



Docker & Docker Compose 튜아보 기

컨테이너 기술을 통한 애플리케이션 패키징 및 배포 플랫폼으로, 현대 소프트웨어 개발과 운영의 혁신을 이끌고 있습니다. Docker는 "내 컴퓨터에서는 잘 됐는데..." 라는 개발자들의 영원한 고민을 완전히 해결하여, 개발부터 운영까지 일관된 환경을 보장합니다.

이 발표에서는 Docker와 Docker Compose의 핵심 개념을 체계적으로 설명할 예정입니다. 컨테이너 기술의 원리를 이해하고, 실제 프로젝트에서 활용할 수 있는 실용적인 지식을 습득해보시면 좋을 거 같습니다.



전통적인 개발 환경의 문제점

환경 불일치 문제

개발자마다 다른 운영체제, 런타임 버전, 라이브러리 설정으로 인해 발생하는 예측 불가능한 오류들. Windows, macOS, Linux 환경 간 차이로 인한 배포 실패와 버그 발생이 일상적인 문제였습니다.

VM 방식의 한계

가상머신은 운영체제 전체를 설치해야 하므로 과도한 자원 낭비가 발생합니다. 실제 애플리케이션이 사용하는 메모리의 73%가 OS에 의해 낭비되며, 부팅과 관리에 막대한 오버헤드가 발생합니다.

복잡한 의존성 관리

다양한 라이브러리와 서비스 간의 복잡한 의존성으로 인한 설정의 어려움. 새로운 팀원이 개발 환경을 구축하는데 반나절이 걸리는 것은 일반적인 현상이었습니다.

이러한 문제들로 인해 개발 생산성이 크게 저하되고, 배포 과정에서 예상치 못한 오류가 빈번하게 발생했습니다. Docker는 바로 이런 근본적인 문제들을 해결하기 위해 등장했습니다.

Container vs Virtual Machine

1-2GB

VM 메모리 사용량
각각 독립된 운영체제 필요

10-...

컨테이너 메모리 사용량
OS 커널 공유로 효율적

60-100..

VM 부팅 시간
운영체제 전체 로딩 필요

2-5초

컨테이너 시작 시간
애플리케이션만 실행

가상머신 방식

- 하이퍼바이저 위에 완전한 OS 설치
- 각 VM마다 독립된 커널과 시스템 리소스
- 강력한 격리성이지만 높은 오버헤드
- 하드웨어 가상화 기반

컨테이너 방식

- 호스트 OS 커널을 공유하여 사용
- 애플리케이션과 의존성만 패키징
- 효율적인 자원 사용과 빠른 실행
- 운영체제 수준 가상화

실제 벤치마크 결과, 컨테이너는 VM 대비 메모리 사용량을 60% 절약하고 성능을 95% 향상시킵니다. 이는 동일한 하드웨어에서 더 많은 애플리케이션을 실행할 수 있음을 의미합니다.



Docker 핵심 원리 - 격리 기술

01

네임스페이스 (Namespaces)

프로세스를 독립적인 환경으로 격리하는 Linux 커널 기능입니다. PID 네임스페이스로 프로세스 ID를 격리하고, Network 네임스페이스로 네트워크 인터페이스를 분리하며, Mount 네임스페이스로 파일시스템을 독립화합니다. User 네임스페이스는 사용자 권한을 격리하여 보안을 강화합니다.

02

Control Groups (Cgroups)

컨테이너가 사용할 수 있는 시스템 리소스의 양을 제한하고 모니터링합니다. CPU 사용률, 메모리 용량, 디스크 I/O, 네트워크 대역폭을 세밀하게 제어할 수 있어 한 컨테이너가 전체 시스템 자원을 독점하는 것을 방지합니다.

03

Copy-on-Write 파일시스템

이미지 레이어를 공유하여 저장공간을 효율적으로 사용합니다. 여러 컨테이너가 동일한 베이스 이미지를 공유하고, 변경사항이 있을 때만 새로운 레이어를 생성합니다. 이로 인해 디스크 사용량이 크게 절약되고 컨테이너 생성 속도가 향상됩니다.

