2021년 광주대학교 특강

OSS를 활용한 MSA

이기하
dasomell@gmail.com
2021.10

목 차

1. 발표자 소개

- **現한화시스템 ICT부문(2021~)**
  - HKS(Hanwha Kubernetes Service) Platform 개발 리딩
- **前SK주식회사 C&C(2012 ~ 2021)**
  - Cloud 프로젝트 다수 구축(2017 ~ 2020)
  - 사내 강의 다수
  - 사내 개발자 대회 다수 분야 3등(2018)
- **〈나도 해보자! 시리즈〉 오픈커뮤니티 세미나 발표**
  - 나도 해보자! 표준프레임워크 개발환경 구축
  - 나도 해보자! Cloud Project with Kubernetes 등
- **오픈플랫폼(PaaS) 전문가과정 강의(2016)**
- **슈퍼개발자K 시즌3 동상 수상(2014)**
- **現오픈커뮤니티 리더(2015 ~)**
- **前T-Hub(SK그룹 기술커뮤니티)**
  - DevOps Master(2020~2021)

2. 마이크로 서비스 아키텍처 (MSA)

❑ MSA 정의

- 마이크로서비스(microservice)는 애플리케이션을 느슨히 결합된 서비스의 모임으로 구조화하는 서비스 지향 아키텍처(SOA) 스타일의 일종인 소프트웨어 개발 기법이다.
- 마이크로서비스 아키텍처에서 서비스들은 섬세(fine-grained)하고 프로토콜은 가벼운 편이다.
- 애플리케이션을 더 조그마하나 여러 서비스로 분해할 때의 장점은 모듈성을 개선시키고 애플리케이션의 이해, 개발, 테스트를 더 쉽게 해주고 애플리케이션 침식에 더 탄력적으로 만들어 준다.
- 규모가 작은 자율적인 팀들이 팀별 서비스를 독립적으로 개발, 전개, 규모 확장을 할 수 있게 함으로써 병렬로 개발할 수 있게 한다.
- 또, 지속적인 리팩토링을 통해 개개의 서비스 아키텍처가 하나로 병합될 수 있게 허용한다. 마이크로서비스 기반 아키텍처는 지속적인 배포와 전개(디플로이)를 가능케 한다.

출처: 위키 백과 “마이크로서비스” (https://ko.wikipedia.org/wiki/마이크로서비스)

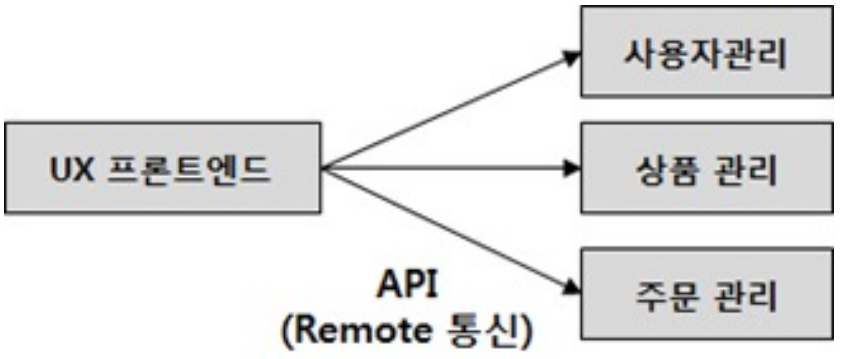
2. 마이크로 서비스 아키텍처 (MSA)

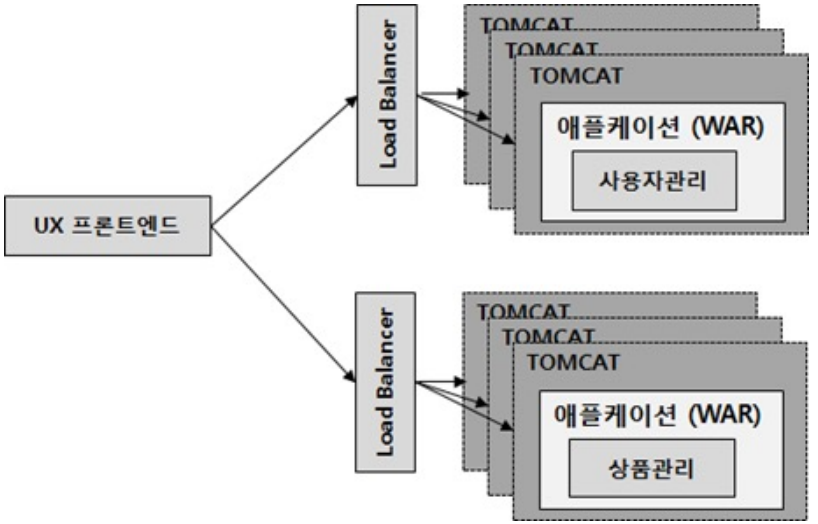
❑ MSA 정의

- 대용량 웹서비스가 많아짐에 따라 정의된 아키텍쳐
  근간은 SOA (Service Oriented Architecture : 서비스 지향 아키텍쳐)
- SOA는 엔터프라이즈 시스템을 중심으로 고안된 아키텍쳐
  마이크로 서비스 아키텍쳐는 SOA 사상에 근간을 두고, 대용량 웹서비스 개발에 맞는 구조로 사상이 경량화 되고, 대규모 개발팀의 조직 구조에 맞도록 변형된 아키텍쳐

❑ 구조

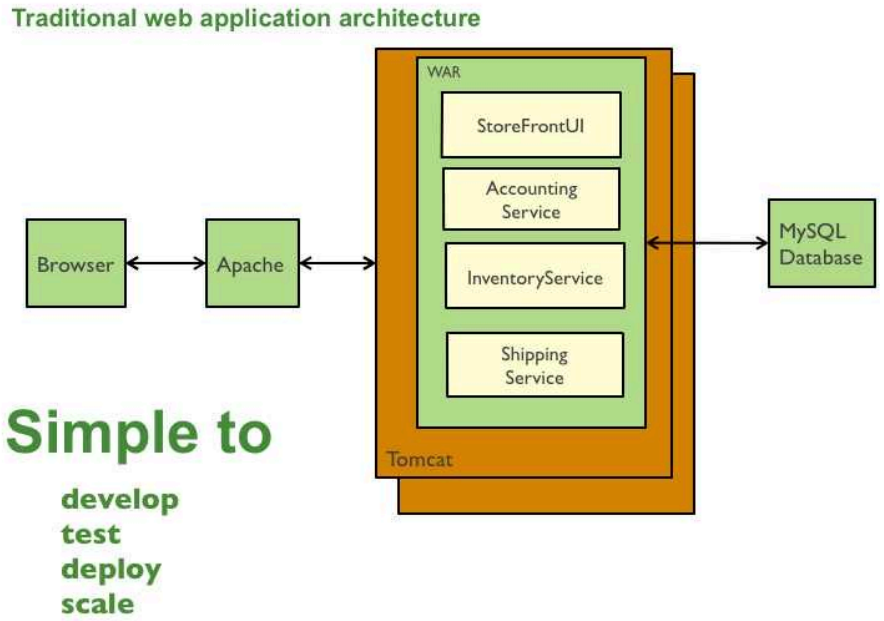
- API를 이용하여 타 서비스와 통신
- 배포도 각 서비스의 독립된 서버로 구성

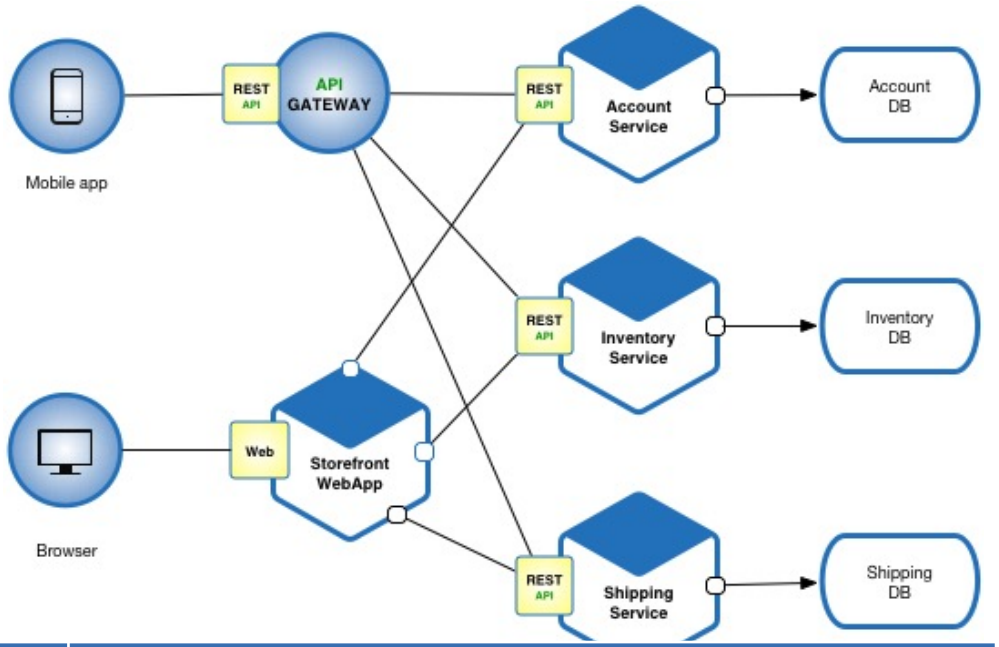




2. 마이크로 서비스 아키텍처 (MSA)

❑ Monolithic vs Microservice Architecture

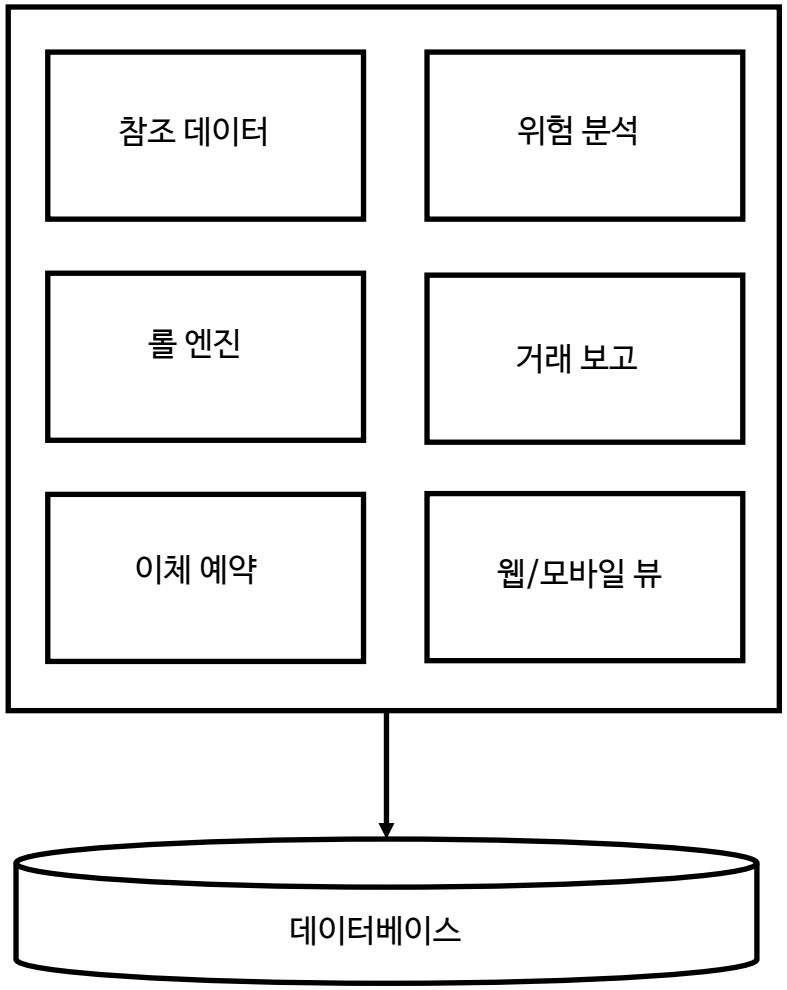




	모놀리스	마이크로 서비스
장점	• 간단한 개발 • 간단한 배포 • 간편한 확장	• 크고 복잡한 서비스를 지속적으로 배포 가능 • 개발자의 이해도 향상 • 향상된 장애 격리 • 기술 스택 선택에 대한 유연성
단점	• 개발 생산성 저하 • 지속적인 배포의 어려움 • 각 구성요소의 독립적 확장 불가 • 새로운 기술의 채택이 어려움	• 분산 시스템의 복잡성 처리 • 배포 복잡성 • 높은 메모리 소모

