NIA 한국지능정보사회진흥원

클라우드 네이티브 기반 행정·공공 서비스 확산 지원

클라우드 네이티브 온라인 설명회

개발자 가이드 (개발 안내서)

발표 목차

클라우드 네이티브 기반 행정·공공 서비스 확산 지원

클라우드 네이티브 발주자 가이드

클라우드 네이티브는 어떻게 발전하고 있는가?

NIA 한국지능정보사회진흥원

클라우드 네이티브 정의

클라우드 네이티브란?

클라우드 네이티브
(형용사/명사)

클라우드 컴퓨팅의 장점을 최대한 활용할 수 있는
(효율적인 자원이용, 탄력적 수요 대응 등)
정보시스템 분석·설계·구현 및 **실행하는 환경**

클라우드 네이티브
애플리케이션

클라우드 환경에서 실행되는 **애플리케이션**

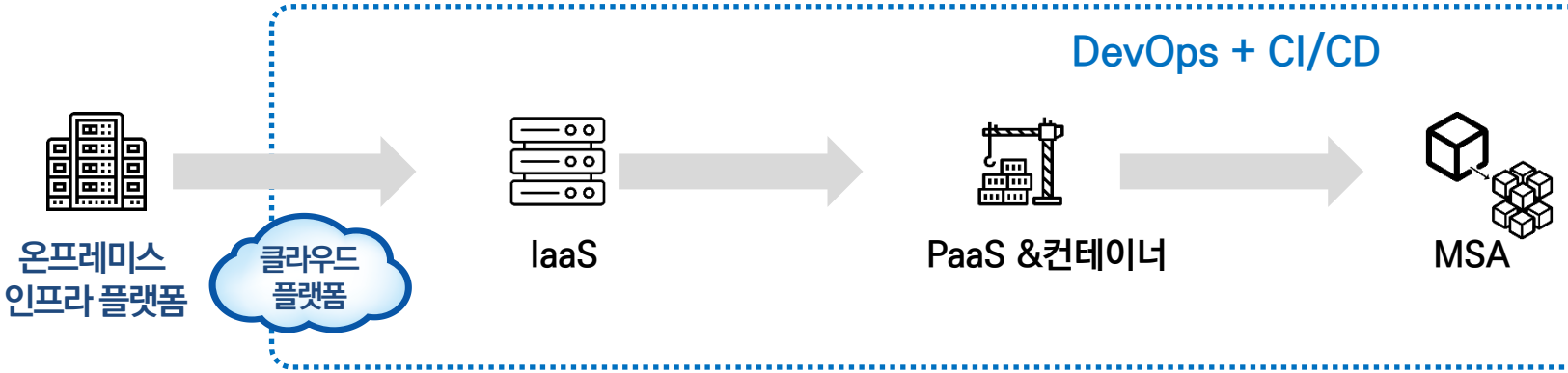
CNCF (Cloud Native Computing Foundation) v1.0
클라우드 네이티브 전환할 수 있는 기술 정의 및 오픈 소스를 관리하는 단체

- 퍼블릭, 프라이빗, 하이브리드 **클라우드 환경에서** 확장성 있는 **애플리케이션**
- 컨테이너, 서비스 메시(Mesh), **마이크로서비스(Micro Service) 인프라 구조**, 선언적 API로 접근
- 자동화, 회복성, 편리성, 가시성을 갖는 **느슨하게 결합된 시스템** (개발 및 실행 환경)
- 엔지니어는 최소한의 수고로, 영향력이 크고, 예측 가능한 변경을 할 수 있는 기술 정의

클라우드 네이티브 변화

클라우드 네이티브는 어떻게 발전하고 있는가?

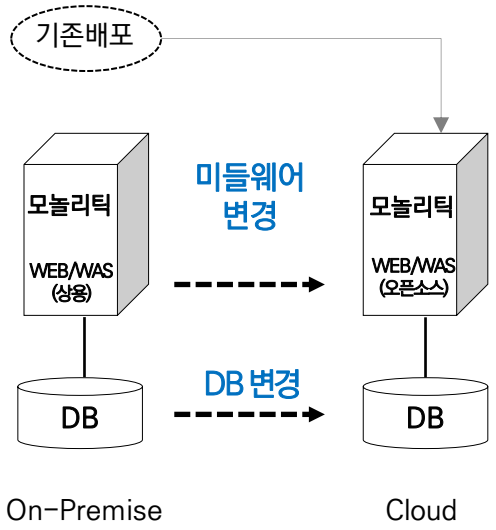
클라우드 네이티브 성숙도 단계 – 애플리케이션 관점



클라우드 네이티브 성숙도 단계 – 인프라 관점

Level 1: Cloud Ready
클라우드 준비 단계

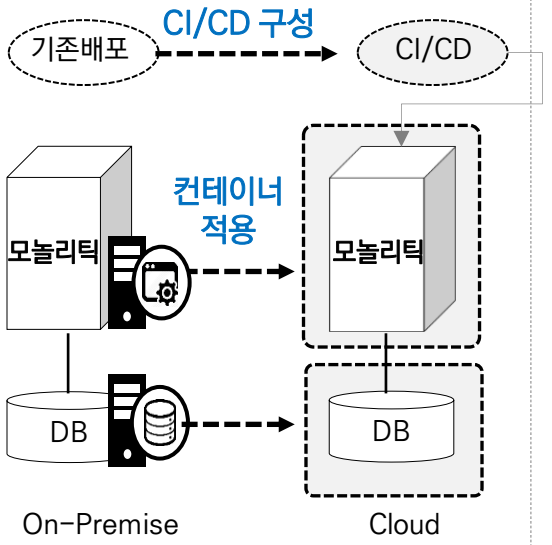
플랫폼 표준화, 비용효율



PaaS
(서버가상화–VM, x86)

Level 2 : Cloud Friendly
클라우드 친화단계

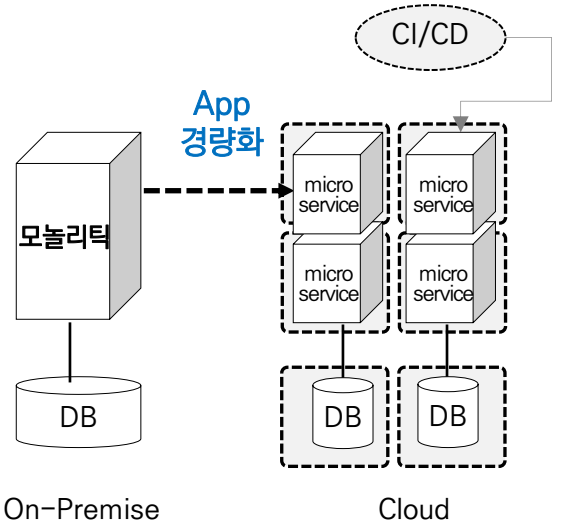
Auto-Scaling, 컨테이너 적용



PaaS + CI/CD + 12 Factors
(서버가상화–컨테이너, x86)

Level 3 : Cloud Native
클라우드 네이티브 단계

APP경량화, 업무 민첩성 대응



PaaS + CI/CD + 12 Factors + MSA
(서버가상화–컨테이너, x86)

클라우드 네이티브 개발 절차

클라우드 네이티브 애플리케이션 개발은 마이크로서비스 도출, 도메인 모델링, 아키텍처 설계 등을 포함한 분석, 설계, 개발, 테스트, 이행단계의 절차로 수행됨

분석	설계	개발	테스트	이행/운영
요구사항 분석	도메인 모델링	서비스 개발/단위테스트	통합 테스트	CI/CD (배포)
마이크로서비스 도출	사용자 화면 (UI) 설계	사용자 화면 (UI) 개발 / 단위테스트	시스템 테스트	데이터 이행
사용자 화면 (UI)분석	데이터 이행 설계	데이터 이행 개발 / 단위테스트	데이터이행 테스트	컷오버
데이터 현황 분석	개발환경 구성 (개발표준 정의)	테스트/ 운영환경 구성	인수 테스트	운영
클라우드 네이티브 아키텍처 설계		* 12 Factors		

클라우드 네이티브 관련 개발 공정영역을 식별함

클라우드 네이티브 기반 행정·공공 서비스 확산 지원
클라우드 네이티브 온라인 설명회

클라우드 네이티브 애플리케이션의 분석방법은 어떻게 되는가?

NIA 한국지능정보사회진흥원

마이크로 서비스 도출

분석단계 마이크로서비스를 도출 방식은 도메인 주도 설계(DDD) 방식과 프로세스 모델링 방식이 존재하며, 기관의 특성을 고려하여 적합한 방식을 적용할 수 있음

※ 결합/응집도, 독립/자율성, 단일책임 등을 고려하여 분리하거나 병합하는 과정을 반복하며 마이크로서비스를 식별

도메인 주도 설계

모델링과 개발과의 불일치 문제를 해결하기 위해 필요하며, 도메인 모델은 자동화 된 비즈니스나 현실세계의 문제(=도메인)를 개념적으로 표현하는 기법

소프트웨어의 본질은 해당 소프트웨어 사용자를 위해
도메인에 관련된 문제를 해결하는 것임

기존의 개발

그러나 …

- 최신 기술에 치중
- 도메인에 대한 지식과 문제에 대한 이해 부족
- 개념의 분할 없이 계속해서 필요한 기능을 핵심 개념에 추가함
- 데이터에 종속적인 애플리케이션
- 모델링과 개발과의 불일치 발생
  - 분석모델-구현모델, 구현모델-코드 간 연계 부족

도메인 주도 설계 필요성

도메인 주도 설계의 필요성

- 모델링과 개발의 불일치 해결
- 공통의 언어(유비쿼터스 언어)의 사용을 통해 도메인과 구현을 충분히 만족하는 모델을 만듦
- 유비쿼터스 언어를 사용하는 용어사전 작성
- '설계'와 '구현'은 계속적인 수정 과정 반복

도메인 모델

- 도메인의 가치를 최우선시하는 모델링 기법
- 핵심 원칙은 핵심 원칙은 모델 기반의 언어(유비쿼터스 언어)를 사용하는 것이 매우 중요
  - 도메인 모델의 사용시 사용자, 도메인 전문가, 분석가, 개발자 등이 동일한 모습으로 도메인을 이해하고 도메인 지식을 공유하는데 도움이 됨
- 모델은 모든 프로젝트 참여자가 이해할 수 있어야 하며, 실제 구현이 가능해야 함

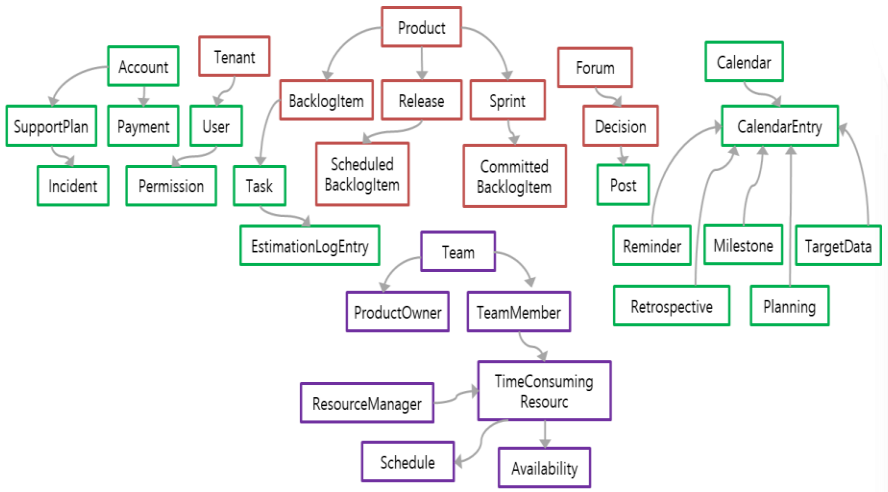
개발자들이 도메인 전문가, 분석가 등과 함께 회의에 참여하고,
도메인과 모델을 정확하게 이해해야 함

※ 유비쿼터스 언어 : 소프트웨어 개발팀에서의 공통의 언어

도메인 주도 설계 (바운디드 컨텍스트 서브도메인의 도출)

전체 비즈니스 도메인을 여러 개의 서브 도메인으로 나눈 후, 서브 도메인 내 동일한 맥락을 경계로 구분하여 바운디드 컨텍스트를 도출함

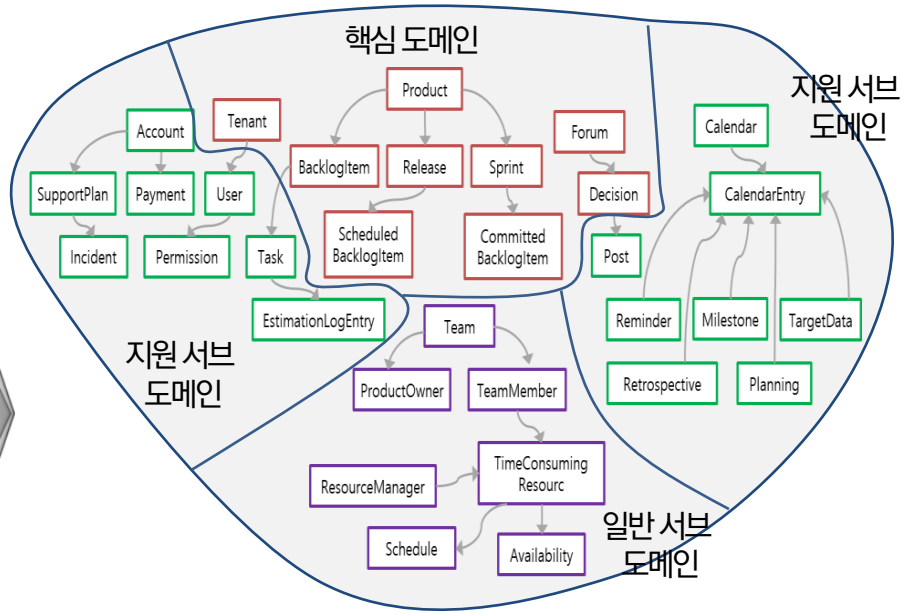
설계 과정에 발생할 수 있는 문제



- 처음에는 핵심이 되는 부분에 대한 설계에서 시작하여 거대한 덩어리의 모델이 만들어짐
- 모델 내에 많은 개념, 다양한 언어 등이 존재함
- 기능 추가 또는 유지보수가 어려움

출처 : 도메인주도설계핵심, 박현철/전장호, 에이콘, 2017

전체 비즈니스 도메인의 하위영역인 서브 도메인 도출



- 비즈니스 상의 문제 해결을 위해 큰 도메인을 여러 개의 서브 도메인으로 구분
- 한 개의 서브 도메인에는 한 개 이상의 바운디드 컨텍스트가 존재함
  - 바운디드 컨텍스트 : 동일한 맥락을 경계로 구분한 것

도메인 주도 설계 (서브도메인과 바운디드 컨텍스트)

서브 도메인은 핵심, 지원, 일반으로 구분되며 서브도메인을 토대로 바운디드 컨텍스트를 정의하고, 바운디드 컨텍스트는 다시 여러 개의 마이크로 서비스로 정의될 수 있음

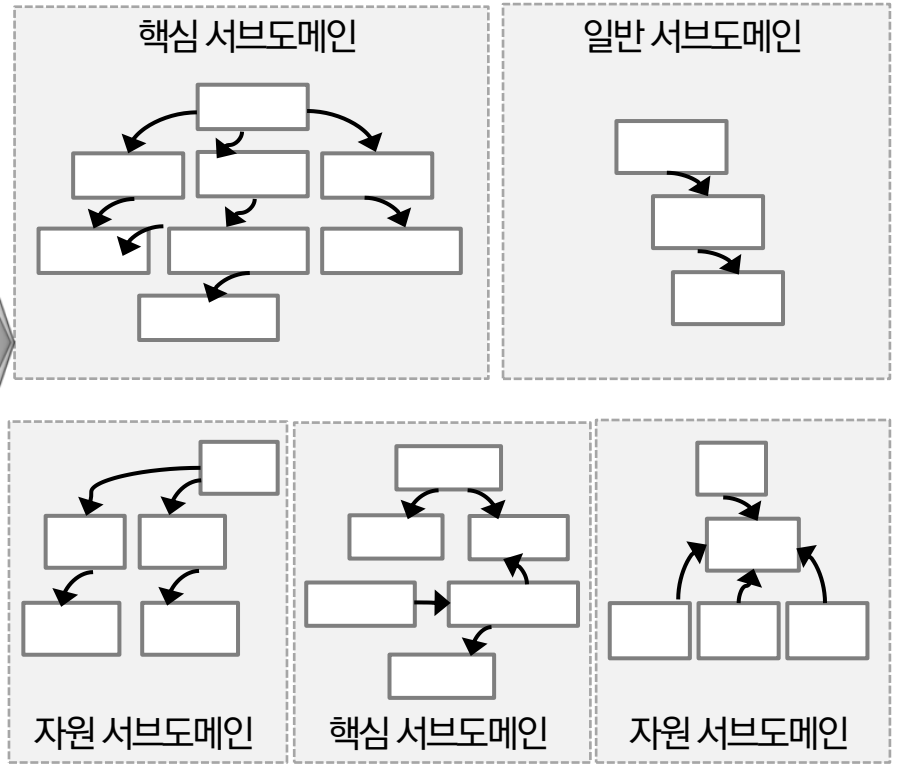
서브도메인 : 바운디드 컨텍스트 = 1 : 1

핵심 서브도메인
- 다른 경쟁자와 차별화를 만들 수 있는 비즈니스 영역
- 높은 우선순위를 갖는 전략적 투자 영역

지원 서브도메인
- 맞춤 개발이 필요한 영역
- 핵심 서브도메인의 성공을 위한 중요한 영역

일반 서브도메인
- 기존 제품 구매를 통해 바로 충족시킬 수 있는 영역
- 핵심/지원 서브도메인이 할당된 팀에서 직접 구현 가능

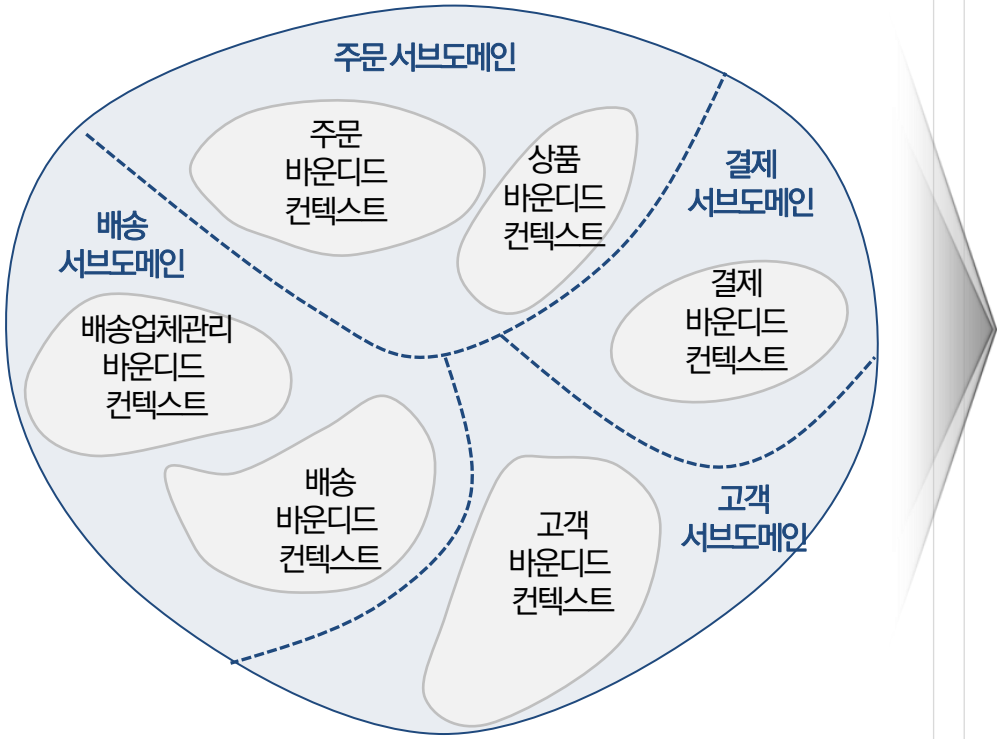
바운디드 컨텍스트 : 마이크로서비스= 1 : N



도메인 주도 설계 (컨텍스트 맵을 통한 마이크로서비스 도출)

전략적 설계는 비즈니스 상 중요한 것을 찾아 중요도에 따라 일을 나누는 방법으로서 컨텍스트 맵을 통한 마이크로서비스 도출

컨텍스트 맵 예시



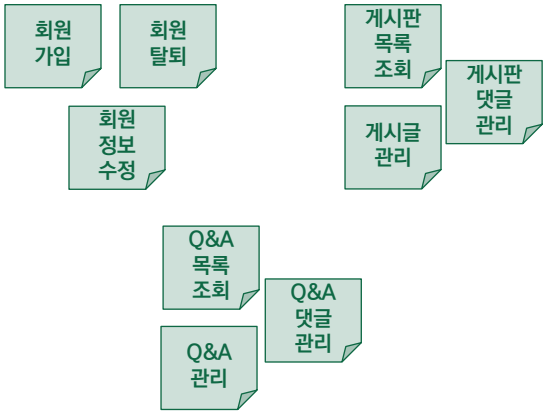
서비스 도출 예시

도메인 주도 설계 (이벤트 스토밍)

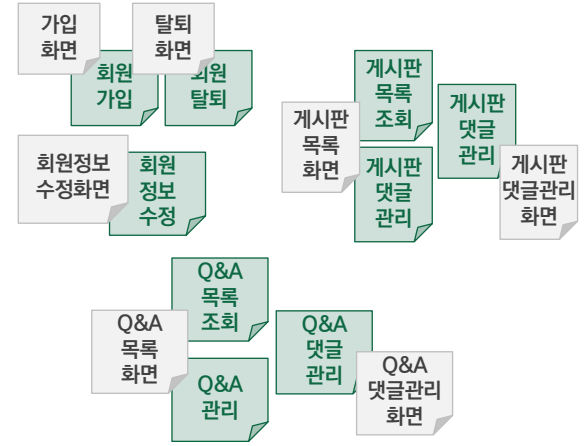
이벤트스토밍은 브레인스토밍을 통한 이벤트 도출, 커맨드 도출, 애그리게잇 식별, 바운디드 컨텍스트 식별의 단계를 거쳐 마이크로서비스를 도출함

마이크로 서비스 식별을 위한 이벤트 스토밍 결과 예시

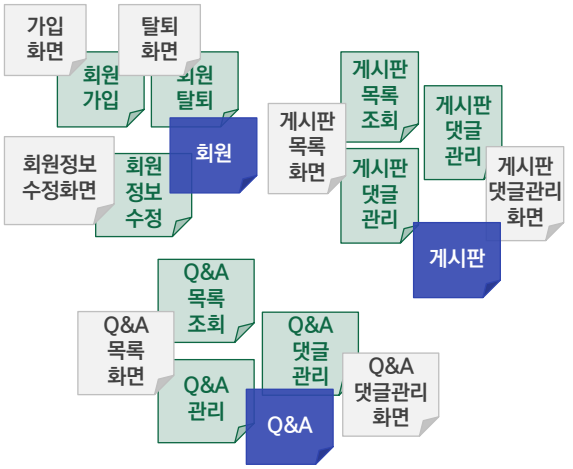
1 이벤트 도출



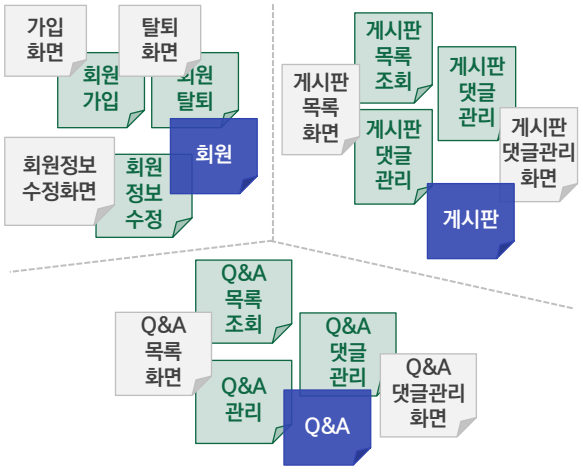
2 커멘드 도출



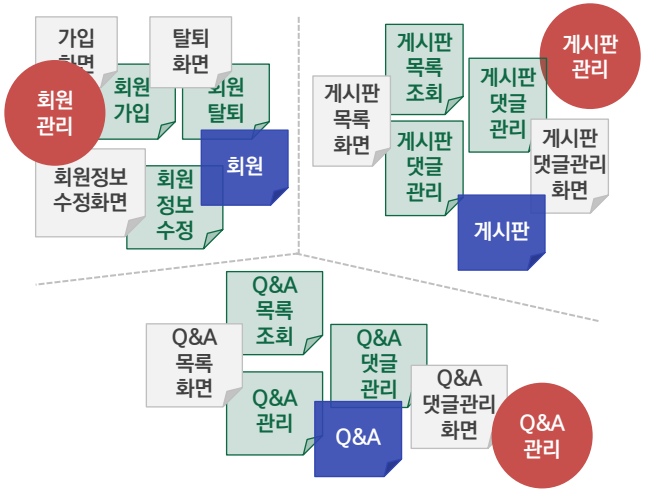
3 엔티티(에그리게잇) 도출



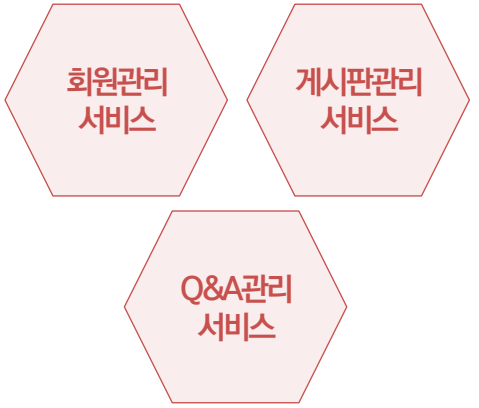
4 컨텍스트 경계 그리기



5 마이크로 서비스 식별 (컨테스트 매핑)



6 마이크로 서비스 식별 결과



업무기능 분해

기존 시스템이 존재하는 경우, 기관이 보유하고 있는 비즈니스 프로세스 또는 애플리케이션 맵과 기능체계도 등을 참고하여 업무기능을 분해하여 마이크로서비스를 도출할 수 있음