❏ **팁:** 이러한 격리 기술들은 Linux 커널 기능을 활용하므로, Windows와 macOS에서는 경량화된 Linux VM 위에서 Docker가 실행됩니다. 하지만 사용자 입장에서는 동일한 경험을 제공합니다.

Docker 네트워크 - 서비스 간 통신

1

Bridge Network 생성

Docker가 자동으로 격리된 네트워크를 생성합니다. 컨테이너들은 이 가상 네트워크에 연결되어 서로 통신할 수 있습니다.

2

자동 DNS 해석

컨테이너 이름으로 직접 통신이 가능합니다. IP 주소를 기억할 필요 없이 컨테이너 이름을 호스트명으로 사용할 수 있습니다.

3

포트 매핑

외부 접근이 필요한 서비스만 선택적으로 호스트 포트에 매핑하여 보안을 강화합니다.

3단계 보안 계층

서버 방화벽

운영체제 수준의 첫 번째 보안 계층으로 외부 트래픽을 필터링합니다.

Docker 포트 매핑

명시적으로 매핑된 포트만 외부에서 접근 가능하도록 제어합니다.

네트워크 격리

컨테이너 간 격리된 가상 네트워크로 내부 통신을 보호합니다.

실무 활용 시나리오: 웹 애플리케이션, 데이터베이스, 캐시 서버가 같은 Docker 네트워크에 있다면, 모든 포트로 자유롭게 통신할 수 있습니다. 하지만 외부에서는 웹 서버의 80/443 포트만 접근 가능하도록 설정하여 데이터베이스를 보호할 수 있습니다.

Docker 볼륨 - 데이터 영속성



Named Volume

Docker가 관리하는 볼륨

- 운영환경에서 권장되는 방식
- 백업과 복원이 용이함
- Docker CLI로 관리 가능
- 컨테이너와 독립적으로 존재

데이터베이스 데이터, 로그 파일 등 중요한 데이터 저장에 최적화되어 있습니다.



Bind Mount

호스트 경로 직접 연결

- 개발환경에서 매우 유용
- 실시간 파일 동기화 지원
- 소스 코드 편집 시 즉시 반영
- 호스트 파일시스템과 완전 통합

개발 중인 소스 코드, 설정 파일 등을 컨테이너와 실시간으로 공유할 때 사용합니다.



tmpfs Mount

RAM 기반 임시 저장소

- 메모리에 직접 저장
- 극도로 빠른 I/O 성능
- 컨테이너 종료 시 자동 삭제
- 보안이 중요한 임시 데이터용

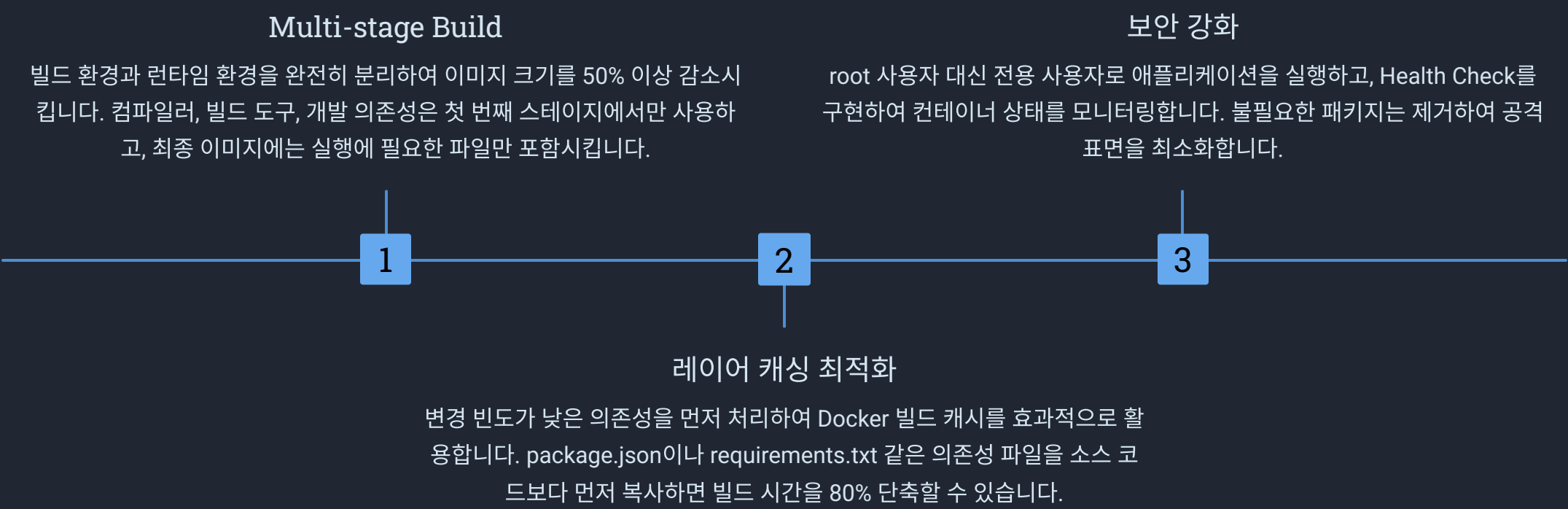
세션 데이터, 임시 캐시, 민감한 정보의 일시적 저장에 활용됩니다.

