겨울방학 스터디
1회차

College of Art & Technology
Chung-Ang University

Complex Intelligent Systems Laboratory

https://cislab.cau.ac.kr

Apache 2.0 freedom

Docker Hardened Images are fully open source and free to use, share, and build on, with no licensing surprises and complete transparency.

Minimal and distroless images

Ultra-minimal distroless Debian and Alpine images that remove everything you don't need, shrinking footprint and attack surface by up to 97%.

Up to 95% CVE reduction

Eliminate vulnerabilities before they reach production with continuously rebuilt images from Docker's hardened pipeline and verified SBOMs.

Extended Lifecycle Support for long-term protection

Add-on protection after upstream support ends, with multi-year CVE patches, updated SBOMs, and verifiable provenance that eliminate long windows of unpatched risk.

Start secure, customize easily

Add your own tools, packages, certificates, or settings while inheriting Docker's hardened pipeline, signed builds, and secure defaults.

1000+ images and applications

Continuously growing catalog of FIPS and STIG ready languages, frameworks, databases, application images and Helm charts, all signed and verified with SLA-backed security.

Apache 2.0 freedom

Docker Hardened Images are fully open source and free to use, share, and build on, with no licensing surprises and complete transparency.

완전한 오픈소스

Minimal and distroless images

Ultra-minimal distroless Debian and Alpine images that remove everything you don't need, shrinking footprint and attack surface by up to 97%.

경량화

Up to 95% CVE reduction

Eliminate vulnerabilities before they reach production with continuously rebuilt images from Docker's hardened pipeline and verified SBOMs.

보안 취약점 감소

Extended Lifecycle Support for long-term protection

Add-on protection after upstream support ends, with multi-year CVE patches, updated SBOMs, and verifiable provenance that eliminate long windows of unpatched risk.

장기적인 보안

Start secure, customize easily

Add your own tools, packages, certificates, or settings while inheriting Docker's hardened pipeline, signed builds, and secure defaults.

쉬운 커스터마이징

1000+ images and applications

Continuously growing catalog of FIPS and STIG ready languages, frameworks, databases, application images and Helm charts, all signed and verified with SLA-backed security.

1000개 이상의 공식 사례

"내 컴퓨터에선 되는데요"

개발자 A

"제 맥북에서는 잘 돌아가요"

Python 3.11, Node 18

개발자 B

"저는 에러나는데요..."

Python 3.8, Node 16

서버

"배포하니까 안 돌아가요"

Python 3.9, Node 14

모두가 겪는 환경 불일치 문제

도커는 왜 써야하는가

프로젝트 A

Python 3.8
Django 3.2
PostgreSQL 12

프로젝트 B

Python 3.11
FastAPI 0.100
PostgreSQL 15

발생하는 문제들

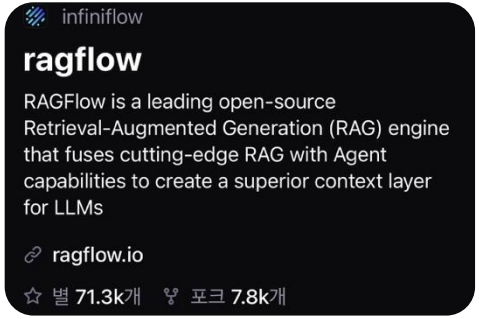
- 같은 PC에서 두 프로젝트 동시 실행 불가
- 버전 충돌로 인한 예상치 못한 버그
- 팀원마다 다른 개발 환경
- 로컬 ≠ 서버 환경 차이

"Build Once, Run Anywhere"

동일한 컨테이너가 어디서든 동일하게 동작

도커는 왜 써야하는가

많은 오픈소스 프로젝트들이 Docker로 설치를 지원




```
docker run infiniflow/ragflow
```

```
docker run open-webui/open-webui
```

```
docker run abi/screenshot-to-code
```