이미지 출처: https://microservices.io/patterns/monolithic.html

2. 마이크로 서비스 아키텍처 (MSA)



모놀리스 아키텍처(거래 애플리케이션)

장점

- 모든 기능은 하나의 소프트웨어 애플리케이션으로 컴파일 되고 배포
- 소규모로 시작
- 전체 애플리케이션은 응집력이 높기 때문에 초기 단계에서 빠르게 개발
- 전체 애플리케이션은 하나의 시스템이므로 배포하고 테스트 하기 쉬움

단점

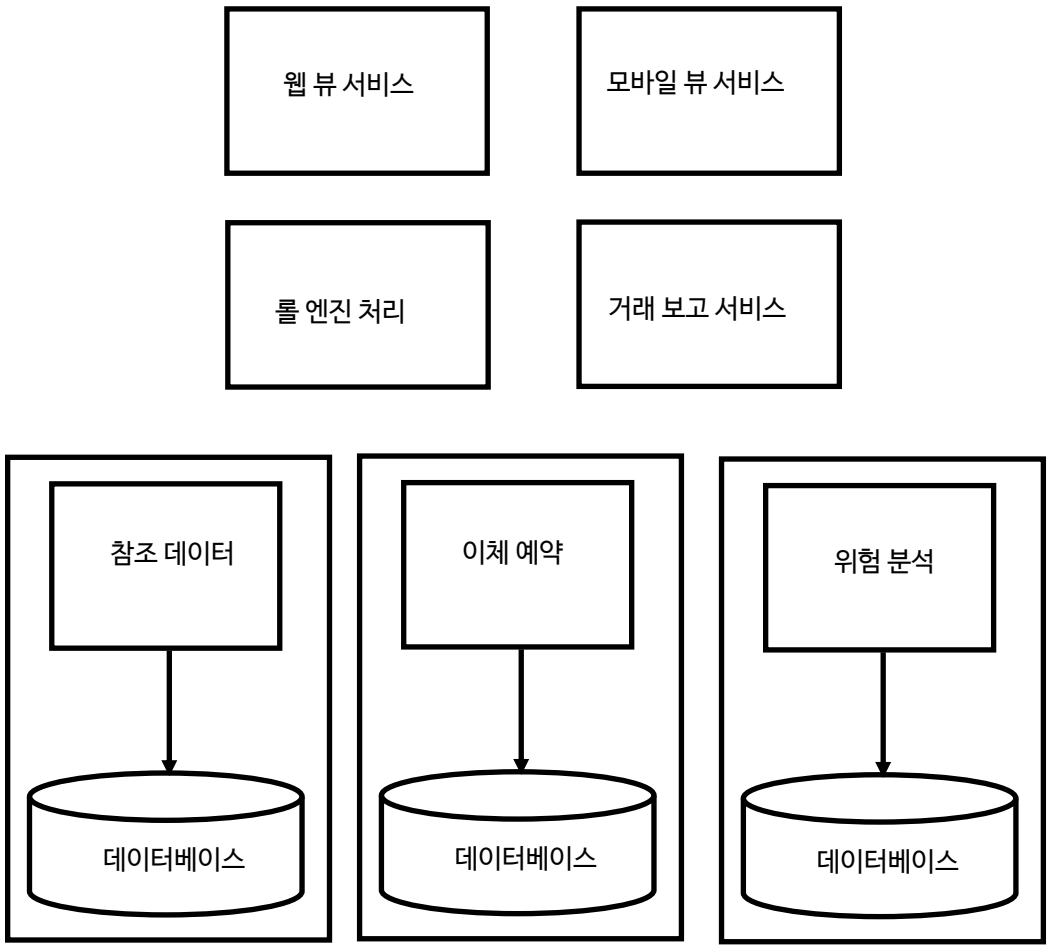
- 애플리케이션이 유기적으로 발전함에 따라 유지 관리, 운영이 어려움
- 시간이 지남에 따라 여러 개발팀에서 유지관리
- 각 팀은 애플리케이션의 각 하위 시스템을 담당
- 하위 시스템은 고도로 결합, 일정기간동안 새로운 기능을 사용할 수 있도록 개발 팀 간에 상호 의존성 존재

하나의 시스템에 따른 문제점

- 빠른 기능 롤아웃
- 더 이상 사용되지 않는 스택
- 가파른 학습 곡선
- 스케일링 문제

출처: Istio로 시작하는 서비스 메시(출판사 에이콘)

2. 마이크로 서비스 아키텍처 (MSA)



마이크로서비스 아키텍처(거래 애플리케이션)

민첩성

- 아키텍처가 제공하는 느슨한 결함으로 개발 가속화
- 복원력과 장애 분리를 촉진
- 특정 서비스를 독립적으로 수정 배포(서비스 품질)

혁신성

- 소규모 독립 개발 팀이 서비스 범위 내에서 완전한 소유권 획득
- 서비스별 적절한 도구와 프레임워크 선택 가능
- 조직의 기술 스택과 혁신을 향상

확장성

- 필요 서비스만 확장 가능
  (느슨한 결함으로 트랙픽 분석 가능)

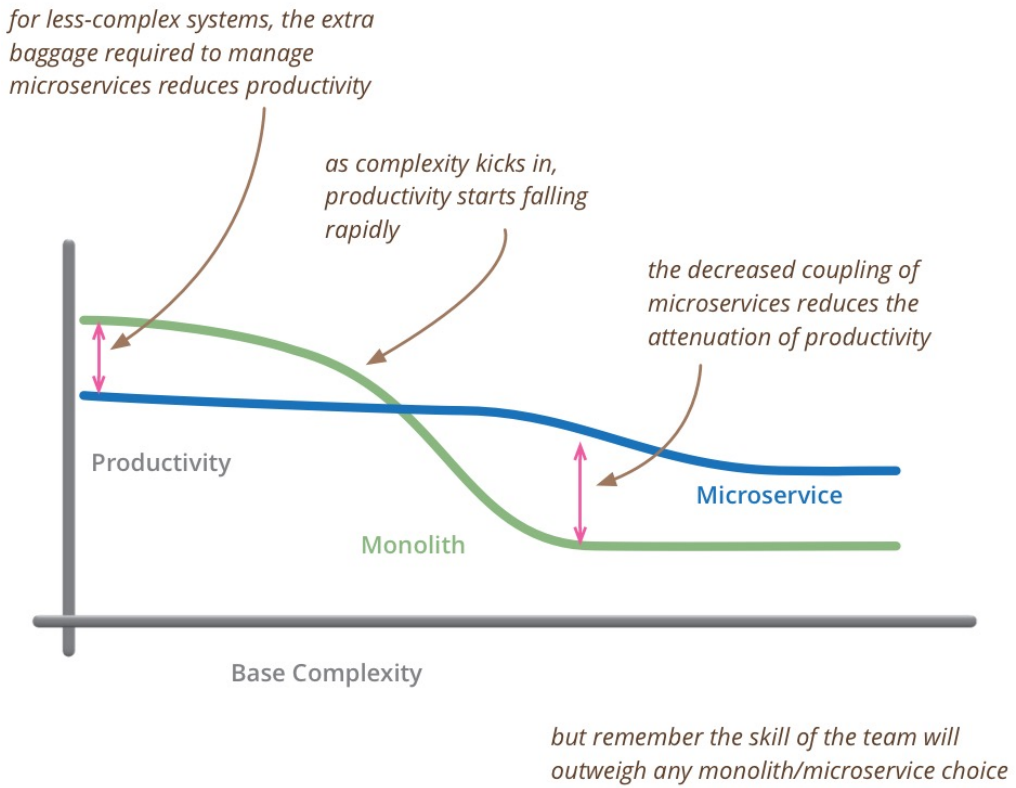
유지 보수성

- 모놀리스 소프트웨어에 가파른 학습 곡선 해결
- 기술 부채 해결 가능

출처: Istio로 시작하는 서비스 메시(출판사 에이콘)

2.1 마이크로 서비스 아키텍처 (MSA) 도전 과제

❑ Microservice 생산성



마이크로 서비스 사용 여부의 핵심은 고려중인 시스템의 **복잡성**입니다.

마이크로 서비스를 사용할 때 자동화 된 배포, 모니터링, 장애 처리, 최종 일관성 및 분산 시스템이 도입하는 기타 요인에 대해 작업해야 합니다

모놀리식으로 관리하기에 특별히 복잡한 시스템을 운영할 상황이 아니면 마이크로서비스는 고려할 필요조차 없다.

– 마틴 파울러 –

이미지 출처: https://martinfowler.com/bliki/MicroservicePremium.html

2.1 마이크로 서비스 아키텍처 (MSA) 도전 과제

❑ Microservice 성숙도 모델

	Level0 Traditional	Level1 Basic	Level2 Intermediate	Level3 Advanced
Application	Monolithic	Service Oriented Integrations	Service Oriented Applications	API Centric
Database	One Size Fit All Enterprise DB	Enterprise DB + No SQLs and Light databases	Polyglot, DBaaS	Matured Data Lake /Near Realtime Analytics
Infrastructure	Physical Machines	Virtualization	Cloud	Containers
Monitoring	Infrastructure	App & Infra Monitoring	APMs	APM & Central Log Management
Process	Waterfall	Agile and CI	CI & CD	DevOps

출처: A Microservices Maturity Model, from “Spring 5.0 Microservices,” 2nd Edition”by Rajesh R V (O’Reilly)

2.1 마이크로 서비스 아키텍처 (MSA) 도전 과제

❑ 신뢰할 수 있는 네트워크

- 분산 아키텍처에는 동작하는 부분이 많으며, 서비스 중 하나가 언제든지 실패할 가능성이 높음
- 서비스를 구축하는 동안 필요한 탄력성 패턴 추가(예: 시스템 이중화)

❑ 네트워크 지연 시간 없음

- 네트워크 통신은 로컬 통신보다 느림
- 부하로 서비스가 느리게 응답
- 서비스 지연 테스트는 매우 어려움(개발자의 성숙도가 중요)
- 회로 차단기 패턴 도입, 데이터 캐싱

❑ 무한한 대역폭

- 프로덕션에 배포되는 서비스 수가 기하 급수적으로 증가
- 애플리케이션에 할당량 할당 및 소비 추적을 위한 매커니즘 필요
- 서버 측 로드 밸런서에 많은 부하로 성능 저하

출처: Istio로 시작하는 서비스 메시(출판사 에이콘)

2.1 마이크로 서비스 아키텍처 (MSA) 도전 과제

❑ 안전한 네트워크

- 많은 기술적 선택으로 팀은 각각의 버그 수정/문제를 추적
- 서비스 간 통신을 제어(서비스 수준 인증으로 알 수 없는 서비스 연결을 필터링)
- 일반 텍스트 교환을 수행하는 애플리케이션은 중요한 데이터 노출(보안 프로토콜 사용)

❑ 변하지 않는 토폴로지[1)]

- 서비스간의 종속성이 낮음(신속하게 구축, 릴리즈, 배포 가능)
- 동적으로 추가된 다른 서비스 연결 필요(서비스 디스커버리 기능 추가로 전체 시스템의 탄력성 향상)

❑ 한 명의 관리자

- 소수의 사람이 분산 시스템에 완전한 운영 지식을 갖는 것은 불가능함
- 마이크로서비스는 팀을 자율적으로 만드는 것이 목표
- 각 팀은 자체 서비스를 유지 관리할 책임이 있음

토폴로지(topology)[1)]: 컴퓨터 네트워크의 요소들을 물리적으로 연결해 놓은 것, 또는 그 연결 방식을 말한다.

