



Контейнеры и управление ими

План занятия

Определение контейнера	3
Dockerfile.....	4
Builder	5
docker-compose	7
Device-api	8
Запуск контейнеров	9
Livecheck	10
Теория про контейнеры	11
Таймлайн	11
Заключение	13
Тизер	13
Домашнее задание.....	13

Определение контейнера

Когда-то разместить грузы на корабле было сложно, но после изобретения контейнера эта задача упростилась.



Контейнер с приложением упрощает работу с приложениями. Стандарт [Open Container Initiative](#) говорит, что приложение, упакованное в стандартизованный контейнер, можно запустить в любом совместимом окружении, например Unix/Linux/Windows, на любом оборудовании, независимо от содержимого контейнера. Контейнер — единица поставки ПО, он собирается и поставляется в производство, его копии запускаются при масштабировании, а при обнулении версии контейнер заменяется новым.

Компоненты, от которых зависит работа приложения, упакованы в контейнере вместе с приложением, так что устанавливать эти компоненты в окружение не нужно.

Dockerfile

Dockerfile — файл описания процедуры сборки контейнера, оформленный в декларативном стиле, похожий на файл конфигурации, а не на программный код. Подробное описание можно найти в [документации](#), а мы рассмотрим работу файлом Dockerfile на примере. Это поможет на практике отметить важные моменты.

```
Dockerfile x
1  # Builder
2
3  ARG GITHUB_PATH=github.com/ozonmp/act-device-api
4
5  FROM golang:1.16-alpine AS builder
6  RUN apk add --update make curl git protoc protobuf-dev
7  COPY . /home/${GITHUB_PATH}
8  WORKDIR /home/${GITHUB_PATH}
9  RUN make deps-go && make build-go
10
11 # gRPC Server
12
13 FROM alpine:latest as server
14 LABEL org.opencontainers.image.source https://${GITHUB_PATH}
15 RUN apk --no-cache add ca-certificates
16 RUN apk --no-cache add curl
17 WORKDIR /root/
18
19 COPY --from=builder /home/${GITHUB_PATH}/bin/grpc-server .
20 COPY --from=builder /home/${GITHUB_PATH}/config.yml .
21 COPY --from=builder /home/${GITHUB_PATH}/migrations/ ./migrations
22
23 RUN chown root:root grpc-server
24
25 EXPOSE 50051
26 EXPOSE 8080
27 EXPOSE 9100
28
29 CMD ["./grpc-server"]
30
```

- Каждая строка файла — команда, представляющая собой промежуточный слой при сборке. Слои кешируются, поэтому при изменении строки, пересборка по умолчанию начнется с нее.
- В файле две команды FROM, означающие сборку в несколько стадий (сборка multi-stage). Устанавливаем нужные зависимости в контейнер для сборки, собираем файл сервиса и передаем его в следующий контейнер. В производство отправится контейнер меньшего размера и безопаснее — в нем нет инструментов сборки.
- Исходный образ для сборки контейнера скачается из публичного репозитория, но можно настроить и локальное хранилище — registry. При скачивании образа указывайте его архитектуру. По умолчанию скачивается образ для архитектуры локального компьютера (localhost), которого вы скачиваете образ. Обратите на это внимание, если локальный компьютер имеет архитектуру CPU ARM M1 или другую редко встречаемую архитектуру.

- Путь к приложению, которое запустится внутри контейнера при его запуске, задан в параметрах команды CMD файла Dockerfile.
- Приложение в контейнере ожидает соединения на сетевых портах. Чтобы к ним можно было подключиться вне контейнера, следует перечислить порты в параметрах команды EXPOSE.

Builder

Сборщик (builder) — приложение, которое понимает инструкции Dockerfile. Сборщиком может быть `docker`, но в [Ozon](#) мы используем `kaniko`. Этот сборщик написан на языке [Golang](#).

