

7/11

Docker compose 복습

vmware → container (docker) → docker compose → kubernetes

docker compose는 여러 서비스 (컨테이너)의 합

따라서 컨테이너는 단일 프로세스로 만들고 컨테이너를 여러 개를 만든다

이렇게 만든 프로젝트의 서비스에 붙는 이름은 project명 - 서비스명 - 1 여러 개 만들수록 이 숫자는 늘어난다.

내부 컨테이너들은 네트워크로 연결 될 수도 있고 파일만 공유가 필요할 경우 볼륨을 마운트해서 사용한다.

yaml 파일을 사용해서 이러한 정보를 프로젝트를 만들 때 명시가 가능하다.

docker compose는 production 환경에서 사용되는 배포 방식

docker compose 자체에 자체 판별하여 restart기능이 있다.

쿠버네티스의 선택은 자유이나

도커 compose는 반드시 사용해야한다.

docker compose → ec2

or ecr, fargate 매니지드 서비스

or eks 매니지드 서비스

or kbs

컨테이너에서 가장 중요한건 Docker file

명령어를 외우지말고 명령어로 뭘 할 수 있었지를 배우는 것이 좋다.

명령어는 다 구글링 가능

- volume

- network

Compose file 기본



매번 버전확인하고 공식문서를 참조해라

2.2. Compose file - Basic

- docker-compose를 위한 configuration 파일은 YAML을 이용하여 작성합니다.
- python과 같이 들여쓰기를 기반으로 구조를 결정합니다.

docker-compose.yaml 파일 예시

```
version: "3.8"
name: "cloud_wave"

services:
```

```
frontend:
  image: nginx:latest
  ports:
    - "80:80"
  networks:
    - private
  configs:
    - server-config
  restart: always

server:
  image: ubuntu:22.04
  entrypoint: /bin/bash
  command:
    - -c
    - "sleep 3600"
  networks:
    - private
    - db
  restart: no

postgres:
  container_name: db_postgres
  image: postgres:16.1-bullseye
  networks:
    - db
  expose:
    - 5432
  volumes:
    - db-data:/var/lib/postgresql/data:rw
  secrets:
    - db_password
  env_file:
    - .env
  environment:
    DB_PASSWORD_PATH: /run/secrets/db_password
  restart: always

volumes:
  db-data:
    name: "db-data"
    labels:
      description: "PostgreSQL 16.1 volume"

configs:
  server-config:
    file: "server.conf"

# Secret not encrypted when swarm is off
secrets:
  db_password:
    file: "secret.txt"

networks:
  private:
    name: "private"
```

```
db:
  name: "db"
```

들어쓰기가 중요

2.2.1. Version

- `version` 은 해당 `compose` 파일에서 사용한 버전을 의미합니다.
- `version` 을 선언하여도 `docker compose` 는 해당 파일을 실행할 수 있는 가장 최신 버전을 사용합니다.

Version이 2인지 3인지 그건 중요하기에 명시를 해줘야 한다.

2.2.2. Name

- 프로젝트의 이름을 의미합니다.
- 프로젝트의 이름은 다음과 같은 순서로 결정됩니다.
 1. `Compose CLI` 의 `-p / --project-name` 가 설정된 경우
 2. `configuration file(docker-compose.yaml)` 에 `name` 이 설정된 경우
 3. `directory` 이름

디렉토리의 이름으로까지 가게되면 중복되는 경우가 많기에 웬만하면 이름 명시해라

2.2.3. Service

<https://docs.docker.com/compose/compose-file/05-services/>

```
# Example
services:
  frontend:
    image: nginx:latest
    ports:
      - "80:80"
    configs:
      - server-config
    restart: always
```

frontend라 이름 짓는다. 이름을 잘 지어야 한다 가독성을 위해

frontend라는 컨테이너에서 nginx:latest 이미지를 사용할 것이며 port를 host 80 컨테이너 80으로 열겠다. 만약 이미지가 로컬에 없다면 레지스트리로 넘어가서 이미지를 다운받는다. 이미지는 layer 단위로만 움직인다. 그래서 이미 있는

layer를 다운받진 않음

- image

image

```
image: string
```

- 컨테이너에서 사용할 이미지입니다.
- 이미지를 `build` 하는 경우 생성된 이미지의 이름으로 사용됩니다.

- container_name

container_name

```
container_name: string
```

- 값을 명시하지 않은 경우, 프로젝트, 서비스, 컨테이너 번호를 기반으로 생성됩니다.
- 알파벳 대소문자, 숫자 및 일부 특수기호(`.`, `_`, `-`)만 사용할 수 있습니다.
- `container_name` 이 설정되어 있는 경우, 해당 서비스를 위한 컨테이너를 1개만 생성할 수 있습니다.

따라서 컨테이너 이름을 안적는 경우도 많다 하나밖에 못 만들기 때문에

- expose

expose

```
expose:  
  - 80 # INT  
  - "8080" # STRING  
  - "1000-1010" # RANGE
```

- 같은 네트워크를 사용하는 서비스에서만 접근 가능하며, `host` 로 `publish` 되지 않습니다.
- `-` 를 이용하여 노출할 포트 범위를 지정할 수 있습니다.

같은 서브넷 내에서만 접근이 가능

- ports

ports

```
ports:
  # Short Syntax
  - [HOST:]CONTAINER[/PROTOCOL]
  # Long Syntax
  - target: 80
    host_ip: HOST
    published: "8080"
    protocol: tcp
```

- 외부에서 접근할 수 있도록 `host` 로 `publish` 할 포트를 정의합니다.
- `-` 를 이용하여 노출할 포트 범위를 지정할 수 있습니다.

외부에서도 접근이 가능하다.

왼쪽 host : 오른쪽 컨테이너

- `entrypoint`

entrypoint

```
# String
entrypoint: STRING
# List
entrypoint:
  - STRING
  - STRING
```

- 컨테이너의 `Dockerfile` 에서 정의된 `ENTRYPOINT` 를 `override` 합니다.
- `ENTRYPOINT` 가 선언된 경우, `Dockerfile` 에서 정의된 `CMD` 는 무시됩니다.

`override`를 할 때에는 어떤 걸 하는지 명확하게 알아야..

`cmd`가 무시된다는 것을주의

- `command`

command

```
# String
command: bundle exec thin -p 3000
# List
command: [ "bundle", "exec", "thin", "-p", "3000" ]
# List - yaml
command:
  - STRING
  - STRING
```

- 컨테이너의 `Dockerfile`에서 정의된 `CMD`를 `override` 합니다.
- 값이 `null` 인 경우 `Image`의 `command`가 사용되지만, `[]` 또는 `''`로 설정된 경우 `Image`의 `command`는 무시됩니다.

null값과 blank값을 구분해라 blank는 초기화지만 null은 그냥 없는 것

- environment

environment

```
# Map Syntax
environment:
  KEY1: value
  KEY2: "value"
  KEY3:

# Array Syntax
environment:
  - KEY1=value
  - KEY2="value"
  - KEY3
```

- `Map` 또는 `Array` 형식으로 작성할 수 있습니다.
- 값이 정의되지 않은 경우, `env_file` 등으로 제공되지 않으면 `unset` 됩니다.
- 동일한 변수가 `env_file`에도 선언되어 있는 경우, `environment`의 값이 설정됩니다.

map 혹은 array로 하는 것은 취향차이이나 팀에서 하라는 대로 해라..