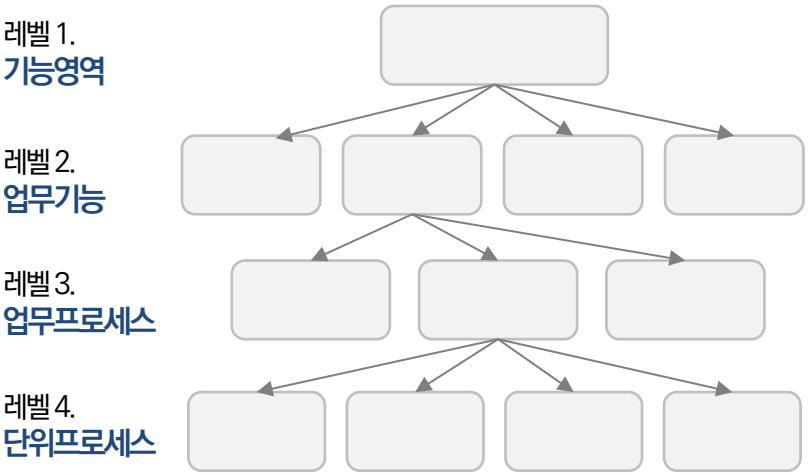
업무기능 분해

- 비즈니스 업무를 분석하여 더 작은 업무로 나누고 그들 간의 계층구조 및 업무간의 순서와 의존성을 분석하는 작업
- 정보시스템을 구축하는 가장 최하위 단위의 프로세스를 도출함

활용 자료

- 기관이 보유하고 있는 비즈니스 프로세스 또는 애플리케이션의 맵 및 체계도 활용할 수 있음

업무기능 분해도 개념



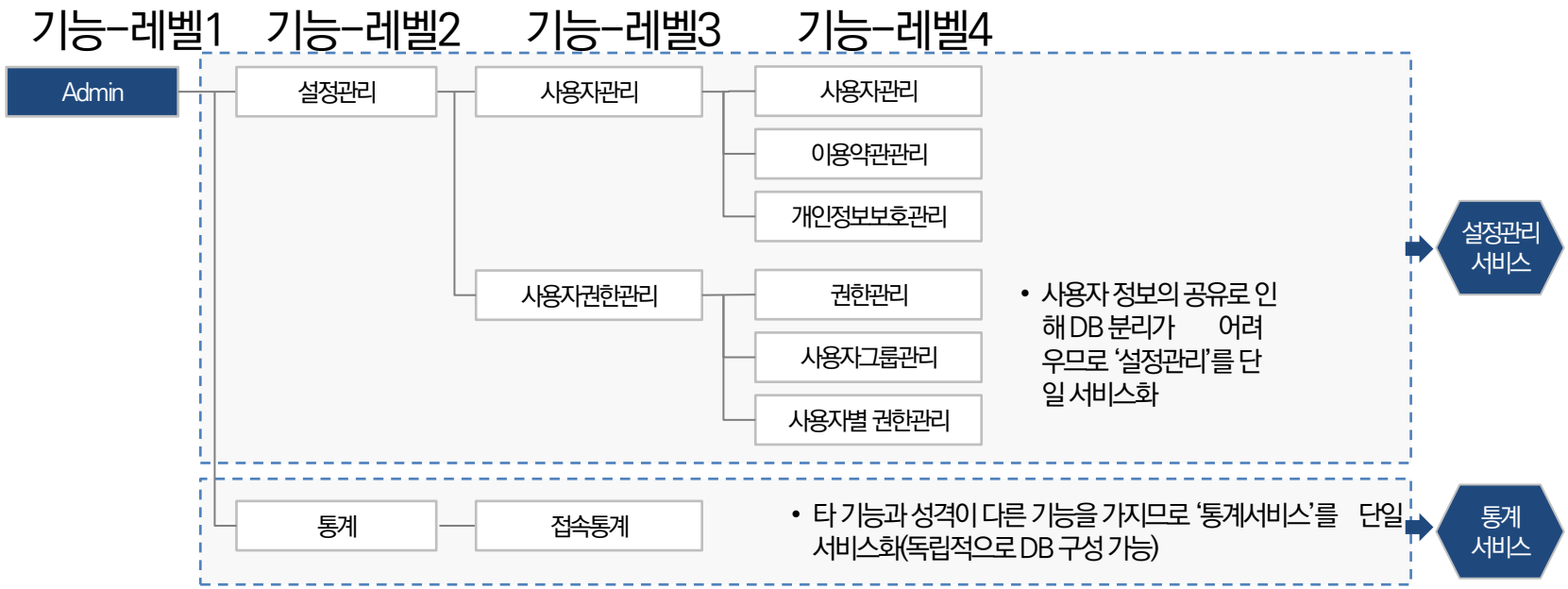
업무기능 분해도

업무기능 분해 (마이크로서비스 식별)

마이크로서비스는 애플리케이션의 기능체계를 토대로 기능간 응집도와 결합도, 독립적 개발 및 배포, 데이터 분리 가능성 등을 고려하여 도출함

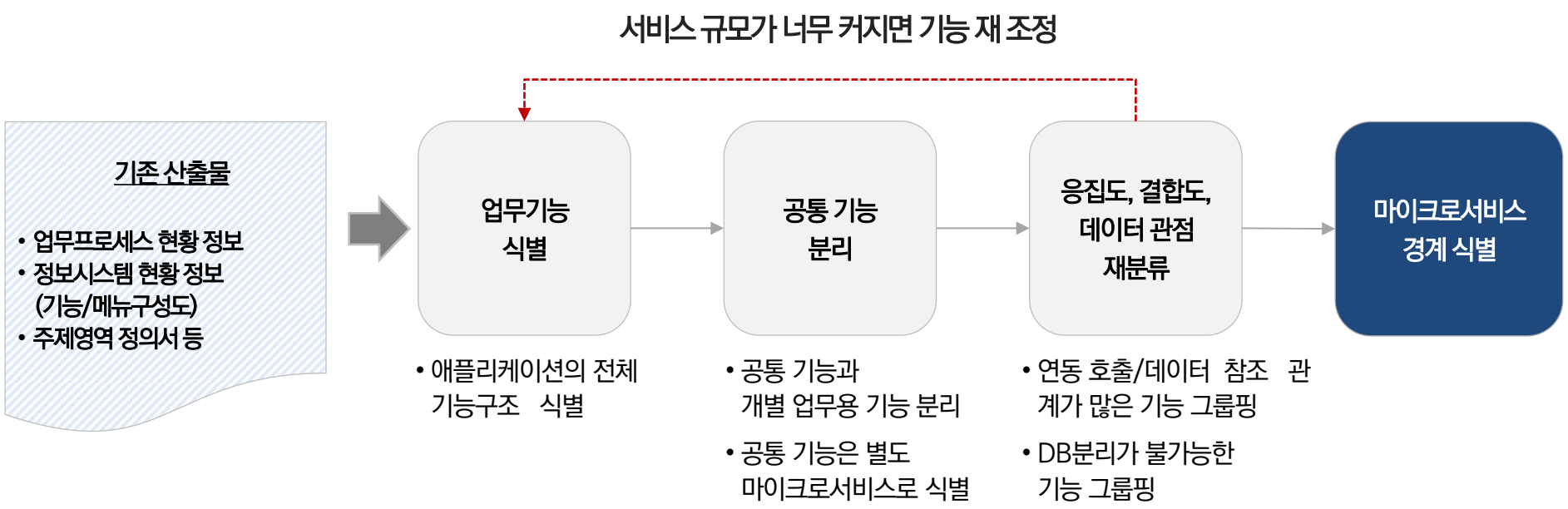
식별 원칙

- 애플리케이션의 하위 응용기능(Level11)) 단위를 마이크로서비스 단위로 식별
  - 결합도에 따라 1개 혹은 복수개의 기능으로 마이크로서비스를 구성할 수 있음
- 마이크로서비스는 독립적인 하나의 기능을 가지며, 높은 응집도와 타 서비스와는 낮은 결합도를 갖도록 구성해야 함
  - 구현가능한 공유 메소드는 마이크로서비스 내에 직접 구현
  - 공통 컴포넌트는 별도의 서비스로 구현
- 독립적으로 개발, 배포, 및 데이터(DB) 분리가 가능한 단위



업무기능 분해 (마이크로서비스 식별)

업무프로세스 및 정보시스템 현황 관련 자료 등을 토대로 업무기능 식별, 공통 기능 분리, 응집도 및 결합도를 고려한 데이터 관점의 재분류 과정을 통해 마이크로서비스 경계를 도출함



고려사항

- 식별된 마이크로서비스 단위로 DB의 물리적/논리적으로 분리가 가능해야 함
- 동일한 데이터(테이블)의 CRUD성 거래가 서로 다른 서비스로 분리되지 않도록 해야 함
- 마이크로서비스 크기는 업무의 특성에 따라 달라 명확한 기준을 제시하기 어려우나 개발주기에 따라 하나의 팀에서 2~4주 단위로 업데이트 할 수 있는 크기가 적당함

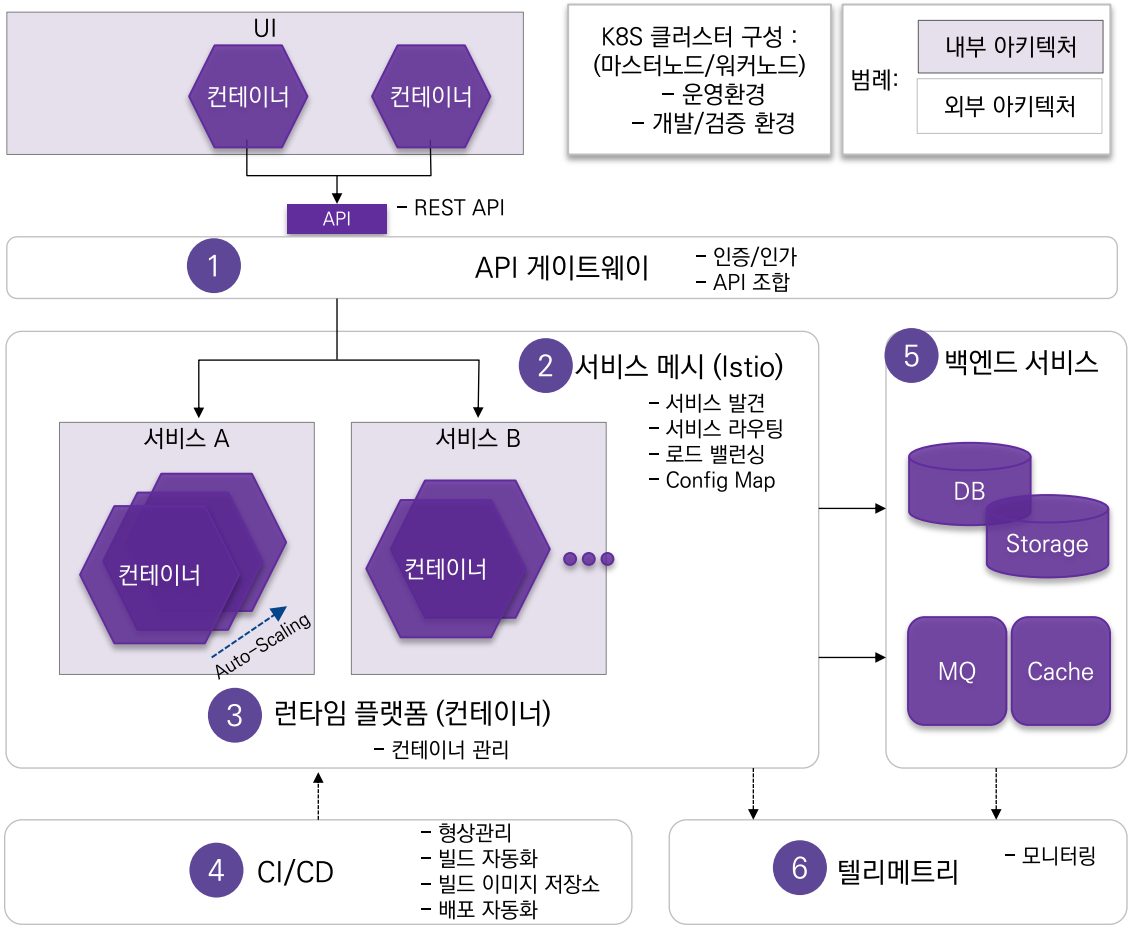
클라우드 네이티브 기반 행정·공공 서비스 확산 지원
클라우드 네이티브 발주자 가이드

클라우드 네이티브 애플리케이션의 **설계원칙**은 어떻게 되는가?

NIA 한국지능정보사회진흥원

클라우드 네이티브 참조 아키텍처

가트너, IBM 등에서 제시하는 클라우드 네이티브 참조 아키텍처를 토대로 클라우드 네이티브 애플리케이션 아키텍처를 설계함



분석 단계 설계 대상

1. API 게이트웨이 ★
2. 서비스메시 ★
3. 런타임플랫폼(컨테이너)
4. CI/CD
5. 백엔드서비스
6. 텔리메트리(모니터링)

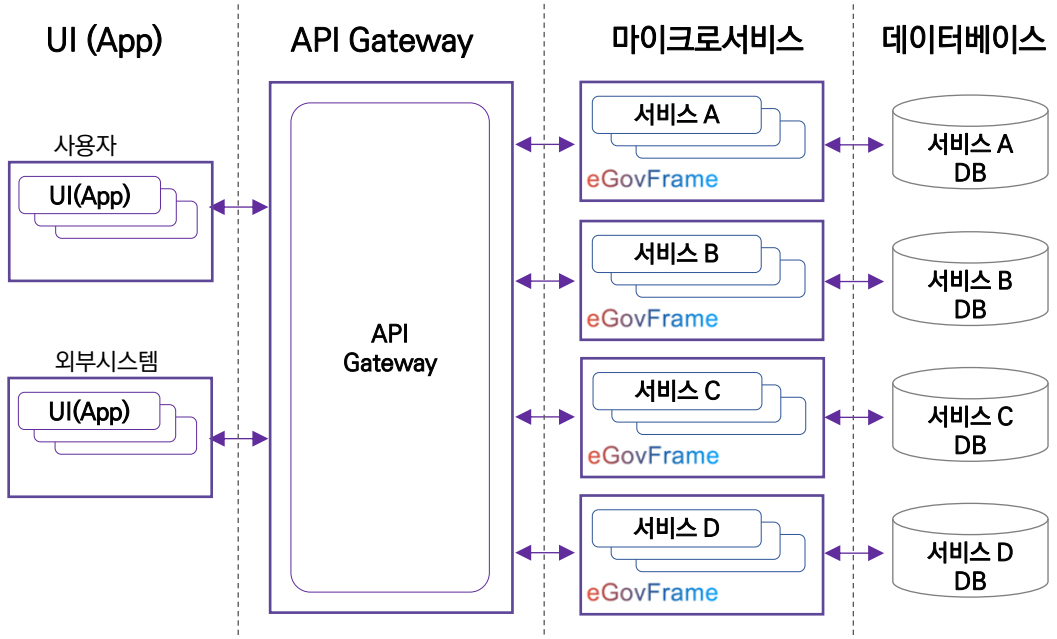
API 게이트웨이

클라이언트(UI)와 API 형태로 인터페이스를 제공하는 마이크로서비스 사이의 단일한 접점으로써 클라이언트 요청에 대한 라우팅 기능을 수행하는 서비스임

개념

- 클라이언트(UI)와 API 형태로 인터페이스를 제공하는 마이크로서비스 사이의 단일한 접점
- 클라이언트 요청에 대한 라우팅(전달) 기능을 수행하는 서버(서비스)

- 사용자(UI)와 API을 제공하는 내부 마이크로서비스의 중간에 위치함
- 마이크로서비스에 대한 단일한 접점(End point)을 외부에 제공함 API 중개자로써 다수 API 서버 (마이크로서비스)의 관리와 모니터링을 용이하게 함



UI (App)
- 사용자 혹은 외부 시스템에서 서비스 요청

API 게이트웨이
- 요청에 알맞은 서비스로 라우팅 (전달)

마이크로서비스
- API 서비스 제공자로 REST 기반의 서비스 노출

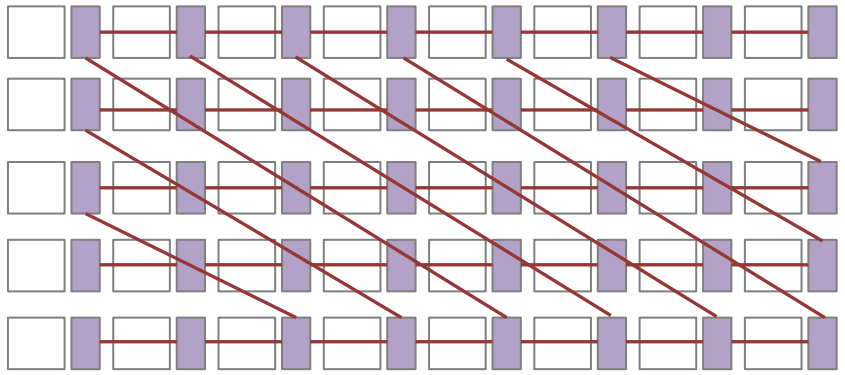
서비스 메시(Service Mesh)

마이크로서비스간 안정적인 통신을 위한 가상의 인프라 계층이며, 서비스메시 구현시 기본적으로 하나의 마이크로서비스 앞단에 경량화 프록시를 배치하여 서비스간 통신을 제어

서비스메시(Service Mesh)개념

- 마이크로서비스 간 안정적인 통신을 위한 가상의 인프라 계층 (Infrastructure Layer)
- 수많은 서비스의 제어와 관리를 위해 서비스간 통신이 중요해 졌으며 서비스메시는 이러한 서비스간 내부 통신 인프라를 제공함

메시 네트워크(Mesh Network)

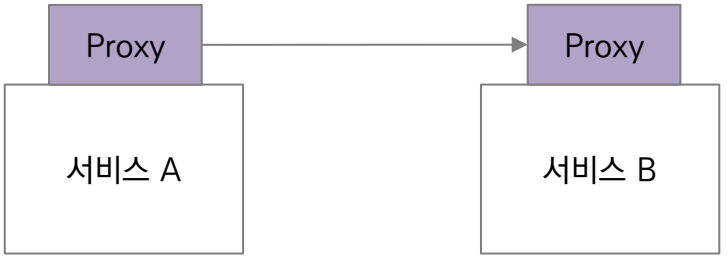


서비스메시구현방식

- 프록시에 라우팅 규칙, 타임아웃, 재시도(Retry) 횟수 등을 설정하여 마이크로 서비스의 비지니스 로직과 분리할 수 있어 기존의 서비스에 영향을 주지 않고 서비스를 제어할 수 있음

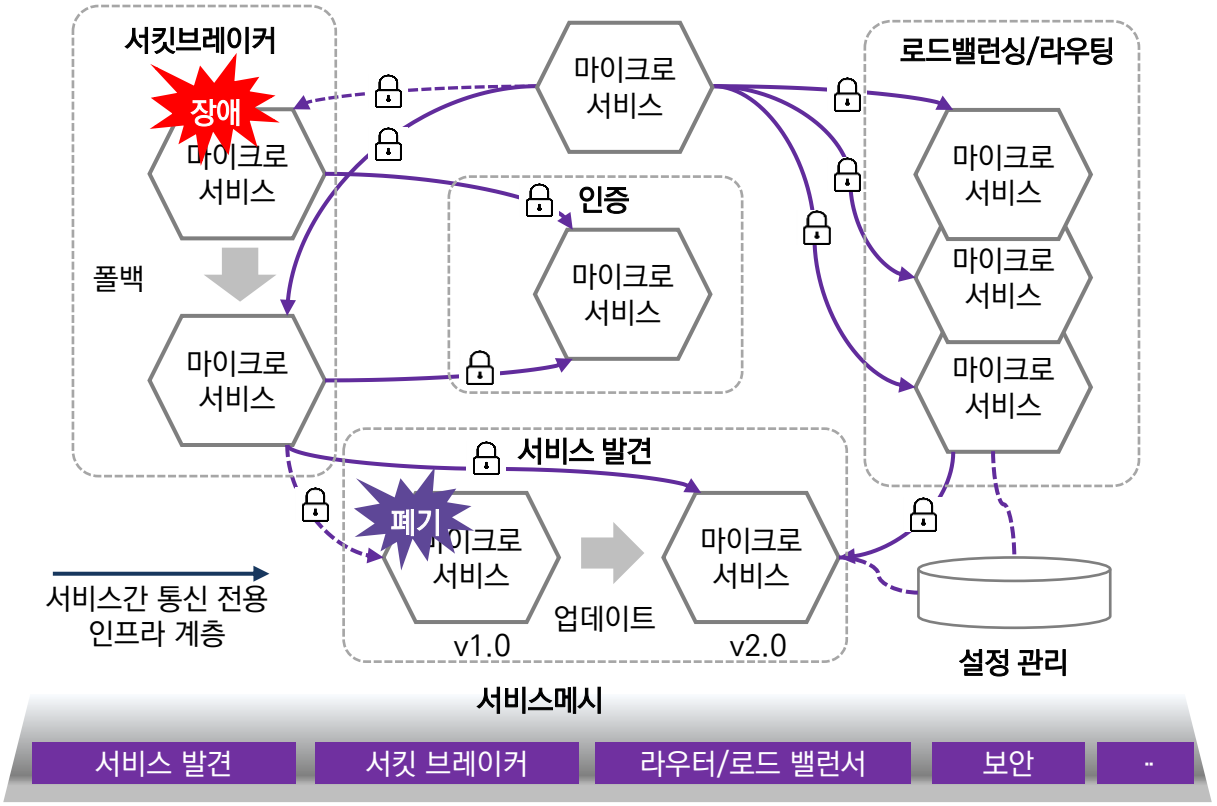
기존 서비스 호출 방식

서비스메시에서 서비스 호출 방식



서비스 메시(Service Mesh)

일반 애플리케이션에 비해 복잡한 환경과 그로 인한 기술적 구성요소를 필요로 하며, 서비스메시는 서비스의 제어, 추적, 관리 등 다양한 기능을 제공함

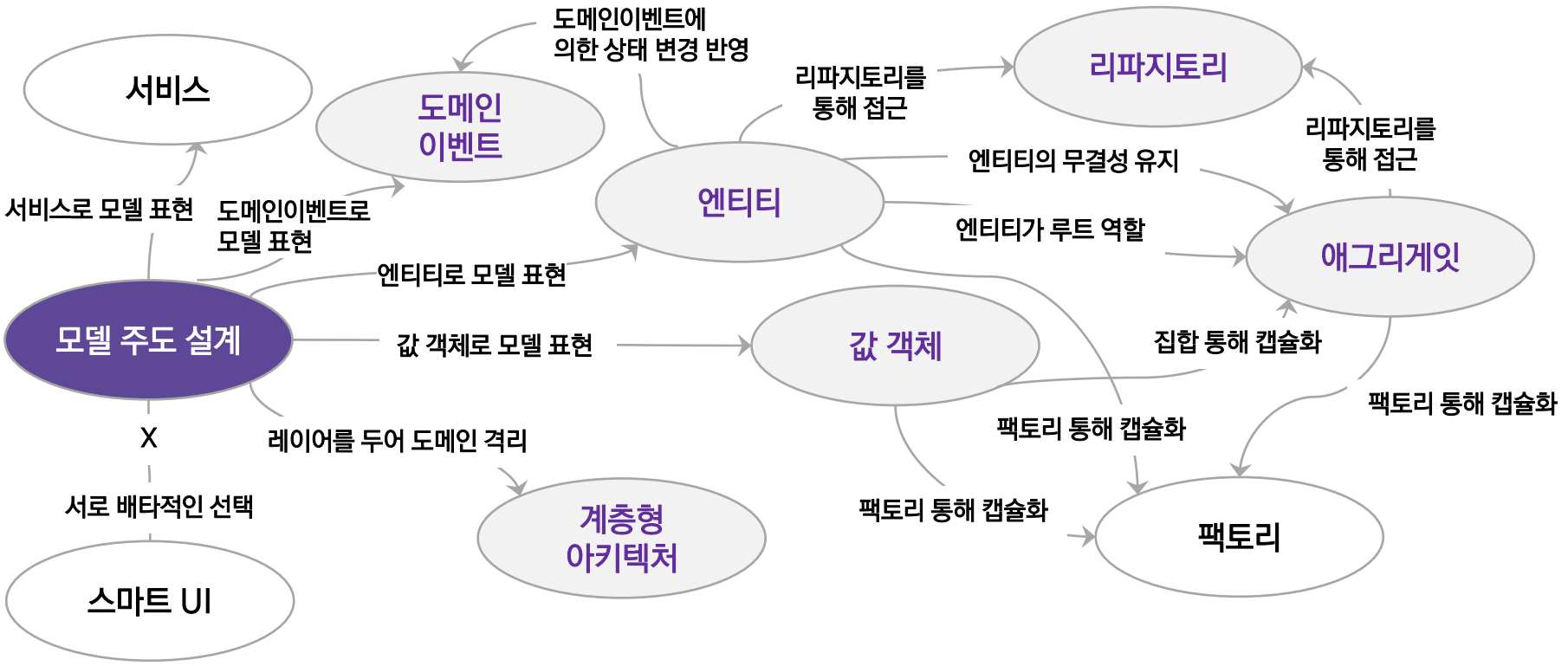


기능	설명
서비스 발견 (Service Discovery)	• 서비스 레지스트리를 통한 서비스 검색을 통해 마이크로서비스의 상태와 IP를 관리, 실시간 정상 서비스 정보 제공
서비스 라우팅 (Service Routing)	• 클라이언트의 요청을 확인된 정상 서비스(컨테이너)로 전달
로드밸런싱 (LoadBalancing)	• 서비스 인스턴스가 다수개로 구성된 경우 트래픽을 분산 처리
환경저장소 (Configuration)	• 환경정보를 컨테이너 실행환경과 분리하여 외부 저장소에 보관
인증 및 인가 (Authentication & Authorization)	• 서비스 통신의 인증, 권한부여 및 암호화 등을 관리
회복탄력성 (Circuit Breaker)	• 특정 서비스의 장애나 응답 지연이 다른 서비스로 전파되지 않도록 장애 회피 및 격리

