**Предисловие**

это полный конспект «Руководства» docker compose в котором описаны некоторые его подкоманды; структура compose.yaml, некоторые его элементы верхнего уровня и атрибуты:

<https://docs.docker.com/compose/>

«Справочник» по compose здесь конспектирован частично с самым необходимым, так-как в документации информация описана кратко.

Подкоманды docker compose:

[https://docs.docker.com/reference/cli/docker/compose/#subcommands](https://docs.docker.com/reference/cli/docker/compose/" \l "subcommands)

Элементы верхнего уровня и атрибуты:

<https://docs.docker.com/reference/compose-file/>

**docker compose**

https://docs.docker.com/compose/

https://docs.docker.com/reference/compose-file/

аркестрация мульти-контейнерных приложений

**compose.yaml**

файл конфигурации, должен находится в рабочей директории; так-же может иметь вид compose.yml, если присутствуют оба - будет выбран .yaml; для обратной совместимости с более ранними версиями так-же поддерживаются файлы docker-compose.yaml и docker-compose.yml

Несколько compose.yaml могут быть merged в один, fragments и extensions могут быть включены в главный файл.

Небольшая тонкость: очередность строк кода или блоков не имеет значения в отличае от dockerfile, НО есть исключения: при слиянии файлов, использовании якорей и алиасов, и др., подробнее далее.

**Subcommands of docker compose:**

[https://docs.docker.com/reference/cli/docker/compose/#subcommands](https://docs.docker.com/reference/cli/docker/compose/" \l "subcommands)

cli подкоманды docker compose, описанны далее и по ссылке в документации.

Примечание: выполняются в папке с файлом compose.yaml

**docker compose up**

запустить все сервисы, определенные в файле compose.yaml (будут созданы образы и запущены контейнеры) и выводить логи контейнеров в реальном времени

Примечание: при уже поднятом приложении повторное выполнение команды перезапустит приложение с обновлением конфигурации приложения описанным в compose.yaml и сборкой новых образов, НО НЕ БУДЕТ пересобирать уже существующие образы приложения

**docker compose up -d**

без логов, делает то-же что команда выше

**docker compose logs**

посмотреть логи, но не в реальном времени

**docker compose -f some/path/anotherFileName.yaml**

передать путь к файлу и/или нестандартное имя файла

**docker compose down**

остановить и удалить работающие службы (контейнеры, сети, но не тома и образы)

**docker compose ls**

список запущенных всех compose проектов

**docker compose ps**

список запущенных контейнеров из compose.yaml

**docker compose up -d --build**

перзапустить/запустить все сервисы и пересобрать все кастомные образы

**docker compose build serviceName**

пересобрать конкретный образ измененный в compose.yaml сервисы, например web

**docker compose up --no-deps -d serviceName**

перезапустить конкретный контейнер из измененного образа без зависимых контейнеров.

Метод описан в документации, но даже там рекомендуется полностью перезапускать все многоконтейнерное приложение после изменения образа во избежание ошибок в работе.

**docker create secret secretName fileWithPasswords.txt**

https://docs.docker.com/compose/how-tos/use-secrets/

https://docs.docker.com/engine/swarm/secrets/

В docker хорошей практикой является не хранить пароли в образах во избежания их утечки при коммитах, а использовать secret. Смотри документацию по ссылкам и в шпаргалке secret.docx. Пока что погружатся в эту тему не буду.

**top-level elements, attributes, and their values of compose.yaml:**

**# Comments**

коментировать можно только построчно

**top-level: value**

**top-level:**

**attribute: value**

**attribute:**

**attribute: value**

**attribute: value**

структура данных в YAML файле должна быть в виде "ключ: значение" как в ассоциативном массиве. Top-level элементы могут иметь как значения так и атрибуты. Attribute тоже могут иметь как значения так и атрибуты со своими значениями.