- env_file

env_file

```
# Single
env_file: .env # string

# Multiple
env_file:
  - ./a.env
  - ./b.env
```

- 기본적으로 `.env` 파일이 사용됩니다.
- 여러개의 파일에 동일한 변수가 선언된 경우, 가장 마지막 파일에 있는 값으로 설정됩니다.
- `env_file` 은 다음과 같은 양식으로 작성합니다.
 - `VAR=[VAL]`

password 같은 경우는 .env에 넣고 gitignore 처리 한다.

상대경로면 현재 어느 위치인지 중요

우선순위는 맨 마지막 것이다. 덮어쓰기처럼

env파일은 위에 array_syntax로 작성

- build

build

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

```
services:
  # Short
  frontend:
    image: example/webapp
    build: ./webapp

  # Long - dockerfile
  backend:
    image: example/database
    build:
      context: backend
      dockerfile: ../backend.Dockerfile
      args:
        GIT_COMMIT: cdc3b19
```

- 서비스의 컨테이너를 build 할 때 사용하며, 각 서비스별로 context 경로를 설정할 수 있습니다.
- dockerfile 대신 dockerfile_inline 을 이용하여 사용할 수도 있습니다.

```
build:
  context: .
  dockerfile_inline: |
    FROM baseimage
    RUN some command
```

- pull_policy

pull_policy

```
pull_policy: string
```

- `Compose` 가 이미지를 어떻게 가져올 것인지를 설정합니다.
- 사용할 수 있는 값은 다음과 같습니다.

Value	Description
<code>always</code>	매번 <code>registry</code> 에서 이미지를 다운로드 합니다.
<code>never</code>	저장된 이미지만을 사용합니다. 이미지가 존재하지 않는 경우 실행되지 않습니다.
<code>missing</code>	저장된 이미지가 없는 경우에만 <code>registry</code> 에서 다운로드 합니다.
<code>build</code>	매번 이미지를 <code>build</code> 합니다.

`always`는 배포 환경에서는 사용 x 계속 컨테이너가 업데이트되기에 오류 상황 거의 나온다.

`never`는 `always`의 반대

`missing` 일반적인 패턴

`build` 항상 새로 build해라 - - `no -cache` 같은 느낌

- restart

restart

```
restart: string
```

- 컨테이너가 종료된 경우 어떻게 처리할 것인지를 정의합니다.
- 사용할 수 있는 값은 다음과 같습니다.

Value	Description
<code>no</code>	어떠한 경우에도 컨테이너를 재실행하지 않습니다.
<code>always</code>	제거되지 않는 한, 컨테이너를 항상 재실행합니다.
<code>on-failure</code>	<code>exit code</code> 가 <code>error</code> 인 경우에만 재실행합니다.
<code>unless-stopped</code>	종료 또는 삭제 하는 경우를 제외하고 항상 재실행합니다.

- platform

platform

```
platform: string

# Example
platform: windows/amd64
platform: linux/arm64/v8
```

- 컨테이너의 platform을 명시합니다.

테스트 용도로만 사용하고 platform을 명시하는 경우는 없고 애초에 사용자 architecture에 맞는 것만 사용된다.

- options

CLI option <-> Service elements

Docker CLI Options	Service Elements
-i	stdin_open
-t	tty
-p	ports
-e	environment
--env-files	env_file
--name	container_name

연습1

연습 문제

[연습] ubuntu 서버 실행하기

아래에 있는 `docker-compose.yml` 파일을 이용하여 프로젝트를 실행할 경우, `ubuntu` 서비스가 생성 후 바로 종료되는 것을 확인할 수 있습니다.

```
# docker-compose.yml
version: '3.8'

services:
  ubuntu:
    image: ubuntu:22.04
    restart: no
```

```
$ docker compose up
$ docker compose ps -a
NAME                IMAGE              COMMAND             SERVICE   CREATED
STATUS              PORTS
example2-ubuntu-1   ubuntu:22.04       "/bin/bash"         ubuntu    15 seconds ago
Exited (0) 4 seconds ago
```

`docker run` 때와 마찬가지로 두가지 방식을 이용하여 유지시킬 수 있습니다.

1. `stdin_open`과 `tty`를 이용합니다.
 - `docker run`에서의 `-it` 옵션과 동일합니다.

```
version: '3.8'

services:
  ubuntu:
    image: ubuntu:22.04
    tty: true
    stdin_open: true
    restart: no
```

2. `sleep`과 같은 명령어를 실행하여 프로세스를 유지합니다.

```
version: '3.8'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    restart: no
```

1

```
root@cd44655d4996:/code/pratice5# docker compose ls -a
NAME                STATUS                                CONFIG FILES
cloud_wave          exited(2), restarting(1), running(2) /code/pratice4/docker-compose.yaml,/code/pratice3/docker-compose.yaml
ex1                 exited(1), running(1)               /code/example1.yaml
pratice5            exited(1)                           /code/pratice5/docker-compose.yaml
root@cd44655d4996:/code/pratice5# docker compose ps -a
WARN[0000] /code/pratice5/docker-compose.yaml: `version` is obsolete
NAME                IMAGE                COMMAND                SERVICE  CREATED          STATUS              PORTS
pratice5-ubuntu-1   ubuntu:22.04         "/bin/bash"           ubuntu   29 seconds ago   Exited (0) 28 seconds ago
```

2

```
root@cd44655d4996:/code/pratice5# docker compose ps
WARN[0000] /code/pratice5/docker-compose.yaml: `version` is obsolete
NAME                IMAGE                COMMAND                SERVICE  CREATED          STATUS              PORTS
pratice5-ubuntu-1   ubuntu:22.04         "/bin/bash"           ubuntu   31 seconds ago   Up 6 seconds
root@cd44655d4996:/code/pratice5# docker compose ls
NAME                STATUS                                CONFIG FILES
cloud_wave          restarting(1), running(2) /code/pratice4/docker-compose.yaml,/code/pratice3/docker-compose.yaml
ex1                 running(1)               /code/example1.yaml
pratice5            running(1)                /code/pratice5/docker-compose.yaml
```

연습2

[연습] 환경변수를 이용하여 compose file 제어하기

환경변수를 통해 docker compose 파일을 설정하는 방법은 일반적으로 다음과 같습니다.

