTRYPOINT не так велика, основная разница в том, что аргументы для CMD чуть проще изменять при запуске контейнера. Детали легко найти в документации. Есть еще один формат этих команд — исполнение напрямую оболочкой системы shell, в этом случае следует просто указать команду целиком, без массива и кавычек. Однако использованная нами только что форма записи более гибкая и обычно предпочтительнее.

Повторим построение образа helloworld уже на основе нашего нового Dockerfile и посмотрим, что теперь получается при запуске контейнера из этого образа:

$ docker build . -t helloworld  
…  
Step 2/5 : WORKDIR /opt/helloworld  
 ---> Using cache  
 ---> f35f404f3440  
Step 3/5 : COPY hello\_world ./  
 ---> Using cache  
 ---> 689899f448f4  
Step 4/5 : COPY print\_loop.sh ./  
 ---> b04eda22b54c  
Step 5/5 : CMD ["/opt/helloworld/print\_loop.sh", "2"]

Как видно, к построению образа добавились наши новые шаги. Запустим новый контейнер:

$ docker run helloworld  
Fri Nov 8 22:40:12 UTC 2019  
привет мир!   
Fri Nov 8 22:40:14 UTC 2019  
привет мир!  
Fri Nov 8 22:40:16 UTC 2019  
...

Теперь в нашем образе находится по большому счету настоящее приложение — оно запускается и печатает в цикле информацию. Так как наш цикл бесконечный, остановить контейнер командой терминала exit не получится — тут пригодятся команды docker ps и stop, которые мы как раз применяли в прошлой главе.

Только что узнанных команд на удивление хватает для построения реальных образов контейнеров. Мы вполне можем перенести свое приложение и его ресурсы в контейнер и запустить его при начале работы контейнера. Теперь давайте посмотрим, как создавать образы для реальных приложений и языков программирования.

## Создание образов для приложений Java, Go, Node.js

Основная задача образа контейнера image — обеспечить упаковку всех необходимых зависимостей для беспроблемного переноса запускаемого в контейнере приложения или микросервиса между любыми серверами и провайдерами облака. В случае реальных, написанных нами программ это означает, что мы должны удостовериться, что все виртуальные машины Java, зависимости скомпилированного приложения, необходимые ему ресурсы правильно сохранены в образе контейнера и смогут запускаться на любой системе, совместимой с контейнерами.

Обычная проблема при создании образа — копирование бинарного файла с программой или сервисом, несовместимым со стандартами Linux, особенно для таких языков, как Go или С++. К примеру, собрав приложение Go на своем ноутбуке Mac, вы не сможете перенести его в контейнер — внутри контейнера действуют стандарты Linux, и ваше приложение не запустится, несмотря на то что среда запуска контейнеров (container runtime) Docker работает на том же самом ноутбуке.

Лучшее решение в этом случае — компилировать и собирать (build) приложение как часть построения образа image инструкциями Dockerfile. В этом случае все происходит непосредственно внутри операционного ядра контейнера, и полученный образ будет совместим с любыми стандартными средами запуска контейнеров, в том числе в коммерческих провайдерах облака.

Что же использовать в качестве базового образа? Снова Ubuntu или, может быть, какую-то еще версию Linux, а затем скопировать туда все необходимые для компиляции и сборки языка программирования инструменты? Мы можем вздохнуть с облегчением — основная часть этой работы уже сделана. Упаковка приложений и сервисов в образы контейнеров стала настолько популярна, что все распространенные языки, их основные версии, нужные для работы с ними инструменты уже доступны на открытом репозитории Docker Hub. Нам остается подобрать нужную версию языка и систему сборки и скопировать файлы с кодом своего приложения.

### Java

Java — по-прежнему король языков программирования, когда речь заходит о больших корпоративных системах и серверных приложениях (enterprise). Ничего не мешает нам запускать сервисы, написанные на Java, внутри контейнеров Docker (кстати говоря, контейнеры в некотором роде уменьшили значимость виртуальной машины JVM и важность знаменитого слогана «написано однажды, запускается везде» — ведь сами контейнеры позволяют это сделать вообще для любого языка и библиотеки).

Самая популярная библиотека для построения RESTful сервисов и серверных приложений — без сомнения, Spring Boot, а система сборки — Maven. Давайте незамедлительно засучим рукава и в течение десяти минут упакуем сервис Java и Spring Boot в образ контейнера image, а затем запустим его.

Вот наш сервис, работающий с протоколом HTTP, — мы создали его с помощью удобного инструмента Spring Initializr, помогающего быстро создать заготовку приложения:

package com.porty.dockerfile;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
/\*\* Простейший HTTP сервис Java с использованием Spring Boot \*/  
@SpringBootApplication  
public class HelloJavaSpringBoot {  
 // запускает стандартный сервер Jetty, порт 8080  
 public static void main(String[] args) {  
 SpringApplication.run(HelloJavaSpringBoot.class);  
 }  
  
  
 @RestController  
 public static final class HelloWorldController {  
 // обрабатываем запрос к корневому пути /  
 @GetMapping("/")  
 public String helloWorld() {  
 return "Привет, это Java Spring Boot из контейнера!";  
 }  
 }  
}

Здесь все просто — мы используем стандартные инструменты библиотеки Spring Boot, чтобы создать приложение (SpringApplication.run) и обработать запросы к корневому маршруту /. Работать это приложение сможет на любой приличной версии Java, 8, 9, 11, 12, 13 (да, именно так, версий в Java теперь с избытком!). Располагаться этот файл для сборки проекта Maven должен в стандартной директории src/main/java.