2.1 마이크로 서비스 아키텍처 (MSA) 도전 과제

❑ 전송 비용 없음

- 마이크로서비스에서는 종속된 서비스를 많이 호출
- 데이터 교환 비용 발생(직렬화, 역직렬화)
- SOAP/XML 〉 JSON 〉 바이너리 프로토콜

❑ 동일 네트워크

- 모든 시스템이 동일한 하드웨어 세트에서 실행되고 모든 애플리케이션이 표준 프로토콜로 통신 하는 경우 네트워크가 같다고 분류
- 애플리케이션이 배포될 때마다 동일한 양의 리소스를 얻도록 제한

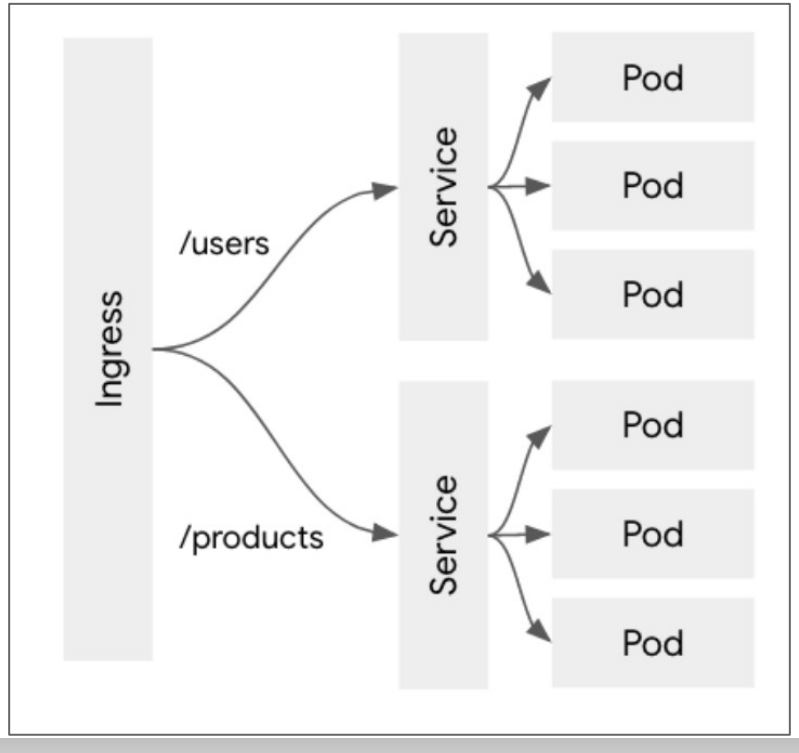
❑ 인프라스트럭처

- 마이크로서비스 아키텍처는 모놀리스 아키텍처보다 훨씬 복잡
- 비용 효율적으로, 필요에 따라 인프라를 프로비저닝[2] 할 수 있는 방법을 강구해야 함

프로비저닝(provisioning) [2]: 사용자의 요구에 맞게 시스템 자원을 할당, 배치, 배포해 두었다가 필요 시 시스템을 즉시 사용할 수 있는 상태로 미리 준비해 두는 것을 말한다.

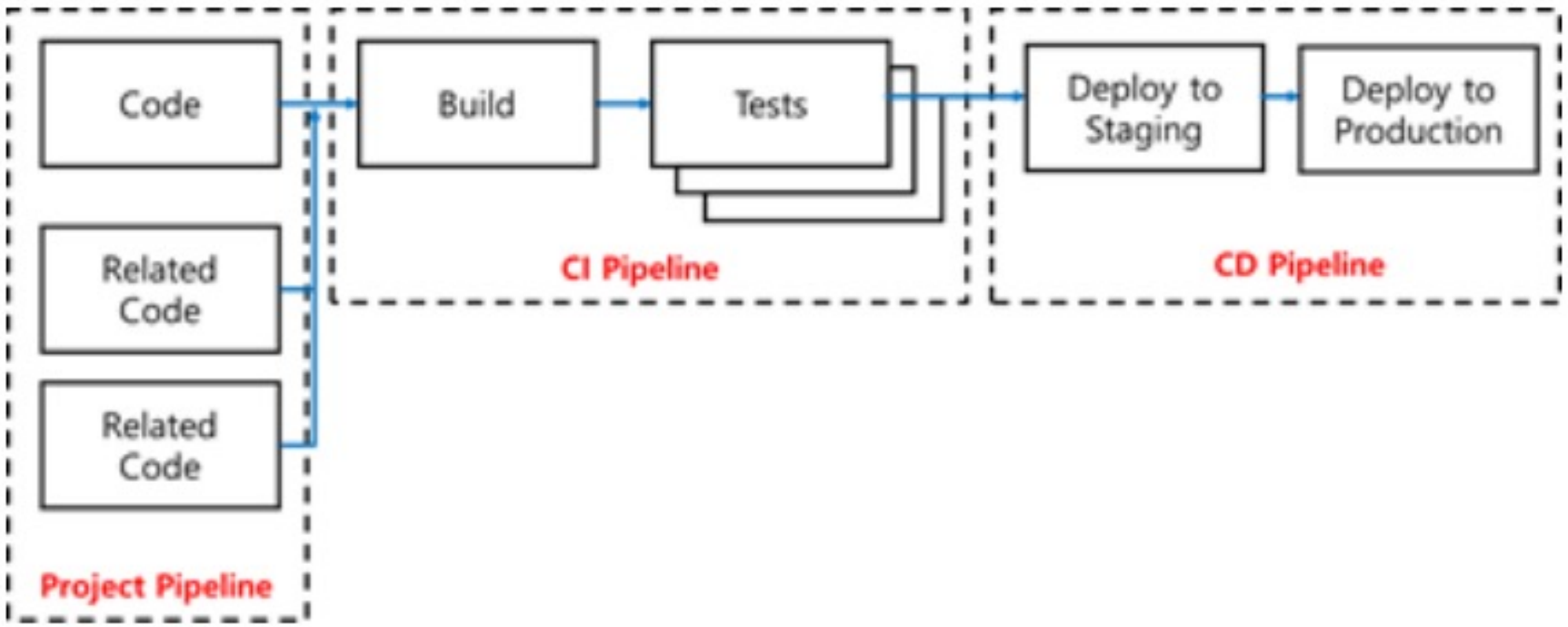
❑ API G/W

- MSA에서 API Gateway는 Public으로 Client에게 공개되어야 하는 실제 유입점
- API 게이트웨이 및 마이크로 게이트웨이는 API 및 마이크로서비스 아키텍처에서 핵심적인 역할을 담당
- MSA 서비스간의 라우팅을 하기 위해서는 API 게이트웨이를 넣는 경우가 많은데, 이 경우에는 API 게이트웨이에 대한 관리 포인트가 생기기 때문에, URL 기반의 라우팅 정도라면, API 게이트웨이처럼 무거운 아키텍쳐 컴포넌트가 아니라, L7 로드 밸런서 정도로 위의 기능을 모두 제공이 가능



❑ CI/CD

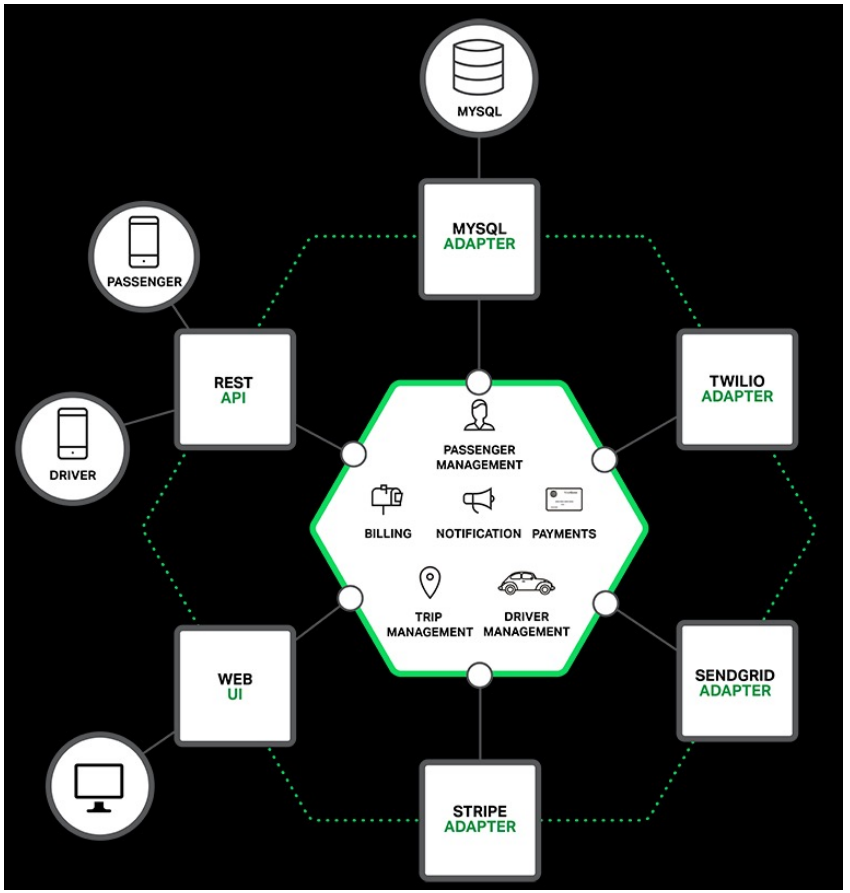
- CI: Continuous Integration(지속적인 통합)
- CD: Continuous Deployment(지속적인 배포)