1. Host에서 환경변수 설정하기
2. 환경 변수파일(.env)를 사용하기
 - o CLI

다음 .env와 docker-compose.yaml을 이용하여 프로젝트를 빌드 후 실행합니다.

```
# .env
FROM=".env file"

# project-1.yaml
version: '3.8'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -C
      - echo 'env from "$FROM"'
    restart: no
```

docker compose up을 이용하여 실행하면 다음과 같이 .env의 환경변수가 기입된 것을 확인할 수 있습니다.

```
$ docker compose up
[+] Building 0.0s (0/0)

                                docker:desktop-linux

[+] Running 1/0
✓ Container example3-ubuntu-1 Recreated

                                0.0s

Attaching to example3-ubuntu-1
example3-ubuntu-1 | env from ".env file"
example3-ubuntu-1 exited with code 0
```

이번에는 host에서 다음 명령어를 통해 다음과 같이 환경변수를 설정한 뒤, 실행해보면 다음과 같은 결과를 얻을 수 있습니다.

```
$ export FROM="host"
$ docker compose up
[+] Building 0.0s (0/0)

                                docker:desktop-linux

[+] Running 1/0
✓ Container example3-ubuntu-1 Recreated

                                0.0s

Attaching to example3-ubuntu-1
example3-ubuntu-1 | env from host
example3-ubuntu-1 exited with code 0
```

환경변수에 대해 default 값을 설정하고 싶은 경우 다음과 같이 작성하면 됩니다.

- \${ENV:-DEFAULT_VALUE}

```
# project-1.yaml
version: '3.8'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -C
      - echo 'env from "$FROM"' && echo 'env from ${BY:-default}'
    restart: no
```

위의 docker-compose.yaml을 실행하면 다음과 같이 "default" 값이 출력된 것을 확인할 수 있습니다.

```
$ docker compose up
[+] Building 0.0s (0/0)

                                docker:desktop-linux

[+] Running 1/0
✓ Container example3-ubuntu-1 Recreated

                                0.0s

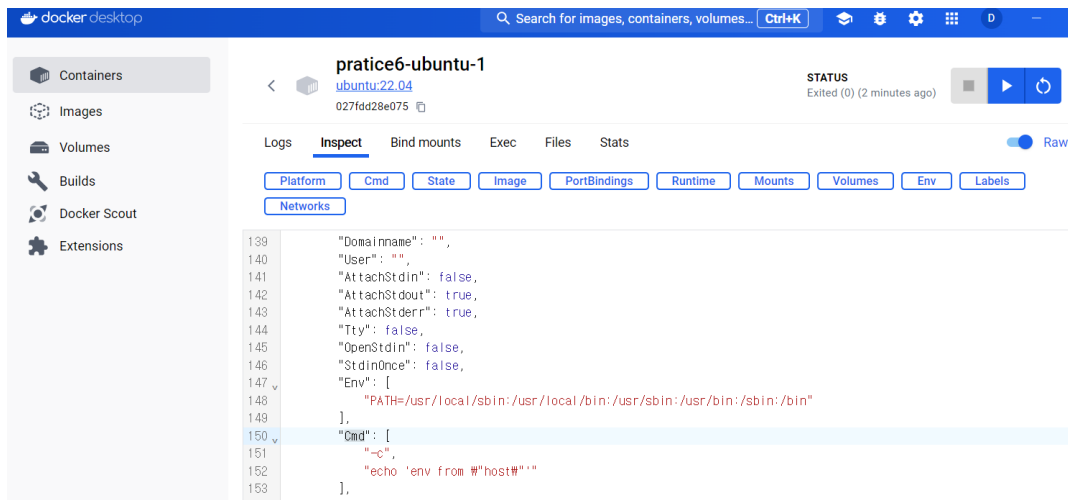
Attaching to example3-ubuntu-1
example3-ubuntu-1 | env from host
example3-ubuntu-1 | env from default
example3-ubuntu-1 exited with code 0
```

```

root@cd44655d4996:/code/pratice6# docker compose up
WARN[0000] /code/pratice6/docker-compose.yaml: `version` is obsolete
[+] Running 2/2
 ✓ Network pratice6_default      Created
 ✓ Container pratice6-ubuntu-1  Created
Attaching to ubuntu-1
ubuntu-1 | env from ".env file"
ubuntu-1 exited with code 0
root@cd44655d4996:/code/pratice6#

```

기본적으로 host에 있는 것을 가져가는데 지정하기 이전에 평문으로 전달이 된 것



연습3

[연습] command 에서 컨테이너 환경변수 사용하기

`docker-compose.yml` 파일에서 `$` 를 이용하여 환경변수를 사용하는 경우, 컨테이너 내부의 환경변수가 사용되지 않습니다.

`command` 에서 컨테이너 내부의 환경변수를 사용하고 싶은 경우 다음과 같이 `$$` 를 이용하면 됩니다.

```
version: '3.8'

services:
  ubuntu:
    image: ubuntu:22.04
    environment:
      - FROM="env definition"
    entrypoint: /bin/bash
    command:
      - -c
      - echo 'env from ${FROM}' && echo env from $$FROM
    restart: no
```

위의 `yml` 파일을 실행하면 다음과 같은 결과 값을 확인할 수 있습니다.

```
$ docker compose up
[+] Building 0.0s (0/0)

                                docker:desktop-linux
[+] Running 1/0
✓ Container example3-ubuntu-1  Recreated

                                0.0s
Attaching to example3-ubuntu-1
example3-ubuntu-1 | env from host
example3-ubuntu-1 | env from "env definition"
example3-ubuntu-1 exited with code 0
```

`docker inspect` 명령어를 통해서도 확인해볼 수 있습니다.

```
$ docker inspect example3-ubuntu-1 -f "{{ .Args }}"
[-c echo 'env from host' && echo env from FROM]
```



```

root@cd44655d4996:/code/pratice6# docker compose up
WARN[0000] /code/pratice6/docker-compose.yaml: `version` is obsolete
[+] Running 1/0
  ✓ Container pratice6-ubuntu-1 Created
Attaching to ubuntu-1
ubuntu-1 | env from vsc
ubuntu-1 | env from "env definition"
ubuntu-1 | env from default
ubuntu-1 exited with code 0

```

```

root@cd44655d4996:/code/pratice6# docker inspect pratice6-ubuntu-1 -f "{{ .Args }}"
[-c echo 'env from vsc' && echo env from ${FROM} && echo 'env from default']

```



yaml 내에 있는 환경변수를 사용할 경우 \$\${}를 무조건 사용해야..

inspect 찍었을 때 값으로 바뀐 형태인지? 명령어로 전달되는 형태인지가 가장 두드러지는 형태
확실히 구분할 수 있어야 한다.