Важно, что образ собирается без `docker`, потому что среда сборки тоже расположена в контейнере. Собирать контейнер в контейнере при помощи `docker` небезопасно, так как из-за его архитектуры понадобится доступ к хостовой системе, изоляция контейнера будет нарушена. `Kaniko` лишен этого недостатка.

Запускаем сборку командой `docker build .`. Обратите внимание на точку в конце команды: она означает, что сборщик должен искать файл с именем Dockerfile в текущей директории, из которой выполняется команда.

Registry

Собранный контейнер останется в директории, из которой выполнялась сборка. Чтобы сделать его доступным для публикации, нужно положить контейнер в хранилище собранных образов — Registry. Самое большое из известных хранилищ — `docker hub`, которое хорошо масштабируется и распространяется под лицензией Apache. Развернуть его в сети компании — хорошая практика.

Dive

В заключение стоит упомянуть про утилиту [Dive](#), полезную для отладки. После установки ей нужно передать имя образа контейнера — и в псевдографическом интерфейсе можно увидеть команды из Dockerfile, изменения на файловой системе после каждой команды.

Наш финальный образ получился компактным и простым в анализе.

Layers

Cmp	Size	Command
	5.6 MB	FROM 81520538fd3d02c
	482 kB	apk --no-cache add ca-certificates
	24 MB	#(nop) COPY file:b81c33543813882fbaaff7e6d30
	775 B	#(nop) COPY file:66c11996ef56e611a97e1e7a655
	4.1 kB	#(nop) COPY dir:978bd9ba2b0f67387451c5c617d6
	0 B	chown root:root grpc-server

Layer Details

Tags: (unavailable)

Id:

f25e17d0261dd95e7257e90e1ec140fa5e59a624b39e78089d8297a0ea7803ce

Digest:

sha256:147aa9713796571f0487475f9e907127287c9e6d8131a2ba96a75dc667c9a4b

Command:

chown root:root grpc-server

Image Details

Total Image size: 30 MB

Potential wasted space: 480 kB

Image efficiency score: 99 %

Current Layer Contents

bin

dev

etc

home

lib

media

mnt

opt

proc

root

run

sbin

srv

sys

tmp

usr

var

^C Quit | Tab Switch view | ^F Filter | ^L Show layer changes | ^A Show aggregated changes |

docker-compose

После успешной сборки контейнера поговорим про его запуск.

В этих источниках описаны свойства, которыми должен обладать запуск контейнера:

- [The Twelve-Factor App](#). Этот сайт — манифест, который описывает опыт разработчиков известной SaaS платформы Heroku. В разделе про процессы подчеркивается, что процесс должен быть без состояния (stateless process), тем более не должен разделять его с другими процессами

В документации [Docker](#) следствие этой мысли применяется к контейнеру. Хороший контейнер должен быть эфемерным, готовым к запуску и остановке в любой момент. И не просто остановке, а к замене на любую другую версию.

Кроме запущенного процесса, у нас есть данные. Где же их хранить? В БД или другом хранилище.

Получается, что контейнеров должно быть больше одного, только контейнера с тестовым сервисом недостаточно. Возникает понятие оркестрации — порядок запуска, инициализация контейнера, сетевая связность. В тестовом проекте мы будем использовать docker-compose так же, как мы используем его для локальной разработки в Ozon.

Формат описания запуска для docker-compose описан на странице <https://docs.docker.com/compose/compose-file/compose-file-v3/>.

В тестовом репозитории уже есть файл docker-compose.yaml, так что мы готовы к запуску, только сначала давайте разберемся подробнее, что происходит внутри.