도메인 모델의 표준 패턴 설계

전술적 설계는 전략적 설계에 의해 도출된 마이크로서비스별로 서비스 내부구조를 상세하게 정의하는 활동

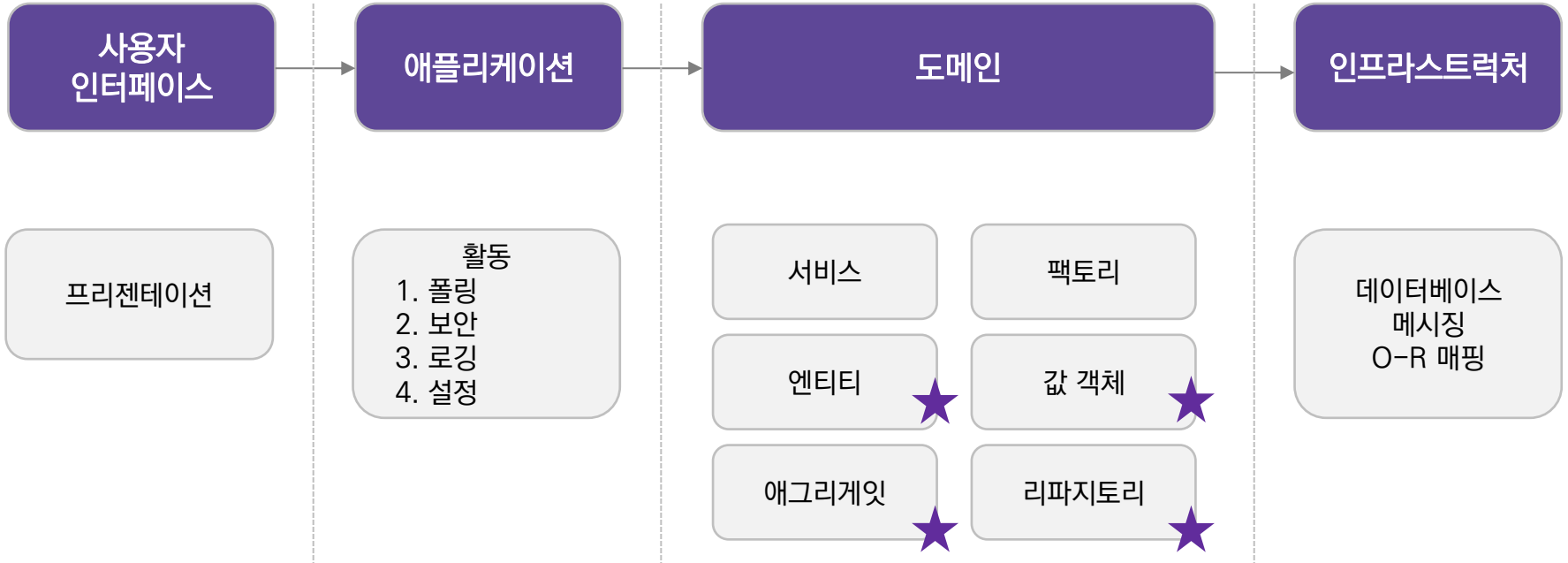
전술적 설계를 위한 도메인 모델링에 사용되는 표준 패턴과 각 패턴과의 관계



출처 : Domain Driven Design, Eric Evans

도메인 모델의 표준 패턴 설계 (계층적 아키텍처)

소프트웨어 시스템을 분리하는 방법으로서 계층적 아키텍처를 채택하고 있음



- 사용자에게 정보를 보여주고 사용자의 명령을 해석

- 사용자에게 정보를 보여주고 사용자의 명령을 해석
- 다른 시스템의 애플리케이션과 상호작용

- 소프트웨어의 핵심으로 업무에 대한 정보와 업무규칙을 표현
- 업무상황을 반영하는 상태를 제어하고 사용

- 상위 계층을 모두 지원(메시지 전송, 위젯 그리기 등)
- 레이어간 통신을 제공하고 비즈니스 객체의 영속성을 구현

도메인 모델의 표준 패턴의 계층적 아키텍처 엔티티 및 값 객체

엔티티(Entity)

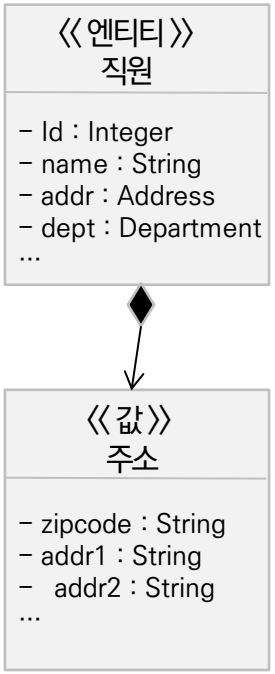
시간이 지나도 지속되는 고유의 ID가 있는 개체

엔티티 도메인 패턴

- 주요 특징
  - 고유의 식별자를 가짐
  - 식별자는 유일하고 생애동안 불변
  - 모델과 관련된 비즈니스 로직을 수행
  - 자신만의 라이프사이클을 가짐

값 객체 (Value Object)

개념적으로 식별이 필요 없고, 단순히 값 만을 갖고 있는 객체

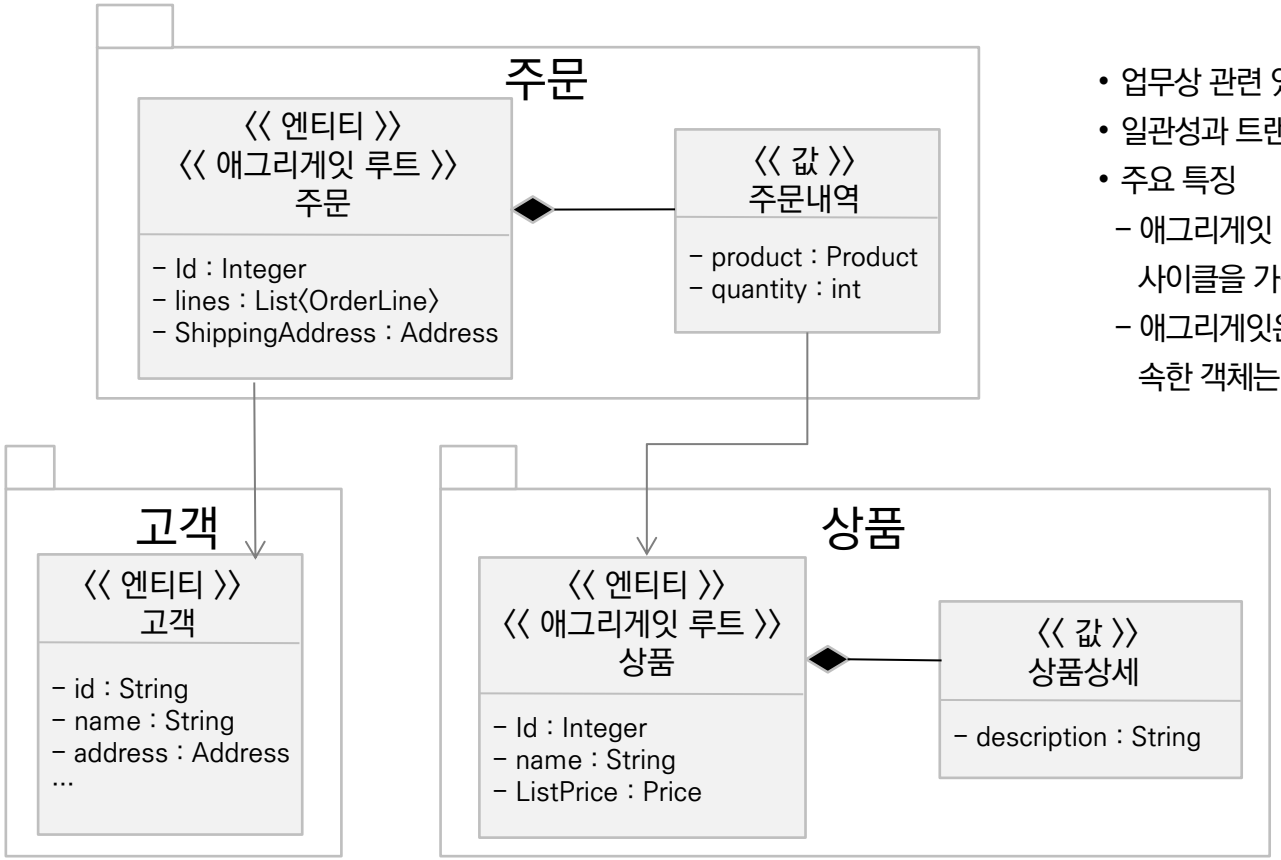


- 고유의 식별자를 갖지 않음
- 데이터를 표현하기 위한 용도로 주로 사용
  - 개념적으로 완전한 데이터 집합
  - 자신에게 알맞은 로직을 제공
  - 엔티티의 속성으로 사용

도메인 모델의 표준 패턴의 계층적 아키텍처 애그리게잇(Aggregate)

애그리게잇(Aggregate)

바운디드 컨텍스트를 구성하는 엔티티와 값 객체의 묶음



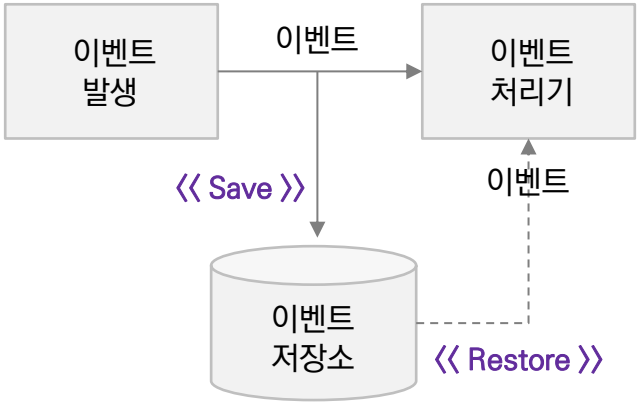
- 업무상 관련 있는 엔티티와 값 객체의 묶음
- 일관성과 트랜잭션, 분산의 단위 캡슐화를 통한 복잡성 관리
- 주요 특징
  - 애그리게잇 내의 객체는 동일하거나 유사한 라이프 사이클을 가짐 (예. 동시에 삭제처리 됨)
  - 애그리게잇은 경계를 가지며, 한 Aggregate에 속한 객체는 다른 Aggregate에 속하지 않음

※ 애그리게잇 루트 역할
- 애그리게잇 루트를 가짐
- 애그리게잇의 내부 객체들 관리
- 애그리게잇 외부에서 접근할 수 있는 유일한 객체

도메인 모델의 표준 패턴의 계층적 아키텍처 저장소와 도메인이벤트

저장소(Repository)

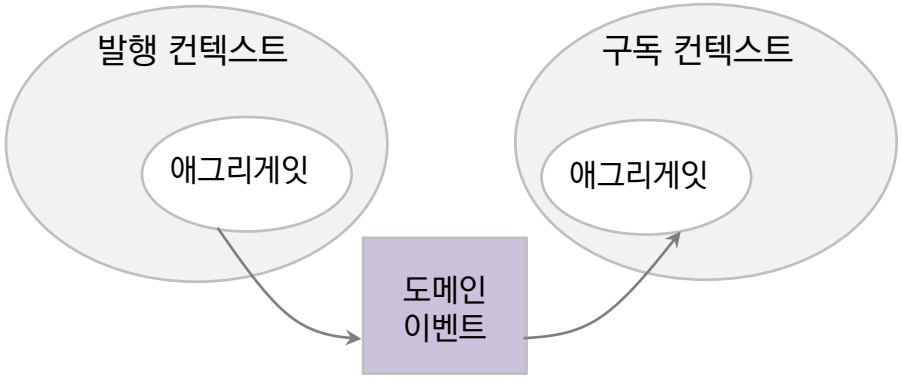
엔티티를 저장하는 공간으로 애그리게잇 단위로 동작



- 생성된 애그리게잇에 대한 영속성 관리
- 애그리게잇에 엔티티 저장, 애그리게잇 업데이트 및 삭제 수행
- ID로 애그리게잇 조회
- 애그리게잇 당 한 개의 저장소 인터페이스
- 자바의 MAP 인터페이스와 유사한 인터페이스 제공

도메인 이벤트(Domain Event)

바운디드 컨텍스트와 다른 자원들은 도메인 이벤트를 받아서 활용

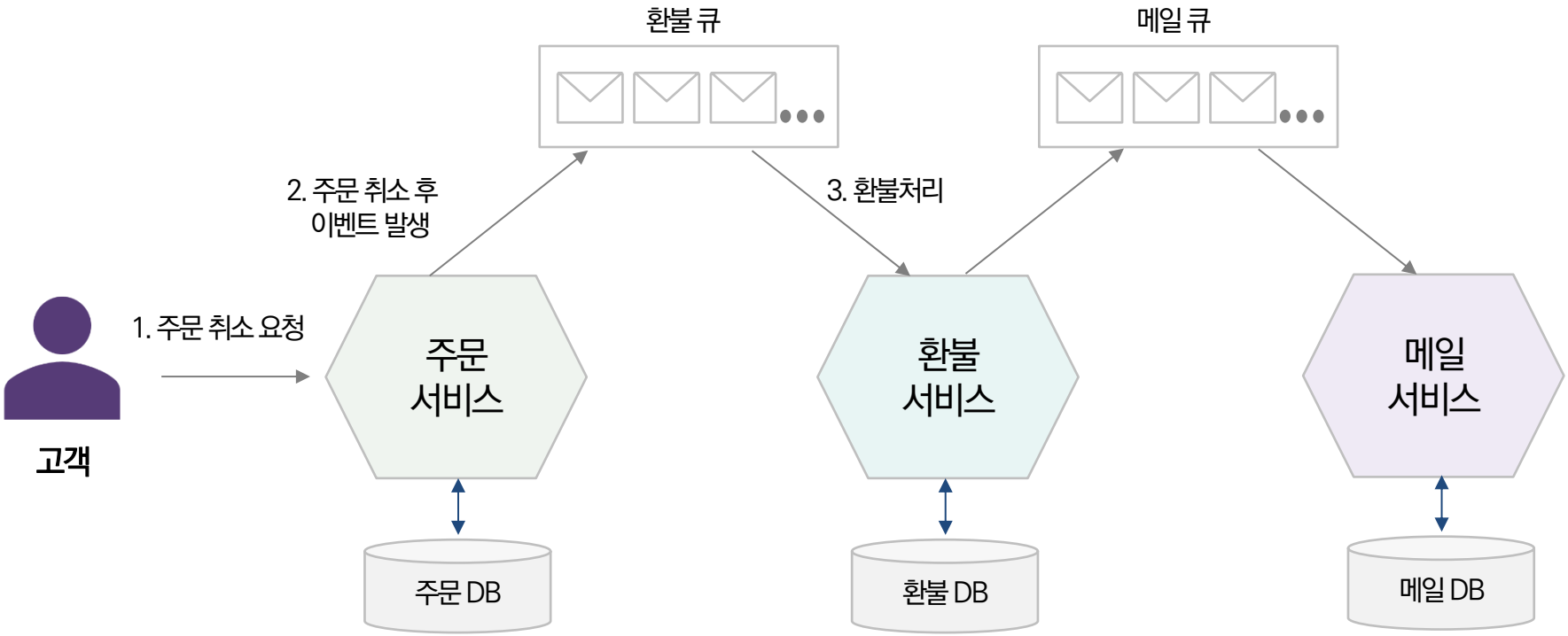


- 도메인 이벤트를 사용하며 도메인 내 변경의 파생 작업을 명시적으로 구현
- 주요 용도
  - 트리거 : 도메인 상태 변화 시 후처리가 필요할 경우 후처리를 실행하기 위한 트리거로 이벤트를 사용
  - 타 시스템간 데이터 동기화 : 이벤트 사용 시 서로 다른 도메인 로직이 섞이는 것을 방지

출처 : 도메인주도설계핵심, 박현철/전장호, 에이콘, 2017

서비스 호출 아키텍처 구조 설계 (이벤트 처리방식)

각 마이크로서비스는 개별 DB를 구성하고, 메시지 브로커의 메시지큐를 통해 이벤트를 전달받는 구조임

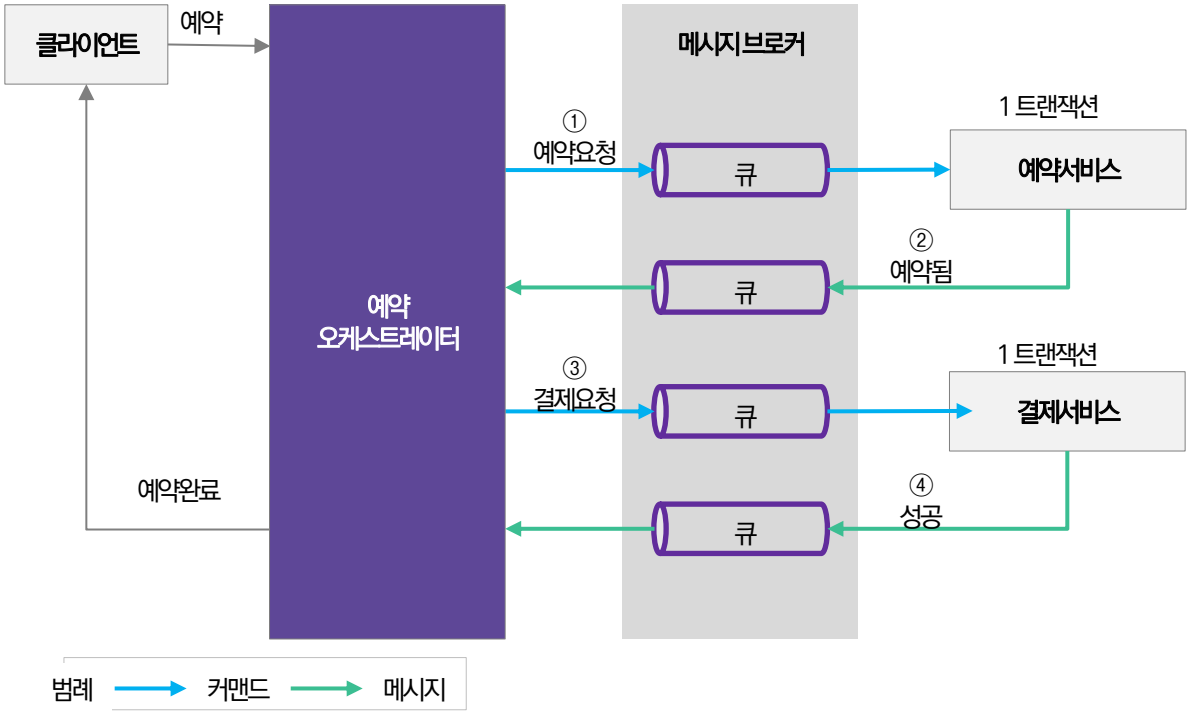


- 고객의 주문 취소 요청 시, 환불 서비스에 장애가 생기더라도 고객의 주문취소 요청을 받아들이고, 환불 처리는 추후에 처리 가능
  → 마이크로서비스로 분리하고, 이벤트 기반 비동기 처리를 할 수 있음
- 주문 서비스에서는 주문 취소 이벤트만 발행하면 되므로, 주문 취소 시 추가 기능은 이벤트를 통해 코드 수정 없이 확장 가능함

출처 : https://galid1.tistory.com/781

서비스 호출 아키텍처 구조 설계 (트랜잭션 처리방식)

비동기 방식 분산 트랜잭션 흐름인 SAGA 패턴 처리 구성



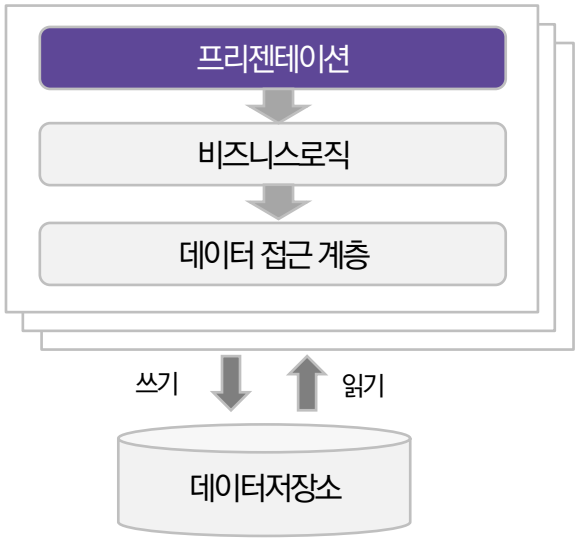
- 마이크로서비스에 할 일을 알려주는 오케스트레이터 클래스를 구현
- 오케스트레이터는 비동기 방식으로 마이크로서비스들과 상호작용
- 첫번째 마이크로서비스가 응답메시지를 주면, 오케스트레이터는 다음 마이크로서비스에게 커맨드를 요청

- SAGA 패턴은 트랜잭션의 관리주체가 DB가 아닌 애플리케이션에 있음
- 각각의 앱에 대한 연속적인 트랜잭션 요청 및 실패할 경우에 롤백 처리(보상 트랜잭션)를 애플리케이션에서 구현해야 함
- 애플리케이션의 트랜잭션 관리를 통해 최종 일관성(Eventually Consistency)을 달성할 수 있기 때문에 분산되어 있는 DB간에 정합성을 맞출 수 있음
- 트랜잭션 일관성을 유지하기 위해서 프로세스 수행 과정상 누락되는 작업이 없는지 면밀히 살펴야 하며 실패할 경우 에러 복구를 위한 보상 트랜잭션 처리 누락이 없도록 설계해야 함

서비스 호출 아키텍처 구조 설계 (트랜잭션 처리방식)

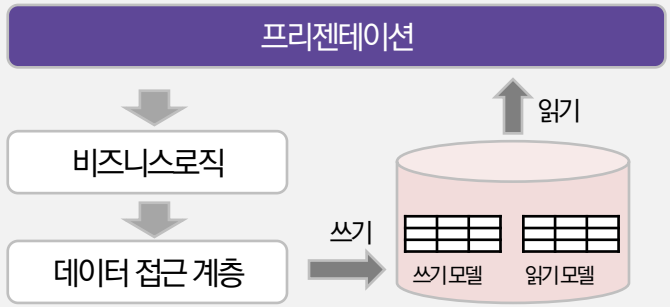
DB 트랜잭션 흐름인 CQRS 패턴 처리 구성

서비스

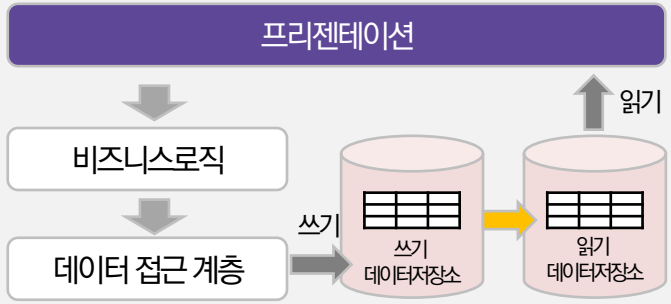


- 서비스의 성능향성을 위해 서비스 인스턴스를 스케일아웃하여 여러 개로 실행한 경우, 데이터 읽기/쓰기 작업으로 인한 리소스 교착 상태 발생 가능

- CQRS(Command Query Responsibility Segregation : 명령 조회 책임 분리)
  - 기존의 일반적 개념이었던 동일한 저장소에 데이터를 넣고 CRUD하는 방식과 달리 DB에 대한 쓰기와 읽기를 분리하는 방식(쓰기보다 읽기 요청이 훨씬 많음)



- 하나의 데이터저장소에서 쓰기모델과 읽기모델을 분리



- 물리적으로 쓰기 저장소와 읽기 저장소를 분리

쓰기와 읽기의 분리를 통해 쓰기 시스템의 부하를 줄이고 대기시간을 줄이는 등의 이점이 있음

클라우드 네이티브 기반 행정·공공 서비스 확산 지원
클라우드 네이티브 발주자 가이드

클라우드 네이티브 애플리케이션의 개발원칙은 어떻게 되는가?