도커는 왜 써야하는가

환경 일관성

개발, 테스트, 운영 환경이 동일하여 "내 컴에선 되는데" 문제 해결

빠른 온보딩

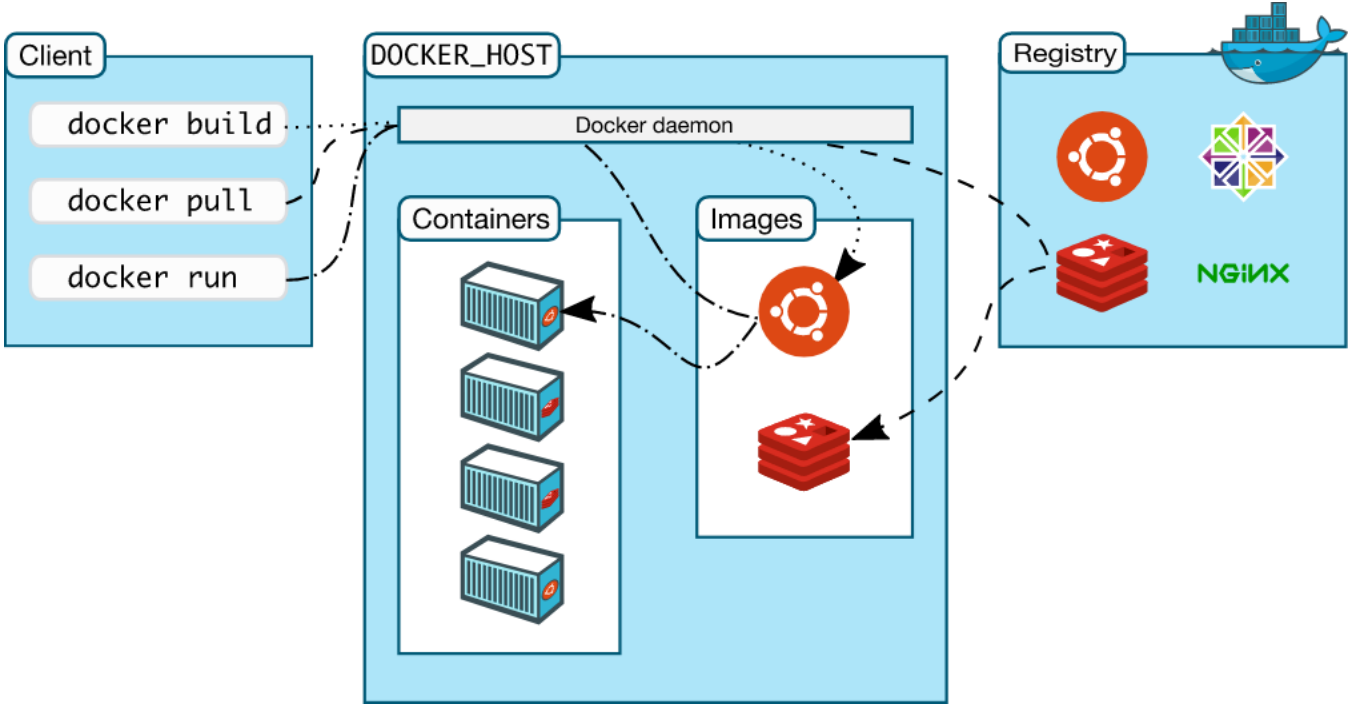
새 팀원도 명령어 한 줄로 개발 환경 구성 완료

쉬운 배포

이미지 빌드 후 어느 서버에서든 동일하게 실행 가능

롤백 용이

이전 버전 이미지로 즉시 복구 가능, 안정적인 운영


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

이미지 (Image)