PostgreSQL

```
postgres:
  container_name: postgres
  platform: linux/x86_64
  image: postgres:13
  healthcheck:
    test: [ "CMD", "pg_isready", "-q", "-d", "postgres", "-U", "postgres" ]
    timeout: 45s
    interval: 10s
    retries: 10
  restart: always
  ports:
    - 5432:5432
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: password
    POSTGRES_DB: act_device_api
    APP_DB_USER: docker
    APP_DB_PASS: docker
  networks:
    - omprnw
  volumes:
    - ./scripts/init-database.sh:/docker-entrypoint-initdb.d/init-database.sh
```

В нашем тестовом проекте мы будем запускать базу данных PostgreSQL (БД). Чтобы не включать в наш образ эту БД, мы используем готовый образ, который поддерживается на Docker Hub: `image: postgres:13`. Тут, как всегда, работает умолчание — если указано только имя образа, он качается с Docker Hub.

База запускается не мгновенно, для проверки ее готовности к работе используем `healthcheck`. Команда `restart: always` обеспечивает повтор проверки: если БД станет недоступна на некоторое время, контейнер с ней перезапустится.

База данных ничего не знает про базу именно нашего приложения. Для создания базы и подключения к ней мы используем скрипт `init-database.sh`, который помещаем внутрь контейнера. Используем команду `volume`, чтобы папка или файл хостовой системы были доступны приложению внутри контейнера.

В блоке `ports` укажем порт 5432, чтобы сделать его доступным снаружи контейнера. Это может понадобиться для отладки приложения в контейнере и поможет подключиться к БД из IDE или запустить собранный тестовый сервис без контейнера.

Device-api

Тут тоже много достойного нашего внимания.

В описании тестового сервиса объявлено, что перед его запуском нужно дождаться результата запуска PostgreSQL: `depends_on`.

У обоих контейнеров общая сеть в команде `networks`, это значит, что наше приложение сможет подключиться к БД. Docker поддерживает DNS, в котором запущенный контейнер определяется

внутри сети по имени, указанном в команде name, поэтому в строке подключения мы используем просто postgres. Как упоминалось, сервис можно запускать и снаружи контейнера. Адреса подключения к БД будут отличаться, поэтому для запуска внутри контейнера мы используем другой файл конфигурации — файл container-config.yml.

```
act-device-api:
  container_name: act-device-api
  build:
    context: .
    dockerfile: Dockerfile
  restart: unless-stopped
  networks:
    - ompnw
  depends_on:
    - postgres
  links:
    - postgres
  ports:
    - 8083:8080 # REST
    - 8082:8082 # gRPC
    - 9100:9100 # Metrics prometheus
    - 8000:8000 # Status prometheus
    - 40000:40000 # Debug port
  healthcheck:
    test: [ 'CMD', 'curl', '-v', 'http://localhost:8000/live' ]
    interval: 10s
    timeout: 10s
    retries: 2
    start_period: 20s
  volumes:
    - ./migrations:/root/migrations
    - ./container-config.yml:/root/config.yml
```

При запуске сервис проверяет, что база данных создана и к ней можно подключиться, и создает в базе необходимые для своей работы таблицы и данные. Поэтому мы папку с миграциями мы делаем доступной внутри контейнера. О миграции мы поговорим на лекции про SQL.

Запуск контейнеров

Все готово к запуску.

```
> docker-compose up -d
> docker-compose ps
```

Ключ `-d` означает, что контейнеры не будут выводить свои `stdout` в консоль, и в ней можно будет выполнить вторую команду. В `ps` мы увидим и результаты старта контейнеров, и `healthcheck`.

Если что-то пошло не так, попробуйте остановить контейнеры и запустить с выводом журналов в консоль (без ключа `-d`) или прочитать журналы запущенного контейнера (ключ `-f`) команды `logs`.

В реальности на вашем ноутбуке может быть запущено несколько проектов разных сервисов. И всем им будет нужна, к примеру, `postgres`. Чтобы избежать влияние сервисов друг на друга, при запуске контейнеров нужно использовать ключ **`-p <project name>`**. Запущенные контейнеры получают уникальное имя с именем проекта в нем и запустится несколько инстансов `postgres`. Этот же ключ нужно будет использовать в остальных командах (`logs`, `ps` и других).