NIA 한국지능정보사회진흥원

12 Factors와 클라우드 네이티브 환경의 관련성

개발원칙과 환경

12 Factors 원칙은 플랫폼을 가정한 개발 원칙으로, 플랫폼이 제공하는 환경을 이해하고 어플리케이션의 개발 및 빌드/배포/운영 환경 전반의 고려사항을 제시함

1 **코드베이스** (Codebase) [CI/CD]	하나의 코드베이스(소스코드)로 버전관리 버전관리 및 배포 관리 자동화 환경	7 **포트바인딩** (Port binding) [배포]	애플리케이션은 독립적이며, http 같은 **포트 바인딩을 이용한 서비스 공개**
2 **종속성** (Dependencies) [Code] [CI/CD]	패키지, 라이브러리 등 **의존관계는 명시적으로 선언하고 분리**	8 **동시성** (Concurrency) [운영]	애플리케이션을 수평적으로 확장, **프로세스 모델을 사용한 스케일아웃**
3 **설정** (Config) [Code] [배포]	소스 코드와 설정 정보를 분리 **설정 정보는 환경변수에 저장**	9 **폐기 가능** (Disposability) [Code] [운영]	**빠른 시작과 그레이스풀 셧다운** (graceful shutdown) **지원으로 안정성 극대화**
4 **백앤드서비스** (Backing Services) [배포]	**백엔드 서비스**(DB, 메시지큐, 캐시등) **는 리소스 매핑으로 처리**(연결/분리 용이)	10 **개발/운영환경일치** (Dev/prod parity) [배포]	**가능한 동일한 개발, 스테이징, 운영 환경의 유지**
5 **빌드/배포/실행** (Build, release, run) [CI/CD]	**빌드와 실행 단계는 철저히 분리**	11 **로그** (Logs) [Code] [운영]	**로그파일을 이벤트 스트림으로 로그 처리** (취합, 인덱싱, 분석)
6 **프로세스** (Processes) [Code] [배포]	**어플리케이션은 Stateless 프로세스로 실행**	12 **운영관리 프로세스** (Admin processes) [운영]	**시스템 관리작업은 일회성 프로세스로 만들어서 실행**

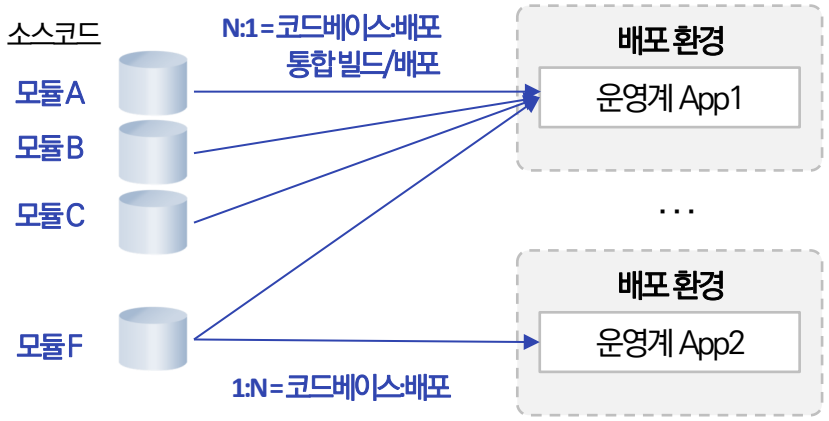
12가지 요소 : 1) 코드 베이스

코드 베이스

애플리케이션은 1개의 코드베이스를 통해 관리되어야 하며, 동일한 코드로 개발 및 운영 환경에 배포되어야 하며, 코드베이스는 각 애플리케이션마다 단 1개 존재 (애플리케이션은 소스코드의 변경과 여러 환경으로 수차례 배포 시에도 추적 가능)

AS-IS : 코드베이스와 배포 앱 간 N:N 관계

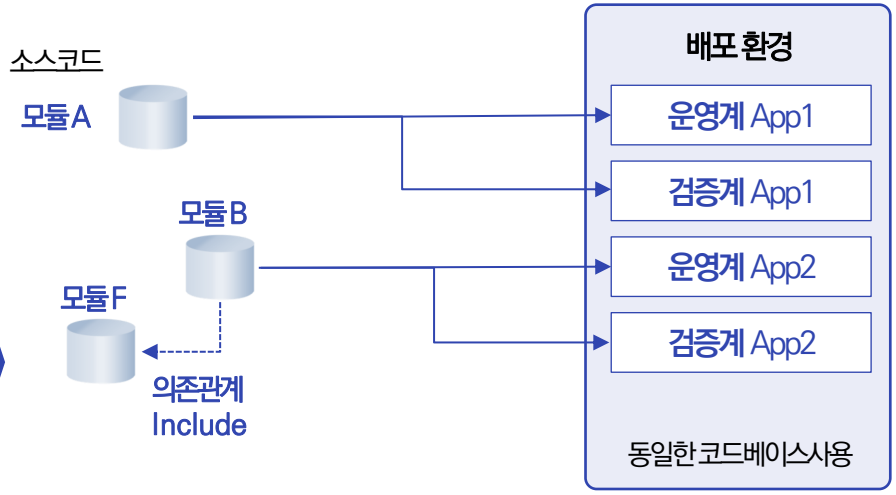
1:N, N:1 의 코드베이스와 배포 앱 관계



- 소스코드와 배포 앱 간의 관계가 1:N 혹은 N:1 관계 형성되어, 배포 앱의 코드 추적과 버전 관기 어려움이 있음
- 애플리케이션 관련 모든 자산, 소스코드, 프로비저닝 스크립트, 환경 설정 등 내용이 통일된 코드베이스(리파지토리)에 저장되지 못함

To-BE : 코드베이스와 배포 앱간 1: 1 관계 운영

코드베이스는 애플리케이션을 관리하는 리파지토리 (저장소)

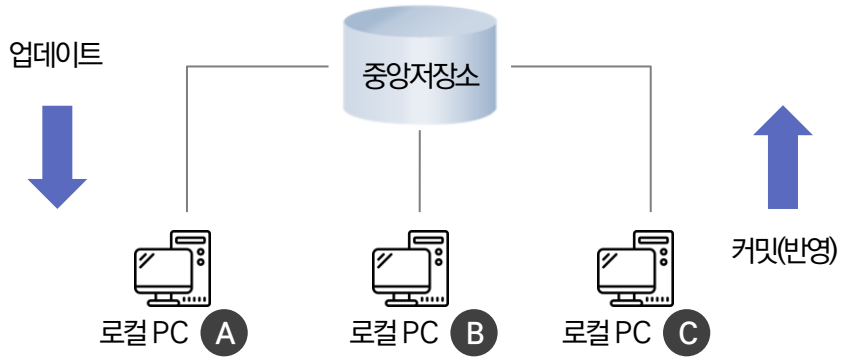


- 코드베이스는 SVN과 같은 중앙관리형과 Git과 같은 분산관리형이 존재함
- 애플리케이션 관련 모든 자산, 소스코드, 프로비저닝 스크립트, 환경 설정 등 내용이 코드베이스(리파지토리)에 저장되어 개발자 및 운영 담당자 등에 의해 사용됨
- 모든 배포는 1개의 코드베이스를 가지고, 코드베이스는 CI/CD 도구에 의해 관리

코드베이스의 유형

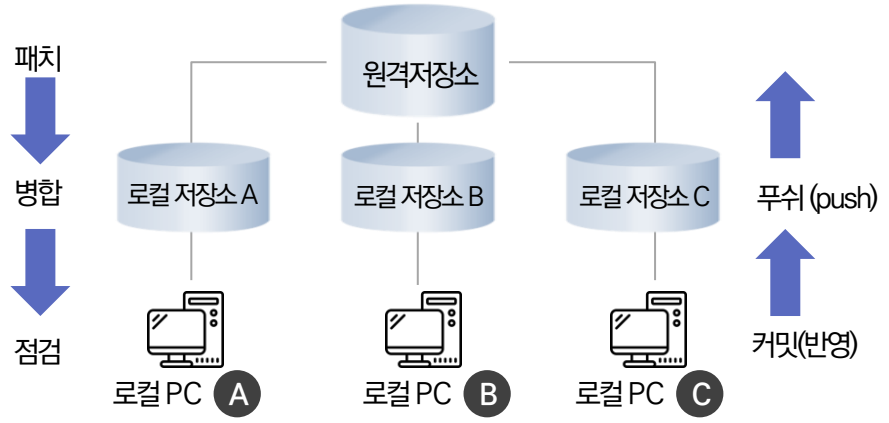
코드베이스는 SVN과 같은 중앙관리형과 Git과 같은 분산관리형으로 나뉘며, 각 방식의 장단점은 다음과 같음

중앙관리형 – SVN



- 로컬 PC에서 커밋(Commit)하면 중앙저장소에 반영
- **장점**
  - 직관적으로 사용 가능
  - 커밋하는 순간 모든 사람에게 공유되고, 모든 사람이 같은 자료를 받아 옴
- **단점**
  - 여러 사람이 하나의 파일을 동시에 수정하고 커밋할 경우, 충돌 발생 가능성
  - 중앙저장소 다운 시 작업의 어려움

분산관리형 – Git



Pull = 패치(fetch) + 병합(merge)

- 로컬 PC에서 커밋하면 로컬저장소에 반영되고, 로컬저장소에서 푸쉬(Push)하면 원격저장소에 반영
- **장점**
  - 하나의 파일에 대해 작업자별 다른 작업, 별도 이력관리 가능 → 동시다발 작업
  - 원격저장소 다운 시 로컬저장소로 작업 가능
  - 로컬저장소에 올리기 때문에 파일 유실 염려 저하, 작업이력 관리 효율 증대, 속도 향상
- **단점**
  - 직관적이지 못하고 개발자가 적응하는데 시간이 소요됨

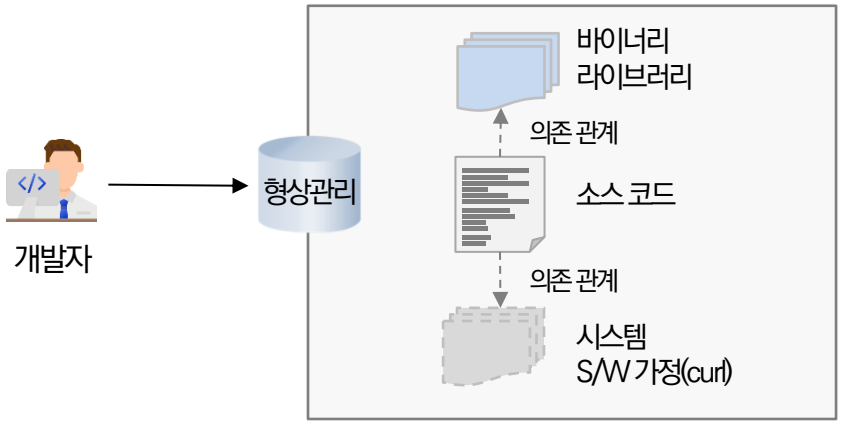
12가지 요소 : 2) 종속성

Code
종속성

종속관계에 있는 모든 애플리케이션은 명시적으로 종속성을 선언하며, 애플리케이션이 필요로 하는 라이브러리를 종속성 파일에 명시적으로 선언하여 사용함

AS-IS : 소스코드 내 시스템 및 라이브러리 포함

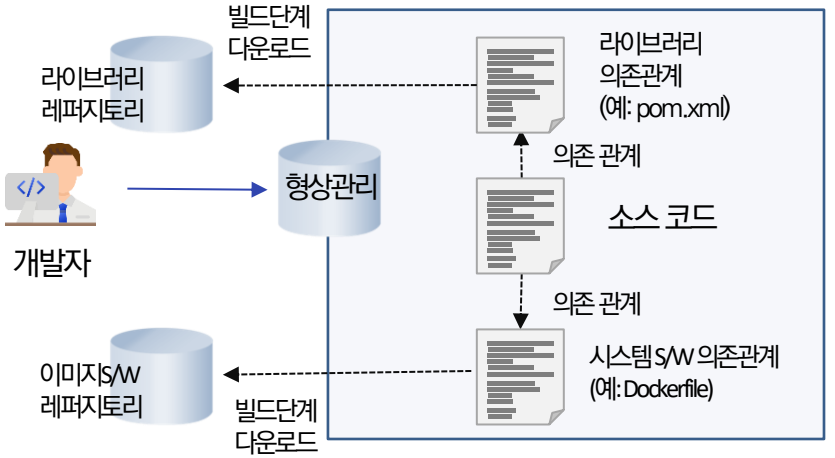
종속성이 라이브러리가 암묵적으로 포함



- 어플리케이션이 의존하는 라이브러리 및 시스템 S/W 가 암묵적으로 관리되어 버전 관리 및 환경 구성이 어려움
- 개발에 새롭게 참여하는 개발자의 환경설정은 재 구성하여야 함
- 애플리케이션 실행 시 종속성 분리(Dependency Isolation) 도구를 사용하지 않음
- 라이브러리 통합관리가 제공되지 않아, 다양한 플랫폼간 이식성을 제공하지 못함

To-BE : 종속성이 있는 라이브러리 통합관리

명시적 의존 관계 관리



- 애플리케이션의 모든 종속성을 명시적으로 선언하여 사용
- **애플리케이션이 필요로 하는 라이브러리를 종속성 매니페스트 파일에 명시적으로 선언하여 사용**
- 클라우드 네이티브 애플리케이션은 다양한 환경에 배포될 수 있으며 각 환경에서 정상 동작하도록 보장
- 모든 종속관계는 종속성 선언을 통해 완전하고 정확하게 이루어짐
- 반복적인 배포 지원

종속성 선언 도구 활용 예시

Maven, Gradle, Node npm 등과 같은 종속성 관리 및 빌드 도구를 활용하여 종속성을 관리하도록 함

종속성 관리 및 빌드 도구를 활용한 종속성 선언

라이브러리 A

라이브러리 B

라이브러리 C

Pom.xml

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.5.5</version>
  <exclusions>
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Maven™

Build.gradle

```
dependencies {
    classpath 'com.android.tools.build:gradle:4.2.0'
}
```

Gradle

Node.js npm

```
{
  "name": "A",
  "dependencies": {
    "B": "^1.0.0",
    "C": "^2.0.0"
  }
}
```

node.js + npm

- 라이브러리 종속성 관리 및 빌드 도구의 사용
- 애플리케이션 소스코드는 코드베이스, 라이브러리는 라이브러리 저장소에서 관리되어야 함
- 모든 라이브러리는 라이브러리 저장소에서 관리되고 내려 받을 수 있어야 함
- **특정 버전 관리 중요**
- Node.js 패키지, Java .jar, .Net의 DLL과 같은 외부 아티펙트 파일은 실행 시 메모리에 로드된 종속성 선언 매니페스트에서 참조됨

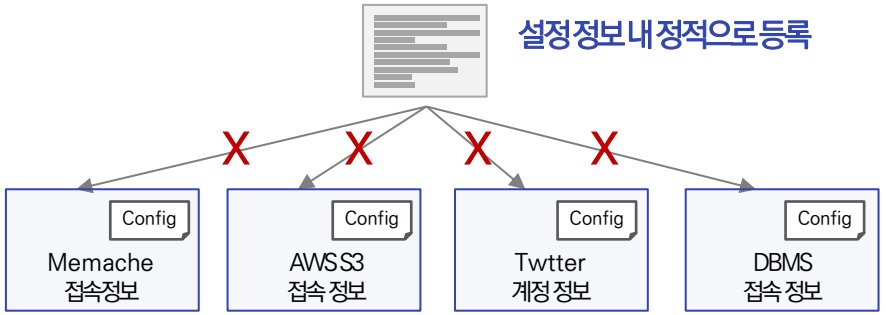
12가지 요소 : 3) 설정

설정

애플리케이션 실행 시 필요로 하는 설정 정보와 코드는 분리하여 관리
(예: 업무처리 로직과는 무관한 시스템 내외부의 리소스, 배포단계, OS 등에 따라 달라지는 설정정보)

AS-IS : 코드내 설정정보가 존재

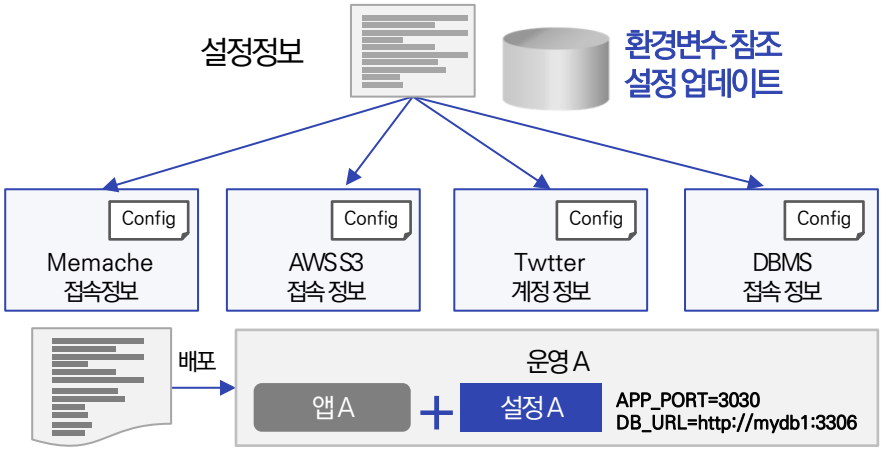
**코드 및 Properties 파일 내
애플리케이션 실행과 관련된 정적인 설정정보**



- 설정을 애플리케이션 코드의 외부에 별도로 두지 않음(소스코드 내부에 위치)
- 환경변수에 설정값 정의
- 애플리케이션 코드는 배포 환경과 관계없이 동일하게 구성할 수 없음

To-BE : 애플리케이션 코드와 설정의 엄격한 분리

**환경변수를 참조하여 환경에 따른
업데이트 가능하도록 설정정보를 정의**



- 배포환경을 위한 설정 파일 작성 시 Spring Cloud Config 사용
- 코드베이스는 여러 환경에 배포되며, 환경에 따라 DB, 스토리지, 기타 클라우드 리소스 정보가 달라짐
- 마이크로서비스 환경에서는 git(spring-cloud-config)와 같은 소스 제어에서 애플리케이션에 대한 설정을 관리

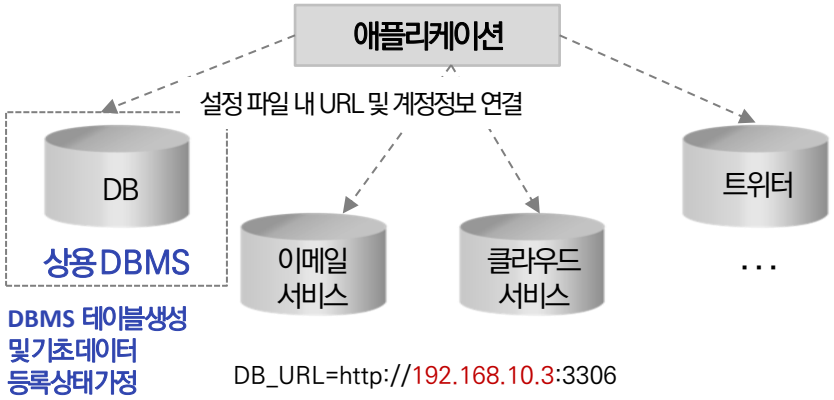
12가지 요소 : 4) 백앤드 서비스

백앤드 서비스

DB, 캐시, 메시지/큐, 이메일 서비스, SNS, 클라우드 서비스 등 백엔드 서비스는 애플리케이션에 연결된(Attached) 리소스로 취급됨

AS-IS : 기존 백엔드 서비스

애플리케이션에서 백엔드 서비스 정적인 환경으로 관리

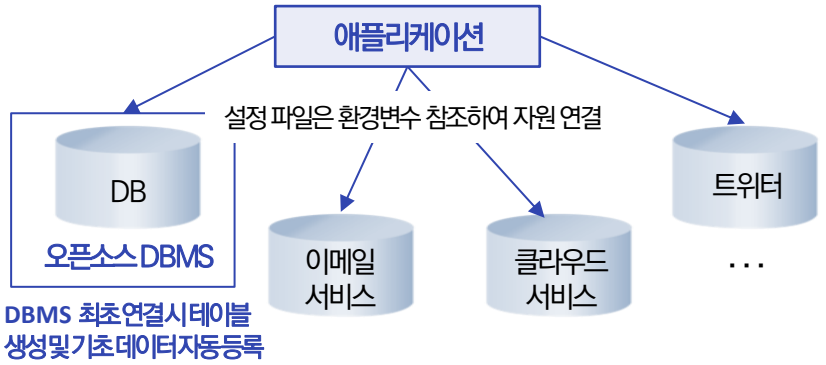


백엔드 서비스 유형
- 데이터베이스: Oracle, MySQL, CouchDB, MariaDB 등
- 메시지/큐: RabbitMQ, Beanstalkd 등
- 이메일(SMTP)서비스 Postfix 등
- 캐시 시스템: Memcached 등

- 어플리케이션 백엔드는 사전 준비된 연결 설정이 마쳐진 상태 (예: DBMS 기초 데이터 생성)
- 백엔드 서비스를 설정파일 내 정적 URL 및 계정정보로 관리

To-BE : 클라우드 네이티브 백앤드 서비스

애플리케이션에서 백엔드 서비스는 언제든 연결되고 변경될 수 있는 연결된 리소스로 관리



설정파일	변경	설정파일
APP_PORT=3030 DB_URL=http://dev:3306	→	APP_PORT=3030 DB_URL=http://prd:3306

"설정파일이나 Property 파일에 접속정보(URL, Locator) 포함"

- 애플리케이션연결된 DBMS의 변경시 애플리케이션 코드변경 없어야 함
- 독립적인 설정 파일에 백엔드 서비스 연결 정보 포함
- 설정파일의 해당 내용 변경만으로 리소스를 변경하도록 함

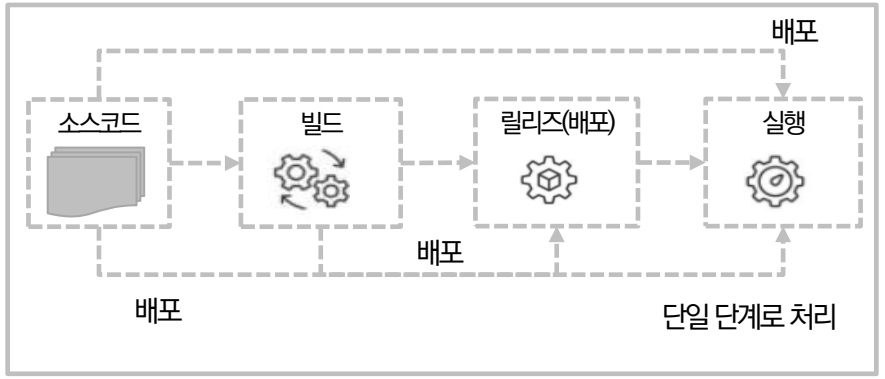
12가지 요소 : 5) 빌드·릴리즈·실행 환경

Code
빌드·릴리즈·실행 환경

코드베이스는 엄격하게 구분된 빌드, 릴리즈, 실행 3단계의 과정을 통해 배포가 이뤄지며, 각 단계는 엄격하게 분리되어야 함

AS-IS : 비순차적, 미분리된 실행환경

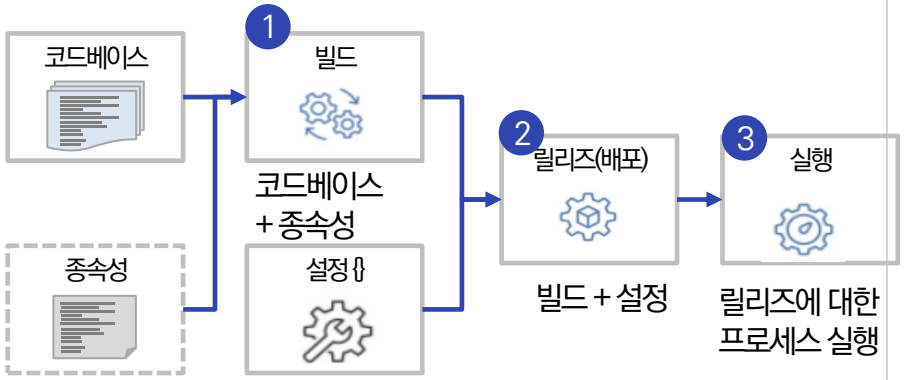
단계가 명확히 구분되지 않는 환경



- 빌드 단계, 릴리즈 단계, 실행 단계에 대한 명확한 구분이 없음
- 순차적인 단계에 따르지 않고, 직접 빌드/배포하는 경우가 다수임

To-BE : CI/CD기반 순차적 실행 환경

CI/CD 파이프라인 기반의 태스크 수행



1 빌드 단계
- 코드 저장소를 빌드라고 불리는 실행가능한 번들로 변환
- 배포 과정에서 지정된 특정 커밋 시점의 코드를 사용해 의존성을 설치하고 바이너리와 에셋 파일을 컴파일

2 릴리즈 단계
- 빌드 단계에서 생성된 빌드를 넘겨받아 현재 배포의 설정과 결합
- 고유한 릴리즈 아이디를 할당하며 변경될 수 없으며, 변경 시 새로 만들어 짐

3 실행 단계
- 특정한 릴리즈에서 필요한 프로세스들을 컨테이너로 실행

빌드·릴리즈·실행 환경 도구 예시

단계	수행 주체	수행 도구	설명
설계	Dev	Spring/Spring Boot, Gradle, Maven	• 개발자가 의존성에 대한 이해도가 가장 높음
1 개발 + **빌드단계**	CI	.war 또는 .jar생성	• **빌드단계에서는 코드베이스의 소스코드와 종속성(라이브러리)를 내려받아 컴파일한 후 하나의 패키지를 만듦** • 1번의 빌드로 다수 배포 • 빌드는 개발자에 의해 실행됨 • 빌드 시 오류는 개발자에 의해 해결 가능
2 **릴리즈 단계**	플랫폼	• Droplet, 도커 이미지	• **릴리즈단계에서는 빌드된 패키지에 환경 설정 정보를 조합** • **모든 릴리즈는 고유의 릴리즈 ID를 가지며, 변경 불가 (변경 시 새로운 릴리즈 ID 부여)** • 민첩한 배포, 업그레이드, 롤백 • 배포도구에서 릴리즈 기능 제공
3 **실행 단계**	플랫폼	• 컨테이너 + 프로세스	• **실행단계에서는 릴리즈에서 만들어진 결과물은 실행환경에서 애플리케이션으로 실행** • 속도(Speed) 중요 • 운영 담당자, 재부팅, 프로세스 재실행 등에 의해 실행 – 자동으로 실행될 수 있으므로 실행 단계의 변경 작업 최소화 필요

• 설계, 개발, 릴리즈, 실행 단계는 엄격하게 구분됨
– 실행 시 변경된 코드를 빌드 단계로 역전파할 수 없으며, 이러한 코드 수정은 불가능

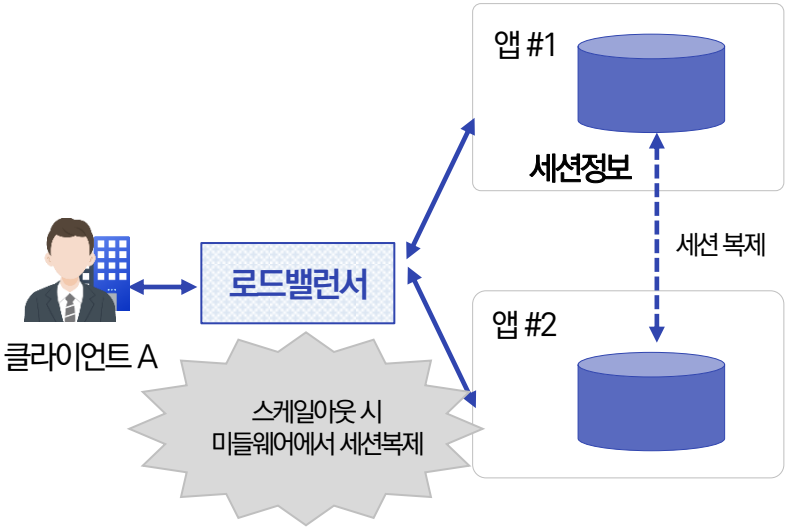
12가지 요소 : 6) 무상태 서비스

Code
무상태 서비스

상태 프로세스는 클라이언트와 세션 정보를 서버에 저장하므로 스케일아웃 시 관련 정보의 이동 작업이 필요한 반면, 무상태 프로세스는 서버에 저장하지 않고 외부 DB에 저장하여 스케일아웃이 용이한 구조임

AS-IS : 상태(Stateful) 프로세스

클라이언트와 서버간 세션의 상태(상태 정보 저장)에 기반하여 클라이언트에 응답을 보냄



- 클라이언트와 세션 정보를 서버에 저장함
- 스케일아웃 시, 클라이언트와 세션 정보를 옮겨주는 부수적 작업 필요

To-BE : 신규 무상태(Stateless) 프로세스

서버는 클라이언트, 서버간 세션 상태와 독립적으로 클라이언트에 응답을 보냄

- 애플리케이션은 실행환경에서 1개 이상의 프로세스로 실행되며, 각 프로세스는 무상태(Stateless)로 어떠한 것도 공유하지 않아야 함
- 클라이언트와 세션 정보를 서버에 저장하지 않고, 외부 DB에 저장함
- 서버에 클라이언트와 세션 정보가 없으므로 스케일아웃이 가능한 구조

