6단원. **Docker** 실전 활용

> 대상: 전공자 · 정보처리기사 취득자 | 목적: 인프라 직무(네트워크·클라우드·시스템·보안) 기술 면접 대비 중요도: ★★★ 최빈출 · ★★ 자주 출제 · ★ 기본 개념

Q1. docker-compose란 무엇이며 어떤 상황에 사용하나요?

★★★

답안 **docker-compose**는 여러 개의 컨테이너로 구성된 애플리케이션을 하나의 **YAML** 파일로 정의하고, 단일 명령어로 전체를 함께 생성·실행·종료할 수 있게 해주는 도구입니다. 예를 들어 웹 서버, 애플리케이션 서버, 데이터베이스, 캐시 서버가 각각 별도의 컨테이너로 실행되는 애플리케이션에서, 각 서비스의 이미지, 환경변수, 포트, 볼륨, 의존 관계(`depends_on`)를 하나의 파일에 정의해두면 `docker compose up` 한 번으로 전체 스택을 손쉽게 실행할 수 있습니다. 이는 특히 로컬 개발 환경 구성이나 간단한 테스트 환경 배포에 유용하며, docker-compose는 기본적으로 컨테이너 간 통신을 위한 전용 네트워크를 자동으로 생성해 서비스 이름만으로 서로를 참조할 수 있게 해줍니다. 다만 다중 호스트에 걸친 대규모 오케스트레이션에는 적합하지 않아 그런 경우에는 쿠버네티스를 사용합니다.

관련 개념 YAML 정의, depends_on, 서비스 디스커버리, 단일 호스트 오케스트레이션

> **Q2.** 멀티스테이지 빌드(**Multi-stage Build**)의 개념과 장점을 설명하세요. ★★★

답안 멀티스테이지 빌드는 하나의 Dockerfile 안에 여러 개의 `FROM` 단계를 두어, 빌드 환경과 실행 환경을 분리하는 기법입니다. 예를 들어 첫 번째 스테이지에서는 컴파일러나 빌드 도구가 포함된 무거운 이미지로 소스코드를 빌드하고, 두 번째 스테이지에서는 실행에 필요한 최소한의 런타임 이미지에 첫 번째 스테이지에서 생성된 산출물(바이너리, 정적 파일 등)만 `COPY --from`으로 복사해옵니다. 이렇게 하면 최종 이미지에는 빌드 도구나 소스코드, 중간 산출물이 전혀 남지 않아 이미지 크기를 대폭 줄이고 공격 표면(불필요한 도구로 인한 보안 취약점)도 줄일 수 있습니다. Java, Go, Node.js처럼 빌드와 실행 환경이 다른 언어의 컨테이너화에서 특히 널리 사용되는 기법입니다.

관련 개념 COPY --from, 빌드 환경/런타임 환경 분리, 공격 표면 축소, 이미지 경량화

> **Q3.** 컨테이너의 **CPU/**메모리 리소스를 제한하는 방법과 필요성을 설명하세요. ★★★

답안 Docker는 리눅스 **cgroups**를 이용해 컨테이너별로 CPU와 메모리 사용량을 제한할 수 있습니다. `docker run` 실행 시 `--memory` 옵션으로 최대 메모리 사용량을, `--cpus` 옵션으로 사용할 수 있는 CPU 코어 수를 제한할 수 있으며, docker-compose에서도 `deploy.resources` 항목으로 동일하게 설정할 수 있습니다. 리소스 제한이 필요한 이유는 하나의 호스트에서 여러 컨테이너가 함께 실행될 때, 특정 컨테이너가 메모리 누수나 무한 루프로 인해 자원을 독점하면 다른 컨테이너나 호스트 자체에 영향을 줄 수 있기 때문입니다(Noisy Neighbor 문제). 메모리 제한을 초과하면 컨테이너 내 프로세스가 OOM Killer에 의해 강제 종료되므로, 애플리케이션의 실제 사용량을 모니터링한 뒤 적절한 여유를 두고 제한값을 설정하는 것이 중요합니다.

