



# Docker+CI/CD - 3주차

## [목차]

### 00. 3주차 오늘 배울 것



목표

#### 01. Dockerfile로 나만의 Docker Image를 만들어요

[Dockerfile 이란?](#)

#### 02. Docker Compose로 여러 개의 컨테이너를 한 번에 관리해요

[Docker Compose란?](#)

#### 03. Docker 모니터링&로깅

[Container 리소스 모니터링](#)

[Container 로깅](#)

#### 3주차 끝



모든 토글을 열고 닫는 단축키

Windows :

`Ctrl + alt + t`

Mac :

`⌘ + ⌥ + t`

## 00. 3주차 오늘 배울 것



**Dockerfile과 Docker Compose 를 사용해서 나만의 앱을 여러 개 실행시켜요.**



목표

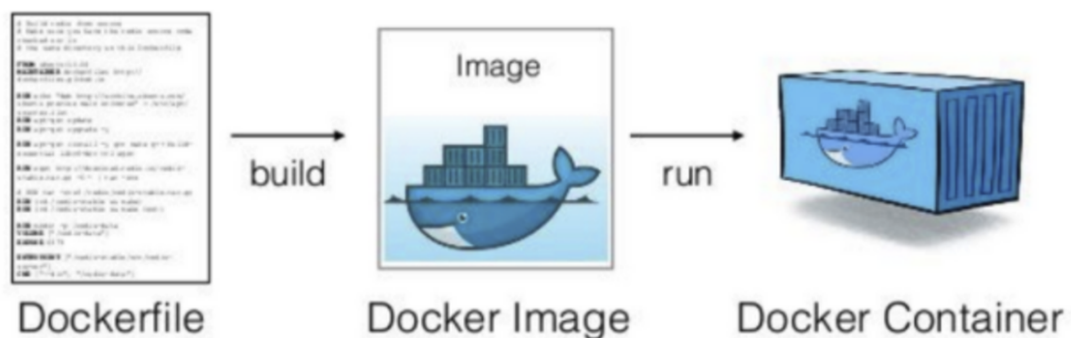
- Dockerfile 을 활용하여 나만의 이미지를 만들 수 있습니다.
- Docker Compose 를 활용하여 여러 개의 컨테이너를 관리할 수 있습니다.
- Docker 컨테이너를 모니터링하고 로깅을 확인할 수 있습니다.

# 01. Dockerfile로 나만의 Docker Image를 만들어요

## Dockerfile이란?

✓ Dockerfile에 대해 알아봅니다.

### ▼ Dockerfile



#### • Dockerfile이 뭐예요?

- Dockerfile은 컴퓨터에서 돌아가는 앱을 만들기 위한 레시피 같은 거예요. 이 레시피대로 하면 Docker 이미지라는 걸 만들 수 있어요. Docker 이미지는 앱을 실행하는 데 필요한 모든 것을 담고 있죠.

#### • Dockerfile 사용하는 이유는?

- 앱을 컨테이너로 만들 때 이미지를 만드는 용도로 Dockerfile을 써요. 이렇게 하면 앱이 필요로 하는 모든 것을 한 곳에 담을 수 있어요.
- 누구나 Dockerfile을 보고 똑같은 앱 환경을 쉽게 만들 수 있어요. 마치 요리 레시피를 따라 하는 것처럼요.
- Dockerfile을 작성하면, 앱을 만드는 과정을 자동화할 수 있어요. 그래서 매번 똑같은 방식으로 앱을 만들고 배포할 수 있어요.

Dockerfile 덕분에 앱을 만드는 일이 더 쉽고, 누구나 똑같은 결과를 얻을 수 있어요. 컴퓨터에서 앱을 실행하는 일이 마치 레시피대로 요리하는 것처럼 간단해져요.

## ▼ ▶ Dockerfile 예제

- Dockerfile은 다음과 같은 형식으로 작성되요

```
# Dockerfile
FROM ubuntu:latest
MAINTAINER Your Name <your-email@example.com>
RUN apt-get update && apt-get install -y nginx
COPY index.html /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

- 위 Dockerfile은 Ubuntu 최신 버전을 기반으로 Nginx를 설치하고, index.html 파일을 Nginx의 HTML 디렉토리에 복사하는 예시예요
- Dockerfile은 다음과 같은 명령어를 포함해요. 자세한 건 뒤에서 다룰게요.
  - FROM: 베이스 이미지를 선택
  - MAINTAINER: 이미지를 만든 사람의 정보를 입력
  - RUN: 이미지에 명령을 실행하여 파일을 추가하거나 삭제
  - COPY: 파일을 이미지에 복사
  - EXPOSE: 컨테이너가 노출할 포트를 설정
  - CMD: 컨테이너가 실행될 때 실행할 명령을 설정
- Docker 이미지를 생성하려면 Docker CLI를 사용하여 다음과 같은 명령을 실행해요.