2.2 마이크로 서비스 아키텍처 (MSA) 기술

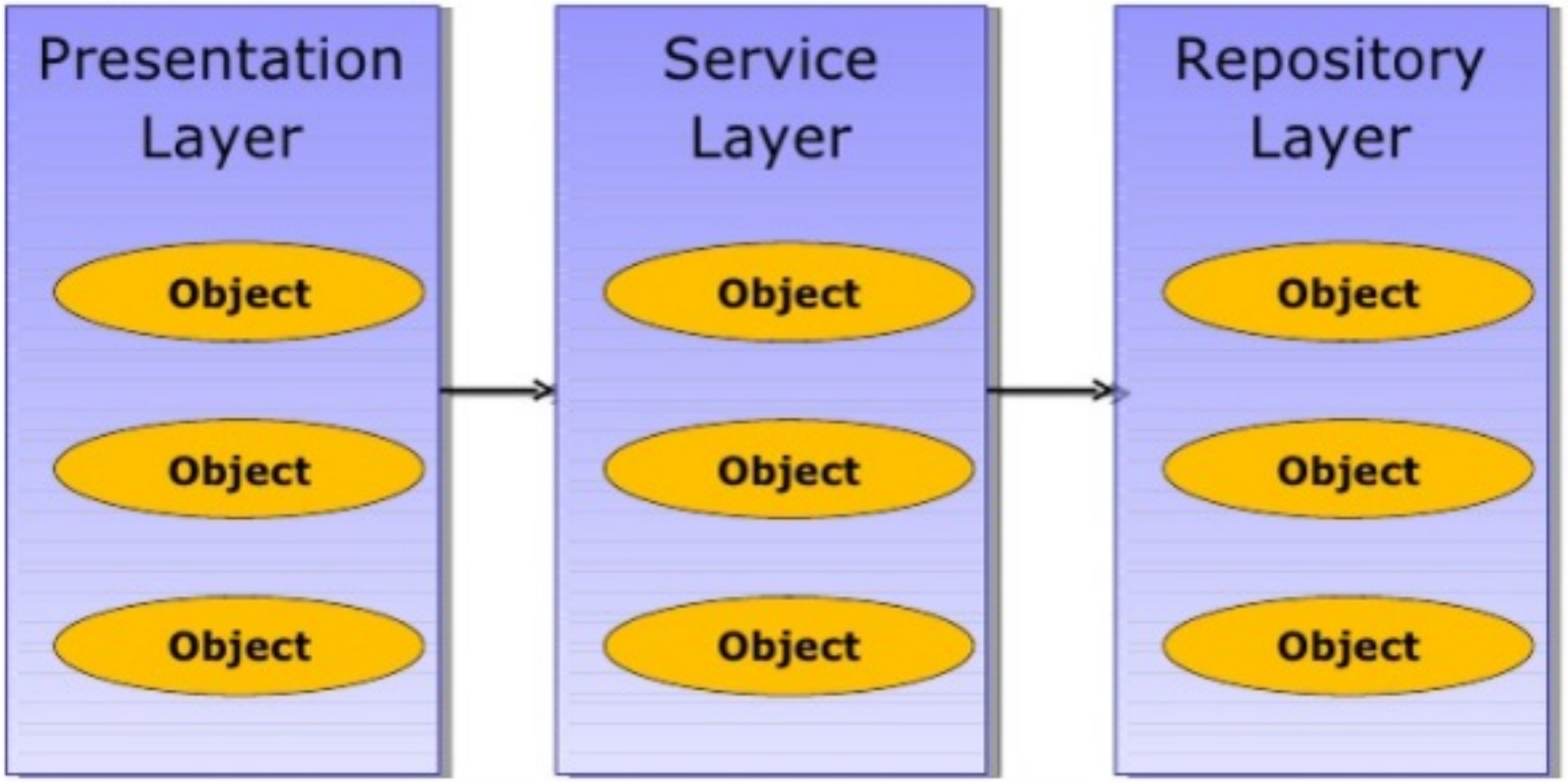
❑ 헥사고널 아키텍처

- 논리적으로는 여러 모듈로 이루어져 있지만 단일(monolith) 애플리케이션으로 배포되는 특성상 개발, 테스트, 패키징, 배포 간편하고 쉬운 장점
- 애플리케이션이 거대해 질 경우 심플한 Monolithic 구조에서는 아래의 문제가 발생
  - 생산성: 복잡도가 증가, 배포 시간도 오래 걸림,
    생산성 저하, 빠른 delivery의 어려움.
  - Scale: 리소스 요구사항이 상충될 때 확장성에 제약.
  - Reliability: 앱이 하나의 프로세스로 돌기 때문에
    한 모듈의 버그가 전면 장애로 이어짐.
  - adopt new tech speed: 새로운 언어,
    프레임워크 기술 적용이 극도로 어려움.
  - Hiring: 오래되고 비생산적인 기술 사용. 우수한 인재 채용이 어려움.



❑ 레이어드 아키텍처

- Presentation: 화면 조작 또는 사용자의 입력을 처리하기 위한 관심사를 모아놓은 레이어
- Domain: 비즈니스와 관련된 도메인 로직을 처리하는 레이어
- Data Source: 도메인에서 필요로 하는 모든 데이터를 조작하기 위한 레이어



2.2 마이크로 서비스 아키텍처 (MSA) 기술

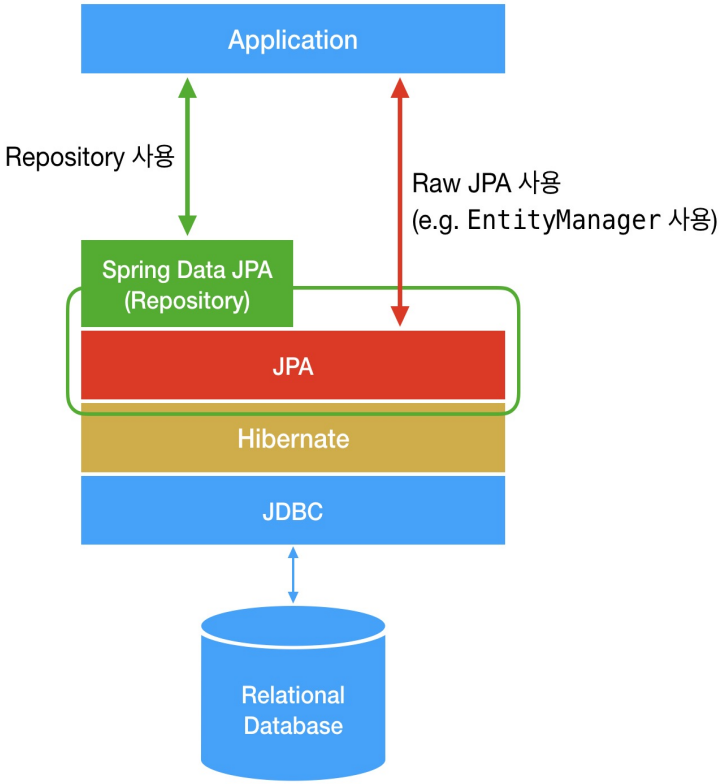
❑ MyBatise, JPA

- MyBatis: JDBC 를 좀더 편하게 사용할 수 있도록 객체를 SQL 이나 저장 프로시저와 매핑 해주는 퍼시스턴스 프레임워크 SQL 구문을 java 메소드와 매핑
- JPA: 자바 객체를 데이터테이블과 매핑(데이터베이스 객체를 자바 객체로 매핑하여 객체 간의 관계를 바탕으로 SQL을 자동으로 생성)

구분	장점	단점
Mybatis	• 다른 프레임워크에 비해 간단하다. • 소스 코드와 SQL 의 분리 (생산성, DBA와의 협업) • SQL을 직접 다뤄 복잡한 쿼리 작성, SQL 함수나 저장 프로시저를 자유롭게 이용가능	• 반복적인 코드와 CRUD SQL 작업 • SQL과 데이터베이스 벤더에 대한 종속성 (oracle -〉 mysql로 바꾸면 함수들을 변경)
JPA	• 생산성 CRUD 같은 간단한 쿼리는 자동 • entity에 추가 속성이 생기면 Mybatis 의 경우 쿼리에 각각 다 추가해줘야하지만 JPA는 Entity에 속성만 추가 시켜주면 됨 • 데이터 접근 추상화 벤더 독립성 • SQL 중심적인 개발에서 객체 중심으로 개발가능	• 상대적으로 높은 학습 곡선 • 복잡한 쿼리작성의 어려움 - JPQL을 통해 해결가능 - Native SQL로 직접 작성가능 - mybatis 와 혼용가능

❑ Spring Data JPA

- Spring에서 제공하는 모듈 중 하나로, 개발자가 JPA를 더 쉽고 편하게 사용할 수 있도록 도와줌
- JPA를 한 단계 추상화시킨 Repository라는 인터페이스를 제공
- 사용자가 Repository 인터페이스에 정해진 규칙대로 메소드를 입력하면, Spring이 알아서 해당 메소드 이름에 적합한 쿼리를 날리는 구현체를 만들어서 Bean으로 등록
- Spring Data JPA의 Repository의 구현에서 JPA를 사용
  (Spring Data JPA가 JPA를 추상화)
- Repository 인터페이스의 기본 구현체인 SimpleJpaRepository의 코드
  (내부적으로 EntityManager을 사용)



❑ Restful API 설계 원칙/성숙도

Level	내용
Level 0 〈 The Swarm of POX	• RPC (Remote Procedure Call) 형태로 resource 구분 없이 설계된 HTTP API 한 URI를 정의합니다. 모든 작업은 이 URI에 대한 POST 요청
Level 1 Resources	• resource 형태로 구분되어 있으나 action을 HTTP command로 CRUD (Create, Read, Update, Delete)로 표현하지 않은 HTTP API • 개별 리소스에 대한 별도의 URI
Level 2 HTTP Verbs	• resource 형태로 구분된 URI와 HTTP command로 CRUD 하나 self-descriptive hypermedia type를 가지지 않는 HTTP API • HTTP 메서드를 사용하여 리소스에 대한 작업을 정의
Level 3 Hypermedia Controls	• response payload 에 관련 URI를 포함하는 hypermedia1 로써의 속성을 지님으로써 code on demand2 속성을 지원할 수 있는 완전한 REST API • 하이퍼미디어(HATEOAS)를 사용

2.2 마이크로 서비스 아키텍처 (MSA) 기술

❑ RESTful API

- 일반적으로 REST라는 아키텍처를 구현하는 웹 서비스를 나타내기 위해 사용되는 용어
- 'REST API'를 제공하는 웹 서비스를 'RESTful'하다고 할 수 있음
- RESTful은 REST를 REST답게 쓰기 위한 방법으로, 누군가가 공식적으로 발표한 것이 아님
  REST 원리를 따르는 시스템은 RESTful이란 용어로 지칭

❑ RESTful의 목적

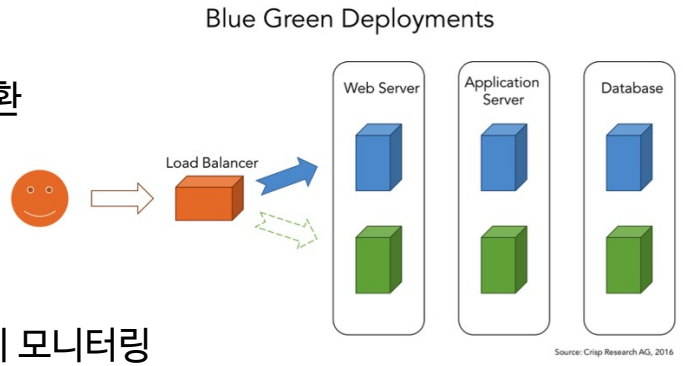
- 이해하기 쉽고 사용하기 쉬운 REST API를 만드는 것
- RESTful한 API를 구현하는 근본적인 목적이 성능 향상에 있는 것이 아니라 일관적인 컨벤션을 통한 API의 이해도 및 호환성을 높이는 것
- 성능이 중요한 상황에서는 굳이 RESTful한 API를 구현에 신중할 필요 있음

❑ RESTful 하지 못한 경우

- Ex1) CRUD 기능을 모두 POST로만 처리하는 API
- Ex2) route에 resource, id 외의 정보가 들어가는 경우(/students/updateName)

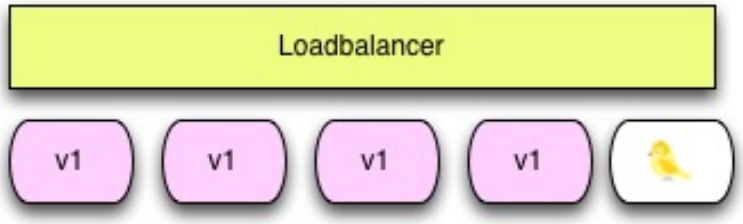
❑ Blue-Green 배포

- 지속적 업데이트는 최소한 오버헤드, 성능 영향, 다운타임으로 천천히 배포할 수 있어서 이상적
  - 완전히 배포된 후 새 버전을 가리키도록 부하 분산기를 수정하는 편이 유용할 경우가 존재하는데, 이 경우 Blue-Green 배포 사용
- 기존 Blue 버전과 새로운 Green 버전의 배포를 각각 하나씩 생성
- 새로운 Green 버전이 실행되면 서비스를 업데이트해 이 버전을 사용하도록 전환
- 단점: 클러스터의 리소스가 2배 이상 필요