연습4

[연습] Docker compose에서 build 사용하기

```
version: '3.8'

services:
  ubuntu:
    container_name: server
    build:
      context: .
      dockerfile_inline: |
        FROM ubuntu:22.04
        RUN apt-get update && apt-get upgrade && apt-get install -y curl
    image: cloudwave/compose:inline_build.v1
    restart: no
```

```
nginx:
  image: nginx:latest
  expose:
    - 80
  restart: always
```

```
$ docker compose -f example_1.yaml -p ex1 up -d --build
```

volume

2.2.4. volume

```
services:
  backend:
    image: example/database
    volumes:
      - db-data:/etc/data

  backup:
    image: backup-service
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
          read_only: true
      - db-data:/var/lib/backup/data:rw

volumes:
  db-data:
    name: "my-app-data"
    external: true
    labels:
      com.example.description: "Database volume"
```

db-data 도커 볼륨 사용 /가 없으니 절대경로 상대경로 x 호스트가 아니다.

볼륨은 자체만 마운트 가능 서브디렉토리 불가능

external =true 가 아니면 볼륨을 새로 하나 만든다.

reverse dns ⇒ com.example.description

익숙해져라 주소 거꾸로 적는다.

- volume 정의

볼륨 정의하기

```
volumes:
  db-data: # volume_key
    name: "db-volume"
    external: true
    labels:
      com.example.description: "Database volume"
```

Name

- 볼륨의 이름입니다.
- 설정되지 않은 경우 `{project_name}_{volume_key}` 형태의 이름으로 명명됩니다.

External

- 생성되어 있는 기존 볼륨의 사용 여부입니다.
- 해당 볼륨이 생성되어 있지 않은 경우, 프로젝트가 실행되지 않습니다.
- `name` 이 지정되어 있지 않은 경우, `volumes` 에 정의된 `key` 가 사용됩니다.
 - `db-volume` 이 명시되지 않았다면, `db_data` 란 이름을 가진 `volume` 을 탐색합니다.

Labels

- 관리를 위한 라벨을 설정합니다.

주의 사항

- `volume` 을 정의하였더라도 `service` 에서 사용하지 않는 경우, 생성되지 않습니다.

`volume_key`는 hash값 구분 편하게 하기 위해 이름 명시해서 만드는게 낫다

라벨링을 하는게 편하다.

`volume`을 정의해도 사용안하면 생성 x

- 볼륨 사용하기

볼륨 사용하기

```
services:
  backend:
    image: example/database
    volumes:
      - db-data:/etc/data

  backup:
    image: backup-service
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
          read_only: true
      - db-data:/var/lib/backup/data:rw
```

- 정의된 볼륨은 서비스의 `volumes`에서 사용할 수 있습니다.
- `Short Syntax`, `Long Syntax` 두 가지 방식으로 서비스의 `volumes`를 작성할 수 있습니다.

Short Syntax

```
volumes:
  - VOLUME:CONTAINER_PATH:ACCESS_MODE
```

Attributes	Description
<code>VOLUME</code>	host의 경로 또는 volume 이름을 사용합니다. volume 이름을 사용하는 경우 경로를 지정할 수 없습니다.
<code>CONTAINER_PATH</code>	볼륨이 마운트될 컨테이너의 경로를 의미합니다.
<code>ACCESS_MODE</code>	해당 볼륨의 Access mode를 설정합니다. - <code>rw</code> : 읽기&쓰기 모두 가능합니다. - <code>ro</code> : 읽기 전용으로 설정합니다.

Long Syntax

<https://docs.docker.com/compose/compose-file/05-services/#long-syntax-5>

```
volumes:
  - type: volume
    source: mydata
    target: /data
    read_only: true
```

Attributes	Description
<code>type</code>	해당 볼륨의 mount type을 설정합니다. - <code>volume</code> : <code>source</code> 로 volume을 사용합니다. - <code>bind</code> : <code>source</code> 로 host의 directory를 사용합니다.
<code>source</code>	volume 이름 또는 host의 directory 경로를 설정합니다.
<code>target</code>	컨테이너에서 볼륨이 mount될 directory 경로를 설정합니다.
<code>read_only</code>	읽기 전용 여부를 설정합니다.

연습1

연습 문제

[연습] External volume 사용하기

다음과 같이 `volume` 을 생성합니다.

```
$ docker volume create vault
vault
```

`volume` 을 `ubuntu` 컨테이너의 `/root/vault` 에 마운트하도록 `docker-compose.yaml` 을 작성합니다.

```
version: '3.8'
name: 'volume-external'

services:
  master:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    volumes:
      - vault:/root/vault

volumes:
  vault:
    external: true
    name: 'vault'
```

연습 2

[연습] read_only 로 설정하여 사용하기

다음 docker-compose.yaml 을 이용하여 프로젝트를 실행합니다.

```
# docker-compose.yaml
version: '3.8'
name: 'volume-external'

services:
  master:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    volumes:
      - vault:/root/vault:rw

  slave:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    volumes:
      - vault:/root/vault:ro

volumes:
  vault:
    external: true
    name: 'vault'
```

```
$ docker compose -f docker-compose.yaml up -d
[+] Building 0.0s (0/0)
docker:desktop-linux
[+] Running 2/2
✓ Container volume-external-master-1 Started
0.0s
✓ Container volume-external-slave-1 Started
0.0s
```

다음과 같이 `exec` 를 이용하여 `master` 서비스의 컨테이너에서 `/root/vault` 에 파일을 생성합니다.

```
$ docker exec volume-external-master-1 /bin/bash -c "echo master >
/root/vault/temp.txt"
```

생성한 파일이 정상적으로 읽여지는지 다음과 같이 확인해볼 수 있습니다.

```
$ docker exec volume-external-master-1 /bin/bash -c "cat /root/vault/temp.txt"

master
```

`slave` 서비스의 컨테이너에서는 `/root/vault` 에 저장한 파일을 읽을 수 있지만, 파일을 기록하는 것은 불가능한 것을 확인할 수 있습니다.

```
$ docker exec volume-external-slave-1 /bin/bash -c "cat /root/vault/temp.txt"

master
$ docker exec volume-external-slave-1 /bin/bash -c "echo slave >
/root/vault/temp.txt"
/bin/bash: line 1: /root/vault/temp.txt: Read-only file system
```


연습3

[연습] `volumes_from` 을 이용하여 `volume` 사용하기

<https://docs.docker.com/compose/compose-file/05-services/#volumes>

다음 파일을 이용하여 새로운 프로젝트를 실행합니다.