```
docker buildx build -t my-nginx:latest .
docker build -t my-nginx:latest . # 위 명령이 실행되지 않는 경우 실행
```

- 위 명령어는 현재 디렉토리에서 Dockerfile을 기반으로 my-nginx:latest라는 이름의 Docker 이미지를 생성하는 예제예요.
- 이렇게 생성된 Docker 이미지는 Docker CLI를 사용하여 컨테이너로 실행할 수 있습니다. 예를 들어, 다음과 같은 명령어를 사용하여 컨테이너를 실행할 수 있어요.

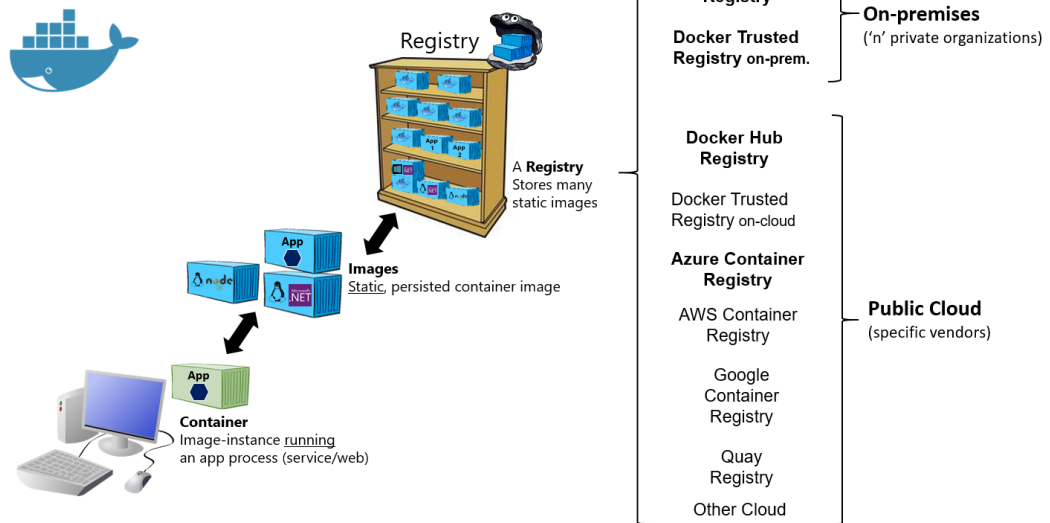
```
docker run -d -p 80:80 my-nginx:latest
```

- 위 명령어는 my-nginx:latest 이미지를 기반으로 컨테이너를 실행하고, 80번 포트를 호스트 머신의 80번 포트로 맵핑하는 예제예요.
- 이렇게 생성된 컨테이너는 Docker CLI를 사용하여 종료하거나 업데이트할 수 있습니다. 예를 들어, 다음과 같은 명령어를 사용하여 컨테이너를 종료할 수 있어요.

```
docker stop my-nginx
```

- 위 명령어는 my-nginx라는 이름의 컨테이너를 종료하는 예제예요.
- 이렇게 생성된 Docker 이미지는 Docker 레지스트리를 사용하여 다른 사용자와 공유할 수 있어요. Docker 레지스트리를 사용하면 Docker 이미지를 저장하고 공유할 수 있고, 다른 사용자가 이미지를 다운로드하여 사용할 수 있어요.

## Basic taxonomy in Docker



## ✓ Dockerfile 의 명령어에 대해 알아봅니다.

### ▼ Dockerfile 명령어

- **FROM:** 베이스 이미지를 지정
  - ex) `FROM ubuntu:22.04`

- **MAINTAINER:** Dockerfile을 작성한 사람의 정보를 입력
  - ex) `MAINTAINER naebaecaem <nbcamp@spartacoding.co>`
- **LABEL:** 이미지에 메타데이터를 추가
  - ex) `LABEL purpose='nginx test'`
- **RUN:** 이미지를 생성하는 동안 실행할 명령어를 입력
  - 사용자를 지정하지 않은 상태라면, root 로 실행
  - ex) `RUN apt update && apt upgrade -y && apt autoremove && apt autoclean`
  - ex) `RUN apt install openjdk-21-jdk`
- **CMD:** 컨테이너를 생성할 때, 실행할 명령어를 입력
  - 컨테이너를 생성할 때만 실행
  - 추가적인 명령어에 따라 설정한 해당 명령어 수정 가능
  - ex) `CMD ["nginx", "-g", "daemon off;"]`
- **ENTRYPOINT:** 컨테이너 시작할 때, 실행할 명령어를 입력
  - 컨테이너를 시작할 때마다 실행
  - 추가적인 명령어 존재 여부와 상관 없이 무조건 실행
  - ex) `ENTRYPOINT ["npm", "start"]`
- **ENV:** 환경 변수를 설정
  - 이미지 안에 각종 환경 변수를 지정
  - ex) `ENV STAGE staging`
  - ex) `ENV JAVA_HOME /usr/lib/jvm/java-8-oracle`
- **WORKDIR:** 작업 디렉토리를 지정
  - ex) `WORKDIR /app`
- **COPY:** 파일을 복사
  - 호스트의 파일이나 디렉토리를 이미지 안에 복사
  - Docker Context, 즉, 빌드 작업 디렉토리 내 파일만 복사 가능
  - ex) `COPY index.html /usr/share/nginx/html`
- **USER:** 사용자를 설정

- Container의 기본 사용자는 root 에요. root 권한이 필요 없는 application이라면 다른 사용자로 변경하여 사용해야 해요.