Spring Initializr помог автоматически указать все необходимые нам зависимости и создал стандартный файл сборки для инструмента Maven. Многие детали опущены, но все можно найти на GitHub в примерах книги (репозиторий ivanporty/cloud-docker-k8s) и тем более в любом примере Spring Boot:

<?xml version="1.0" encoding="UTF-8"?>  
<project>  
  
<artifactId>hello-world</artifactId>  
 <packaging>jar</packaging>  
<name>Hello Java and SpringBoot</name>  
<version>1.0.0</version>  
...  
  
<dependencies>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
…

Мы указываем, что будем собирать свое приложение в виде архива JAR, назовем его hello-world версии 1.0.0. Остальное указывает, какие компоненты и библиотеки Spring Boot нам понадобятся.

Это все! Мы можем собрать и запустить этот сервис и посмотреть, как он отвечает на запросы через порт 8080 (это порт по умолчанию). Давайте теперь соберем и упакуем сервис в образ контейнера Docker. Такой стандартный и очень простой файл сборки Dockerfile по большому счету подойдет для большинства приложений Java:

# базовый образ - OpenJDK 11 и установленный Maven  
FROM maven:3.6.2-jdk-11  
  
# Соберем и запустим приложение в этой директории  
WORKDIR /app  
  
# Для сборки проекта Maven нужны исходные тексты программы  
# и непосредственно файл сборки pom.xml  
COPY pom.xml ./  
COPY src/ ./src/  
  
# Компиляция, сбока и упаковка приложения в архив JAR  
RUN mvn package  
  
# Запуск приложения виртуальной машиной Java из базового образа  
CMD ["java", "-jar", "/app/target/hello-world-1.0.0.jar"]

Вот что мы сделали для упаковки приложения Java и Maven:

* Для начала взяли базовый образ maven, его легко найти на репозитории Docker Hub. Есть различные версии, мы использовали версию, основанную на версии OpenJDK 11;
* Указали рабочую директорию (app) и скопировали туда файл сборки и код приложения из папки src. Прелесть системы Maven в том, что это все, что нам требуется, чтобы собрать практически любое приложение Java;
* Теперь, прямо в процессе построения образа нового контейнера, мы запустили компиляцию и упаковали приложение в архив JAR;
* Наконец, полученный архив запускается стандартной командой java -jar при старте контейнера. Плагин Spring Boot для Maven позаботится о том, чтобы все зависимости и библиотеки приложения были упакованы в один большой архив (fat jar).

Построим новый образ java-hello:

$ docker build . -t java-hello  
  
Step 1/6 : FROM maven:3.6.2-jdk-11  
 ---> 3b2476ab3d10  
Step 2/6 : WORKDIR /app  
 ---> 8ff277d1acce  
Step 3/6 : COPY pom.xml ./  
 ---> 96fae6707e9a  
Step 4/6 : COPY src/ ./src/  
 ---> 613b4042db7e  
Step 5/6 : RUN mvn package  
 ---> Running in 652db37dca12  
[INFO] Scanning for projects...  
Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.1.4.RELEASE/spring-boot-starter-parent-2.1.4.RELEASE.pom  
…  
[INFO] BUILD SUCCESS  
…  
Step 6/6 : CMD ["java", "-jar", "/app/target/hello-world-1.0.0.jar"]

Все шаги логичны и нам уже знакомы — но обратите внимание на то, что Maven будет заново скачивать все зависимости и библиотеки JAR из Интернета и компилировать приложение каждый раз при построении контейнера. В этом есть плюс — это «чистая» сборка, не зависящая от кэша и состояния вашей машины. Большой минус — постоянное скачивание библиотек и долгое время сборки. Чуть позже мы увидим различные решения этой проблемы.

Давайте запустим наш контейнер — не забудьте, это серверное приложение, и надо переадресовать необходимые порты на порты локальной машины командой -p:

$ docker run -p 8080:8080 java-hello  
…  
2019-11-08 INFO 1 --- Starting HelloJavaSpringBoot v1.0.0  
…  
Jetty started on port(s) 8080 (http/1.1) with context path '/'

Контейнер успешно запущен из созданного нами образа, и компоненты Spring Boot запустили встроенный HTTP-сервер, отвечающий по адресу 8080. Мы перенаправили этот порт на такой же порт своей собственной операционной системы и теперь можем проверить, что отвечает наш сервис (из отдельного терминала, так как терминал будет занят запущенным контейнером, можно также запустить контейнер в отсоединенном режиме с помощью ключа -d):

$ curl localhost:8080  
Привет, это Java Spring Boot из контейнера!

Образ создан и полностью автономен и работоспособен. Для запуска нашего нового Java-сервиса не нужно больше ничего — ни установленных виртуальных машин Java определенных версий, ни настроек пути PATH, ни дополнительных библиотек JAR. Контейнеры сдерживают свое обещание — собранный один раз образ с сервисом или приложением теперь может быть использован сколь угодно много раз для запуска этого сервиса на любых серверах, кластерах и других вычислительных ресурсах, не требуя никаких дополнительных настроек!

### Go

Язык Go стал намного популярнее за пределами создавшей его компании Google как раз на волне популярности контейнеров и управляющих ими систем, особенно Kubernetes. Именно на Go написаны Docker и Kubernetes, а также несколько известных платформ схожей направленности, таких как OpenShift. Go — намеренно простой язык, настолько простой, что полностью игнорирует ставшие такими привычными концепции программирования, как классы, объекты и исключения (exceptions). Для эффективности применяется компиляция в бинарный код и автоматическая сборка мусора, чтобы избежать печальных проблем с ручным управлением памятью в C++.

