클라우드 네이티브 정보시스템 구축을 위한

개발자 안내서

NIA 한국지능정보사회진흥원

서울특별시 중구 청계천로 14 (04520)
대표전화 053-230-1114

클라우드 네이티브 정보시스템 구축을 위한

개발자 안내서

NIA 한국지능정보사회진흥원

본 안내서는 **클라우드 네이티브 정보시스템 구축 사업을 계획하는 발주자와 수행하는 개발자**에게 클라우드 네이티브에 대한 이해도를 높이고, 적극적인 활용을 지원하고자 발간하게 되었습니다.

일러두기

• 안내서를 왜 발간하게 되었는가?

4차 산업혁명 시대를 맞아 국민의 일상에 디지털 전환이 가속화되고 있고, 이에 적합한 대국민 서비스 제공을 위해 공공서비스의 클라우드 전환이 추진되고 있습니다.

클라우드 전환 및 도입 효과를 높이기 위해 단순한 기술 인프라 위주의 클라우드 도입보다 클라우드 환경에 최적화된 새로운 형태의 클라우드 네이티브 정보시스템 구축이 필요합니다. 즉, 기존의 크고, 단일한 서비스 구조를 마이크로서비스 아키텍처로 구현하여 개발, 배포, 운영함으로써 빠르고 안정적인 대국민 서비스를 제공할 수 있습니다.

본 안내서는 공공부문 클라우드 네이티브 관련 정보화 사업에 참여하는 발주자와 개발자에게 관련 기술 정보와 활용 방안 등을 제공하여 성공적인 사업 추진을 지원하고자 발간되었습니다.

• 누가 안내서를 읽어야 할까?

클라우드 관련 정보화 사업을 준비하는 중앙행정기관, 지방자치단체, 공공기관 등 발주자와 클라우드 네이티브 정보시스템 구축 및 운영 사업에 참여하거나 관심이 있는 개발자입니다.

• 언제 안내서를 활용할까?

발주 기관에서 클라우드 기반 정보화 사업을 기획하고 발주하기 전에 발주자 안내서를 통해 클라우드 네이티브의 개념, 주요 기술 등을 이해하고 도입 적합성을 검토할 수 있습니다.

공공·민간 부문 클라우드 네이티브 정보시스템 구축을 위해 주요 기술과 구현·운영 방안을 학습하고자 하는 개발자들은 항시 개발자 안내서를 참고할 수 있습니다.

• 안내서는 어떻게 구성되나?

발주자를 위한

PART Ⅰ - 발주자 안내서

클라우드 네이티브 개요, 도입 필요성, 구성요소 및 원칙, 도입 적합성 검토, 구축사업 추진 시 고려사항으로 구성

정보시스템 개발자를 위한

PART Ⅱ - 개발자 안내서

클라우드 네이티브 정보시스템 구축 절차와 구축 단계별 개발 방안으로 구성

목차

PART Ⅰ 발주자 안내서

01 클라우드 네이티브 개요

- 1.1 클라우드 네이티브 정의 15
- 1.2 클라우드 네이티브 애플리케이션 16
  - 1.2.1 클라우드 네이티브 애플리케이션 정의 16
  - 1.2.2 클라우드 애플리케이션 성숙도 단계 17
- 1.3 클라우드 네이티브 구성요소 18
- 1.4 클라우드 네이티브 특장점 19
  - 1.4.1 클라우드 네이티브 특징 19
  - 1.4.2 클라우드 네이티브 장점 21
  - 1.4.3 클라우드 네이티브 단점 24

02 클라우드 네이티브 도입 필요성

- 2.1 클라우드 동향 28
  - 2.1.1 클라우드 확산 배경 28
  - 2.1.2 클라우드 정책 동향 29
  - 2.1.3 클라우드 시장 동향 34
  - 2.1.4 클라우드 기술 동향 37
- 2.2 클라우드 네이티브 도입 사례 40
  - 2.2.1 해외 공공 부문 40
  - 2.2.2 해외 민간 부문 43
  - 2.2.3 국내 민간 부문 47
- 2.3 클라우드 네이티브 도입 필요성 52
  - 2.3.1 지능정보화의 추진 52
  - 2.3.2 공공부문 정보화 현황 54

목차

PART Ⅰ 발주자 안내서

03 클라우드 네이티브 구성요소 및 원칙

3.1 개요 60
3.1.1 클라우드 네이티브 구성요소 및 원칙 60
3.2 마이크로서비스 61
3.2.1 마이크로서비스 아키텍처 정의 61
3.2.2 마이크로서비스 아키텍처와 모놀리식 아키텍처의 차이점 62
3.2.3 마이크로서비스 아키텍처의 필요성 64
3.2.4 마이크로서비스 아키텍처의 구성 65
3.3 컨테이너 67
3.3.1 컨테이너 정의 67
3.3.2 컨테이너와 가상머신의 차이 68
3.3.3 도커(Docker) 69
3.3.4 쿠버네티스(Kubernetes) 70
3.4 데브옵스 71
3.4.1 데브옵스 개요 71
3.4.2 데브옵스 프로세스 72
3.4.3 데브옵스 문화 73
3.4.4 데브옵스 조직 74
3.4.5 데브섹옵스(DevSecOps) 개념 75
3.5 CI/CD 76
3.5.1 CI/CD 개념 76
3.5.2 CI/CD 파이프라인 77
3.6 애자일 방법론 78
3.6.1 애자일 방법론 개요 78
3.6.2 애자일 방법론과 폭포수 방법론의 차이점 79
3.7 12가지 요소 81
3.7.1 12가지 요소(12 Factors) 개념 81
3.7.2 12가지 요소 원칙 설명 82
3.7.3 12가지 요소 원칙의 특징 83

목차

PART Ⅰ 발주자 안내서

03 클라우드 네이티브 구성요소 및 원칙

3.8 클라우드 네이티브 애플리케이션 아키텍처 84
3.8.1 클라우드 네이티브 애플리케이션 아키텍처 개요 84
3.8.2 애플리케이션 실행 영역 85
3.8.3 백엔드 서비스 86
3.8.4 개발·실행 지원 서비스 87
3.8.5 운영 지원 서비스 88
3.8.6 클라우드 인프라 89

04 클라우드 네이티브 적합성 검토

4.1 개요 92
4.2 클라우드 네이티브 적합성 검토 체크리스트 93
4.2.1 클라우드 네이티브 도입 목표 93
4.2.2 클라우드 네이티브 적합성 검토 항목 도출 94
4.2.3 클라우드 네이티브 적합성 검토 항목 설명 96
4.2.4 클라우드 네이티브 적합성 검토 체크리스트 102
4.3 클라우드 네이티브 적합성 검토 수행 105
4.3.1 클라우드 네이티브 적합성 검토 수행 개요 105
4.3.2 클라우드 적합성 검토 후 도입 여부 결정 106
4.3.3 클라우드 네이티브 도입 우선순위 결정 107
4.4 클라우드 네이티브 적합성 검토 예시 108
4.4.1 표준 프레임워크 포털 108
4.4.2 공영홈쇼핑 영업시스템과 온라인몰 110
4.4.3 한국부동산원 감정평가 및 공시가격 정보체계 113

목차

PART Ⅰ 발주자 안내서

05 클라우드 네이티브 정보시스템 구축 사업 추진 고려사항

5.1 사업관리 단계별 고려사항 117
- 5.1.1 정보화사업관리 프로세스 117
- 5.1.2 클라우드 네이티브 관련 발주자 업무 118

5.2 사업계획서 및 제안요청서 작성 시 고려사항 120
- 5.2.1 사업계획서 및 제안요청서 작성 개요 121
- 5.2.2 사업 추진 방향성 및 사업 범위 작성 122
- 5.2.3 상세 요구사항 작성 123

5.3 개발비 산정 시 고려사항 125
- 5.3.1 개발비 구성요소 125
- 5.3.2 기능식별 방식의 변화 126

5.4 데브옵스 조직 관련 고려사항 128

목차

PART II 개발자 안내서

06 클라우드 네이티브 정보시스템 개발 절차

6.1 클라우드 네이티브 정보시스템 개발 절차 개요 132
6.2 클라우드 네이티브 정보시스템 구축 단계별 개발 절차 설명 133

07 클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출 137
7.1.1 마이크로서비스 도출 유형 137
7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출 138
7.1.3 이벤트 스토밍을 통한 마이크로서비스 도출 145
7.1.4 업무기능 분해를 통한 마이크로서비스 도출 150
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계 155
7.2.1 개요 155
7.2.2 클라우드 네이티브 애플리케이션 아키텍처 구성 156
7.2.3 API 게이트웨이 157
7.2.4 서비스 메시 163
7.2.5 런타임 플랫폼 168
7.2.6 CI/CD 177
7.2.7 백엔드 서비스 181
7.2.8 텔레메트리 184

08 클라우드 네이티브 정보시스템 설계 단계

8.1 도메인 모델링을 통한 마이크로서비스 설계
8.1.1 마이크로서비스 상세 설계 개요 187
8.1.2 도메인모델의 패턴 188
8.1.3 마이크로서비스 프로젝트 설계 193

목차

PART II 개발자 안내서

8.2 마이크로서비스 아키텍처 설계 195
- 8.2.1 마이크로서비스 아키텍처 개념 195
- 8.2.2 마이크로서비스 아키텍처 패턴 196
- 8.2.3 쿼리 패턴 – API 조합 패턴 설계 197
- 8.2.4 쿼리 패턴 – CQRS 패턴 설계 201
- 8.2.5 트랜잭션 관리 설계 203
- 8.2.6 서비스 간 통신 방식 설계 205
- 8.2.7 서비스 메시 설계 206

8.3 12 Factors 기반 개발 원칙 211
- 8.3.1 개발원칙 개요 211
- 8.3.2 코드베이스 212
- 8.3.3 종속성 213
- 8.3.4 설정 214
- 8.3.5 백엔드 서비스 215
- 8.3.6 빌드·릴리즈·실행 환경 216
- 8.3.7 무상태 서비스 217
- 8.3.8 포트 바인딩 218
- 8.3.9 동시성 219
- 8.3.10 폐기 가능성 220
- 8.3.11 개발/운영 환경 일치 221
- 8.3.12 로그 222
- 8.3.13 관리 프로세스 223

09 클라우드 네이티브 정보시스템 구현·운영 단계

9.1 개발· 테스트·운영 환경 구성 226

목차

PART II 개발자 안내서

9.2 스프링 부트 기반 마이크로서비스 개발 235
9.2.1 개요 235
9.2.2 카탈로그 서비스 개발 236
9.2.3 고객 서비스 개발 242
9.2.4 카탈로그와 고객 서비스 연동 및 테스트 247
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축 249
9.3.1 스프링 클라우드 주요 컴포넌트 249
9.3.2 서킷 브레이커-히스트릭스(Hystrix) 250
9.3.3 클라이언트 로드밸런서 – 리본(Ribbon) 253
9.3.4 서비스 레지스트리 – 유레카(Eureka) 256
9.3.5 API 게이트웨이 – 줄(Zuul) 263
9.3.6 설정 서버 – 컨피그(Config) 269
9.3.7 스프링 클라우드 버스(Bus) 282
9.3.8 폴리글랏 지원 – 사이드카(Sidecar) 288
9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana) 294
9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana) 300
9.3.11 분산 서비스 로그 추적-슬루스 & 집킨(Sleuth, Zipkin) 308
9.4 컨테이너 기반 마이크로서비스 빌드·배포 312
9.4.1 도커 컨테이너 기반 빌드·배포 312
9.4.2 쿠버네티스 기반 배포 313
9.4.3 도커와 쿠버네티스를 활용한 배포 314

PART II

개발자 안내서

클라우드 네이티브 정보시스템 구축을 위한

개발자 안내서

06 클라우드 네이티브 정보시스템 개발 절차

6.1 개요

- 클라우드 네이티브 정보시스템 개발 절차는 분석, 설계, 구현, 테스트 및 이행의 단계로 진행된다. 요구사항이 명확한 대부분의 공공 정보화사업은 폭포수 방법론에 따라 수행하고, 새로운 서비스를 구현하는 경우에는 애자일 방법론에 따라 점진적이고 반복적으로 수행한다.
- 클라우드 네이티브 정보시스템 개발은 기존 정보시스템 개발 공정과 다르게 마이크로서비스 도출· 설계·개발, 마이크로서비스 아키텍처 설계·구축, 빌드·배포 등의 태스크가 포함된다.

[그림 6-1] 클라우드 네이티브 정보시스템 개발 절차

단계	태스크			
분석	요구사항 분석			
	마이크로서비스 도출	클라우드 네이티브 아키텍처 설계		
설계	마이크로서비스 설계 (도메인모델링)	마이크로서비스 아키텍처 설계	프런트엔드(UI) 설계	개발 원칙 수립
구현	마이크로서비스 개발 및 테스트	마이크로서비스 아키텍처 구축	프런트엔드(UI) 개발 및 테스트	개발·테스트·운영환경 구축
	빌드·배포		사용자 테스트	
테스트 및 이행	통합테스트	데이터이행	릴리즈 (운영이관)	안정화 및 운영

점진적 반복 (설계·구현 단계의 마이크로서비스 설계 ~ 사용자 테스트)

클라우드 네이티브 관련 태스크

클라우드 네이티브 정보시스템 개발 절차

6.2 클라우드 네이티브 정보시스템 개발 절차 설명

- **클라우드 네이티브 정보시스템 개발 공정단계별 태스크에서 수행하는 작업은 다음과 같다.**

[표 6-2] 클라우드 네이티브 정보시스템 개발 절차 설명

단계	태스크	설명	비고
분석	요구사항 분석	• 사용자에 대한 인터뷰 및 설문 등을 통해 업무 개선 및 시스템 개선 요구사항을 수집, 분석하고 검토함 - 업무 개선 요구 : 현행 업무 문제점, 개선사항 등 분석 - 시스템 개선 요구 : 현행 시스템의 문제점, 개선사항, 시스템 영향도, 제약사항 등 분석	
	마이크로서비스 도출	• 업무기능의 응집도와 유사성을 고려하여 독립적으로 배포하고, 확장할 수 있는 마이크로서비스를 도출함 - 도메인 주도 설계, 이벤트 스토밍, 업무기능 분해 방식을 통해 마이크로서비스를 도출	
	클라우드 네이티브 아키텍처 설계	• 클라우드 네이티브 애플리케이션 아키텍처의 구성요소 설계 - API 게이트웨이, 서비스 메시, 런타임 플랫폼, CI/CD, 백엔드 서비스, 텔레메트리 등 구성요소의 기능을 검토하고, 클라우드 플랫폼이 제공하는 장비 및 SW 확인 및 결정	개념 아키텍처
설계	마이크로서비스 설계 (도메인모델링)	• 도메인모델링을 통한 마이크로서비스 내부 구조를 상세 설계하고, 이를 기반으로 개발 단계에 최종 코드를 구현함 - 도메인이 제공하는 기능과 주요 데이터를 포함	
	마이크로서비스 아키텍처 설계	• 마이크로서비스 환경 설정, 서비스 관리, 서비스 게이트웨이. 모니터링, 큐잉(메시지 브로커) 등 환경 구성 설계 • 도메인 모델의 표준 패턴을 이용하여 도메인 구현모델 설계 - 쿼리 패턴(API 조합 패턴, CQRS 패턴), - 트랜잭션 관리(SAGA 패턴)	논리 아키텍처
	프런트엔드(UI) 설계	• 효율적인 업무처리를 위한 사용자 인터페이스(UI) 설계 - 사용자 경험(UX) 설계, 화면 구성 및 디자인(UI) 설계 등	
	개발 원칙 수립	• 클라우드 네이티브 정보시스템을 구축하기 위한 개발원칙을 수립함 • SaaS 개발원칙인 12 Factors를 토대로 각 기관에서 클라우드 네이티브 정보시스템을 구축하기 위한 개발 원칙을 수립함	

클라우드 네이티브 정보시스템 개발 절차

6.2 클라우드 네이티브 정보시스템 개발 절차 설명

[표 6-2] 클라우드 네이티브 정보시스템 개발 절차

단계	태스크	설명	비고
구현	개발·테스트·운영환경 구축	• 개발 및 단위 테스트를 위해 필요한 SW를 로컬 개발환경에 설치하고 검증 • 공공 또는 민간 클라우드 플랫폼 기반으로 개발· 테스트 ·운영 환경 구축	
	마이크로서비스 개발 및 테스트	• 도메인 모델의 결과물을 기반으로 마이크로서비스를 코드로 구현하고 단위 테스트를 수행(서비스와 리파지토리 구현) • 구현된 업무를 적절한 기술로 구현하기 위한 패키지 구성	스프링부트 활용
	마이크로서비스 아키텍처 구축	• 마이크로서비스 아키텍처 시스템 환경을 구성 - 스프링 클라우드 등 오픈소스 라이브러리를 활용 - 주요 구성 요소 : API 게이트웨이, 설정, 서비스 디스커버리, 서킷 브레이커, 이벤트 버스, 로드 밸런서, 서비스 라우터, 폴리글랏 지원 등	물리 아키텍처
	프론트엔드(UI) 개발 및 테스트	• 사용자 인터페이스(UI)를 개발하고 테스트함	
	빌드·배포	• 도커, 쿠버네티스 등 컨테이너를 기반으로 마이크로서비스를 빌드·배포함 • 저장소에 커밋한 코드의 모든 변경 사항을 검증한 후 빌드 수행한 후 테스트 단계로 넘어감 • 빌드되고 테스트된 결과물을 원하는 환경으로 배포함	
	사용자 테스트	• 빌드된 마이크로서비스를 사용자, 개발자 등이 함께 테스트를 수행하고 개선 요구사항을 반영하여 해결하는 과정	
테스트·이행	통합테스트	• 설계와 분석 단계를 통해 개발된 마이크로서비스 간 연계, 시스템 성능 테스트 수행 및 문제 해결 수행 • 통합테스트 결과에 따라 릴리즈(새로운 버전 출시)를 결정	
	데이터 이행	• 기존 시스템의 클라우드 네이티브 전환 또는 신규 시스템 구축 시 초기 데이터 또는 기존 데이터를 이행 구축	
	릴리즈(운영이관)	• 테스트가 완료된 애플리케이션을 운영환경으로 릴리즈	
	안정화 및 운영	• 클라우드 네이티브 정보시스템 구축 후 서비스 안정화 및 사후관리	

07

클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로 서비스 도출
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.1 마이크로서비스 도출 유형

- 분석 단계에 마이크로서비스를 도출하는 방식은 도메인 주도 설계(DDD : Domain Driven Design), 이벤트 스토밍(Event Storming), 업무기능 분해 등이 존재한다. 기관의 특성과 사업 추진 현황을 고려하여 적합한 방식을 적용하도록 한다.
- 도메인 주도 설계는 업무 단위 간 경계의 식별을 통해 마이크로서비스를 도출하고, 이벤트 스토밍은 도메인 전문가와 개발자 등 이해관계자의 브레인스토밍 워크숍을 통해 마이크로서비스를 도출한다.
- 기존의 레거시 시스템이 존재하는 경우에는 업무기능 분해를 통해 마이크로서비스를 식별할 수 있다.
- 마이크로서비스 도출 시 업무기능의 크기를 고려하도록 한다.

[그림 7-1] 마이크로서비스 도출 유형

1) 도메인(Domain) : 실세계에서 사건이 발생하는 집합

07 클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

7.1.2.1 도메인 주도 설계 개념

- 도메인 주도 설계는 모델링과 개발의 불일치 문제를 해결하기 위해 필요하며, 모든 참여자들이 공통의 언어(유비쿼터스 언어) 사용을 통해 비즈니스나 현실의 문제를 개념적으로 표현한 도메인 모델을 만들어 낸다.
- 도메인 모델은 자동화된 비즈니스나 현실의 문제, 즉 도메인을 개념적으로 표현하는 기법으로 도메인의 가치를 최우선시한다. 도메인 모델의 핵심 원칙은 모델링 수행 시 사용자, 도메인 전문가, 분석가, 개발자 등이 동일한 모습으로 도메인을 이해하고 도메인 지식을 공유하는 것이다.

[그림 7-2] 도메인 모델링 과정의 참여자

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

- 소프트웨어의 본질은 해당 소프트웨어 사용자를 위해 도메인에 관련된 문제를 해결하는 것이지만 기존의 개발방식은 다음과 같은 이유로 도메인 관련 문제 해결이 원활하지 못하다.
  - **최신 기술 중심적인 설계 반복**
  - **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 설계는 개발 참여자가 공통의 언어(유비쿼터스 언어) 사용을 통해 모델링과 개발의 불일치를 해결하고, 설계와 구현은 계속적인 수정 과정을 반복함으로써 개발 품질을 향상시킬 수 있다.

클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

7.1.2.2 도메인 주도 설계의 주요 내용

- 도메인 주도 설계는 잘 설계된 마이크로서비스 집합을 가장 잘 활용할 수 있는 프레임워크를 제공하며, 기본 설계인 전략적 설계와 상세 설계인 전술적 설계로 나누어진다.
- 전략적 설계는 비즈니스상 전략적으로 중요한 것을 구분하고 찾는 과정으로 유비쿼터스 언어로 바운디드 컨텍스트(Bounded Context, 제한된 경계)를 도출하고 컨텍스트 맵을 작성하며, 최종적으로 마이크로서비스를 도출한다.
- 전술적 설계는 도메인 모델을 만드는 데 사용할 수 있는 디자인 패턴 집합을 제공하며, 마이크로서비스 내부 아키텍처 설계에 도움이 된다.

[그림 7-3] 도메인 주도 설계의 주요 내용

도메인 주도 설계

전략적 설계(기본 설계)	전술적 설계(상세 설계)
• 분석 단계에 수행 • 마이크로서비스 도출 - 바운디드 컨텍스트 도출을 통한 마이크로서비스 도출 - 유비쿼터스 언어 사용 - 컨텍스트 맵	• 설계 단계에 수행 • 마이크로서비스 내부 구조 설계 - 애그리게잇(Aggregate) - 엔티티(Entity) - 값 객체(Value Object) - 리파지토리(Repository) - 서비스(Service) - 도메인 이벤트(Domain Event) - 모듈(Module)

※ 마이크로서비스 도출은 도메인 주도 설계의 전략적 설계를 통해 가능함

07 클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

7.1.2.3 도메인 주도 설계를 통한 마이크로서비스 도출 절차

- 마이크로서비스가 비즈니스 변화에 민첩하게 대응하고, 신속하게 배포되기 위해서는 각각의 마이크로서비스는 독립적으로 도출되어야 한다. 즉 마이크로서비스 간 느슨한 결합(Loosely coupled)과 강한 응집력(High cohesion)이 보장될 수 있어야 한다.
- 도메인 전문가와 개발자 등이 유비쿼터스 언어로 도메인 간의 경계를 고려하여 바운디드 컨텍스트를 도출하고, 이들 간의 관계를 컨텍스트 맵으로 작성한다. 바운디드 컨텍스트의 분석을 통해 서비스의 분할 또는 통합을 검토하여 최종적으로 마이크로서비스를 도출한다.

[그림 7-4] 마이크로서비스 도출 절차

명확하게 식별된 바운디드 컨텍스트 내부를 모델링하기 위한 **유비쿼터스 언어**[1)]를 정의

유비쿼터스 언어를 활용해서 비즈니스를 분석하고 핵심이 되는 개념을 식별

식별된 개념을 분석해서 **바운디드 컨텍스트**[2)]를 식별

식별된 컨텍스트 간 관계를 찾아 **컨텍스트 맵**[3)]을 작성

바운디드 컨텍스트로의 분할에 따른 효과 분석을 통해 서비스의 분할 또는 통합을 검토

최종적으로 마이크로서비스 도출

1) 유비쿼터스(Ubiquitous) 언어 : 프로젝트 팀원이 공통으로 사용하는 보편적인 언어
2) 바운디드 컨텍스트(Bounded Contexts) : 도메인의 주요 개념들을 정의하고, 도메인 간의 경계를 식별
3) 컨텍스트 맵(Context Map) : 경계 간 관계를 분석하여 매핑 관계를 작성

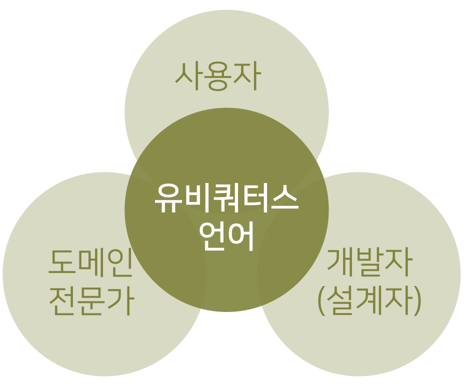
07 클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

7.1.2.3 도메인 주도 설계를 통한 마이크로서비스 도출 절차

[그림 7-5] 참고. 유비쿼터스 언어

구분	내용
유비쿼터스 언어 개념	• 도메인 전문가, 사용자, 분석가, 개발자 등 모델링에 참여하는 사람들이 사용하는 공통된 언어 
유비쿼터스 언어 필요성	• 개발 시 커뮤니케이션 오류의 발생 가능성 최소화 • 바운디드 컨텍스트 모델링 시 활용 - 유비쿼터스 언어로 정의된 모델을 기반으로 경계를 만듦
유비쿼터스 언어의 관리	• 용어사전 작성 - 위키(Wiki) 등 다양한 형태로 도메인 전문가, 아키텍트, 개발자 등 프로젝트 구성원이 모여서 워크숍 형태로 유비쿼터스 언어로 용어사전을 작성함 - 설계 및 개발 과정에 용어 관련 각종 변경사항을 지속적으로 관리함 • 용어사전 예시 - 회원 : 서비스 이용을 위해 사이트에 회원 가입을 한 사람 - 고객 : 상품 및 서비스를 구매한 사람
유비쿼터스 언어의 활용	• 설계 과정의 각종 모델링, 문서 작성, 의사소통, 각종 코드 설계 등에 활용 • 개발 중간 과정에 참여하는 개발자들이 쉽게 업무를 이해할 수 있음

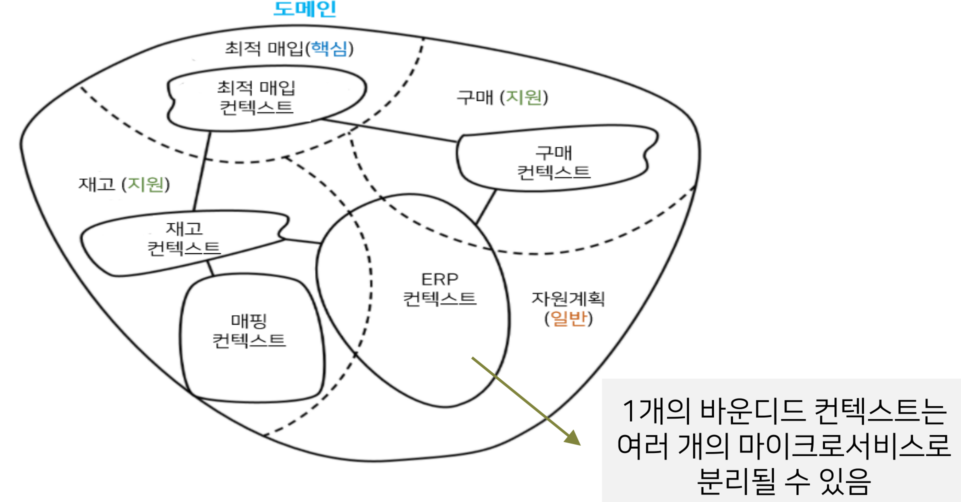
07 클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

7.1.2.3 도메인 주도 설계를 통한 마이크로서비스 도출 절차

[그림 7-6] 참고. 도메인, 서브 도메인, 바운디드 컨텍스트

구분	내용
도메인 개념	• 소프트웨어가 해결해야 할 문제 영역 • 도메인은 서브 도메인으로 구성됨 - 세부 업무 영역 또는 조직 단위
서브 도메인 개념	• 도메인 내 문제 해결을 위해 도메인을 여러 개의 서브 도메인으로 나눔 • 서브 도메인은 핵심, 지원, 일반 서브 도메인으로 분류됨 - **핵심** 서브 도메인 : 핵심적인 비즈니스 로직이 담긴 영역으로, 차별화된 영역 - **지원** 서브 도메인 : 핵심 서브 도메인을 기능적으로 보조할 수 있는 영역 - **일반** 서브 도메인 : 범용적으로 사용될 수 있는 도메인
바운디드 컨텍스트 개념	• 도메인의 문제를 해결하기 위한 솔루션 영역 • 도메인 모델이 존재하는 명시적인 경계 • 바운디드 컨텍스트는 전체 비즈니스 도메인을 여러 개의 서브 도메인으로 나눈 후, 서브 도메인 내 동일한 맥락을 경계로 구분하여 바운디드 컨텍스트를 도출함
바운디드 컨텍스트, 서브 도메인, 마이크로서비스 관계	• 바운디드 컨텍스트와 서브도메인의 관계는 1:1이 이상적임 • 바운디드 컨텍스트가 커질수록 더 작은 단위로 분할 가능하므로 1:N 관계가 될 수도 있음 

클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

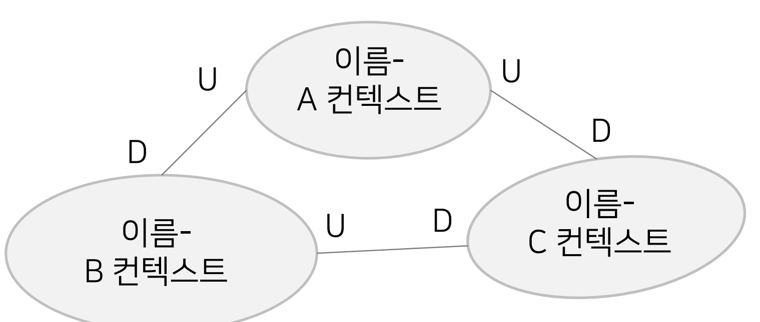
7.1.2.3 도메인 주도 설계를 통한 마이크로서비스 도출 절차

[그림 7-7] 참고. 컨텍스트 맵

컨텍스트 맵 개념

- 두 개 이상의 바운디드 컨텍스트 사이의 연관성을 보여주는 다이어그램으로 문제 해결의 공간에 초점을 맞춤
- 모델과 모델이 만나는 경계 지점을 서술하며, 항상 현재 상태 그대로의 상황을 표현함
- 맵은 특정한 형식으로 문서화할 필요가 없지만, 모든 구성원은 경계가 어디에 위치하는지 알아야 함

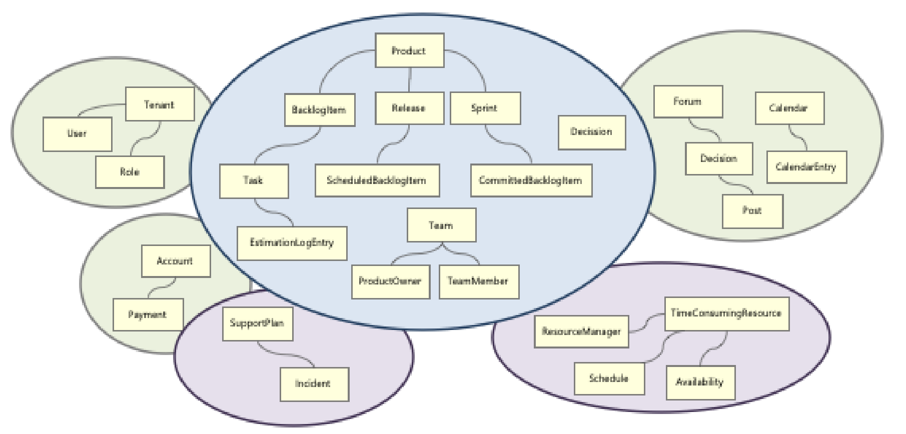
추상적 수준의 컨텍스트 맵



U : 업스트림(Upstream), D : 다운스트림(Downstream)

- 위의 컨텍스트 맵은 해당 프로젝트에 속한 팀의 맵으로 다른 프로젝트팀에서 이를 참고할 수 있지만, 다른 프로젝트팀 고유의 맵도 작성되어야 함
- 컨텍스트 맵은 상호 교류하는 시스템의 목록을 제공하고, 팀 내 의사소통의 촉매 역할을 함

컨텍스트 맵 예시



출처 : 도메인주도설계핵심, 박현철/전장호, 에이콘, 2017

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.2 도메인 주도 설계를 통한 마이크로서비스 도출

7.1.2.3 도메인 주도 설계를 통한 마이크로서비스 도출 절차

- 컨텍스트 맵에서 서브 도메인은 각 업무에 특화된 유비쿼터스 언어로 정의되고 그 업무에 특화된 데이터로 구성되므로 독립적으로 운영되는 마이크로서비스와 유사한 성격을 가진다.
- 서브 도메인을 구성하는 바운디드 컨텍스트를 기반으로 마이크로서비스를 도출할 수 있으며, 도출 예시는 다음과 같다.