```
RUN ["useradd", "user"]
USER user
RUN ["/bin/bash", "-c", "ls"]
```

- **EXPOSE:** 컨테이너에서 노출할 포트를 지정

- ex) `EXPOSE 80`
- ex) `EXPOSE 443`

#### ▼ ▶ Docker파일 예제

- ▶ FastAPI 앱을 실행하는 예제

```
FROM python:3.11

RUN pip install pipenv

WORKDIR /app

ADD . /app/

RUN pipenv --python 3.11
RUN pipenv run pip install poetry
RUN pipenv sync
RUN pipenv run pip install certifi

ARG STAGE

RUN sh -c 'echo "STAGE=$STAGE" > .env'
RUN sh -c 'echo "PYTHONPATH=." >> .env'
RUN chmod +x ./scripts/run.sh
RUN chmod +x ./scripts/run-worker.sh

CMD [ "./scripts/run.sh" ]
```

- ▶ nginx 이미지를 생성하는 예제

- 실제로는 `nginx:latest` 이미지를 사용하면 되지만, 예제를 위해 만들어 보았어요.

```
# Dockerfile
FROM ubuntu:22.04
MAINTAINER your-name <your-email@example.com>
LABEL purpose=Web Server

# nginx 패키지 설치
RUN apt-get update && apt-get install -y nginx

# nginx 설정 파일 복사
COPY nginx.conf /etc/nginx/nginx.conf

# Nginx 실행
CMD ["nginx", "-g", "daemon off;"]
```

- nginx.conf

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include        /etc/nginx/mime.types;
    default_type   application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] $status $body_bytes_sent "$http_referer" "$http_user_agent" "$http_x_forwarded_for"';

    access_log    /var/log/nginx/access.log  main;
```

```

    sendfile            on;
    #tcp_nopush         on;

    keepalive_timeout  65;

    gzip  on;
    gzip_disable "msie6";

    include /etc/nginx/conf.d/*.conf;
}

```

## 02. Docker Compose로 여러 개의 컨테이너를 한 번에 관리해요

### Docker Compose란?



**Docker Compose** 를 통해 여러 개의 컨테이너를 관리합니다.

#### ▼ Docker Compose 를 사용하는 이유는?

- **편하게 설정하기:** Docker Compose는 여러 컨테이너를 한 파일에 적어서 설정할 수 있어요. 이 파일에는 컨테이너가 무슨 이미지를 쓸지, 어떤 포트를 사용할지, 환경 변수는 뭐가 필요한지 등을 적어둬요. 이렇게 하면 여러 컨테이너를 한 번에 쉽게 설정할 수 있죠.
- **자동으로 배포하기:** 설정 파일이 있으면, Docker Compose가 알아서 컨테이너들을 만들어 주고 실행해 줘요. 개발자가 일일이 명령어를 입력할 필요가 없어요.
- **의존성 관리:** 컨테이너들이 서로 의존하는 관계가 있으면, Docker Compose가 이를 관리해 줘요. 예를 들어, A 컨테이너가 B 컨테이너를 필요로 하면, A를 먼저 켜고 나서 B를 실행하는 식이죠.
- **모니터링과 로깅:** Docker Compose는 컨테이너들이 어떻게 돌아가는지 지켜보고, 로그도 모아줘요. 이렇게 하면 문제가 생겼을 때 빨리 찾아서 고칠 수 있어요.
- **확장성:** 여러 컨테이너를 하나의 그룹으로 관리할 수 있어요. 이게 좋은 이유는, 예를 들어 웹 앱을 만드는 여러 컨테이너를 한꺼번에 관리하고 확장하기 쉽기 때문이에요.



- **유연성:** Docker Compose는 개발 환경, 테스트 환경, 실제 운영 환경에서도 같은 설정 파일을 써서 일관성을 유지할 수 있어요.
- **보안 강화:** 컨테이너들의 네트워크를 분리해서 외부로부터의 접근을 제한할 수도 있어요. 이렇게 하면 보안이 더 강화돼요.
- **유지보수가 쉬워요:** 설정 파일 하나로 컨테이너들을 관리하기 때문에, 뭔가 바꿀 일이 있으면 파일만 수정하면 돼요. 그러면 Docker Compose가 알아서 변경사항을 적용해 줘요.

Docker Compose를 사용하면 여러 컨테이너를 더 쉽게 관리하고, 자동으로 설정하고, 확장하고, 보안을 강화할 수 있어요. 개발자 입장에서는 이런 도구가 엄청나게 편리하죠!

#### ▼ 어디에 사용하나요?

##### • 개발 환경에서

- 앱을 개발할 때, 앱을 따로 떼어 놓고 실행하고 테스트할 수 있는 환경이 필요해요. Docker Compose를 쓰면 이런 환경을 쉽게 만들고 관리할 수 있어요.
- Compose 파일은 앱이 필요로 하는 모든 서비스들(데이터베이스, 큐, 캐시, 웹 API 등)을 정리해주고, `docker compose up` 명령어로 이 모든 것을 한 번에 시작할 수 있어요.
- 이런 기능들 덕분에 개발자가 새 프로젝트를 시작할 때 시간을 많이 절약할 수 있어요. 여러 페이지에 걸친 설명서 대신에 Compose 파일 하나로 모든 설정을 할 수 있으니까요.