Писать на Go несложно, синтаксис похож на C, библиотеки гораздо выше уровнем, ну а сборка мусора сразу же избавляет от многих головных болей. Давайте напишем заготовку будущего RESTful микросервиса на основе сервера HTTP из стандартной библиотеки Go:

package main  
  
import (  
 "fmt"  
 "log"  
 "net/http"  
)  
  
func main() {  
  
 http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w, "Привет из контейнера с сервисом Go")  
 })  
  
 log.Fatal(http.ListenAndServe(":8080", nil))  
  
}

Всего несколько строк кода позволяют нам запустить HTTP-сервер (http.ListenAndServe), мы используем обычный порт 8080, а отвечать на запросы станем с корневого пути /, используя метод http.HandleFunc. Компилятор Go соберет для нас бинарную, быструю версию сервиса для необходимой нам платформы.

Мы хотим собрать и запустить этот микросервис для работы в контейнере и можем сделать это прямо в процессе построения образа, в файле Dockerfile:

# базовый образ — компилятор и все необходимое для Go 1.18  
FROM golang:1.18  
  
# Соберем и запустим приложение в этой директории  
WORKDIR /app  
  
# Скопируем код программы в файловую систему контейнера  
COPY main.go .  
  
# Соберем программу из исходного кода в файл hello-go  
RUN go build -o hello-go main.go  
  
# Запустим программу при запуске контейнера  
CMD ["./hello-go"]

Репозиторий Docker Hub содержит базовые образы, в которых есть все необходимое для сборки и запуска приложений Go определенной версии. Мы берем последнюю на момент написания версию 1.18, копируем файл с кодом программы, собираем ее и указываем, что при запуске контейнера входной точкой будет наша новая программа.

Запустим сборку образа контейнера и сразу же запустим его (в отсоединенном (-d, detach) режиме), не забыв переадресовать порт контейнера 8080 на свою операционную систему:

$ docker build . -t go-hello  
…  
Step 1/5 : FROM golang:1.18  
1.13: Pulling from library/golang  
...  
Step 5/5 : CMD ["./hello-go"]  
Successfully built d553c426de6c  
Successfully tagged go-hello:latest  
...  
  
$ docker run -d -p 8080:8080 go-hello  
…  
  
$ curl localhost:8080  
Привет из контейнера с сервисом Go

Как мы видим, собранный как часть образа контейнера бинарный микросервис Go прекрасно запускается и обслуживает порт 8080. Неважно, на какой операционной системе вы находитесь — Unix, Mac, Linux или Windows, Docker запустит минимальную виртуальную машину Linux, запустит все шаги по сборке образа вашего контейнера внутри пространства Linux, а затем безопасно и изолированно запустит из полученного образа новый контейнер и программу из бинарного кода Linux.

Во многом благодаря изоляции и идеальной переносимости контейнеров Go стал намного популярнее: не нужно больше думать о платформах, зависимостях и необходимости сборки приложения под каждую необходимую архитектуру — достаточно один раз скомпилировать приложение и упаковать его в образ контейнера. Это же верно и для других собираемых в бинарный код языков, таких как C++ и Rust.

### Node.js

Node.js — отличный способ применить свой опыт в JavaScript для разработки серверных приложений и тех же самых микросервисов. Это интерпретатор node и набор библиотек (модули Node.js, module), которые позволяют использовать асинхронную модель программирования, особенно подходящую для RESTful-сервисов и обработки сетевых запросов.

Если вы слышали о Node.js или пробовали работать с ним, то следующий код покажется вам прекрасно знакомым:

const http = require('http');  
  
const hostname = '0.0.0.0';  
const port = 3000;  
  
const server = http.createServer((req, res) => {  
 res.statusCode = 200;  
 res.setHeader('Content-Type', 'text/plain');  
 res.end('Привет от контейнера с сервером Node.js!');  
});  
  
server.listen(port, hostname, () => {  
 console.log(`Сервер запущен по адресу http://${hostname}:${port}/`);  
});

Мы используем модуль http и создадим локальный HTTP-сервер (createServer), привязав его к порту 3000 (почему не 8080? По какой-то причине 3000 гораздо популярнее в мире Node.js!). Останется его запустить функцией listen.

Нетрудно догадаться, что репозиторий Docker Hub содержит все необходимое для работы с основными версиями Node.js, в виде образа контейнера. Давайте используем версию Node.js 17 в качестве базового образа:

# базовый образ - npm, node и все остальное для Node.js 17  
FROM node:17  
  
# Исходный код  
COPY hello-world.js .  
  
# Точка входа - запуск кода интерпретатором node  
CMD ["node", "hello-world.js"]

Для Node.js у нас получился самый простой Dockerfile — мы просто копируем свой код внутрь файловой системы контейнера, а затем запускаем интерпретатор node при запуске контейнера, указав точку входа командой CMD.

Повторим уже хорошо известную последовательность действий — соберем свой новый образ и запустим его, не забыв, что номер порта у нас теперь 3000:

$ docker build . -t nodejs-hello  
…  
  
$ docker run -d -p 3000:3000 nodejs-hello  
Сервер запущен по адресу http://0.0.0.0:3000/  
…  
  
$ curl localhost:3000  
Привет от контейнера с сервером Node.js!

