WEEK_1

Microservice와 Spring Cloud 소개

> 2010년대부터 IT 업계에서는 '**Antifragile**' 개념과 함께 '**클라우드 네이티브(Cloud Native)**' 패러다임이 본격적으로 주목받기 시작했다. 이는 기존 모놀리식 아키텍처의 한계를 극복하고, 장애나 실패를 감지해 자동으로 복구하거나 개선할 수 있는 시스템 설계를 구축하기 위한 방향으로 이어졌다.

Antifragile : **스트레스를 통해 오히려 더 강해지는 특성**

Cloud Native Architecture

클라우드 환경의 이점을 극대화하기 위해 설계부터 배포, 운영까지 모든 과정을 클라우드 기반으로 구성하는 소프트웨어 아키텍처 패턴

핵심 개념은 다음과 같다.

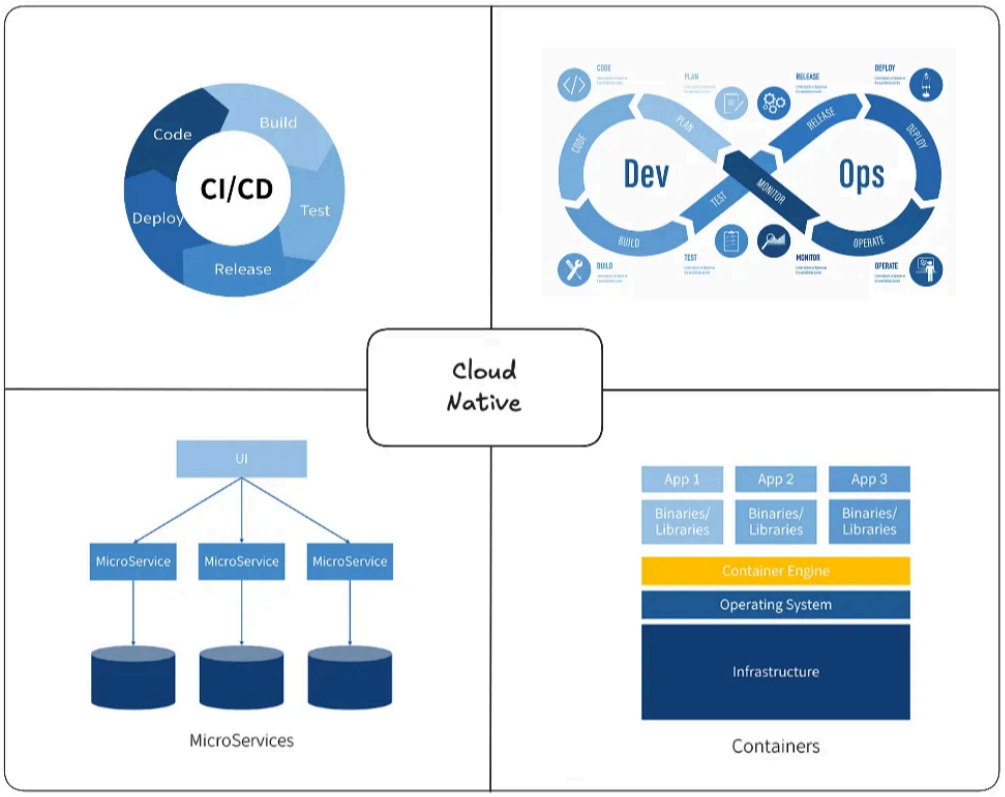
- **확장 가능한 아키텍처**
  - 시스템의 수평적 확정에 유연
  - 확장된 서버로 시스템의 부하 분산, 가용성 보장
  - 시스템 또는 서비스 애플리케이션 단위의 패키지 (컨테이너 기반 패키지)
  - 모니터링
- **탄력적 아키텍처**
  - 서비스 생성 = 통합 - 배포, 비즈니스 환경 변화에 대응 시간 단축
  - 분할된 서비스 구조
  - 무상태 통신 프로토콜
  - 서비스의 추가와 삭제 자동으로 감지
  - 변경된 서비스 요청에 따라 사용자 요청 처리 (동적 처리)
- **장애 격리(Fault isolation)**
  - 특정 서비스에 오류가 발생해도 다른 서비스에 영향을 주지 않음

전통적인 Architecture와의 차이점

항목	전통적인 방식	Cloud Native 방식
배포 방식	수동, 서버에 직접 배포	자동화된 CI/CD
인프라	물리 서버 또는 고정된 VM	클라우드 기반, 동적 할당
확장성	수동 확장, 어려움	자동 확장 가능
애플리케이션 구조	모놀리식(Monolith)	마이크로서비스(MSA)