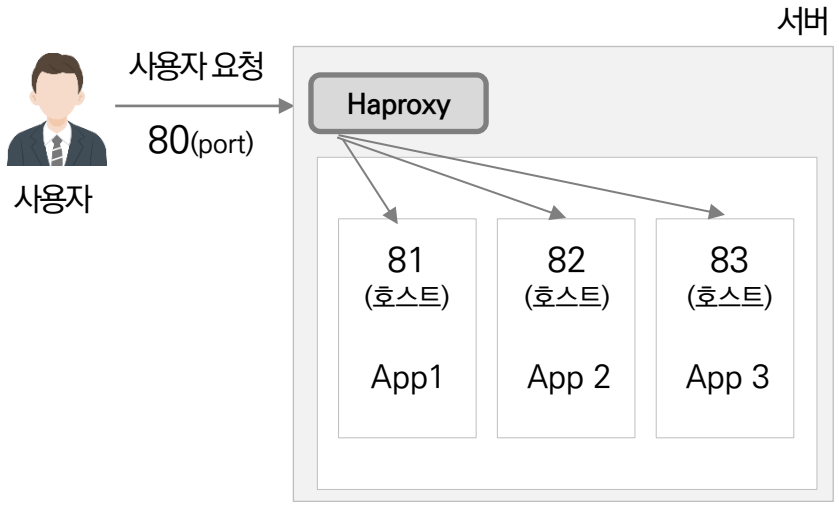
12가지 요소 : 7) 포트 바인딩

포트 바인딩

포트 바인딩은 메시지를 송수신하는 위치와 방법을 결정하는 구성 정보을 의미하며, 배포된 애플리케이션을 타 애플리케이션에서 접근할 수 있도록 포트 바인딩을 통해 서비스를 공개함(서비스 공개 및 접근성 제공)

AS-IS : 온프레미스 환경의 앱 포트 수동 정의

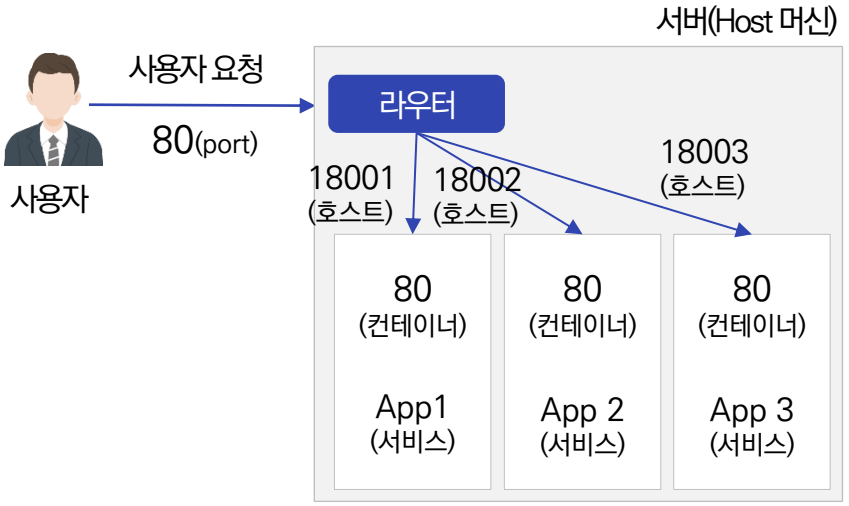
포트 충돌이 되지 않도록 배포 시 수동 관리



- 온프레미스에서 복수의 WAS 실행 시 서로 다른 포트로 실행
- 도메인 분기 처리를 위한 L7 라우터를 배치
- **서버 환경에 따라 배포 시 앱의 포트 변경이 필요**

To-BE : 클라우드 네이티브 앱의 포트 바인딩

플랫폼에서 포트 바인딩을 통해 자동으로 매핑 처리



- **호스트 포트와 컨테이너 포트간 바인딩을 플랫폼에서 처리**
- **사용자 도메인과 컨테이너 호스트 포트간 매핑을 플랫폼에서 처리**
- 사용자는 앱 서비스를 위한 도메인 지정만으로 서비스 라우팅을 요청할 수 있음

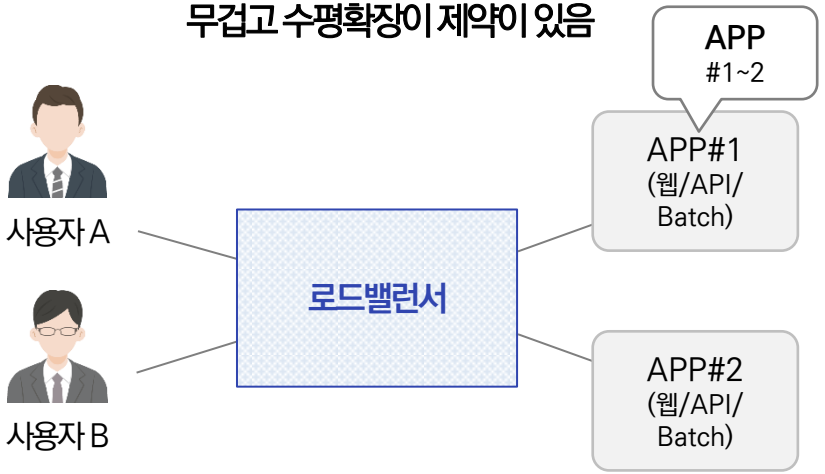
12가지 요소 : 8) 동시성(확장성 포함)

동시성

모든 일을 처리하는 하나의 프로세스 대신 기능별로 분리된 마이크로 프로세스를 실행하며, 프로세스 모델을 기반으로 수직적(Scale-up) 및 수평적(Scale-out) 확장성을 제공함

AS-IS : 모든 일을 처리하는 하나의 프로세스

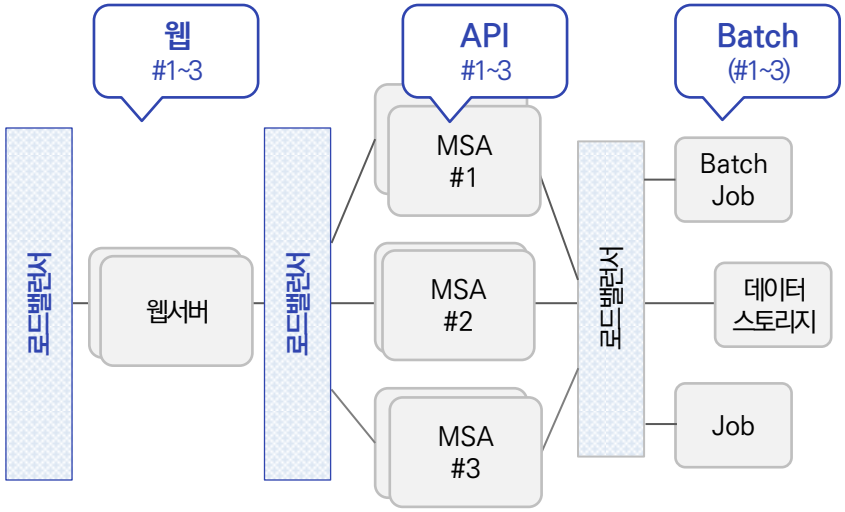
모든 일 (웹, API, Batch)을 처리하는 하나의 프로세스로 배포 무겁고 수평확장이 제약이 있음



- 하드웨어 또는 소프트웨어 기반 부하분산 (L4)
- 전동적인 웹로직과 같은 미들웨어 기반으로 웹, API, 배치 어플리케이션을 통합하여 배포하는 방식
- 특정 유형의 프로세스 스케일 아웃이 어렵고 전체 노드를 증설하거나 스케일 업 필요

To-BE : 프로세스 유형별 수평적 확장이 가능한 구조

수직적 (Scale-up) 확장 및 수평적 (Scale-out) 확장 제공



- 소프트웨어 기반 L4, L7기능 제공
- 프로세스 유형의 확장, 워크로드의 다양성, 마이크로서비스 지원
- 동시성을 지원하는 경우에는 해당 요청사항을 충족하도록 애플리케이션 유형 별 수평적으로 확장할 수 있음
- 비즈니스 서비스에 대한 부담으로 병목현상 발생 시 해당 계층을 독립적으로 확장 가능

12가지 요소 : 9) 폐기 가능성

Code
폐기 가능성

MSA는 각종 요청에 의해 스케일 업/다운이 빈번히 발생하므로 프로세스는 Shutdown 신호를 받았을 때 정상종료해야 함 .
빠른 시작(Start-up) 및 정상 종료(Graceful Shutdown)로 안정성 극대화

AS-IS : 기존 프로그램 처리

미들웨어 담당자에 의한 중지 및 기동으로, 정기 PM 시간에 기동 및 중지. 자원 초기화 및 종료를 모니터링 및 관리

- 미들웨어가 제공하는 기동/중지 시그널을 고려하여 앱의 설계 필요
- 미들웨어 관리자에 의해 기동/종료가 관리되고 리소스 초기화를 점검

To-BE : 빠른 시작과 정상종료가 가능한 폐기 가능성 처리

앱 부하 조건에 따라 자동적인 앱 스케일 발생하며, 애플리케이션 기동/중시 시 자원 관리가 자동으로 처리

- 플랫폼이 제공하는 기동/중지 시그널을 고려하여 앱의 설계 필요
- 어플리케이션은 마이크로서비스로 설계되어 수초 이내의 빠른 시작과 종료
- 플랫폼에 의해 자동으로 관리되며, 어플리케이션은 자동화된 자원 관리 설계 고려

애플리케이션 종료 시 DB 연결과 자원 사용 종료 예시

애플리케이션 사용 종료 시 모든 DB 연결 및 각종 리소스 사용이 올바르게 종료되고, 모든 종료 활동이 기록되어야 함

애플리케이션 사용 종료 시 DB 연결과 자원 사용 종료

```
const shutdown = async (signal) => {
   logger.info(`Disconnecting message broker at ${new Date()}`);
   messageBroker.disconnect();

   logger.info(`Disconnecting database at ${new Date()}`);
   database.disconnect();

   let shutdownMessage;

   if (!signal) {
       shutdownMessage = (`MyCool service shutting down at ${new
Date()}`);
   } else {
       shutdownMessage = (`Signal ${signal} : MyCool service
shutting down at ${new Date()}`);
   }
   const obj = {status: "SHUTDOWN", shutdownMessage, pid:
process.pid};
   await server.close(() => {
       console.log(obj);
       process.exit(0);
   });
};
```

- **Disposability의 원칙**
  - 애플리케이션이 신속하게 시작되고 정상적으로 종료되어야 함
- **애플리케이션의 시작**
  - 사용자가 애플리케이션에 접근하기 전에 DB 연결, 네트워크 리소스에 대한 액세스, 기타 구성 작업 등이 수행되어야 함
- **애플리케이션의 종료**
  - 사용자가 애플리케이션 사용을 종료할 경우 모든 DB 연결 및 각종 리소스 사용이 올바르게 종료되고, 모든 종료 활동이 기록되어야 함

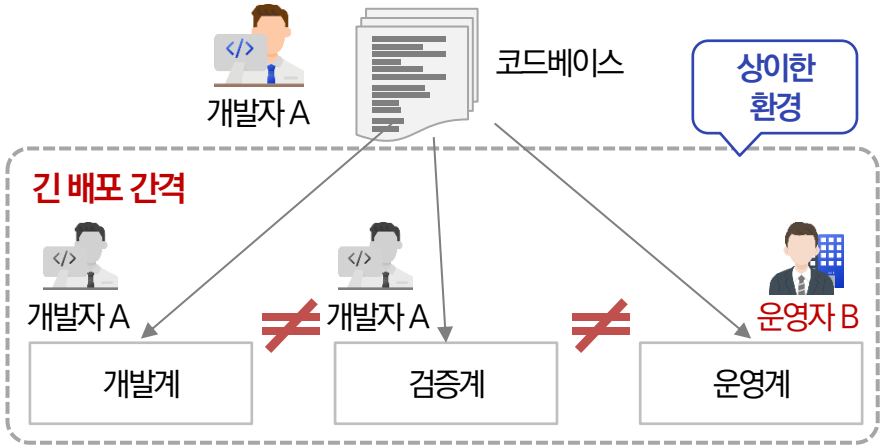
12가지 요소 : 10) 개발/운영 환경 일치

개발/운영 환경

가능한 동일한 개발/검증/운영 환경을 유지하여 장애 최소화 및 지속적 배포가 가능하게 함

AS-IS : 개발/운영 환경

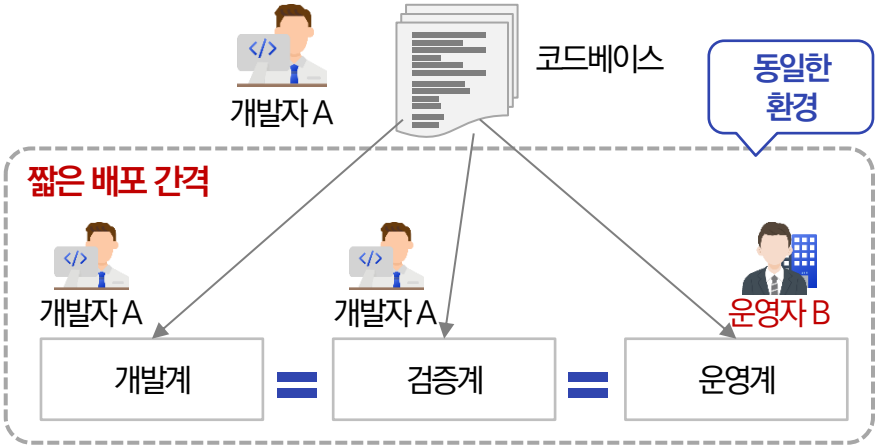
개발환경과 운영환경의 차이



- 코드 개발자와 코드 배포자 상이
  - **운영자가 운영계 배포를 담당**하므로 코드 개발자와 코드 배포자가 다름
- 긴 배포 간격
  - 개발자가 코드 작성 후 개발계, 검증계, 운영계 환경에 반영하기 까지 **수 일에서 수 개월 소요**
- **기존의 개발/운영 환경은 코드 개발자와 배포자가 상이** : 긴 배포 간격, 도구 등의 환경 차이가 존재하여 운영환경에서 장애 발생 가능성이 높음

To-BE : 12 Factor(클라우드 네이티브) 개발/운영 환경

개발, 테스트(검증), 운영 환경을 최대한 동일하게 유지



- 코드 개발자와 코드 배포자 일치
  - 운영자 아니라 코드 **개발자가 직접 배포**
- 짧은 배포 간격
  - 개발자가 코드 작성 후 개발계, 검증계, 운영계 환경에 반영하기 까지, **수 분에서 수 시간 소요**
- 개발 환경과 운영환경의 차이 최소화를 통한 **운영환경에서 예기치 않은 오류/다운타임의 위험성 감소 등**

12가지 요소 : 11) 로그

Code 로그

어플리케이션 로그는 파일이 아닌 로그 스트림 형태로 표준 출력하며, 플랫폼에 의해 로그 스트림이 통합되고 관리됨

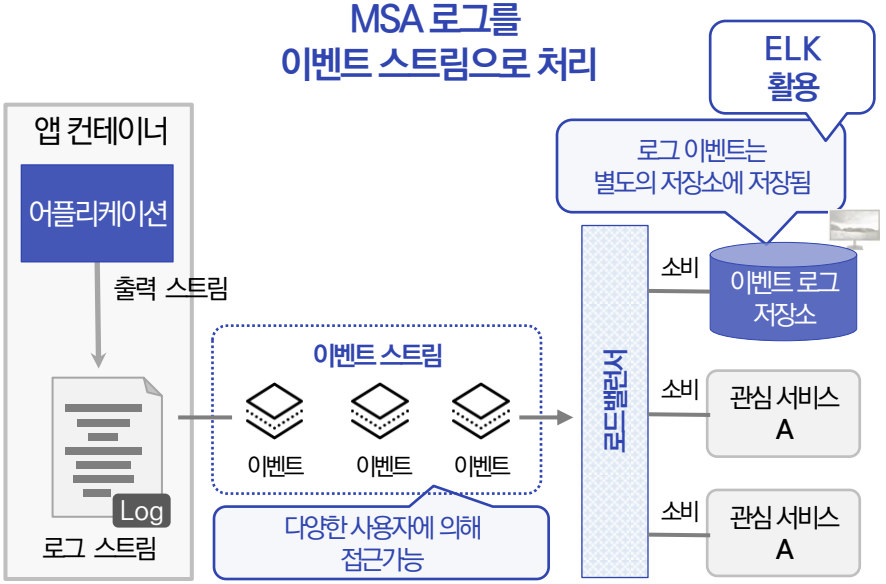
AS-IS : 기존 로그 수집

**로그는 로컬 파일로 저장하며,
로컬 파일을 중앙 저장소로 이동하여 통합하고 분석됨**

- 애플리케이션은 로그를 파일로 저장
- 파일 생성 및 로테이션 조건을 어플리케이션 혹은 서버 내 설정으로 관리
- 로그 분석 체계에 의해 로컬 로그 파일을 중앙 저장소로 복사하여 로그 분석 수행
  (애플리케이션에 대한 로그수집을 위한 프로그램 변경 및 Agent P/G설치 등)

To-BE : ELK기반 로그수집 (Elastic Search, Logstash, Kibana)

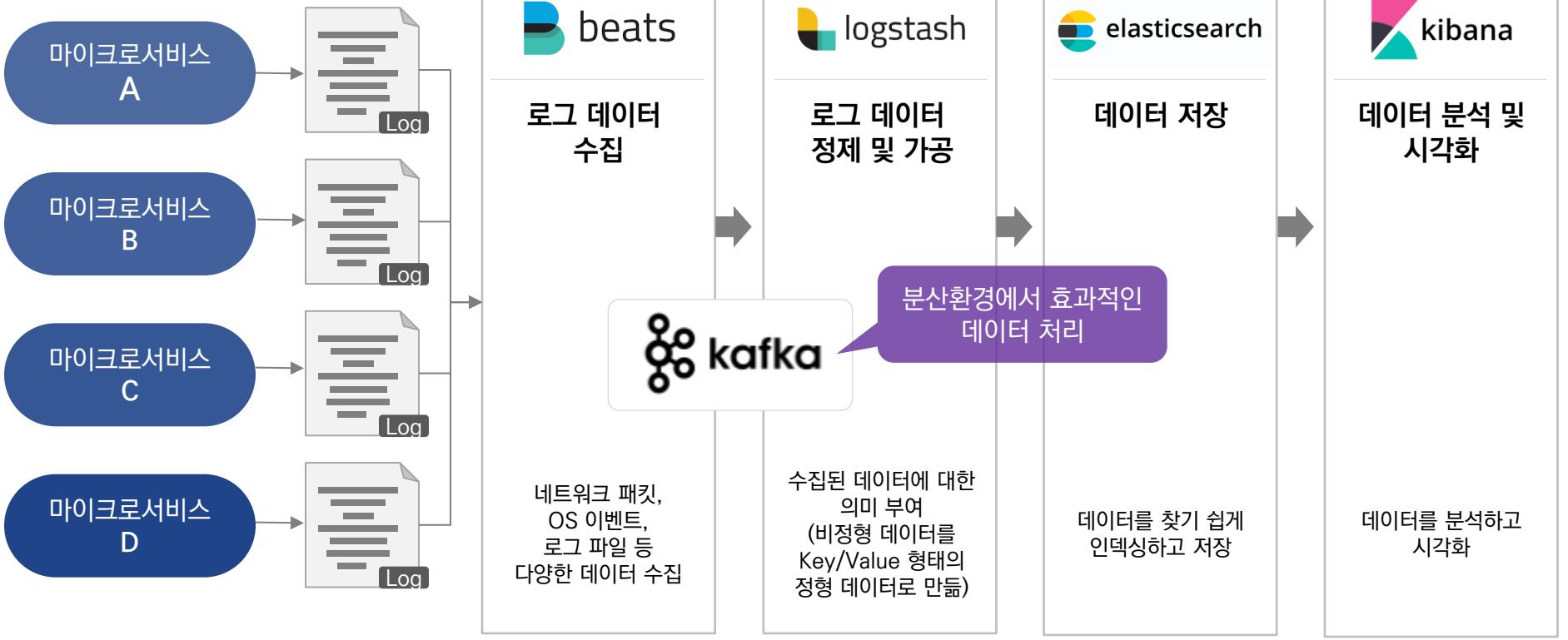
**MSA 로그를
이벤트 스트림으로 처리**



- 로그는 실행중인 모든 프로세스와 서비스의 아웃풋 스트림으로부터 수집된 이벤트가 시간순으로 정렬되는 스트림임
- 로그를 이벤트 스트림으로 취급하여 로그를 로컬에 저장하지 않고, 별도의 저장소에 보관
  – 마이크로서비스는 사용량에 따라 변화하므로 인스턴스가 생성/삭제될 수 있으므로 로컬에 로그 저장 시 로그가 초기화될 수 있음

ELK 스택을 이용한 로그 분석 예시

ELK 스택 = Elastic Search (검색) + Logstash (정제 및 가공) + Kibana (시각화) + beats (로그수집)



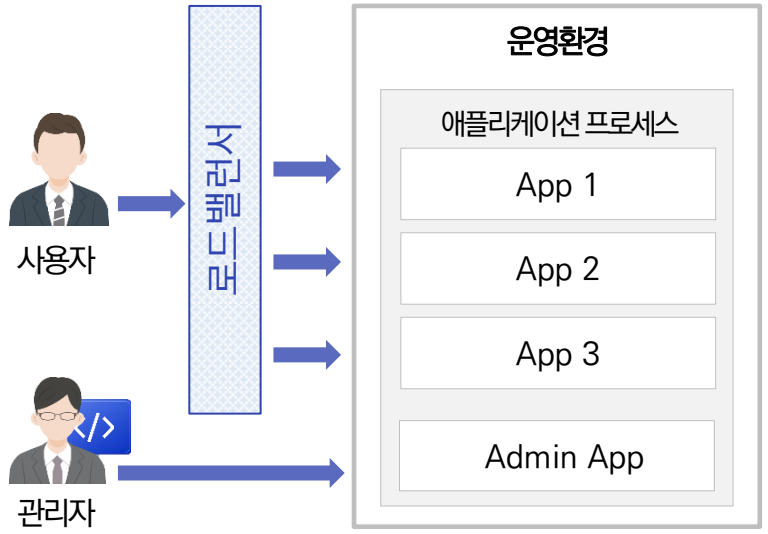
12가지 요소 : 12) 관리 프로세스

관리 프로세스

관리/유지보수 프로세스를 일회성 프로세스로 실행하며, 관리/유지보수 프로세스는 일회성 프로세스로 애플리케이션 프로세스와 분리되어 실행되지만, 애플리케이션과 동일한 빌드/릴리즈/배포 사이클로 실행됨

AS-IS : 프로세스 미분리

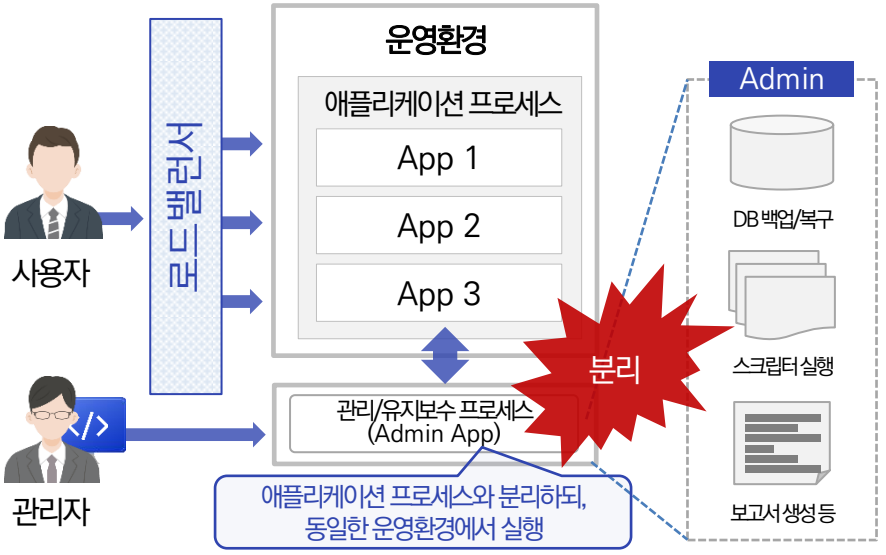
애플리케이션 프로세스와 관리 프로세스 동일환경에서 동작



- 관리/유지보수 프로세스는 애플리케이션 영역과 분리 되어있지 않음
- 관리 프로세스가 운영시스템에 영향을 줄 수 있음

To-BE : 사용자와 관리자 프로세스 분리

관리/유지보수 프로세스는 애플리케이션 영역과 분리되어 동작



- DB 마이그레이션, DB 백업/복구, 스토리지 이동, 스크립트 실행, 보고서 생성 등의 일회성 관리/유지보수 프로세스는 애플리케이션과 분리하되, 어플리케이션과 동일한 환경에서 실행되어야 함
- 일회성 처리를 위한 기능들도 동기화 문제를 해결하기 위하여 코드베이스와 함께 관리하고 빌드/릴리즈/배포 관리되어야 함

루비 및 파이썬의 관리 및 REPL을 활용한 일회성 스크립트 실행 예시

루비 및 파이썬 등의 개발언어 사용 시 관리 프로세스는 웹 프로세스와 같은 애플리케이션이 실행되는 환경과 동일한 곳에서 실행되어야 하고, REPL shell 제공 언어를 이용하여 일회성 관리 프로세스를 실행할 것을 권고함

루비

Ruby

동일한 디렉토리에서 실행

루비 웹 프로세스	$ bundle exec thin start
DB 마이그레이션	$ bundle exec rake db：migrate

파이썬

python

동일한 디렉토리(/bin/python)에서 실행

토네이도 웹 서버 (가상환경)	$ import tornado.web
DB 마이그레이션 (manage.py 사용)	$ python manage.py migrate 〈앱〉

12 Facter App은 별도의 설치나 구성 없이 REPL shell을 제공하는 언어를 선호

REPL

Read	(코드를) 읽고
Eval	(읽은 코드를) 평가(실행) 하고
Print	(실행한 결과를) 출력하는
Loop	루프 (반복)

REPL 은 사용자 입력한 하나의 문장(표현식) 단위로 평가/실행하고 결과를 출력하는 명령줄 셸 및 프로그래밍 언어와 유사한 환경을 의미

- **로컬 배포 시**
  - 개발자는 일회성 관리 프로세스를 어플리케이션을 체크아웃한 디렉토리 안에서 직접 실행
- **운영계 배포 시**
  - 운영계 배포 시에는 SSH나 운영계 환경이 제공하는 다른 원격 도구를 사용해 관리 프로세스를 실행할 수 있음