실무 권장사항

- 운영환경: Named Volume 사용으로 안정성 확보
- 개발환경: Bind Mount로 개발 효율성 향상
- 고성능 요구: tmpfs Mount로 I/O 병목 해결



Dockerfile - 이미지 최적화



Dockerfile 최적화 예제

```
# Multi-stage build 예제
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production

FROM node:18-alpine AS runtime
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nodejs -u 1001
WORKDIR /app
COPY --from=builder /app/node_modules ./node_modules
COPY . .
USER nodejs
EXPOSE 3000
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s \
  CMD node healthcheck.js
CMD ["node", "server.js"]
```

50%

이미지 크기 감소
Multi-stage build 적용 시

80%

빌드 시간 단축
레이어 캐싱 최적화 시

95%

보안 위험 감소
비 root 사용자 실행 시

Docker Compose - 통합 관리

기존 방식의 문제점

개별 컨테이너마다 복잡한 docker run 명령어를 수십 줄씩 실행해야 했습니다:

```
docker run -d --name web \
  --network mynet \
  -p 80:3000 \
  -e NODE_ENV=production \
  -v logs:/app/logs \
  myapp:latest

docker run -d --name db \
  --network mynet \
  -e POSTGRES_PASSWORD=secret \
  -v pgdata:/var/lib/postgresql/data \
  postgres:13
```


Docker Compose 해결책

YAML 파일 하나로 전체 애플리케이션 스택을 정의하고 관리합니다:

```
version: '3.8'
services:
  web:
    image: myapp:latest
    ports: ["80:3000"]
    environment:
      NODE_ENV: production
    volumes: ["logs:/app/logs"]
    depends_on: [db]


  db:
    image: postgres:13
    environment:
      POSTGRES_PASSWORD: secret
    volumes: ["pgdata:/var/lib/postgresql/data"]

volumes:
  logs:
  pgdata:
```




자동 네트워크 연결

모든 서비스가 자동으로 동일한 네트워크에 연결되며, 서비스명으로 서로를 찾을 수 있습니다. 복잡한 네트워크 설정이나 IP 주소 관리가 불필요합니다.



의존성 관리

depends_on 설정을 통해 서비스 시작 순서를 제어합니다. 데이터베이스가 완전히 준비된 후에 웹 애플리케이션이 시작되도록 보장할 수 있습니다.