##### • 자동화된 테스트 환경에서

- 자동화된 테스트는 앱이 잘 돌아가는지 확인하는 데 중요해요. Docker Compose는 이런 테스트를 위한 별도의 환경을 쉽게 만들고 없앨 수 있어요.
- Compose 파일에 테스트 환경을 정의해두고, 간단한 명령어 몇 개로 테스트 환경을 만들고 테스트를 실행한 다음, 다시 환경을 없앨 수 있어요. 예를 들어 다음과 같은 명령어를 사용하면 돼요.

```
docker compose up -d
./run_tests
docker compose down
```

##### • 단일 호스트 배포에서

- Docker Compose는 주로 개발과 테스트에 많이 쓰이지만, 실제로 앱을 운영하는 환경(프로덕션)에도 쓸 수 있어요. 매번 새 버전이 나올 때마다 이런 용도로도 쓰기 좋게 계속 개선되고 있어요.

#### ▼ Docker Compose의 특징점

- **한 번에 여러 컨테이너 설정하기:**

- Docker Compose는 여러 컨테이너의 설정을 하나의 YAML 파일에 넣어서 관리해요. 이 파일 하나로 여러 컨테이너의 모든 환경을 설정하고, 그걸로 여러 컨테이너를 한 번에 실행할 수 있죠.

- **빠른 서비스 실행:**

- 설정 값들을 저장해 두고 다시 쓸 수 있어요. 만약 설정이 바뀌지 않았다면, Docker Compose는 이전에 저장해둔 정보를 다시 사용해서 서비스를 더 빨리 시작할 수 있어요.

- **같은 네트워크에서 쉽게 연결:**

- `docker-compose.yaml` 파일에 있는 애플리케이션들은 모두 같은 네트워크에 자동으로 연결돼요. 이렇게 하면 복잡한 네트워크 설정 없이도 여러 컨테이너가 서로 쉽게 통신할 수 있어요.

이런 특징들 덕분에 Docker Compose는 여러 컨테이너를 관리하고 실행하는 데 정말 편리한 도구예요. 설정도 간편하고, 빠르게 실행할 수 있고, 컨테이너들끼리의 연결도 쉽게 할 수 있죠.

#### ▼ Docker Compose 실행하기

1. 각 애플리케이션의 Dockerfile 작성하기

- 보통 내가 만든 애플리케이션을 실행하기 위한 Dockerfile 만 작성

2. `docker-compose.yaml` 파일 작성하기

- 내가 만든 애플리케이션을 실행하기 위해 필요한 database라든지 redis라든지 다른 서비스들을 한꺼번에 정의하는 파일을 작성

3. `docker compose up` 으로 실행하기

## ✓ YAML file을 설명합니다.

### ▼ YAML 파일이란?

#### • YAML 파일이란?

- YAML 파일은 컴퓨터가 읽을 수 있는 설정 파일이에요. 사람이 읽기에도 쉬운 텍스트 형식으로 되어 있죠. 'YAML Ain't Markup Language'의 줄임말이에요, 즉 'YAML은 마크업 언어가 아니다'라는 뜻이죠.

#### • 어떻게 생겼나요?

- YAML 파일은 일반 텍스트로 쓰여 있어요. 설정이나 데이터를 쉽게 알아볼 수 있는 형식으로 나열해요. 예를 들어, 목록이나 키-값 쌍 같은 것들이죠.

#### • 왜 쓰나요?

- YAML 파일은 설정을 정리하고 관리하기에 아주 좋아요. 예를 들어, Docker Compose에서는 YAML 파일을 사용해서 여러 컨테이너의 설정을 한 곳에 쉽게 정리할 수 있어요.
- YAML은 읽기 쉽고, 쓰기도 간단해서 많은 프로그램과 도구에서 선호하는 설정 파일 형식이에요.

#### • 특징은?

- YAML 파일은 구조가 명확하고 간단해서, 사람이 보기에 이해하기 쉬워요. 들여쓰기를 사용해서 각 설정의 관계를 나타내죠.

YAML은 다음과 같은 문법을 사용해요

- **키-값 쌍:** 키와 값으로 이루어진 쌍으로 구성됩니다. 키와 값은 콜론(:)으로 구분됩니다.
- **리스트:** 쉼표(,)로 구분된 값들의 리스트로 구성됩니다.
- **딕셔너리:** 중괄호({})로 둘러싸인 키-값 쌍의 리스트로 구성됩니다.
- **불린 값:** true, false, yes, no 등의 값으로 표현됩니다.
- **문자열:** 큰 따옴표(")나 작은 따옴표(')로 둘러싸인 문자열로 표현됩니다.
- **!! 주의사항 !!**

- 보통 들여쓰기가 잘못된 경우 yaml 파일을 의도와 다르게 해석하게 되니 들여쓰기에 주의하셔야 합니다.
- <https://www.yamllint.com/> 에서 들여쓰기를 검사할 수 있어요



### Docker Compose 파일 예제를 설명합니다.

#### ▼ Docker compose 파일 설명

##### ▼ 예제

```
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - 80:80
    volumes:
      - ./web:/usr/share/nginx/html
    depends_on:
      - api
    links:
      - api:api
  api:
    image: java:latest
    volumes:
      - ./api:/app
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis
      - MYSQL_HOST=mysql
      - MYSQL_USER=root
      - MYSQL_PASSWORD=password
      - MYSQL_DATABASE=test
    depends_on:
      - mysql
```

```

    - redis
links:
  - mysql:mysql
  - redis:redis
redis:
  image: redis:latest
  ports:
    - 6379:6379
mysql:
  image: mysql:latest
  ports:
    - 3306:3306
  volumes:
    - ./mysql:/var/lib/mysql
  environment:
    - MYSQL_ROOT_PASSWORD=password
    - MYSQL_DATABASE=test
    - MYSQL_USER=root
    - MYSQL_PASSWORD=password

```

▼ 위의 예제는 다음과 같은 내용을 포함하고 있습니다.