- 위에서 생성한 `volume-external` 프로젝트는 실행중이어야 합니다.

```
# docker-compose.yml
version: '3.8'
name: 'volume-external2'

services:
  other:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    volumes_from:
      - container: volume-external-slave-1:ro
```

```
$ docker compose -f docker-compose.yml up -d
[+] Building 0.0s (0/0)
docker:desktop-linux
[+] Running 2/2
✓ Network volume-external2_default    Created
    0.1s
✓ Container volume-external2-other-1  Started
    0.0s
```

별도의 `volumes` 를 정의하지 않고도 `volumes_from` 을 이용하여 다른 서비스의 `volume` 을 사용할 수 있는 것을 확인할 수 있습니다.

```
$ docker exec volume-external2-other-1 /bin/bash -c "cat /root/vault/temp.txt"
master
$ docker exec volume-external2-other-1 /bin/bash -c "echo other >
/root/vault/temp.txt"
/bin/bash: line 1: /root/vault/temp.txt: Read-only file system
```

```
volumes_from:
  - container:ex3-master-1:rw
```

```
✓ Container from_test-other-1 Running
⊗ root@cd44655d4996:/code/pratice9# docker exec from_test-other-1 /bin/bash -c "echo other > /root/vault/temp.txt"
/bin/bash: line 1: /root/vault/temp.txt: Read-only file system
● root@cd44655d4996:/code/pratice9# docker compose -p from_test up -d
WARN[0000] /code/pratice9/docker-compose.yaml: `version` is obsolete
[+] Running 1/1
✓ Container from_test-other-1 Started
● root@cd44655d4996:/code/pratice9# docker exec from_test-other-1 /bin/bash -c "cat /root/vault/temp.txt"
master
● root@cd44655d4996:/code/pratice9# docker exec from_test-other-1 /bin/bash -c "echo other >> /root/vault/temp.txt"
● root@cd44655d4996:/code/pratice9# docker exec from_test-other-1 /bin/bash -c "cat /root/vault/temp.txt"
master
other
```

네트워크

2.2.5. network

```
services:
  frontend:
    image: example/webapp
    networks:
      - private

networks:
  private:
    name: "private_net"
    external: true
    labels:
      com.example.description: "private network"
```

- 네트워크 정의하기

네트워크 정의하기

```
networks:
  private: # network_key
    name: "private_net"
    external: true
    labels:
      com.example.description: "private network"
```

Name

- 네트워크의 이름입니다.
- 설정되지 않은 경우 `{project_name}_{network_key}` 형태의 이름으로 명명됩니다.

External

- 생성되어 있는 기존 네트워크 사용 여부입니다.
- 해당 네트워크가 생성되어 있지 않은 경우, 프로젝트가 실행되지 않습니다.
- `name` 이 지정되어 있지 않은 경우, `networks` 에 정의된 `key` 가 사용됩니다.
 - `private_net` 이 명시되지 않았다면, `private` 란 이름을 가진 `network` 를 탐색합니다.

Labels

- 관리를 위한 라벨을 설정합니다.

주의 사항

- `network` 를 정의하였더라도 `service` 에서 사용하지 않는 경우, 생성되지 않습니다.

- 네트워크 사용하기

네트워크 사용하기

```
services:
  backend:
    image: example/database
    # List Syntax
    networks:
      - private

  backup:
    image: backup-service
    # Map Syntax
    networks:
      private:
```

```
aliases:
  - alias1

bastion:
  image: bastion
  network_mode: host
```

- 정의된 볼륨은 서비스의 `networks` 에서 사용할 수 있습니다.
- `host` 또는 `none` 네트워크를 사용하려는 경우, `network_mode` 를 이용하여 설정합니다.

주의사항

- 1개 이상의 컨테이너에서 동일한 `alias` 를 사용할 경우, 응답할 컨테이너를 특정지을 수 없습니다.

연습 1

연습 문제

[연습] host 네트워크를 사용하는 컨테이너 만들기

docker-compose.yaml 을 이용하여 host 를 사용하는 컨테이너를 실행합니다.

```
# docker-compose.yaml
version: '3.8'
name: 'network-host'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    network_mode: host
```

```
$ docker compose up -d
[+] Building 0.0s (0/0)
docker:desktop-linux
[+] Running 1/0
✓ Container network-host-ubuntu-1 Created
0.0s
```

테스트를 위해서 다음과 같이 nginx 컨테이너를 실행합니다.

```
$ docker run -d -p 80:80 --name nginx nginx:latest
```

ubuntu 컨테이너에 curl 을 설치합니다.

```
$ docker exec network-host-ubuntu-1 /bin/bash -c "apt-get update && apt-get
upgrade && apt-get install -y curl"
```

다음과 같이 컨테이너 내부에서 localhost:80 으로 nginx 응답이 오는 것을 확인합니다.

```
$ docker exec network-host-ubuntu-1 /bin/bash -c "curl localhost:80"
<!DOCTYPE html>
<html>
<head>
<title>welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

연습2

[연습] 이미 생성된 네트워크 사용하기

bridge 드라이버를 사용하는 네트워크를 생성합니다.

```
$ docker network create private -d bridge
1be27c483205a3b2dfbdf4ae36c2cc20ae0a53bc939eb7bd050a65574a9b9299

$ docker network ls -f name=private
NETWORK ID      NAME      DRIVER      SCOPE
1be27c483205    private  bridge      local
```

다음 compose 파일을 이용하여 private 네트워크를 사용하는 프로젝트를 실행합니다.

```
# docker-compose.yml
version: '3.8'
name: 'network-external'

services:
```

```
ubuntu:
  image: ubuntu:22.04
  entrypoint: /bin/bash
  command:
    - -c
    - sleep infinity
  networks:
    private:
      private:
        external: true
```

```
$ docker compose up -d

[+] Building 0.0s (0/0)

docker:desktop-linux

[+] Running 1/1
✓ Container network-external-ubuntu-1 Started
```

다음과 같이 name 을 제거하여도 프로젝트가 정상적으로 실행됩니다.

```
# docker-compose.yml
version: '3.8'
name: 'network-external'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    networks:
      private:
        private:
          external: true

networks:
  private:
    private:
      external: true
```

```
$ docker compose up -d

[+] Building 0.0s (0/0)

docker:desktop-linux

[+] Running 1/1
✓ Container network-external-ubuntu-1 Started
```

하지만, name 을 my-private 로 변경하면 다음과 같이 프로젝트가 실행되지 않는 것을 확인할 수 있습니다.

```
version: '3.8'
name: 'network-external'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    networks:
      private:
        private:
          external: true

networks:
  private:
    name: "my-private"
    private:
      external: true
```

```
$ docker compose up -d
[+] Building 0.0s (0/0)

docker:desktop-linux
network my-private declared as external, but could not be found
```


연습3

[연습] alias 설정하기

다음 `docker-compose.yml` 을 이용하여 프로젝트를 생성합니다.

```
# docker-compose.yml
version: '3.8'
name: 'network-alias'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    networks:
      private:

networks:
  private:
```

`inspect` 를 이용하여 컨테이너에 설정된 `alias` 를 확인하면 다음과 같이 3개가 표기되는 것을 확인할 수 있습니다.

- 컨테이너 이름
- 서비스 이름
- 컨테이너 ID

```
$ docker inspect network-alias-ubuntu-1 | jq ".[0].NetworkSettings.Networks" | jq
-r '.[].Aliases'
[
  "network-alias-ubuntu-1",
  "ubuntu",
  "d1295a0aae14"
]
```

다음과 같이 `docker-compose.yml` 에 `alias` 를 추가하여 업데이트 해줍니다.