❑ 카나리 배포(Canary Releases)

- 새로운 버전의 Application을 Production 환경으로 보내어 어떻게 동작하는지 모니터링 (다른 Application과의 연계, CPU, MEM, DISK 사용량 등을 미리 검증)
- 새로운 버전의 앱을 프로덕션 환경에 보내어, LB에서 일부 세션이 그 곳에 물리도록 만든다. 만약 이상 동작을 한다면 바로 Canary를 철수시켜 정상적인 버전으로 새로운 세션을 맺게 만들어 준다.
- 이 Release를 이용하여 Full Release 전에 발생할 수 있는 여러가지 Issues를 검증



2.3 12-Factor App 방법론

- **코드베이스 (Codebase)**
  - 버전 관리되는 하나의 코드베이스로 여러 곳에 배포
- **종속성 (Dependencies)**
  - 의존 관계를 명시적으로 선언하고 분리 환경에 의존하지 않도록 함
- **설정 (Config)**
  - 설정 정보는 애플리케이션 코드와 분리하고 환경 변수에 저장
- **백엔드 서비스 (Backing services)**
  - 백엔드 서비스를 연결된 리소스로 취급
- **빌드, 릴리스, 실행 (Build, Release, Run)**
  - 빌드, 릴리스, 실행의 3 단계를 엄격하게 분리
- **프로세스 (Processes)**
  - 응용 프로그램을 하나 또는 여러 개의 독립적인 프로세스로 실행

2.3 12-Factor App 방법론

❑ 포트 바인딩 (Port binding)

- 포트 바인딩을 통해 서비스를 공개

❑ 동시성 (Concurrency)

- 프로세스 모델에 따라 수평적 확장

❑ 폐기 용이성 (Disposability)

- 빠른 시작이 가능하며, Graceful Shutdown 시 서비스에 영향을 미치지 않도록 하여 안정성 극대화

  ※ Graceful Shutdown : 정상적인 종료로써 소프트웨어 기능으로 컴퓨터를 끄거나 OS 가 프로세스를 안전하게 종료하고 연결을 해제하는 작업을 수행할 수 있는 경우이다.

❑ 개발/운영 일치 (Dev/prod parity)

- 개발 환경과 운영 환경을 최대한 동일하게 유지
- CI/CD (Continuous Integration/Continuous Delivery)환경이 갖춰져 있어야 함

❑ 로그 (Logs)

- 로그를 이벤트 스트림으로 취급함
- 중앙 집권적인 서비스를 통해 로그 이벤트를 수집하고, 인덱싱하여 분석하는 환경이 가능해야 함

❑ 관리 프로세스 (Admin processes)

- 관리 작업(admin/maintenance)을 일회성 프로세스로 실행

2.4 Service Mesh

- **Configuration Management**
  - 설정변경 시 서비스의 재빌드와 재부팅 없이 즉시 반영
- **Service Discovery**
  - API Gateway 가 서비스를 검색하는 매커니즘
- **Load Balancing**
  - 서비스간 부하 분산
- **API Gateway**
  - API 서버 앞단에서 API 엔드포인트 단일화 및 인증, 인가, 라우팅 기능 담당 Centralized Logging
- **서비스별 로그의 중앙집중화**
  - Centralized Metrics
- **서비스별 메트릭 정보의 중앙집중화**
  - Distributed Tracing

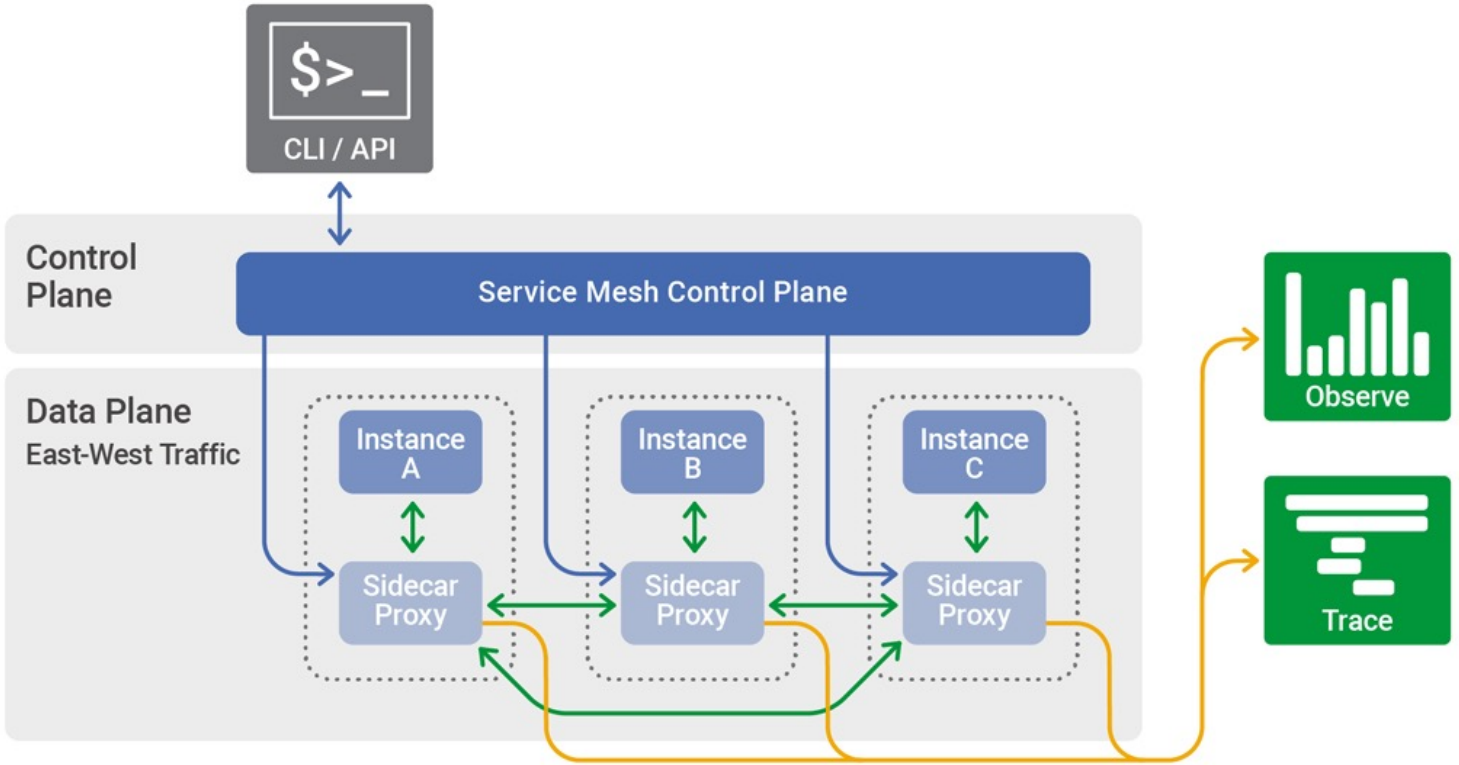
2.4 Service Mesh

- **Distributed Tracing**
  - 서비스간 호출 추적과 성능, 분석 관리
- **Resilience & Fault Tolerance**
  - 서비스간 장애 전파 차단
- **Auto Scaling & Self Healing**
  - 자동 스케일아웃과 복구 자동화
- **Packaging, Deployment & Scheduling**
  - 패키징, 빌드 및 배포 자동화
- **Test Automation**
  - 서비스 테스트 자동화

2.5 Service Mesh 적용 방안

❑ Kubernetes 의 Istio 솔루션을 사용하는 방안

- 마이크로서비스 앞단에 통신 제어를 담당하는 경량화된 프록시를 배치하는 디자인패턴을 가진 Kubernetes 의 Istio를 적용하는 방안
- 각 서비스 인스턴스에 사이드카라고하는 프록시 인스턴스를 제공하여 구현
- 이 사이트카에 Envoy 가 배포되어 애플리케이션을 수정하지 않고도 서비스의 인·아웃 통신 트래픽을 제어



2.5 Service Mesh 적용 방안

❑ Spring Cloud 기반 Service Mesh 직접 구축

- Spring Cloud를 Spring Boot 와 함께 사용하면 분산처리 환경의 안정적인 Service Mesh 를 직접 구현 가능
- Spring Cloud는 애플리케이션 스택의 일부로 모든 MSA 관심사를 해결하도록 잘 통합된 다양한 자바 라이브러리들의 묶음
- 개발자가 분산 시스템의 공통 패턴인 구성관리, Service Discovery, Circuit Breakers, 지능형 라우팅, 프록시, 분산 세션, 클러스트 상태 등을 신속하게 구축할 수 있는 도구를 제공
- 개발자의 PC 나 노트북에서 뿐만 아니라 Cloud Foundry, Kubernetes 같은 클라우드 플랫폼에서도 원활하게 작동

2.5 Service Mesh 적용 방안

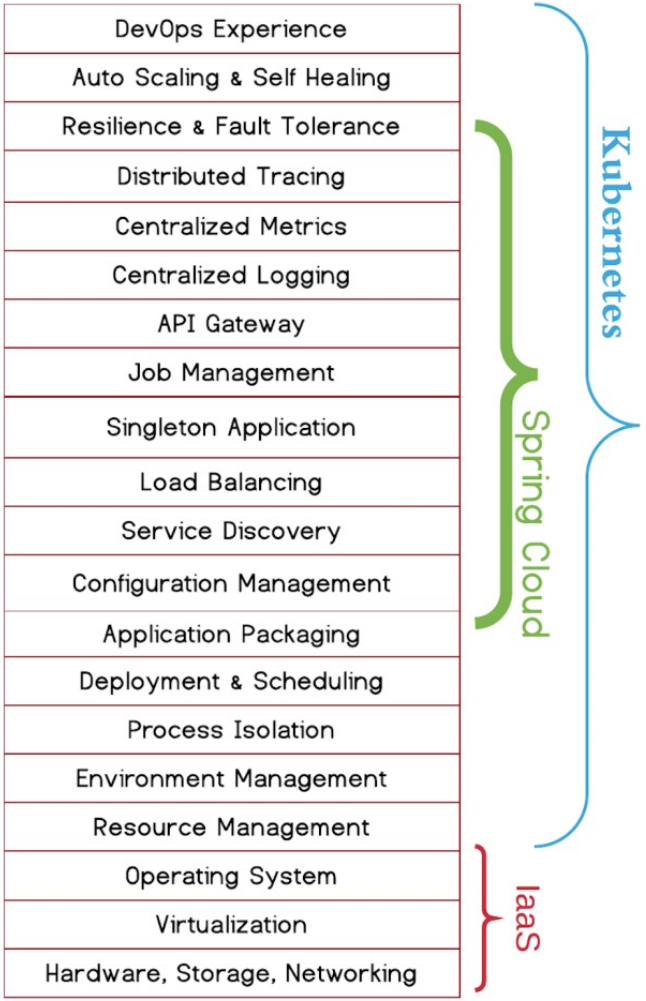
❑ Spring Cloud 와 Kubernetes 의 기술 요소 매핑

MSA 관심사	Spring Cloud	Kubernetes
Configuration Management	Config Server, Consul, Netflix Archaius	Kubernetes ConfigMap & Secrets
Service Discovery	Netflix Eureka, Hashicorp Consul	Kubernetes Service & Ingress Resources
Load Balancing	Netflix Ribbon	Kubernetes Service
API Gateway	Netflix Zuul	Kubernetes Service & Ingress Resources
Service Security	Spring Cloud Security	-
Centralized Logging	ELK Stack (LogStash)	EFK Stack (Fluentd)
Centralized Metrics	Netflix Spectator & Atlas	Heapster, Prometheus, Grafana
Distributed Tracing	Spring Cloud Sleuth, Zipkin	OpenTracing, Zipkin
Resilience & Fault Tolerance	Netflix Hystrix, Turbine & Ribbon	Kubernetes Health Check & resource isolation
Auto Scaling & Self Healing	-	Kubernetes Health Check, Self Healing, Autoscaling
Packaging, Deployment & Scheduling	Spring Boot	Docker/Rkt, Kubernetes Scheduler & Deployment
Job Management	Spring Batch	Kubernetes Jobs & Scheduled Jobs
Singleton Application	Spring Cloud Cluster	Kubernetes Pods