Cloud Native Application

클라우드 환경을 적극 활용하기 위해 컨테이너, 마이크로서비스, DevOps, CI/CD 등을 기반으로 개발된 확장 가능하고 회복력 있는 애플리케이션



Cloud Native 핵심 구성 요소

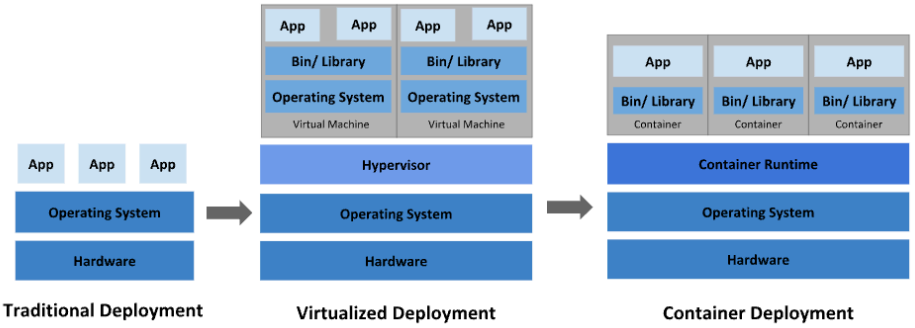
1. **마이크로 서비스** : 하나의 애플리케이션을 작고 독립적인 서비스 단위로 나누어 개발
2. **컨테이너화** : 애플리케이션과 그 실행 환경(라이브러리, 의존성 등)을 하나의 단위로 패키징
3. **배포 자동화 파이프라인 (CI/CD)** : CI와 CD를 통해 코드 변경 사항을 자동으로 테스트하고 빌드
4. **DevOps** : 코드 배포, 인프라 관리, 모니터링 등 운영 업무를 자동화하고 안정적인 서비스를 제공

▼ **Containerization**

컨테이너 가상화

- 기존에 로컬 환경에서 운영하던 시스템을 클라우드 환경으로 이전함으로써, 더 적은 비용으로 탄력적이고 안정적인 시스템 구축이 가능해짐

- Traditional Deployment → Virtualized Deployment → Container Deployment



배포 방식 진화 흐름 (전통적인 물리 서버 중심 배포, Hypervisor를 이용한 VM 기반 가상화 배포, 컨테이너 기반 배포)

→ OS 레벨에서 격리된 경량화된 환경(Ex. Docker 컨테이너)을 통해 애플리케이션을 실행하며 빠른 배포, 이식성, 확장성이 뛰어나 Cloud Native 및 DevOps 환경에 최적화됨

▼ **CI/CD**

지속적 통합, CI (Continuous Integration)

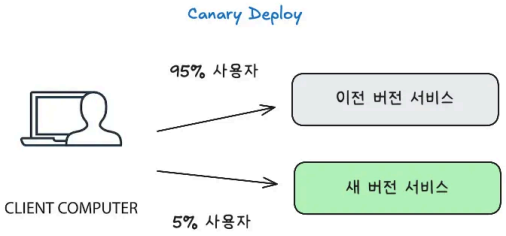
- 통합 서버, 소스 관리(SCM), 빌드 도구, 테스트 도구 등을 의미
- Ex. Github Actions, Jenkins, Team CI, Travis CI
- 일반적으로 Git과 같은 형상 관리 시스템과 연동하여 사용

지속적 배포, CD (Continuous Delivery / Deploy)

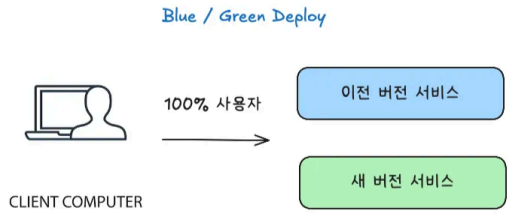
- 지속적인 전달 혹은 배포를 뜻함
- Continuous Delivery는 운영 배포를 수동으로 수행하는 방식
- Continuous Deployment는 운영 배포까지 자동으로 수행하는 방식