[그림 7-8] 컨텍스트 맵을 통한 마이크로서비스 도출 예시

컨텍스트 맵

주문 서브도메인
주문 바운디드 컨텍스트
상품 바운디드 컨텍스트

결제 서브도메인
결제 바운디드 컨텍스트

배송 서브도메인
배송업체관리 바운디드 컨텍스트
배송 바운디드 컨텍스트

고객 서브도메인
고객 바운디드 컨텍스트

바운디드 컨텍스트를 마이크로서비스로 전환

마이크로서비스 도출

고객 서비스
주문 서비스
결제 서비스
상품 서비스
배송업체관리 서비스
배송 서비스

07 클라우드 네이티브 정보시스템 분석 단계

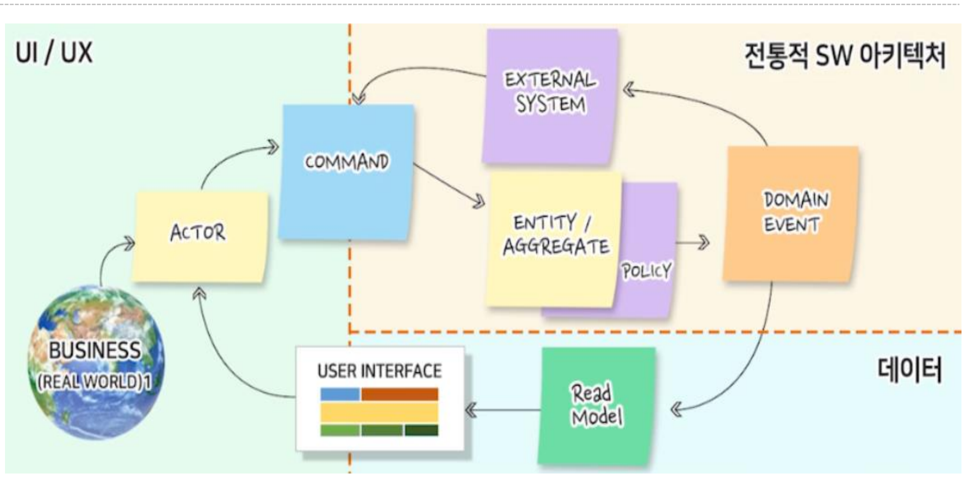
7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.3 이벤트 스토밍을 통한 마이크로서비스 도출

7.1.3.1 이벤트 스토밍의 개념 및 특징

- **이벤트 스토밍은 이벤트(Event)와 브레인스토밍(Brain Storming)의 합성어로 도메인의 이벤트를 통해 마이크로서비스를 도출하기 위한 브레인스토밍 활동이다.**
- **도메인 전문가와 개발자 등 모든 이해관계자가 함께 모여 도메인에 대한 공통의 이해를 통해 도메인 주도 이벤트 방식으로 바운디드 컨텍스트를 식별하고, 이를 기반으로 마이크로서비스를 도출한다.**

[그림 7-9] 이벤트 스토밍의 특징

<table>
<tr><td>이벤트 스토밍
특징</td><td>• 사전에 훈련된 지식과 도구 없이 수행 가능
• 단시간 내 업무 흐름을 표현하는 모델을 만들 수 있음
• 복잡한 시스템의 동작과 결과를 시각적으로 표현할 수 있음
• 협력을 통해 업무 이해도 향상, 오류 및 누락된 개념 파악
• 바운디드 컨텍스트와 마이크로서비스를 빠르게 도출
• 디자인과 코딩으로의 원활한 전환 가능</td></tr>
<tr><td>이벤트 스토밍
구성요소</td><td>


출처 : Alberto Brandolini, “Introducing to EventStorming”
• 위 이미지는 이벤트 스토밍이 현실 세계의 비즈니스 도메인을 다양한 의미의 스티커를 활용하여 표현한 것임
• 사람(Actor)은 원하는 것을 얻기 위해 명령(Command)을 내리게 되고, 이 명령에 의해 도메인 이벤트(Domain Event)가 발생하게 됨
• 도메인 이벤트가 발생된 후에 다시 명령을 내리기 위해 필요한 정보를 참고하여(Read Model) 사용자 인터페이스(User Interface)로 표현함</td></tr>
</table>

07 클라우드 네이티브 정보시스템 분석 단계

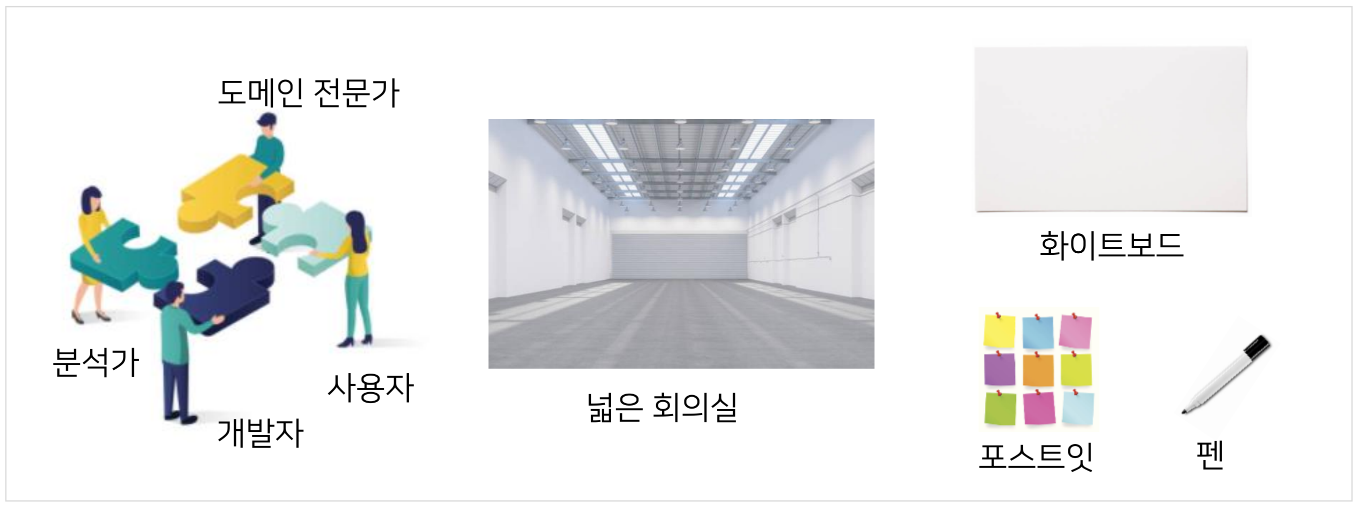
7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.3 이벤트 스토밍을 통한 마이크로서비스 도출

7.1.3.2 이벤트 스토밍 준비 및 수행 절차

- **이벤트 스토밍을 진행하기 위해 도메인 전문가, 분석가, 사용자, 개발자 등 이해관계자가 모일 수 있는 넓은 회의실, 화이트보드, 포스트잇, 펜 등을 준비한다.**

[그림 7-10] 이벤트 스토밍의 준비



- **이벤트 스토밍 과정을 통해 이벤트 도출, 커맨드 도출, 애그리게잇 식별, 바운디드 컨텍스트 식별의 단계를 거쳐 마이크로서비스를 도출한다.**

[그림 7-11] 이벤트 스토밍 수행 절차

클라우드 네이티브 정보시스템 분석 단계

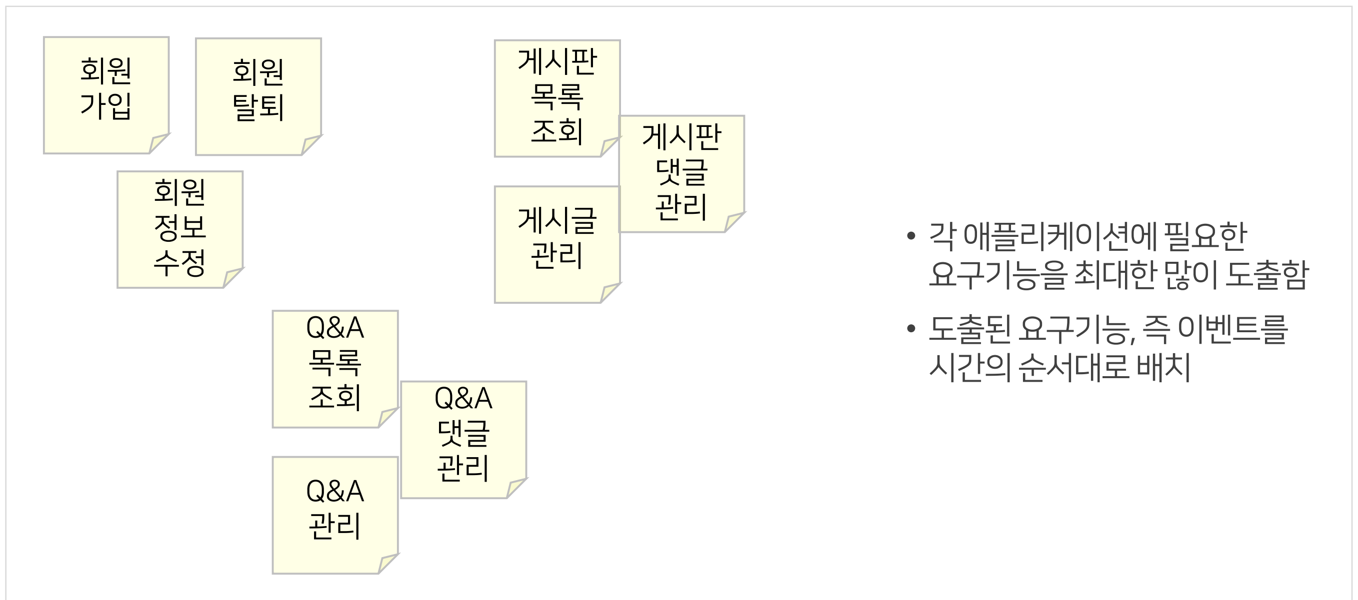
7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.3 이벤트 스토밍을 통한 마이크로서비스 도출

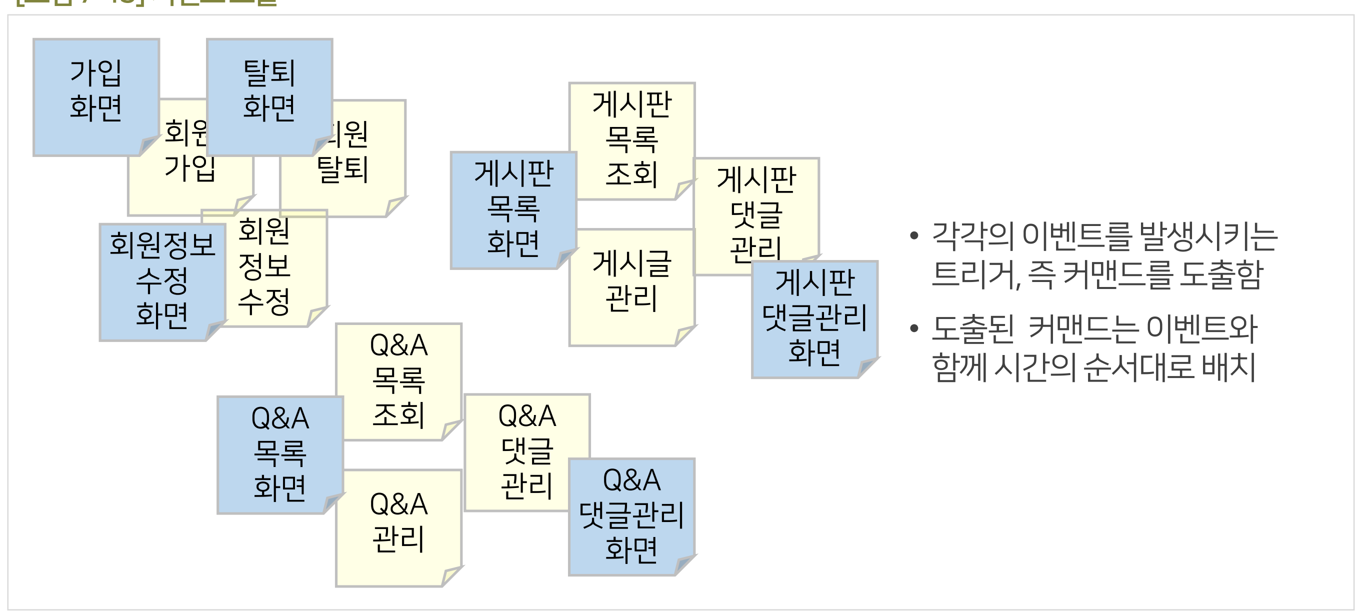
7.1.3.3 이벤트 스토밍 수행 예시

- **이벤트 스토밍을 통해 시간의 순서에 따라 애플리케이션 이벤트를 도출하고, 이벤트를 발생시키는 트리거, 즉 커맨드를 도출한다.**

[그림 7-12] 이벤트 도출



[그림 7-13] 커맨드 도출



07 클라우드 네이티브 정보시스템 분석 단계

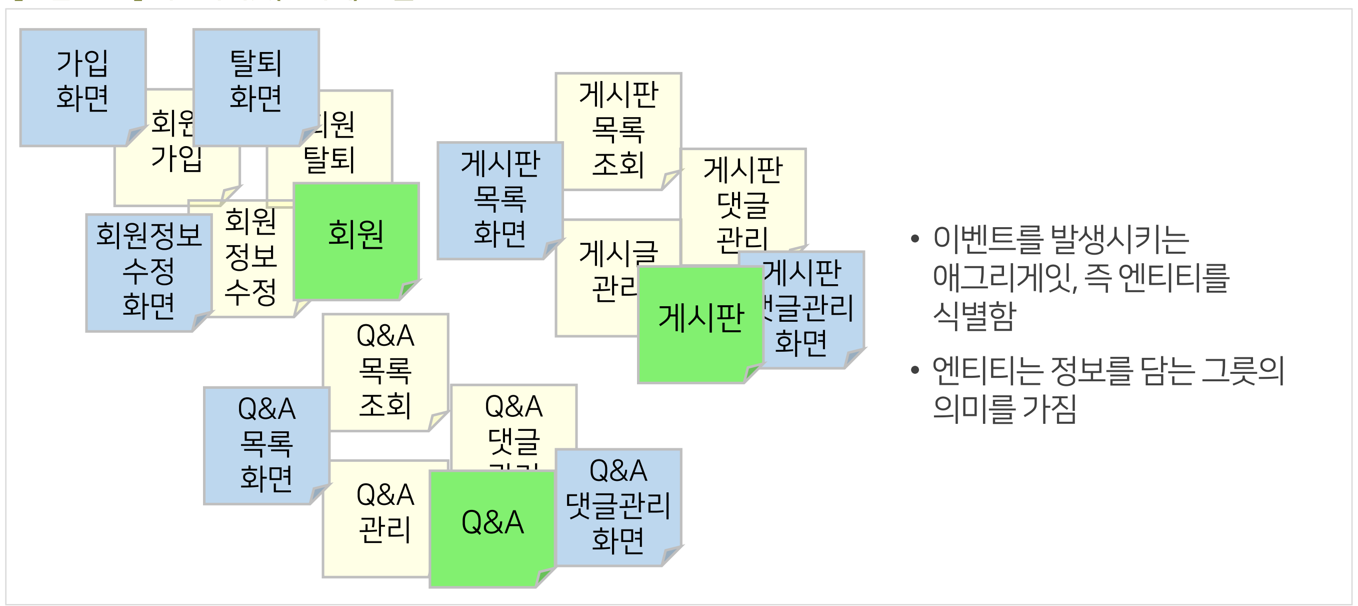
7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.3 이벤트 스토밍을 통한 마이크로서비스 도출

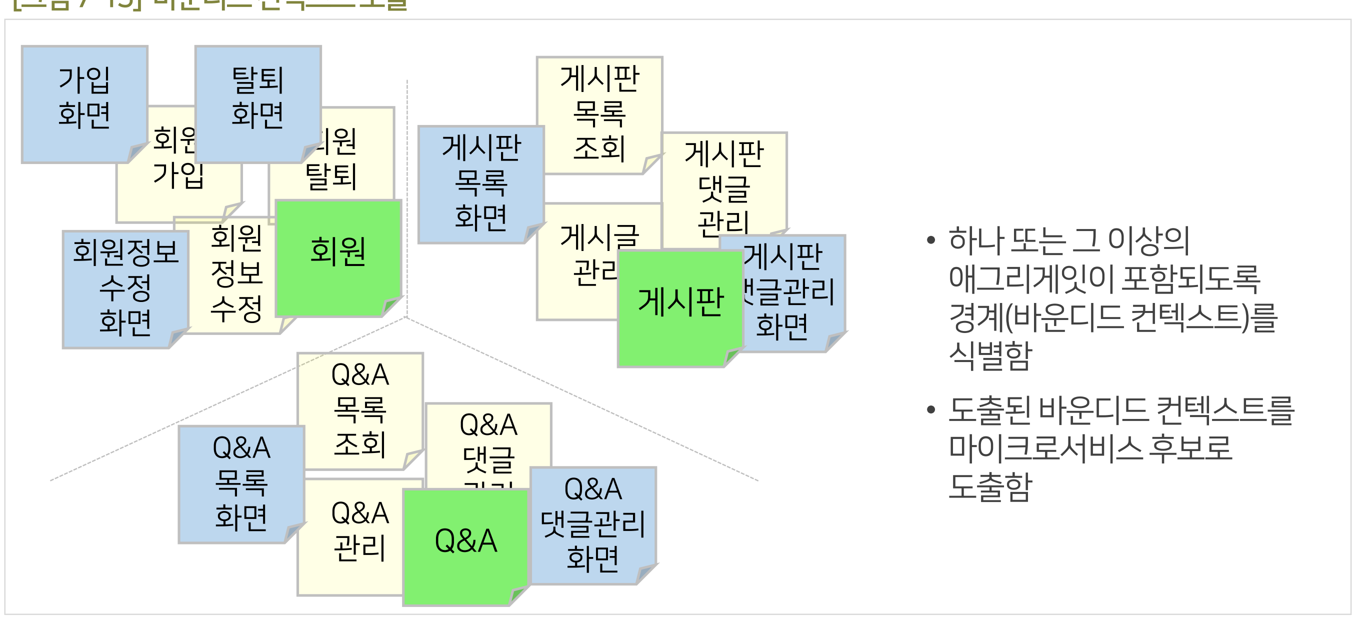
7.1.3.3 이벤트 스토밍 수행 예시

- **이벤트를 발생시키는 애그리게잇(엔티티)을 도출한 후, 하나 또는 그 이상의 애그리게잇이 포함되도록 바운디드 컨텍스트를 도출한다.**

[그림 7-14] 애그리게잇(엔티티) 도출



[그림 7-15] 바운디드 컨텍스트 도출



클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.3 이벤트 스토밍을 통한 마이크로서비스 도출

7.1.3.3 이벤트 스토밍 수행 예시

- **이벤트 스토밍 결과에 의해 도출된 바운디드 컨텍스트를 토대로 상호 호출관계, 독립적 배포 가능성을 고려하여 마이크로서비스를 도출한다.**
- **도출된 마이크로서비스를 1개의 팀에서 운영하기에 작거나 크다고 판단될 경우, 마이크로서비스를 합치거나 분리하여 정의할 수 있다.**

[그림 7-16] 컨텍스트 맵을 통한 마이크로서비스 도출 예시

상호 호출관계, 독립적 배포 가능성 등 검토

이벤트 스토밍 결과

회원관리: 가입 화면, 회원 가입, 탈퇴 화면, 회원 탈퇴, 회원정보 수정 화면, 회원 정보 수정, 회원

게시판관리: 게시판 목록 화면, 게시판 목록 조회, 게시판 댓글 관리, 게시글 관리, 게시판, 게시판 댓글관리 화면

Q&A관리: Q&A 목록 화면, Q&A 목록 조회, Q&A 댓글 관리, Q&A 관리, Q&A, Q&A 댓글관리 화면

바운디드 컨텍스트를 마이크로서비스로 전환

마이크로서비스 도출

회원관리 서비스

게시판 관리 서비스

Q&A관리 서비스

※ 1개 팀에 의해 원활하게 운영하기 위해 마이크로서비스를 합치거나 분리할 수 있음

07 클라우드 네이티브 정보시스템 분석 단계

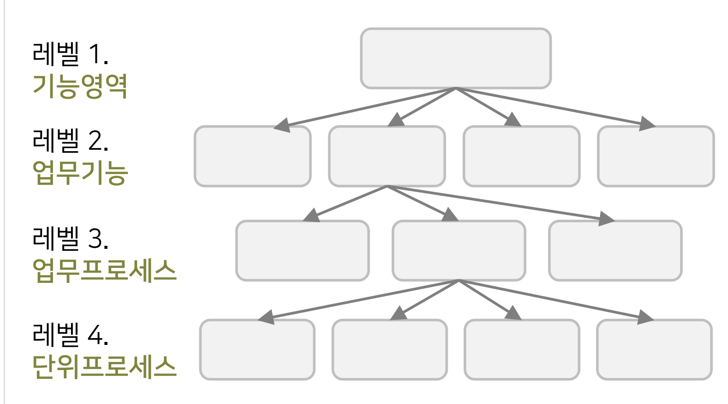
7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.4 업무기능 분해를 통한 마이크로서비스 도출

7.1.4.1 업무기능 분해 개념 및 마이크로서비스 식별 원칙

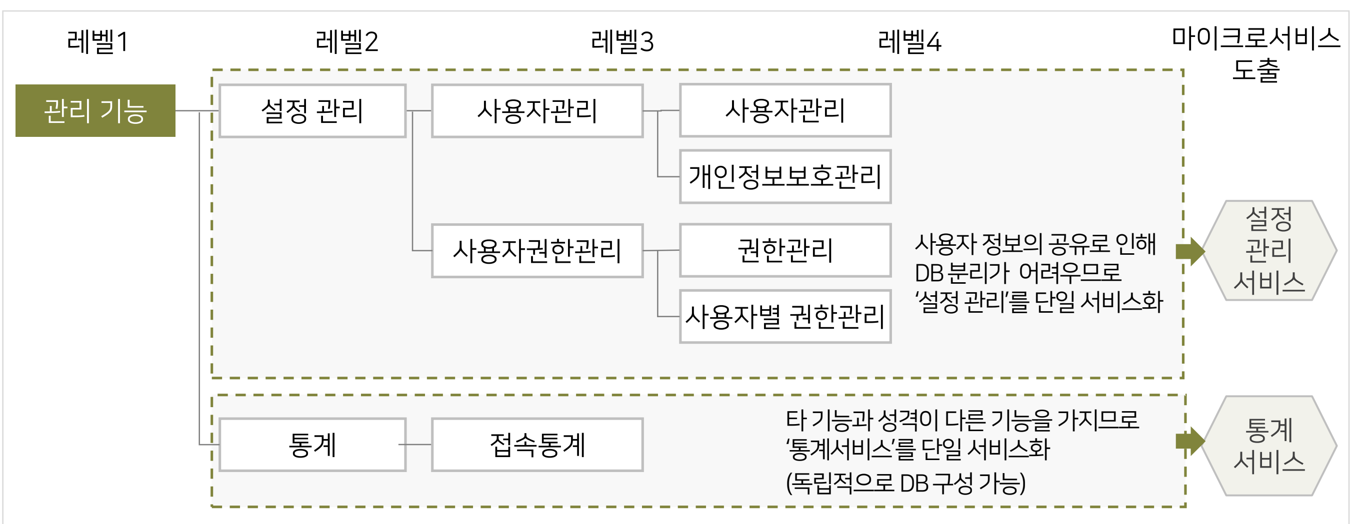
- **업무 프로세스가 잘 정리되어 있거나 기존의 정보시스템이 존재하는 경우, 업무기능을 분해함으로써 마이크로서비스를 도출할 수 있다.**
- **마이크로서비스는 애플리케이션의 기능체계를 토대로 기능 간 응집도와 결합도, 독립적 개발 및 배포, 데이터 분리 가능성 등을 고려하여 도출한다.**

[그림 7-17] 업무기능 분해 개념



- 비즈니스 업무를 분석하여 더 작은 업무로 나누고 그들 간의 계층구조 및 업무 간의 순서와 의존성을 분석하는 작업
- 정보시스템을 구축하는 가장 최하위 단위의 프로세스를 도출함
- 기관이 보유하고 있는 업무 프로세스 관련 자료 또는 애플리케이션의 맵 및 체계도를 활용할 수 있음

[그림 7-18] 업무기능 분해 기반 마이크로서비스 식별 원칙



- 애플리케이션의 하위 단위 프로세스(레벨 4)를 마이크로서비스로 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
  - 구현 가능한 공유 메소드는 마이크로서비스 내에 직접 구현하고, 공통 컴포넌트는 별도의 서비스로 구현
- 독립적으로 개발, 배포, 데이터(DB) 분리가 가능한 단위

07 클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.4 업무기능 분해를 통한 마이크로서비스 도출

7.1.4.2 업무기능 분해를 통한 마이크로서비스 도출 절차

- **마이크로서비스를 식별하기 위해 업무 프로세스 현황, 정보시스템 현황, 데이터 모델 산출물 등 기존 산출물을 분석한다.**
- **기존 산출물을 분석하여 애플리케이션의 업무기능 식별, 공통 기능 분리, 호출관계를 고려한 기능 결합 등을 고려하여 마이크로서비스를 도출한다.**

[그림 7-19] 마이크로서비스 도출 절차

기존 산출물
- 업무프로세스 현황 정보
- 정보시스템 현황 정보 (기능, 메뉴구성도)
- 데이터 모델 산출물 등

↓

업무기능 식별
- 애플리케이션의 전체 기능구조 식별

↓

공통 기능 분리
- 공통 기능과 개별 업무용 기능 분리
- 공통 기능은 별도 마이크로서비스로 식별

↓

호출관계 고려한 기능 결합
- 연동 호출, 데이터 참조 관계가 많은 기능 그룹핑
- DB 분리가 불가능한 기능 그룹핑

↓

마이크로서비스 도출

규모가 커지면 기능 재조정

도출 시 고려사항
- 식별된 마이크로서비스 단위로 DB를 물리적, 논리적으로 분리할 수 있어야 함
- 동일한 데이터(테이블)의 CRUD 성격의 거래가 서로 다른 서비스로 분리되지 않도록 해야 함
- 마이크로서비스 크기는 업무의 특성에 따라 달라 명확한 기준을 제시하기 어려우나 개발 주기에 따라 하나의 팀에서 2~4주 단위로 업데이트할 수 있는 크기가 적당함

※ CRUD(Create, Read, Update, Delete) : 데이터 생성, 읽기, 변경, 삭제 처리

07 클라우드 네이티브 정보시스템 분석 단계

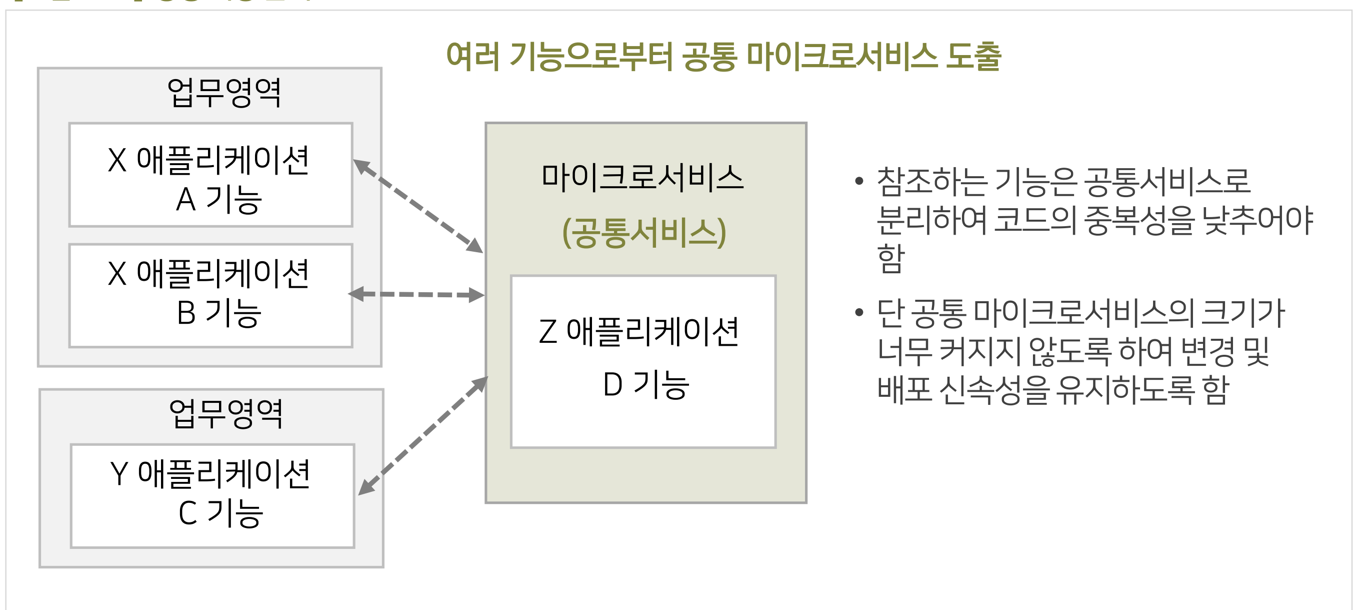
7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.4 업무기능 분해를 통한 마이크로서비스 도출

7.1.4.2 업무기능 분해를 통한 마이크로서비스 도출 절차

- **여러 업무기능으로부터 공통 마이크로서비스를 도출한 후 서비스 간 응집도, 결합도, 데이터 관점을 고려한 호출 관계를 파악하여 관련 기능을 결합할 수 있다.**

[그림 7-20] 공통 기능 분리



[그림 7-21] 복잡한 호출관계를 갖는 기능 결합

응집도, 결합도, 데이터 관점 기능 단위 재조정

마이크로서비스

X 애플리케이션

A 기능

직접호출

데이터 참조

메소드 공유

X 애플리케이션

B 기능

- 하나의 마이크로서비스 내부에서 업무처리가 완료될 수 있도록 독립성이 보장되어야 함
- 동일 애플리케이션 내부 기능 간 호출 관계가 있을 경우, 내부 기능 간 상호 참조 및 공유가 빈번한 경우는 하나의 마이크로서비스로 결합

클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.4 업무기능 분해를 통한 마이크로서비스 도출

7.1.4.2 업무기능 분해를 통한 마이크로서비스 도출 절차

- 업무 공통 기능의 분리, 복잡한 호출관계를 가지는 기능의 결합 등을 통해 재조정된 업무기능 분해도를 정리한 후, 마이크로서비스를 도출한다.
- 아래 그림은 주문 애플리케이션의 업무 기능 분해도의 레벨 2를 기준으로 마이크로서비스를 도출한 예시이다.

[그림 7-22] 마이크로서비스 도출 예시 – 주문 서비스

재조정된 업무기능 분해도

레벨 1	레벨 2	레벨 3
주문	주문관리	주문처리
		주문이력
	배송관리	배송이력
		배송처리
		배송업체관리
	상품관리	재고관리
		상품카탈로그
	결제관리	밴사관리
	결제처리	카드결제
		계좌이체
		포인트결제
	포인트관리	포인트적립

마이크로서비스 도출

- 주문 서비스
- 배송 서비스
- 배송업체 관리서비스
- 상품 서비스
- 결제 서비스
- 포인트 서비스

클라우드 네이티브 정보시스템 분석 단계

7.1 클라우드 네이티브 적용을 위한 마이크로서비스 도출

7.1.4 업무기능 분해를 통한 마이크로서비스 도출

7.1.4.2 업무기능 분해를 통한 마이크로서비스 도출 절차

- 아래 그림은 일반적인 홈페이지 애플리케이션의 기능을 하위 레벨로 분해하여 결합도, 응집도를 검토하여 마이크로서비스를 식별한 예시이다.
- 기본적으로 레벨 2 기능을 마이크로서비스로 식별하되 정보공개의 경우 레벨 3에 데이터 오너십, 배포 단위, 스케일링 등을 고려하여 정보공개 서비스와 공공데이터 개방 서비스로 분리하였다.