- **version** : Docker Compose의 버전을 정의합니다.
- **services** : 네 개의 서비스를 정의합니다.
  - **web** 서비스는 Nginx 이미지를 사용하며, 포트 80번을 호스트 머신에 노출합니다.
  - **api** 서비스는 Java 이미지를 사용하며, 포트 8080번을 호스트 머신에 노출합니다.
  - **redis** 서비스는 Redis 이미지를 사용하며, 포트 6379번을 호스트 머신에 노출합니다.
  - **mysql** 서비스는 MySQL 이미지를 사용하며 포트 3306번을 호스트 머신에 노출합니다.
- **volumes** : **web** 서비스는 현재 디렉토리 내 **web** 디렉토리를 컨테이너에 연결합니다. **api** 서비스는 현재 디렉토리 내 **api** 디렉토리를 컨테이너에 연결합니다. **mysql** 서비스는 현재 디렉토리 내 **mysql** 디렉토리를 사용하여 MySQL 데이터를 영구적으로 저장하게 합니다.

- `depends_on` : 각 서비스 간의 의존성을 정의합니다. `web` 서비스는 `api` 서비스에 의존합니다. `api` 서비스는 `mysql` 과 `redis` 서비스에 의존합니다.
- `links` : 각 서비스 간의 링크를 정의합니다. `web` 서비스는 `api` 서비스에 링크됩니다. `api` 서비스는 `mysql` 과 `redis` 서비스에 링크됩니다.
- 이 예제에서는 `web` 서비스와 `api` 서비스가 각각 독립적인 컨테이너로 실행됩니다. `web` 서비스는 `api` 서비스에 의존하며, `api` 서비스는 `mysql` 과 `redis` 서비스에 의존합니다. `redis` 와 `mysql` 서비스는 각각 독립적인 컨테이너로 실행됩니다.

- 코드 설명

```
version: "3"
services:
  service1:
    # ...service1 설정
  service2:
    # ...service2 설정
networks:
  # 네트워크 설정, 선택적
volumes:
  # 볼륨 설정, 선택적
```

- `version` : docker engine 의 버전과 연관
  - <https://docs.docker.com/compose/compose-file/compose-versioning/>

Compose file format	Docker Engine release
Compose specification	19.03.0+
3.8	19.03.0+
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+
2.3	17.06.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+

- **services** : 컨테이너 대신 서비스 개념으로 간주

- 예제

```

services:
  web:
    build:
      context: .      # Dockerfile 의 위치
      dockerfile: Dockerfile    # Dockerfile 파일명
    container_name: testapp_web_1    # 생략하는 경우
    # 자동으로 부여 docker run 의 --name 옵션과 동일
    ports: "8080:8080"    # docker run 의 -p 옵션과 동일
    expose: "8080"        # 호스트머신과 연결이 아니라
    # 링크로 연결된 서비스 간 통신이 필요할 때 사용
    networks: testnetwork    # networks 를 최상위에
    # docker run의 --net 옵션과 동일
    volumes: ../var/lib/nginx/html    # docker ru

```

```

environment:
  - APPENV=TEST    # docker run 의 -e 옵션과 동일
command: npm start # docker run 의 가장 마지막
restart: always    # docker run 의 --restart 옵션
depends_on: db      # 이 옵션에 지정된 서비스가 시작된 후
links: db          # Docker가 네트워크를 통해 컨테이너를 연결
                  # 컨테이너를 연결할 때 Docker는 환경 변수를 만들고
                  # 컨테이너를 알려진 호스트 목록에 추가하여 서로를 검색할
deploy:            # 서비스의 복제본 개수 등 지정
  replicas: 3
  mode: replicated

```

○ **volumes** :

- 도커 볼륨 혹은 호스트 볼륨을 마운트하여 사용합니다. 도커 볼륨의 경우 `docker-compose.yml` 파일에 선언된 볼륨만 `docker-compose.yml` 에서 사용할 수 있습니다.

```

version: "3.9"
services:
  web:
    # ...
    volumes:
      - README.md:/docs/README.md # 호스트의 README
      - logvolume01:/var/log # 선언된 도커 볼륨 logvolume01
    # ...
volumes:
  logvolume01: {} # 도커볼륨 logvolume01 선언

```

○ **networks** :

- 서비스(컨테이너)가 소속된 네트워크 입니다. 따로 지정하지 않을 경우 `default_${project}`와 같이 지정됩니다. 기본적으로 컨테이너는 같은 네트워크에 있어야 서로 통신이 가능합니다.