Интерпретатор node также успешно запущен внутри изолированного пространства контейнера. Мы сможем запустить сколь угодно много и какие угодно версии Node.js, с любыми комбинациями модулей, а контейнеры позаботятся об изоляции и легкой переносимости между любыми серверами и облаками.

Еще одно: не забывайте, что все использованные нами в этом разделе базовые образы с инструментами и инфраструктурой языков можно применить и для быстрых экспериментов или даже для непосредственной разработки.

Если вы не хотите устанавливать на свою рабочую машину «зоопарк» из технологий, пакетов и инструментов командной строки, вы всегда можете запустить один из основных образов с Docker Hub в интерактивном режиме (docker run -it) и использовать все нужные вам инструменты, компиляторы и зависимости в изолированном от своей машины пространстве процессов и отдельной файловой системе.

## Многоступенчатая сборка. Размер образа image

Итак, конструировать новые образы image для запуска контейнеров совсем несложно благодаря простоте и прозрачности формата Dockerfile и огромному выбору существующих базовых образов для работы с любыми версиями и вариантами языков распространенных языков программирования. Компилировать и собирать приложение из исходного кода оптимально прямо в процессе сборки образа, чтобы минимизировать влияние своей собственной локальной операционной системы, ее зависимостей и настроек своих компиляторов и инструментов и получить максимально «чистый», не зависящий ни от чего образ, способный запустить приложение на любых серверах.

Но есть одна неприятность. Давайте посмотрим на построенные только что нами образы, использовав команду docker images, — она выдаст нам полный список скачанных нами из Docker Hub и построенных собственными руками образов на своей машине и покажет нам *размер* каждого из них. Вот что мы увидим для только что собранных нами образов для популярных языков программирования:

REPOSITORY TAG SIZE  
nodejs-hello latest 908MB  
go-hello latest 810MB  
java-hello latest 662MB  
...

Казалось бы, в век скоростного доступа в Интернет и довольно низкой стоимости хранения данных размер примерно в один гигабайт не является чем-то шокирующим. Тем не менее это не совсем то, что *обещала* нам сама концепция контейнеров. Вспомним еще раз: контейнер использует ядро существующей операционной системы. Ему необходимы только используемые приложением дополнительные инструменты и библиотеки. Он должен запускаться практически мгновенно.

Но простейший веб-сервер размером в один гигабайт? Это чрезвычайно неэффективно, это снизит скорость запуска и масштабирования системы из множества контейнеров. В конечном итоге хранение данных в коммерческом облаке не бесплатно, и большое количество огромных образов скажется на стоимости облачных услуг.

Причина, конечно же, в базовом образе. Мы строим свое приложение прямо в «чистом» контейнере, в процессе сборки образа из инструкций Dockerfile, и это, без сомнения, правильно. Но после этого наше приложение или сервис «тащит» за собой все инструменты и библиотеки, необходимые только для сборки и компиляции, но не для его работы.

Решение — многоступенчатая сборка (multi-stage build), специально созданная Docker для подобных случаев. Этот тот случай, когда лучше сразу увидеть все в действии. Посмотрим, что мы можем сделать, чтобы уменьшить чудовищный размер образа go-hello со скомпилированным сервисом Go (который, вообще-то, скомпилирован в бинарный код Linux и должен иметь минимальный размер из всех языков!):

# первая ступень - компилятор и все необходимое для Go 1.17  
FROM golang:1.17 as builder  
  
# Соберем и запустим приложение в этой директории  
WORKDIR /app  
  
# Скопируем код программы в файловую систему контейнера  
COPY main.go .  
  
# Соберем программу из исходного кода в файл hello-go  
# Необходимо указать дополнительные флаги сборки Go  
RUN CGO\_ENABLED=0 GOOS=linux go build -a -o hello-go main.go  
  
# вторая ступень - спартанская версия Linux Alpine  
FROM alpine:3.15  
  
# Используем такую же рабочую директорию  
WORKDIR /app  
  
# Скопируем собранный бинарный код из первой ступени  
COPY --from=builder /app/hello-go .  
  
# Запустим программу при запуске контейнера  
CMD ["/app/hello-go"]

Многоступенчатая сборка — не что иное, как возможность использовать промежуточные, временные образы, использовать для них любые, не обязательно одинаковые базовые образы в процессе сборки и использовать файлы из предыдущих этапов, копируя их в следующий этап.

Именно это мы и проделали в своем новом Dockerfile, собрав образ со своим микросервисом в два этапа. Комментарии очевидны, можно только подвести итоги.

* Каждая ступень сборки начинается с инструкции FROM и указывает базовый образ только для этой ступени. Может быть произвольное число инструкций FROM, базовый образ в последней инструкции и будет окончательным для нового образа. В нашем примере это широко распространенная «спартанская» версия Linux Alpine размером около 5 мегабайт! Она популярна для встроенных приложений со строгими требованиями к ресурсам. В ней есть базовый набор инструментов Linux, достаточных для базовой отладки приложения.
* Каждой ступени можно присвоить имя (в нашем примере — builder) или же указывать ступень по номеру (начиная с нуля).
* Команда COPY позволяет копировать файлы из предыдущих ступеней (по имени или номеру) с помощью флага --from.
* Каждая ступень может заново указывать рабочую директорию, копировать свой набор файлом и быть совершенно самостоятельной.
* Для работы собранного сервиса Go в системе Alpine нужно собрать его специальным образом, с минимальными зависимостями (статическая сборка, за это в основном отвечает флаг CGO\_ENABLED), остальные детали можно найти в документации Go.