```
version: '3.8'
name: 'network-alias'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    networks:
      private:
        aliases:
          - server

networks:
  private:
```

```
$ docker compose up -d
```

그러면 다음과 같이 기존에 있던 `alias` 에 추가로 설정한 `server` 가 추가된 것을 확인해볼 수 있습니다.

```
$ docker inspect network-alias-ubuntu-1 | jq ".[0].NetworkSettings.Networks" |
jq -r '.[].Aliases'
[
  "network-alias-ubuntu-1",
  "ubuntu",
  "server",
  "1679779c1978"
]
```

```
docker inspect network-alias-ubuntu-1 | jq ".[0].NetworkSettings.Networks" | jq -r
```

위로 inspect 명령어 수정

실습

[실습] 생성된 network 를 이용하여 서로 다른 프로젝트의 서비스와 연결하기

- across_project 네트워크를 생성하세요.
- network-across-project-1 프로젝트를 생성하세요
 - ubuntu:22.04
 - curl 이 설치되어 있어야합니다.
 - across_project 네트워크의 alias를 main 으로 설정하세요
- network-across-project-2 프로젝트를 생성하세요
 - nginx:latest
 - 80 포트를 expose
 - across_project 네트워크의 alias를 web 으로 설정하세요
- docker network inspect 를 이용하여 각 서비스의 IP를 확인하세요

```
$ docker network inspect across_project | jq '.[].Containers'
```

- ubuntu 서버에서 curl을 이용하여 web 을 호출하세요
 - IP
 - DNS(alias)

1 프로젝트 yaml

```
version: "3.8"
name: 'network-across-project-1'
services:
  ubuntu:
    # build:
    #   context: ..
    #   dockerfile_inline: |
    #     FROM ubuntu:22.04
    #     RUN apt-get update && apt-get upgrade && apt-get install -y curl
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
```

```

- -c
- apt-get update && apt-get upgrade && apt-get install -y curl && sleep infi
#- -c
#- sleep infinity
networks:
  p1:
    aliases:
      - main

networks:
  p1:
    name: "across_project"
    external: true

```

2 프로젝트 yaml

```

version: "3.8"
name: 'network-across-project-2'
services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    networks:
      p2:
        aliases:
          - web

networks:
  p2:
    name: "across_project"
    external: true

```

host 내에선 접근을 시도해보아도 같은 네트워크 내가 아니기 때문에 접속 x

docker exec network-across-project-1-ubuntu-1 /bin/bash -c "curl 172.26.0.2/16" 이렇게 날려야..

윈도우 vs환경에서 날리는건 host에서 날리는거와 동일

Containers [Give feedback](#)


























Container CPU usage ⓘ

4.05% / 1600% (16 CPUs available)

Container memory usage ⓘ

549.39MB / 15.1GB

[Show charts](#)

<input type="checkbox"/>	Name	Image	Status	Port(s)	CPU (%)	Last start... ↓	Actions
<input type="checkbox"/>	>  cloud_wa		Running (2/5)		0%	3 seconds ago	  
<input type="checkbox"/>	▼  network-		Running (1/1)		0%	20 minutes ago	  
<input type="checkbox"/>	 ubuntu 324904 ubuntu:22.04		Running		0%	20 minutes ago	  
<input type="checkbox"/>	▼  network-		Running (1/1)		0%	33 minutes ago	  
<input type="checkbox"/>	 nginx- 73fe37c nginx:latest		Running	80:80 	0%	33 minutes ago	  
<input type="checkbox"/>	>  from_tes		Running (1/1)		0%	3 hours ago	  

Showing 14 items

해당 container에서 ubuntu 서버 exec로 직접 접근한뒤 curl web or curl IP or curl 축약버전 id로 가능하다.

또한 기본 포트가 80이라 축약되었지만 curl web:80을 통해 보내도 동일하다.

config & secret

2.2.6. config & secret

```
services:
  redis:
    image: redis:latest
    configs:
      - http_config
      - source: my_config
        target: /redis_config
        uid: "103"
        gid: "103"
        mode: 0440 # Octal notation
    secrets:
      - source: server-certificate
        target: server.cert
        uid: "103"
        gid: "103"
        mode: 0440 # Octal notation
  configs:
    http_config:
      file: ./httpd.conf
  secrets:
    server-certificate:
      file: ./server.cert
```

Config Vs. Secret

	Config	Secret
목적	외부에서 서버 설정을 위한 변수를 제공	비밀번호와 같은 민감한 데이터를 제공
기본 Mount 위치	<code>/<config-name></code>	<code>/run/secrets/<secret-name></code>
암호화 여부	X	O (Swarm 을 사용하는 경우) X (그 외)

Config Vs. Mount(Volume)

- 컨테이너 외부에서 파일을 제공한다는 점에서는 두개는 유사합니다.
- Volume 과는 달리 Config 는 Mode 를 이용하여 파일의 권한을 세부적으로 관리할 수 있습니다.

연습

연습 문제

[연습] config를 특정 디렉토리에 mount하기

다음 명령어를 이용하여 password.txt를 생성합니다.

임의의 텍스트 파일을 메모장등을 이용해서 생성해도 무방합니다.

```
# Window - CMD
$ echo %RANDOM% > password.txt

# Linux
$ openssl rand -base64 14 > password.txt
```

다음 docker-compose.yml을 이용하여 프로젝트를 실행합니다.

```
# docker-compose.yml
version: '3.8'
name: 'config-mount'

services:
  ubuntu:
    image: ubuntu:22.04
    entrypoint: /bin/bash
    command:
      - -c
      - sleep infinity
    configs:
      - source: password
        target: /root/web_password.txt
```

```
configs:
  password:
    file: password.txt
```

docker exec를 통해 config 파일을 확인해볼 수 있습니다.