[그림 7-23] 마이크로서비스 도출 예시 – 홈페이지 서비스

홈페이지 기능 분해도

레벨 1 / 레벨 2 / 레벨 3

- 홈페이지
  - 정보공개
    - 정보공개제도 안내
    - 정보공개처리 현황
    - 공공데이터 개방
  - 알림
    - 보도자료
    - 입찰공고
  - 간행물
    - 정보화 백서
    - 정책 저널
    - ICT 동향 분석
  - 기관 소개

마이크로서비스 도출

- 정보공개 서비스
- 공공데이터 개방 서비스
- 알림 서비스
- 간행물
- 기관 소개

07 클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.1 개요

- 클라우드 네이티브 애플리케이션은 세부적인 기능 단위로 분해된 다수의 마이크로서비스가 컨테이너 형태로 독립적으로 실행되므로 분산된 마이크로서비스에 대한 통합적인 배포와 관리가 필요하다. 따라서 기존의 애플리케이션에 비해 다소 복잡한 아키텍처 환경과 기능 요소를 고려해야 한다.
- 아래 그림은 가트너, IBM 등에서 제시하는 클라우드 네이티브 참조 아키텍처로 API 게이트웨이, 서비스 메시, 런타임 플랫폼, CI/CD, 백엔드 서비스, 텔레메트리 서비스로 구성된다.

[그림 7-24] 클라우드 네이티브 애플리케이션 아키텍처 참조 모델

UI
컨테이너
컨테이너
K8S 클러스터 구성 :
(마스터노드/워커 노드)
- 운영환경
- 개발/검증 환경
범례
내부 아키텍처
외부 아키텍처
API
- REST API
1 API 게이트웨이
- 인증/인가
- API 조합
2 서비스 메시
- 서비스 발견
- 서비스 라우팅
- 로드밸런싱
- 설정관리
서비스 A
컨테이너
오토스케일링
서비스 B
컨테이너
3 런타임 플랫폼 (컨테이너)
- 컨테이너 관리
5 백엔드 서비스
DB
스토리지
MQ
캐시
4 CI/CD
- 형상관리
- 빌드 자동화
- 빌드 이미지 저장소
- 배포 자동화
6 텔레메트리[1]
- 모니터링

[출처 : 가트너, IBM 등 자료 분석 정리]

1) 텔레메트리(Telemetry, 원격 측정) : 관측 대상으로부터 이격된 지점에서 다양한 관측을 수행하고 그 데이터를 취득하는 기술을 의미하며, 클라우드 네이티브 애플리케이션은 후속 분석을 위해 중앙 집중화된 위치로 데이터를 자동으로 수집하고 전송하는 텔레메트리 기능이 필요함

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.2 클라우드 네이티브 애플리케이션 아키텍처 구성

- **클라우드 네이티브 애플리케이션은 내부의 서비스에 대한 단일한 접점을 외부로 제공하는 API 게이트웨이, 마이크로서비스의 제어를 담당하는 서비스 메시, 애플리케이션 실행 단위인 컨테이너의 실행환경 관리 기능을 제공하는 런타임 플랫폼, 마이크로서비스 단위의 독립적인 개발, 배포를 지원하는 자동화된 CI/CD, 애플리케이션이 구동하기 위한 지원기능을 제공하는 백엔드 서비스, 클라우드 기반 인프라와 가상화 환경의 애플리케이션에 대한 헬스체크 및 모니터링 환경 등 6개 요소로 구성된다.**

[표 7-1] 클라우드 네이티브 애플리케이션 아키텍처 구성요소

구분	설명
1 API 게이트웨이	• 내부의 서비스를 API 형태로 외부에 제공하기 위한 단일한 접점을 제공하며, 외부의 서비스 요청에 대한 라우팅뿐만 아니라 프로토콜 변환, 인증/인가, 사용량 제한, 로깅 및 API 조합 기능 수행 • 일반적으로 API 게이트웨이 솔루션을 적용하여 기능 구현
2 서비스 메시	• 마이크로서비스 간 내부 통신을 위한 가상의 인프라 계층을 제공하며 이를 통해 다양한 서비스 제어 및 관리 기능 수행 • 클라우드 네이티브 환경에서 마이크로서비스가 증가하여 복잡한 구조가 되면서 서비스들 간의 통신을 통제하고 생명주기를 관리할 수 있는 서비스 메시의 중요성 확대
3 런타임 플랫폼	• 컨테이너 실행환경과 컨테이너의 배포, 관리, 확장, 네트워크 자동화 등의 컨테이너 오케스트레이션[1]을 제공 • 도커 등의 컨테이너 제공 솔루션과 쿠버네티스 등의 컨테이너 오케스트레이션 솔루션을 통해 구축 • **내부 아키텍처** : 마이크로서비스 단위, 서비스 설계 패턴 및 구조 등의 애플리케이션 내부와 관련된 아키텍처 • **외부 아키텍처** : 클라우드 네이티브 애플리케이션의 개발, 배포, 실행환경 등 외적인 아키텍처
4 CI/CD	• CI/CD는 애플리케이션 개발 단계를 자동화하여, 애플리케이션을 보다 짧은 주기로 고객에게 제공하는 방법으로 배포가 잦은 MSA 기반 클라우드 네이티브 애플리케이션에 필수적인 요소
5 벡엔드 서비스	• 애플리케이션을 실행하기 위해서 네트워크를 통해서 사용할 수 있는 모든 서비스를 말하며, MySQL과 같은 데이터베이스, 캐시 시스템, SMTP[2] 서비스 등 애플리케이션과 통신하는 부가적인 자원들을 지칭하는 포괄적인 개념
6 텔레메트리	• 분산환경에서 서비스들을 모니터링하고, 서비스별로 발생하는 이슈들에 대응할 수 있도록 환경을 구성하는 역할

1) 컨테이너 오케스트레이션(Orchestration) : 컨테이너의 배포, 관리, 확장, 네트워킹을 자동화하는 것을 말함
2) SMTP(Simple Mail Transfer Protocol) : 간이 전자 우편 전송 프로토콜로 인터넷에서 이메일을 보내기 위한 프로토콜임

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.3 API 게이트웨이

7.2.3. 1 API 게이트웨이 개념

- API 게이트웨이(Gateway)는 클라이언트(UI)와 API 형태로 인터페이스를 제공하는 마이크로서비스 사이의 단일한 접점으로서 클라이언트 요청에 대한 라우팅(전달) 기능을 수행하는 서버(서비스)이다.
- 사용자(UI)와 API을 제공하는 내부 마이크로서비스의 중간에 위치하여 마이크로서비스에 대한 단일한 접점을 외부에 제공한다. 즉 API 중개자로서 다수 API 서버의 관리와 모니터링을 용이하게 한다.
- 그리고 내부의 서비스로 외부로부터 들어오는 접근을 내부 구조를 드러내지 않고 처리하기 위한 요소이다. 사용자 인증(Consumer Identity Provider)과 권한 정책관리(Policy Management)를 수행한다.
- API 게이트웨이를 사용하는 주된 이유 중 하나는 여러 백엔드 서비스를 호출하고 결과를 집계하고 분석할 수 있으며, 마이크로서비스에 릴레이되는 요청 수를 파악할 수 있기 때문이다.

[그림 7-25] API 게이트웨이 적용 구조 및 역할

UI (App)	API 게이트웨이	마이크로서비스	데이터베이스
사용자: UI 외부시스템: UI	API 게이트웨이	마이크로서비스 eGovFrame (×4)	DB (×4)
• 사용자 혹은 외부 시스템에서 서비스 요청 • API 게이트웨이를 통하여 마이크로서비스를 호출하는 API의 사용자	• 요청에 알맞은 서비스로 라우팅(전달) • 인증/권한관리, 사용자 관리, API 관리 등의 부가 기능 수행	• API 서비스 제공자로 REST 기반의 서비스 노출 • 데이터베이스 관련 처리 수행	

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.3 API 게이트웨이

7.2.3.2 API 게이트웨이 주요 기능

- **API 게이트웨이는 클라우드 네이티브 애플리케이션의 주요 특징 중 하나인 경량 메커니즘(HTTP)으로 통신하는 독립적인 서비스 환경을 구축하는 데 핵심적인 역할을 수행한다.**
- **단순히 서비스 요청에 대한 라우팅 기능뿐만 아니라 프로토콜 변환, 인증/인가, 사용량 제한, 로깅 및 API 조합 기능 등의 복합적인 기능을 수행한다.**

[표 7-2] API 게이트웨이 주요 기능

구분	기능 설명
인증(Authentication) 및 인가(Authorization)	• 인증 : 요청한 클라이언트의 신원을 확인 • 인가 : 특정 작업을 수행하도록 허가 받은 클라이언트인지 확인 • MSA에서 각 서비스의 API 호출에 대해 인증 및 인가를 하는 것은 같은 소스코드를 서비스 인스턴스마다 심어주어야 한다는 것을 의미 • API 게이트웨이에서 인증서 관리, 인증, SSL[1], 프로토콜 변환과 같은 기능을 수행함으로써 각각의 서비스 부담을 줄이고 서비스 관리 및 업그레이드를 보다 쉽게 할 수 있음
요청 라우팅	• API 게이트웨이의 가장 기본적인 기능 • 요청이 들어오면 라우팅 정보를 찾아서 어느 서비스로 요청을 보낼지 결정 • 만약 API 게이트웨이가 없다면 클라이언트에서 여러 서비스들에 대해 요청을 진행해야 함
프로토콜 변환	• 외부에서 들어오는 다양한 프로토콜(http, RPC[2], TCP[3] 등)과 컨텐츠 타입(JSON[4], XML[4]등)을 내부의 마이크로서비스가 사용하는 프로토콜 및 컨텐츠 타입으로 변경
로드밸런싱	• 서비스 인스턴스에 대한 부하분산 가능 • 클라이언트와 백엔드 간의 API 통신량을 줄여주어 대기시간을 줄이고 효율성을 높임
서비스 오케스트레이션	• 여러 개의 마이크로서비스를 묶어 새로운 서비스를 만드는 개념 • 클라이언트의 한 번의 요청에 여러 서비스에 분산된 데이터를 조합하여 응답 • 서비스 메시 기능과 유사
서비스 디스커버리 (Discovery)	• 클라우드 환경에서 서비스의 위치(IP 주소와 포트번호)를 탐색하는 기능 • API 게이트웨이에서는 서버 측 또는 클라이언트 측 기준으로 디스커버리 기능을 구현할 수 있음
기타	• 캐싱 : 서비스 요청 횟수를 줄이고자 응답을 캐시 • 지표 수집 : 과금 분석용 API 사용 지표 수집

1) SSL(Secure Sockets Layer) : 웹사이트와 브라우저 사이에 전송된 데이터를 암호화한 인터넷 기반 보안 프로토콜
2) RPC(Remote Procedure Call) : 별도의 원격 제어를 위한 코딩 없이 다른 주소 공간에서 함수나 프로시저를 실행할 수 있게 하는 프로세스 간 통신 기술
3) TCP(Transmission Control Protocol) : 전송계층(4계층) 프로토콜 중 하나로 IP 위에서 특정 프로세스까지 패킷을 전달하기 위한 프로토콜
4) JSON(JavaScript Object Notation) : 자바스크립트 객체 문법으로 구조화된 데이터를 표현하기 위한 문자 기반의 표준 포맷
5) XML(eXtensible Markup Language) : W3C에서 개발된, 다른 특수한 목적을 갖는 마크업 언어를 만드는데 사용하도록 권장하는 다목적 마크업 언어

07 클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.3 API 게이트웨이

7.2.3.3 API 게이트웨이 주요 솔루션

- 클라우드 네이티브 아키텍처에서 API 라우팅과 관리를 담당하는 API 게이트웨이의 중요성은 점점 더 커지고 있다. 또한 시스템이 복잡해지고 하이브리드 클라우드와 멀티 클라우드의 활용이 늘어남에 따라 단일 API 게이트웨이가 아닌 멀티 API 게이트웨이의 사용이 증가하는 추세에 있다.
- 클라우드 환경에서 사용될 수 있는 오픈소스 솔루션과 상용 솔루션이 존재한다.
- 이중에서 오픈소스 중심으로 주요 솔루션의 특징과 기능을 살펴보도록 한다.

가. 스프링 클라우드 게이트웨이

- 스프링 클라우드 게이트웨이는 스프링 및 자바 위에 API 게이트웨이를 빌드하기 위한 라이브러리를 제공한다. 여러 기준에 따라 요청을 라우팅하는 유연한 방법을 제공할 뿐만 아니라 보안, 탄력성 및 모니터링과 같은 교차 문제에 중점을 두고 있다.
- 스프링 클라우드 게이트웨이는 스프링 리액티브(Spring Reactive) 생태계 위에 구현된 API 게이트웨이로 게이트웨이 핸들러 매핑을 사용하여 들어오는 요청을 적절한 대상으로 라우팅하는 간단하고 효과적인 방법을 제공한다.
- 스프링 클라우드 게이트웨이는 넷플릭스 줄 서버와 동일하게 API 게이트웨이의 기능을 제공하면서 동작 원리에서 줄 서버의 단점을 보완하였다.
- 라이선스 : 아파치 라이선스(Apache License) 2.0

[표 7-3] 스프링 클라우드 게이트웨이와 넷플릭스 줄서버의 차이점

구분	스프링 클라우드 게이트웨이	넷플릭스 줄서버
블로킹 (Blocking)	• 블로킹(Blocking) - 요청을 보내고 응답이 올 때까지 다음으로 진행하지 않고 기다림	• 논블로킹(Non-Blocking) - 요청을 보내고 바로 다음으로 진행하여 다른 일을 하다가 응답이 오면 그에 맞는 처리
동작 원리	• 필터(Filter)들만으로 동작	• 조건자(Preficates, 수행을 위한 사전 요구 조건)와 필터(Filter)를 조합하여 동작
WEB/WAS	• 톰캣(Tomcat) 사용 - 서블릿 컨테이너만 있는 웹 애플리케이션 서버	• 넷티(Netty) 사용 - 비동기 네트워킹을 지원하는 애플리케이션 프레임워크

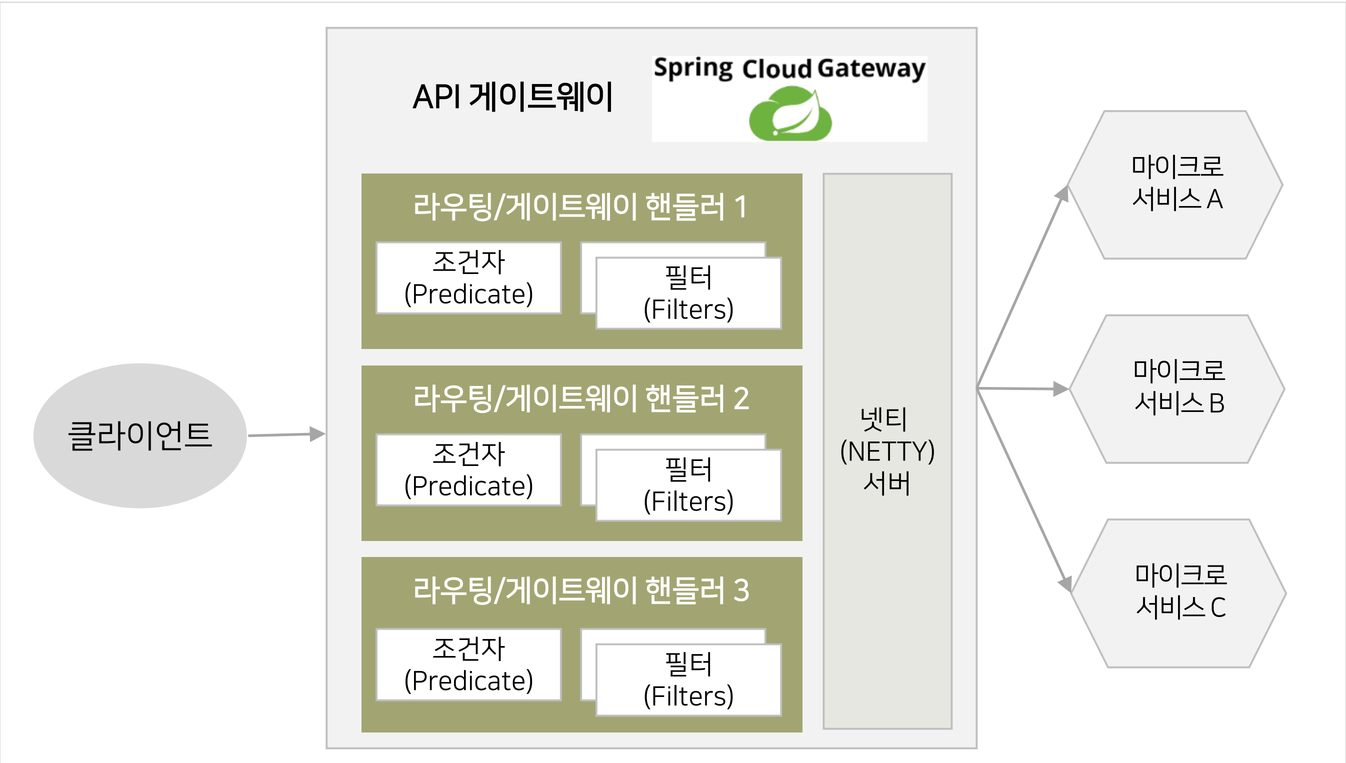
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.3 API 게이트웨이

7.2.3.3 API 게이트웨이 주요 솔루션

가. 스프링 클라우드 게이트웨이

[그림 7-26] 스프링클라우드 게이트웨이 아키텍처



구분	설명
라우터 (Router)	• 라우터는 목적지 URI(Uniform Resource Identifier, 통합 자원 식별자), 조건자 목록과 필터의 목록을 식별하기 위한 고유 ID로 구성 • 라우터는 모든 조건자가 충족되었을 때만 매칭
조건자 (Predicate)	• 각 요청을 처리하기 전에 실행되는 로직, 헤더와 입력된 값 등 다양한 HTTP 요청이 정의된 기준에 맞는지를 찾음
필터 (Filter)	• HTTP 요청 또는 나가는 HTTP 응답을 수정할 수 있게 함 • 다운스트림 요청을 보내기 전이나 후에 수정할 수 있음 • 라우트 필터는 특정 라우트에 한정

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

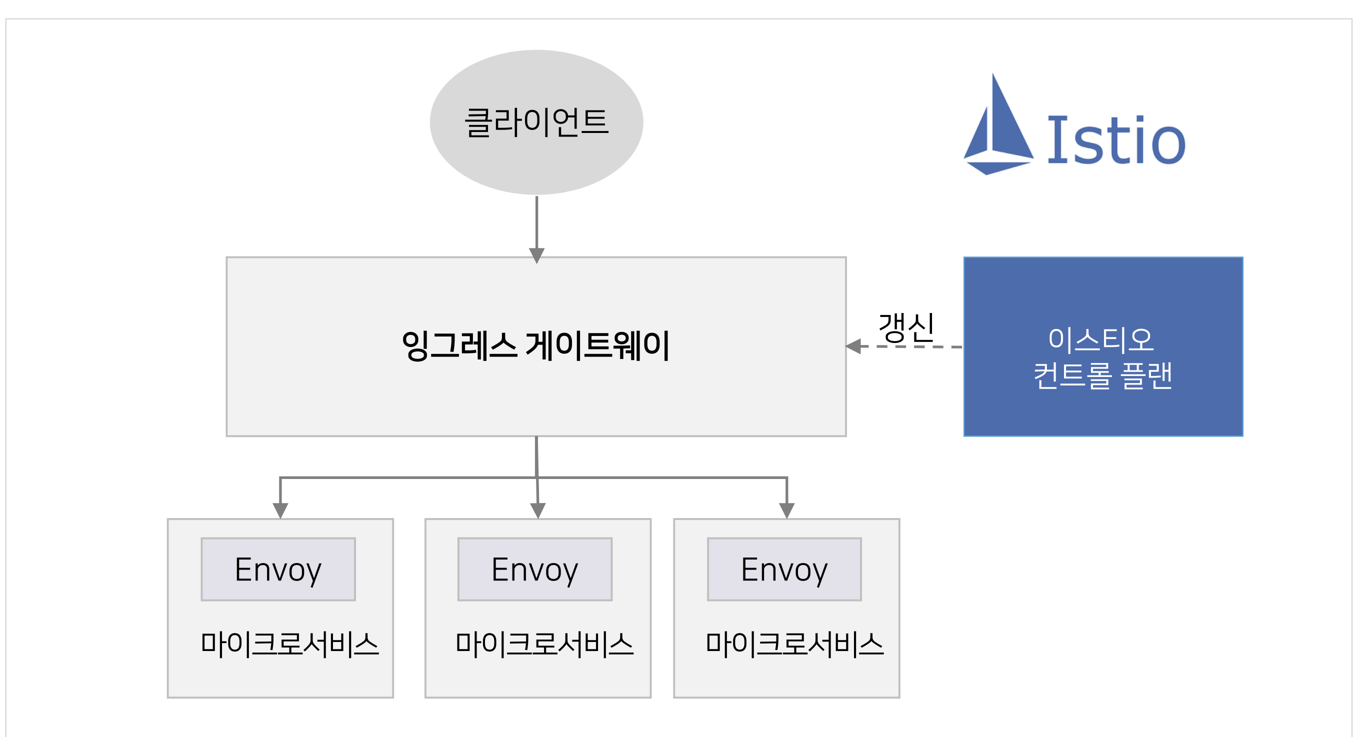
7.2.3 API 게이트웨이

7.2.3.3 API 게이트웨이 주요 솔루션

나. 이스티오(Istio)

- **쿠버네티스 클러스터에서는 쿠버네티스 오브젝트인 잉그레스 컨트롤러(Ingress Controller)를 내부 서비스 의 접점 역할, 즉 API 게이트웨이로 사용한다.**
- **이스티오는 서비스 메시를 제공하는 솔루션으로 잉그레스 컨트롤러 역할을 수행하는 이스티오 게이트웨이를 포함하고 있다. 이를 통해 쿠버네티스 클러스터에서 이스티오로 배포된 마이크로서비스에 대한 진입점 역할을 수행한다.**
- **즉, 이스티오는 서비스 메시 기능 일부로 게이트웨이를 제공하고 있으며 쿠버네티스 및 이스티오를 통한 서비스 메시를 적용한 상태에 적합하다.**
- **라이선스 : 아파치 라이선스 2.0**

[그림 7-27] 이스티오 API 게이트웨이 아키텍처



1) TLS(Transport Layer Security) : 인터넷에서의 정보를 암호화해서 송수신하는 전송 계층 보안 기술

클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

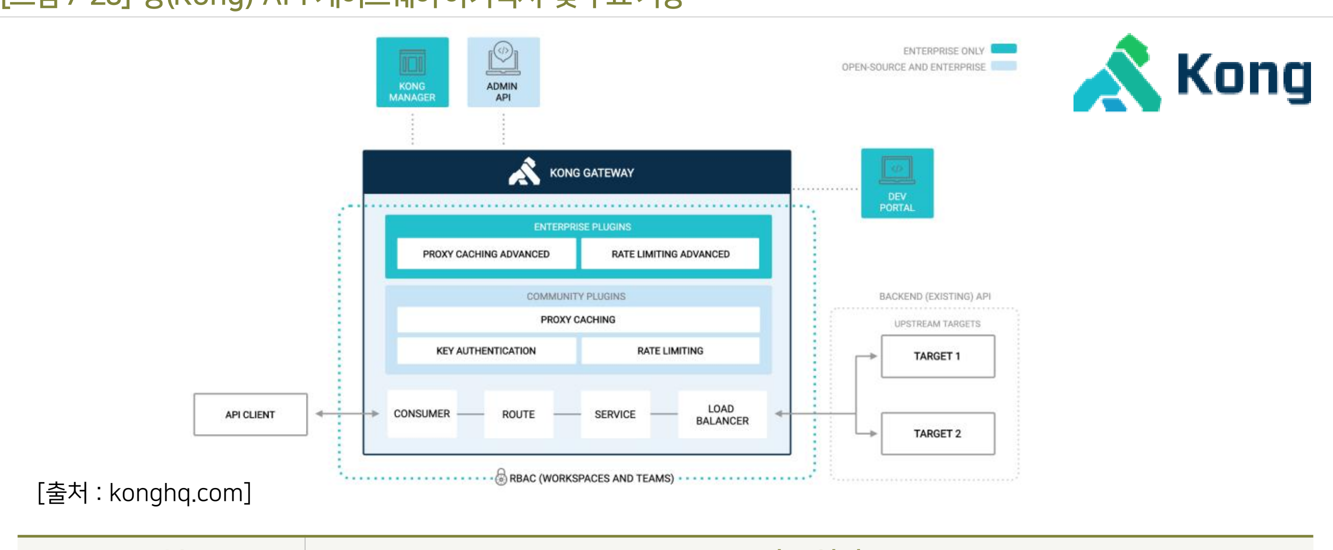
7.2.3 API 게이트웨이

7.2.3.3 API 게이트웨이 주요 솔루션

다. 콩(Kong) 게이트웨이

- **콩 API 게이트웨이는 하이브리드 또는 멀티 클라우드 등 다양한 클라우드 환경에 최적화되어 있는 API 게이트웨이이다.**
- **콩 엔진엑스(Nginx, 경량 웹서버), 카산드라(Cassandra, 분산형 NoSQL 데이터베이스)+루아(Lua, 스크립트 언어)를 기반으로 하며, 고성능 및 확장성을 갖추고 있다.**
- **클라우드 네이티브 및 쿠버네티스 네이티브를 지향하여 아마존 AWS, MS AZURE, IBM 클라우드 등 글로벌 클라우드 플랫폼용 배포 버전을 포함하고, 프라이빗 클라우드 플랫폼을 위한 도커 및 쿠버네티스 잉그레스용 배포 버전을 지원하고 있다.**
- **라이선스 : 아파치 라이선스 2.0, 콩 엔터프라이즈**

[그림 7-28] 콩(Kong) API 게이트웨이 아키텍처 및 주요 기능



[출처 : konghq.com]

구분	기능 설명
오픈소스 버전	• 서비스 및 경로 개체를 사용하여 서비스 노출 • 속도 제한 및 프록시 캐싱 설정 • 키 인증을 통한 인증 • 로드밸런싱
엔터프라이즈 버전	• 관리자 기능 : 모니터링하고 관리하기 위한 시각적 브라우저 기반 도구 • 역할 기반 권한 관리 • 개발자가 서비스를 찾고 액세스하고 이용할 수 있는 개발자 포털

07 클라우드 네이티브 정보시스템 분석 단계

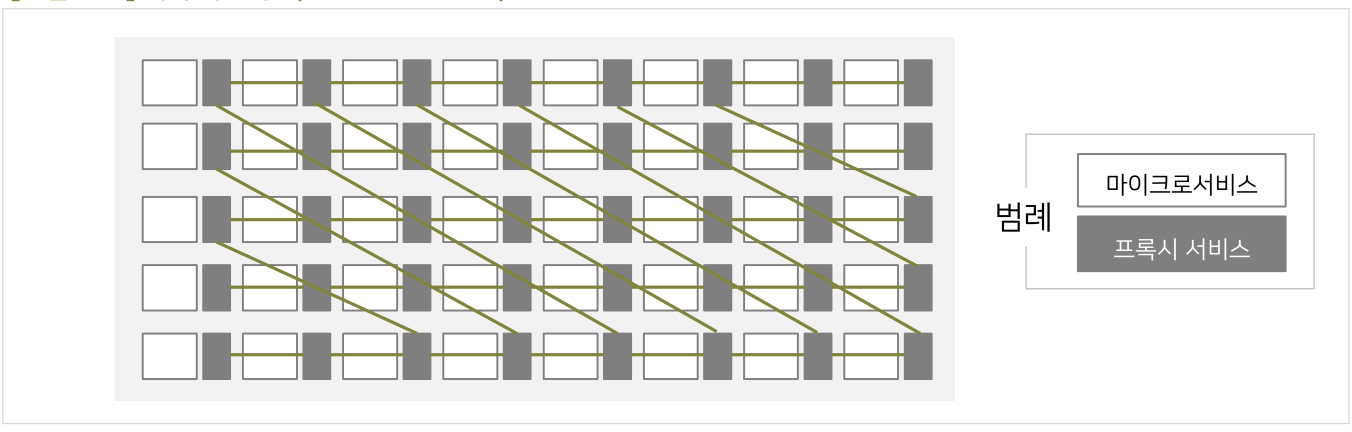
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.4 서비스 메시

7.2.4.1 서비스 메시(Service Mesh) 개념

- 마이크로서비스의 도입으로 수백~수천 개의 서비스 인스턴스가 동작하고 PaaS 환경에서 설정에 따라 많은 인스턴스가 동적으로 뜨고 사라지게 된다.
- 서비스 메시는 마이크로서비스 간 안정적인 통신을 위한 가상의 인프라 계층이다. 수많은 마이크로서비스 의 제어와 관리를 위해 서비스 간 통신이 중요해졌으며 서비스 메시는 이러한 서비스 간 내부 통신 인프라를 제공한다.

[그림7-29] 메시 네트워크(Mesh Network)



- 서비스 메시의 구현은 기본적으로 하나의 마이크로서비스의 앞단에 경량화 프록시를 배치하여 서비스 간 통신을 제어하는 방식을 사용한다.
- 프록시에 라우팅 규칙, 타임아웃, 재시도 횟수 등을 설정하여 마이크로서비스의 비즈니스 로직과 분리할 수 있어 기존의 서비스에 영향을 주지 않고 서비스를 제어할 수 있다.

[그림 7-30] 서비스 메시 구현 방식

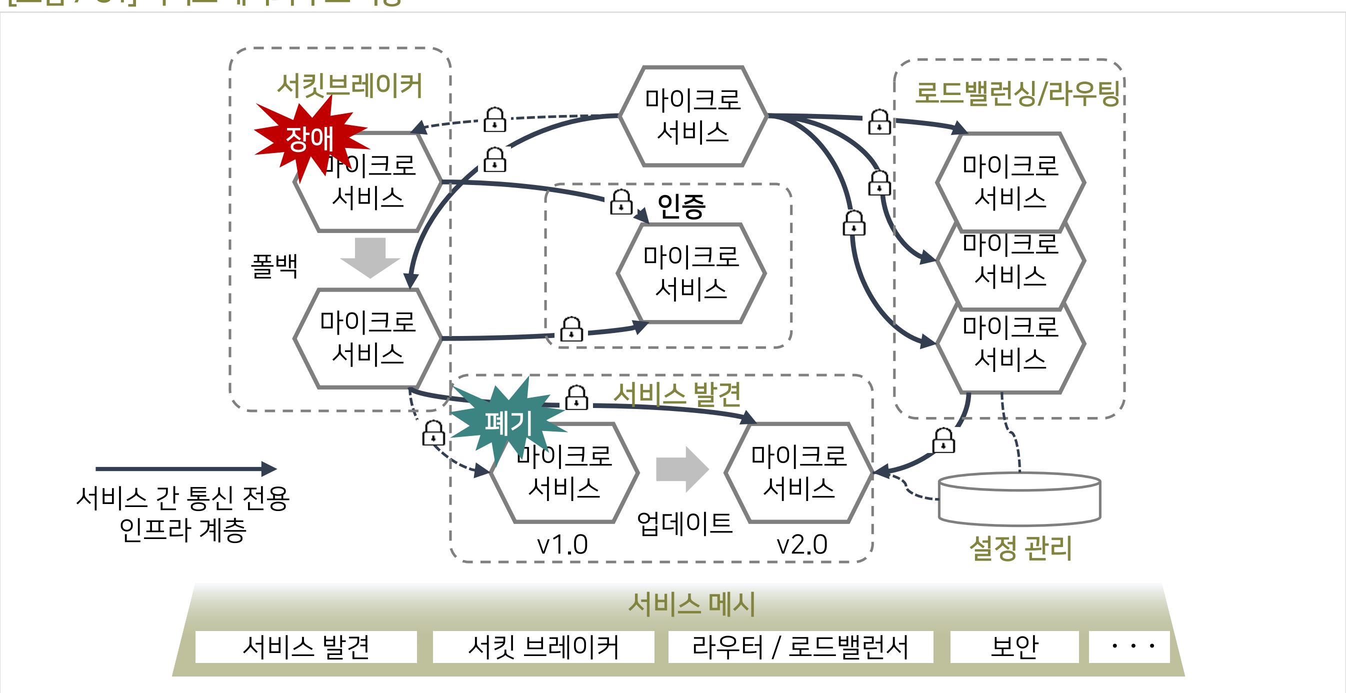
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.4 서비스 메시

7.2.4.2 서비스 메시의 주요 기능

- 클라우드 네이티브 환경에서는 작은 단위의 마이크로서비스가 컨테이너라는 독립된 실행환경에서 서비스를 제공하기 때문에 복잡한 환경과 기술적 구성요소를 필요로 한다. 즉 다수의 서비스의 운영을 위해 서비스 디스커버리, 서비스 라우팅, 로드밸런싱, 장애 회피 등의 기능이 필요하며 서비스 메시는 이러한 서비스 관리, 제어를 위한 실행환경을 제공한다.