Соберем наш образ снова, используя усовершенствованный Dockerfile из двух ступеней, и посмотрим его размер:

$ docker build . -t go-hello  
...  
$ docker images | grep go-hello  
go-hello latest 12.9MB

Мы уменьшили образ в 80 раз! Теперь образ действительно соответствует девизу контейнера — быстрая, легкая виртуализация без огромных пакетов, инструментов и полной операционной системы. Запустив новый образ, мы сможем убедиться, что качество сервиса нисколько не пострадало от уменьшения размера образа в десятки раз.

Если вы совсем не планируете использовать терминал и оболочку shell, можно уменьшить наш образ до минимума. Пустой базовый образ scratch — это минимум того, что есть в Docker. В нем нет вообще ничего — что будет означать, как мы помним, просто доступ к ядру операционной системы. Для нашего сервиса Go этого достаточно. Если мы поменяем alpine на scratch и снова соберем образ, вот что у нас получится:

$ docker images | grep go-hello  
go-hello latest 7.39MB

Еще на 5 мегабайт меньше, по сути, это просто размер бинарного файла, собранного компилятором Go. Идеально для встроенных систем и ограничений в объемах данных, не забудьте только, что запуск в интерактивном режиме (it) и работа с терминалом внутри такого контейнера будут уже невозможны.

### Многоступенчатая сборка Java

Давайте посмотрим, что можно сделать для оптимизации размера образа (image) с нашим микросервисом на Java и Spring Boot (java-hello). Сразу скажем, что таких фантастических результатов, как с Go, достигнуть не получится — нам в любом случае понадобится виртуальная машина Java, ее библиотеки и все дополнительные JAR-файлы для работы Spring Boot. Но значительное уменьшение образа все равно возможно — снова применим двухступенчатую сборку и вместо пакета разработки (JDK) используем пакет запуска JRE (Java Runtime Environment), значительно меньший по размеру:

# первая ступень — базовый образ — OpenJDK 11 и установленный Maven  
FROM maven:3.6.2-jdk-11 as builder  
  
# Соберем и запустим приложение в этой директории  
WORKDIR /app  
  
# Для сборки проекта Maven нужны исходные тексты программы  
# и непосредственно файл сборки pom.xml  
COPY pom.xml ./  
COPY src/ ./src/  
  
# Компиляция, сборка и упаковка приложения в архив JAR  
RUN mvn package  
  
# Минимальная версия JRE, версия 11, открытая версия OpenJDK  
FROM openjdk:11-jre-slim  
  
# Используем такую же рабочую директорию  
WORKDIR /app  
# Скопируем архив JAR из первой ступени  
COPY --from=builder /app/target/hello-world-1.0.0.jar .  
  
# Запуск приложения виртуальной машиной Java OpenJDK 11  
CMD ["java", "-jar", "/app/hello-world-1.0.0.jar"]

Все инструкции нам уже хорошо знакомы по многоступенчатой сборке сервиса Go. Вторая ступень будет использовать минимальный образ JRE на базе OpenJDK, варианта Java с открытым исходным кодом, версии 11-slim — с минимально необходимым набором модулей (modules). Вы найдете этот обновленный Dockerfile в той же директории java-hello с исходным кодом и инструкциями Maven. Соберем образ заново:

$ docker build . -f Dockerfile-multistage -t java-hello  
…  
$ docker images | grep java-hello  
go-hello latest 218MB

Как видно, у нас получилось уменьшить размер образа «всего лишь» в три раза, но это огромный выигрыш. Можно уменьшить размер еще больше, найдя подходящую версию Java на базе Linux Alpine, обычно это более старая версия Java 8, впрочем, прекрасно работающая для большинства серверных приложений. Попробуйте это сделать в качестве небольшого упражнения.

Чуть другой подход можно применить для Node.js — там этап компиляции и сборки, по сути, отсутствует, но можно оптимизировать количество пакетов и инструментов, оставив только необходимое для запуска приложения, и значительно уменьшить окончательный размер образа своего сервиса.

## Репозитории образов. Метки, версии и latest

Все собранные нами в этой главе образы image до сих пор хранятся в нашей локальной файловой системе и совершенно недоступны для других членов вашей команды, для запуска в облаке или же для тестирования в системах непрерывной интеграции и развертывания CI/CD. В прошлой, обзорной, главе про Docker мы выяснили, что образы хранятся в доступных через Интернет репозиториях, с историей меток и изменений, подобно системе контроля версий кода, и сходно с главным репозиторием открытого кода GitHub главный репозиторий для образов Docker называется Docker Hub.

Что прекрасно, простое использование репозитория Docker Hub совершенно бесплатно. Конечно, это знаменитая модель freemium (начать пользоваться бесплатно, улучшить сервис за плату) — как только вам понадобится использовать Docker Hub в больших масштабах и для построения масштабных приложений, придется подписаться на платные услуги. Зачастую коммерческие облака предлагают свои собственные хранилища образов контейнеров, которые могут быть чуть выгоднее Docker Hub, если вы уже используете сервисы контейнеров (такие как Kubernetes).