2.5 Service Mesh 적용 방안

❑ 마이크로서비스 필요조건에 따른 범위

– MSA 의 필요조건을 최하위 하드웨어부터 최상위 DevOps 까지를 나열하여 Spring Cloud 과 쿠버네티스 플랫폼이 관여하는 범위



2.5 Service Mesh 적용 방안

❑ Kubernetes 와 결합된 Spring Cloud 의 예

Capability	Spring Cloud 와 쿠버네티스
DevOps	Self service, multi-environment capabilities
Auto Scaling & Self Healing	Pod/Cluster Autoscaler, HealthIndicator, Scheduler
Resilience & Fault Tolerance	HealthIndicator, Hystrix, HealthCheck, Process Check
Distributed Tracing	Zipkin
Centralized Metrics	Kubernetes Metrics Server, Prometheus, Grafana
Centralized Logging	EFK
Job Management	Spring Batch, Schedulled Job
Load Balancing	Ribbon, Service
Service Discovery	Service
Configuration Management	Spring Cloud Kubernetes, ConfigMap, Secret
Service Logic	Spring Framework
Application Packaging	Spring Boot
Deployment & Scheduling	Deployment strategy, A/B, Canary, Scheduler strategy
Process Isolation	Docker, Pods
Environment Management	Namespaces, Authorizations
Resource Management	CPU and memory limits, Namespace resource quotes

3. Spring Cloud 기반 마이크로 서비스 이해

❑ 배경

- 응용프로그램 개발은 Spring 프레임워크의 확산 및 보급에 따라 보다 간단하고 빠른 개발이 가능한 패러다임으로의 변화
- Spring 프레임워크의 기반이 되는 의존성 주입(DI : Dependency Injection)과 관점 지향 프로그래밍(AOP : Aspect Oriented Programming) 방식은 응용 프로그램의 새로운 기준으로 자리 잡음
- Spring 프레임워크 발전함에 따라 관련된 설정의 복잡성을 높아졌으며, 배포(deploy) 측면에서 보면 큰 변화가 없는 상태
- 특히, 응용 프로그램의 복잡성 및 대화형에 대한 근본적인 개발 방식과 배포 방식의 변화가 필요
- 이를 해결하기 위한 방안으로 마이크로 서비스 아키텍처(MSA : Micro Servcie Architecture) 기반의 시스템 개발 및 운영 방안이 제시
- 이 마이크로 서비스 아키텍처는 개별적으로 운영되는 컴포넌트들로 대형 시스템을 구축하고, Spring 은 컴포넌트 레벨에서 Spring 이 항상 수행해 왔던 프로세스 레벨에서 느슨하게 결합된(loosely-coupled) 컴포넌트의 프로세스 구성을 제공

❑ Spring Boot

- Spring Boot 는 표준프레임워크가 제공하는 기본적인 공통기반 영역(Foundation Layer ; Spring Core, Batch Spring Web 등)을 기반 위에 모듈 단위의 배포 및 설정 최소화를 지원하는 실행(execution) 환경의 기반을 제공
- 스프링부트는 단독 실행되는, 실행하기만 하면 되는 상용화 가능한 수준의 스프링 기반 애플리케이션을 쉽게 만들어 낼 수 있음
- 최소한의 설정으로 스프링 플랫폼과 third-party 라이브러리들을 사용 가능

3. Spring Cloud 기반 마이크로 서비스 이해

- 프레임워크 활용 측면에서 보다 손 쉽게 설정을 구성하거나 개발 및 배포를 빠르게 지원
- embedded 방식의 container 를 사용하여 web server 를 통한 배포가 아닌, 독립적으로 실행 가능한 웹 애플리케이션을 구성
- 제공 기능
  - 단독 실행이 가능한 스프링 애플리케이션을 생성
  - 내장형 Tomcat, Jetty 또는 Undertow 를 지원 (WAR 파일로 배포 시 불필요)
  - 기본으로 설정되어 있는 'starter' 컴포넌트들을 쉽게 환경 설정 (dependency, build 등)
  - Library 인식을 통한 자동 환경 구성 지원
  - 상용화 수준의 통계(metrics), 상태 점검(health check) 및 외부 설정 제공 설정을 위한
  - XML 코드 불필요

❑ Spring Boot Starters

- 스타터(Starters)는 응용 프로그램에 포함 할 수 있는 편리한 종속성 관리의 집합
- 샘플 코드와 복사-붙여넣기의 의존성 관리를 거치지 않고도 필요한 모든 Spring 및 관련 기술을 한 번에 관리
  - 예를 들어, 데이터베이스 액세스를 위한 Spring 및 JPA 를 사용하려면 “spring-boot-starter-data-jpa”프로젝트를 종속성에 포함 시켜 사용
- 스타터에는 프로젝트를 신속하게 시작하고 실행하는데 필요한 많은 종속성이 포함되어 있으며 일관되게 지원 관리되는 종속성 세트를 제공
- 스프링 부트 애플리케이션 스타터

이름	설명
spring-boot-starter	자동 구성 지원, 로깅 및 YAML 을 포함한 핵심 스타터
spring-boot-starter-activemq	Apache ActiveMQ 를 사용한 JMS 메시징 스타터
spring-boot-starter-amqp	Spring AMQP 및 Rabbit MQ 사용을 위한 스타터
spring-boot-starter-aop	Spring AOP 및 AspectJ 를 이용한 Aspect 지향 프로그래밍 스타터
spring-boot-starter-artemis	Apache Artemis 를 사용한 JMS 메시징 스타터
spring-boot-starter-batch	스프링 배치 사용을 위한 스타터
spring-boot-starter-cache	Spring Framework 의 캐싱 지원 사용을 위한 스타터
spring-boot-starter-data-cassandra	Cassandra 분산 데이터베이스 및 Spring Data Cassandra 사용을 위한 스타터

- 스프링 부트 애플리케이션 스타터(계속)

이름	설명
spring-boot-starter-data- cassandra-reactive	Cassandra 분산 데이터베이스 및 Spring Data Cassandra Reactive 사용을 위한 스타터
spring-boot-starter-data- couchbase	Couchbase 문서 지향 데이터베이스 및 Spring Data Couchbase 사용을 위한 스타터
spring-boot-starter-data- couchbase-reactive	Couchbase 문서 지향 데이터베이스 및 Spring Data Couchbase Reactive 를 사용하기위한 스타터
spring-boot-starter-data- elasticsearch	Elasticsearch 검색 및 분석 엔진 및 Spring Data Elasticsearch 사용을 위한 스타터
spring-boot-starter-data-jdbc	스프링 데이터 JDBC 사용을 위한 스타터
spring-boot-starter-data-jpa	Hibernate 와 함께 Spring Data JPA 를 사용하기위한 스타터
spring-boot-starter-data-ldap	스프링 데이터 LDAP 사용을 위한 스타터
spring-boot-starter-data-mongodb	MongoDB 문서 지향 데이터베이스 및 Spring Data MongoDB 사용을 위한 스타터
spring-boot-starter-data- mongodb-reactive	MongoDB 문서 지향 데이터베이스 및 Spring Data MongoDB Reactive 사용을 위한 스타터
spring-boot-starter-data-neo4j	Neo4j 그래프 데이터베이스 및 Spring Data Neo4j 사용을 위한 스타터
spring-boot-starter-data-r2dbc	Spring Data R2DBC 사용을 위한 스타터
spring-boot-starter-data-redis	Spring Data Redis 및 Lettuce 클라이언트와 함께 Redis 키-값 데이터 저장소를 사용하기위한 스타터
spring-boot-starter-data-redis-reactive	Spring Data Redis 반응 형 및 Lettuce 클라이언트와 함께 Redis 키-값 데이터 저장소를 사용하기위한 스타터
spring-boot-starter-data-rest	Spring Data REST 를 사용하여 REST 를 통해 Spring Data 저장소를 노출하기위한 스타터

- 스프링 부트 애플리케이션 스타터(계속)

이름	설명
spring-boot-starter-data-solr	Spring Data Solr 과 함께 Apache Solr 검색 플랫폼을 사용하기위한 스타터
spring-boot-starter-freemarker	FreeMarker 보기를 사용하여 MVC 웹 애플리케이션 빌드를 위한 스타터
spring-boot-starter-groovy-templates	Groovy 템플릿 뷰를 사용하여 MVC 웹 애플리케이션 구축을 위한 스타터
spring-boot-starter-hateoas	Spring MVC 및 Spring HATEOAS 로 하이퍼 미디어 기반 RESTful 웹 애플리케이션 구축을 위한 스타터
spring-boot-starter-integration	스프링 통합 사용을 위한 스타터
spring-boot-starter-jdbc	DB 연결 풀에서 JDBC 를 사용하기 위한 스타터
spring-boot-starter-jersey	JAX-RS 및 Jersey를 사용하여 RESTful 웹 애플리케이션을 빌드하기위한 스타터. 대안spring-boot-starter-web
spring-boot-starter-jooq	jOOQ 를 사용하여 SQL 데이터베이스에 액세스하기위한 스타터. spring-boot-starter-data-jpa 또는에 대한 대안 spring-boot-starter-jdbc
spring-boot-starter-json	JSON 을 읽고 쓰는 스타터
spring-boot-starter-jta-atomikos	Atomikos 를 사용한 JTA 트랜잭션 스타터
spring-boot-starter-jta-bitronix	Bitronix 를 사용한 JTA 트랜잭션 스타터. 2.3.0 부터 사용되지 않음
spring-boot-starter-mail	Java Mail 및 Spring Framework 의 이메일 전송 지원을 위한 스타터
spring-boot-starter-mustache	mustache 사용하여 웹 애플리케이션을 빌드하기 위한 스타터
spring-boot-starter-oauth2-client	Spring Security 의 OAuth2 / OpenID Connect 클라이언트 기능을 사용하기 위한 스타터
spring-boot-starter-oauth2-resource-server	Spring Security 의 OAuth2 리소스 서버 기능을 사용하기위한 스타터

- 스프링 부트 애플리케이션 스타터(계속)

이름	설명
spring-boot-starter-quartz Quartz	스케줄러 사용을 위한 스타터
spring-boot-starter-rsocket	Rsocket 클라이언트 및 서버 구축을 위한 스타터
spring-boot-starter-security	스프링 시큐리티 사용을 위한 스타터
spring-boot-starter-test	JUnit, Hamcrest 및 Mockito 를 포함한 라이브러리로 Spring Boot 애플리케이션을 테스트하기위한 스타터
spring-boot-starter-thymeleaf	Thymeleaf 보기를 사용하여 MVC 웹 애플리케이션 빌드를위한 스타터
spring-boot-starter-validation	Hibernate Validator 와 함께 Java Bean Validation 을 사용하기위한 스타터
spring-boot-starter-web	Spring MVC 를 사용하는 RESTful 애플리케이션을 포함한 웹 구축을 위한 스타터. Tomcat 을 기본 내장 컨테이너로 사용
spring-boot-starter-web-services	스프링 웹 서비스 사용을 위한 스타터
spring-boot-starter-webflux	Spring Framework 의 Reactive Web 지원을 사용하여 WebFlux 애플리케이션 구축을 위한 스타터
spring-boot-starter-websocket	Spring Framework 의 WebSocket 지원을 사용하여 WebSocket 애플리케이션 구축을 위한 스타터