관련 개념 cgroups, OOM Killer, Noisy Neighbor, requests/limits(쿠버네티스 연계)

> **Q4.** 컨테이너 로그를 수집하고 모니터링하는 방법을 설명하세요. ★★

답안 Docker는 컨테이너의 표준 출력(stdout)과 표준 에러(stderr)를 로그로 자동 수집하며, `docker logs` 명령어로 확인할 수 있습니다. 기본 로깅 드라이버는 `json-file`이지만, 컨테이너가 많아지고 여러 호스트에 분산될 경우 각 호스트에 로그가 흩어져 있어 통합 분석이 어려워지므로, 실무에서는 로그를 중앙 집중화하는 것이 일반적입니다. 대표적으로 Fluentd나 Logstash 같은 로그 수집기를 통해 로그를 Elasticsearch에 적재하고 Kibana로 시각화하는 **ELK/EFK** 스택을 사용하거나, 클라우드 환경에서는 CloudWatch Logs 같은 관리형 로그 서비스로 전송합니다. 모니터링 측면에서는 컨테이너의 CPU/메모리 사용률을 확인할 수 있는 `docker stats` 외에, Prometheus와 Grafana를 이용해 메트릭을 수집하고 대시보드로 시각화하는 방식이 널리 사용됩니다.

관련 개념 로깅 드라이버, ELK/EFK 스택, Prometheus/Grafana, 중앙 집중 로깅

> **Q5. Docker** 컨테이너 레지스트리 운영 시 이미지 보안을 어떻게 관리하나요? ★★

답안 이미지 보안 관리를 위해서는 먼저 신뢰할 수 없는 출처의 베이스 이미지를 사용하지 않고, 공식 이미지나 검증된 이미지를 사용하는 것이 기본입니다. 이미지에 알려진 취약점(CVE)이 포함되어 있는지 CI/CD 파이프라인에 이미지 스캐닝 단계를 추가해 빌드나 배포 전에 자동으로 검사하며, 취약점이 발견된 이미지는 배포를 차단하도록 정책을 설정합니다. 또한 컨테이너 안에서 불필요하게 **root** 권한으로 프로세스를 실행하지 않도록 Dockerfile에 별도 사용자를 생성해 지정하고, 민감한 정보(API 키, 비밀번호)를 이미지에 하드코딩하지 않고 환경변수나 시크릿 관리 도구를 통해 런타임에 주입해야 합니다. 레지스트리 접근 권한도 최소 권한 원칙에 따라 통제하고, 이미지에 서명을 추가해 신뢰된 이미지만 배포되도록 하는 것도 보안 강화 방법입니다.

관련 개념 CVE 스캐닝, non-root 사용자, 시크릿 관리, 이미지 서명

> **Q6. docker-compose**에서 서비스 간 의존성과 네트워크 통신은 어떻게 이루어지나요? ★★

답안 docker-compose는 실행 시 기본적으로 프로젝트 전용의 사용자 정의 브리지 네트워크를 자동 생성하며, 같은 네트워크에 속한 서비스들은 IP 주소가 아닌 서비스 이름을 호스트명처럼 사용해 서로 통신할 수 있습니다(내장 DNS를 통한 서비스 디스커버리). 서비스 간 실행 순서를 제어하려면 `depends_on`을 사용해 특정 서비스가 먼저 시작되도록 지정할 수 있지만, 이는 컨테이너의 "시작"만 보장할 뿐 애플리케이션이 실제로 요청을 받을 준비가 되었는지는 보장하지 않기 때문에, 데이터베이스처럼 초기화 시간이 필요한 서비스는 `healthcheck`와 `condition: service_healthy`를 함께 사용해 실제 준비 상태까지 확인하는 것이 안전합니다. 이러한 메커니즘 덕분에 개발자는 IP 관리 없이 서비스 이름만으로 안정적인 멀티 컨테이너 애플리케이션을 구성할 수 있습니다.

관련 개념 서비스 디스커버리, 내장 DNS, healthcheck, depends_on 한계

> **Q7.** 컨테이너 헬스체크(**Healthcheck**)의 필요성과 설정 방법을 설명하세요. ★★

답안 컨테이너가 정상적으로 실행 중(**Running**)이라는 상태와, 애플리케이션이 실제로 정상 동작하며 요청을 처리할 수 있는 상태는 다를 수 있습니다. 예를 들어 프로세스는 살아있지만 내부 로직이 무한 대기 상태에 빠져 응답하지 못하는 경우가 있는데, 이를 감지하기 위해 Dockerfile이나 docker-compose에 `HEALTHCHECK` 명령어를 정의해 주기적으로 특정 명령(예: HTTP 엔드포인트 호출)을 실행하고 그 결과로 컨테이너의 건강 상태를 판단합니다. 헬스체크가 실패하면 컨테이너 상태가 `unhealthy`로 표시되며, 이를 오케스트레이션 도구나 로드밸런서와 연동하면 비정상 컨테이너를 트래픽 대상에서 자동으로 제외하거나 재시작할 수 있습니다. 이는 쿠버네티스의 liveness/readiness probe와 개념적으로 동일한 역할을 합니다.