Впрочем, для экспериментов и простых приложений бесплатных услуг Docker Hub более чем хватает. Надо лишь создать свою учетную запись. После этого нужно войти в нее со своей машины:

$ docker login

Давайте попробуем отправить образ с нашим сервисом Go (он самый маленький и быстрый) в репозиторий Docker Hub (push):

$ docker push go-hello  
..  
access.. denied

Ничего не вышло, в доступе было отказано. Дело в том, что по умолчанию, если не указывать учетную запись, все образы отправляются в стандартную библиотеку Docker Hub, к которой у нас нет доступа — туда помещаются только самые популярные, тщательно проверенные на безопасность образы. Метка для пользовательских образов должна включать в себя учетную запись в следующем формате — {учетная\_запись\_Docker}/имя образа:[необязательная версия].

Перестраивать образ заново нет необходимости — мы можем просто указать новую метку с помощью команды docker tag и отправить помеченный учетной записью образ в репозиторий Docker Hub:

$ docker tag go-hello {учетная\_запись\_Docker}/go-hello  
$ docker push {учетная\_запись\_Docker}/go-hello  
The push refers to repository [docker.io/{учетная\_запись\_Docker}/go-hello]  
eac13675ca89: Pushed   
37ce9121a810: Pushed  
...

Теперь все прошло успешно. Обновите страницу со списком образов, хранящуюся в вашей учетной записи, и вы увидите новый образ, только что отправленный в репозиторий.

Все образы, собранные нами в этой главе, в своих метках (tag) использовали только название, но никогда не указывали *версию*. Если версия не указывается, образ помечается версией latest — что просто означает, что именно этот образ был собран последним.

Казалось бы, в чем проблема? Вспомним еще раз, что запускаемый на основе образа контейнер обеспечивает максимальную переносимость и неизменность (immutability) системы. Так как уже созданный и собранный образ поменять нельзя, воссоздание системы на основе известных образов и версий становится тривиальной задачей — например, всегда важно воспроизвести производственное (production) окружение, чтобы понять причину сложной ошибки, которая происходит в условиях реальной эксплуатации, но не в среде разработки или тестирования (QA environment).

Метка latest же чрезвычайно подвержена постоянным изменениям, в том числе случайным. Любой образ, построенный без указания определенной версии, автоматически получает версию latest, и предыдущая версия образа просто исчезает. Более того, если при запуске контейнера образ с версией latest уже есть в кэше сервера или в кэше системы управления контейнерами, он не будет заново скачиваться, даже если образ с этой меткой в репозитории был обновлен (Kubernetes позволяет обойти это ограничение, но это надо делать явно, через дополнительные настройки системы).

Гораздо лучшая практика работы с метками — указание точных версий и по возможности использование одной версии только для одного образа, с автоматическим увеличением номера версии при изменении функциональности приложения в образе. Особенно хорошо для этого подходят *семантические версии* (SemVer, детали можно найти в Интернете). В общем случае они начинаются с версии 0.1.0 и всегда следуют формату X.Y.Z, где

* X — главная (major) версия, она увеличивается при больших изменениях функциональности и программных интерфейсов API, как правило, несовместимых с предыдущей главной версией;
* Y — дополнительная (minor) версия, увеличивается при появлении новой функциональности, полностью совместимой с предыдущей главной версией;
* Z - версия «патча» (patch), обычно прибавляют при исправлении мелких ошибок, без каких-либо новых возможностей системы, как еще называют такие исправления, «заплатки» (отсюда и слово patch).

Использование уникальной метки для всех образов гарантирует возможность воспроизведения поведения системы, контейнера и сохраняет список изменений в *каждой* когда-либо построенной версии сервиса, приложения, его компонента (микросервиса) и, в общем случае, всей сложной распределенной системы в целом.

В нашем случае лучше пометить образ так:

$ docker tag go-hello {учетная\_запись\_Docker}/go-hello:0.1.0

При дальнейшей разработке новые образы будут увеличивать версию (согласно схеме SemVer) и всегда соответствовать упакованной в образ функциональности приложения.

В некотором смысле все, что мы сказали о метке latest относительно наших собственных образов, верно и для базовых образов, которые мы указываем с помощью инструкций FROM. К примеру, какая версия Ubuntu или Alpine будет использована в инструкции FROM ubuntu|alpine? Последняя, но совершенно неизвестно, какая именно! Указывая точные версии вместо неопределенной версии, мы увеличиваем стабильность и предсказуемость своих образов.

## Альтернативы Dockerfile. Jib

Контейнеры стали заслуженно популярны, и мы видим, что построить образы для них несложно. Однако не все языки программирования одинаково хороши для работы с многоступенчатыми инструкциями Dockerfile, даже если использованы все рекомендованные практики для построения образов минимальных размеров и с минимальными зависимостями.

Яркий пример — приложения и сервисы Java и связанные с JVM языки, такие как Scala и Kotlin. Практически все они используют системы сборки Maven, Gradle и похожие на них (SBT) и все свои зависимости (библиотеки JAR) хранят и скачивают с центральных хранилищ, обычно Maven Central.

Написав свой многоступенчатый образ для Java, мы тем не менее полностью игнорируем кэш и локально доступные, уже скачанные библиотеки JAR. Это неизменная зависимость приложения, подписанная и надежно защищенная от изменений самим механизмом Maven Central. Мы же заново, раз за разом, полностью скачиваем все зависимости приложений через Интернет, делая процесс сборки приложения медленным и неэффективным.

Один вариант — просто скопировать весь кэш Maven или Gradle внутрь контейнера по время сборки приложения, но это, опять же, неэффективно — это могут быть тысячи библиотек, используемых другими приложениями.

Здесь поможет плагин Jib, специально созданный Google для оптимизации сборки образов Java-приложений. Он интегрируется с системой сборки Maven или Gradle, анализирует список зависимостей и создает специальный кэш (в мире Docker это часть образа называется слоем layer), который не меняется и используется заново каждый раз при последующей сборке проекта. Выигрыш в эффективности и скорости сборки образа потрясающий.