→ 자동화 파이프라인 구축

카나리 배포와 블루그린 배포



새로운 버전을 일부 사용자에게 먼저 배포하여 문제 여부를 확인한 뒤 점진적으로 전체에 적용하는 방식



기존 버전(Blue)과 새 버전(Green)을 동시에 유지하다가, 검증이 끝나면 트래픽을 일괄 전환하는 방식

▼ **DevOps**

- 개발(Development), 품질 보증(QA), 운영(Operations) 영역 간의 통합과 협업
- 고객의 요구사항을 빠르게 반영하고 만족도 높은 결과를 제시하는 것이 목표
- 서비스의 구조를 작은 단위로 분할하여 통합, 테스트, 배포가 용이할 수 있도록 함
- Ex. Docker, Kubernetes, Terraform, Prometheus, Grafana 등의 도구와 함께 활용됨

12-Factors

클라우드 네이티브 애플리케이션을 설계할 때 따라야 할 12가지 원칙

Codebase	하나의 코드베이스, 여러 배포 환경 (한 저장소 per 앱)
Dependencies isolation	의존성은 명시적으로 선언하고 격리해서 관리
Configuration	설정은 코드와 분리하여 환경 변수로 관리
Linkable Backing Services	DB, MQ 등 외부 리소스를 서비스처럼 연결 (로컬/원격 구분 X)
Stages of creation (Build, Release, Run)	빌드, 릴리스, 실행을 명확히 분리
Stateless Processes	애플리케이션은 상태를 저장하지 않는(Stateless) 프로세스로 실행
Port Binding	자체적으로 HTTP 포트를 바인딩하여 서비스 제공 (예: Tomcat 내장)
Concurrency	프로세스를 수평 확장(Scale out) 가능하게 설계
Disposability	빠른 시작과 종료, 장애 시 빠르게 대체되도록
Dev/Prod Parity	개발과 운영 환경의 차이를 최소화
Logs	로그는 이벤트 스트림으로 취급 (stdout/stderr로 출력)
Admin Processes	관리 작업(마이그레이션 등)은 별도의 일회성 프로세스로 실행

이후 추가된 3가지 원칙

API First	모든 기능은 먼저 API로 설계되어야 하고, 명확하게 문서화되어야 한다.
Telemetry (관측 가능성)	모든 지표는 수치화 및 시각화되어 실시간으로 모니터링 가능해야 한다.
Authentication / Authorization	API 사용 시 인증(Authentication)과 인가(Authorization)는 필수적이다.

12 Factors의 주요 원칙 요약

- 구성과 코드를 분리하라
- 외부 의존성은 격리하고 선언하라
- 상태 없는 프로세스로 구성하고 빠르게 시작하고 종료할 수 있도록 하라
- 환경 간 차이를 최소화하고, 일관된 배포 및 실행 흐름을 갖춰라
- 로그나 관리 작업은 외부 시스템에서 제어하라

Monolithic vs. Microservice

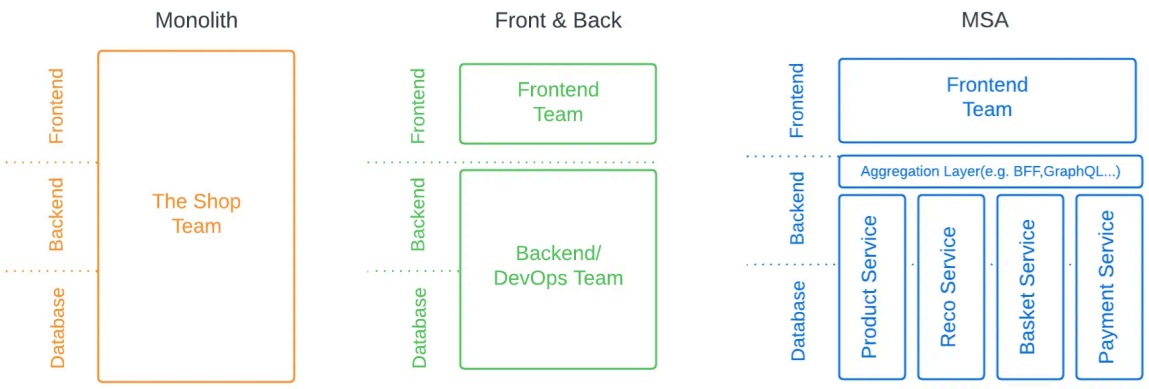
Monolithic Architecture

- 모든 업무 로직이 하나의 애플리케이션 형태로 패키지 되어 서비스
- 애플리케이션에서 사용하는 데이터가 한곳에 모여 참조되어 서비스되는 형태

MSA(Microservice Architecture)

- 기능별로 서비스를 작게 나누어 각각 독립적으로 개발, 배포, 확장 가능한 구조
- 각 마이크로 서비스는 독립된 데이터베이스와 로직을 가짐

- 서비스 간 통신은 REST API, 메시징 시스템 등을 통해 이루어짐



Monolith vs Front & Back vs Microservice Architecture

- 프론트엔드와 백엔드의 분리 개발 방식은 사용자 화면(UI)과 서버 로직을 독립적으로 개발하는 구조이다. 예를 들어, 모바일 앱에서는 안드로이드(Java/Kotlin)나 iOS(Swift 등)로 프론트를 개발하고, 서버와는 HTTP 같은 프로토콜로 통신한다. 이렇게 분리하면 각각의 환경에서 최적화된 개발이 가능하고, UI만 수정할 경우 백엔드 재빌드 없이 배포할 수 있다.