[그림 7-31] 서비스 메시의 주요 기능



구분	기능 설명
서비스 디스커버리	• 서비스 레지스트리를 통한 서비스 검색을 통해 마이크로서비스의 상태와 IP를 관리, 실시간 정상 서비스 정보 제공
서비스 라우팅	• 클라이언트의 요청을 확인된 정상 서비스(컨테이너)로 전달
로드밸런싱	• 서비스 인스턴스가 다수 개로 구성된 경우 트래픽을 분산 처리
환경 설정	• 환경 정보를 컨테이너 실행환경과 분리하여 외부 저장소에 보관
인증 및 인가	• 서비스 통신의 인증, 권한 부여 및 암호화 등을 관리
서킷 브레이커	• 특정 서비스의 장애나 응답 지연이 다른 서비스로 전파되지 않도록 장애 회피 및 격리

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.4 서비스 메시

7.2.4.3 서비스 메시의 종류

- **마이크서비스 기반의 애플리케이션을 개발하고 서비스 메시를 구현하는 데 있어서 소스코드와 서비스 메시를 구현하는 코드가 얼마나 밀접하게 엮여 있는지 정도에 따라 아래와 같이 3가지 유형으로 분류할 수 있다.**
- **기존에는 애플리케이션 소스코드에 서비스 메시를 위한 코드를 삽입하는 방식이었으나 최근에는 기존의 서비스와 독립적인 서비스 메시를 위한 별도의 프록시 서비스 구현방식으로 발전하고 있다.**

[그림 7-32] 서비스 메시 기술의 스펙트럼

A Spectrum of Service Mesh Technologies Based on Code Coupling

Service Code Tightly Coupled to Mesh | Service Code Independent of Mesh

Microsoft Azure Service Fabric, lagom, [M] Go-Micro, SENECA, AxonFramework CQRS FRAMEWORK FOR JAVA

finagle, Spring Cloud, steeltoe, NETFLIX OSS Ribbon/ Hystrix/ Eureka/ Archaius, Hydra

NETFLIX OSS Prana

Istio, envoy, linkerd, AVI Networks, ASPEN MESH, Consul, ContainerPilot

1 Mesh-Native Code	2 Mesh-Aware Code		3 Mesh-Agnostic Code
Service Code	Service Code	Service Code	Service Code
Framework Programming Model Framework Runtime	Framework Libraries	API Sidecar Process Framework Libraries	Sidecar Proxy (Egress & Ingress)
Mesh Services	Mesh Services	Mesh Services	Mesh Services

Configuration, Discovery, Routing, Load Balancing, Instrumentation, Resiliency, AuthN/AuthZ, Security

구분	1 Mesh-Native Code	2 Mesh-Aware Code	3 Mesh-Agnostic Code
설명	• PaaS의 일부로 서비스 코드에 포함되는 유형 • 프레임워크 기반의 프로그래밍 모델이기 때문에 서비스 메시를 구현하는 데에 특화된 코드가 필요	• 라이브러리로 구현되어 API 호출을 통해 서비스 메시에 결합되는 유형 • 프레임워크 라이브러리를 사용하는 형태로 서비스 메시를 이해하고 코드를 작성해야 함	• 사이드카(Side car) 프록시를 이용하여 서비스 메시를 마이크로서비스에 주입하는 유형 • 사이드카 프록시 형태로 동작되어 서비스 메시와 무관하게 코드를 작성
코드 의존도	• 공급업체 및 특정 코드에 의존	• 라이브러리를 이용하여 코드를 구현해야 하며, 일부는 속성(Properties)으로 관리되어야 함	• 코드영향도는 없으나, 설정 파일(yml)로 배포되어야 함
SW	• Microsoft Azure Service fabric, lagom, SENECA 등	• Spring Cloud, Netflix OSS 등	• Istio, Consul, Linkerd 등

[출처 : 코드 결합에 기반한 서비스 메시 기술의 스펙트럼, 2018, Gartner]

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.4 서비스 메시

7.2.4.4 서비스 메시와 API 게이트웨이 비교

- **API 게이트웨이와 서비스 메시가 하는 일은 라우팅, 인증, 모니터링, 서비스 검색, 서비스 등록 등으로 동일하지만 외부에 노출되는 것과 작동 위치에서 차이점이 있다.**
- **API 게이트웨이의 적용 위치는 클라이언트-to-서버이고 외부 노출이 되지만, 서비스 메시의 적용 위치는 서버-to-서버로 외부 노출은 없다.**
- **서비스 메시와 API 게이트웨이는 동작 위치, 라우팅, 핵심 기능, 장애 발생, 분석, 구현 방식 관점에서 다음과 같은 차이점이 존재한다.**

[표 7-4] 서비스 메시의와 API의 비교

구분	서비스 메시	API 게이트웨이
동작 위치	• 내부망(쿠버네티스 클러스터)	• 외부망과 내부망 사이
라우팅	• 서비스 간 처리 시 로컬 네트워크 스택의 일부로 처리(사이드카 패턴)	• 별도 네트워크에 도입되는 독립적인 API 게이트웨이 구성
핵심 기능	• 내부 애플리케이션 간 통신을 쉽고 안전하게 제공 • 서비스 코드에서 대부분의 애플리케이션 네트워크 기능을 분리하고 오프로드할 수 있는 통신 인프라	• 외부의 요청을 내부 서비스로 안전하게 전달 • 서비스를 관리형 API로 노출
장애 발생	• 서비스별로 요청이 분산되어 쉽게 확장 가능한 구조(API 게이트웨이처럼 외부에서 접속하는 부분의 확장 중요)	• API 게이트웨이에 모든 요청이 집중(SPOF, Single Point Of Failure)
분석	• 내부망에 위치하며, 애플리케이션의 네트워크 경계에서 통신	• API에 대한 사용자/공급자의 호출에 대해 수집, 분석 가능
구현 방식	• 원하는 애플리케이션 간의 네트워크를 정의하여 배포(쿠버네티스 기준)	• 클러스터 앞단에 API 게이트웨이를 구현하여 적용

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.4 서비스 메시

7.2.4.4 서비스 메시와 API 게이트웨이 비교

- **서비스 메시와 API 게이트웨이는 동시에 작동될 수 있다. 서킷 브레이커 등 일부 중복 기능이 존재하지만 근본적으로 두 개념은 다른 기능을 제공한다.**
- **아래의 이미지는 서비스 메시와 API 게이트웨이가 동시에 동작되는 구조를 설명한 것이다.**

[그림 7-33] 서비스 메시와 API 게이트웨이의 동작 구조

대부분의 API 게이트웨이는 내장된 서비스 메시 기능을 가짐

API 게이트웨이

API 서비스 | API 서비스 | API 서비스

서비스 메시

마이크로서비스 통합/조합

비즈니스로직 / 서비스 메시 (×3)

비즈니스 로직 조합

기본 네트워크 기능

애플리케이션네트워크 기능

핵심/단위 마이크로서비스

비즈니스로직 / 서비스 메시 (×5)

- 서비스 메시는 대부분의 서비스 구현과 함께 사이드카로 사용되며 서비스의 비즈니스 기능과 독립적임
- **API** 게이트웨이는 모든 **API** 서비스(명확하게 정의된 비즈니스 기능)를 호스팅하며 솔루션 비즈니스 기능의 일부임
- **API** 게이트웨이에는 서비스 간 통신 기능이 내장되어 있을 수 있지만 **API** 게이트웨이가 서비스 메시를 사용하여 다운스트림 서비스(**API** 게이트웨이 → 서비스 메시 → 마이크로서비스)를 호출하는 것을 막지는 않음

[출처: Kasun Indrasiri, Service Mesh vs API Gateway]

클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.1 컨테이너 정의 및 구조

- **컨테이너란 경량화된 가상화 기술로서 호스트 서버의 운영체제(OS) 수준에서 프로세스로 격리된 환경을 제공한다.**
- **리눅스 컨테이너 기술을 바탕으로 호스트 OS의 CPU, 네트워크 I/O, 블록 I/O, 메모리 등의 자원을 커널 레벨에서 격리시켜 프로세스와 네임스페이스를 호스트 시스템으로부터 독립적으로 동작하도록 하여 추가적인 작업 없이 프로세스를 실행할 수 있다.**

[그림 7-34] 컨테이너 기술 구조

1) LXC(Linux Containers)는 단일 컨트롤 호스트 사이에서 여러 개의 고립된 리눅스 시스템(컨테이너)들을 실행하기 위한 리눅스 커널 수준의 가상화 방법

2) IPC(Inter-Process Communication) : 프로세스들 사이에 서로 데이터를 주고받는 행위 또는 그에 대한 방법이나 경로를 뜻함

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.1 컨테이너 정의 및 구조

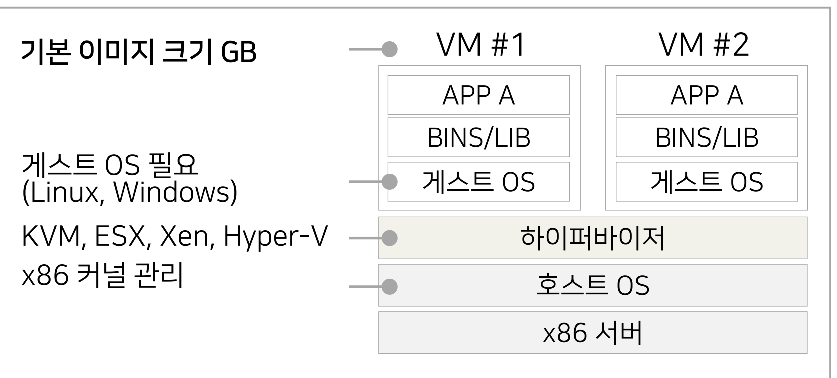
- **클라우드 환경에서 애플리케이션 구축 용도에 따라 가상화 기술이 다르게 적용된다. 서버 가상화 기술인 하이퍼바이저 방식은 2000년 초반부터 널리 이용되고 있고, OS 커널을 공유하는 컨테이너 방식은 리눅스 기반 시스템에서 프로세스 간 격리를 위해 사용하던 기술들을 조합하여 발전시킨 것이다.**

[그림 7-35] VM 방식과 컨테이너 방식의 비교

VM 방식 - 하드웨어 수준의 가상화

아키텍처 특징

- 호스트 서버에 개별적인 OS를 가진 서버 구성
  → 수직적 가상화 방식

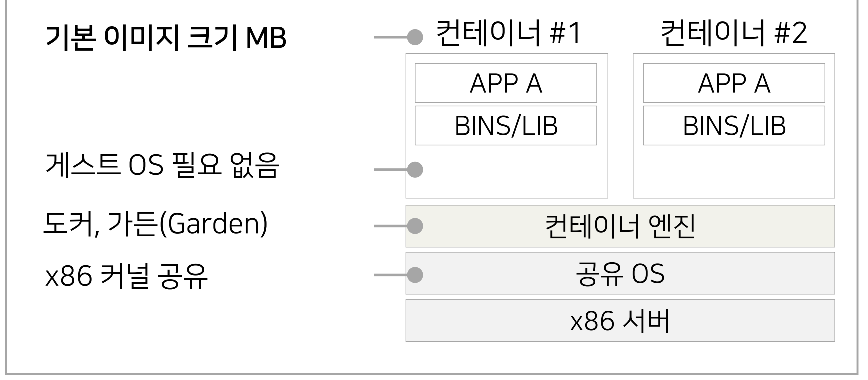


- VM별 OS, 애플리케이션 구성
- 하드웨어 가상화 기반의 애플리케이션 런타임
- 하이퍼바이저를 통한 호스트 OS와 VM 레이어의 커널 제어
- 배포 빈도가 적고, 보안이 중요한 시스템에 적합
- 가상화 기반 인프라 전환이 특징인 IaaS 유형에 적합

컨테이너 방식 - OS 수준의 가상화

아키텍처 특징

- 호스트 서버에 컨테이너 프로세스별로 논리적으로 격리
  → 수평적 분할 방식



- 컨테이너 단위로 배포되는 개별 런타임 스택을 통해 패키지 형태의 애플리케이션 구성
- 배포 빈도가 높고, 실시간 처리 시스템에 적합
- 플랫폼, 개발환경 표준화 및 애플리케이션 경량화 기반 PaaS에 적합

클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

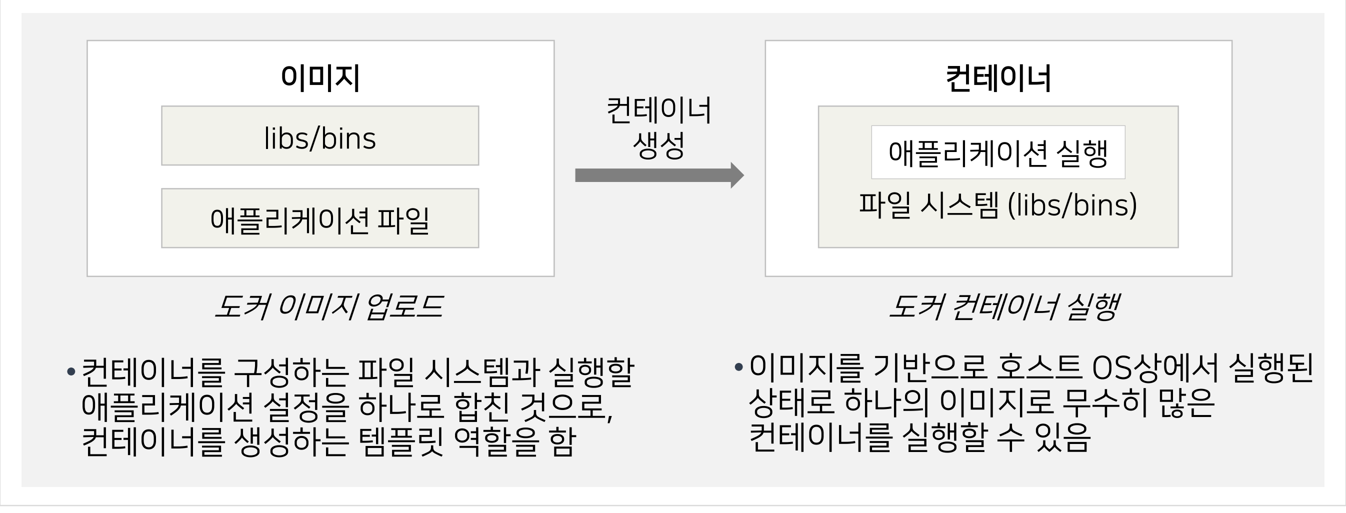
7.2.5.2 컨테이너 솔루션 - 도커

- **도커(Docker)란 리눅스 컨테이너 기술을 적용한 가장 대표적인 컨테이너 제공 솔루션이다. 도커는 컨테이너 이미지를 생성하고, 관리하고, 공유하고, 여러 곳으로 옮기고, 도커 호환 호스트에 배치해 컨테이너를 구동하는 작업을 수행한다.**
- **도커는 도커 이미지, 이미지 저장소, 컨테이너 실행환경으로 구성되며, 도커 이미지에서 도커 컨테이너 프로세스가 실행되는 구조이다.**

[표 7-5] 도커의 구성

구분	설명
도커 이미지	• 컨테이너 실행에 필요한 모든 파일과 설정 값 등을 포함하고 있는 실행파일 집합 • 개발 바이너리와 환경 설정을 패키징 • 도커 파일로 이미지 생성 과정을 작성한 후 빌드 명령어를 통해 이미지 생성
이미지 저장소 (Registry)	• 생성한 도커 이미지의 저장, 공유를 위한 저장소 • 개발자가 도커 이미지를 업로드(Push)하고, 도커 실행 시 다운로드(Pull) • 공개 도커 저장소(hub.docker.com)와 사설 도커 저장소 이용 가능
컨테이너 실행환경	• 도커 이미지를 기반으로 실행되는 일반 리눅스 컨테이너 • 컨테이너는 도커 실행 호스트에서 실행되는 하나의 프로세스 • 컨테이너는 호스트 및 타 프로세스와 격리

[그림 7-36] 이미지와 컨테이너



07 클라우드 네이티브 정보시스템 분석 단계

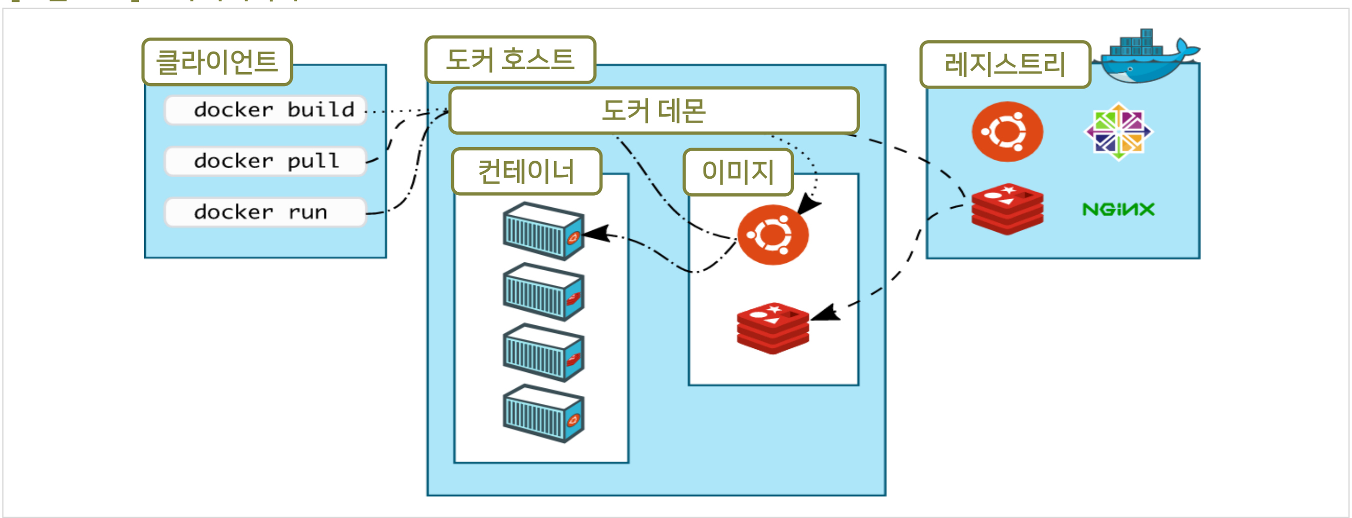
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.2 컨테이너 솔루션 - 도커

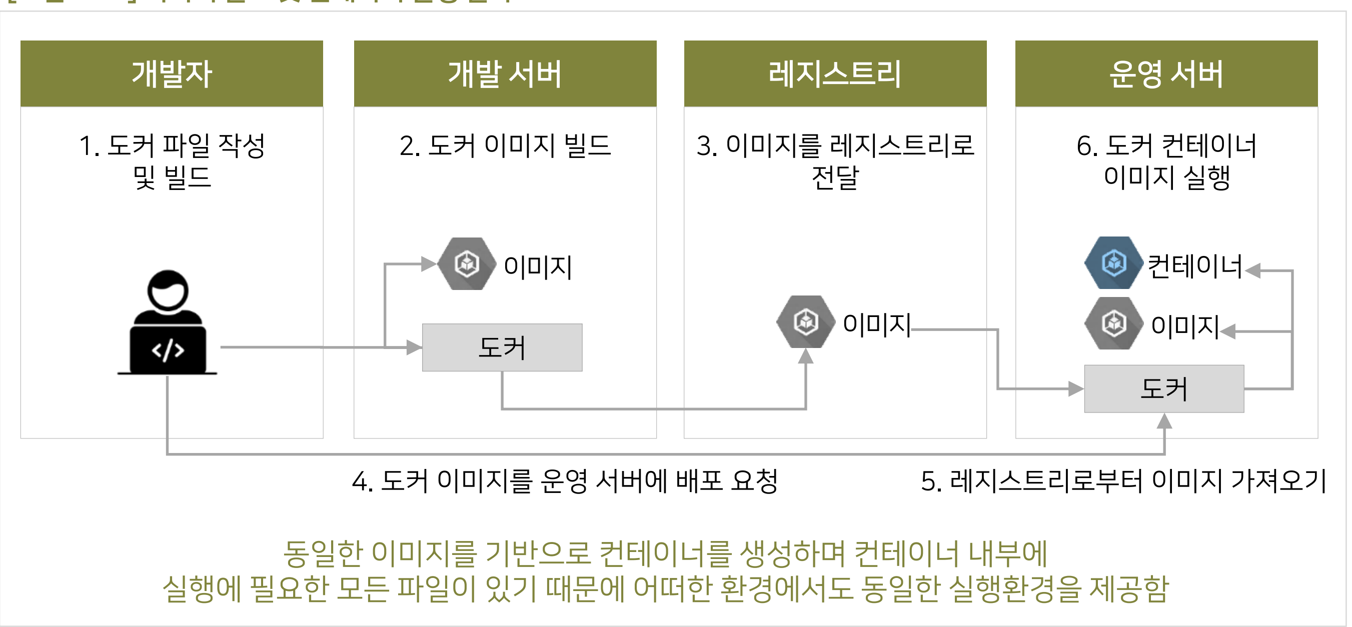
- **도커 클라이언트는 도커 데몬과 유닉스 소켓 또는 REST API를 사용하여 통신을 하며, 도커 데몬이 컨테이너를 구축, 실행 및 배포할 수 있도록 한다. 도커 클라이언트와 데몬은 동일한 시스템에서 실행될 수도 있고, 도커 클라이언트를 원격으로 도커 데몬에 연결하여 사용할 수도 있다.**

[그림 7-37] 도커 아키텍처



- **도커 파일을 생성하고, 빌드한 후 생성된 이미지를 레지스트리로 전달하고 각 운영 서버 등에서 이미지를 다운로드 받아서 실행하는 절차는 다음과 같다.**

[그림 7-38] 이미지 빌드 및 컨테이너 실행 절차



07 클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

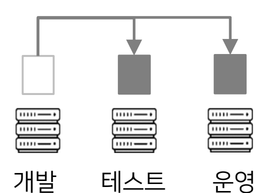
7.2.5.3 컨테이너 오케스트레이션

- **컨테이너 오케스트레이션이란 컨테이너의 배포, 관리, 확장, 네트워킹을 자동화하는 기술을 말한다.**
- **애플리케이션 운영을 위해 컨테이너를 배포하고 구성하는 것을 말하며 컨테이너 오케스트레이션 도구를 통해 수행한다.**

[그림 7-39] 컨테이너 오케스트레이션 주요 기능

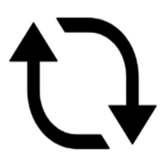
온디맨드 딜리버리
(On Demand Delivery)

필요한 컴퓨팅 자원
(컨테이너) 즉시 제공



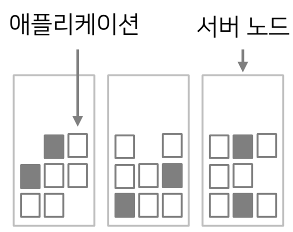
일관성 & 연속성
(Consistency & Continuous)

이미지 기반으로
구성, 배포



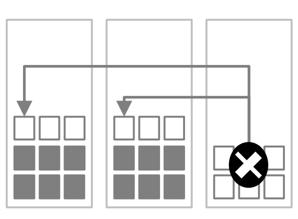
롤링 업데이트
(Rolling Update)

업그레이드 시
다운타임 최소화



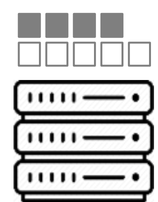
동적 스케줄링
(Dynamic Scheduling)

애플리케이션
동적 자동 배치



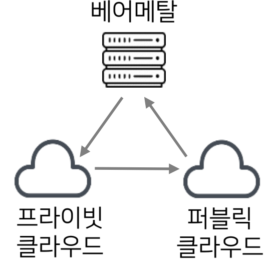
셀프 복구
(Self Recovery)

장애 발생 시,
정상 서버 노드로
자동 재배치



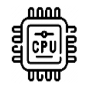
애플리케이션 스케일링
(Application Scaling)

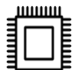
애플리케이션 워크로드
단위 오토스케일링



이식성
(Portable)

멀티/하이브리드
클라우드 기반
애플리케이션 운영





자원 사용 제어
(Resource Usage Control)

CPU, 메모리
최대 자원 지정

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.3 컨테이너 오케스트레이션

- **컨테이너 오케스트레이션 도구는 컨테이너와 마이크로서비스 아키텍처를 규모에 따라 관리할 프레임워크 를 제공한다. 컨테이너 라이프사이클 관리에 사용할 수 있는 컨테이너 오케스트레이션 도구는 다양하며, 그중 쿠버네티스, 도커 스웜, 아파치 메소스(Mesos)가 널리 사용된다.**

[표 7-6] 컨테이너 오케스트레이션 주요 도구

도구	설명
쿠버네티스 (Kubernetes)	• 구글에서 오픈소스로 공개한 컨테이너 오케스트레이션 플랫폼 • 도커와 별도의 솔루션으로 제공되었지만, CNCF 재단에 소스 이관 후 도커 엔진에 포함 • 베어메탈, VM, 퍼블릭 클라우드 등의 다양한 환경에서 적용됨 • 대규모 컨테이너 서비스 배포 및 관리에 장점 • 다양한 생태계가 구축되어 있음 → 대형 규모, 세밀하고 다양한 설정 기능이 필요한 경우에 적합
도커 스웜 (Docker Swarm)	• 도커에서 공식적으로 만든 오케스트레이션 도구 • 여러 개의 도커 호스트를 클러스터링하여 단일 가상 도커 호스트를 구성하는 단순한 구조에 효과적 • 호스트 OS에 에이전트만 설치하면 간단하게 작동하고 설정이 쉽고 에이전트를 외부에 설치하지 않음 • 도커 명령어 및 도커 컴포즈(Docker Compose)를 그대로 사용하는 쉬운 접근 용이성 → 중소형 규모, 관리할 서버가 적고, 많은 기능이 필요하지 않은 경우에 적합
아파치 메소스 (Apache Mesos)	• 대규모 클러스터 관리 플랫폼 • 리눅스 컨트롤 그룹을 사용하여 CPU, IO, 파일 시스템, 메모리 격리를 제공 • 분산된 시스템 커널 또는 프레임워크에 컴퓨터 자원을 공급하는 클러스터 플랫폼 • 하둡(Hadoop), 카프카(Kafka) 및 스파크(Spark)와 같은 다른 오픈 소스 서비스와 함께 응용프로그램을 배치해야 하는 환경에 용이 → 초대형, 수천 개의 확장, 호스트와 랙에 대한 관리 필요시 적합

클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.4 컨테이너 오케스트레이션 도구 - 쿠버네티스

- **쿠버네티스는 컨테이너 오케스트레이션 도구의 한 종류이며 엄청난 인기로 인해 사실상 표준으로 사용되고 있다. 컨테이너 오케스트레이션 도구는 컨테이너 호스트와 컨테이너의 증가에 따라 분산된 서버의 다수 컨테이너를 효율적으로 관리하기 위해 등장하게 되었다.**
- **쿠버네티스를 이용한 상용 컨테이너 플랫폼으로는 레드햇 오픈쉬프트 컨테이너 플랫폼(Redhat Openshift Container Platform), 피보탈 쿠버네티스 서비스(Pivotal Kubernetes Service)가 있다.**
- **쿠버네티스는 줄여서 K8s라고 부르기도 한다.**
- **쿠버네티스의 주요 기능은 상태 관리, 스케줄링, 클러스터, 서비스 디스커버리, 리소스 모니터링, 스케일링, 롤아웃/롤백 기능이 있다.**

[표 7-7] 쿠버네티스 주요 기능

구분	설명
자동화된 복구	• 상태를 선언하고 선언한 상태를 유지하고, 노드가 죽거나 컨테이너 응답이 없을 경우 자동으로 복구 • 쿠버네티스는 실패한 컨테이너를 다시 시작하고, 컨테이너를 교체하며, '사용자 정의 상태 검사'에 응답하지 않는 컨테이너를 죽이고, 서비스 준비가 끝날 때까지 그러한 과정을 클라이언트에 보여주지 않음
스케줄링	• 클러스터의 여러 노드 중 조건에 맞는 노드를 찾아 컨테이너를 배치
클러스터	• 가상 네트워크를 통해 하나의 서버에 있는 것처럼 통신 • 컨테이너화된 작업을 실행하는 데 사용할 수 있는 쿠버네티스 클러스터 노드 제공
서비스 디스커버리	• 서로 다른 서비스를 쉽게 찾고 통신 • DNS[1] 이름을 사용하거나 자체 IP 주소를 사용하여 컨테이너를 노출할 수 있음
스토리지 오케스트레이션	• 로컬 저장소, 공용 클라우드 공급자 등과 같이 원하는 저장소 시스템을 자동으로 탑재할 수 있음
스케일링	• 리소스에 따라 자동으로 서비스를 조정 • 컨테이너에 대한 트래픽이 많으면, 네트워크 트래픽을 로드밸런싱하고 배포하여 배포가 안정적으로 이루어질 수 있음
자동화된 롤아웃/롤백 (Rollout/Rollback)	• 쿠버네티스를 사용하여 배포된 컨테이너의 원하는 상태를 서술할 수 있으며 현재 상태를 원하는 상태로 설정한 속도에 따라 변경할 수 있음 • 예를 들어 쿠버네티스를 자동화해서 배포용 새 컨테이너를 만들고, 기존 컨테이너를 제거하고, 모든 리소스를 새 컨테이너에 적용할 수 있음

1) DNS(Domain Name System) : 도메인 이름과 IP 주소를 서로 변환하는 역할

클라우드 네이티브 정보시스템 분석 단계

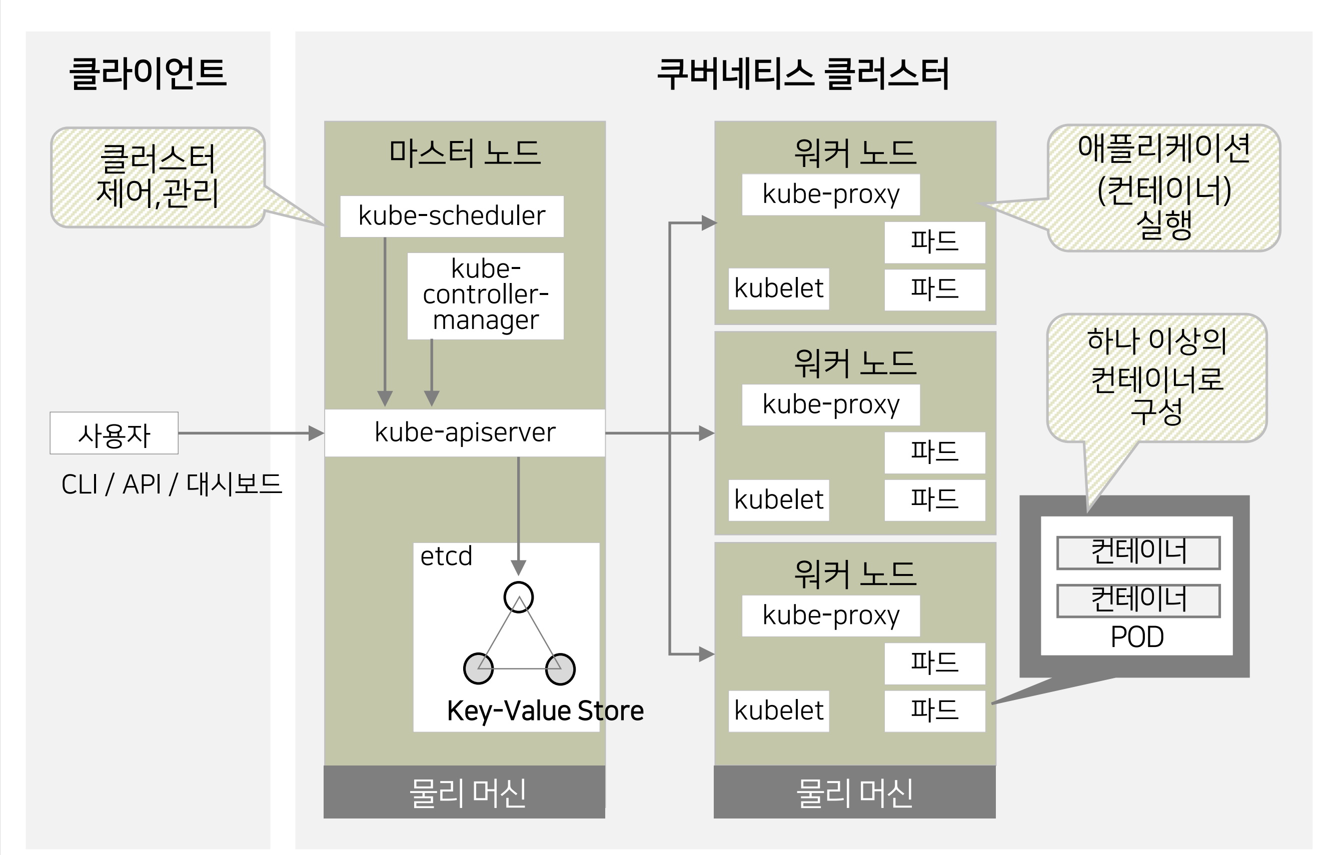
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.4 컨테이너 오케스트레이션 도구 - 쿠버네티스