12가지 요소 적용 – CF vs. Kubernetes PaaS 플랫폼 환경에서 구현

구분		Cloud Foundry	Heroku	Kubernetes
개념		오픈소스 PaaS플랫폼	AWS에서 제공하는 PaaS플랫폼	오픈 소스 컨테이너플랫폼
특징		배포관리, 컨테이너 관리만 보유	빌드관리, 배포관리, 컨테이너 관리 모두 보유	빌드관리, 배포관리, 컨테이너 관리 모두 보유
비교 항목	빌드 관리		빌드관리 (Slug)	빌드관리 (Docker Image)
	배포 관리	배포관리 (Droplet)	배포관리 (Release)	배포관리 (Deployment / StatefulSet)
	컨테이너 관리	컨테이너관리 (Garden)	컨테이너 관리 (Dynos)	컨테이너 관리 (Pod + Docker Container)

12가지 요소 적용 – CF vs. Kubernetes PaaS 플랫폼 환경에서 구현

구분	활용 용도	Cloud Foundry	Kubernetes
1) 코드 베이스	단일 형상관리 기반의 버전 관리 및 배포 관리 자동화 환경 제공	• 어플리케이션의 버전관리를 제공하지 않음(변경 추적을 위한 별도의 버전관리시스템을 적용 필요) • 어플리케이션과 배포환경간 추적관리를 제공하지 않음 (별도 관리를 위한 시스템 구축 필요)	• 전체 시스템을 선언적으로 정의 (Docker 빌드, Deployment 배포) • 기본 Kubernetes는 Git 환경을 포함하고 있지 않음 (변경 추적을 위한 별도의 버전관리 및 GitOps 적용 필요) • GitOps 도구와 통합 서비스 제공 (예 weaveworks gitops)
2) 종속성	의존관계는 명시적으로 선언하고 분리	• 언어별 Buildpack 기본 환경 제공 • 언어별 maven, npm, pip, gem 등 의존 관계 관리	• Dockerfile 이용하여 언어별 기본환경 정의 • 언어별 maven, npm, pip, gem 등 Dockerfile 지원
3) 설정	설정 정보는 환경변수에 저장	$ cf cups myaccess -p '{"uri":"example.com:300"}' $ cf bind-service APP-NAME mysqldb	• ConfigMap, Deployment 내 명시
4) 백앤드 서비스	백엔드 서비스는 리소스 연결로 처리	• Service Broker 연결로 처리	• Service(Cluster IP) 연결로 처리
5) 빌드/ 릴리즈/ 실행	빌드와 실행 단계는 엄격히 분리	• 릴리즈 단계 아티팩트 관리(빌드 단계 관리는 없음) 빌드실행 단계에서 코드 변경 불가	• 빌드 및 릴리즈 단계 아티팩트 분리 빌드 단계 Docker Image 지원 실행 단계 Deployment, StatefulSet 타입의 릴리즈 관리 및 롤백을 지원

12가지 요소 적용 – CF vs. Kubernetes PaaS 플랫폼 환경에서 구현

구분	활용 용도	Cloud Foundry	Kubernetes
6) 프로세스	앱은 Stateless 프로세스로 실행	Stateless 지원 (Web Container는 Session Replication을 지원하지 않음)	Stateless & Stateful 모두 지원
7) 포트 바인딩	웹 어플리케이션은 포트 바인딩을 사용하여 서비스를 공개	공통 라우터에서 L7(HTTP/HTTPS), L4 지원	Ingress 오브젝트 이용하여 L7, L4(TCP) 서비스 지원
8) 동시성	프로세스 유형별 스케일 아웃 지원	단일 배포 유형 제공 설정 파일: manifest.yml 실행: $ cf push –f manifest.yml --- applications: - name: web path: web/build/libs/web.jar instances: 2 - name: worker path: worker/build/libs/worker.jar no-route: true instances: 4	Stateless, Stateful, Job, BatchJob 유형 제공
9) 폐기 가능성	빠른 시작과 정상 종료(graceful shutdown)를 지원하여 안정성을 극대화	기동 시 서비스 Probe 기능 제공(GoRouter Timeout은 300초)	Liveness Probe, Readiness Probe, Graceful Shutdown 기능 제공

12가지 요소 적용 – CF vs. Kubernetes PaaS 플랫폼 환경에서 구현

구분	활용 용도	Cloud Foundry	Kubernetes

10) 개발/실행 환경	가능한 동일한 개발, 스테이징, 운영 환경 유지	$ cf cups mysqldb -p '{"uri":"mysql://root:secret@dbserver.example.com:3306/mydatabase"}'
		$ cf bind-service APP-NAME mysqldb
		소스 코드: Cloud cloud = cloudFactory.getCloud(); DataSource ds = cloud.getServiceConnector("mysqldb", DataSource.class, null /* default config */);
			• Kubernetes 는 서비스간 의존성을 Service 객체로 처리
			• Kubernetes 내에 실행되는 서비스는 ClusterIP 타입 속성의 Service 를 이용하여 하위 서비스와 LB 역할 수행
			• Kubernetes 외에 실행되는 서비스는 ExternalIP 혹은 ExternalName 타입 속성으로 구성하여 외부 서비스와 LB 연결 수행

11) 로그	로그는 이벤트 스트림으로 처리	$ cf logs appname Monitoring/Logging 마켓플레이스	$ kubectl logs podname

12) 관리 프로세스	어드민 및 관리 작업은 일회성 프로세스로 실행	일반 App 처리
		$ cf ssh APP --no-route
		$ cf push APP -c 'rake db:migrate' -i 1
			Job, BatchJob 처리

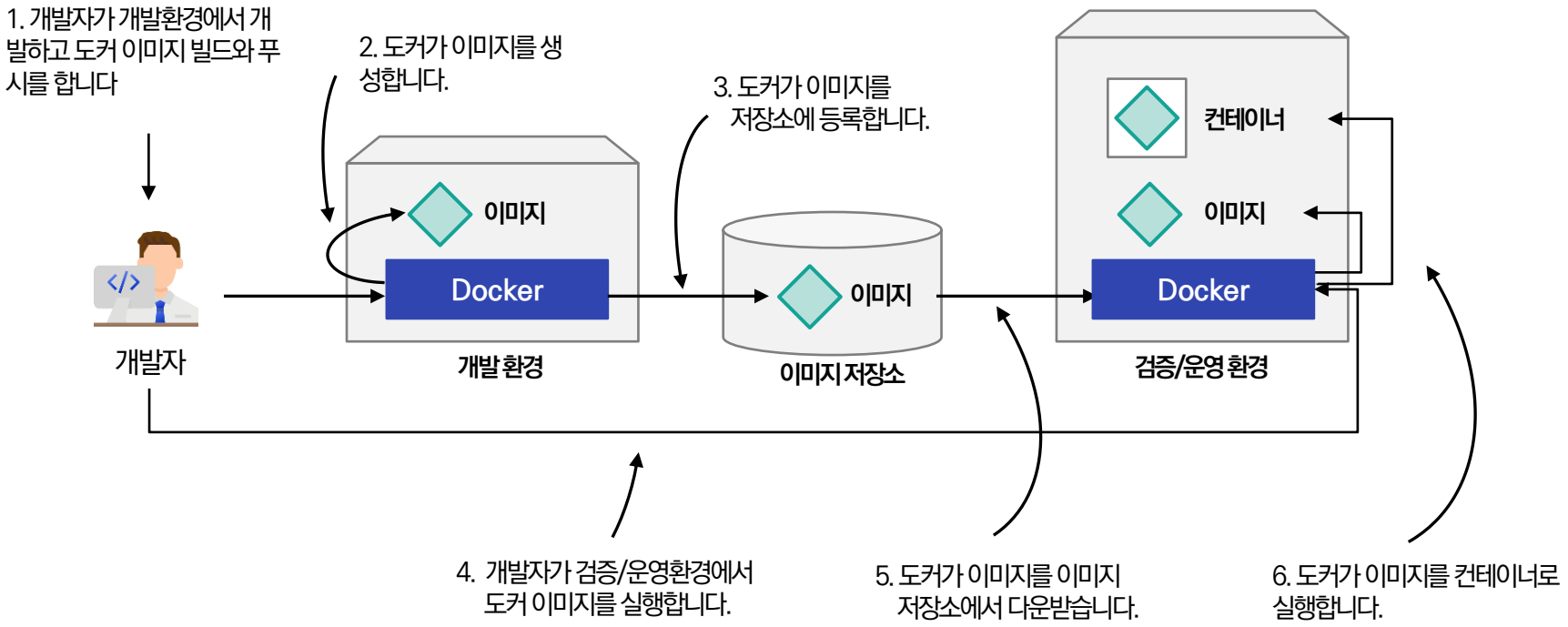
Kubernetes Native 어플리케이션 환경 소개

Docker 이미지	• 개발 바이너리와 환경 설정을 패키징 • 파일시스템, 실행 파일 경로와 같은 메타데이터 포함 • Dockerfile 로 정의
이미지 저장소 (Registry)	• Docker 이미지 공유를 위한 저장소 • 개발자가 Docker 이미지를 업로드(Push)하고, Docker 실행 시 다운로드(Pull) • 공개 Docker 저장소(hub.docker.com)와 사설 Docker 저장소 이용 가능
컨테이너 실행환경	• Docker 이미지를 기반으로 실행되는 일반 Linux 컨테이너 • 컨테이너는 Docker 실행 호스트에서 실행되는 하나의 프로세스 • 컨테이너는 호스트 및 타 프로세스와 격리

```
$ cat Dockerfile
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

Dcoker 기술 Docker는 이미지 빌드, 이미지 저장소, 컨테이너 실행환경으로 구성

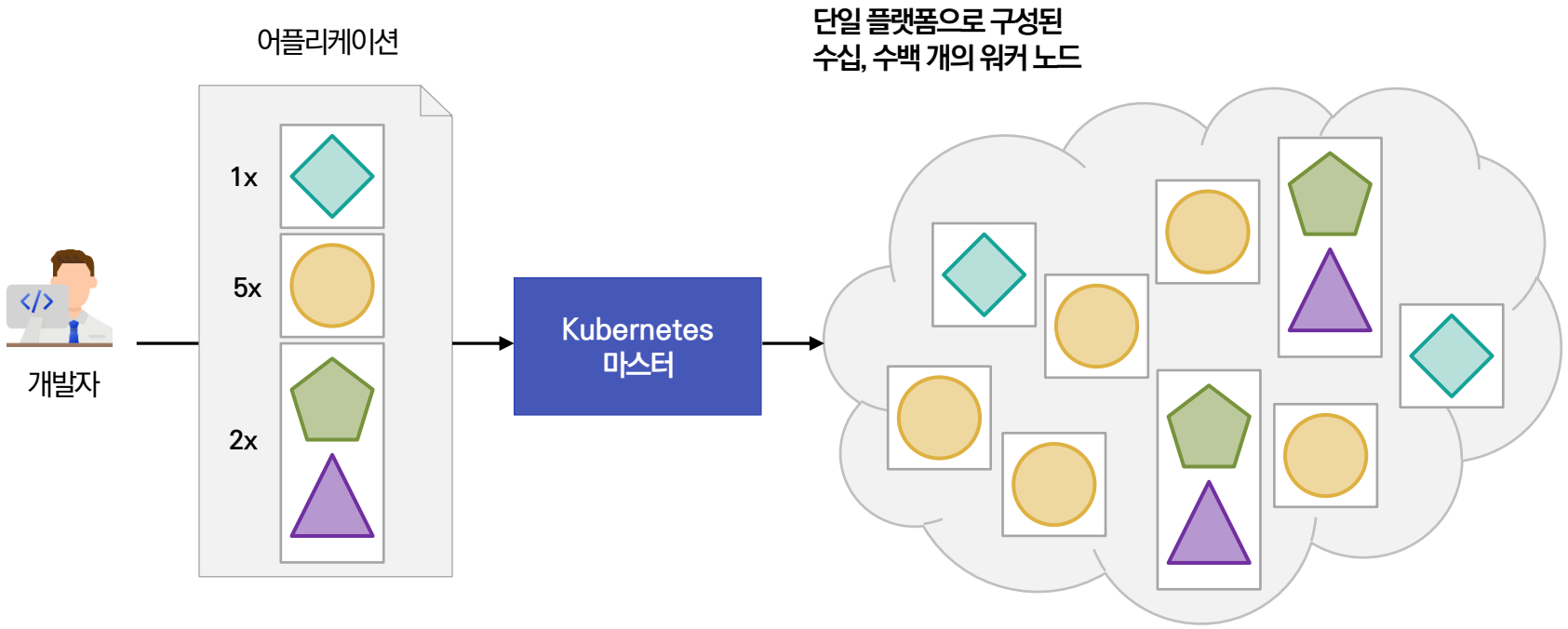
Kubernetes Native 어플리케이션 환경 소개



Dcoker 기술

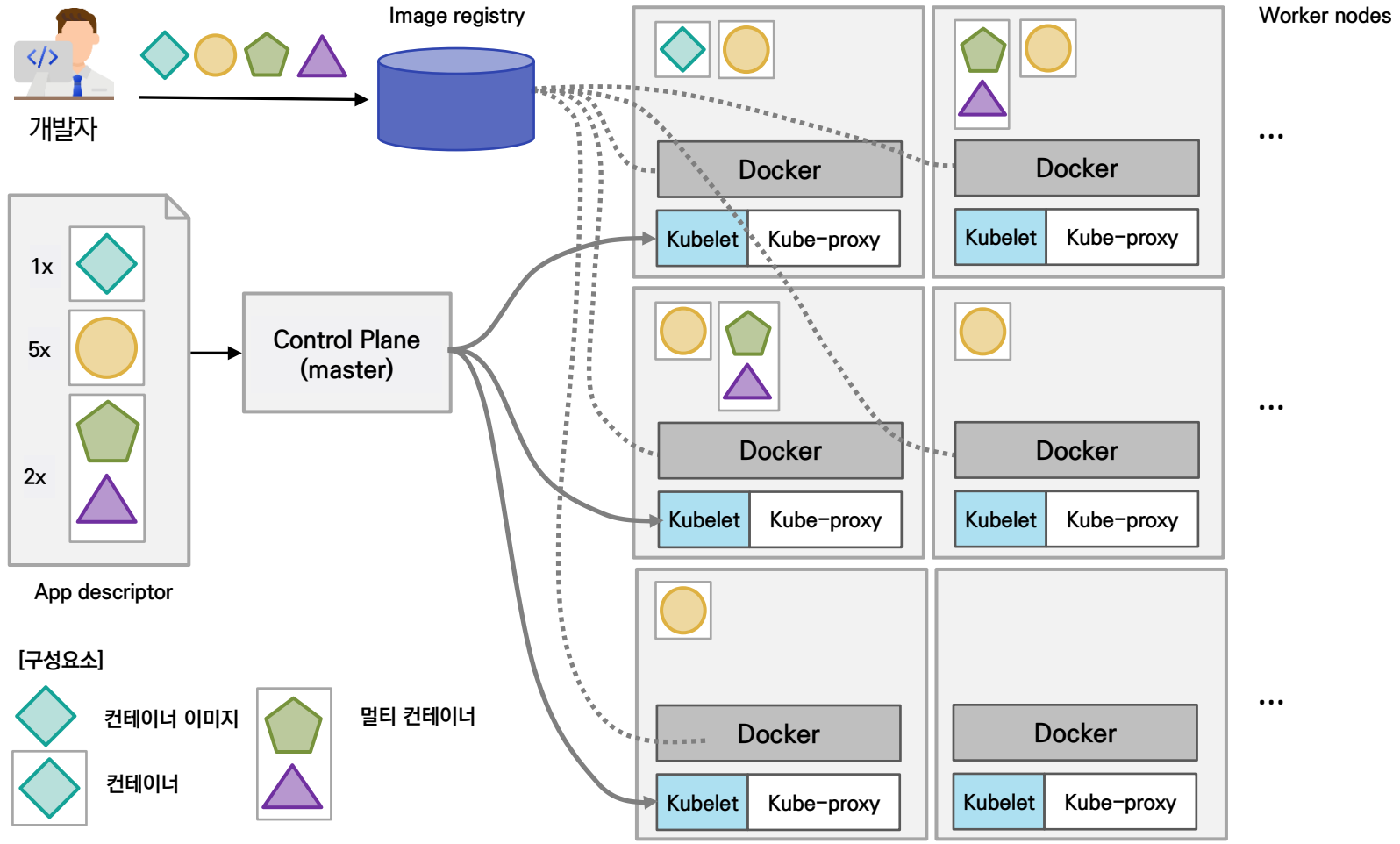
Docker는 이미지 빌드, 이미지 저장소, 컨테이너 실행환경으로 구성

Kubernetes Native 어플리케이션 환경 소개



Kubernetes Kubernetes 는 복수의 Docker 실행환경을 통합 관리하는 PaaS 표준 플랫폼

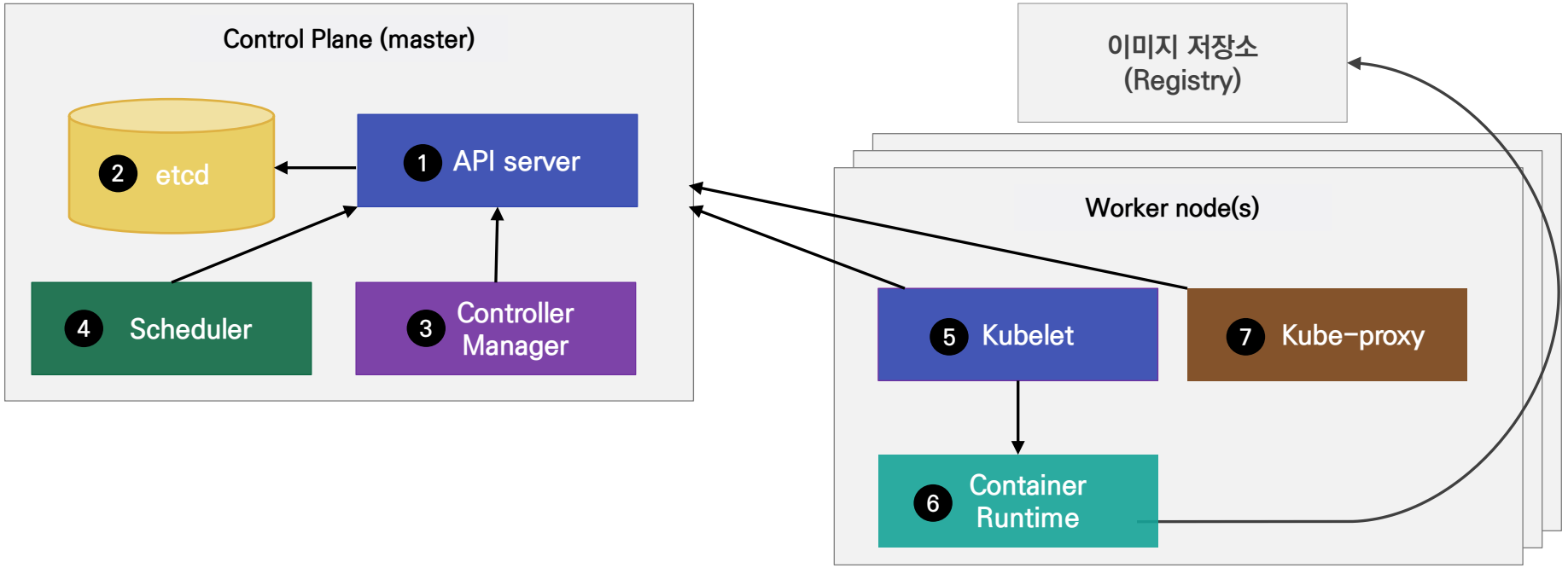
Kubernetes Native 어플리케이션 환경 소개



Kubernetes

Kubernetes는 Master와 Worker 노드로 구성, 사용자 정의 앱의 배포와 운영을 수행

Kubernetes Native 어플리케이션 환경 소개



1. API Server : 클라이언트에 API 진입점을 제공하고 K8S 개체를 관리
2. etcd : 클러스터 정보를 저장하는 고가용성 키-값 저장소
3. Controller Manager : Pod 및 Service 생성 요청을 감시하고, 생성하는 역할
4. Scheduler : 노드가 배정되지 않은 Pod를 감지하고 노드를 지정
5. Kubelet : 노드에 할당된 Pod, 볼륨 관리
6. Container Runtime : Docker 이미지 다운받아 실행
7. Kube-proxy : Pod 접속을 위한 Service 네트워크 처리

Kubernetes Kubernetes는 Master와 Worker 노드로 구성, 사용자 정의 앱의 배포와 운영을 수행

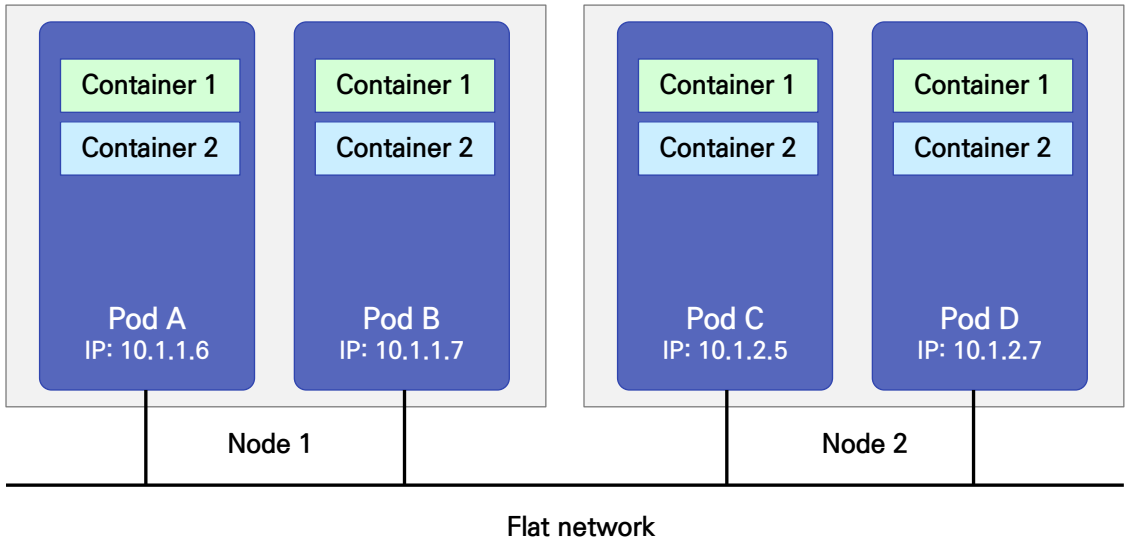
Kubernetes Native 어플리케이션 환경 소개

리소스	용도	리소스	용도
Node	컨테이너가 배치되는 서버	Storage Class	퍼시스턴트 볼륨이 확보하는 스토리지의 종류를 정의
Namespace	쿠버네티스 클러스터 안의 가상 클러스터	StatefulSet	상태를 유지하는 파드를 생성 관리
Pod	컨테이너 집합 중 가장 작은 단위, 컨테이너의 실행 방법을 정의	Job	상주 실행을 목적으로 하지 않는 파드를 생성하고 종료
ReplicaSet	같은 스펙을 갖는 파드를 여러 개 생성하고 관리하는 역할	CronJob	Cron 문법으로 스케쥴링되는 잡
Deployment	레플리카 세트의 리비전을 관리	Secret	인증 정보 같은 기밀 데이터를 정의
Service	파드의 집합에 접근하기 위한 경로 정의	Role	네임스페이스 안에서 조작 가능한 쿠버네티스 리소스의 규칙을 정의
Ingress	서비스를 쿠버네티스 클러스터 외부로 노출	RoleBinding	쿠버네티스 리소스 사용자와 롤을 연결
ConfigMap	설정 정보를 정의하고 파드에 전달	ClusterRole	클러스터 전체에서 조작 가능한 쿠버네티스 리소스의 규칙을 정의
Persistent Volume	파드가 사용할 스토리지의 크기 및 종류를 정의	ClusterRole Binding	쿠버네티스 리소스 사용자와 클러스터롤을 연결
Persistent VolumeClaim	퍼시스턴트 볼륨을 동적으로 확보	Service Account	파드가 쿠버네티스 리소스를 조작할 때 사용하는 계정

Kubernetes

Kubernetes 제공 리소스 유형

Kubernetes Native 어플리케이션 환경 소개



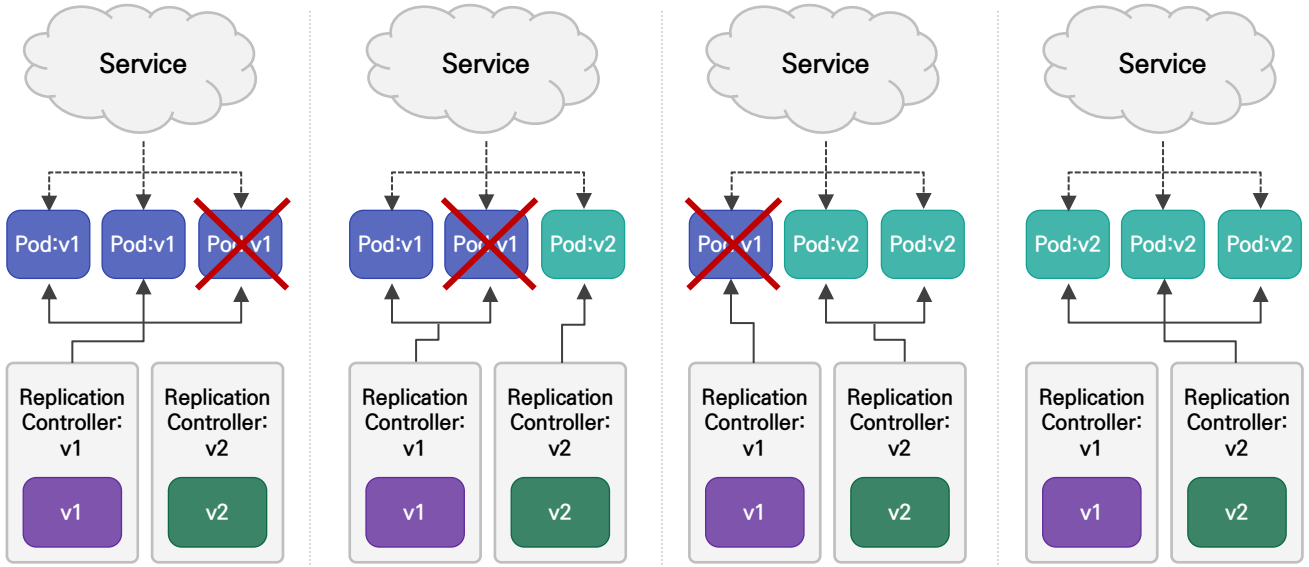

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
```

```
$ kubectl create -f kubia-manual.yaml
pod "kubia-manual" created
```