```
$ docker exec config-mount-ubuntu-1 /bin/bash -c "cat /root/web_password.txt &&
ls -al /root/"
6M9tLBo4u8EEIbb6Pis=
total 20
drwx----- 1 root root 4096 Dec 28 07:58 .
drwxr-xr-x 1 root root 4096 Dec 28 07:58 ..
-rw-r--r-- 1 root root 3106 Oct 15 2021 .bashrc
-rw-r--r-- 1 root root 161 Jul 9 2019 .profile
-rw-r--r-- 1 root root 21 Dec 28 07:51 web_password.txt
```



```
root@cd44655d4996:/code/project1# docker exec config-mount-server-1 /bin/bash -c "cat /root/web_password.txt && ls -al /root/"
26279
total 16
drwx----- 1 root root 4096 Jul 11 06:51 .
drwxr-xr-x 1 root root 4096 Jul 11 06:51 ..
-rw-r--r-- 1 root root 3106 Oct 15 2021 .bashrc
-rw-r--r-- 1 root root 161 Jul 9 2019 .profile
-rwxrwxrwx 1 root root 8 Jul 11 06:44 web_password.txt
```

cmd에서 해야한다

compose file 고급

2.3. Compose file - Advance

2.3.1. anchor & Alias

<https://docs.docker.com/compose/compose-file/11-extension/>

```
version: '3.8'

x-common:
  &common
  restart: always
  volumes:
    - source:/code
  environment:
    &default-env
    BY: "x-common"

x-value: &v1 x

services:
  ubuntu:
    <<: *common
    image: ubuntu:22.04
    environment:
      <<: *default-env
      FROM: "env definition"
      X: *v1
    entrypoint: /bin/bash
    command:
      - -c
      - echo 'env from ${FROM}' && echo env from ${BY}
    restart: no
```

```
volumes:
  source:
```

- Anchor & Alias 를 이용하여 반복적으로 사용되는 요소들을 모듈화하여 사용할 수 있습니다.
- 재사용을 위한 모듈은 x- 를 prefix 로 사용하여야 합니다.
- <<: 를 사용하면 yaml 을 병합하는 형태로 사용하게 됩니다.

x- 를 붙이고 설정하면 된다.

<< 표시는 아래에 있는거 전부 가져온다.

*는 값만 가져온다

<<: *common은

```
restart: always
volumes:
- source:/code
environment:
&default-env
BY: "x-common"
```

가져온다

restart: always는 무시된다.

environment 또한 무시된다. 이미 있기 때문

하지만 BY는 가져온다

살아남는건 volumes만

연습

[예시] Anchor & Alias를 이용한 YAML 파일 확인하기

위에 있는 `docker-compose.yaml`를 대상으로 `docker compose config`를 사용하면, 다음과 같이 Anchor & Alias가 반영된 YAML을 확인할 수 있습니다.

```
$ docker compose config
WARN[0000] The "FROM" variable is not set. Defaulting to a blank string.
name: compose
services:
  ubuntu:
    command:
      - -C
      - echo 'env from ' && echo env from ${BY} && echo env from alias ${X}
    entrypoint:
      - /bin/bash
    environment:
      BY: x-common
      FROM: env definition
      X: x
    image: ubuntu:22.04
    networks:
      default: null
    restart: "no"
    volumes:
      - type: volume
        source: source
        target: /code
        volume: {}
networks:
  default:
    name: compose_default
volumes:
  source:
    name: compose_source
x-common:
  environment:
    BY: x-common
  restart: always
  volumes:
    - source:/code
x-value: x
```

프로필

2.3.2. profile

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

<https://docs.docker.com/compose/compose-file/15-profiles/>

```
version: '3.8'

services:
  postgres:
    image: postgres:16.1-bullseye
    environment:
      - POSTGRES_PASSWORD=mysecretpassword

  server:
    image: ubuntu:22.04
    stdin_open: true # docker run -i
    tty: true        # docker run -t
    depends_on:
      - postgres

  pgadmin:
    image: dpage/pgadmin4:7.4
    environment:
      - PGADMIN_DEFAULT_EMAIL=user@sample.com
      - PGADMIN_DEFAULT_PASSWORD=SuperSecret
    depends_on:
      - postgres
      - server
    profiles:
      - debug
```

최근에 나온거라 안쓸 수 있다.

연습

[예시] profile 을 이용하여 일부 서비스만 실행하기

```
# docker-compose.yml
version: '3.8'

services:
  postgres:
    image: postgres:16.1-bullseye
    environment:
      - POSTGRES_PASSWORD=mysecretpassword

  server:
    image: ubuntu:22.04
    stdin_open: true # docker run -i
    tty: true        # docker run -t
    depends_on:
      - postgres

  pgadmin:
    image: dpage/pgadmin4:7.4
    environment:
```

```
      - PGADMIN_DEFAULT_EMAIL=user@sample.com
      - PGADMIN_DEFAULT_PASSWORD=SuperSecret
    depends_on:
      - postgres
      - server
    profiles:
      - debug
```

--profile 없이 프로젝트를 실행시키면 다음과 같이 profiles 가 선언되지 않은 2개의 서비스만 실행 되는 것을 확인할 수 있습니다.

```
$ docker compose -f docker-compose.yml up -d
[+] Building 0.0s (0/0)                                docker-
container:multi-arch-builder
[+] Running 4/4
✓ Network compose_default      Created           0.1s
✓ Container compose-postgres-1 Started         0.1s
✓ Container compose-server-1   Started         0.1s
```

debug 프로파일을 이용하여 프로젝트를 재실행하면, 기존에 생성되지 않았던 pgadmin 서비스가 실행 되는 것을 확인할 수 있습니다.

```
$ docker compose -f docker-compose.yml --profile debug up -d
[+] Building 0.0s (0/0)                                docker-
container:multi-arch-builder
[+] Running 3/3
✓ Container compose-postgres-1 Running           0.0s
✓ Container compose-server-1   Running           0.0s
✓ Container compose-pgadmin-1   Started          0.1s
```

```

root@cd44655d4996:/code/pratice12# docker compose up -d
WARN[0000] /code/pratice12/docker-compose.yaml: `version` is obsolete
[+] Running 3/3
 ✓ Network profile_test_default      Created
 ✓ Container profile_test-postgres-1 Started
 ✓ Container profile_test-server-1   Started
root@cd44655d4996:/code/pratice12# docker compose --profile debug up -d
WARN[0000] /code/pratice12/docker-compose.yaml: `version` is obsolete
[+] Running 3/3
 ✓ Container profile_test-postgres-1 Running
 ✓ Container profile_test-server-1   Running
 ✓ Container profile_test-pgadmin-1   Started

```

deploy

2.3.3. deploy

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

```

deploy:
  resources:
    limits:
      cpus: '0.50'
      memory: 50M
      pids: 1
    reservations:
      cpus: '0.25'
      memory: 20M
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s
  update_config:
    parallelism: 2

```

```

delay: 10s
order: stop-first

```

하드웨어 기준점 limit을 넘어가면 kill을 날린다 죽이는 기준
reservation은 여유값 계산

restart_policy 재시작 옵션 자세히 명세
3번 까지가 최대 시도 텀 딜레이를 5초

update할 때 최대 2개까지의 서버만 모두 다 멈추면 서비스가 멈추니까
stop first 리소스가 부족하니까 서버를 먼저 내리고 다시 올린다.

예시 1

[예시] 서비스를 컨테이너 3개로 구성하기

다음 `docker-compose.yml` 을 이용하여 프로젝트를 실행합니다.