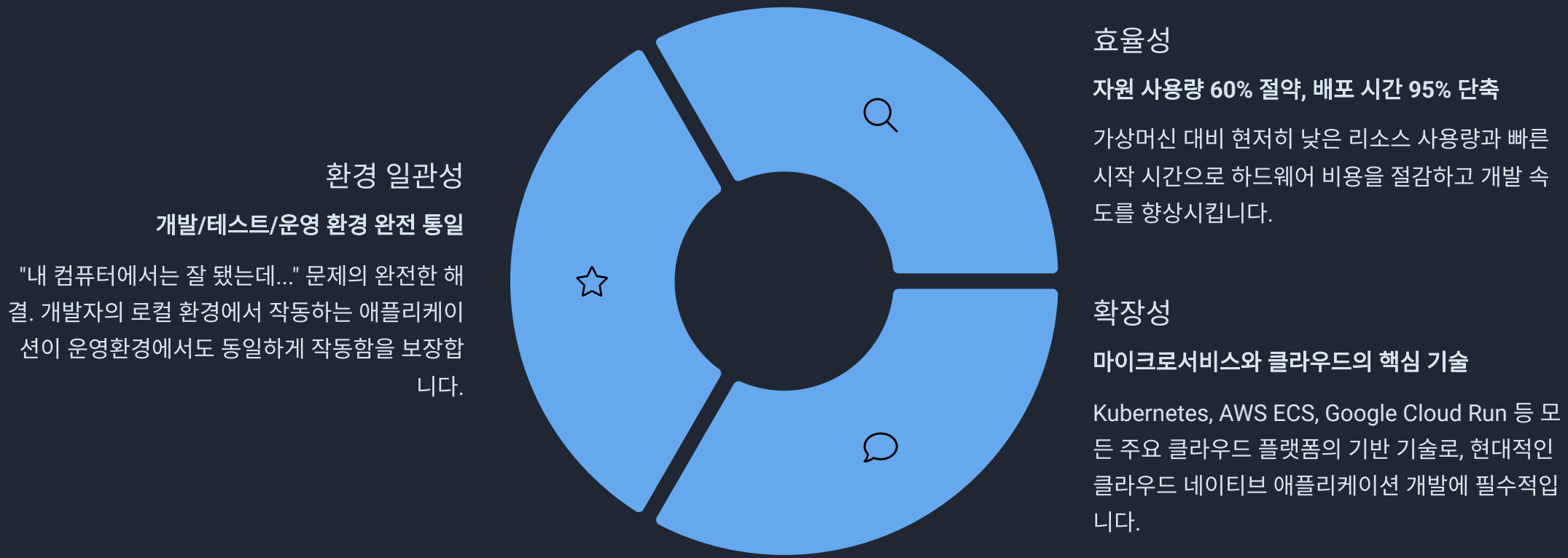


원클릭 배포

docker-compose up 명령어 하나로 전체 시스템을 구성하고 시작할 수 있습니다. 개발환경 설정부터 운영환경 배포까지 일관된 방식으로 관리됩니다.

실무 활용: 마이크로서비스 아키텍처에서 여러 개의 서비스와 데이터베이스, 캐시, 메시지 큐 등을 하나의 compose 파일로 관리하면 개발팀의 생산성이 크게 향상됩니다.

Docker의 핵심 가치



Docker 도입 후 기대 효과

- 개발 생산성 향상: 환경 설정 시간 단축으로 개발에 집중
- 운영 안정성: 예측 가능한 배포와 일관된 실행 환경
- 비용 효율성: 하드웨어 자원의 효율적 활용
- 기술 발전성: 클라우드 네이티브 기술 스택의 기반



핵심 정리 및 Q&A

지금까지 Docker의 주요 개념과 활용 방법에 대해 상세히 살펴보았습니다. 개발 환경의 일관성부터 효율적인 배포, 그리고 마이크로서비스 아키텍처의 기반이 되는 Docker의 핵심 가치를 이해하셨기를 바랍니다.



개발 환경 통일

어디서든 동일하게 작동하는 환경을 구축하여 "내 컴퓨터에서는 되는데" 문제를 해결합니다.



생산성 및 효율성

빠른 컨테이너 시작, 효율적인 자원 사용, 빌드 시간 단축으로 개발 및 운영 효율을 극대화합니다.



확장성과 안정성

컨테이너 기반으로 마이크로서비스와 클라우드 환경에서 유연한 확장과 안정적인 운영이 가능합니다.



간편한 관리

Dockerfile로 이미지 빌드를 표준화하고, Docker Compose로 복잡한 다중 컨테이너 애플리케이션을 쉽게 정의하고 관리합니다.

질의응답 (Q&A)

지금까지 설명드린 내용 중 궁금한 점이 있으시거나, Docker와 관련된 경험을 공유해주실 분이 있다면 자유롭게 말씀해주세요.