Pod

Pod 는 컨테이너를 포함하는 배포 단위로, Pod내 컨테이너는 IP와 볼륨을 공유

Kubernetes Native 어플리케이션 환경 소개

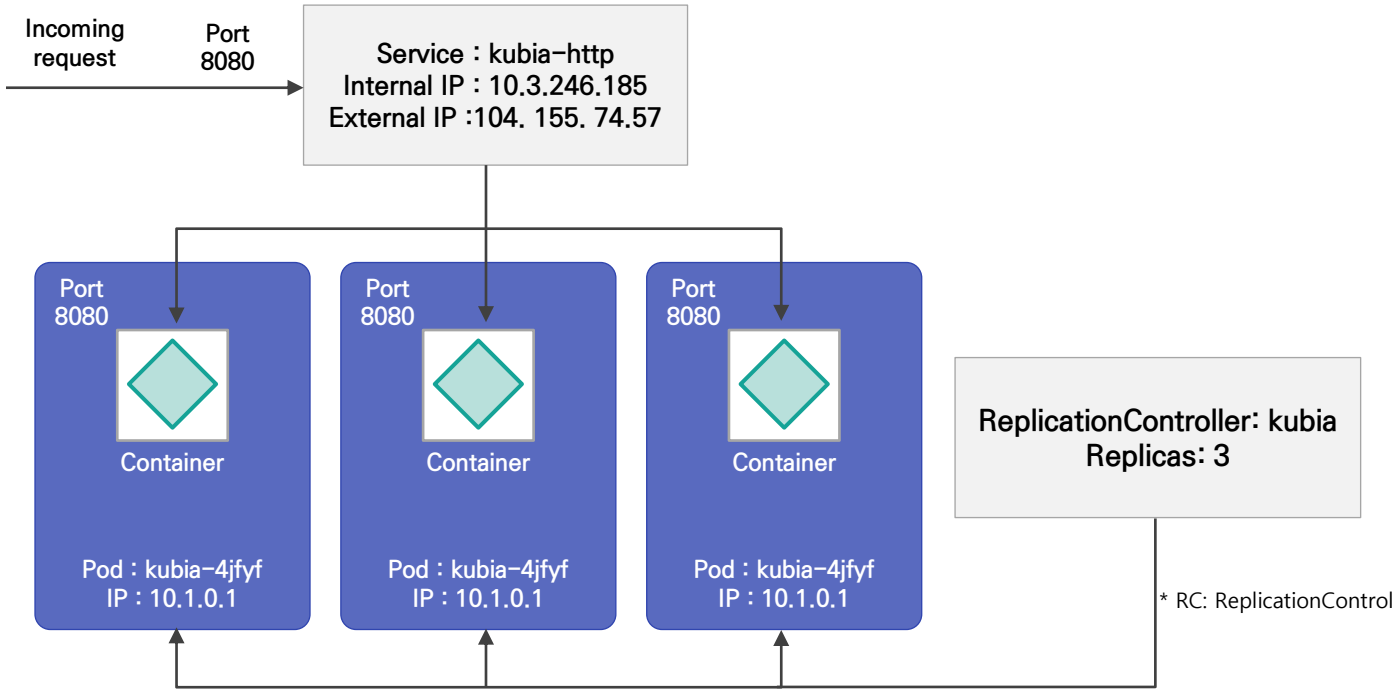



```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  template:
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:
      - image: luksa/kubia:v1
        name: nodejs
```

Deployment

Deployment는 선언적인 Pod 수 관리와 롤링 업데이트 관리를 지원

Kubernetes Native 어플리케이션 환경 소개



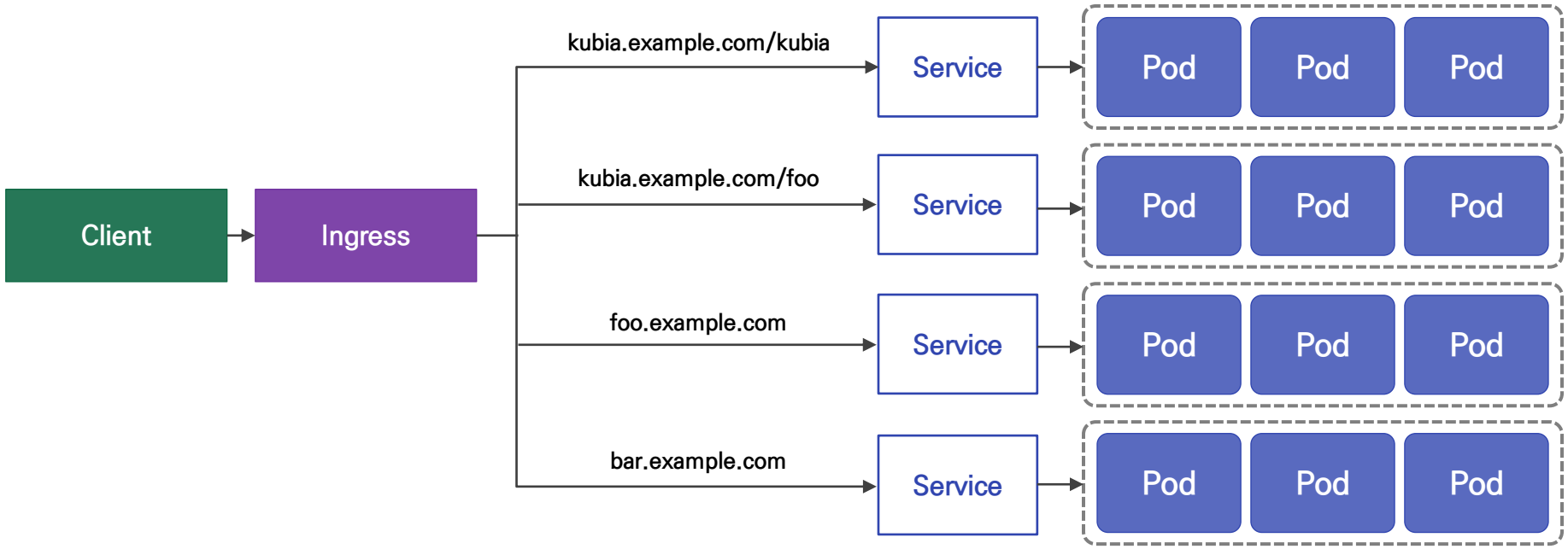

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

* RC: ReplicationController

Service

Service는 복수의 Pod 에 라우팅하며, IP와 Port를 가짐. RC*는 Pod 수를 관리

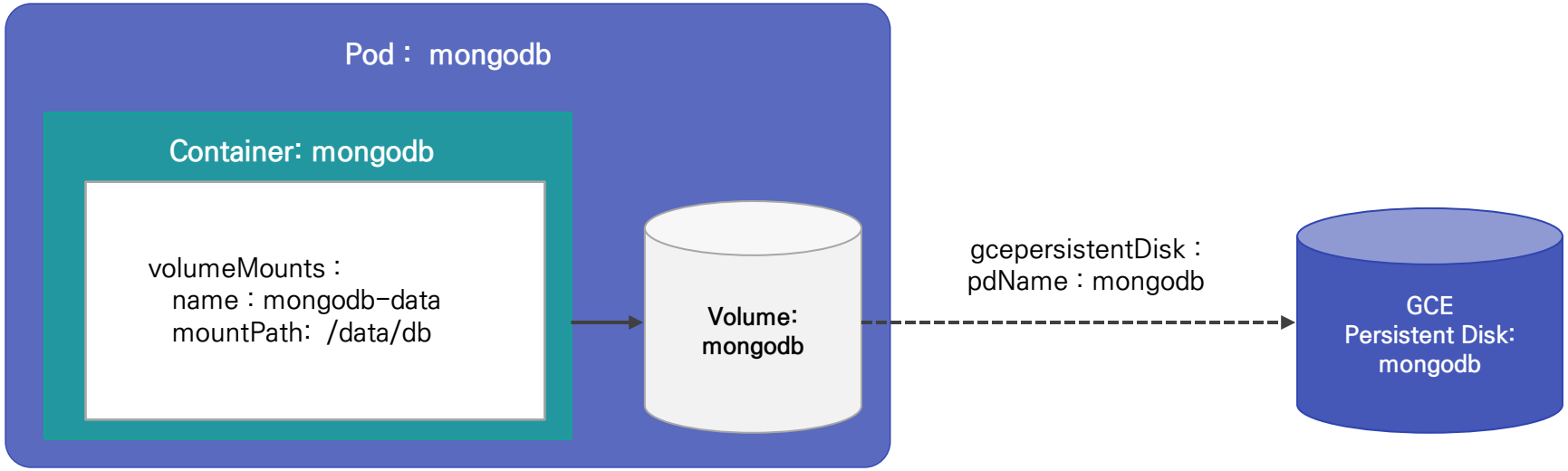
Kubernetes Native 어플리케이션 환경 소개



Ingress

Ingress 는 대외에서 Service 접근을 허용하는 L7 로드밸런서 역할

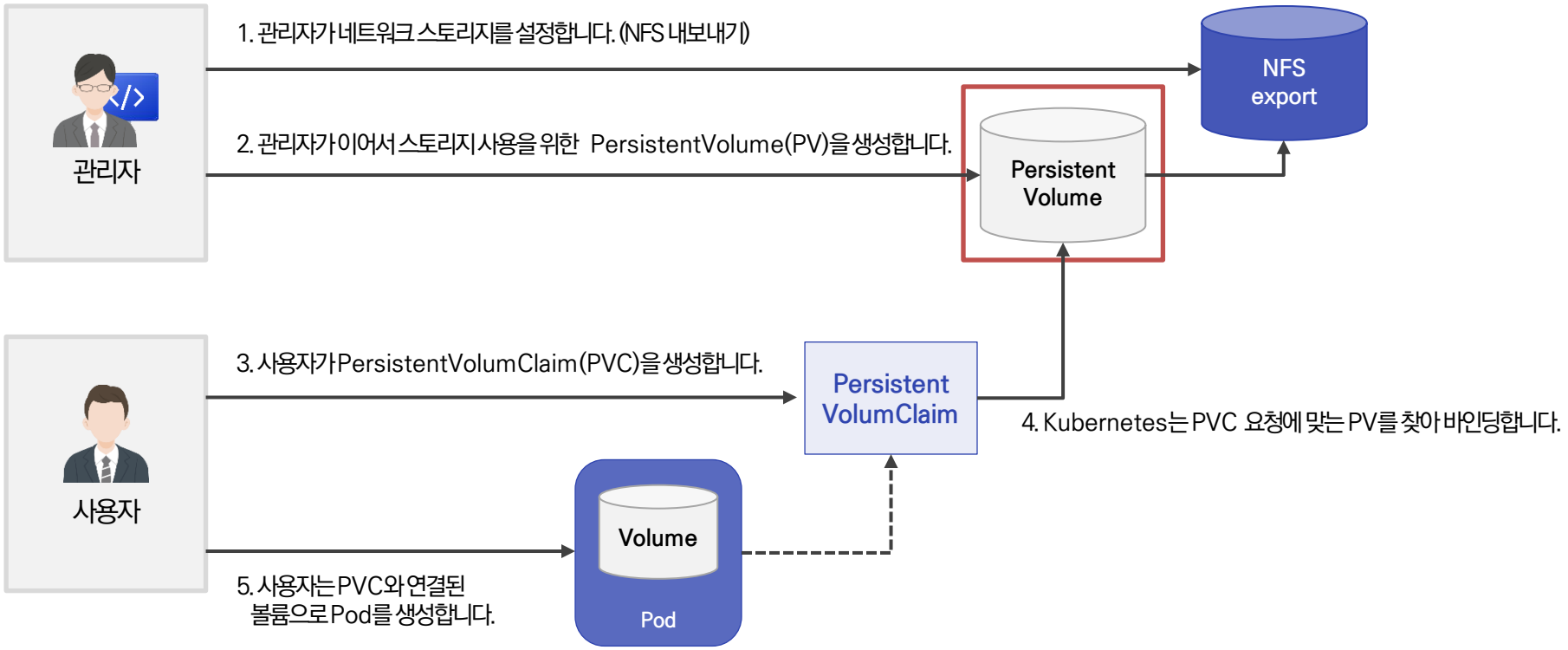
Kubernetes Native 어플리케이션 환경 소개



Persistent Volume

PersistentVolume 은 Pod 내 데이터 영속성을 유지하기 위해 사용

Kubernetes Native 어플리케이션 환경 소개

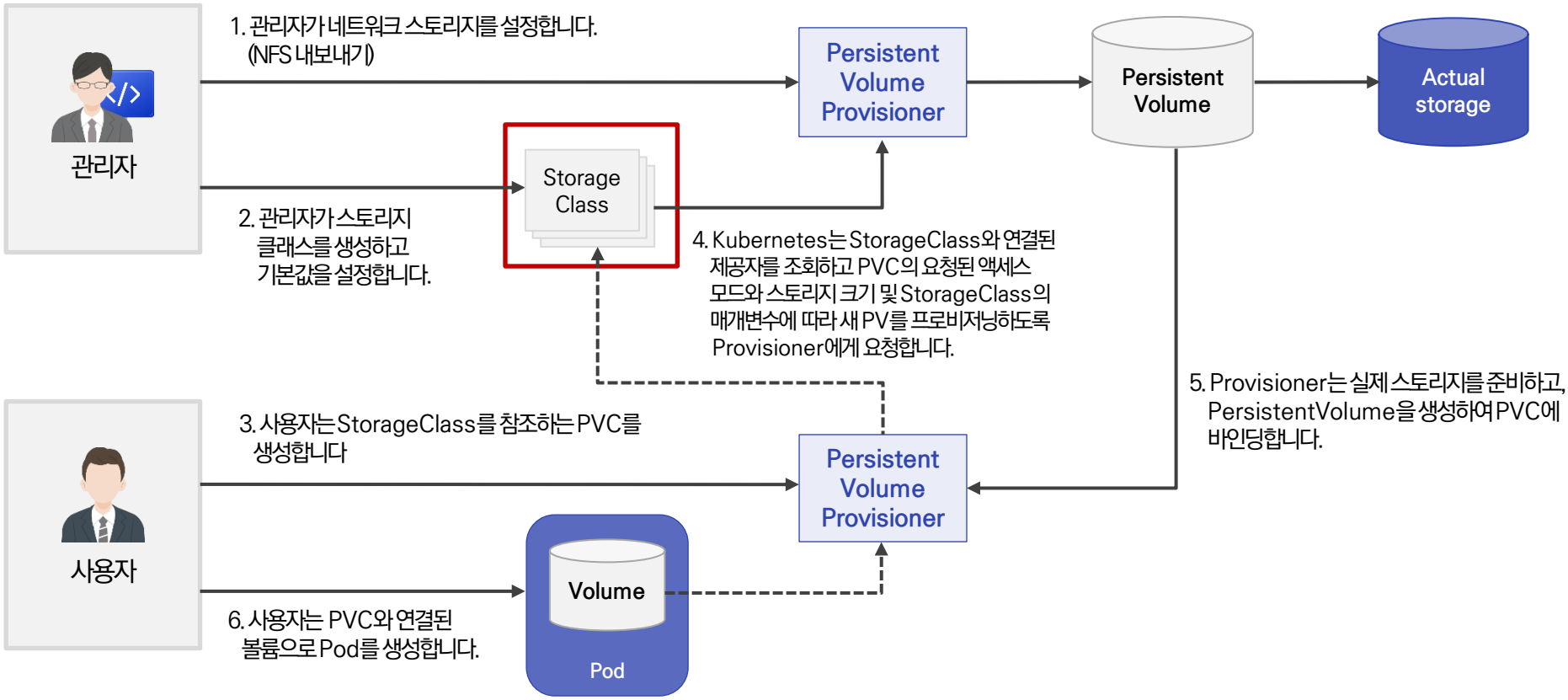


정적 볼륨 생성 예시

Persistent Volume

PersistentVolume 은 Pod 내 데이터 영속성을 유지하기 위해 사용

Kubernetes Native 어플리케이션 환경 소개

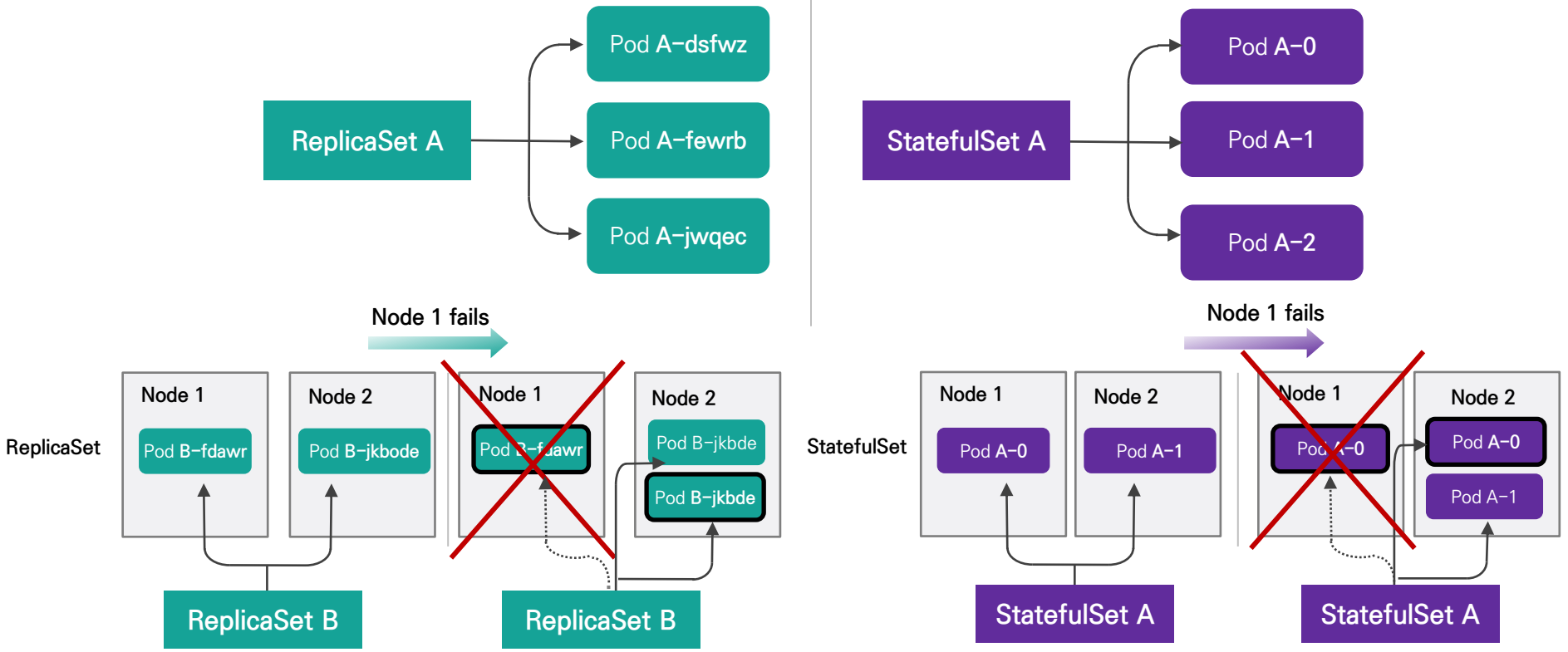


동적 볼륨 생성 예시

Persistent Volume

관리자는 StorageClass를 미리 생성하고, 사용자는 PVC 이용 Pod에 볼륨 할당 사용

Kubernetes Native 어플리케이션 환경 소개

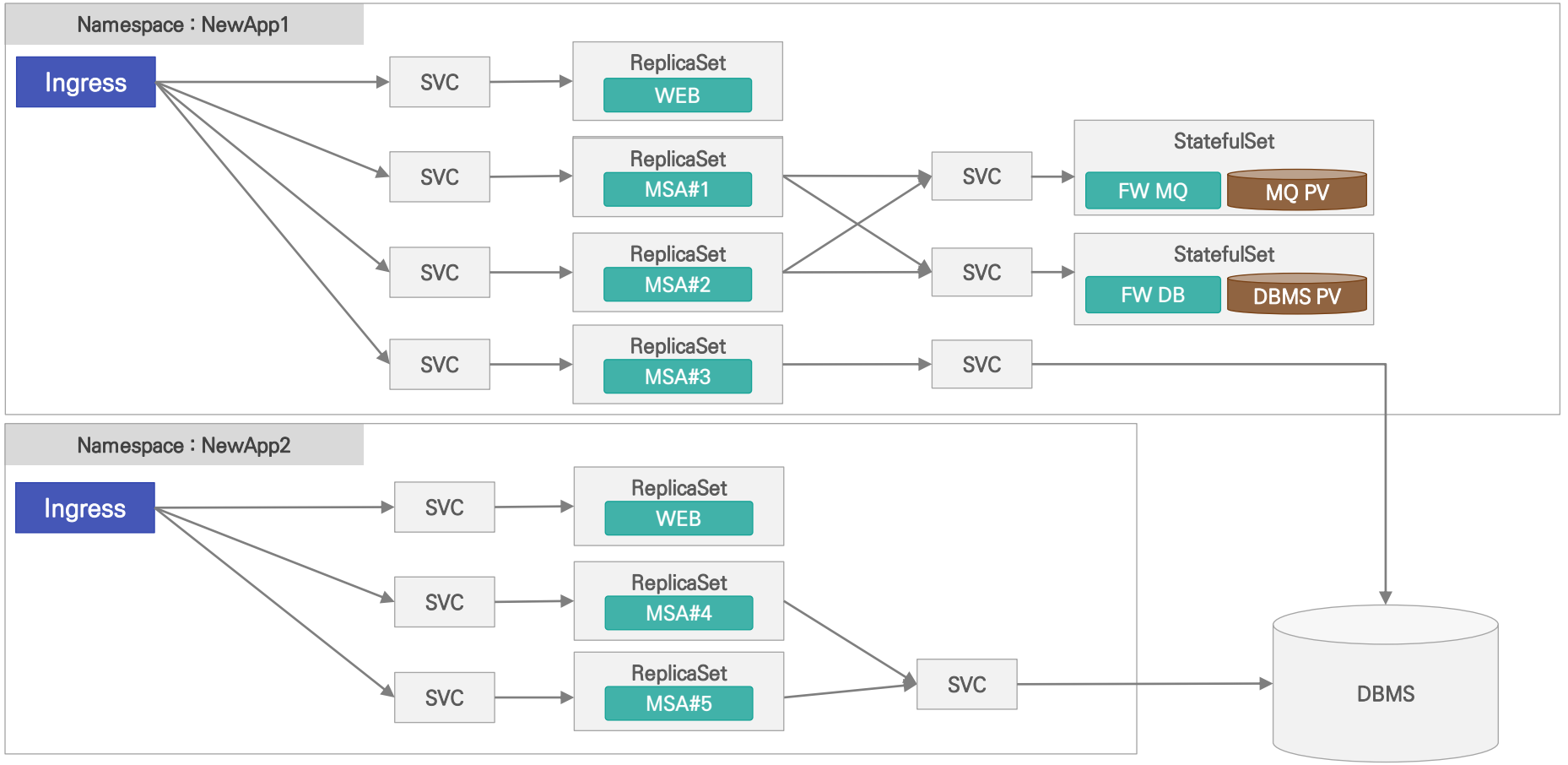


ReplicaSet은 Cattle, StatefulSet 은 Pet 과 같은 존재

ReplicaSet

ReplicaSet은 웹과 같은 비상태 서비스, StatefulSet은 DB와 같은 상태관리 서비스

Kubernetes Native 어플리케이션 환경 소개



* ReplicaSet은 Deployment로 배포

ReplicaSet Kubernetes 어플리케이션 배포 예시

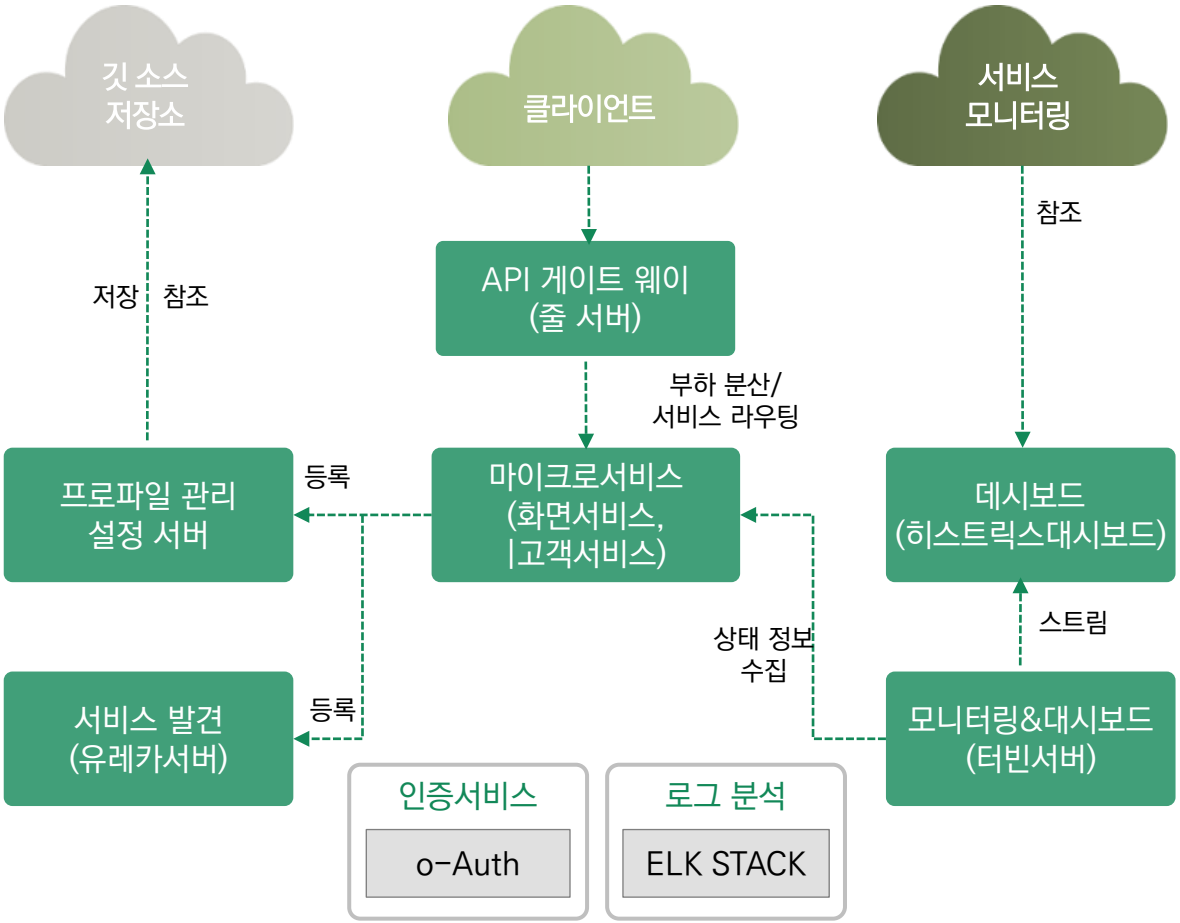
클라우드 네이티브 기반 행정·공공 서비스 확산 지원
클라우드 네이티브 발주자 가이드

클라우드 네이티브 애플리케이션의 구현 및 운영(배포)은 어떻게 되는가?