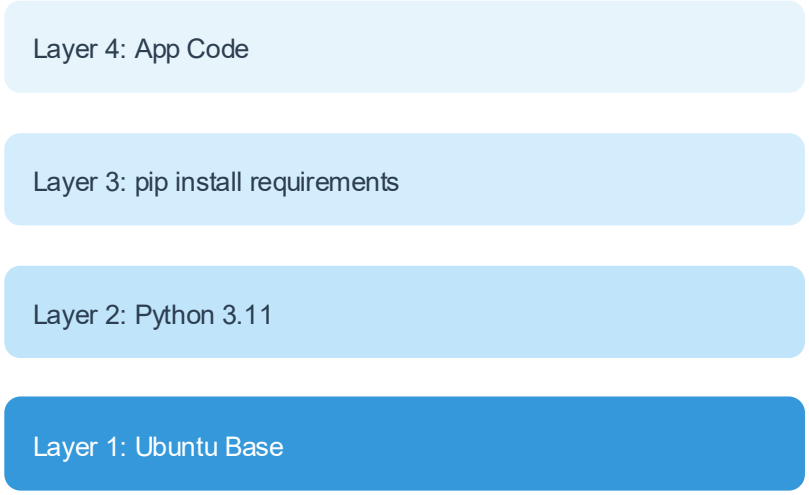
정의

컨테이너를 만들기 위한 읽기 전용 템플릿

특징

- **읽기 전용** - 한번 생성되면 변경 불가
- **레이어 구조** - 여러 층으로 구성되어 효율적 저장
- **재사용 가능** - 같은 이미지로 여러 컨테이너 생성

레이어 구조 예시



각 레이어는 캐시되어 빌드 속도 향상

컨테이너 (Container)

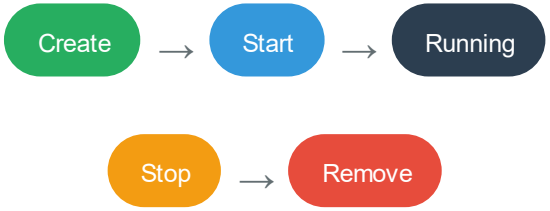
정의

이미지의 실행 인스턴스 (실제 동작하는 프로세스)

특징

- **읽기/쓰기 가능** - 실행 중 변경 가능
- **격리된 환경** - 독립적인 파일 시스템과 네트워크
- **휘발성** - 삭제하면 변경사항 사라짐

컨테이너 생명주기



💡 하나의 이미지로 여러 컨테이너 생성 가능

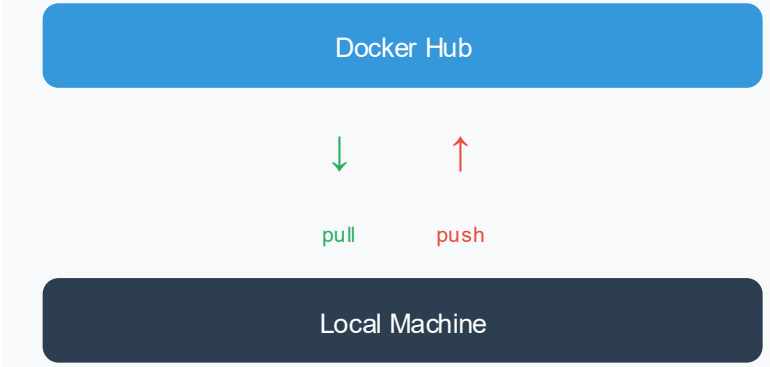
레지스트리 (Registry)

정의

Docker 이미지를 저장하고 배포하는 저장소

Docker Hub 공식 이미지 예시

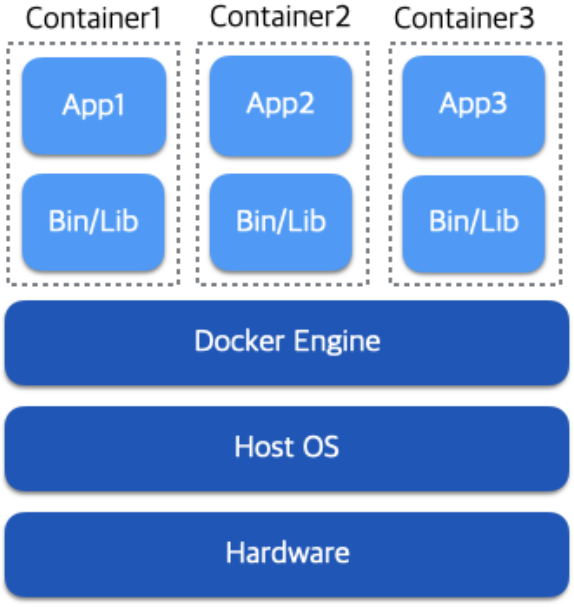
nginx　postgres　python　node　redis



컨테이너(Container)