Уровень вложенности определяется отступами. Отступы должны выполнятся пробелами: количество пробелов на уровень не нормируется, но должно быть одинаковым в приделах одного документа, в примерах документации используется два пробела.

**name: some**

определяет имя для элемента, может использоватся как top-level и как attribute.

верхний уровень и аналоги для командной строки:

name: myproject

docker compose -p myproject up

docker compose --project-name myproject up

атрибуты:

networks:

frontend:

name: custom\_frontend

volumes:

db-data:

name: "my-app-data"

services:

ubuntu:

container\_name: ubuntu-container-app

примечание: по умолчанию имя проекта будет совпадать с именем родительской директории.

**services:**

top-level элемент содержащий ассоциативный массив, keys которого являются имена контейнеров, а values их настройки

services:

web-server:

image: nginx:latest

...

db:

build: .

...

backend:

build:

context: ./dir

dockerfile: Dockerfile-alternate

...

каждый service должен включать раздел image и/или build с инструкцией сборки образа:

<https://docs.docker.com/reference/compose-file/build/>

**post\_start:**

**pre\_stop:**

[https://docs.docker.com/compose/how-tos/lifecycle/#services-lifecycle-hooks](https://docs.docker.com/compose/how-tos/lifecycle/" \l "services-lifecycle-hooks)

аналог ENTRYPOINT,но выполняется дополнительно к Dockerfile командам при старте и перед остановкой контейнера, пример смотри в документации по ссылке.

**entrypoint и command**

[https://docs.docker.com/reference/compose-file/services/#entrypoint](https://docs.docker.com/reference/compose-file/services/" \l "entrypoint)

https://docs.docker.com/reference/compose-file/services/#command

в compose.yaml можно переопределить указанные в образе или Dockerfile ENTRYPOINT и CMD

Примечание: хорошей практикой является использовать эти команды в Dockerfile, т.е. образ не должен изменятся в compose.yaml, а быть полностью готовым для использования.

**profiles: [name]**

https://docs.docker.com/compose/how-tos/profiles/

сервис с таким атрибутом будет запущен только при указании его имени в опции --profile или установленной переменной среды:

services:

frontend:

image: frontend

profiles: [frontend]

phpmyadmin:

image: phpmyadmin

depends\_on: [db]

profiles: [debug]

backend:

image: backend

db:

image: mysql

docker compose --profile frontend --profile debug up

COMPOSE\_PROFILES=frontend,debug docker compose up

**depends\_on:**

https://docs.docker.com/compose/how-tos/startup-order/

https://docs.docker.com/reference/compose-file/services/#depends\_on

порядок запуска контейнеров; точнее этот атрибут указывает какие контейнеры должны быть запущены до запуска контейнера в котором используется этот атрибут.

Примечание: здесь пример короткого синтаксиса с минимальным набором функций, в документации по второй ссылке есть длинный синтаксис с дополнительными функциями.

services:

web:

build: .

depends\_on:

- db

- redis

redis:

image: redis

db:

image: postgres

**healthcheck:**

[https://docs.docker.com/reference/compose-file/services/#healthcheck](https://docs.docker.com/reference/compose-file/services/" \l "healthcheck)

определяет критерии "здоровья" контейнера (starting, healthy, unhealthy); другие контейнеры при помощи, например, "depends\_on" могут проверять состояние этого контейнера и реагировать соответствующе. Без healthcheck Docker знает только базовое состояние контейнера: запущен ли он или остановлен.

Например, можно поднимать web приложение если DB не только стартовала, но и "здорова":

services:

web:

build: .

depends\_on:

db:

condition: service\_healthy

db:

image: postgres

healthcheck:

test: ["CMD", "pg\_isready", "-U", "postgres"]

interval: 10s

timeout: 5s

retries: 5

**environment:**

https://docs.docker.com/compose/how-tos/environment-variables/set-environment-variables/

переменные среды для контейнера. Есть много фич, например наследование переменных системы или загрузка из файла .env, смотри документацию по ссылке.