- **쿠버네티스는 가장 상위 수준에서 관리 서버인 마스터 노드(Master Node)와 컨테이너가 실행되는 워커 노드(Worker Node)로 구성되며, 이를 쿠버네티스 클러스터라고 한다.**

[그림 7-40] 쿠버네티스 구성도



구성요소	설명
클러스터 (Cluster)	• 컨테이너를 실행하기 위한 노드 집합으로 물리 서버 위에 클러스터라는 가상의 컨테이너 실행 영역을 정의, 하나 이상의 마스터 노드와 워커 노드로 구성
노드 (Node)	• 물리서버 혹은 가상서버로 쿠버네티스 관리서버(마스터노드)나 실제 컨테이너가 배포되는 서버(워커 노드)로 할당됨
마스터 노드 (Master Node)	• 클러스터 관리 영역으로 클러스터의 상태 관리 및 모니터링을 수행
워커 노드 (Worker Node)	• 애플리케이션 실행 영역으로 컨테이너를 포함하는 파드가 배포, 실행되는 영역
파드(Pod)	• 단일한 노드에 배포되는 하나 이상의 컨테이너 집합으로 가장 작은 쿠버네티스 배포 단위

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.5 런타임 플랫폼

7.2.5.4 컨테이너 오케스트레이션 도구 - 쿠버네티스

- **쿠버네티스는 컨테이너 오케스트레이션 도구로서의 역할뿐만 아니라 개발환경, DevOps, 실행환경, 서비스 환경, 보안 등의 기능을 가진다.**

[표 7-8] 참고. 쿠버네티스의 상세 기능

구분	기능	설명
개발환경	배포형상 생성	• 컨테이너 이미지 생성(개발자 업무) 및 레지스트리 저장
	사용자 도구	• kubectl CLI, 쿠버네티스 대시보드(Kubernetes Dashboard)
DevOps	Log Streaming	• 컨테이너 로그 중앙집중 저장(Cluster-Level-Logging)
실행환경	지원 실행환경 (컨테이너런타임)	• 도커, RunC, RKT, OCI Runtime-Spec
	애플리케이션 배포	• 컨테이너 이미지 배포
	애플리케이션 관리	• 애플리케이션 자원 설정 및 변경, 애플리케이션 스케일링 • 애플리케이션 삭제, 업데이트
	App 서비스 관리	• 서비스 바인딩 정보 이용 백엔드 서비스 연동
	컨테이너 관리	• 컨테이너 헬스체크, 오케스트레이션
	워크로드 관리	• 파드(애플리케이션 인스턴스) 관리, 파드-IP 매핑, 파드-스토리지 매핑 • 복합 컨테이너 파드 생성 및 자원 연결, 관리 • 파드 복제 제어(리플리카 수의 보장), 볼륨(스토리지) 라이프사이클 관리
	(마이크로) 서비스 관리	• 복수 파드의 논리적 집합, 단위 마이크로서비스 또는 벡엔드서비스 관리 • 서비스 엔드포인트(TCP 또는 UDP) 관리, 멀티 포트 지원, 백엔드 서비스 매핑 • 가상 IP & 서비스 프록시 관리, 환경변수 기반 또는 DNS 기반 서비스 디스커버리
운영환경	운영 도구	• kubectl CLI, 쿠버네티스 대시보드
	클러스터 관리	• 작업 스케줄링(파드 할당), 노드 시작 및 셀프 등록 • 네임스페이스(가상 클러스터) 관리, 자원 쿼터 할당, 오브젝트 메타데이터 관리 • 오브젝트 라벨링 & 선택, 멀티 클러스터 관리
	CCM (Cloud Controller Manager)	• 노드 정보 추출 및 초기화, Health Checking • 다른 노드 내 컨테이너 간 통신을 위한 라우터 설정(GCP, Google Cloud Platform) • 서비스 백엔드를 위한 로드밸런서 설정, 퍼시스턴트 볼륨 라벨(Persistent Volume Label) 제어
	모니터링	• 리소스 사용량 모니터링(메트릭 기반), 오브젝트 상태 모니터링 및 자동복구
서비스환경	서비스 카탈로그	• 서비스 브로커 등록, 서비스 브로커 계획 정보 생성, 서비스 인스턴스 프로비저닝 • 서비스 바인딩, 서비스 바인딩 정보 애플리케이션 매핑
	지원 서비스 카탈로그	• 헬름 챠트(Helm Chart) 기반 클러스터 운영자에 의한 결정
보안	**API 접근 제어**	• API 인증 및 역할 기반 인가
	실행환경 접근 제어	• 정책 기반 자원 사용 제어, Pod 간 네트워크 접근 제어 • 컨테이너 실행 시 호스트, 클라우드 메타데이터 권한 제어 • 노드 위 파드 배치 제어(워크로드 분산 효과)
	보안 통신	• 클러스터 내 인증서 기반 보안 통신(TLS)

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.6 CI/CD

7.2.6.1 CI/CD 파이프라인

- CI/CD는 반복적인 개발, 테스트, 배포 과정에 대한 자동화와 모니터링을 제공하는 도구 기반 프로세스로 애플리케이션 개발 단계를 자동화하여 애플리케이션을 보다 짧은 주기로 제공하는 것을 목적으로 한다.
- CI(Continuous Integration)는 애플리케이션에 대한 새로운 코드 변경 사항이 정기적으로 빌드 및 테스트되어 공유 리파지토리에 지속적으로 통합(Continuous Integration)하는 것이고 CD(Continuous Delivery)는 지속적인 서비스 제공(Continuous Delivery) 및/또는 배포(Deployment)를 의미한다.
- CI/CD는 애플리케이션의 개발, 테스트 및 배포에 이르는 라이프 사이클에 걸쳐 자동화를 제공하여 CI/CD 파이프라인으로 불리고 있다.

[그림 7-41] CI/CD 파이프라인

1 코드&커밋
2 빌드&설정
3 스캔&테스트
4 릴리즈
5 배포

개발 → CI (Continuous Integration) → CD (Continuous Delivery/Deploy) 운영

단계	① 코드&커밋 (Code & Commit)	② 빌드&설정 (Build & Config)	③ 스캔&테스트 (Scan & Test)	④ 릴리즈 (Release)	⑤ 배포 (Deploy)
내용	개발자가 소스 저장소에 소스를 통합, 저장	소스 커밋이 발생하면 소스 저장소에서 가져와서 빌드	정적 분석 및 Junit 도구 등을 이용한 단위 테스트 수행	빌드된 소스를 릴리즈 버전으로 패키징	패키징된 릴리즈 버전을 개발환경, 운영환경으로 배포
주요 도구	git VISUALSVN SERVER	Apache maven gradle	JUnit	Nexus	Jenkins

클라우드 네이티브 정보시스템 분석 단계

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.6 CI/CD

7.2.6.2 클라우드 환경에서의 CI/CD 파이프라인

- **클라우드 환경에서는 수시로 서비스의 배포가 발생하며 변경되는 부분만을 이미지로 생성, 레지스트리에 저장 가능하며 최종적으로 기본 이미지와 추가되는 이미지를 합쳐서 쿠버네티스를 통해 컨테이너를 생성, 배포한다.**
- **클라우드 환경에서의 CI/CD가 기존 레거시 환경에 비교해 가장 큰 차이점이 바로 애플리케이션을 컨테이너 단위로 배포한다는 점이다.**
- **기존에는 빌드된 애플리케이션을 직접 배포했지만 클라우드 환경에서는 애플리케이션 및 애플리케이션의 실행에 필요한 모든 SW를 컨테이너라는 실행환경에 묶어서 배포하게 된다.**

[그림 7-42] 클라우드 환경의 CI/CD 파이프라인

지속적 통합(CI)
지속적 배포(CD)
</>
개발자
커밋
형상관리
빌드
컨테이너 이미지 저장
애플리케이션
배포
컨테이너 실행
애플리케이션
이미지 레이어 구조
A 기본 이미지
애플리케이션 A
실행파일/라이브러리
애플리케이션 A 변경부분
변경부분
A 변경 이미지
애플리케이션 A
변경부분
실행파일/라이브러리

구분	활동(태스크)	설명	도구
지속적 통합 (CI)	• 소스코드 통합	• 소스코드 커밋	• 형상관리 • 깃, SVN
	• 빌드 , 컨테이너화	• 소스코드 빌드 • 도커 이미지 생성	• 빌드 도구 • 도커, 메이븐
	• 컨테이너 레지스트리 등록	• 레지스트리 푸쉬(Push)	• 컨테이너 레지스트리 서비스 • 도커허브, ECR, ACR, GCR
지속적 배포 (CD)	• 배포	• 배포 수행	• 배포 및 자동화 도구 • 젠킨스, AWS Code Pipeline, Azure DevOps Pipeline, GCP Code Build
	• 파이프라인 자동화	• 일부 과정 혹은 전체 과정 자동화	

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.6 CI/CD

7.2.6.2 클라우드 환경에서의 CI/CD 파이프라인

- **컨테이너 기반 클라우드 플랫폼을 활용함으로써 클라우드 환경에서의 CI/CD 파이프라인 체계를 확보할 수 있다. 프라이빗 클라우드를 구축할 경우에는 자체적으로 CI/CD 파이프라인 체계를 구현해야 한다.**
- **다음은 각종 클라우드 플랫폼에서 가장 많이 지원하는 CI/CD 파이프라인 도구에 대한 설명이다.**

[표7-9] 주요 클라우드 플랫폼의 CI/CD 파이프라인 도구

구분	솔루션	설명
형상 관리	git 깃(Git)	• 협업 프로젝트의 소스 관리 및 분산된 형상(버전) 관리 시스템 • 주요 특징 - 보안 및 안정적인 소스 저장공간 제공 - 로컬/원격 저장소 지원(로컬 저장소에 커밋하므로 처리 속도가 빠름) - 소스 변경사항에 대한 스냅샷 저장 - 체크섬을 통한 파일 관리 - 저장소 내용은 데이터를 추가하기만 가능하고 되돌리거나 삭제 불가 - 데이터 무결성, 분산, 비선형 워크플로우 지원
	VISUALSVN SERVER SVN(Subversion)	• 중앙 집중형 버전 관리 시스템 • 주요 특징 - 자동 쓰기를 지원하므로 쓰기 도중 중단으로 인한 저장소 내의 불일치나 손상을 피할 수 있음 - 파일 이름 변경, 이동, 디렉토리 버전 관리 지원 - 개발 버전과 배포 버전을 섞이지 않고 쉽게 관리 - 커밋 시에 자동으로 로그 기록 가능 - 서버-클라이언트 양방향 데이터 전송으로 네트워크 소통량을 최소화
빌드 도구	Apache maven Maven	• pom.xml을 이용한 정형화된 빌드 시스템 및 플러그인을 구동해 주는 프레임워크 • 주요 특징 - 프로젝트 산출물들의 쉬운 검색과 필터링 - 의존성 관리 및 원격 저장소 지원 - 여러 목표를 묶어서 라이프사이클 단계들을 만들고 실행
	gradle Gradle	• Groovy를 기반으로 한 오픈소스 빌드 도구 • 주요 특징 - 멀티 프로젝트 지원 - Apache Ivy에 기반한 강력한 의존성 관리 - 원격 저장소나 pom, ivy 파일 없이 연결되는 의존성 관리 지원

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.6 CI/CD

7.2.6.2 클라우드 환경에서의 CI/CD 파이프라인

[표7-9] 주요 클라우드 플랫폼의 CI/CD 파이프라인 도구

구분	솔루션	설명
테스트 자동화 도구	JUnit Junit	• 자바를 위한 단위 테스트 프레임워크(이클립스 V3.2 이후는 기본 내장) • 주요 특징 - 구현 단계에서 각 모듈이 완성되었을 경우 개별적인 컴포넌트를 테스트 - 자동화된 테스트 기법 제공 - 단위 모듈별 테스트를 통해 소스 코드의 품질 보장 - 단위 테스트를 통해 통합 테스트 시 회귀결함 감소 - 테스트 결과에 대한 시각화 제공
릴리즈 도구	Nexus Nexus	• 메이븐에서 사용할 수 있는 가장 널리 사용되는 무료 저장소 • 오픈 소스 버전은 메이븐을 사용하는 경우에는 거의 필수적으로 사용 • 주요 특징 - 다수의 중앙집중식 리파지토리 관리 - 프록시 개념을 통해 개발자들에게 더욱 쉬운 리파지토리 연동 편의성 제공 - 스냅샷 : 같은 버전으로 여러 번 배포 가능하고, 수시로 릴리즈되는 바이너리를 저장하는 저장소 - 릴리즈 : 같은 버전으로 한 번밖에 배포할 수 없으며, 정식 릴리즈를 통해 배포되는 바이너리를 저장하는 저장소
배포 도구	Jenkins Jenkins	• 소프트웨어 개발 시 지속적으로 통합 서비스를 제공하는 CI 도구 • 주요 특징 - 자동화된 테스트 및 빌드 - 각종 배치 작업의 간략화 - 코드 표준 준수 여부 검사 - 도메인 – 서비스 – UI로 이어지는 빌드 파이프라인 구성 - 빠른 개발 속도 지원

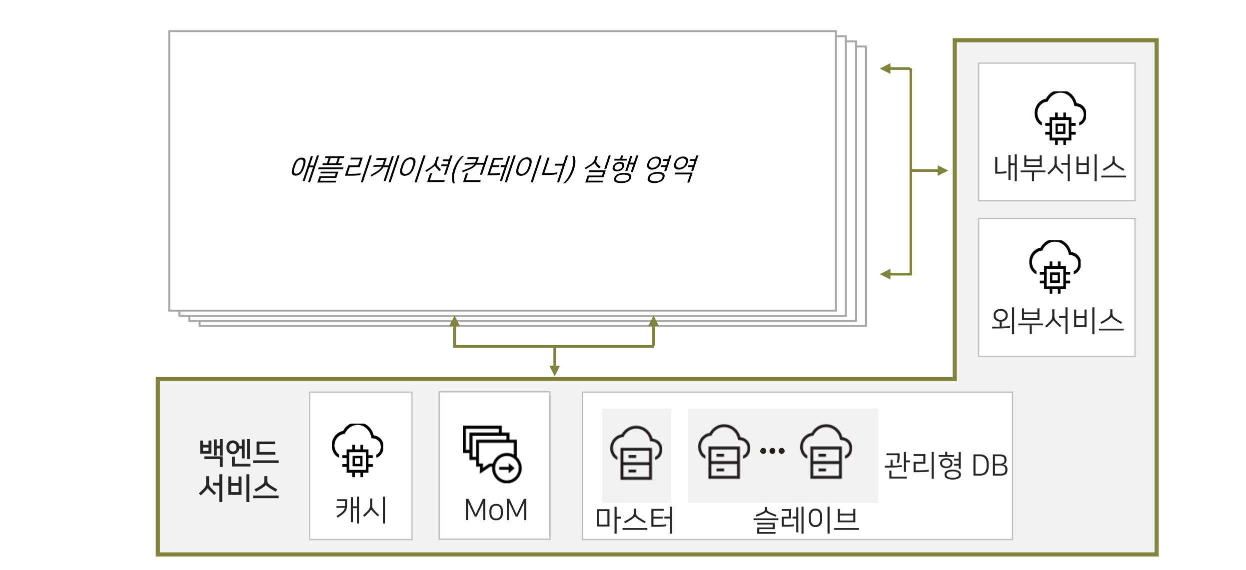
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.7 백엔드 서비스

7.2.7.1 개념 및 종류

- 클라우드 네이티브 애플리케이션을 실행하기 위해 네트워크로 연결된 모든 리소스를 백엔드(Backing) 서비스라고 한다.
- 클라우드 플랫폼에서 제공하는 서비스, 외부 연계 서비스 및 직접 구축한 서비스를 모두 포함하며 다른 백엔드 서비스로 전환이 가능하도록 구성되어야 한다. 이때 클라우드 네이티브 애플리케이션을 수정하지 않아야 한다.
- 주요 백엔드 서비스는 DB, 캐시, MoM, 외부 서비스, 내부 서비스 등이 있다.

[그림 7-43] 백엔드 서비스 구조 및 종류



구분	설명
DB	• 클라우드 플랫폼에서 제공 및 관리되는 DB 서비스
캐시(Cache)	• 세션관리, 성능향상 등의 용도를 위한 메모리 캐싱 서비스
MoM	• 비동기 메시징 서비스 ※ MoM(Message Oriented Middleware)
외부서비스	• 인증, 간편결제, 오픈 API 형태 외부 연계 서비스
내부서비스	• AI, 빅데이터, 챗봇 등의 클라우드 플랫폼 제공 서비스

클라우드 네이티브 정보시스템 분석 단계

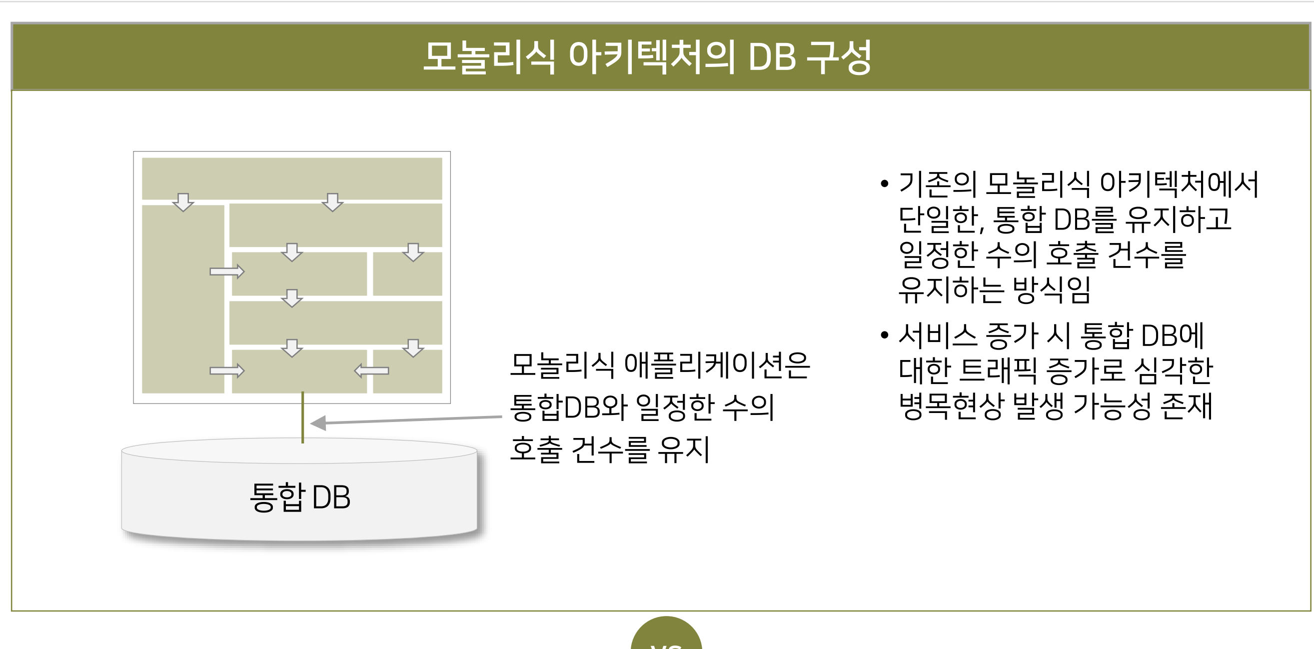
7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

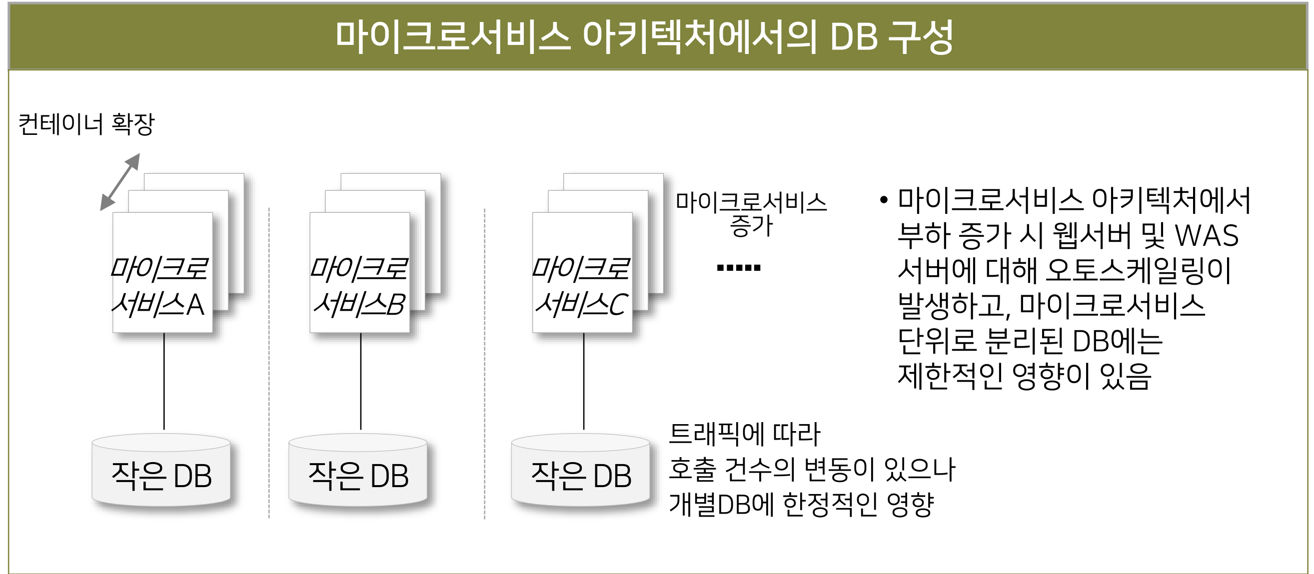
7.2.7 백엔드 서비스

7.2.7.2 클라우드 네이티브 환경의 DB 구성

- **모놀리식 아키텍처는 서비스 증가 시 트래픽 증가에 의한 심각한 병목현상이 발생하지만 클라우드 네이티브 환경의 마이크로서비스 아키텍처에서는 부하 증가 시 웹서버 및 WAS서버에 대해 오토스케일링이 발생하고, 마이크로서비스 단위로 분리된 DB에 대한 영향을 최소화하는 구조이다.**

[그림 7-44] 모놀리식 아키텍처에서의 DB 서비스





7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.7 백엔드 서비스

7.2.7.2 클라우드 네이티브 환경의 DB 구성

- **마이크로서비스 아키텍처에서는 안정적인 DB 성능을 확보하기 위해 메모리 캐싱, 비동기 활동, Read-Write 분리, 마이크로서비스 조합 패턴 구현 등의 방안을 적용할 수 있다.**

[그림 7-45] 마이크로서비스 아키텍처에서의 성능 확보 방안

WEB서버
서비스 ••• 서비스 서비스 ••• 서비스
오토스케일링
4
WAS서버
서비스 ••• 서비스 서비스 ••• 서비스
DB & 백엔드 서비스
1 캐시
2 MoM
3 마스터 ••• 슬레이브
클라우드 DB

구분	설명
1 메모리 캐싱 활용	• 자주 읽는 데이터는 DB에서 데이터를 읽는 대신 메모리상에 데이터 영역을 구성하여 활용 예) 세션정보, 상품 목록 등
2 비동기 활용	• 실시간으로 결과를 응답하는 경우를 제외하고 메시지 기반으로 요청과 응답 처리를 분리 예) SAGA 패턴
3 읽기-쓰기 분리	• 단순히 데이터를 읽기만 하는 요청을 위해 Read 서버를 구성하여 요청을 분산 • 일부 클라우드 플랫폼에서 제공하는 관리형 DB의 경우 읽기용 서버에 한해 일정 개수까지 스케일아웃을 지원
4 마이크로서비스 조합 패턴 구현	• 마이크로서비스 설계 시 데이터를 조합하는 마이크로서비스를 별도로 분리하여 구현

7.2 클라우드 네이티브 애플리케이션 아키텍처 설계

7.2.8 텔레메트리

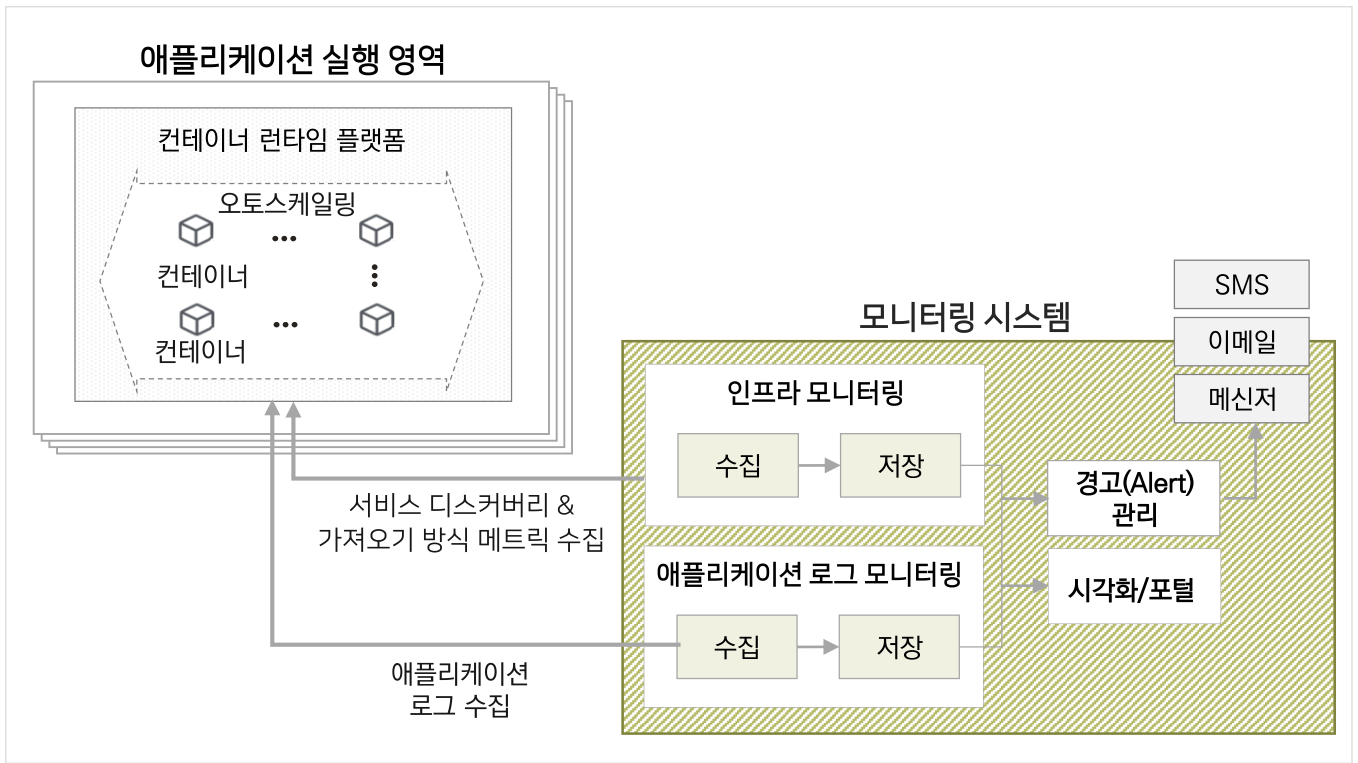
7.2.8.1 모니터링 정의 및 구성

- 텔레메트리 서비스는 클라우드 네이티브 애플리케이션 및 인프라 리소스(CPU, 메모리 등)에 대한 로그 및 사용 데이터를 수집, 분석, 시각화 등 애플리케이션 전반의 모니터링 기능을 제공한다.
- 클라우드 네이티브 애플리케이션은 마이크서비스 단위로 작아지고 OS 수준에서 가상화된 환경에서 모니터링 대상 컨테이너 또한 동적으로 생성되므로 기존 모니터링과 다르게 서비스 디스커버리 및 가져오기(Pull) 방식의 데이터 수집이 필요하다.

[표 7-10] 모니터링의 종류

구분	설명
인프라 모니터링	• 컨테이너 클러스터, 노드, 컨테이너 등 상태, 리소스 가용량 및 사용량 등을 수집, 제공
애플리케이션 모니터링	• 애플리케이션 헬스체크, 에러, 재시작 횟수 등의 이벤트 정보

[그림 7-46] 모니터링 구성



클라우드 네이티브 정보시스템 구축을 위한

개발자 안내서

08

클라우드 네이티브 정보시스템 설계 단계

8.1 도메인 모델링을 통한 마이크로서비스 설계
8.2 마이크로서비스 아키텍처 설계
8.3 12 Factors 기반 개발 원칙

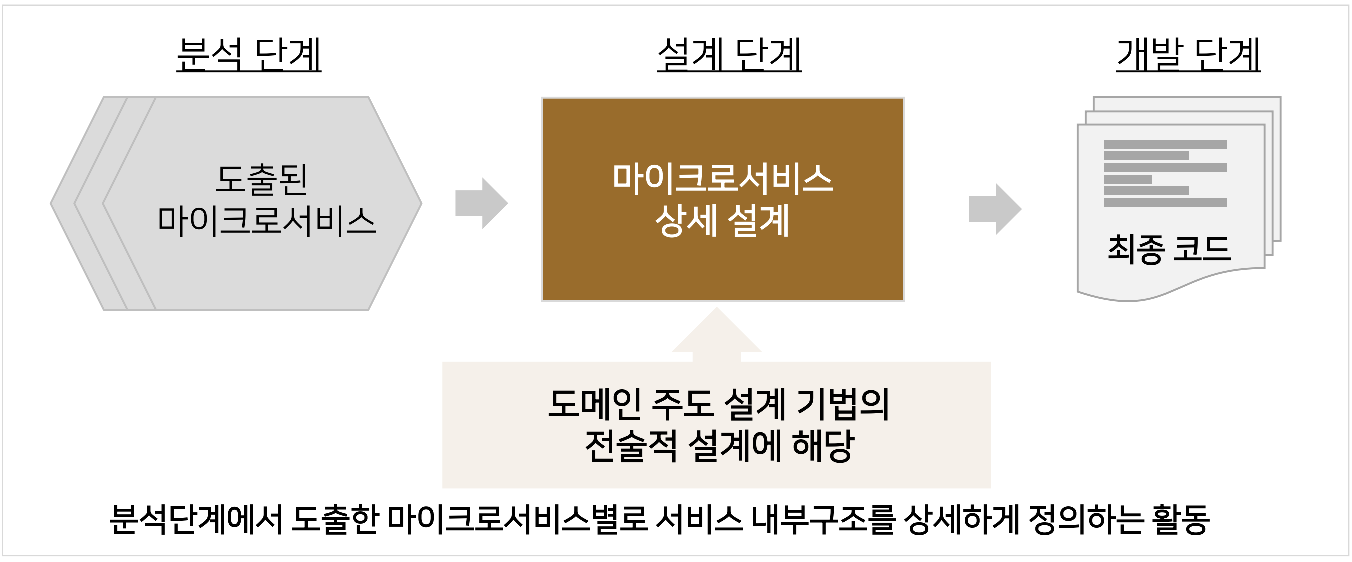
08 클라우드 네이티브 정보시스템 설계 단계

8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.1 개요

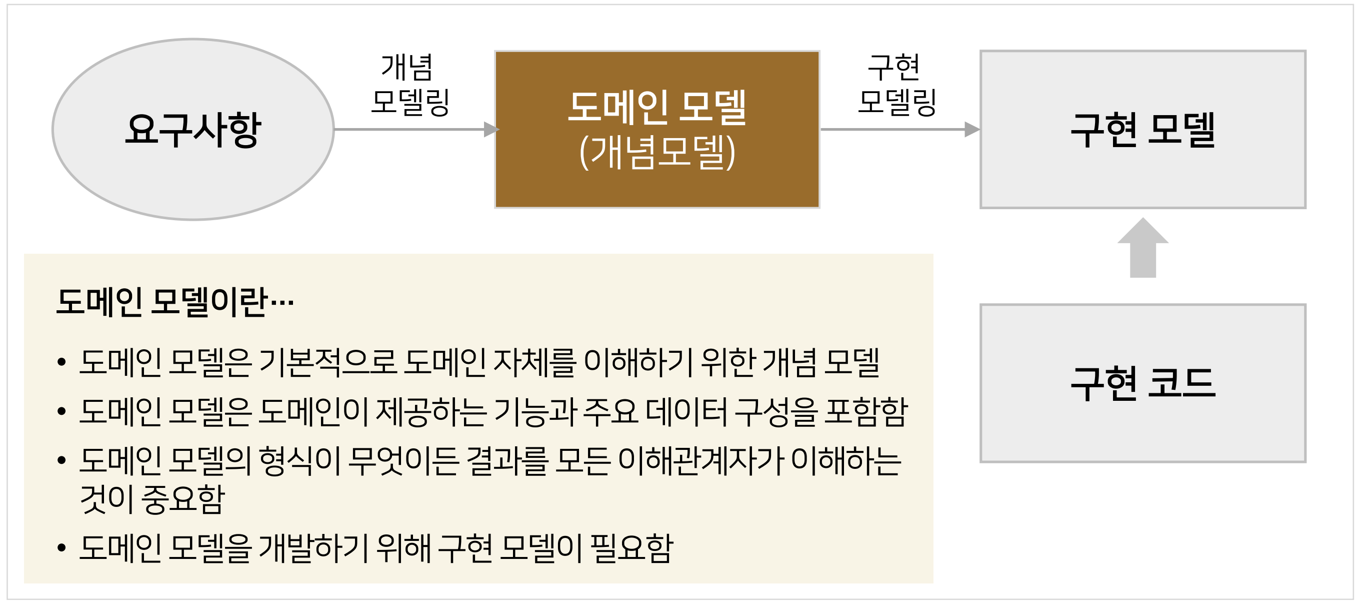
- 분석 단계에서 마이크로서비스를 도출하고, 서비스 간 연관관계를 식별하고, 마이크로서비스별 스펙이 정리되면 설계 단계에서는 마이크로서비스 상세 설계를 수행한다. 마이크로서비스 상세 설계는 마이크로서비스의 내부구조를 상세 정의하는 것이며, 이를 기반으로 최종 코드를 구현할 수 있다.