○ **healthcheck**

- 서비스 컨테이너가 "healthy"한지 계속 체크



- Dockerfile에 정의된 것을 먼저 따르지만 docker-compose 파일에서 재지정 가능
- 예제

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

### ▼ Docker Compose CLI

- `docker-compose [COMMAND] [SERVICES...]` 의 형태로 지정된 서비스(컨테이너)만 제어  
가 가능합니다. 예를 들어서 web, redis 중에 web만 기동하고 싶을 경우 `docker-  
compose up -d web` 와 같이 실행합니다.
- `docker-compose up`
  - `docker-compose up` 실행시 다음의 순서로 진행합니다. 이미 생성된 경우  
해당 단계를 건너뜁니다. (멥등성)
    1. 서비스를 띄울 네트워크 생성
    2. 필요한 볼륨 생성(혹은 이미 존재하는 볼륨과 연결)
    3. 필요한 이미지 풀(pull)
    4. 필요한 이미지 빌드(build)
    5. 서비스 실행 (depends\_on 옵션 사용시 서비스 의존성 순서대로 실행)
- `--build`
  - 이미 빌드가 되었더라도 강제로 빌드를 진행합니다.
- `-d`
  - 백그라운드로 실행합니다.
- `--force-recreate`
  - `docker-compose.yml` 파일의 변경점이 없더라도 강제로 컨테이너를 재생성  
합니다. 다시 말해서 컨테이너가 종료되었다가 다시 생성됩니다.
- `docker-compose down`

- 서비스를 멈추고 삭제합니다. 컨테이너와 네트워크를 삭제합니다.
- `--volume`
  - 선언된 도커 볼륨도 삭제합니다.
- `docker-compose stop`, `docker-compose start`
  - 서비스를 멈추거나, 멈춰 있는 서비스를 시작합니다.
- `docker-compose ps`
  - 현재 환경에서 실행 중인 각 서비스의 상태를 표시합니다.
- `docker-compose logs`

컨테이너 로그를 확인합니다.

- `-f`
  - `tail -f`와 유사하게 컨테이너 로그를 실시간으로 확인합니다. (follow)
- `docker-compose exec`
  - 실행 중인 컨테이너에 해당 명령어를 실행합니다.

```
docker-compose exec django ./manage.py makemigrations
docker-compose exec db psql postgres postgres
```

- `docker-compose run`
  - 특정 명령어를 일회성으로 실행하지만 컨테이너를 batch성 작업으로 사용하는 경우에 해당합니다. 이미 기동하고 있는 컨테이너에 명령어를 실행하고자 하면 `docker-compose exec`을 사용하는 반면에 `docker-compose run`을 사용할 경우 컨테이너를 기동시키고 특정 명령어를 실행이 완료된 후에 컨테이너를 종료합니다.

```
docker-compose exec web echo "hello world" # 이미 실행된 컨테이너에 명령어 실행
docker-compose run web echo "hello world" # web 컨테이너를 새로 생성하고 실행
```



**Docker Compose 파일 실전 예제를 소개합니다.**

▼ ► 실전에서 쓰이는 예제

## ▼ ► fastapi app

- Dockerfile

```
FROM python:3.10

RUN pip install pipenv

WORKDIR /app

ADD . /app/

RUN pipenv --python 3.10
RUN pipenv run pip install poetry
RUN pipenv sync
RUN pipenv run pip install certifi

ARG STAGE

RUN sh -c 'echo "STAGE=$STAGE" > .env'
RUN sh -c 'echo "PYTHONPATH=." >> .env'
# ENV PYTHONPATH=/app
RUN chmod +x ./scripts/run.sh
RUN chmod +x ./scripts/run-worker.sh

# ENV PYTHONPATH=/app
CMD ["./scripts/run.sh"]
```

- docker-compose.yml

```
version: "3"
services:
  channel-api:
    image: xxxx.dkr.ecr.ap-northeast-2.amazonaws.com
    build:
      context: .
      dockerfile: dockerfile
    args:
```

```

        STAGE: ${STAGE:-develop}
ports:
  - "8000:8000"
environment:
  - NEW_RELIC_CONFIG_FILE=newrelic.ini
  - NEW_RELIC_ENVIRONMENT=ecs-${STAGE:-develop}
logging:
  driver: awslogs
  options:
    awslogs-stream-prefix: channel
    awslogs-group: /ecs/reaction/${AWS_ENV_STAGE}
    awslogs-region: ap-northeast-2

reverseproxy:
  image: xxxxx.dkr.ecr.ap-northeast-2.amazonaws.com
ports:
  - "80:80"
  - "81:81"
logging:
  driver: awslogs
  options:
    awslogs-stream-prefix: reverseproxy
    awslogs-group: /ecs/reaction/${AWS_ENV_STAGE}
    awslogs-region: ap-northeast-2

```

#### ▼ 실습: 개발환경에서 Docker Compose 사용하기

- <https://github.com/nbcdocker/spring-boot-sample>
- 해당 코드를 clone

```

cd ~
git clone https://github.com/nbcdocker/spring-boot-sample

```

- docker-compose 로 실행

```

cd ~/spring-boot-sample
docker-compose up -d

```

```
docker-compose logs -f
```

- mysql, redis 를 Docker Compose로 실행하여 코드에서 사용
  - redis: 26379
  - mysql: 23306
- redis 연결 테스트
  - <http://localhost:8080/test-cache> 에 redis 를 사용한 5초 단위 캐시 적용 확인

### 03. Docker 모니터링&로깅

✓ 내가 실행하는 앱이 어떤 상태인지 Docker 모니터링을 통해 알아보고, 실행 과정을 기록한 로깅도 살펴봅니다.