- 스프링 부트 프로덕션 스타터

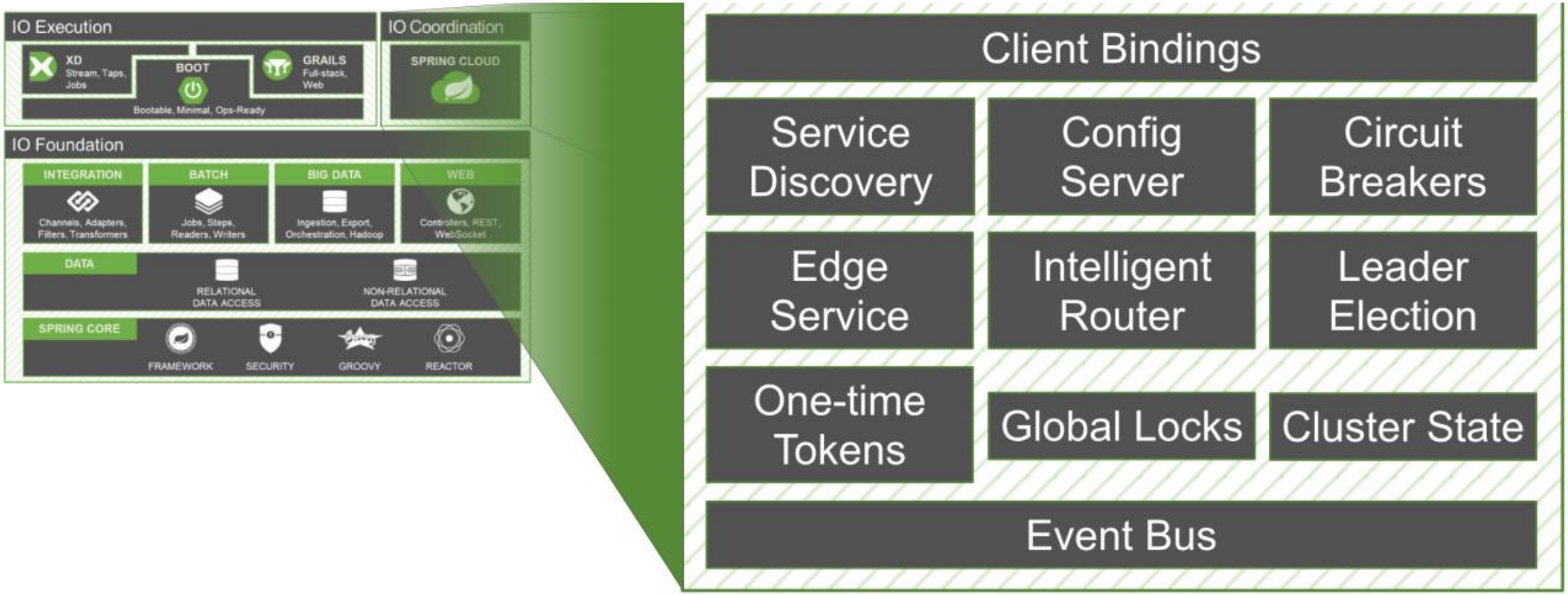
이름	설명
spring-boot-starter-actuator	애플리케이션을 모니터링하고 관리 할 수 있는 프로덕션 준비 기능을 제공하는 Spring Boot Actuator 사용을 위한 스타터

– 스프링 부트 테크니컬 스타터

이름	설명
spring-boot-starter-jetty	내장 서블릿 컨테이너로 Jetty를 사용하기위한 스타터. 대안spring- boot-starter-tomcat
spring-boot-starter-log4j2	로깅을 위해 Log4j2를 사용하기위한 스타터. 대안spring-boot-starter- logging
spring-boot-starter-logging	Logback 을 이용한 로깅 스타터. 기본 로깅 스타터
spring-boot-starter-reactor-netty	임베디드 Reactive HTTP 서버로 Reactor Netty 를 사용하기위한 스타터.
spring-boot-starter-tomcat	임베디드 서블릿 컨테이너로 Tomcat을 사용하기위한 스타터 spring-boot-starter-web에 의해 사용되는 기본 서블릿 컨테이너 스타터
spring-boot-starter-undertow	Undertow를 임베디드 서블릿 컨테이너로 사용하기위한 스타터. 대안spring-boot-starter-tomcat
spring-boot-starter-jetty	내장 서블릿 컨테이너로 Jetty를 사용하기위한 스타터. 대안spring- boot-starter-tomcat
spring-boot-starter-log4j2	로깅을 위해 Log4j2를 사용하기위한 스타터. 대안spring-boot-starter- logging

❑ Spring Cloud

- Spring Cloud 는 개발자가 분산 시스템 구성에 필요한 다양한 기능(설정 관리 및 공유, 서비스 등록 및 관리, 서비스 요청 라우팅 등)을 제공
- 분산 시스템을 구성하기 위한 복잡한 설정 및 서비스들을 효율적이고 신속하게 구현할 수 있도록 지원
- 개발된 서비스는 작은 노트북, Bare metal 데이터 센터 및 Cloud Foundry 와 같은 클라우드 플랫폼을 포함한 모든 분산 환경에서 동작을 보장



3. Spring Cloud 기반 마이크로 서비스 이해

- 제공 기능
  - 분산 및 버전으로 구분된 설정 관리 서비스 등록 및 조회
  - 라우팅 및 상태 체크
  - 서비스 대 서비스 호출
  - 서비스 분산 로딩(Load balancing) 서비스 간 호출 분리(Circuit Breaker) 클러스터링 환경 관리
  - 분산 메시징

– Spring Cloud 컴포넌트

서비스	설명	컴포넌트
Config 서비스	별도의 통합된 설정 관리 서비스 제공을 통해 환경 독립적 서비스 제공	Spring Config
Service Discovery 서비스	서비스에 대한 물리적 위치 정보 대신 논리적 서비스 위치 정보 제공	Eureka (Spring Cloud Netflix)
Event Bus 서비스	분산 메세징 지원을 위한 서비스 연계 지원	Spring Cloud Bus (AMQP & RabbitMQ)
Circuit Breaker 서비스	서비스 간 호출 시, 문제가 있는 서비스에 대한 차단 지원 서비스	Hystrix (Spring Cloud Netflix)
Client Load Balancing	서비스 호출 시에 분산 형태로 호출 할 수 있는 client 적용 서비스 library	Ribbon (Spring Cloud Netflix)
Service Router 서비스	서비스 호출 시, routing 을 통해 실제 서비스에 위치 제공	Zuul (Spring Cloud Netflix)

– Spring Cloud 컴포넌트(계속)

서비스	설명	컴포넌트
API Gateway 서비스	Microservice 에 대한 API 관리 및 모니터링 서비스	Zuul (Spring Cloud Netflix)
Cluster 서비스	Service level 의 Cluster 를 지원하기 위한 서비스 제공	Spring Cloud Cluster
Security 서비스	Load balanced 환경에서의 OAuth2 인증 지원 서비스	Spring Cloud Security
Polyglot 지원 서비스	non-JVM 프로그래밍 언어 지원을 위한 서비스	Spring Cloud Sidecar
Kubernetes 지원 서비스	Spring Cloud 애플리케이션을 위한 쿠버네티스 Discovery 와 ConfigMaps 지원 서비스	Spring Cloud Kubernetes
API Gateway 서비스	Microservice 에 대한 API 관리 및 모니터링 서비스	Zuul (Spring Cloud Netflix)

❑ Spring 프로젝트 적용을 통하여 효율적인 마이크로 서비스 개발 및 운영을 지원

- Spring Boot : 컴포넌트 레벨에서 마이크로 서비스 아키텍처로 설정 간소화 및 독립 서비스를 지원 (Embedded Web Server, Stand-alone application 등의 Inner Architecture 영역 지원)
- Spring Cloud : 시스템 레벨에서의 마이크로 서비스 아키텍처로 컴포넌트들 간의 효율적인 분산 서비스를 지원 (Load Balancing, Service Discovery 등의 Outer Architecture 영역 지원)

❑ Spring Boot 을 활용한 MSA 애플리케이션 Demo

	화면 서비스	커스터머 서비스
서비스	Catalogs	Customers
Request	/catalogs/{customerId}	/customers/customerId
Response	String Type (JSON)	String Type (JSON)

4.1 Spring Cloud 의 컴포넌트 활용

❑ Circuit Breaker - Hystrix

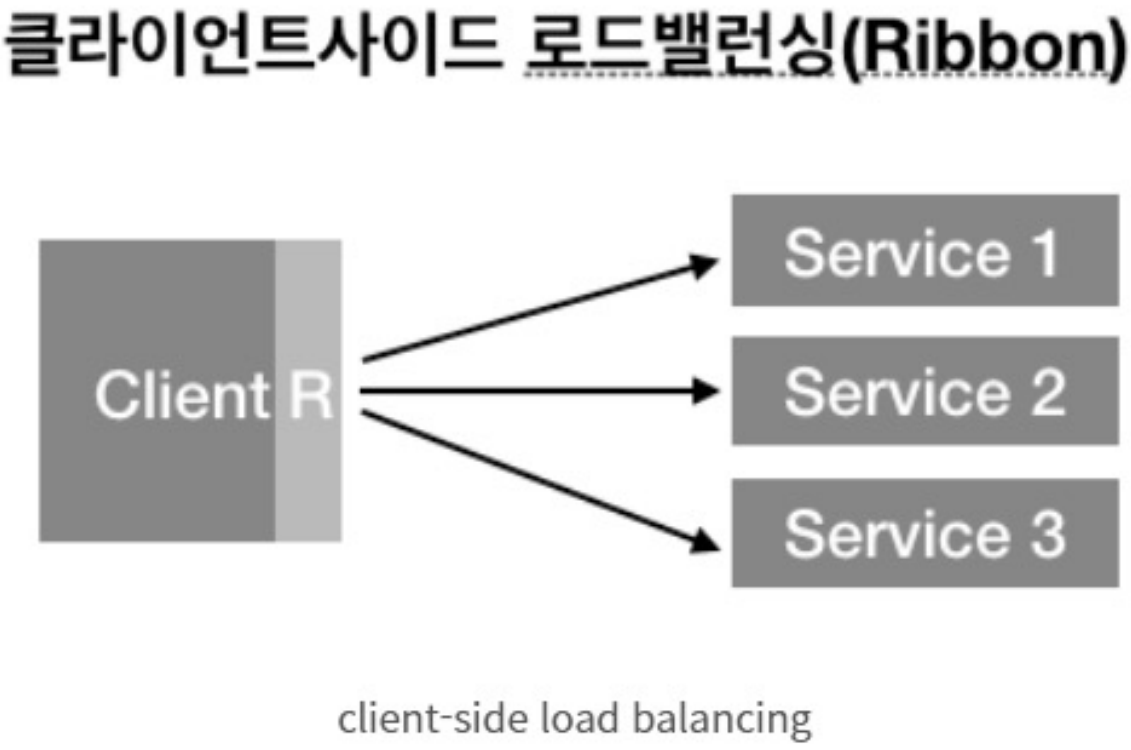
- Hystrix 는 분산환경을 위한 장애 및 지연 내성(Latency and Fault Tolerance)을 갖도록 도와주는 라이브러리
- Circuit Breaker Pattern 디자인을 적용하여 MSA 애플리케이션의 장애 전파를 방지