핵심 정의

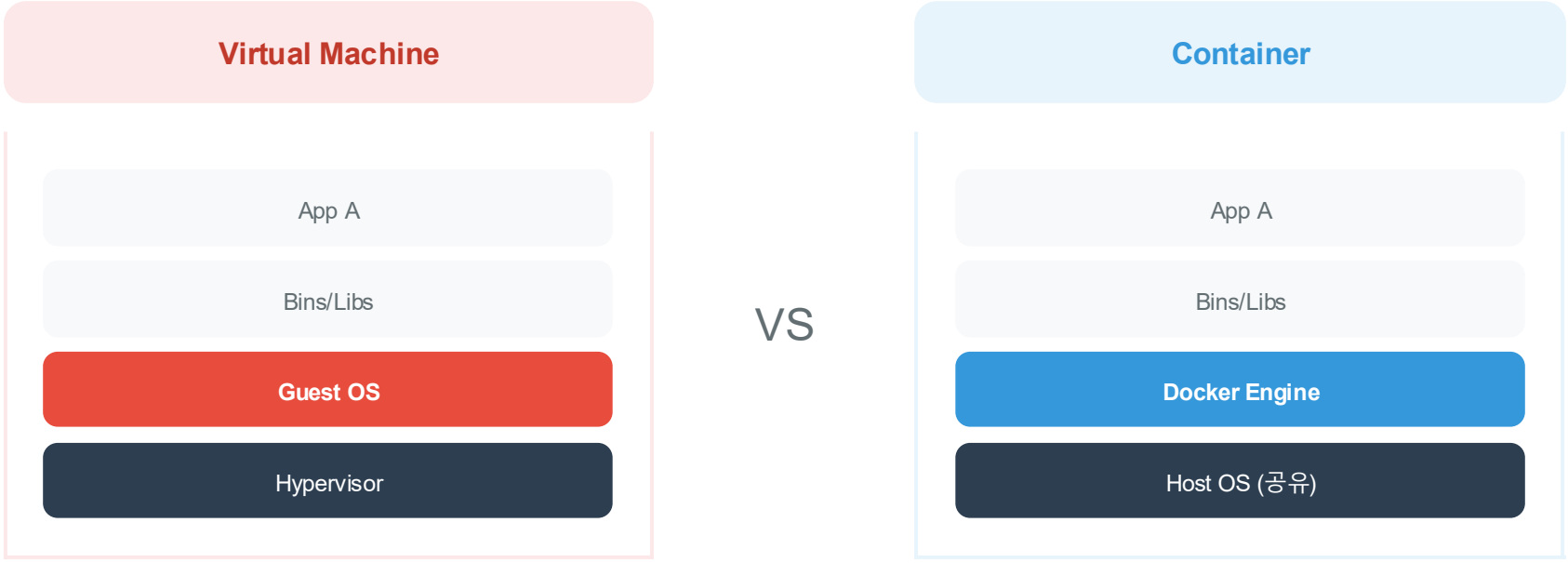
애플리케이션과 그 실행에 필요한 **모든 의존성**을
하나의 패키지로 묶어 **격리된 환경에서 실행**하는 기술