#### ▼ 모니터링이란?

- 도커 모니터링이란?
  - 도커에서 모니터링은 컨테이너가 어떻게 돌아가고 있는지 지켜보는 거예요. 컨테이너의 성능, 사용 중인 자원(예: CPU나 메모리), 네트워크 사용량 같은 것들을 확인할 수 있죠.
- 왜 중요한가요?
  - 이런 정보를 알면 컨테이너가 잘 돌아가고 있는지, 어떤 문제가 있는지 파악할 수 있어요. 예를 들어, 어떤 컨테이너가 너무 많은 메모리를 쓰고 있다면, 그걸 조정해야 할 필요가 있겠죠.
- 어떻게 하나요?
  - 도커에는 모니터링을 위한 몇 가지 기본 도구가 있어요. 예를 들어, `docker stats` 명령어를 쓰면 실행 중인 컨테이너의 자원 사용량을 실시간으로 볼 수 있죠.
  - 또한, 외부 모니터링 도구를 사용해서 더 자세한 정보를 얻거나, 여러 컨테이너의 데이터를 한눈에 볼 수도 있어요.
- 그래서 뭐가 좋나요?

- 이런 모니터링 정보를 통해 문제를 빨리 발견하고 해결할 수 있어요. 그리고 컨테이너를 효율적으로 관리하고 최적화할 수 있죠.

## Container 리소스 모니터링

✓ 내가 실행하는 앱이 어떤 상태인지 Docker 모니터링을 통해 알아봅니다.

### ▼ docker stats: 컨테이너 모니터링의 시작점

#### • Docker Stats란 무엇인가요?

- 'Docker stats'는 Docker에서 제공하는 간단하고 실용적인 모니터링 도구예요. 이 명령어를 사용하면 현재 실행 중인 Docker 컨테이너들이 얼마나 많은 자원(CPU, 메모리 등)을 사용하고 있는지 실시간으로 볼 수 있어요.

#### • 왜 중요한가요?

- 컨테이너가 시스템 자원을 얼마나 사용하고 있는지 알면, 성능 문제를 빨리 찾아내고 해결할 수 있어요. 예를 들어, 메모리 사용량이 너무 높으면 시스템에 부하가 걸릴 수 있으니까요.

#### • 어떻게 사용하나요?

- 매우 간단해요! 터미널에서 `docker stats` 라고 입력하기만 하면 돼요. 그러면 현재 실행 중인 모든 컨테이너의 상태를 한눈에 볼 수 있죠.
- 이 명령어는 CPU 사용률, 메모리 사용량, 네트워크 I/O, 디스크 I/O 등 여러 중요한 정보를 보여줘요.

#### • 실제 예시

- 예를 들어보면, 이렇게 사용할 수 있어요: 이 명령어를 실행하면, 실행 중인 모든 컨테이너의 상태가 표시돼요. 각 컨테이너의 이름, ID, CPU 사용량, 메모리 사용량 등이 실시간으로 업데이트되면서 나타나죠.

```
docker stats
```

#### • 추가 팁

- 특정 컨테이너의 상태만 보고 싶다면, 컨테이너의 이름이나 ID를 명령어 뒤에 추가하면 돼요.

```
docker stats [컨테이너 이름 또는 ID]
```

- 이렇게 하면 해당 컨테이너의 자원 사용 상태만 볼 수 있어요.

#### ▼ htop: 시스템 모니터링의 필수 도구

- **htop이란 무엇인가요?**

- 'htop'은 리눅스 시스템을 모니터링하는데 사용되는 강력한 도구예요. 일반적인 시스템 모니터와 비슷하지만, 사용하기 편하고 여러 가지 유용한 기능을 제공해요.

- **왜 htop을 사용하나요?**

- 컴퓨터의 CPU, 메모리 사용량 같은 중요한 정보를 실시간으로 볼 수 있어요. 이 정보를 보면 컴퓨터가 어떻게 돌아가고 있는지, 어떤 프로세스가 많은 자원을 사용하고 있는지 알 수 있죠.

- **htop의 주요 기능은 무엇인가요?**

- **실시간 모니터링:** CPU, 메모리, 스왑 사용량을 실시간으로 볼 수 있어요.
- **프로세스 관리:** 실행 중인 프로세스를 쉽게 확인하고, 필요하면 종료시킬 수도 있어요.
- **사용자 친화적 인터페이스:** 색상이 있는 그래픽과 간단한 조작으로 정보를 쉽게 읽을 수 있어요.