Добавим Jib в наш проект Java и Spring Boot — мы использовали там сборку Maven:

...  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>com.google.cloud.tools</groupId>  
 <artifactId>jib-maven-plugin</artifactId>  
 <version>1.8.0</version>  
 <configuration>  
 <to>  
 <image>{учетная\_запись\_Docker}/java-hello:0.1.0</image>  
 </to>  
 </configuration>  
  
 </plugin>  
...

Плагин объявлен в стандартной секции build/plugins. В конфигурации плагина мы также указываем стандартную метку для нашего образа — учетную запись, название и версию. Дальше остается только собрать образ нашего сервиса с помощью Jib:

$ mvn compile jib:build  
...  
[INFO] Total time: 13.127 s

Первая сборка займет некоторое время, но каждая последующая сборка будет очень быстрой и эффективной. Размер полученного образа java-hello также станет еще почти в два раза меньше — по умолчанию Jib использует сжатый базовый образ distroless, разработанный в Google.

Обратите внимание, что по умолчанию цель сборки jib:build всегда отправляет (аналог docker push) собранный образ в указанный репозиторий и указывать его нужно обязательно (что мы и сделали в специальной секции <image) — в противном случае произойдет ошибка, такая же, что произошла, когда мы попытались выполнить docker push, не указав своей учетной записи Docker Hub.

Но что, если вы хотите собрать образ на локальной машине, без отправки его в репозиторий? На этот случай пригодится другая команда — jib:dockerBuild, она построит образ с помощью Docker локально, без отправки в репозиторий:

$ mvn compile jib:dockerBuild  
…

Вот что мы получили, использовав Jib:

* Нам больше не нужно писать, оптимизировать и поддерживать Dockerfile! Jib выберет базовый образ, соберет проект с помощью Maven/Gradle (это уже сделано нами) и оптимизирует кэш для сборки, используя все локально доступные зависимости и библиотеки JAR;
* Более того, Jib не требует наличия самого Docker! Это может быть удобно в системах непрерывной сборки и интеграции CI/CD и автоматических скриптах, так как установку Docker не всегда удобно делать автоматически;
* Автоматическая оптимизация размера полученного образа. Собственный базовый образ можно указать в конфигурации;
* Скоростная сборка образа — при изменении кода Jib построит новый образ, в котором будет изменена часть кэша (слой layer), отвечающая только за классы Java. Как правило, образ будет готов в течение секунд, а не минут. Это особенно удобно и выгодно при разработке с Kubernetes, если вы часто создаете новые версии своих образов для тестирования.

Java в некотором роде выделяется среди других языков из-за уже готового кэша библиотек JAR и стандартного формата практически всех проектов на основе Maven/Gradle, что и позволяет Jib успешно оптимизировать построение контейнеров.

## Альтернативы Dockerfile. Buildpacks

Другие языки не всегда имеют решения, подобные Jib для Java, но общую идею понять нетрудно — по сути, ваш проект собирается стандартным образом (для Java это Gradle и Maven, для Node.js — инструкции сборки в файле package.json, у Python и Go — свои в основном общепринятые стандарты), ну а дальше собранные артефакты надо перенести в образ (image) контейнера простыми командами Dockerfile и выбрать подходящий базовый образ, как правило, самый популярный для выбранного языка и его версии на Docker Hub.

Возникает вопрос, стоит ли вообще заниматься созданием и поддержкой своих собственных инструкций Dockerfile для сборки образов, если проект, по сути, просто собирается и упаковывается практически стандартным образом? Подобный подход получил название «исходный код в образ» (source/code to image). Он полностью игнорирует низкоуровневый процесс создания образов и рассматривает их как еще один способ упаковки полностью собранного приложения — к примеру, Java-приложение может быть набором файлов .class, архивом JAR, ну или образом image.

Преимущества очевидны: для упаковки приложения в образ нужно, как правило, вызвать простую команду и указать репозиторий, где контейнер будет храниться. Некоторые облачные сервисы (к примеру Heroku) вообще строят образы автоматически при отсылке кода в репозиторий Git. Наличия файла Dockerfile не требуется. Базовый образ, как правило, отобран из соображений безопасности, минимальной функциональности и оптимизирован по размеру.

В качестве начальной точки можно рекомендовать проект Cloud Native Buildpacks — набор общих решений для сборки образов контейнеров без обязательного наличия Dockerfile. У проекта Buildpacks есть свой инструмент командной строки под названием pack, вы можете найти его на сайте buildpacks.io или GitHub. После установки попробуем собрать свой Java-проект, но на этот раз без использования Dockerfile или Jib, как если бы проект вообще не планировалось запускать как контейнер:

# не забудьте восстановить Dockerfile из исходников GitHub  
$ cd dockerfile/java-hello  
$ rm Dockerfile\*  
$ pack build java-hello-pack  
…  
Suggested builders:  
Google..  
Heroku..  
Paketo..