특징

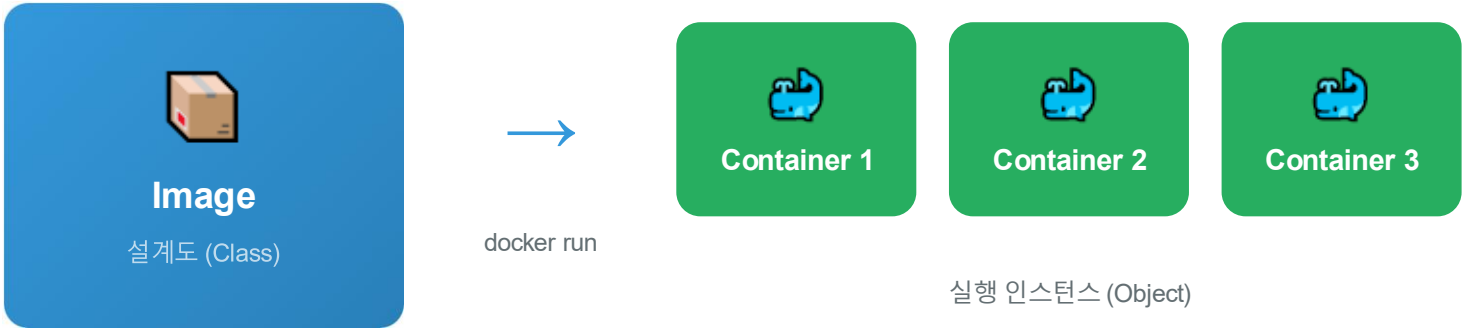
- 호스트 OS 커널을 공유 (가벼움)
- 프로세스 수준의 격리
- 빠른 시작/종료 (초 단위)

VM vs Container



무겁고 느림 (GB 단위, 분 단위 부팅)

가볍고 빠름 (MB 단위, 초 단위 시작)



프로그래밍 비유: **Image**는 Class, **Container**는 Class로 생성한 Instance

정의

도커 이미지를 만들기 위한 설정 파일

특징

- 텍스트 파일 (확장자 없음)
- 순차적 명령어 실행
- 버전 관리 가능 (Git)

Dockerfile

↓

docker build

↓

Docker Image

Python FastAPI 프로젝트용 Dockerfile 예시

```
# 베이스 이미지 지정
FROM python:3.11-slim

# 작업 디렉토리 설정
WORKDIR /app

# 의존성 파일 복사
COPY requirements.txt .

# 의존성 설치
RUN pip install -r requirements.txt

# 소스코드 복사
COPY . .

# 포트 노출
EXPOSE 8000

# 실행 명령
CMD ["uvicorn", "main:app", "--host", "0.0.0.0"]
```

핵심 명령어

FROM
베이스 이미지

WORKDIR
작업 디렉토리

COPY
파일 복사

RUN
빌드 시 실행

CMD
컨테이너 시작 시

FROM - 베이스 이미지

모든 Dockerfile은 FROM으로 시작

```
FROM python:3.11-slim
```

자주 사용하는 베이스 이미지

Python
`python:3.11-slim`

Node.js
`node:20-alpine`

Ubuntu
`ubuntu:22.04`

Alpine
`alpine:3.18`

💡 **slim**과 **alpine** 태그는 경량화된 이미지로 빌드 속도와 크기에 유리

WORKDIR / COPY

WORKDIR

```
WORKDIR /app
```

작업 디렉토리 설정. 이후 명령어들은 이 경로 기준으로 실행

COPY

```
COPY . .
```

호스트 파일을 컨테이너로 복사 (소스 대상)

COPY 패턴 예시

```
COPY requirements.txt .
```

특정 파일만 복사

```
COPY src/ /app/src/
```

디렉토리 복사

```
COPY . .
```

현재 디렉토리 전체 복사

.dockerignore로 제외할 파일 지정 가능

RUN vs CMD

RUN

```
RUN pip install -r requirements.txt
```

실행 시점

이미지 빌드 시 실행

용도

- 패키지 설치
- 파일 생성/수정
- 설정 작업

VS

CMD

```
CMD ["uvicorn", "main:app"]
```

실행 시점

컨테이너 시작 시 실행

용도

- 서버 실행
- 앱 시작
- 기본 명령어

이미지 빌드: docker build

```
docker build -t my-app:v1 .
```

-t my-app:v1
이미지 이름과 태그 지정

. (마침표)
빌드 컨텍스트 (Dockerfile 위치)