```
# docker-compose.yml
name: deploy-replica
services:
  web:
    image: nginx:latest
    expose:
      - 80
    deploy:
      replicas: 3
```

```
$ docker compose up -d
[+] Building 0.0s (0/0)
                                docker:desktop-linux
[+] Running 3/3
✓ Container deploy-replica-web-3 Started 0.1s
✓ Container deploy-replica-web-2 Started 0.1s
✓ Container deploy-replica-web-1 Started 0.1s
```

다음과 같이 `web` 서비스가 `nginx` 컨테이너 3개로 구성되어진 것을 확인할 수 있습니다.

```
$ docker compose -p deploy-replica ps
```

NAME	STATUS	IMAGE	COMMAND	SERVICE	CREATED
deploy-replica-web-1	up 46 seconds	nginx:latest	"/docker-entrypoint..."	web	47
deploy-replica-web-2	up 46 seconds	nginx:latest	"/docker-entrypoint..."	web	47
deploy-replica-web-3	up 46 seconds	nginx:latest	"/docker-entrypoint..."	web	47

하지만, 다음과 같이 `container_name` 을 설정하게 되면 실행시 에러가 발생하는 것을 확인할 수 있습니다.

```
# docker-compose.yml
name: deploy-replica
services:
  web:
    container_name: "web"
    image: nginx:latest
    expose:
      - 80
    deploy:
      replicas: 3
```

```
$ docker compose up -d
[+] Building 0.0s (0/0)
                                docker:desktop-linux
WARNING: The "web" service is using the custom container name "web". Docker
requires each container to have a unique name. Remove the custom name to scale
the service.
```

예시 2

[예시] resource 할당하기

다음 `docker-compose.yml` 을 이용하여 프로젝트를 실행합니다.

```
# docker-compose.yml
name: deploy-replica
services:
  web:
    image: nginx:latest
    expose:
      - 80
    deploy:
      replicas: 1
      resources:
        limits:
          memory: 500M
```

```
$ docker compose up -d
[+] Building 0.0s (0/0)
                                docker:desktop-linux
[+] Running 3/3
✓ Container deploy-replica-web-1 started
                                0.6s
```

`docker stats` 를 이용하여 메모리 제한을 확인할 수 있습니다.

```
$ docker stats deploy-replica-web-1
```

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET
e07340054e9a	deploy-replica-web-1	0.00%	5.805MiB / 500MiB	1.16%	
806B / 0B	0B / 12.3kB	7			

depend_on

2.3.4. depend_on

<https://docs.docker.com/compose/startup-order/>

https://docs.docker.com/compose/compose-file/05-services/#depends_on

```
# docker-compose.yml
version: '3.8'

services:
  postgres:
    image: postgres:16.1-bullseye
    environment:
      - POSTGRES_PASSWORD=mysecretpassword

  server:
    image: ubuntu:22.04
    stdin_open: true
    tty: true
    # Short Syntax - List
    depends_on:
      - postgres

  pgadmin:
    image: dpage/pgadmin4:7.4
    environment:
      - PGADMIN_DEFAULT_EMAIL=user@sample.com
      - PGADMIN_DEFAULT_PASSWORD=SuperSecret
    # Long Syntax - Map
    depends_on:
      postgres:
        condition: service_started
        restart: false
      server:
        condition: service_started
    profiles:
      - debug
```

Long Syntax

Attributes	Description
<code>condition</code>	상위 서비스의 상태 조건을 정의합니다. - <code>service_started</code> : 서비스가 실행된 상태 - <code>service_healthy</code> : 서비스가 <code>healthy</code> 인 상태 - <code>service_completed_successfully</code> : 서비스가 실행 완료된 상태
<code>restart</code>	상위 서비스가 업데이트된 경우, 서비스를 재실행합니다.
<code>required</code>	<code>false</code> 인 경우 상위 서비스가 실행되지 않았더라도 서비스를 실행합니다.

어떠한 서비스가 정상 실행되기 위해서 다른 서비스의 실행의 선행이 필요할 경우 `depends_on` 사용

health check

2.3.5. health check

<https://docs.docker.com/engine/reference/builder/#healthcheck>

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 1m30s
  timeout: 10s
  retries: 3
  start_period: 40s
  start_interval: 5s
```

- `healthcheck` 를 이용하여 서비스의 상태를 확인할 수 있습니다.
- 시간은 `{value}{unit}` 형태로 작성하며, 지원하는 시간 단위는 다음과 같습니다.
 - `us(microseconds)`
 - `ms(millisecons)`
 - `s(seconds)`
 - `m(minutes)`
 - `h(hours)`

Attributes	Description
<code>test</code>	컨테이너 상태 체크를 위한 명령어를 설정합니다. <code>NONE</code> 으로 설정한 경우 <code>Dockerfile</code> 에 정의된 <code>HEALTHCHECK</code> 를 비활성화합니다.
<code>interval</code>	상태 체크 간격을 의미합니다.
<code>timeout</code>	응답까지 걸리는 최대 시간을 의미합니다. 만약 해당 시간까지 응답이 도착하지 않으면 <code>fail</code> 로 판단합니다.
<code>start_period</code>	컨테이너 실행까지 소요되는 시간을 의미합니다. 해당 시간동안에는 상태가 <code>fail</code> 인 경우에도 카운트되지 않습니다.

Attributes	Description
<code>start_interval</code>	<code>start_period</code> 내에서의 체크 간격을 의미합니다.
<code>disable</code>	<code>HEALTHCHECK</code> 를 비활성화 합니다. <code>test: ["NONE"]</code> 과 동일합니다.



데브옵스라면 헬스체크가 매우중요하다. 이게 필수

쿠버네티스가 아닌 aws cloud더 라고 그렇다

dev ops 1순위

관측가능성

모니터링 : log , metric , health check

헬스 체크 : 사고를 미연에 방지

metric : 어디서 사고가 났는지 확인 주로 하드웨어 확인 메모리, cpu,

queue, iops, tps/kps

서버 구동에 예상되는 모든 값들

log 일정 time 이내에 에러가 몇 번 이상 발생하면 담당자에게 slack을 보내라

DB → 100000명 800,000 request /s (rps) 400,000 QPS

배포(deploy)

v1 → v2 작업이다

1) blue green

미리 stand by

업데이트하려는 서버가 장애가 난 경우 바로 셧다운 해야 함

업데이트하기 위해 현재 서버 수의 2배가 필요

바로 전환

2) rolling update

서버를 천천히 업데이트 시켜나감 모든 서버를 한번에 하는 것이 아닌 것

다만 고객들이 업데이트 중간에 재접속할 경우 서버가 다르기에 값이 다를 수 있음

3) canary

v1 에서 소수만 떼다 소수만 업데이트 시키고 문제 없는지 확인

문제 없으면 롤링 업데이트 문제 생기면 바로 롤백

롤링 업데이트의 업그레이드 버전