- 마이크로서비스 아키텍처(MSA)는 백엔드 서비스들을 의미 있는 기능 단위로 나눠 각각 독립적으로 운영하는 방식이다. 각 서비스는 자체 비즈니스 로직과 데이터베이스를 가지며, 통합된 하나의 시스템이 아닌, 작고 유연한 단위로 구성된다.

Microservice 특징

항목	설명
Small Well Chosen Deployable Units	작은 단위로 잘 나누어 독립 배포 가능
Bounded Context	명확한 경계와 책임 영역
RESTful	REST API 등 표준 프로토콜로 서비스 간 통신
Configuration Management	환경별 설정 효과적 관리
Cloud Enabled	클라우드 환경에서 원활한 운영 가능
Dynamic Scale Up And Scale Down	자동 확장 및 축소 가능
CI/CD	지속적 통합 및 배포 자동화
Visibility	모니터링, 로그 등으로 상태 명확 파악

Microservice 도입 시 고려할 점 (Challenges)

항목	설명
Everything should be a microservice? - X	모든 것을 마이크로서비스로 만들 필요는 없음
Multiple Rates of Change	서비스별 변경 빈도 차이
Independent Life Cycles	독립적인 서비스 생명주기
Independent Scalability	개별 서비스별 확장 가능
Isolated Failure	장애가 전체에 영향 미치지 않도록 격리
Simplified Interactions with External Dependencies	외부 의존성과의 상호작용 단순화
Polyglot Technology	다양한 기술 스택 유연 사용

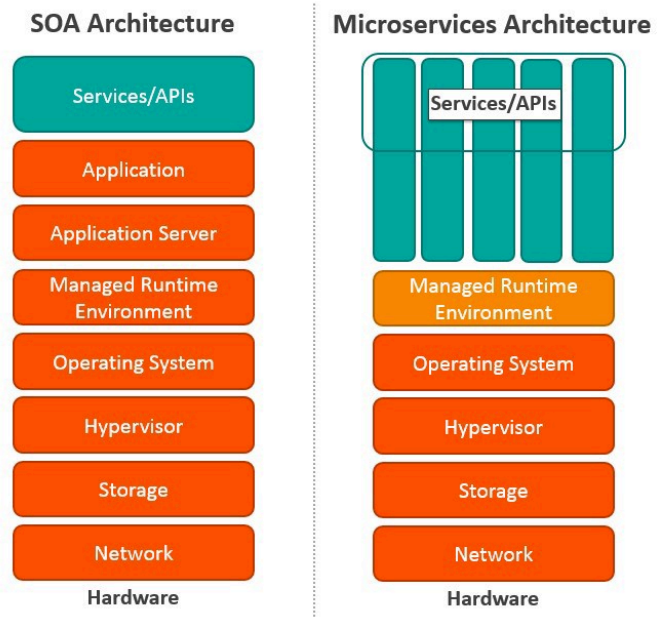
SOA vs. MSA

SOA(Service Oriented Architecture) - 서비스 지향 아키텍처

- 비즈니스 측면에서의 서비스 재사용성 (재사용을 통한 비용 절감을 지향)
- ESB(Enterprise Service Bus)라는 서비스 채널을 이용함으로써 서비스를 공유하고 재사용

MSA(Microservice Architecture) - 마이크로 서비스 아키텍처

- 한 가지 작은 서비스에 집중
- 서비스 간의 결합도를 낮추어 변화에 능동적으로 대응
- 서비스를 공유하지 않고 독립적으로 실행 (각 독립된 서비스가 노출된 REST API를 사용)



https://medium.com/@SoftwareDevelopmentCommunity/what-is-service-oriented-architecture-fa894d11a7ec

> SOA는 ESB라는 중앙 미들웨어로 서비스들을 연결하는 반면, MSA는 REST API나 Kafka 같은 경량 메시징을 통해 서비스들이 독립적으로 통신한다.