빌드 과정

1. Dockerfile 읽기 → 2. 각 명령어 순차 실행 → 3. 레이어 생성 → 4. 이미지 생성

💡 레이어는 캐시되어 변경되지 않은 부분은 재사용 빌드 속도 향상

이미지 관련 명령어

이미지 다운로드

```
docker pull nginx
```

Docker Hub에서 nginx 이미지를 로컬로 다운로드

이미지 목록 확인

```
docker images
```

로컬에 저장된 모든 이미지 목록 표시

이미지 삭제

```
docker rmi nginx
```

지정한 이미지를 로컬에서 삭제

컨테이너 실행: docker run

```
docker run -d -p 8080:80 --name my-nginx nginx
```

-d
백그라운드 실행 (detached mode)

-p 8080:80
포트 매핑 (호스트:컨테이너)

--name
컨테이너 이름 지정

nginx
사용할 이미지 이름

자주 쓰는 옵션들

-e : 환경변수 설정

-v : 볼륨 마운트

--rm : 종료 시 자동 삭제

컨테이너 상태 확인: docker ps

실행 중인 컨테이너 확인

```
docker ps
```

출력 결과 해석

```
CONTAINER ID   IMAGE   COMMAND                  STATUS       PORTS                  NAMES
a1b2c3d4e5f6   nginx   "/docker-entrypoint..."  Up 2 minutes   0.0.0.0:8080->80/tcp   my-nginx
```

CONTAINER ID: 고유 식별자

STATUS: 컨테이너 상태

PORTS: 포트 매핑 정보

NAMES: 컨테이너 이름

로그 확인: docker logs

기본 로그 확인

```
docker logs my-nginx
```

컨테이너의 stdout/stderr 출력 확인

실시간 로그 확인 (follow)

```
docker logs -f a1b2c3d4e5f6
```

실시간으로 로그 스트리밍 (Ctrl+C로 종료)

컨테이너 접속: docker exec

컨테이너 내부 쉘 접속

```
docker exec -it my-nginx /bin/bash
```

-i (interactive)
표준 입력 유지

-t (tty)
가상 터미널 할당

/bin/bash
실행할 명령어

활용 예시

```
# 컨테이너 내부에서 파일 확인
ls -la /etc/nginx/
# 설정 파일 확인
cat /etc/nginx/nginx.conf
```

컨테이너 중지 및 삭제

컨테이너 중지

```
docker stop my-nginx
```

Graceful shutdown (SIGTERM)

강제 삭제 (실행 중이어도)

```
docker rm -f my-nginx
```

컨테이너 삭제

```
docker rm my-nginx
```

중지된 컨테이너만 삭제 가능

이미지 삭제

```
docker rmi my-nginx
```

💡 **Tip**: 컨테이너를 삭제해도 이미지는 남아있습니다.

명령어 모음

이미지

```
docker pull [이미지]
```

```
docker images
```

```
docker rmi [이미지]
```

```
docker build -t [태그] .
```

컨테이너 실행

```
docker run -d -p [H:C] [이미지]
```

```
docker ps / docker ps -a
```

```
docker stop [컨테이너]
```

```
docker rm [컨테이너]
```

디버깅

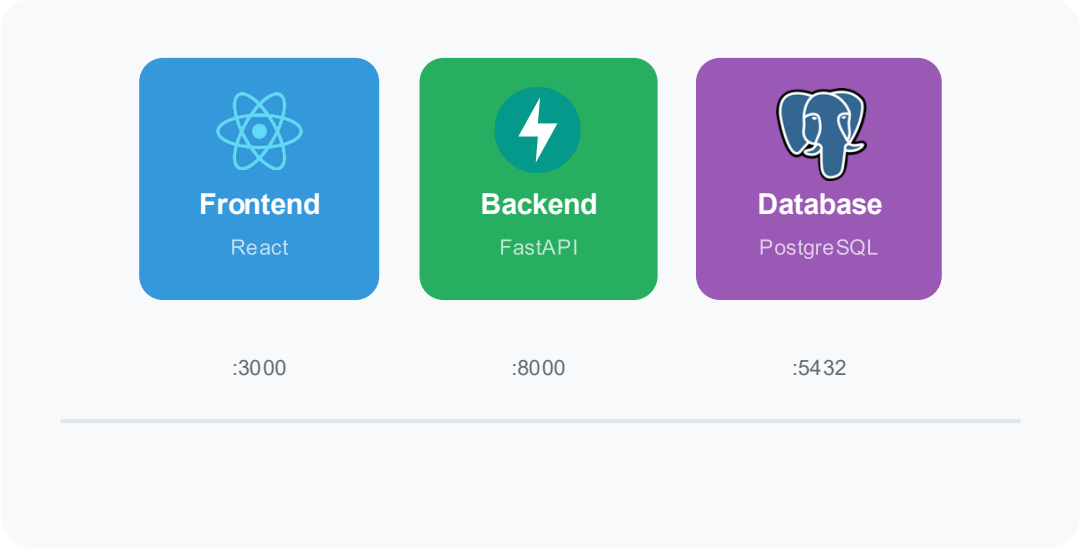
```
docker logs -f [컨테이너]
```

```
docker exec -it [컨테이너] bash
```

```
docker inspect [컨테이너]
```

```
docker stats
```