❑ Hystrix 의 역할

- 장애 및 지연 내성 (Latency and Fault Tolerance)
  - 분산환경에서 한 개의 서비스가 실패하는 경우 해당 서비스의 실패로 의존성이 있는 타 서비스까지 장애가 전파
  - 외부에서 호출하는 서비스를 Hystrix 로 Wrapping 하면 실패가 전파되는 것을 Fallbacks 를 활용하여 미연에 방지하고 빠르게 복구
  - 각 서비스의 HystrixCommand 는 Circuit Breaker Pattern 으로 외부에 영향을 받지 않도록 쓰레드와 세마포어 방식으로 분리(Isolation)
- 실시간 구동 모니터링 (Realtime Operations)
  - Hystrix 를 사용하면 실시간 모니터링과 설정 변경을 지원
  - 서비스와 설정값의 변경이 어떻게 시스템에 적용되는지 대쉬보드를 통해서 확인 가능
  - (Netflex/Servo 를 활용한 metrix 서비스를 사용할 수 있으며, third-party 라이브러리로 Prometheus 를 통하여 모니터링)
- 병행성 (Concurrency)
  - Parallel execution 을 제공하며 여러 설정 값에 대한 변경을 지원
  - 내부적으로 중복되는 Request 처리를 줄이기 위하여 Request Caching 과 일관 처리를 위한 Request Collapsing 기능을 제공

❑ Client Load Balancer – Ribbon

- Ribbon 은 Client 에 탑재할 수 있는 소프트웨어 기반의 Load Balancer
- 일반적으로 사용하는 하드웨어적인 L4 Switch 를 사용하지만, MSA 에서는 소프트웨어적으로 구현된 클라이어트사이드 로드벨런싱으로 주로 사용
- Ribbon 은 분산 처리 방법으로 여러 서버를 라운드 로빈 방식으로 부하 분산 기능을 제공



❑ Ribbon 의 구성요소(Rule, Ping, ServerList)

- Rule : 요청을 보낼 서버를 선택하는 논리
  - Round Robbin : 한 서버씩 돌아가며 전달 (기본 설정)
  - Available Filtering : 에러가 많은 서버를 제외
  - Weighted Response Time : 서버별 응답 시간에 따라 확률 조절
- Ping : 서버가 살아 있는지 체크하는 논리
  - Static, dynamic 모두 가능
- ServerList : 로드 밸런싱 대상 서버 목록
  - Configuration 을 통해 static 하게 설정 가능
  - Eureka 등을 기반으로 dynamic 하게 설정 가능

4.1 Spring Cloud 의 컴포넌트 활용

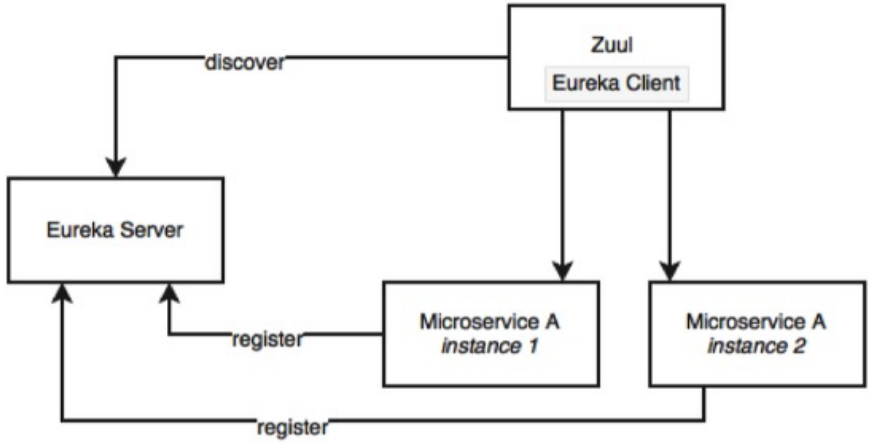
❑ Service Registry - Eureka

- Eureka는 MSA의 장점 중 하나인 동적인 서비스 증설 및 축소를 위하여 필수적으로 필요한 서비스의 자가 등록, 탐색 및 부하 분산에 사용될 수 있는 라이브러리
- 마이크로 서비스들의 정보를 레지스트리 서버에 등록할 수 있도록 기능을 제공
- Eureka 는 Eureka 서버와 클라이언트로 구성
  - Eureka 서버는 Eureka 클라이언트에 해당하는 마이크로 서비스들의 상태 정보가 등록되어 있는 레지스트리 서버
  - Eureka 클라이언트는 서비스가 시작 될 때 Eureka 서버에 자신의 정보를 등록하고 이후 주기적으로 자신의 가용 상태(health check)를 알리며, 일정 횟수 이상의 ping 이 확인되지 않으면 Eureka 서버에서 해당 서비스를 제외
  - Eureka 는 Ribbon 과 결합하여 사용할 수 있으며 이를 통해 서버 목록을 자동으로 관리 및 갱신

4.1 Spring Cloud 의 컴포넌트 활용

❑ API Gateway - Zuul

- API Gataway 란 MSA 에서 언급되는 주요 컴포넌트 중 하나이며, 모든 클라이언트 요청에 대한 end-point 를 통합하는 서비스
- 마치 프록시 서버처럼 동작하며, 인증 및 권한, 모니터링, logging 등의 추가적인 기능도 지원
- 기존의 모놀리틱 아키텍처와 같이 모든 비즈니스 로직이 하나의 서버에 존재하는 것 달리 MSA 는 도메인 별 하나 이상의 서비스(서버)가 존재하며, 또한, 한 서비스가 한 개 이상의 서버가 존재할 수 있으므로 사용자(클라이언트) 입장에서는 다수의 end-point 를 알아야 힘
- end- point 가 변경될 경우는 관리하기가 힘들어진다. 이러한 문제를 해결하기 위하여 하나로 통합할 수 있는 API Gateway 가 필요
- Zuul 서버는 단일 end-point 역할을 하며, 이를 위하여 Eureka, Hystrix, Ribbon 등의 여러 기능을 내장



4.1 Spring Cloud 의 컴포넌트 활용

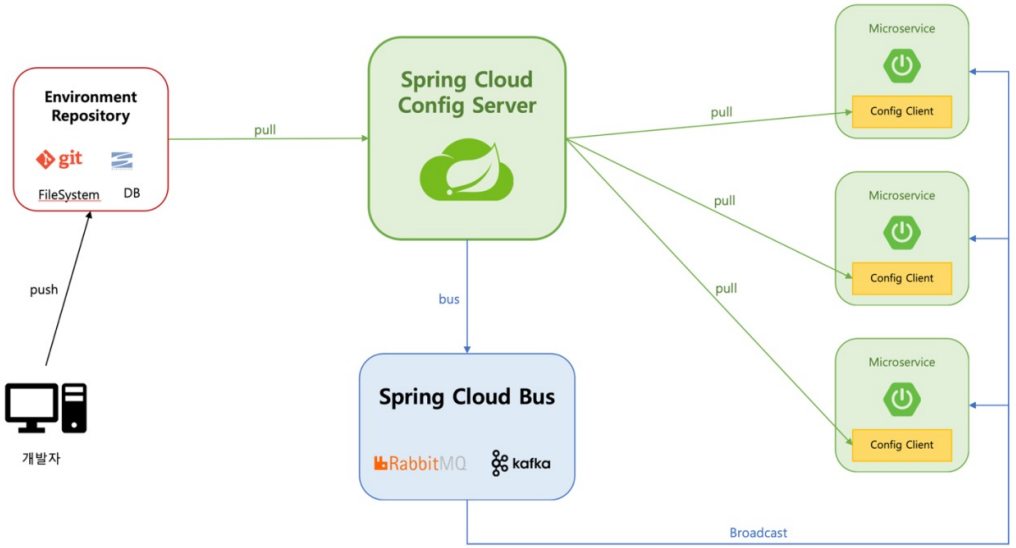
- Zuul 의 주요 기능
  - 인증 및 보안 (Authentication and Security) : 클라이언트 요청 시, 각 리소스에 대한 인증 요구 사항을 식별하고 이를 충족하지 못한 경우 해당 요청을 거부 기능을 제공
  - 모니터링 (Insights and Monitoring): 모든 트래픽이 지나기 때문에 의미있는 데이터와 지표를 수집할 수 있는 기능을 제공
  - 동적 라우팅(Dynamic Routing) : 필요에 따라 즉시 원하는 백엔드 클러스터로 트래픽을 동적으로 라우팅하는 기능을 제공
  - 부하테스트: 성능 측정을 위하여 신규서비스 또는 백엔드에 트래픽을 점진적으로 증가하는 방식으로 부하를 유발할 수 있는 기능을 제공
  - 트래픽 드랍(Load Shedding) : 각 요청에 대해 용량을 할당하고, 제한된 이상의 요청이 발생한 경우 이를 제한할 수 있는 기능을 제공
  - 정적 응답 처리 (Static Response Handling) : 특정 요청에 대해서는 백엔드로 트래픽을 보내는 대신 즉시 API Gateway 에서 응답을 처리하는 기능을 제공

❑ Config 서버

- Config 서버는 분산시스템에서 애플리케이션의 환경설정정보 특히 서비스, 비즈니스 로직과 연관성 있는 정보들을 애플리케이션과 분리해 외부에서 관리하도록 한 환경설정 서버
- 이를 사용하면 수많은 애플리케이션들의 환경설정 속성정보를 중앙에서 관리 가능
- 환경설정 속성정보란 DB 접속 정보나 미들웨어(연계서버) 접속정보, 애플리케이션을 구성하는 각종 메타데이터들을 지칭
- 이런 속성값들은 보통 application.properties 또는 application.yml 파일에 기록하여 사용
- 이러한 정보들을 마이크로서비스들이 제각각 따로 관리하게 될 경우 환경설정값이 변경되게 되면 전체 마이크로서비스들을 다시 빌드해야 하는데, 마이크로서비스의 개수가 많으면 많을수록 변경사항을 적용하는데 많은 비용이 발생
- 이러한 문제를 해결하기 위해 Spring Cloud Config 컴포넌트가 제공
- Config 서버를 구축하여 사용하면 마이크로서비스들의 환경설정파일을 중앙 서버 한 곳에 관리 가능

❑ Config 서버 구성도

- Config 서버가 구동될 때 Environment Repository 에서 설정 내용을 가져옴
- Environment Repository 는 VCS, File System, DB 로 구성 가능(보통 Git 이나 SVN 같은 VCS 를 많이 사용)
- 각 마이크로 서비스들은 서비스가 구동될 때 Config 서버에서 설정 내용을 내려 받아 애플리케이션이 초기화되고 구동
- 만약 서비스 운영 중에 설정파일을 변경 해야 할 경우에는 Spring Cloud Bus 를 이용하여 모든 마이크로 서비스의 환경설정을 업데이트 가능
- Spring Cloud Bus 는 각 서비스들간의 실시간 메시지 통신 역할을 하는 RabbitMQ 또는 Kafka 같은 경량 메시지 브로커들을 사용할 수 있도록 해주며, 이 메시지 브로커들을 이용하여 각 서비스 들에게 Config 변경사항을 전파



참고 문헌

❑ **표준프레임워크 MSA 적용 개발 가이드**

- https://www.egovframe.go.kr/home/ntt/nttRead.do?pagerOffset=0&searchKey=&searchValue=&menuNo=76&bbsId=171&nttId=1809

❑ **Cloud를 위한 표준프레임워크 발전방향 연구보고서**

감사합니다