Microservice Architectures Structures

Service Mesh Capabilities (Service Mesh가 제공하는 주요 기능이나 역할)

서비스 메시란, 마이크로서비스 간 통신을 추상화하고 관리해주는 인프라 계층으로, 주로 서비스 간 통신을 위한 네트워크 제어 및 보안을 담당한다.

Service Mesh가 MSA 인프라에서 수행하는 역할?

- **MSA 아키텍처**에서는 서비스 간의 복잡한 통신을 담당하는 **미들웨어 계층**이 필요
  → Service Mesh가 맡음. 아래에는 Service Mesh가 제공하는 기능 목록
  - 프록시 역할, 인증, 권한 부여, 암호화, 서비스 검색, 요청 라우팅, 로드 밸런싱
  - 자가 치유 복구 서비스

- 서비스 간의 통신과 관련된 기능을 자동화 (개발자는 비즈니스 로직에만 집중)

MSA 기반 기술 분류

분류	설명	예시 기술
Gateway	외부 요청을 내부 서비스로 라우팅하고 인증, 필터링, 로깅 등의 역할 수행	Spring Cloud Gateway, Kong, NGINX, AWS API Gateway
Resilient Service Mesh / Meta Services	서비스 간 통신, 트래픽 관리, 장애 복구, 보안 등을 자동화하고 추상화	Istio, Linkerd, Consul Connect, Envoy
Runtime	마이크로서비스가 실행되는 경량화된 환경, 컨테이너 및 오케스트레이션 도구 포함	Docker, Kubernetes, CRI-O, containerd
Backing Services	마이크로서비스에서 사용하는 데이터베이스, 캐시, 메시징 큐 등 외부 종속 자원	PostgreSQL, Redis, Kafka, RabbitMQ, S3
Frameworks	마이크로서비스 개발을 지원하는 경량 프레임워크 및 라이브러리	Spring Boot, Spring Cloud, Micronaut, Quarkus
Automation (CI/CD)	코드 변경 → 테스트 → 빌드 → 배포까지의 자동화 파이프라인 구성	GitHub Actions, GitLab CI/CD, Jenkins, ArgoCD
Telemetry (관측 가능성)	서비스 상태를 수집·시각화하고, 추적/로그/모니터링을 통한 장애 감지 및 성능 최적화 지원	Prometheus, Grafana, ELK Stack, Zipkin, Jaeger

Spring Cloud vs Kubernetes

Spring Cloud : **마이크로서비스를 코드로 개발할 수 있도록 지원**하는 프레임워크.

Kubernetes : **마이크로서비스를 배포하고 운영할 수 있도록 관리**하는 플랫폼.

→ 둘을 함께 사용하는 경우가 많음

구분	Spring Cloud	Kubernetes
역할	MSA를 개발자가 쉽게 구현하도록 돕는 **프레임워크**	마이크로서비스를 실행/운영하는 **인프라 플랫폼**
관점	개발자 중심: 서비스 개발을 위한 도구	운영자 중심: 서비스 배포와 관리 자동화
주요 기능	서비스 디스커버리, 설정 관리, API Gateway, 로드밸런싱 등	컨테이너 오케스트레이션, 스케일링, 롤링 배포, 복구 등
기반 기술	Java + Spring Boot 기반	Docker + 컨테이너 기반
구성 방식	코드로 설정 및 구현	YAML로 설정하여 클러스터에서 실행
배포 환경	자체 서버, 클라우드 모두 가능	클라우드 환경에 최적화 (EKS, GKE, AKS 등)

Spring Cloud로 Eureka(서비스 디스커버리), Config Server(환경설정)를 구성했다면 Kubernetes는 그 Spring Cloud 서비스를 컨테이너로 실행하고, 트래픽을 라우팅하고, 죽으면 재시작하고, 스케일을 조절해줌

Spring Cloud

Spring Boot + Spring Cloud

- Main Projects (사용할 프로젝트)
  - Spring Cloud Config
  - Spring Cloud Netflix
  - Spring Cloud Security
  - Spring Cloud Sleuth
  - Spring Cloud Starters
  - Spring Cloud Gateway

    - Spring Cloud OpenFeign
- 필요한 서비스 구성
    - Centralized configuration management (환경 설정 관리)
        - Spring Cloud Config Server
    - Location Transparency
        - Naming Server (Eureka) |
    - Load Distribution (Load Balancing)
        - Ribbon (Client Side)
        - Spring Cloud Gateway
    - Easier REST Clients
        - FeignClient
    - Visibility and monitoring
        - Zipkin Distributed Tracing
        - Netflix API gateway
    - Fault Tolerance
        - Hystrix