Мы вызвали команду pack build [имя образа], однако остался еще один шаг — выбрать, какой именно build pack будет использован. Cloud Native Buildpacks — это стандарт, описывающий процесс создания образов из исходного кода, и у него есть несколько реализаций — как мы видим, доступны реализации от Google, Heroku или Paketo. Используем сборку Google (флаг --builder):

$ pack build java-hello-pack --builder gcr.io/buildpacks/builder:v1  
v1: Pulling from buildpacks/builder  
…  
===> ANALYZING  
…  
===> DETECTING  
4 of 5 buildpacks participating  
…  
===> BUILDING  
=== Java - Runtime (google.java.runtime@0.9.1) ===  
Using latest Java 11 runtime version.  
…  
Successfully built image java-hello-pack

Как мы видим, сборка работает в несколько этапов, сначала проходит анализ исходного кода и определение возможных способов его сборки — в нашем случае в корне проекта есть файл pom.xml, логика сборщика buildpack найдет его и, конечно же, вызовет сборку с помощью Maven, скорее всего, сборщик поддерживает и дополнения Spring Boot и, скорее всего, вызовет одну из оптимизированных для таких проектов целей сборки Maven.

Надо сказать, что в первый раз сборка образа займет достаточно долгое время — локальные кэши еще не созданы, и будет загружен сам сборщик buildpack (который сам по себе является образом image), все необходимые зависимости Maven, базовые образы и выбранная версия Java. Следующие сборки будут намного быстрее.

Запустить и проверить полученный образ командой docker run будет отличным маленьким упражнением.

Что мы получаем, используя стандартные инструменты сборки образов:

* Мы не потратили ни секунды на создание Dockerfile и обдумывание его деталей;
* Сборщики buildpacks постоянно обновляются, обновляется и собранный ими образ вашего приложения — к примеру, недавно обнаруженная уязвимость в Log4J будет автоматически исправлена в базовом образе и версиях JDK — конечно, свой код вам придется обновлять самостоятельно;
* Вы получаете опыт и экспертизу создателей buildpacks, которые будут постоянно обновлять и оптимизировать создаваемые ими образы;
* Исчезают возможности внести ошибку в собственный Dockerfile, а ошибки в нем бывают критические (уязвимости в базовых образах, оставленные в реальной эксплуатации отладочные инструменты и многое другое).

Впрочем, если посмотреть на инструменты source-to-image на GitHub, станет понятно, что они лишь умеренно популярны (если судить по количеству внешних разработчиков и «звезд») и не стали де-факто стандартом упаковки в образы контейнеров. Это вполне объяснимо — мы увидели в этой главе, что написать Dockerfile не так и сложно, при этом есть полная свобода выбора, можно начать с самого начала (базовый образ scratch), добавить свои собственные мелочи и быть уверенным в том, что же именно содержит полученный образ. Для многих команд и компаний эта уверенность и простота Dockerfile перевешивает еще более простые вызовы инструментов source-to-image.

Сделать выбор между созданием и поддержкой собственных Dockerfile или использованием подхода «исходный код в образы» можете только вы и ваша команда. У каждого подхода есть очевидные преимущества. Можно сказать, что при экспериментах и создании множества маленьких проектов использование buildpacks (или Jib) оправданно — это намного лучше, чем копировать почти идентичные Dockerfile и поддерживать их. Если же у вас уже стабильный проект с минимальным количеством изменений при создании образа, собственный Dockerfile не требует такого внимания и при этом дает полную свободу выбора базового образа и любых дополнений внутри образа вашего приложения или микросервиса.

Сама идея buildpacks пришла из облачного сервиса Heroku, одного из первопроходцев «платформы как сервиса» (platform as a service, PaaS). Вместо низкого уровня услуг (запуск любого кода на виртуальных машинах) платформы, как правило, запускают код приложения и избавляют программистов от низкоуровневых деталей. Это особенно хорошо работает для небольших приложений. Идеи PaaS получили продолжение в Google App Engine, OpenShift, а позже стали еще проще в виде облачных функций (AWS Lambda, Cloud Functions). Но, как мы вскоре увидим, Kubernetes намного мощнее этих платформ и способен предоставить те же самые возможности.

## Резюме

* Контейнеры и Docker хороши для экспериментов, интеграционного тестирования и запуска известных операционных систем и баз данных, но главная их задача — упростить и сделать более гибкой разработку новых приложений и микросервисов. Поместить свое приложение в контейнер позволяет сборка образа image с помощью инструкций Dockerfile.
* Каждый новый образ контейнера строится на основе базового (base) образа. Выбор базового образа влияет на размер собранного образа с приложением и количество доступных сервисов и инструментов.
* Каждый язык программирования предлагает набор стандартных базовых образов для компиляции, сборки и запуска написанных на нем приложений. Использовать в качестве базового образа Linux и заново скачивать и устанавливать все необходимые компиляторы и пакеты, как правило, расточительно и неэффективно.
* Хорошей практикой является компиляция и сборка приложения прямо внутри временного контейнера во время работы команды docker build. Это позволяет получить «чистую» сборку на основе исходного кода, без случайных зависимостей от локальной системы.
* Многоступенчатый (multi-stage) Dockerfile позволяет собрать образ контейнера в несколько ступеней, значительно уменьшив количество ненужных инструментов в окончательном образе с приложением.
* По умолчанию версией в метке (tag) для образа контейнера является latest. Ее легко перезаписать, потеряв старый образ. Оптимальным вариантом является использование семантических версий для каждого, даже самого маленького изменения в функциональности приложения в образе.
* Существуют эффективные альтернативы Dockerfile — к примеру, для приложений Java можно просто использовать уже имеющиеся у программистов знания Maven/Gradle, чтобы собирать эффективные образы с помощью плагина Jib.
* По большому счету, можно собрать своё приложение на любом популярном языке в образ, минуя создание Dockerfile — мы попробовали это с помощью buildpacks.