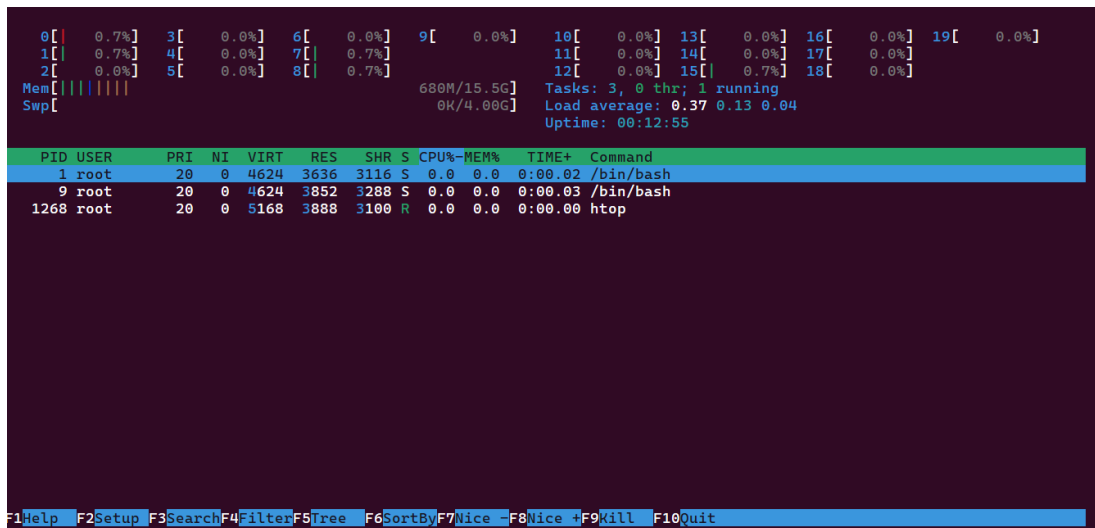
- **htop 사용 방법은?**

```
docker run --name test-tools -ti -d ubuntu:22.04

docker exec -ti test-tools /bin/bash
apt update; apt upgrade -y; apt install htop -y;
htop
exit
```

- 키보드 화살표를 사용해서 프로세스 목록을 넘겨볼 수 있고, **F9**를 눌러 프로세스를 종료할 수도 있어요.

- **사용 예**



### ▼ df

- disk free의 약자로, 리눅스 시스템 전체의 디스크 사용량 확인 가능해요
- ubuntu 22.04에서 실행해보기

```
docker exec -ti test-tools /bin/bash
df -h
exit
```

### ▼ du

- 디렉토리 별로 사용 공간을 나타내줘요
- ubuntu 22.04에서 실행해보기

```
docker exec -ti test-tools /bin/bash
du -sh # 현재 디렉토리의 총 디스크 사용량을 GB 단위로 보여줌
du -h --max-depth=1 # 현재 디렉토리 한 단계 아래 디렉토리 까지만
exit
```

## Container 로깅

✓ 내가 만든 앱의 실행 과정을 기록한 로깅도 살펴봅니다.

### ▼ Container 로깅



- Docker는 모든 컨테이너 로그의 표준 출력( `stdout` ) 또는 표준 에러( `stderr` )를 캡처하여 `json-file` 로깅 드라이버를 사용하여 json 형식으로 파일에 기록해요

#### ▼ 로그 파일의 위치

- Ubuntu에서는 `/var/lib/docker/containers/[컨테이너ID]/[컨테이너ID]-json.log`에 로그가 기록돼요.

#### ▼ 예제

- ▶ 특정 컨테이너의 로그 보기

```
docker run --name logs-test --rm -d ubuntu:22.04 /bin/bash

# logs-test 컨테이너의 로그를 전체 출력하기
docker logs logs-test

# logs-test 컨테이너의 로그를 tailing하기
docker logs -f logs-test

# 마지막 10줄부터 로그를 계속 보기
docker logs -f --tail 10 logs-test
```

- ▶ 로그 파일 설정 확인하기

```
docker inspect logs-test --format "{{.LogPath}}"
```

#### ▼ 로그 로테이션 설정

- 실제로 운영하다 보면 로그 파일의 크기가 계속 커질 수 있기 때문에 로그 파일의 최대 크기와 최대 파일 개수를 지정함
- ▶ Container 별 로깅 드라이버 구성

```
docker run -d \
--log-driver json-file \
--log-opt max-size=10m \
--log-opt max-file=10 \
--name nginxtest \
--restart always \
```

```
-p 80:80 \  
-p 443:443 \  
nginx:latest
```

▼ 실습: nginx 를 Container로 실행시키고 log file 크기와 개수를 제한 확인

- Container 의 log file 크기와 개수를 제한하기

```
docker run -d \  
--log-driver json-file \  
--log-opt max-size=1m \  
--log-opt max-file=5 \  
--name nginxtest \  
--restart always \  
-p 80:80 \  
-p 443:443 \  
nginx:latest
```

- 로그 보기

```
docker logs -f nginxtest
```

- 로그를 임의로 많이 생성하기

```
sudo apt update && sudo apt install -y wrk  
  
wrk -t10 -c100 -d30s http://localhost:80/
```

- docker compose 파일에서 설정
  - app 컨테이너는 10mb 이하의 로그파일을 최대 10개까지 생성합니다.

```
services:  
  app:  
    ...  
    logging:  
      driver: 'json-file'  
      options:
```

```
max-size: '10m'  
max-file: '10'
```

- 사용한 컨테이너 정리


```
docker stop nginxtest  
docker container rm nginxtest
```

## 3주차 끝

 [전체 강의자료 보기](#)

[다음 주차](#) 

 [Docker+CI/CD](#)

 [Docker+CI/CD - 4주차](#)

---

Copyright © TeamSparta All rights reserved.