KMAP 프로젝트 구조



각 컨테이너 역할

- **Frontend**: 사용자 인터페이스
- **Backend**: API 서버
- **Database**: 데이터 저장

왜 Compose가 필요한가?

😰 Compose 없이 실행하면...

```
docker run -d --name db postgres
docker run -d --name api --link db my-api
docker run -d --name web --link api -p 80:80 my-web
```

- 매번 긴 명령어 입력
- 실행 순서 기억 필요
- 의존성 관리 어려움

😊 Compose 사용하면...

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

- 한 줄로 모든 서비스 시작
- 설정 파일로 버전 관리
- 의존성 자동 처리

Docker Compose

docker-compose.yml 예시

```
version: "3.8"
services:
frontend:
build: ./frontend
ports:
3000:3000"
backend:
build: ./backend
ports:
8000:8000"
depends_on:
db
db:
image: postgres:15
volumes:
db_data:/var/lib/postgresql/data
```

주요 키워드

services
컨테이너 정의

build / image
빌드 경로 또는 이미지

ports
포트 매핑

depends_on
의존성 (시작 순서)

volumes
데이터 영속화

KMAP Compose 파일 분석 (1) - Frontend

```
frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  ports:
    "3000:3000"
  volumes:
    ./frontend/src:/app/src
  depends_on:
    backend
```

설정 설명

build
./frontend 폴더의 Dockerfile로 빌드

ports: "3000:3000"
호스트 3000 컨테이너 3000

volumes (개발용)
소스 변경 시 실시간 반영 (Hot Reload)

depends_on
backend가 먼저 시작된 후 실행

KMAP Compose 파일 분석 (2) - Backend + DB

Backend

```
backend:
build: ./backend
ports:
"8000:8000"
environment:
DATABASE_URL=postgresql://
user:pass@db:5432/kmap
depends_on: [db]
```

Database

```
db:
image: postgres:15
environment:
POSTGRES_USER=user
POSTGRES_PASSWORD=pass
POSTGRES_DB=kmap
volumes:
db_data:/var/lib/postgresql
```