[그림 8-1] 마이크로서비스 상세 설계 개념



- 마이크로서비스 상세 설계는 도메인 주도 설계 방법론의 전술적 설계, 즉 도메인 모델링을 통해 이뤄진다.

[그림 8-2] 참고. 도메인 모델의 개념



08 클라우드 네이티브 정보시스템 설계 단계

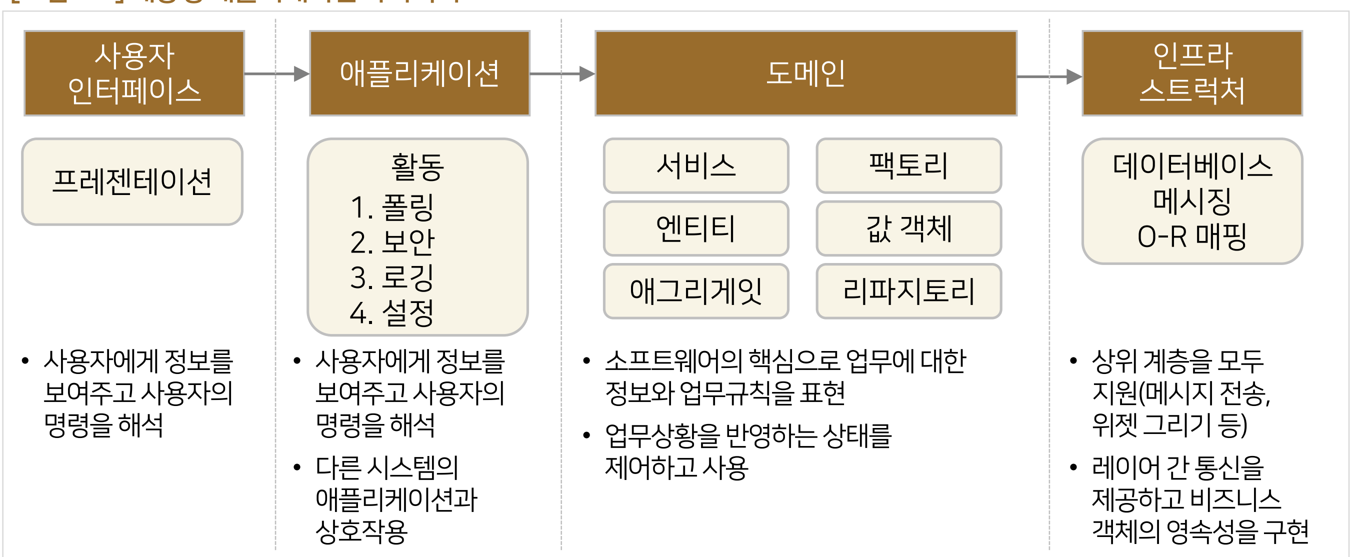
8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.2 도메인 모델의 패턴

8.1.2.1 계층형 아키텍처와 도메인 모델 표준 패턴

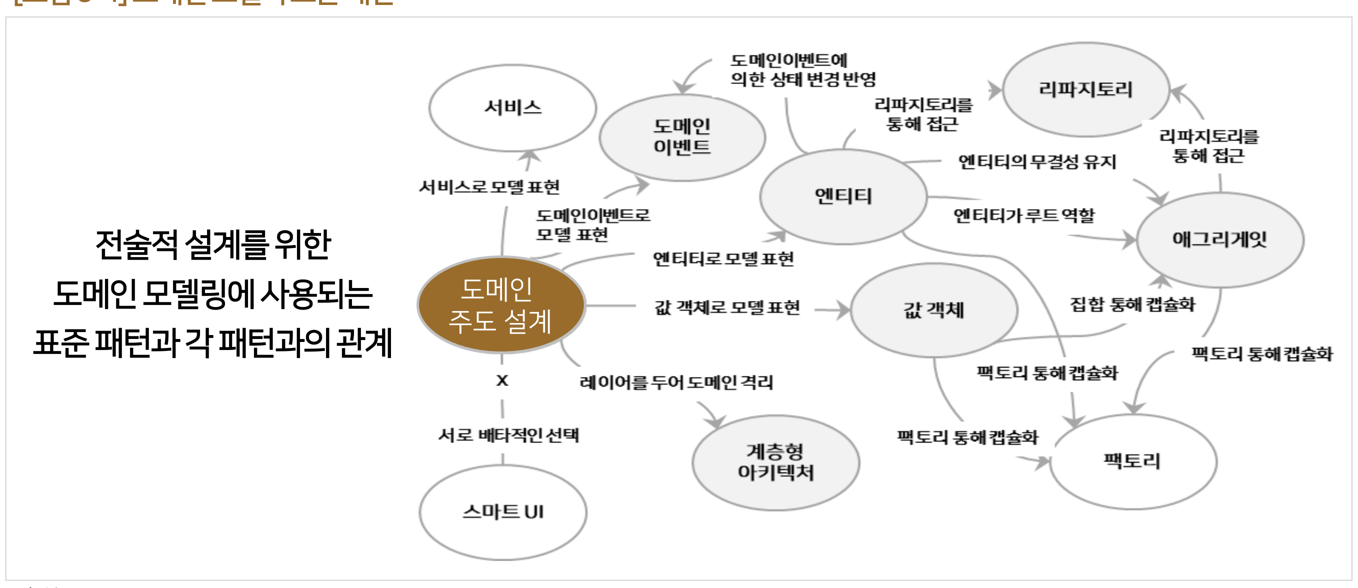
- **애플리케이션 아키텍처는 사용자 인터페이스(UI), 애플리케이션, 도메인, 인프라스트럭처의 4개 계층으로 구성된다.**

[그림 8-3] 계층형 애플리케이션 아키텍처



- **도메인 모델은 계층형 애플리케이션 아키텍처 상의 각 계층을 객체 지향적인 기법으로 구현하는 패턴을 의미한다. 도메인 모델의 표준 패턴을 이용하여 도메인 모델링을 할 경우, 다른 팀과 다른 사람의 업무를 쉽게 이해할 수 있는 설계를 할 수 있다.**

[그림 8-4] 도메인 모델의 표준 패턴



[출처 : Domain Driven Design, Eric Evans]

클라우드 네이티브 정보시스템 설계 단계

8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.2 도메인 모델의 패턴

8.1.2.2 엔티티(Entity) 및 값 객체 (Value Object)

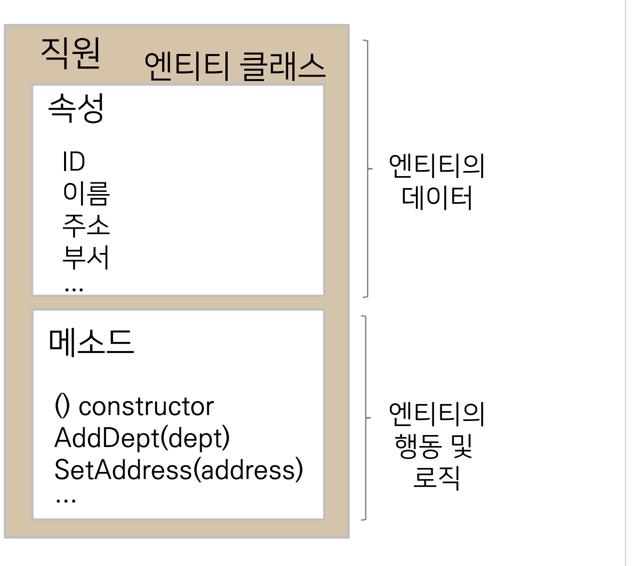
- **엔티티는 시간이 지나도 지속되는 고유한 식별자(ID)를 가지는 객체이다.**
- **엔티티 도메인 패턴은 엔티티의 데이터, 행동 및 로직으로 구성된다.**

[그림 8-5] 엔티티

엔티티

- 고유의 식별자를 가짐
- 식별자는 유일하고 생애 동안 불변
- 모델과 관련된 비즈니스 로직을 수행
- 자신만의 라이프사이클을 가짐

※ 기존 데이터모델의 테이블 개념
- 예) 고객, 상품, 주문 등



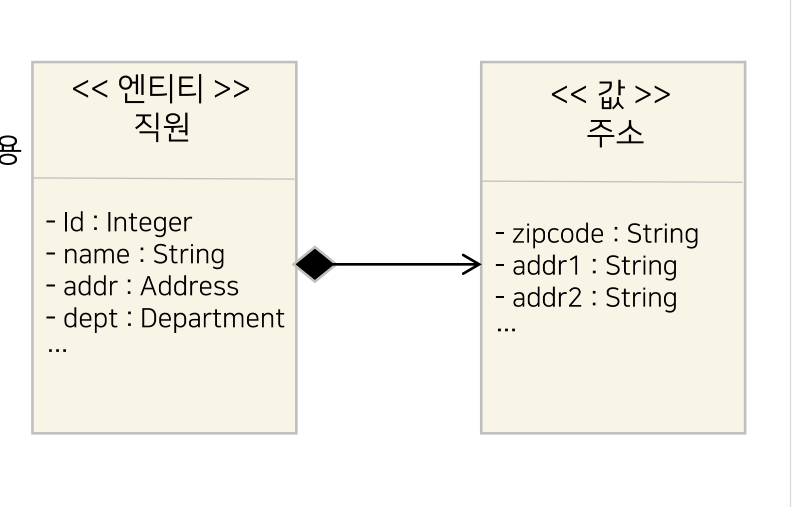
- **값 객체(Value Object)는 개념적으로 식별이 필요 없고, 단순히 값만을 가지고 있는 객체이다.**

[그림 8-6] 도메인 모델의 표준 패턴

값 객체

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

※ 기존 데이터모델의 컬럼 개념
- 예) 전화번호, 주소, 금액 등



8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.2 도메인 모델의 패턴

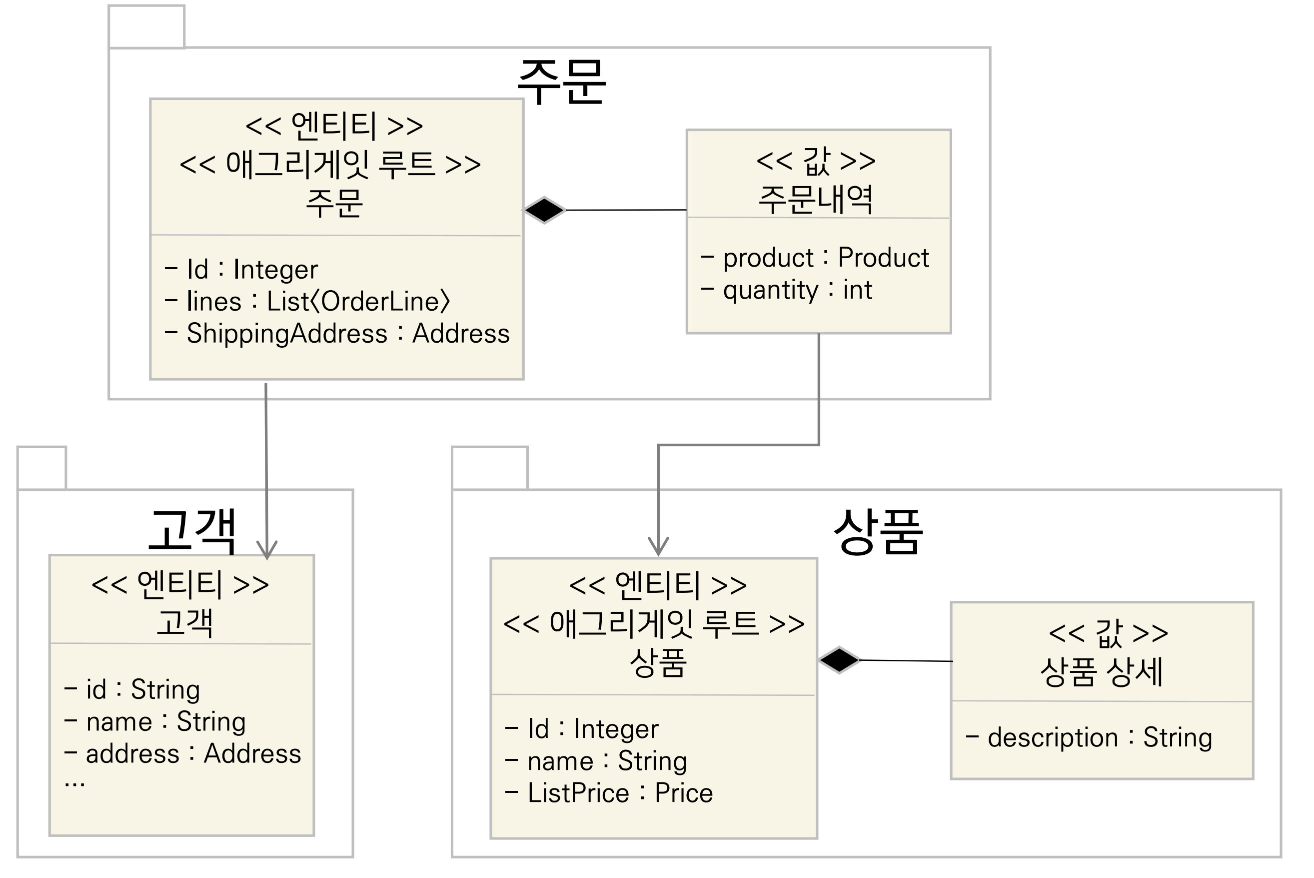
8.1.2.3 애그리게잇 (Aggregate)

- **애그리게잇은 도메인을 구성하는 엔티티와 값 객체의 묶음이다. 복잡한 도메인을 이해하고 관리하기 쉬운 단어로 만들려면 상위 수준에서 모델을 조망할 수 있다.**

[그림 8-7] 애그리게잇

애그리게잇

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



클라우드 네이티브 정보시스템 설계 단계

8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.2 도메인 모델의 패턴

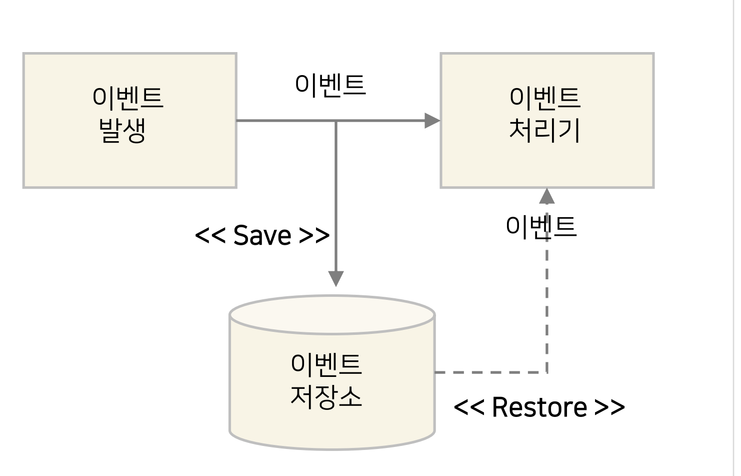
8.1.2.4 저장소(Repository) 및 도메인 이벤트 (Domain Event)

- **저장소는 생성된 애그리게잇에 대한 영속성을 관리하며, 애그리게잇에 대한 엔티티 저장, 애그리게잇 업데이트 및 삭제 작업이 이뤄진다.**

[그림 8-8] 저장소

저장소

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

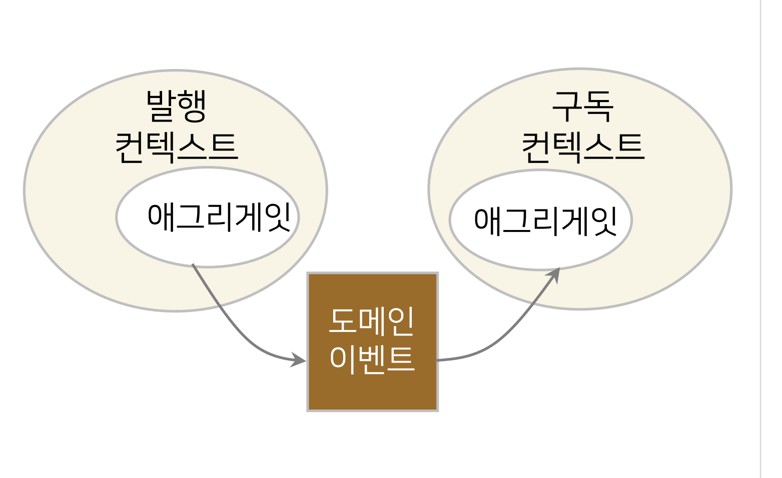


- **도메인 이벤트는 동일한 도메인의 다른 부분을 인식하고자 도메인에서 발생한 이벤트이다. 이벤트가 발생한다는 것은 과거에 발생된 일에 의해 상태가 변경됨을 의미한다.**

[그림 8-9] 도메인 이벤트

도메인 이벤트

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



08 클라우드 네이티브 정보시스템 설계 단계

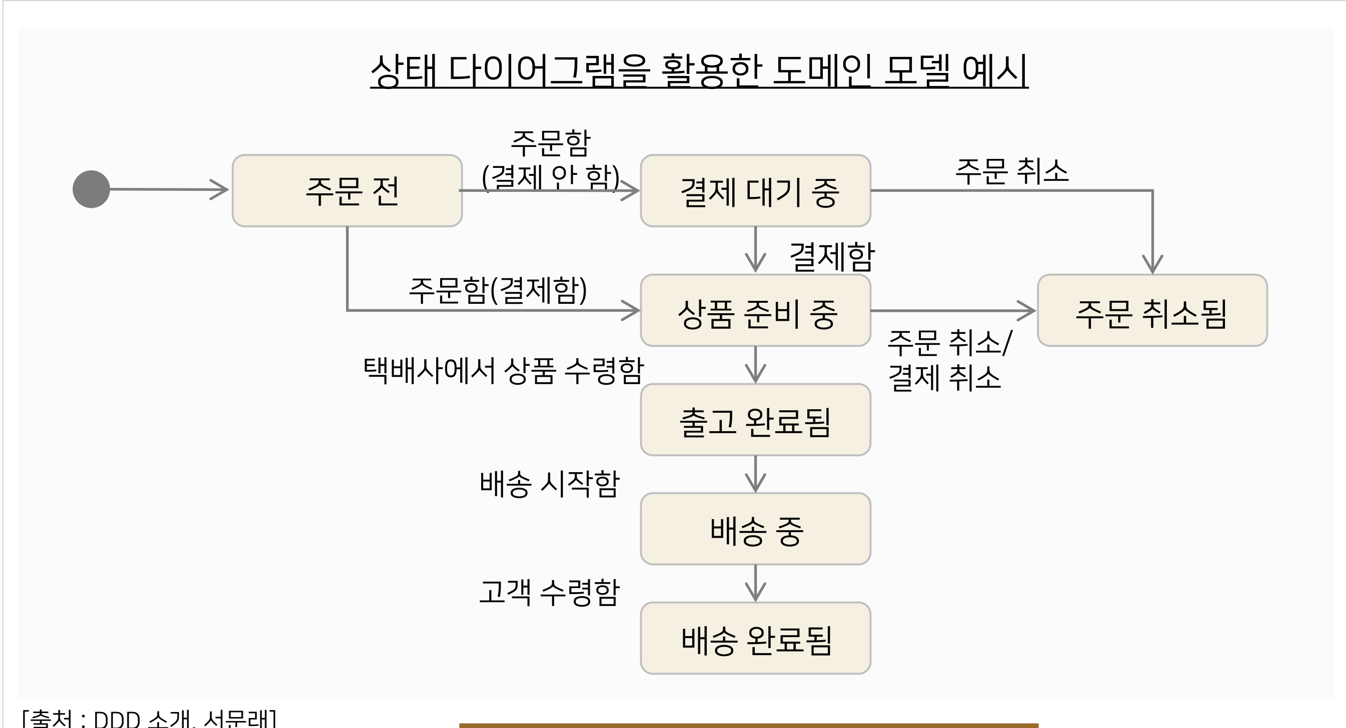
8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.2 도메인 모델의 패턴

8.1.2.5 도메인 모델의 코드화

- **다양한 형식으로 도메인 모델을 작성할 수 있으며, 아래는 상태 다이어그램을 활용하여 도메인 모델링을 한 것이다. 도메인 모델은 구현과정을 통해 프로세스와 업무규칙이 코드의 형태로 개발된다.**

[그림 8-10] 도메인 모델의 코드화 예시



[출처 : DDD 소개, 서문래]

도메인 모델의 코드화

"출고 전 배송지 변경 가능"을 구현한 코드

```
public class Order {
        private OrderState state;
        private ShippingInfo shippingInfo;

        public void changeShippingInfo(ShippingInfo newShippingInfo) {
                if (!state.isShippingChangeable()) {
                        throw new IllegalStateException("can't change shipping
                }
                this.shippingInfo = newShippingInfo;
        }

        public void changeShipped() {
                // 로직 검사
                this.state = OrderState.SHIPPED;
        }
        ...
}
```

"주문 취소는 배송 전에만 가능"을 구현한 코드

```
public enum OrderState {
        PAYMENT_WAITING {
                public boolean isShippingChangeable() {
                        return true;
                }
        },
        PREPARING {
                public boolean isShippingChangeable() {
                        return true;
                }
        },
        SHIPPED, DELIVERING, DELIVERY_COMPLETED;

        public boolean isShippingChangeable() {
                return false;
        }
}
```

[출처 : Incheol's TECH BLOG]

클라우드 네이티브 정보시스템 설계 단계

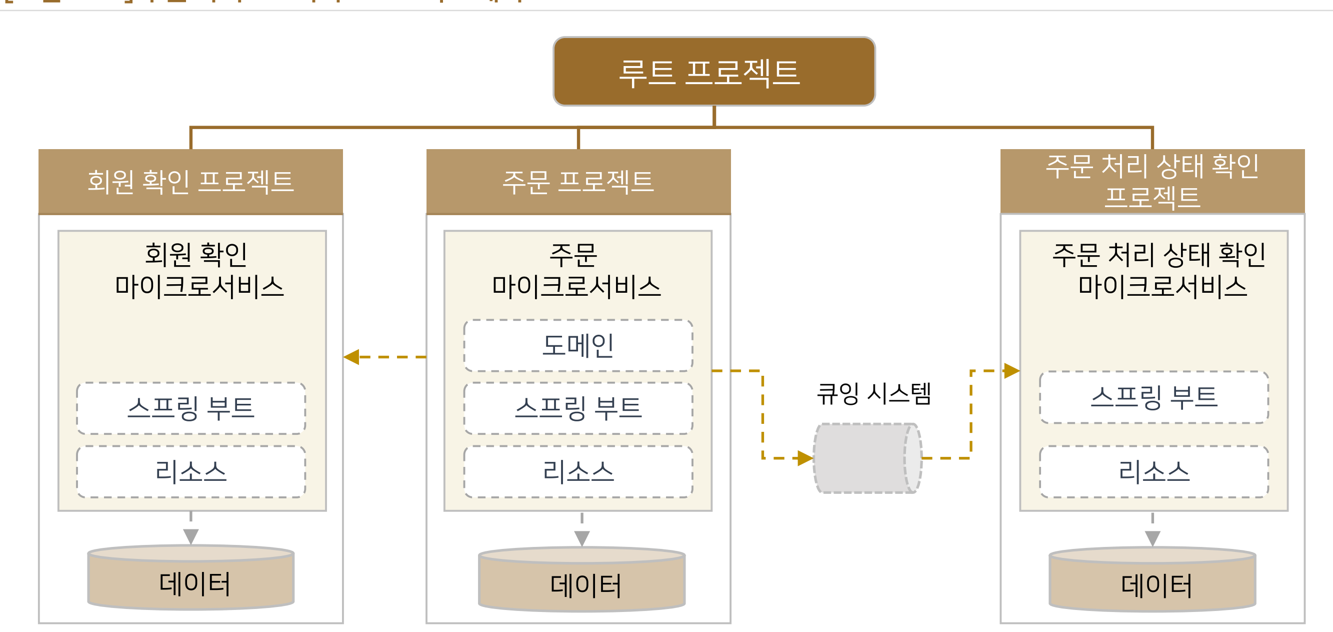
8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.3 마이크로서비스 프로젝트 설계

8.1.3.1 마이크로서비스 프로젝트 개념

- **마이크로서비스 프로젝트란 자바 기반으로 마이크로서비스를 만들 때 하나의 마이크로서비스를 관리하기 위한 '소스의 묶음' 단위이다.**
- **하나의 마이크로서비스 프로젝트는 여러 개의 마이크로서비스로 구성된다.**

[그림 8-11] 주문 마이크로서비스 프로젝트 예시



프로젝트 구분	프로젝트 명	프로젝트 설명
루트	msa-book	• 마이크로서비스 프로젝트 관리
마이크로서비스	msa-service-member	• 회원확인 프로젝트
	msa-service-order	• 주문 서비스 프로젝트
	msa-service-status	• 주문 처리상태 확인 서비스 프로젝트

- 루트 프로젝트는 전체 프로젝트를 위한 것이고, '회원 확인', '주문', 주문 처리 상태 확인' 프로젝트는 마이크로서비스임
- 루트 프로젝트는 하위 3개 프로젝트의 이름과 패키지 구조 및 공통으로 사용할 공통 라이브러리를 지정함
- 4개의 프로젝트는 한 팀에서 개발하고 배포함

[출처 : 자바 기반의 마이크로서비스 이해와 아키텍처 구축하기, 박성훈]

클라우드 네이티브 정보시스템 설계 단계

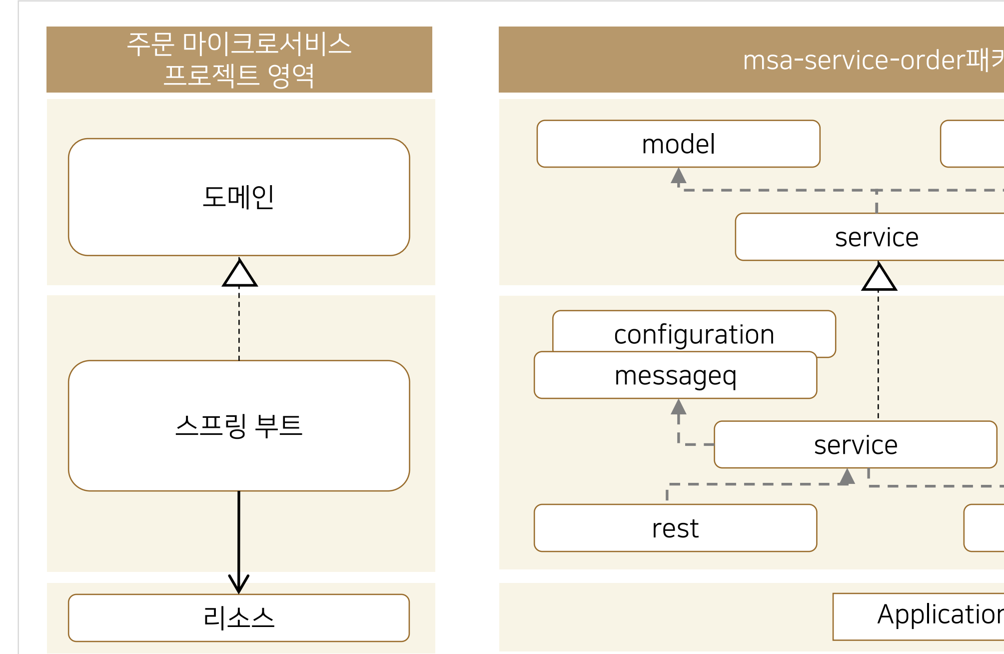
8.1 도메인 모델링을 통한 마이크로서비스 설계

8.1.3 마이크로서비스 프로젝트 설계

8.1.3.2 마이크로서비스 프로젝트 패키지 구조

- **마이크로서비스 프로젝트 구조는 크게 도메인 중심의 기술 독립적인 영역과 기술 종속적인 영역(스프링 부트)으로 구분된다. 프로젝트를 도메인과 기술종속적 영역으로 분리함으로써 서비스 간 독립성을 유지하고, 비즈니스 변화에 유연하게 대응할 수 있다.**

[그림 8-12] 주문 마이크로서비스 프로젝트 패키지 구조 및 구성 내역



패키지	서브 패키지	설명
도메인	model	• 회원 확인 프로젝트
	repository	• 주문 서비스 프로젝트
	service	• 주문 처리상태 확인 서비스 프로젝트
스프링 부트	configuration	• 데이터베이스, 로깅, 메시지 등 설정
	messageq	• 큐잉 시스템 설정
	Repository	• JPA, Mybatis 등 데이터 저장 방식 구현
	rest	• REST URI 설정 및 구현
	Service	• 도메인에서 구현한 서비스 실행
리소스	Application.yml(파일)	• 마이크로서비스명, PORT 등 설정

[출처 : 자바 기반의 마이크로서비스 이해와 아키텍처 구축하기, 박성훈]

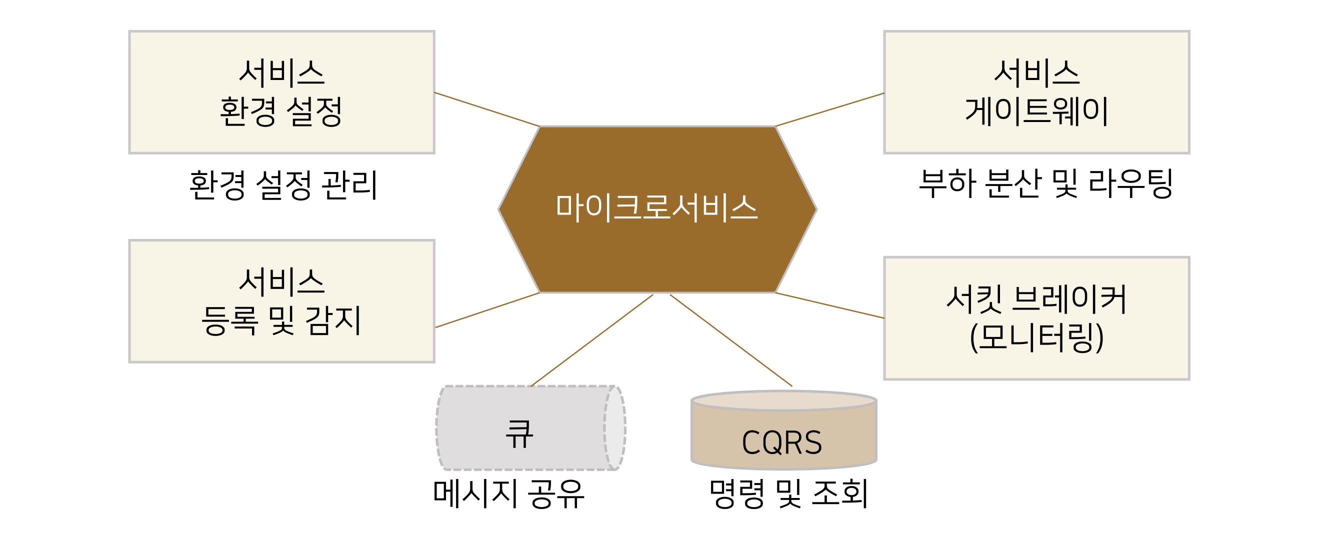
클라우드 네이티브 정보시스템 설계 단계

8.2 마이크로서비스 아키텍처 설계

8.2.1 마이크로서비스 아키텍처 개념

- **마이크로서비스와 마이크로서비스 아키텍처를 구성하기 위해서 환경 설정 관리, 서비스 라우팅, 서비스 등록 감지, 서킷 브레이커, 메시징 시스템, CQRS와 같은 몇 가지의 기능과 이를 잘 구성하고 운영하기 위한 서비스 구성 체계, 테스트 체계, 빌드 및 배포 체계 등이 필요하다.**
- **넷플릭스에서 공개한 오픈소스를 이용하여 환경 설정 관리, 서비스 등록 및 감지, 서비스 라우팅, 서비스 모니터링 등 스프링 클라우드 기반의 마이크로서비스 아키텍처 구성 설계와 운영 방안에 대해 알아본다.**

[그림 8-13] 마이크로서비스 아키텍처 구성 예시



구성요소	설명
서비스 환경 설정	• 외부 환경 설정 관리는 시스템에서 참조해야 하는 환경 설정 정보 등을 별도의 저장소에서 관리하여 애플리케이션이 배포된 환경에 구애받지 않고 적절한 환경 정보들을 참조할 수 있는 기능을 제공하는 서비스임(예. IP, 포트, 서비스명, 변수 등)
서비스 등록 및 감지	• 마이크로서비스가 시스템 등록되는 것을 자동으로 감지하여 서비스 게이트웨이가 자동으로 인지할 수 있게 지원하는 기능임
서비스 게이트웨이 (API 게이트웨이)	• 마이크로서비스에 대한 요청을 받아서 해당 요청에 필요한 서비스를 연결해 주는 역할을 하며, 마이크로서비스 아키텍처에서는 독립된 서비스가 각자 서버에서 구동됨
서킷 브레이커	• 서킷 브레이커 기능은 특정 서비스가 정상적으로 동작하지 않을 경우, 다른 기능으로 대체 수행시켜서 장애를 회피하는 기능임
큐잉 시스템 (메시지 브로커)	• 마이크로서비스 간 데이터의 전달이 필요하고 느슨할 결합을 위해 큐잉 시스템을 사용하며, 카프카 같은 메시지 큐가 대표적 솔루션임
CQRS	• CQRS(Command and Query Responsibility Segregation)는 명령을 처리하는 책임과 조회를 처리하는 책임을 분리 구현한다는 개념임

8.2 마이크로서비스 아키텍처 설계

8.2.2 마이크로서비스 아키텍처 패턴

[그림 8-14] 참고. 마이크로서비스 아키텍처 패턴 (크리스 리처드슨)

마이크로서비스 아키텍처 패턴

- 마이크로서비스 아키텍처 패턴은 전체 애플리케이션을 마이크로서비스 아키텍처로 구성할 때 유용한 패턴의 모음집임
- 마이크로서비스 아키텍처 패턴은 크게 애플리케이션 패턴, 애플리케이션 인프라 패턴, 인프라 패턴으로 구성됨
  - 애플리케이션 패턴 : 개발자가 부딪치는 문제를 해결
  - 애플리케이션 인프라 패턴 : 개발에도 영향을 미치는 인프라 문제를 해결
  - 인프라 패턴 : 주로 개발 영역 밖의 인프라 문제를 해결
- 마이크로서비스 아키텍처 패턴을 활용하여 설계 단계의 개발 표준을 수립하도록 함

[출처 : 마이크로서비스 패턴, 크리스 리처드슨]

8.2 마이크로서비스 아키텍처 설계

8.2.3 쿼리 패턴 – API 조합 패턴 설계

8.2.3.1 API 조합 패턴 필요성

- 기존의 모놀리식 애플리케이션은 하나의 DB에서 여러 테이블을 조인하여 다양한 형태로 데이터를 프론트엔드에 전달하는 것이 용이한 구조였다.
- 기존 업무를 마이크로서비스로 전환할 때 기존의 복잡한 쿼리 및 DB 조인 관계를 마이크로서비스 아키텍처에서 처리해야 하는 경우 네트워크 비용, 구현의 복잡성, 개발 관련 역할 등을 고려하여 설계하고 구현해야 한다.
- 마이크로서비스 아키텍처에서는 DB조인이 안 되기 때문에 몇 가지 API 조합패턴, 프론트엔드에서 직접 여러 번 호출 처리, API 조합기를 통한 처리, API 게이트웨이에서 데이터 조합 등을 통해 처리할 수 있다.