Livecheck

Как мы видели на примере БД или даже нашего сервиса, запуск контейнера неатомарен, многое может пойти не так. Поэтому мы используем для сервисов `livecheck`-пробу. Сервис обязан отдавать HTTP-код 200 при обращении по адресу `:8000/live` только тогда, когда он действительно готов обрабатывать запросы. Для системы оркестрации это служит сигналом, что на этот сервис может идти трафик, а если проверка не проходит, сервис перезапускается, или выполняется другое действие, чтобы сделать сервис доступным.

Теория про контейнеры

Давайте задумаемся, только что, используя всего два файла конфигурации, мы сделали работу DBA, разработчика и администратора: запустили БД, собрали и запустили сервис, сделали так, что они могут работать друг с другом, потратили на это совсем немного времени.

Эта легкость возникает благодаря стандартизации. Сейчас ей занимается проект [Open Container Initiative](#). Появилась она благодаря усилиям Google, Red Hat и Docker, а в основу легли спецификации Docker.

Спецификация делится на две части: [OCI Runtime Specification](#) (где запускать образ) и [OCI Image Specification](#) (как его собирать).

Со сборкой образа мы уже немного знакомы, стоит сказать пару слов про Runtime - среду выполнения контейнера.

В случае Docker контейнер это изолированный процесс (или несколько процессов). Docker нужно ядро Linux, потому что оно поддерживает нужные механизмы контейнеризации namespaces, cgroups, Kernel capabilities и некоторые другие. Некоторые механизмы были разработаны и без участия Docker-сообщества, но все вместе помогли ему воплотить удачную контейнеризацию.

На MacOS и Windows нет ядра Linux, поэтому Docker запускает виртуальную машину с ним, вот, почему Mac так греется с запущенными контейнерами.

Таймлайн

Ниже представлен таймлайн технологий, так или иначе участвующих в контейнеризации Docker или представляющих альтернативные технологии. Он важен для понимания предпосылок современной контейнеризации.

1999, FreeBSD jail Each jail is a virtual environment running on the host machine with its own files, processes, user and superuser accounts.

1999, with the 2.2 Linux kernel release: capabilities <https://linux.die.net/man/7/capabilities>.

2002, mount namespace.

2005, OpenVZ VE (virtual environments), VPS - виртуализация, только дистрибутивы Linux, модифицированное ядро.

2006, IPC namespace, UTS namespace.

2007, KVM (Kernel-based Virtual Machine).

2008, LXC.

2008, PID namespace.

2008, cgroups (v1).

2009, Network namespace.

2010, The initial RFC patchset of OverlayFS was submitted by Miklos Szeredi in 2010.

2013, User namespace.

2013, Docker first release.

2014, k8s first release.

2015, OCI specifications.

2015, LXC deprecated.

2016, Cgroup namespace.

Заключение

На занятии мы разобрались с конфигурацией и запустили наш тестовый сервис и его базу данных. Это не упрощение, а принятая практика разработки в Ozon, разрабатываемый сервис должен собираться и запускаться на машине разработчика.

Конечно, дальше он будет запускаться и в среде k8s, но стандартизация позволяет собирать и запускать контейнеры, не задумываясь о том, где они дальше будут работать.

Контейнеризация дает возможность собирать и тестировать только часть приложения, это экономит ресурсы, добавлять, менять и убирать сервисы «на лету», и, в конечном итоге бизнесу не важно, делает это Docker, k8s или любой другой оркестратор.

Тизер

На следующем занятии мы попробуем тестовый сервис в деле и рассмотрим на его примере теорию протоколов HTTP, REST и gRPC.

Домашнее задание

Вам нужно будет запустить тестовый сервис не в контейнере, а снаружи него. Часто это нужно для отладки, запуска тестов, и это хорошая возможность разобраться с файлом конфигурации нашего приложения и с docker-compose.