NIA 한국지능정보사회진흥원

스프링 클라우드 기반의 마이크로서비스 아키텍처 개발 환경 구성

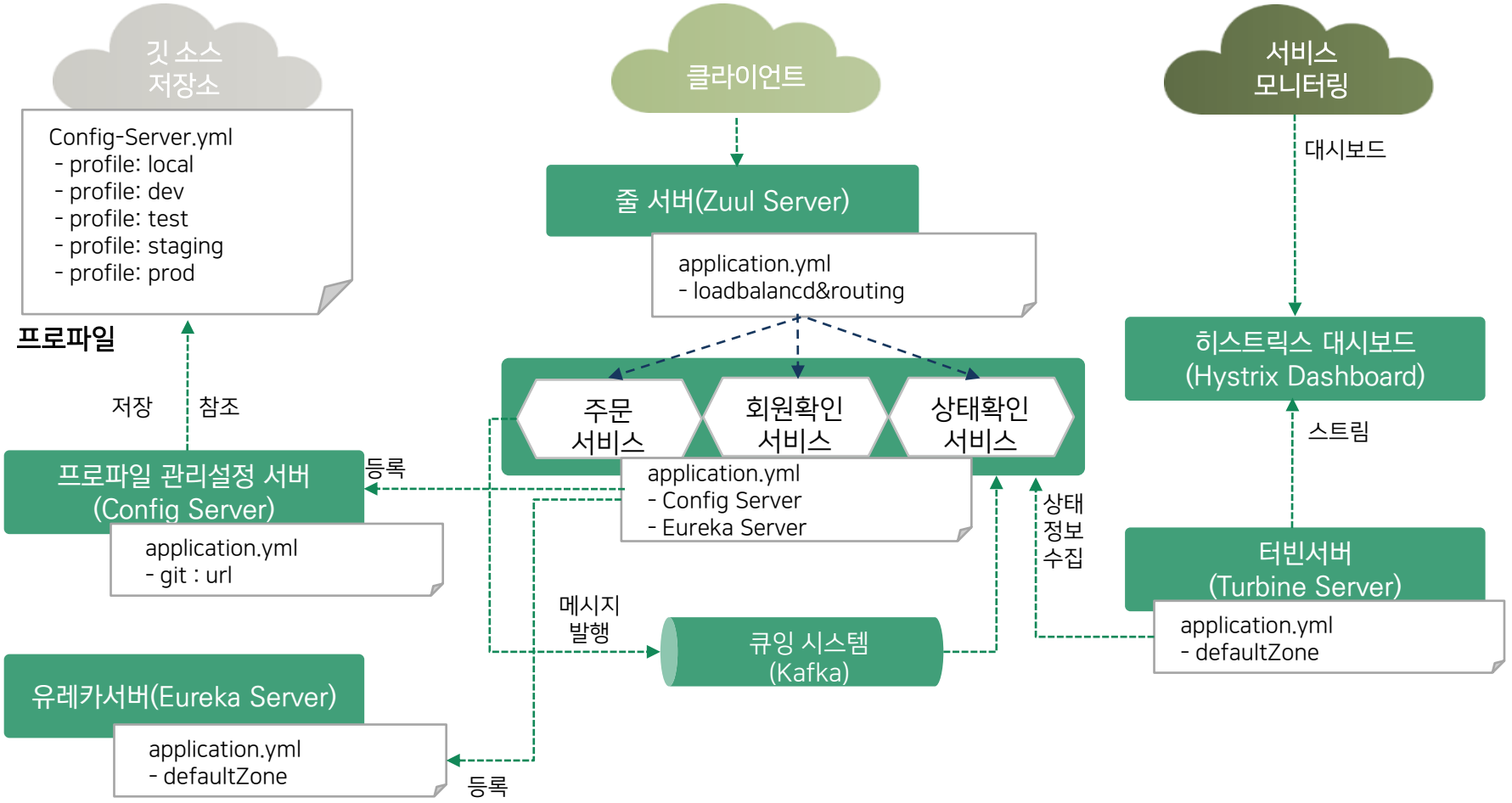
스프링 클라우드에서 제공하는 넷플릭스 오픈소스인 줄(Zuul), 유레카(Eureka)와 터빈(Turbine), 히스트릭스 대시보드(Hystrix Dashboard) 등의 라이브러리를 사용하여 마이크로서비스 개발환경 구성



구분	구성요소	설명
소스 저장소	깃(Git) 소스 저장소	• 소스 뿐만 아니라 각각의 마이크로서비스가 사용할 프로카일 정보를 파일로 관리할 수 있는 공간
서비스 관리	설정 (config) 서버	• 깃 저장소에 저장된 프로파일 정보를 읽어들여 마이크로서비스가 필요할 때 사용할 수 있도록 서비스 제공
	유레카 (Eureka) 서버	• 마이크로서비스의 기동 유무에 관한 정보를 관리하여 마이크로서비스가 등록되거나 삭제될 때 자동으로 감지
서비스 게이트 웨이	줄(Zuul) 서버	• 클라이언트의 서비스 요청을 적절히 분산시키고, 요청한 서비스가 실행될 수 있도록 서비스 라우팅 수행
스트림 처리	터빈 (Turbine) 서버	• 분산된 마이크로서비스에서 생성하는 서비스 응답 상태 스트림 메시지를 한군데로 수집
	히스트릭스 (Hystrix) 대시보드	• 터빈에서 보내는 분산 마이크로서비스의 스트림 메시지를 대시보드에 시각적으로 보여줌

스프링 클라우드 기반의 마이크로서비스 아키텍처 개발 환경 구성

스프링 클라우드 아키텍처를 참조모델로 하여 아키텍처를 구성함으로써 마이크로서비스와 에코시스템 간 전체적인 동작 흐름



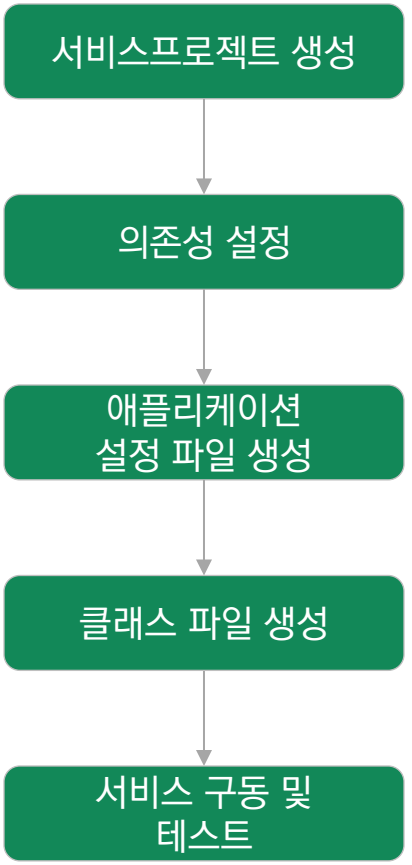
스프링 클라우드 기반의 마이크로서비스 제작

클라우드 네이티브 정보시스템 구현 과정을 소개하기 위해 스프링 부트 기반으로
마이크로 서비스를 제 작(디스플레이를 담당하는 '카탈로그 서비스', '고객 서비스')

마이크로 서비스 구성 (예시)

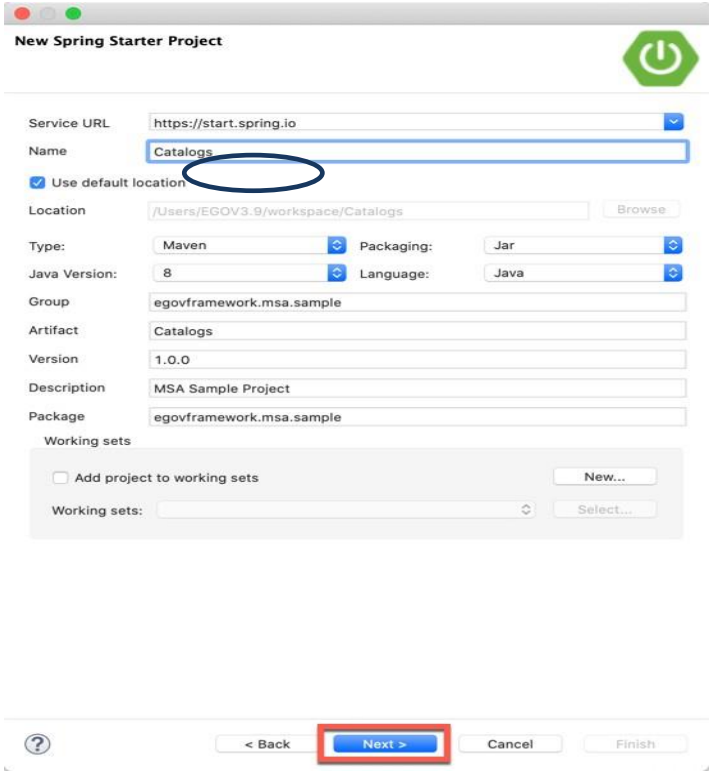
구분	카탈로그 서비스	고객 서비스
서비스	Catalogs	Customers
요청(Request)	/catalogs/{customerId}	/customers/customerId
응답(Response)	String Type (JSON)	String Type (JSON)
서비스 설명	• 상품의 전시 및 화면에 대한 로직을 처리하는 서비스 • 별도의 데이터를 가지고 있지 않지만 사용자의 입력 및 출력에 대한 서비스 제공 • 요청에 따라 각각의 서비스를 호출하여 요청에 응답	• 고객 정보를 조회할 수 있는 서비스 • 요청에 따라 고객정보를 반환 • 실제 데이터베이스를 활용한 데이터 입출력을 제외 • 향후 DAO 및 DataAccess에 해당하는 로직을 별도로 구현

마이크로서비스 제작 과정 (예시)



스프링 클라우드 기반의 마이크로서비스 제작

화면 레이어를 담당하는 카탈로그(Catalogs) 서비스를
전자정부 표준프레임워크 개발환경을 활용하여 생성 (표준 프레임워크 v3.10 기준)



- Service URL : https://start.spring.io
- Use default location : 체크
  (기본 프로젝트 경로 변경을 원하면 해제 후 지정)
- Type : Maven Packaging : Jar
- Java Version : 8
- Language : Java
- Group : egovframework.msa.sample
- **Artifact : Catalogs**
- Version : 1.0.0
- Description : MSA Sample Project Group Id : egovframework.msa.sample

① New > Project > Spring Boot > Spring Starter Project 를 선택 후 아래와 같이 입력한 후Next 선택
② Next > Finish 또는 Finish 를 바로 선택하여 프로젝트 생성

스프링 클라우드 기반의 마이크로서비스 제작

카탈로그 서비스 프로젝트 생성 및 카달로그 서비스의 의존성 설정

카탈로그 서비스 프로젝트 생성

SW 명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	Resource 파일	SpringBoot 설정 파일
CatalogsApplication.java	egovframework.msa.sample	클래스 파일	애플리케이션 구동 파일
CatalogsController.java	egovframework.msa.sample.controller	컨트롤러 클래스 파일	Restful Api Controller
CustomerApiService.java	egovframework.msa.sample.service	**인터페이스 파일**	
CustomerApiServiceImpl.java	egovframework.msa.sample.serviceImpl	**클래스 파일**	

카탈로그 서비스의 의존성 설정 (Pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
        <relativePath />
    </parent>
    <groupId>egovframework.msa.sample</groupId>
    <artifactId>Catalogs</artifactId>
    <version>1.0.0</version>
    <name>Catalogs</name>
    <description>MSA Sample Project</description>

 <dependencies>
     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter</artifactId>
         <exclusions>
             <exclusion>
                 <groupId>org.springframework.boot</groupId>
                 <artifactId>spring-boot-starter-logging</artifactId>
             </exclusion>
         </exclusions>
     </dependency>

     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <scope>test</scope>
         <exclusions>
             <exclusion>
                 <groupId>org.junit.vintage</groupId>
                 <artifactId>junit-vintage-engine</artifactId>
             </exclusion>
         </exclusions>
     </dependency>

     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
```

의존성 추가

※ 프레임워크 의존성 설정 부분 생략

스프링 클라우드 기반의 마이크로서비스 제작

스프링 부트 및 프레임워크의 의존성을 추가한 후,
애플리케이션 설정을 위한 application.yml 파일 생성

application.yml 파일은 /src/main/resources 디렉토리에 위치하며,
yml(yaml) 파일 대신 properties 형태의 파일 사용
Application.yml 파일 소스에 카탈로그 서비스의 이름과 contextPath 및 접속 포트를 설정

MSA 애플리케이션의 구조 파일들을
모두 작성하였으면,
다음으로 실제 로직을
담고 있는 각각의 클래스 파일들을 작성

application.yml 파일 소스 내용

```yaml
server: port:
  8081
spring:
  application:
    name: catalog
```

CatalogsApplication.java 파일 작성

```java
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan("egovframework.*")
@SpringBootApplication
public class CatalogsApplication {

    public static void main(String[] args) {
        SpringApplication.run(CatalogsApplication.class);
    }

}
```

스프링 클라우드 기반의 마이크로서비스 제작

애플리케이션 구현을 위한 CatalogsController.java, CustomerApiService.java,
CustomerApiServiceImpl.java 파일 작성 및 생성

CatalogsController.java 파일 작성 파일 소스 내용

```java
package egovframework.msa.sample.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import egovframework.msa.sample.service.CustomerApiService;

@RestController
@RequestMapping("/catalogs/customerinfo")
public class CatalogsController {

    @Autowired
    private CustomerApiService customerApiService;

    @GetMapping(path = "/{customerId}")
    public String getCustomerInfo(@PathVariable String customerId) {
        String customerInfo = customerApiService.getCustomerDetail(customerId);
        System.out.println("response customerInfo : " + customerInfo);

        return String.format("[Customer id = %s at %s %s ]", customerId,
System.currentTimeMillis(), customerInfo);
    }

}
```

```java
package egovframework.msa.sample.service;

public interface CustomerApiService {
    String getCustomerDetail(String customerId);
}
```

```java
package egovframework.msa.sample.serviceImpl;

import org.springframework.stereotype.Service;
import egovframework.msa.sample.service.CustomerApiService;

@Service
public class CustomerApiServiceImpl implements CustomerApiService {

    @Override
    public String getCustomerDetail(String customerId) {
        return customerId;
    }

}
```

스프링 클라우드 기반의 마이크로서비스 제작

카탈로그의 CatalogsApplication.java 파일을 자바 애플리케이션으로 실행하면,
스프링 부트를 통하여 임베디드 톰캣으로 카달로그 서비스 구동 테스트 확인

카탈로그 서비스 구동 및 테스트

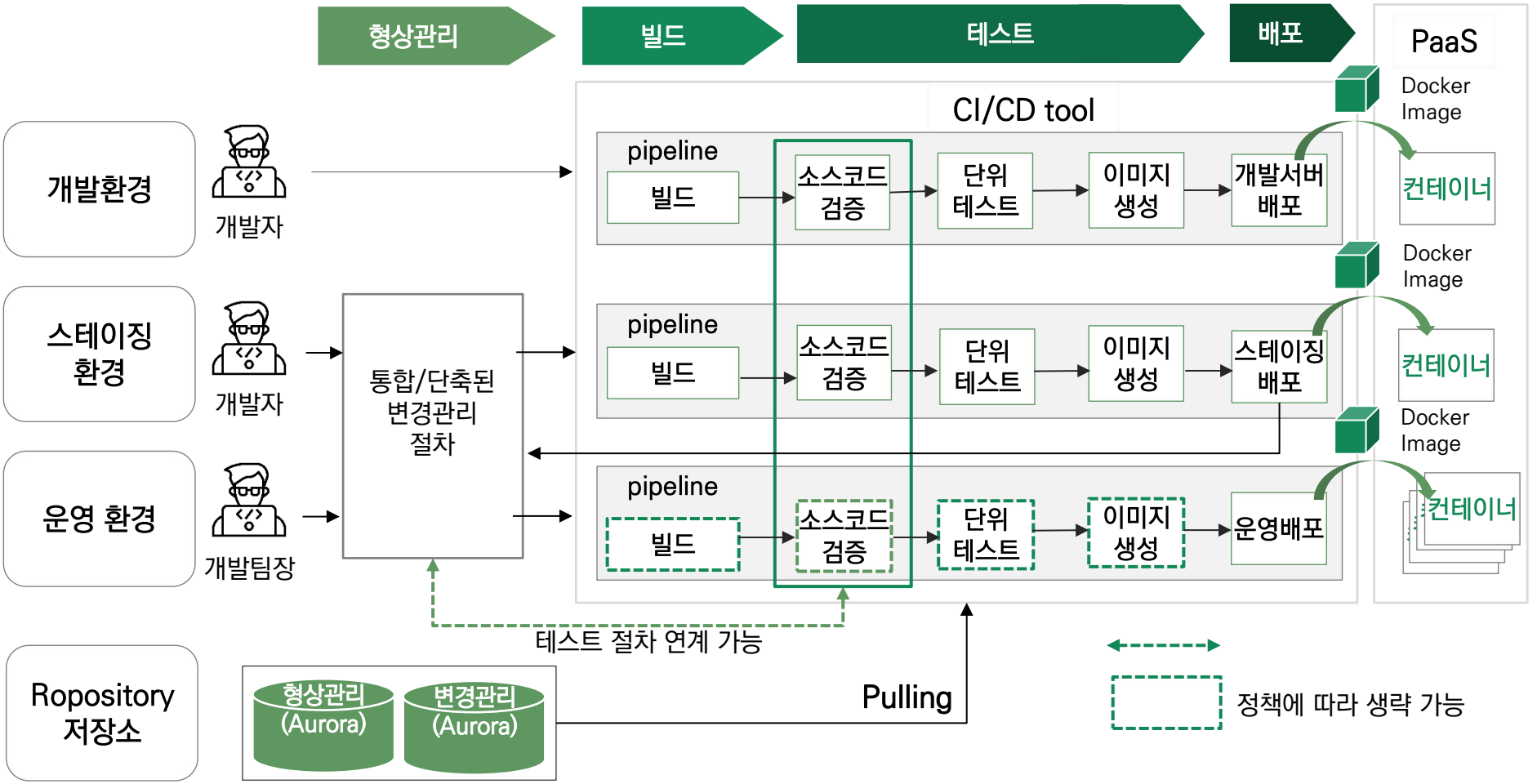
```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.4.RELEASE)

2020-10-19 15:47:01.511  INFO 40527 --- [           main] e.msa.sample.CatalogsApplication
: Starting CatalogsApplication on jeonghunhuiui-iMac.local with PID 40527
(/Users/EGOV3.9/workspace/Catalogs/target/classes started by hhjeong in
/Users/EGOV3.9/workspace/Catalogs)
2020-10-19 15:47:01.513  INFO 40527 --- [           main] e.msa.sample.CatalogsApplication
: No active profile set, falling back to default profiles: default
2020-10-19 15:47:02.187  INFO 40527 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat initialized with port(s): 8081 (http)
2020-10-19 15:47:02.196  INFO 40527 --- [           main] o.apache.catalina.core.StandardService
: Starting service [Tomcat]
2020-10-19 15:47:02.196  INFO 40527 --- [           main] org.apache.catalina.core.StandardEngine
: Starting Servlet engine: [Apache Tomcat/9.0.38]
2020-10-19 15:47:02.256  INFO 40527 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring embedded WebApplicationContext
2020-10-19 15:47:02.256  INFO 40527 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
in 711 ms
2020-10-19 15:47:02.403  INFO 40527 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor
: Initializing ExecutorService 'applicationTaskExecutor'
2020-10-19 15:47:02.524  INFO 40527 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat started on port(s): 8081 (http) with context path ''
2020-10-19 15:47:02.533  INFO 40527 --- [           main] e.msa.sample.CatalogsApplication
: Started CatalogsApplication in 1.567 seconds (JVM running for 1.816)
2020-10-19 15:47:05.921  INFO 40527 --- [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-10-19 15:47:05.921  INFO 40527 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet
: Initializing Servlet 'dispatcherServlet'
2020-10-19 15:47:05.925  INFO 40527 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet
: Completed initialization in 3 ms
```

카탈로그 서비스 테스트 결과 확인

웹 브라우저를 실행하고
URL :
을 입력하면 customerid가 1234인 내용을 확인

클라우드 네이티브 애플리케이션 배포 관리

마이크로서비스의 신속한 배포를 위해 기존의 프로세스를 간소화하고, CI/CD tool을 통해 빌드·테스트를 자동화하며, 컨테이너 이미지를 생성하여 다중 컨테이너 환경에 배포



도커이미지 생성, 실행 및 등록 환경 구현

도커파일 생성

FROM
- 도커 이미지의 바탕이 될 베이스 이미지를 지정
- FROM 에서 받아오는 도커 이미지는 기본적으로 도커 허브 레지스트리에서 참조
- 각 이미지의 버전을 구분하는 식별자로 태크가 지정됨

RUN
- 도커 이미지를 실행할 때 컨테이너 안에서 실행할 명령을 정의

COPY
- 도커가 동작 중인 호스트 머신의 파일이나 디렉토리를 도커 컨테이너 안으로 복사하는 명령을 정의

CMD
- 도커 컨테이너를 실행할 때 컨테이너 안에서 실행할 프로세스 지정
- 애플리케이션 자체를 실행하는 명령

도커 이미지 빌드
- docker build 명령으로 빌드
- -t 옵션으로 이미지명[:태그명]을 지정,
- Dockerfile 위치를 지정
- docker images 명령으로 생성 이미지를 조회

도커 컨테이너 실행
- docker run 명령으로 실행
- 백그라운드 처리시 -d 옵션으로 실행

도커 컨테이너 실행
- 컨테이너 외부의 요청을 컨테이너 내부로 전달
- 포트포워딩 설정시 -p 옵션으로 실행
- -p 호스트포트:컨테이너포트

도커 컨테이너 실행
- docker ps 명령으로 실행한 도커 컨테이너의 상태 조회

도커 컨테이너 실행
- docker stop 명령으로 실행
- 종료할 컨테이너ID를 지정

클라우드 네이티브 애플리케이션 배포 관리

도커 이미지 등록

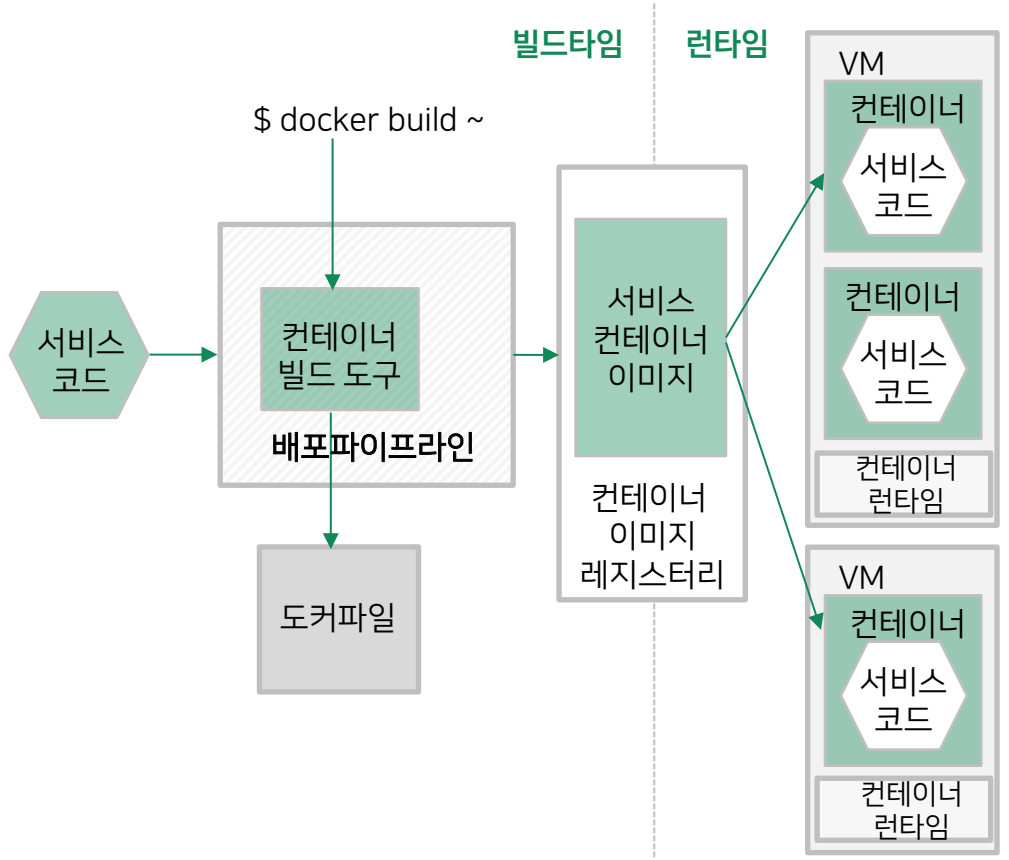
도커파일 작성
- 작업 경로에서 디렉토리 1개 생성 후 해당 디렉토리에 도커파일을 생성

도커 레지스트리 접속
- 도커 레지스트리에 접속함

도커 이미지 빌드
- 생성할 이미지 이름을 지정하고 이미지를 빌드함
- 빌드 명령어를 실행하면 Dockerfile에 정의한 내용이 한 줄 한 줄 실행되는 걸 볼 수 있음

도커 이미지 등록
- 생성한 도커 이미지를 도커 허브에 올리는 과정
- 도커 허브 홈페이지에 회원 가입이 선행되어야 함

컨테이너를 이용한 마이크로 서비스 배포 과정



감사합니다.