[그림 8-15] 마이크로서비스 애플리케이션의 조회 구조

프론트엔드(뷰)
주문내역 조회화면
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
상태 : 준비 중
접수 : 11:30 AM
결제 : 현장 결재(카드)
주소 : XXXXXX
???
마이크로서비스 증가에 따른 병목현상 발생
주문 서비스
메뉴 서비스
주방 서비스
계산 서비스
배송 서비스
마이크로서비스 DB
주문 DB
주문ID : XXX-XXX
접수 : 11:30 AM
메뉴 DB
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
주방 DB
주문ID : XXX-XXX
상태 : 준비 중
회계 DB
주문ID : XXX-XXX
결제 : 현장 결재(카드)
배달 DB
주문ID : XXX-XXX
주소 : XXXXXX

클라우드 네이티브 정보시스템 설계 단계

8.2 마이크로서비스 아키텍처 설계

8.2.3 쿼리 패턴 – API 조합 패턴 설계

8.2.3.2 프론트엔드에서의 조회 패턴

- **프론트엔드에서의 조회 패턴은 프론트엔드 서비스에서 직접 여러 번의 REST API를 호출하여 데이터를 조회한 후 화면에 출력하는 방식을 말한다.**
- **가장 일반적이며 쉽게 구현되는 방식이지만, 프론트엔드와 백엔드 간 결합도가 높아지고, 네트워크 비용이 증가하는 단점이 있다.**

[그림 8-16] 프론트엔드 조회 패턴 예시

프론트엔드 (뷰)
주문내역 조회화면
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
상태 : 준비 중
접수 : 11:30 AM
결제 : 현장 결재(카드)
주소 : XXXXXX
프론트엔드에서 개별 서비스에 대해 여러 번 API를 호출하여 데이터를 조회한 후 화면에서 통합 출력
1
2
3
4
5
주문 서비스
메뉴 서비스
주방 서비스
계산 서비스
배송 서비스
마이크로서비스 DB
주문 DB
주문ID : XXX-XXX
접수 : 11:30 AM
메뉴 DB
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
주방 DB
주문ID : XXX-XXX
상태 : 준비 중
회계 DB
주문ID : XXX-XXX
결제 : 현장 결재(카드)
배달 DB
주문ID : XXX-XXX
주소 : XXXXXX

클라우드 네이티브 정보시스템 설계 단계

8.2 마이크로서비스 아키텍처 설계

8.2.3 쿼리 패턴 – API 조합 패턴 설계

8.2.3.3 API 조합 서비스 패턴

- API 조합 서비스를 통해 데이터를 조합하여 프론트엔드에 응답하는 방식이며, 개발자는 API 조합 서비스 를 개발하여야 한다.
- 프론트엔드에서는 하나의 마이크로서비스만을 호출하고, 조합 역할을 하는 서비스는 여러 마이크로서비 스를 API 호출하여 데이터를 조합한 후 프론트엔드에 응답한다.
- API 조합 패턴은 구현이 용이하지만 별도의 서비스 개발이 필요하고, 프론트엔드 입장에서 한 번에 데이터를 받아야 하기 때문에 응답 대기시간이 길어질 수도 있다. 또한 API 조합 서비스에서는 병렬로 호출하도록 로직을 구현해야 하며, 순차적으로 구현해야 할 경우 딜레이 타임이 길어지는 단점도 있다.

[그림 8-17] API 조합 서비스 패턴 예시

프론트엔드 (뷰)
주문내역 조회화면
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
상태 : 준비 중
접수 : 11:30 AM
결제 : 현장 결재(카드)
주소 : XXXXXX
프론트엔드에서는 API조합 서비스만 호출, 결과 수신 후 화면에 출력
주문내역 서비스
API 조합 서비스를 별도로 두고 개별 서비스를 여러 번 호출하여 데이터를 조회한 후 프론트엔드로 반환
1
2
3
4
5
주문 서비스
메뉴 서비스
주방 서비스
계산 서비스
배송 서비스
마이크로서비스 DB
주문 DB
주문ID : XXX-XXX
접수 : 11:30 AM
메뉴 DB
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
주방 DB
주문ID : XXX-XXX
상태 : 준비 중
회계 DB
주문ID : XXX-XXX
결제 : 현장 결재(카드)
배달 DB
주문ID : XXX-XXX
주소 : XXXXXX

8.2 마이크로서비스 아키텍처 설계

8.2.3 쿼리 패턴 – API 조합 패턴 설계

8.2.3.4 API 게이트웨이에서 조합하는 패턴

- **API 게이트웨이를 통해 데이터를 조합하여 프론트엔드에 응답하는 방식이며, API 게이트웨이에서 API를 조합하는 기능이 필요하다. 즉, API 조합기 역할을 API 게이트웨이가 대신 수행하는 방식이다.**
- **이 패턴의 장점은 API 게이트웨이의 제공 기능을 이용하게 되어 별도의 조합 서비스 개발 부담이 적다는 것이다.**
- **하지만 API 조합 기능을 지원하는 API 게이트웨이를 찾기 어려운 점이 단점이 될 수 있다.**

[그림 8-18] API 게이트웨이 조합 패턴 예시

프론트엔드 (뷰)
주문내역 조회화면
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
상태 : 준비 중
접수 : 11:30 AM
결제 : 현장 결재(카드)
주소 : XXXXXX
프론트엔드에서는 API 게이트웨이 호출 결과 수신 후 화면에서 출력
API 게이트웨이
API 게이트웨이의 API 조합 기능을 이용하여 개별 서비스를 여러 번 호출하여 데이터를 조회한 후 프론트엔드로 반환
1
2
3
4
5
주문 서비스
메뉴 서비스
주방 서비스
계산 서비스
배송 서비스
마이크로서비스 DB
주문 DB
주문ID : XXX-XXX
접수 : 11:30 AM
메뉴 DB
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
주방 DB
주문ID : XXX-XXX
상태 : 준비 중
회계 DB
주문ID : XXX-XXX
결제 : 현장 결재(카드)
배달 DB
주문ID : XXX-XXX
주소 : XXXXXX

08 클라우드 네이티브 정보시스템 설계 단계

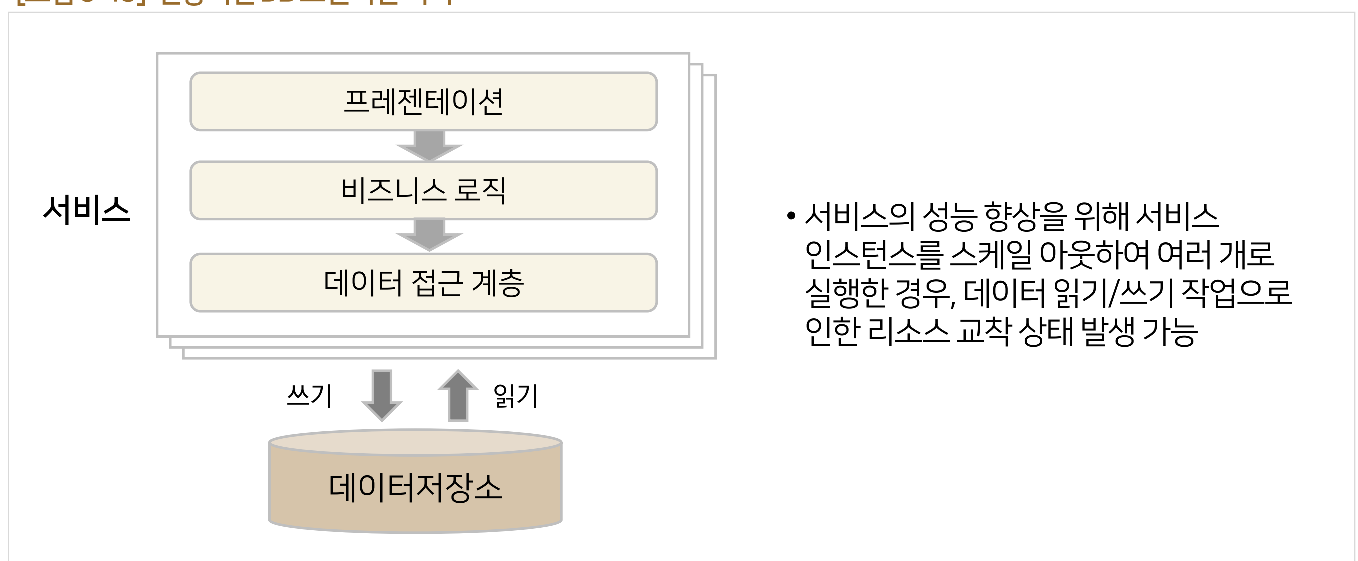
8.2 마이크로서비스 아키텍처 설계

8.2.4 쿼리 패턴–CQRS 패턴 설계

8.2.4.1 CQRS 개념

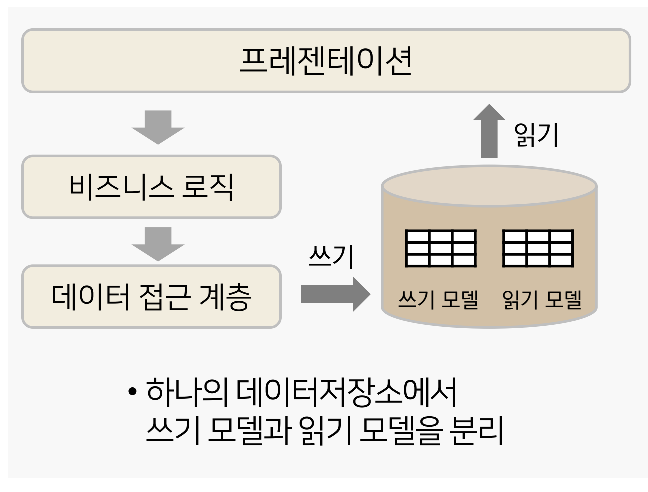
- CQRS(Command Query Responsibility Segregation : 명령과 조회의 책임 분리)는 기존에 동일한 저장소에 데이터를 넣고, 입력·수정·삭제·조회를 한꺼번에 처리하는 방식에서 입력·수정·삭제·조회와 같은 데이터에 대한 명령(Command)과 데이터에 대한 조회(Query)를 분리하는 것을 의미한다.
- 전통적인 DB 트랜잭션 처리 방식과 CQRS 패턴에 의한 트랜잭션 처리 방식의 차이점은 다음과 같다.

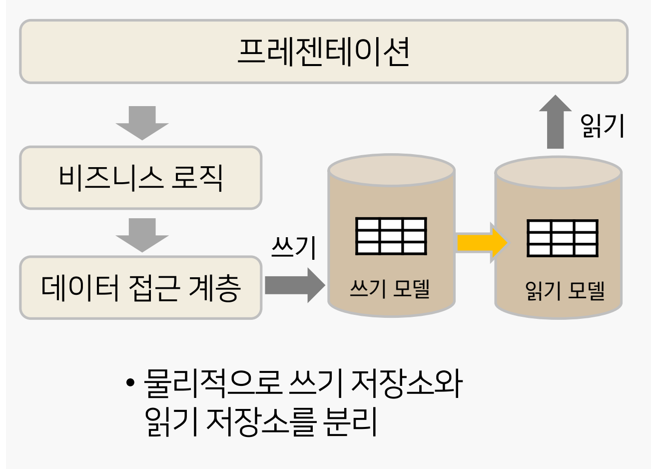
[그림 8-19] 전통적인 DB 트랜잭션 처리



[그림 8-20] CQRS 패턴에 의한 트랜잭션 처리

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





8.2 마이크로서비스 아키텍처 설계

8.2.4 쿼리 패턴-CQRS 패턴 설계

8.2.4.2 CQRS 패턴을 적용한 설계

- **CQRS는 복잡한 쿼리를 처리해야 할 경우 별도의 마이크로서비스를 호출하고, 다른 마이크로서비스로는 CRUD를 처리한다.**
- **CQRS 패턴의 적용 시 다양한 쿼리를 효율적으로 구현하고, 응답 시간을 단축할 수 있다. 하지만 구현이 복잡하고, 데이터의 시차가 발생할 수 있다.**
- **간단한 데이터 조인은 API조합 패턴으로 구현하고 복잡한 쿼리를 처리해야 할 경우에 CQRS 패턴을 적용하는 것을 권장한다.**

[그림 8-21] CQRS 패턴의 적용 예시

프론트엔드 (뷰)
주문내역 조회화면
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
상태 : 준비 중
접수 : 11:30 AM
결제 : 현장 결재(카드)
주소 : XXXXXX
주문내역 조회 서비스 (읽기 전용) 및 뷰 DB 추가 개발 필요
GET /orders/orderhistory/{orderId}
POST/PUT/DELETE
POST/PUT/DELETE
주문 서비스
메뉴 서비스
주방 서비스
계산 서비스
배달 서비스
주문내역 조회서비스
마이크로서비스 DB
주문 DB
주문ID : XXX-XXX
접수 : 11:30 AM
메뉴 DB
주문ID : XXX-XXX
메뉴 : 탕수육, 짜장면
주방 DB
주문ID : XXX-XXX
상태 : 준비 중
회계 DB
주문ID : XXX-XXX
결제 : 현장 결재(카드)
배달 DB
주문ID : XXX-XXX
주소 : XXXXXX
뷰 DB
이벤트 발행
이벤트 발행
이벤트 발행
이벤트 발행
이벤트 발행
이벤트 구독
메시지 브로커

클라우드 네이티브 정보시스템 설계 단계

8.2 마이크로서비스 아키텍처 설계

8.2.5 트랜잭션 관리 설계

8.2.5.1 SAGA 패턴 개념 및 필요성

- SAGA 패턴이란 마이크로서비스 간 이벤트를 주고받아 특정 마이크로서비스에서의 작업이 실패하면 이전까지의 작업이 완료된 마이크서비스들에게 보상(Complementary) 이벤트를 소싱함으로써 분산 환경 에서 원자성(Atomicity)을 보장하는 패턴이다.
- SAGA 패턴은 트랜잭션의 관리 주체가 DBMS가 아닌 애플리케이션이다. 애플리케이션이 분산되어 있을 때 각 애플리케이션 하위에 존재하는 DB는 로컬 트랜잭션 처리만 담당한다.
- 기존의 모놀리식 환경에서는 분산 트랜잭션 처리를 위해 DBMS가 제공하는 2PC(2 Phase Commit) 기법에 의한 커밋과 롤백 처리를 통해 데이터의 일관성을 유지한다.
- 하지만 마이크로서비스 아키텍처에서는 서로 다른 애플리케이션이 API를 통해 트랜잭션을 처리하기 때문에 2PC와 같은 방식은 사용할 수 없다. 마이크로서비스는 각각의 트랜잭션 처리를 위해 개별 DB를 보유하고 있으며, 각 서비스는 타 서비스의 개별 DB를 호출하지 못하고, 자신의 API를 통해서만 접근 가능하기 때문이다.

[그림] 8-22] SAGA 패턴의 필요성

"마이크로서비스 아키텍처에서 서비스 간 분리된 DB의 일관성 유지를 위해 SAGA 패턴 필요"

주문API
주문 서비스
고객API
고객 서비스
호출 금지
주문 DB
분산 트랜잭션 문제 발생 가능
고객 DB

- 마이크로서비스는 각각의 비즈니스 처리를 위한 영구 개별 DB를 보유함
- 마이크로서비스의 개별 DB는 타 서비스에 의해 직접적으로 호출되지 않음
- 마이크로서비스는 자신의 API를 통해서만 접근 가능

08 클라우드 네이티브 정보시스템 설계 단계

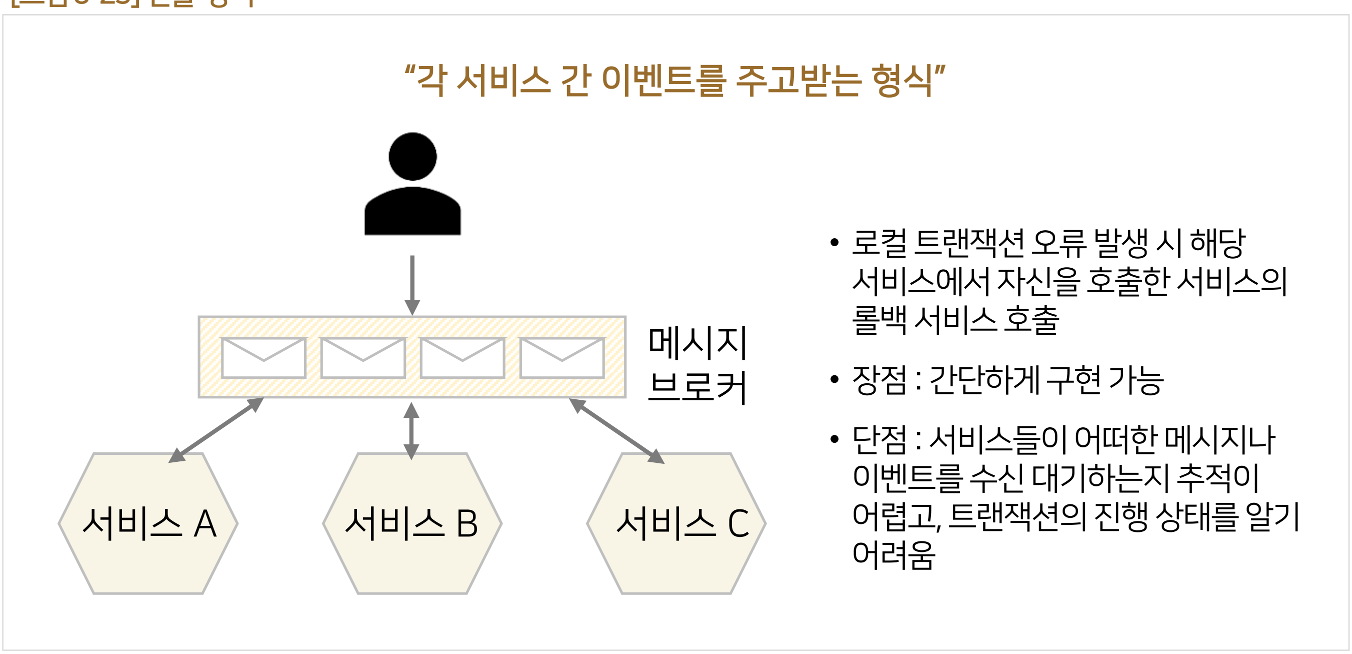
8.2 마이크로서비스 아키텍처 설계

8.2.5 트랜잭션 관리 설계

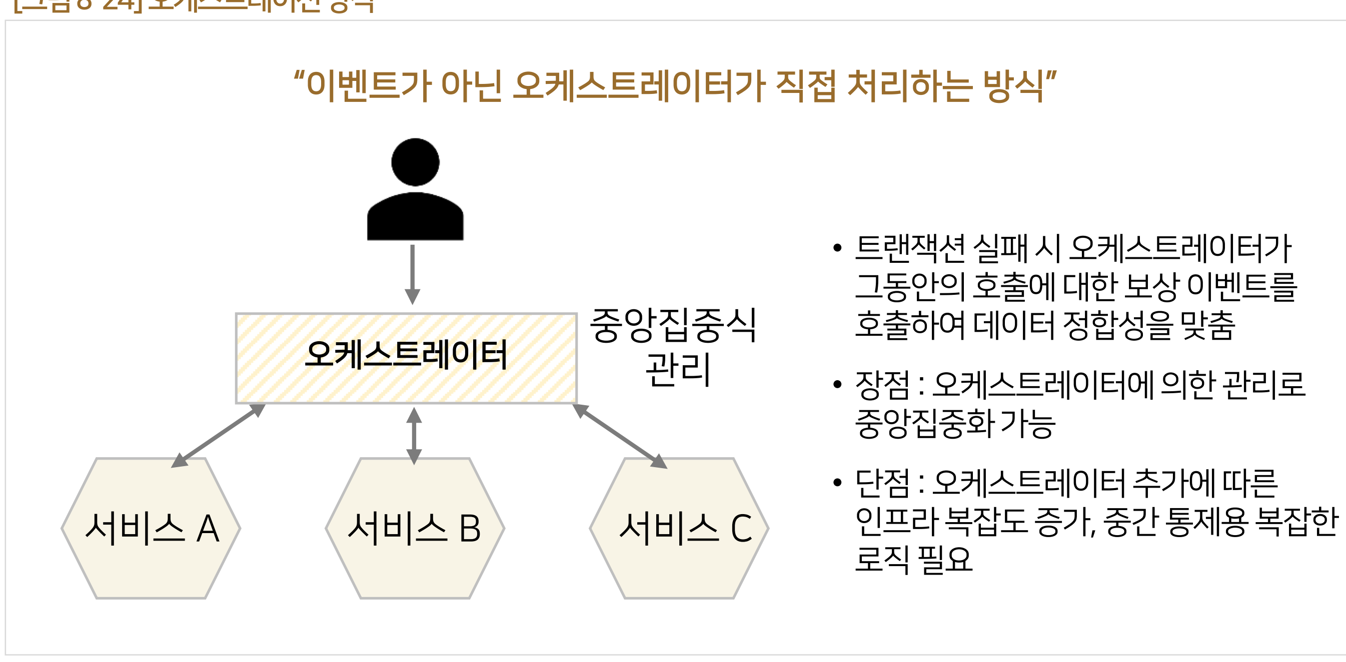
8.2.5.2 SAGA 패턴의 유형

- **SAGA 패턴은 분산 트랜잭션 시나리오에서 마이크로서비스 전반의 데이터 일관성을 관리하는 방법으로 이벤트 기반의 연출(Choreography) 방식과 명령어 기반의 오케스트레이션 방식이 존재한다.**

[그림 8-23] 연출 방식



[그림 8-24] 오케스트레이션 방식



8.2 마이크로서비스 아키텍처 설계

8.2.6 서비스 간 통신 방식 설계

8.2.6.1 서비스 간 통신 패턴 유형

- **마이크로서비스 아키텍처에서 각 서비스는 미리 정의된 API를 통해 통신을 하게 된다.**
- **서비스 간 통신 패턴은 동기식(Sync) 통신 방식과 비동기식(Async) 통신 방식이 존재하며, 이 두 개 방식의 차이점은 다음과 같다.**

[그림 8-25] 동기식 통신 방식

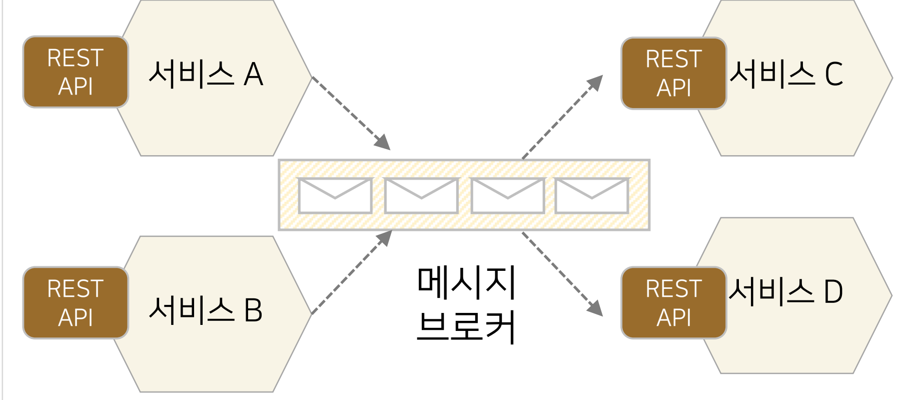
"동기식 통신은 요청과 응답 기반 프로토콜 사용"

UI
GET/서비스 A
요청
REST API
서비스 A
응답
GET/서비스 B
요청
REST API
서비스 B
응답

- 동기적 통신은 HTTP나 RPC 같은 프로토콜을 이용하여 다른 서비스가 노출한 API를 호출하는 방식
- 호출자가 응답을 기다리기 때문에 동기적 메시징 패턴임
- 서비스 A → 서비스 B를 호출하는 관계에서 서비스 A는 서비스 B가 끝날 때까지 기다려야 함

[그림 8-26] 비동기식 통신 방식

"비동기식 통신은 메시지 교환을 위해 메시지 브로커 사용"



- 비동기식 통신은 서비스에서 메시지 브로커로 메시지를 보낸 뒤 응답을 기다리지 않는 방식임
- 서비스 A → 서비스 B를 호출하는 관계에서 서비스 A는 서비스 B를 끝날 때까지 기다리지 않아도 됨

※ 비동기식 통신은 이벤트에 반응한다는 의미로 이벤트 기반 아키텍처(EDA : Event Driven Architecture)라고 함

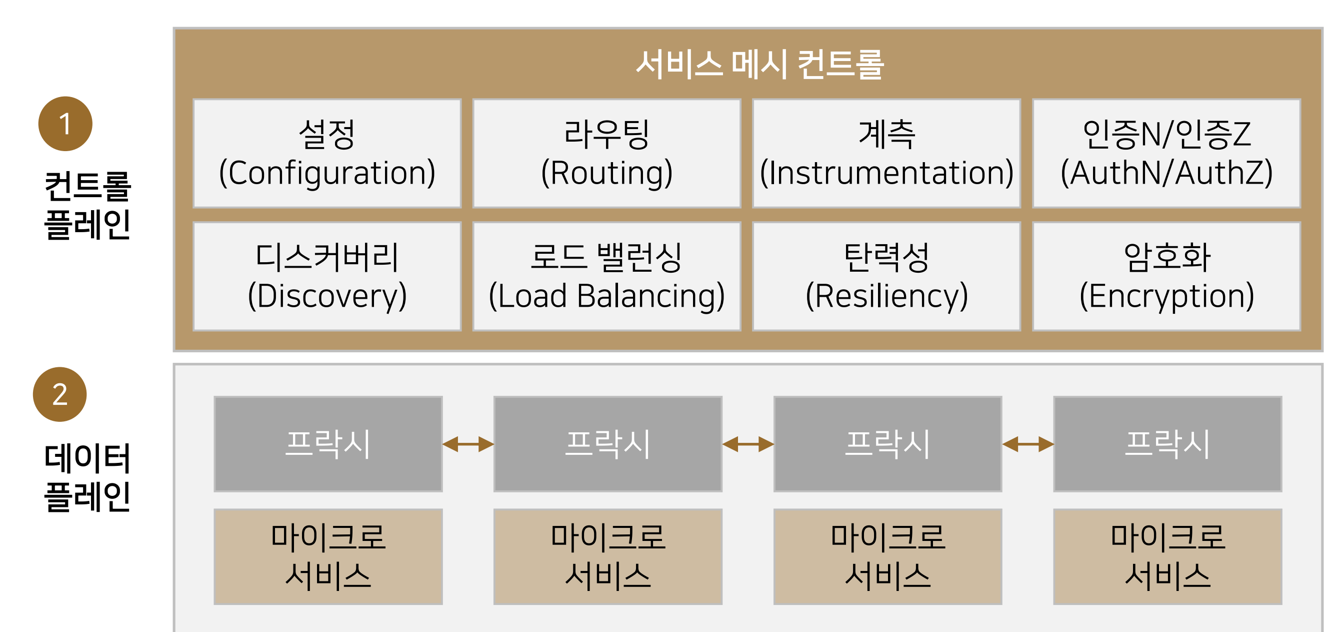
8.2 마이크로서비스 아키텍처 설계

8.2.7 서비스 메시 설계

8.2.7.1 서비스 메시의 주요 구성

- **서비스 메시는 마이크로서비스 아키텍처에서 서비스 간 통신을 처리하는 인프라 레이어로 정의할 수 있다.**
- **애플리케이션 개발 시 비즈니스 코드와 통신을 위한 코드가 혼재될 경우, 문제가 복잡해질 수 있다. 개발자가 매커니즘과 구성 방법을 이해해야 했으나, 복잡성을 서비스 프락시에 주입하여 처리할 수 있다.**
- **서비스 메시는 논리적으로 컨트롤 플레인(Control Plain)과 데이터 플레인(Data Plain)으로 분할된다.**

[그림 8-27] 서비스 메시의 주요 구성



[출처 : Service Mesh Capabilities, 가트너, 2018]

1 컨트롤 플레인

- 컨트롤 플레인은 트래픽을 제어하는 정책 및 구성에 따라 프락시에게 설정값을 전달하고 관리하는 컨트롤러 역할

2 데이터 플레인

- 프락시를 통해 마이크로서비스 간에 오고 가는 네트워크 통신을 조정하고 제어함
- 가장 인기 있는 데이터 플레인은 Envoy Proxy가 가장 많이 활용됨
- Envoy Proxy는 C++로 개발된 고성능 프락시

8.2 마이크로서비스 아키텍처 설계

8.2.7 서비스 메시 설계

8.2.7.2 서비스 메시 구현 유형

- **서비스 메시 구현을 위해 스프링 클라우드와 이스티오를 가장 많이 사용하고 있으며, 쿠버네티스 기반에서는 스프링 클라우드와 쿠버네티스를 혼합한 구성도 가능하다.**
- **마이크로서비스 아키텍처에서 서비스 메시 구현 시 소스코드와 서비스 메시를 구현하는 코드가 얼마나 연관되어 있는지 정도에 따른 구현 유형은 다음과 같이 분류된다.**