Пример простых переменных в разных формах записи:

services:

webapp:

environment:

DEBUG: "true"

или

- DEBUG=true

https://docs.docker.com/compose/how-tos/environment-variables/envvars-precedence/

приоритет переменных при конфликтах, например переменная есть в .env, атребуте и образе.

https://docs.docker.com/compose/how-tos/environment-variables/envvars/

установка или изменение предопределенных переменных среды в Docker Compose

https://docs.docker.com/compose/how-tos/environment-variables/variable-interpolation/

интерполяция переменных, что бы это не значило)

**secrets:**

https://docs.docker.com/compose/how-tos/use-secrets/

Для избежания утечки паролей при коммитах, хорошей практикой в docker является хранение паролей в secrets.

На момент написания этой шпаргалки seccrets не рассмотрены, но docker AI дает хорошую инструкцию. Secrets требует активированного Docker Swarm mode, но можно использовать текстовые файлы с localhost напрямую. Смотри документацию по ссылке и конспект по secrets.

**Configs top-level elements**

https://docs.docker.com/reference/compose-file/configs/

таким-же образом как secrets можно монтировать файлы с конфигурационными параметрами для использования их в контейнере, например для development и production сред, и настраивать разное поведение контейнера в зависимости от среды без необходимости его пересборки. Есть разные способы монтирования. Пока не погружался в тему.

**networks:**

[https://docs.docker.com/compose/how-tos/networking/#multi-host-networking](https://docs.docker.com/compose/how-tos/networking/" \l "multi-host-networking)

https://docs.docker.com/reference/compose-file/networks/

настройки сети для многоконтейнерного приложения.

По умолчанию, то есть если настройка отсутствует в compose.yaml, создается сеть с именем app-name\_default, драйвером bridge и NAT сервиром, поэтому контейнеры могут общатся друг с другом по имени.

При необходимости задать свои настройки сети необходимо в файле compose.yaml создать top-level элемент с настройками сети и в service-level атрибуте контейнера сослатся на top-level элемент:

services:

apache:

image: httpd:latest

ports:

- "80:80"

networks:

- app\_network

laravel:

image: your-laravel-image:tag

networks:

- app\_network

mysql:

image: mysql:latest

networks:

- app\_network

networks:

app\_network:

driver: bridge

Так-же у многоконтейнерного приложения может быть несколько сетей, например, если контейнерам Laravel и mysql необходимо полностью запретить доступ в интернет:

services:

apache:

image: httpd:latest

ports:

- "80:80"

networks:

- frontend

- backend

laravel:

image: your-laravel-image:tag

networks:

- backend

mysql:

image: mysql:latest

networks:

- backend

networks:

frontend:

driver: bridge

backend:

driver: bridge

internal: true

В этом примере сеть backend имеет параметр internal: true - запрет доступа в интернет

**expose:**

[https://docs.docker.com/reference/compose-file/services/#expose](https://docs.docker.com/reference/compose-file/services/" \l "expose)

публикация порта сервиса (контейнерва) в многоконтейнерном приложении; порт становится доступным только другим контейнерам данного многоконтейнерного приложения, но НЕ доступен с localhost.

Примечание: хотя в Dockerfile инструкция EXPOSE служить только для декларации, фактически не раскрывает порт в docker engine, но при использовании Dockerfile с инструкцией EXPOSE в compose.yaml этот порт будет раскрыт другим контейнерам данного многоконтейнерного приложения даже без использования атрибута expose в compose.yaml

services:

web:

image: nginx

expose:

- "80"

- "8000"

- "8080-8085/tcp"

**ports:**

[https://docs.docker.com/reference/compose-file/services/#ports](https://docs.docker.com/reference/compose-file/services/" \l "ports)

связь порта localhost и порта контейнера многоконтейнерного приложения

services:

wordpress:

image: wordpress

ports:

- "8080:80"

- "9090-9091:8080-8081"

- "127.0.0.1:8001:8001"

- "6060:6060/udp"

Примечание: есть «длинный синтаксес» более человекочитаемый, смотри документацию

**volumes:**

настройки хранилищ для многоконтейнерного приложения.