관련 개념 HEALTHCHECK 명령어, unhealthy 상태, liveness/readiness probe, 자동 재시작

> **Q8.** 컨테이너 이미지 배포를 위한 **CI/CD** 파이프라인 구성을 설명하세요. ★★

답안 전형적인 컨테이너 CI/CD 파이프라인은 개발자가 소스코드를 저장소에 푸시하면 시작됩니다. **CI(**지속적 통합**)** 단계에서는 코드를 빌드하고 단위 테스트를 실행한 뒤, 문제가 없으면 Dockerfile을 기반으로 이미지를 빌드하고 취약점 스캔을 거쳐 고유한 태그(커밋 해시 등)를 붙여 컨테이너 레지스트리에 푸시합니다. **CD(**지속적 배포) 단계에서는 새로 푸시된 이미지를 실제 운영 환경(쿠버네티스 클러스터나 EC2 등)에 배포하는데, 이때 롤링 업데이트나 블루-그린 배포 전략을 사용해 무중단으로 새 버전을 반영합니다. 이러한 파이프라인은 GitHub Actions, Jenkins, GitLab CI 같은 도구로 자동화되며, 배포 후에는 헬스체크와 모니터링을 통해 정상 동작을 확인하고 문제 발생 시 이전 버전으로 신속히 롤백할 수 있는 체계를 갖추는 것이 중요합니다.

관련 개념 CI/CD 파이프라인, 롤링 업데이트, 블루-그린 배포, 롤백

> **Q9.** 도커 컨테이너의 데이터 영속성을 보장하면서 여러 컨테이너가 볼륨을 공유하는 방법을 설명하세요. ★

답안 여러 컨테이너가 동일한 데이터를 공유해야 하는 경우, Docker의 명명된 볼륨 **(Named Volume)**을 생성해 여러 컨테이너의 마운트 경로에 동일한 볼륨을 연결하면 됩니다. 예를 들어 로그를 생성하는 애플리케이션 컨테이너와 이를 수집하는 로그 수집 컨테이너가 같은 볼륨을 공유하도록 구성하면, 컨테이너 간 프로세스 격리는 유지하면서도 파일 시스템 레벨에서 데이터를 주고받을 수 있습니다. 다만 단일 호스트를 벗어나 여러 노드에 컨테이너가 분산된 환경(쿠버네티스 등)에서는 로컬 볼륨만으로는 데이터 공유가 어렵기 때문에, NFS 같은 네트워크 파일시스템이나 클라우드의 관리형 파일 스토리지를 이용한 분산 스토리지 솔루션이 필요합니다.

관련 개념 Named Volume, NFS, 분산 스토리지, PersistentVolume(쿠버네티스 연계)

> **Q10.** 컨테이너 환경에서 시크릿(비밀번호, **API** 키)을 안전하게 관리하는 방법을 설명하세요. ★★

답안 가장 흔한 실수는 비밀번호나 API 키를 Dockerfile에 하드코딩하거나 이미지 자체에 포함시키는 것으로, 이미지가 유출되면 시크릿도 함께 노출되므로 반드시 피해야 합니다. 대신 런타임에 환경변수로 주입하는 방법이 있지만, 환경변수는 `docker inspect`나 프로세스 정보를 통해 노출될 위험이 있어 완전히 안전하지는 않습니다. 더 안전한 방법은 Docker Swarm의 secrets 기능이나 쿠버네티스의 Secret 오브젝트처럼 전용 시크릿 관리 메커니즘을 이용해 파일 형태로 컨테이너 내부의 특정 경로에만 마운트하는 방식이며, 대규모 조직에서는 HashiCorp Vault나 AWS Secrets Manager 같은 외부 시크릿 관리 서비스와 연동해 시크릿을 중앙에서 관리하고 접근 이력을 감사하는 방식을 사용합니다.

관련 개념 Docker secrets, 쿠버네티스 Secret, HashiCorp Vault, Secrets Manager

---