주요 포인트

environment
환경변수로 설정 주입

image (vs build)
공식 이미지 직접 사용

volumes (Named)
DB 데이터 영속화

db:5432 - 서비스명으로 통신

Docker Compose 명령어 모음

시작

```
docker compose up -d
```

모든 서비스 백그라운드 시작

종료

```
docker compose down
```

모든 서비스 중지 및 삭제

로그 확인

```
docker compose logs -f
```

모든 서비스 로그 실시간 확인

상태 확인

```
docker compose ps
```

서비스별 실행 상태 확인

재빌드

```
docker compose up -d --build
```

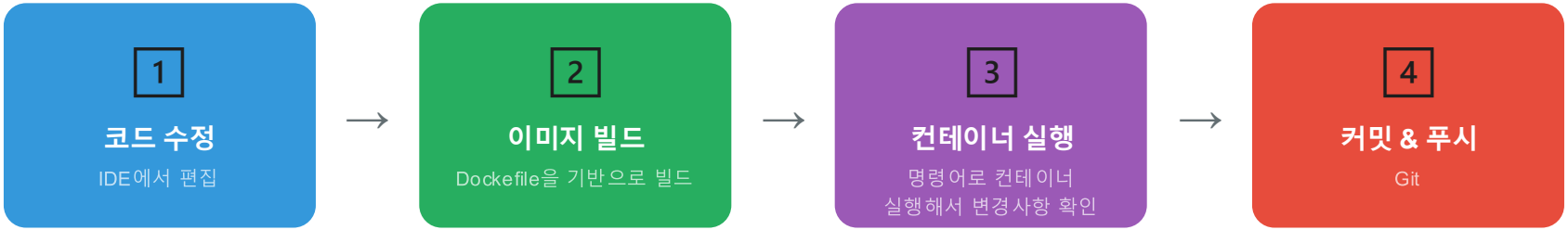
이미지 재빌드 후 시작

특정 서비스만

```
docker compose restart backend
```

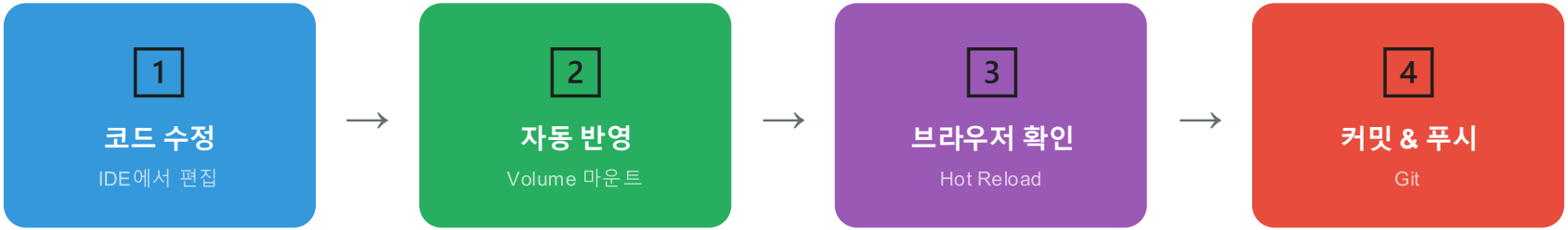
특정 서비스만 재시작

Docker Compose 개발 시나리오 (1)



💡 Docker 이미지는 수정 불가능하기 때문에 **무조건 빌드해서 새로 생성**해야 컨테이너에 코드 변경사항 적용됨

Docker Compose 개발 시나리오 (2)



Docker Compose + Volume 마운트로 **컨테이너 재시작 없이** 실시간 개발 가능

Docker Compose 명령어 실습

실제 서버에서 Docker 명령어를 실행해봅니다

1. 컨테이너 실행

```
docker compose up -d
```

2. 컨테이너 확인

```
docker ps
```

3. 로그 확인

```
docker logs -f backend
```

4. 컨테이너 접속

```
docker exec -it backend bash
```

5. 재시작

```
docker compose restart backend
```

실습 과제

Windows / Mac

Docker Desktop 설치

https://www.docker.com/products/docker-desktop

Linux (Ubuntu)

```
sudo apt update
sudo apt install docker.io docker-compose
```

설치 확인

```
# 버전 확인
docker --version

# 테스트 실행
docker run hello-world
```

✓ "Hello from Docker!" 메시지가 나오면 성공!

KMAP 프로젝트 로컬 실행하기

Step 1. 저장소 클론

```
git clone https://github.com/cxinsys/kmap.git
cd kmap
```

Step 2. Docker 실행

```
docker compose up -d
```

Step 3. 확인

```
브라우저에서 http://localhost:3000 접속
```

✅ **제출물**

웹페이지 접속 여부에 대한 스크린샷을 디스코드 **#스터디-공유**에 공유

Q&A

다음 회차 안내

📅 1/16 (목) - 팀 워크플로우 & Claude Code

📝 과제: Docker 블로그 글쓰기
(다음주 화요일까지)