Примечание: хорошей практикой является создавать все хранилища в top-level элементе volumes и ссылатся на них в сервисах для лудшей структурированности файла, но есть возможность настраивать шранилище в сервисе.

services:

backend:

image: example/database

volumes:

- db-data:/var/lib/mysql

- config:/etc/mysql/conf.d:ro

- backend-tmp:/tmp

frontend:

image: example/webapp

volumes:

- db-data-simple:/app/data-simple

- db-data:/app/data

- logs:/app/logs

volumes:

db-data-simple:

db-data:

driver: local

driver\_opts:

type: nfs

o: addr=10.40.0.199,nolock,soft,rw

device: ":/docker/example"

labels:

com.example.description: "Database volume"

com.example.department: "IT/Ops"

config:

driver: local

driver\_opts:

type: none

o: bind

device: ./config

logs:

driver: local

driver\_opts:

type: none

o: bind

device: ./logs

backend-tmp:

driver: tmpfs

driver\_opts:

size: 100M

Примечение: пример сгенерирован Docker LLM и носит ознакомительный характер.

db-data-simple - хранилище volume типа с настройками по умолчанию;

db-data - хранилище volume типа с явно указанными настройками:

driver: local - локальный volume;

driver\_opts - определяет специфические опции для драйвера:

type: nfs - NFS (Network File System) volume;

o: addr=10.40.0.199,nolock,soft,rw - адрес сервера и параметры монтирования;

device: ":/docker/example" - путь на NFS сервере;

labels - метаданные к volume:

com.example.description: "Database volume" - описание volume.

com.example.department: "IT/Ops" - отдел, ответственный за volume.

config и logs - хранилища bind mounts типа , определенные с использованием driver\_opts;

backend-tmp - хранилище tmpfs типа.

**docker compose -f compose.yaml -f compose.prod.yaml up -d**

https://docs.docker.com/compose/how-tos/multiple-compose-files/merge/

объединить **merge** способом два .yaml файла и создать из объединенного файла многоконтейнерное приложение.

Cценарий использования: есть compose.yaml для production, но для разворочивания приложения на debug сервере необходимы другие настройки, чтобы постоянно не менять настройки создается дополнительный common.yaml файл с настройками которые при merge будут заменять и дополнять настройки основного compose.yaml файла.

Тема большая, включает конфликты слияния, но сейчас подробно рассмотренна не будет, смотри документацию.

**extends:**

импорт данных из другого .yaml файла с возможностью неоднократно использовать в разных местах compose.yaml:

common-services.yaml файл

services:

webapp:

build: .

ports:

- "8000:8000"

volumes:

- "/data"

compose.yaml

services:

web:

build: alpine

command: echo

extends:

file: common-services.yaml

service: webapp

webapp:

extends:

file: common-services.yaml

service: webapp

пример из документации, imho не совсем корректный, не понятно как два контейнера будут использовать один порт.

примечание: extends не может ссылатся на свой файл, во избежании бесконечных циклов.

**include:**

Используется для больших приложений, где есть необходимость разделить конфигурацию приложения на более мелкие част для удобства чтения, так-же удобно для нескольких команд работающих над одним приложением (frontend, backend).

Примечание: при конфликте блоков кода падает в ошибку - не работает как merge.

frontend.yaml:

services:

web:

build: .

ports:

- "3000:3000"

backend.yaml:

services:

api:

build: .

ports:

- "5000:5000"

compose.yaml:

services:

# Здесь будут объявлены общие сервисы

include:

- file: frontend.yaml

- file: backend.yaml