[표 8-1] 서비스 메시 유형별 구성

기능 구분	라이브러리 방식		사이드카 방식
	스프링 클라우드 단독	스프링 클라우드 & 쿠버네티스	이스티오
서비스 디스커버리	넷플릭스 유레카	K8s 서비스	이스티오
서비스 라우팅	스프링 클라우드 게이트웨이	스프링 클라우드 게이트웨이	이스티오
로드밸런싱	스프링 클라우드 로드밸런서	K8s 서비스	이스티오
환경저장소	스프링 클라우드 컨피그	K8s 컨피그맵 & 시크릿	이스티오
인증 및 인가	스프링 클라우드 시큐리티	스프링 클라우드 시큐리티	이스티오
서킷브레이커	스프링 클라우드 Resilience4j	스프링 클라우드 Resilience4j	이스티오
특징	• 가장 오래된 기술이며, 적용 사례가 많음 • 컨테이너, 가상 서버 등 다양한 인프라 환경에 적용 가능 • VM, 쿠버네티스, 클라우드 파운드리 등 다양한 플랫폼 환경에 적용 가능 • 자바, 스프링 프레임워크에만 사용 가능 • 코드 레벨의 라이브러리 의존도가 높음	• 스프링 클라우드의 메시 기능과 쿠버네티스의 메시 기능을 통합하여 적용 • 자바만 사용 가능하지만, 코드 레벨의 라이브러리 의존도가 적음 - 히스트릭스 등 일부 예외 • 기존 스프링 클라우드 기반 메시 구현 변경 영향도 최소화하여 서비스 메시 구현	• 가장 최근 기술이며, 적용 사례가 적음 • 오픈쉬프트 4.3버전에서 이스티오 환경을 제공 • 쿠버네티스 환경에 적용 가능 • 개발언어, 프레임워크에 대한 종속성 없음 • 코드 레벨의 라이브러리 의존도가 없으나, yml방식의 설정에 대한 이해 필요

8.2 마이크로서비스 아키텍처 설계

8.2.7 서비스 메시 설계

8.2.7.3 스프링 클라우드 단독 방식

- **스프링 클라우드는 넷플릭스와 스프링 프레임워크에서 오픈소스 프로젝트로 제공되며, 라이브러리를 코드에 주입하여 사용하는 방식이다.**

[그림 8-28] 스프링 클라우드 단독 방식

클라이언트

API 게이트웨이
Gateway
LoadBalancer
Eureka Client
Resilience4j
Spring Boot

인증 및 인가
Spring Security
Spring Boot

서비스 레지스트리
Eureka
Spring Boot

마이크로서비스
Sleuth
LoadBalancer
Eureka Client
Resilience4j
Spring Boot

마이크로서비스
Sleuth
LoadBalancer
Eureka Client
Resilience4j
Spring Boot

마이크로서비스
Sleuth
LoadBalancer
Eureka Client
Resilience4j
Spring Boot

설정
Spring Config
Spring Boot

추적
Zipkin
Spring Boot

모니터링
Turbine
Spring Boot

범례
Spring Cloud

구성 요소	설명
Spring Boot	• 스프링 기반 애플리케이션을 쉽게 구현할 수 있도록 지원하고, 단독 실행이 가능한 내장 WAS컨테이너를 포함한 프레임워크
Spring Cloud Config	• 중앙집중식 컨피그 서비스로 애플리케이션 설정에 필요한 파일을 마이크로서비스와 분리 저장하는 역할을 담당 • 각 마이크로서비스 설정을 깃 또는 컨피그 파일로 저장관리
Spring Cloud Eureka	• 서비스 디스커버리를 제공하는 오픈소스로, 서버와 클라이언트로 구성 • 마이크로서비스가 기동 시 자신의 정보를 유레카 서버에 등록하고, 등록된 정보를 다른 클라이언트에게 전달하여 서로 통신이 가능하도록 함
Spring Security	• 서비스에 액세스할 수 있는 인증/인가를 처리하는 프레임워크로 인증서버가 발행한 토큰을 기반으로 서비스 통신 처리 • 자바스크립트 웹토큰(JWT)을 지원
Resilience4j	• 마이크로서비스의 회복성 패턴을 위한 서킷 브레이커
LoadBalancer	• 클라이언트가 서비스 호출에 대한 분산처리가 되도록 함
Turbin	• 여러 컨테이너의 히스트릭스 정보를 통합/조회
Sleuth/Zipkin	• Sleuth를 이용하여 트랜잭션의 고유 추적 식별자를 부여하며, 짚킨(Zipkin) 서버를 구성하여 트레이스 로그(Trace Log)를 수집 및 모니터링

8.2 마이크로서비스 아키텍처 설계

8.2.7 서비스 메시 설계

8.2.7.4 스프링 클라우드 & 쿠버네티스 방식

- **쿠버네티스는 서비스 디스커버리, 설정 관리, 로드밸런싱 기능을 기본으로 제공하고 있기 때문에 스프링 클라우드와 조합하여 서비스 메시를 구성할 수 있다.**

[그림 8-29] 스프링 클라우드 & 쿠버네티스 방식 구성

클라이언트
Ingress Controller
인증 및 인가
Spring Security
Spring Boot
Kubernetes Components
Kubernetes ConfigMap
Kubernetes Service
Kubernetes Service
Kubernetes Service
마이크로서비스
Resilience4j
Spring Boot
마이크로서비스
Resilience4j
Spring Boot
마이크로서비스
Resilience4j
Spring Boot
범례
스프링 클라우드
쿠버네티스

구성 요소	설명
Spring Boot	• 스프링 기반 애플리케이션을 쉽게 구현할 수 있도록 지원하고, 단독 실행이 가능한 내장 WAS컨테이너를 포함한 프레임워크
Ingress Controller	• 쿠버네티스 클러스터 외부에서 클러스터 내부 서비스로 접근할 수 있는 게이트웨이 역할 • Http 기반 L7 수준의 가상 호스팅 제공 도메인 기반 해당 서비스로 라우팅
Kubernetes Components	• 컨테이너/POD의 라이프사이클을 관리하고, 컨테이너/POD의 DNS, IP, port정보를 식별, 관리 • DNS 및 Config를 통해 Service 라우팅 정보 제공
Kubernetes ConfigMap	• Spring Cloud Config 대신 Kubernetes ConfigMap을 통한 설정 관리 저장소
Kubernetes Service	• 쿠버네티스에서 POD집합에 대한 단일 DNS명 및 고유 로컬서비스 IP를 부여하는 리소스로 POD 디스커버리 및 로드밸런싱 역할
Spring Security	• 서비스에 엑세스할 수 있는 인증/인가를 처리하는 프레임워크로 인증 서버가 발행한 토큰을 기반으로 서비스 통신처리 • 자바스크립트 웹토큰(JWT)을 지원
Resilience4j	• 마이크로서비스의 회복성 패턴을 위한 서킷 브레이커

8.2 마이크로서비스 아키텍처 설계

8.2.7 서비스 메시 설계

8.2.7.5 이스티오 방식

- **오픈소스 프로젝트로 제공되는 이스티오는 소스코드와 분리되어 인프라 형태로 마이크로서비스를 관리하는 방식이다.**

[그림 8-30] 이스티오 구성

구성 요소	설명
Spring Boot	• 스프링 기반 애플리케이션을 쉽게 구현할 수 있도록 지원하고, 단독 실행이 가능한 내장 WAS컨테이너를 포함한 프레임워크
Istio Gateway	• 이스티오 리소스를 쿠버네티스 클러스터 외부로 노출하기 위한 진입점 • 호스트명, 포트, 프로토콜 등을 정의
Virtual Service	• URI(Uniform Resource Identifier, 통합 자원 식별자)와 서비스의 매핑 정보를 정의 • 매핑 정보에 따라 라우팅 처리 및 API 버전에 대한 트래픽 가중치 분배
Destination Rule	• 마이크로서비스의 Connection pool, 로드밸런싱 정책(RoundRobin, Least Connection, Hash) • 서킷 브레이커 처리
Kubernetes Service	• 쿠버네티스에서 POD집합에 대한 단일 DNS명 및 고유 로컬서비스 IP를 부여하는 리소스로 POD 디스커버리 및 로드밸런싱 역할
Istio Control Plane	• Mixer : service mesh 전체 영역에서 액세스 제어 및 모니터링 지표 수집 • Pilot : Envoy 사이드카를 통한 서비스 디스커버리, 서킷브레이커 등 기능 • Citadel : TLS 암호화나 또는 사용자 인증에 필요한 인증서(Certification)을 관리
Jaeger	• 마이크로서비스 간 분산 트랜잭션 추적 및 모니터링
Kiali	• 마이크로서비스 간의 관계를 시각화

클라우드 네이티브 정보시스템 설계 단계

8.3 12가지 요소 기반 개발 원칙

8.3.1 개발 원칙 개요

- **12가지 요소(12 Factors)는 애플리케이션이 클라우드 환경에서 올바르게 동작하기 위해 지켜야 할 12가지 원칙을 말한다. 12가지 요소는 클라우드 PaaS 플랫폼과 클라우드 프레임워크 등을 활용하여 애플리케이션의 개발, 빌드, 배포, 운영 전반의 환경 구성과 개발 관련 원칙을 제시한다.**

[그림 8-31] 12 Factors 기반 개발 원칙

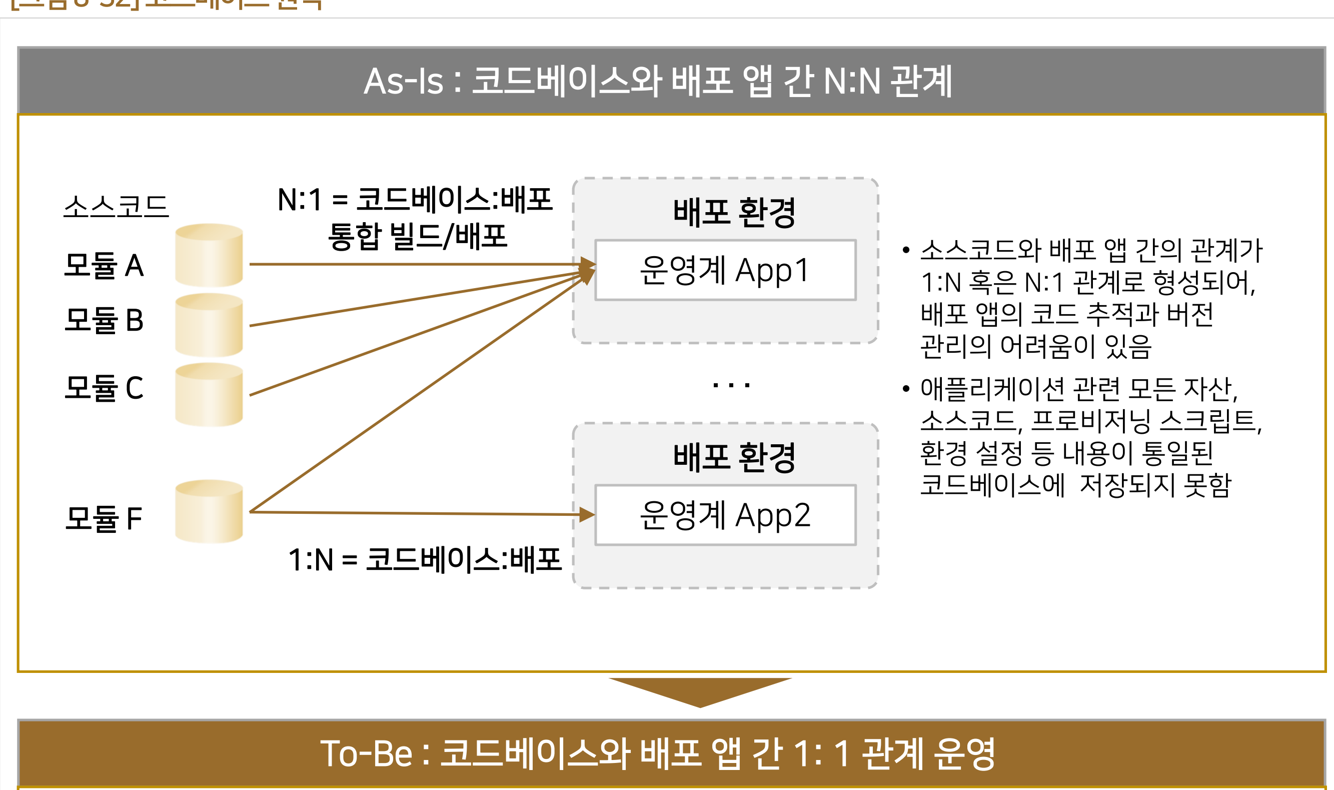
No.	요소	구분	설명
1	코드베이스 (Codebase)	CI/CD	• 하나의 코드베이스(소스코드)로 버전 관리 및 배포 관리 자동화 환경
2	의존성 (Dependencies)	코드, CI/CD	• 패키지, 라이브러리 등 의존관계는 명시적으로 선언하고 분리
3	설정 (Config)	코드, 배포	• 소스 코드와 설정 정보를 분리하고, 설정 정보는 환경변수에 저장
4	백엔드 서비스 (Backing Services)	배포	• 백엔드 서비스(DB, 메시지큐, 캐시 등)는 리소스 매핑으로 처리(연결/분리 용이)
5	빌드/배포/실행 (Build, Release, Run)	CI/CD	• 빌드와 실행 단계는 철저히 분리
6	프로세스 (Processes)	코드, 배포	• 애플리케이션은 무상태(Stateless) 프로세스로 실행
7	포트바인딩 (Port binding)	배포	• 애플리케이션은 독립적이며, http 같은 포트 바인딩을 이용한 서비스 공개
8	동시성 (Concurrency)	운영	• 애플리케이션을 수평적으로 확장, 프로세스 모델을 사용한 스케일아웃
9	폐기가능 (Disposability)	코드, 운영	• 빠른 시작과 정상적 종료(graceful shutdown) 지원으로 안정성 극대화
10	개발/운영환경 일치 (Dev/prod parity)	배포	• 가능한 동일한 개발, 스테이징, 운영 환경의 유지
11	로그 (Logs)	코드, 운영	• 로그파일을 이벤트 스트림으로 로그 처리(취합, 인덱싱, 분석)
12	운영관리 프로세스 (Admin processes)	운영	• 시스템 관리작업은 일회성 프로세스로 만들어서 실행

8.3 12가지 요소 기반 개발 원칙

8.3.2 코드베이스

- 애플리케이션은 1개의 코드베이스를 통해 관리되어야 하며, 동일한 코드로 개발 및 운영 환경에 배포되어야 하며, 코드베이스는 각 애플리케이션마다 단 1개 존재한다.
- 애플리케이션은 소스코드의 변경과 여러 환경으로 수차례 배포 시에도 추적 가능하다.

[그림 8-32] 코드베이스 원칙



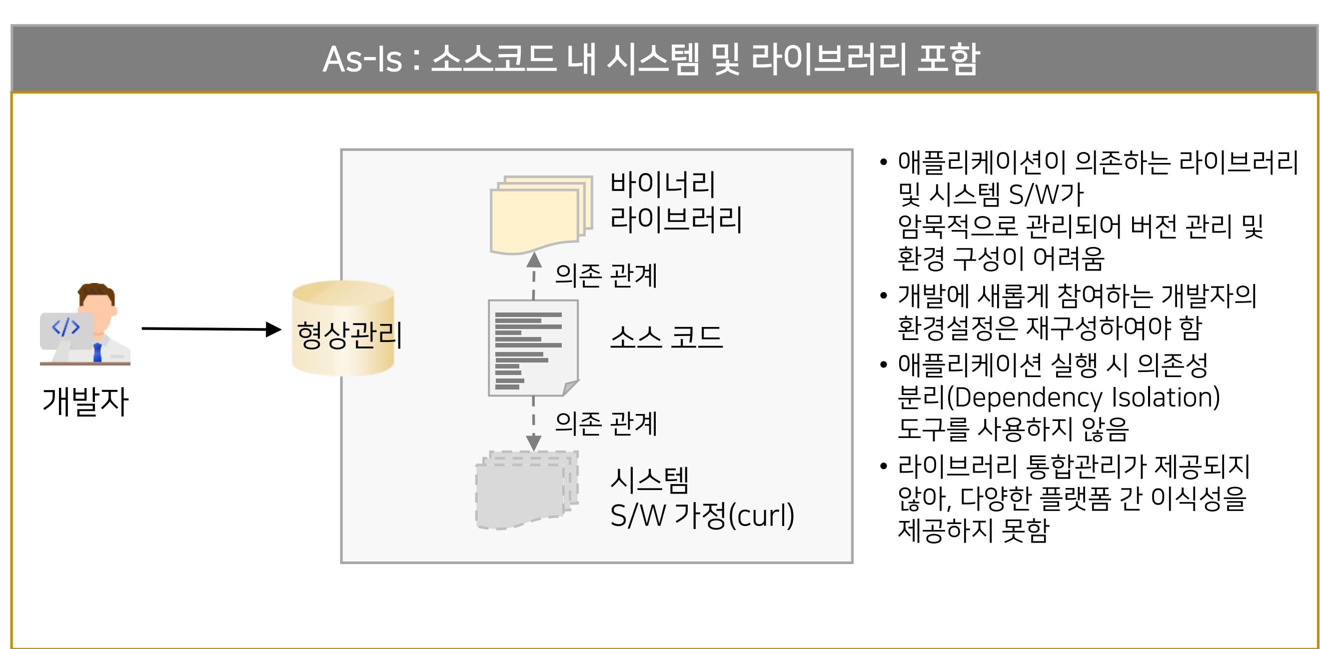
08 클라우드 네이티브 정보시스템 설계 단계

8.3 12가지 요소 기반 개발 원칙

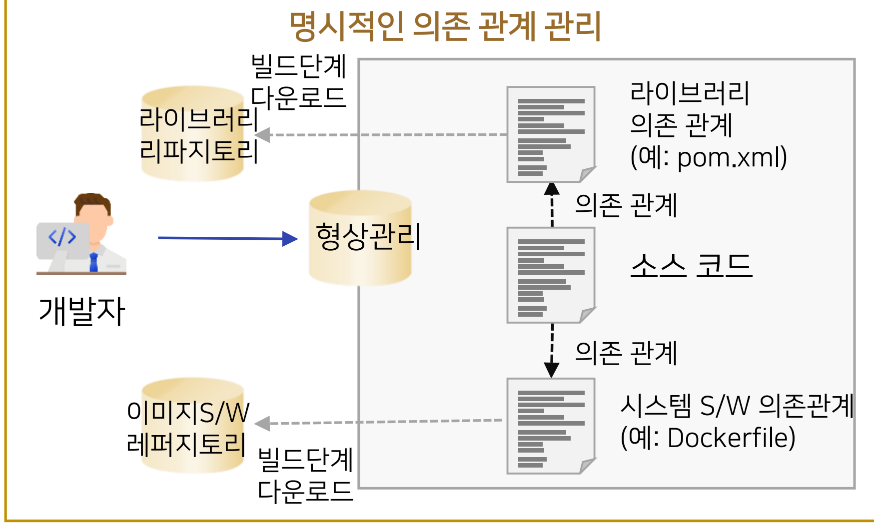
8.3.3 의존성

- **의존 관계에 있는 모든 애플리케이션은 명시적으로 의존성을 선언하며, 애플리케이션이 필요로 하는 라이브러리를 의존성 파일에 명시적으로 선언하여 사용한다.**
- **코드베이스는 개발자와 운영자의 협업을 지원하고, 체계적인 형상관리 및 배포를 지원한다.**

[그림 8-33] 의존성 원칙



To-Be : 의존성이 있는 라이브러리 통합관리



- 애플리케이션의 모든 의존성을 명시적으로 선언하여 사용
- 애플리케이션이 필요로 하는 라이브러리를 의존성 매니페스트(Manifest) 파일에 명시적으로 선언하여 사용
- 클라우드 네이티브 애플리케이션은 다양한 환경에 배포될 수 있으며 각 환경에서 정상 동작하도록 보장
- 모든 의존 관계는 의존성 선언을 통해 완전하고 정확하게 이루어짐
- 반복적인 배포 지원

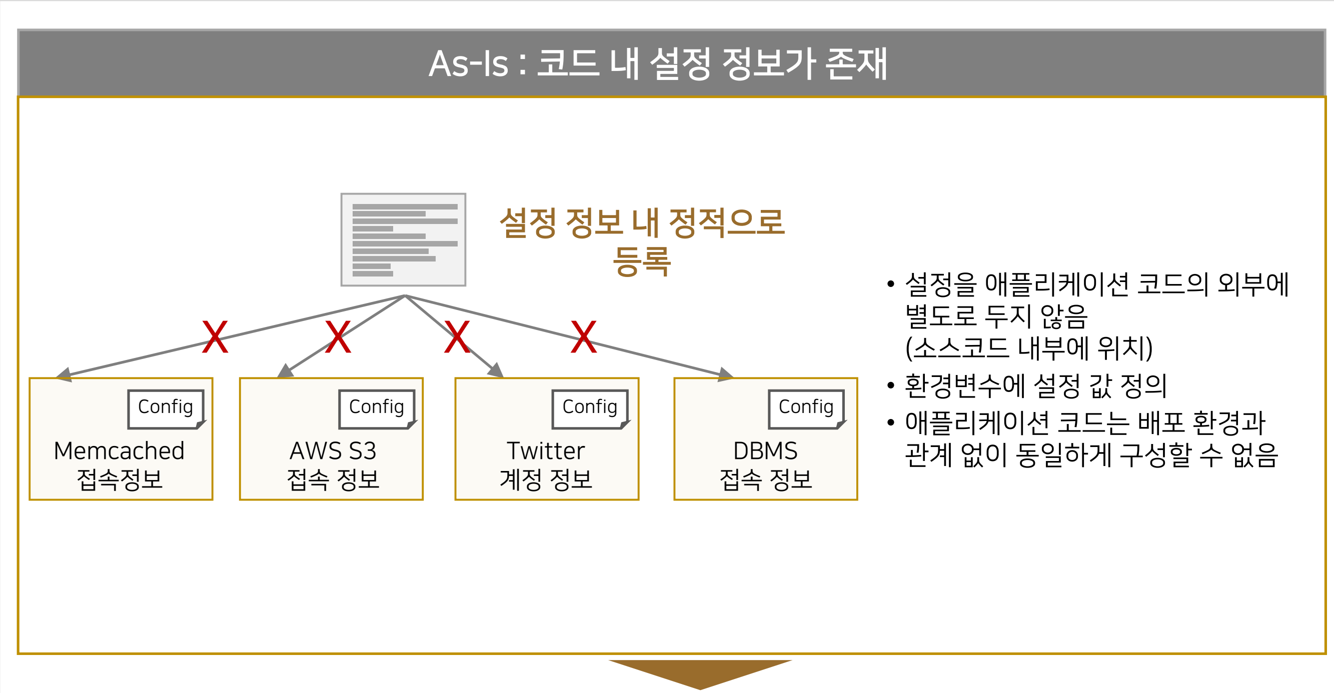
08 클라우드 네이티브 정보시스템 설계 단계

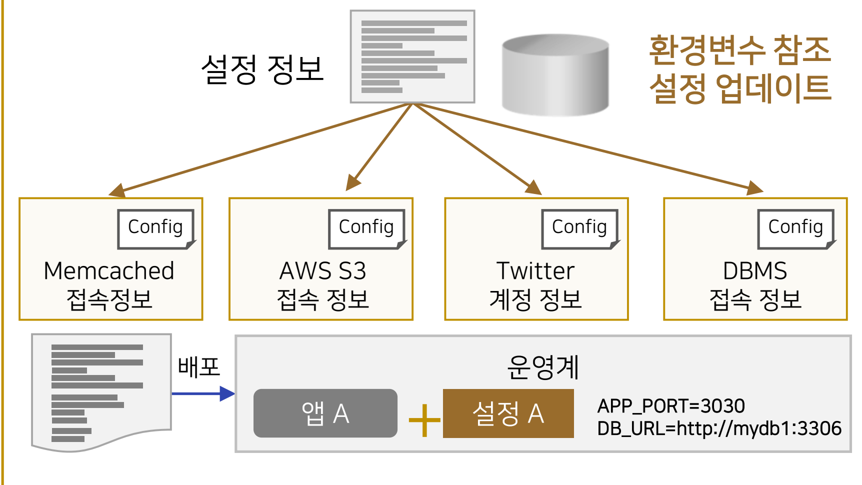
8.3 12가지 요소 기반 개발 원칙

8.3.4 설정

- **애플리케이션 실행 시 필요로 하는 설정 정보와 코드는 분리하여 관리한다.**
- **예를 들어 업무처리 로직과는 무관한 시스템 내외부의 리소스, 배포 단계, OS 등에 따라 설정 정보가 달라진다.**

[그림 8-34] 설정 원칙





8.3 12가지 요소 기반 개발 원칙

8.3.5 백엔드 서비스

- **DB, 캐시, 메시지/큐, 이메일 서비스, SNS, 클라우드 서비스 등 백엔드 서비스는 애플리케이션에 연결된(Attached) 리소스로 취급된다. 업무 및 기술의 변화, 비용적인 문제로 백엔드 서비스의 변경 시 설정(Config) 파일의 변경만으로 유연하고 신속한 대처가 가능하다.**

[그림 8-35] 백엔드 서비스 원칙

As-Is : 정적인 백엔드 환경으로 관리

애플리케이션

설정 파일 내 URL 및 계정 정보 연결

DB

상용 DBMS

DBMS 테이블 생성 및 기초 데이터 등록 상태 가정

이메일 서비스

클라우드 서비스

트위터

· · ·

DB_URL=http://192.168.10.3:3306

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

- 애플리케이션 백엔드는 사전 준비된 연결 설정이 마쳐진 상태 (예: DBMS 기초 데이터 생성)
- 백엔드 서비스를 설정 파일 내 정적 URL 및 계정 정보로 관리

To-Be : 애플리케이션에 연결된 리소스로 취급

애플리케이션

설정 파일은 환경변수 참조하여 자원 연결

DB

오픈소스 DBMS

DBMS 최초 연결 시 테이블 생성 및 기초 데이터 자동 등록

이메일 서비스

클라우드 서비스

트위터

· · ·

- 애플리케이션에 연결된 DBMS의 변경 시 애플리케이션 코드 변경이 없어야 함
- 독립적인 설정 파일에 백엔드 서비스 연결 정보 포함
- 설정 파일의 해당 내용 변경만으로 리소스를 변경하도록 함

설정 파일
APP_PORT=3030
DB_URL=http://dev:3306

변경 →

설정 파일
APP_PORT=3030
DB_URL=http://prd:3306

"설정 파일이나 Property 파일에 접속 정보(URL, Locator) 포함"

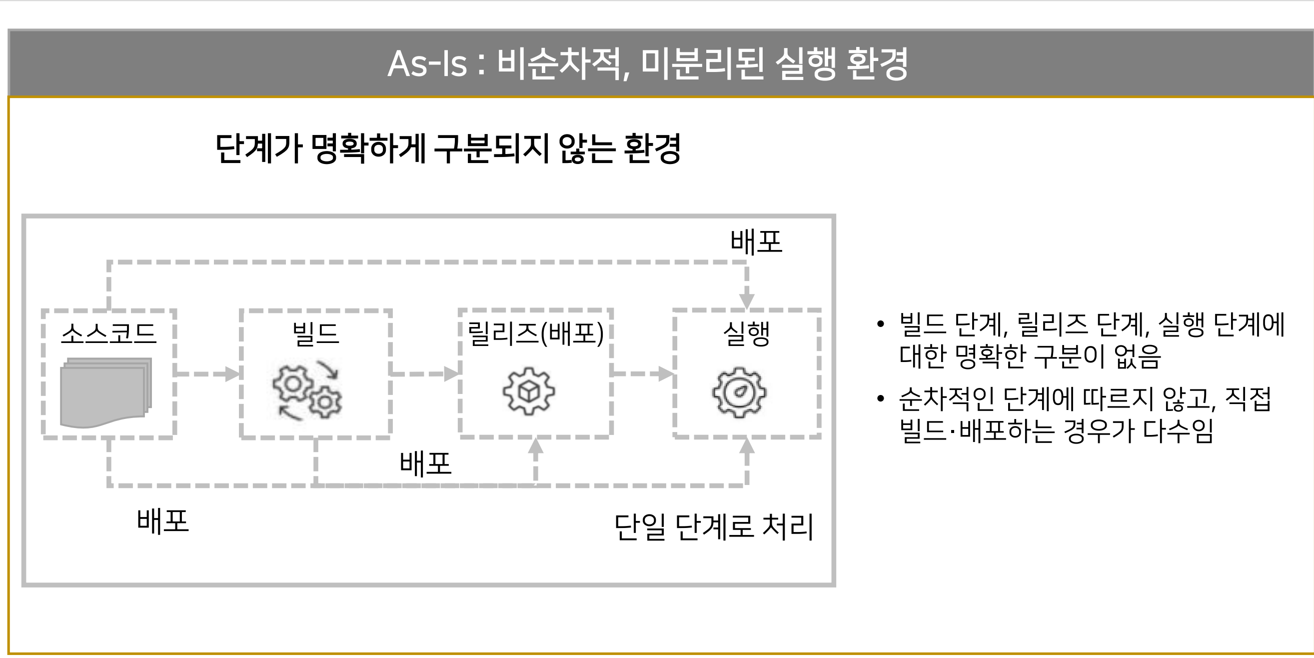
08 클라우드 네이티브 정보시스템 설계 단계

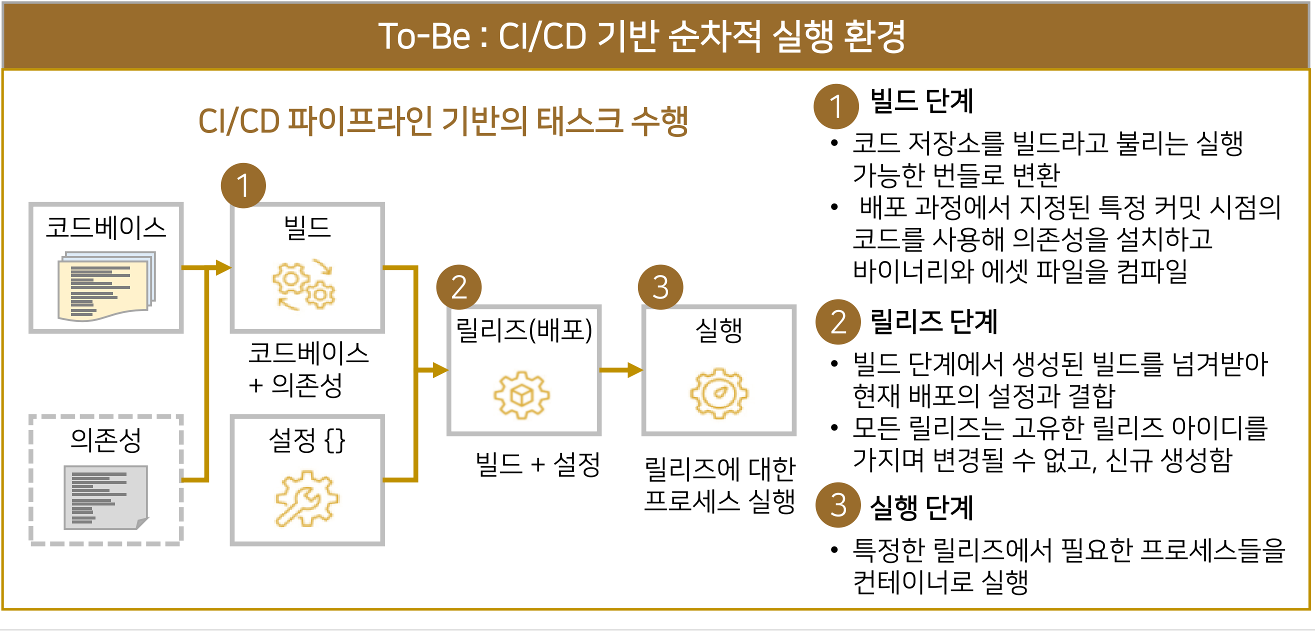
8.3 12가지 요소 기반 개발 원칙

8.3.6 빌드·릴리즈·실행 환경

- **코드베이스는 엄격하게 구분된 빌드, 릴리즈, 실행 3단계의 과정을 통해 배포가 이뤄지며, 각 단계는 엄격하게 분리되어야 한다. 이를 통해 예기치 않은 오류를 최소화할 수 있다.**
- **실행 시 코드 및 설정 파일에 대한 변경을 하지 않도록 한다.**

[그림 3-36] 빌드·릴리즈·실행 환경 원칙





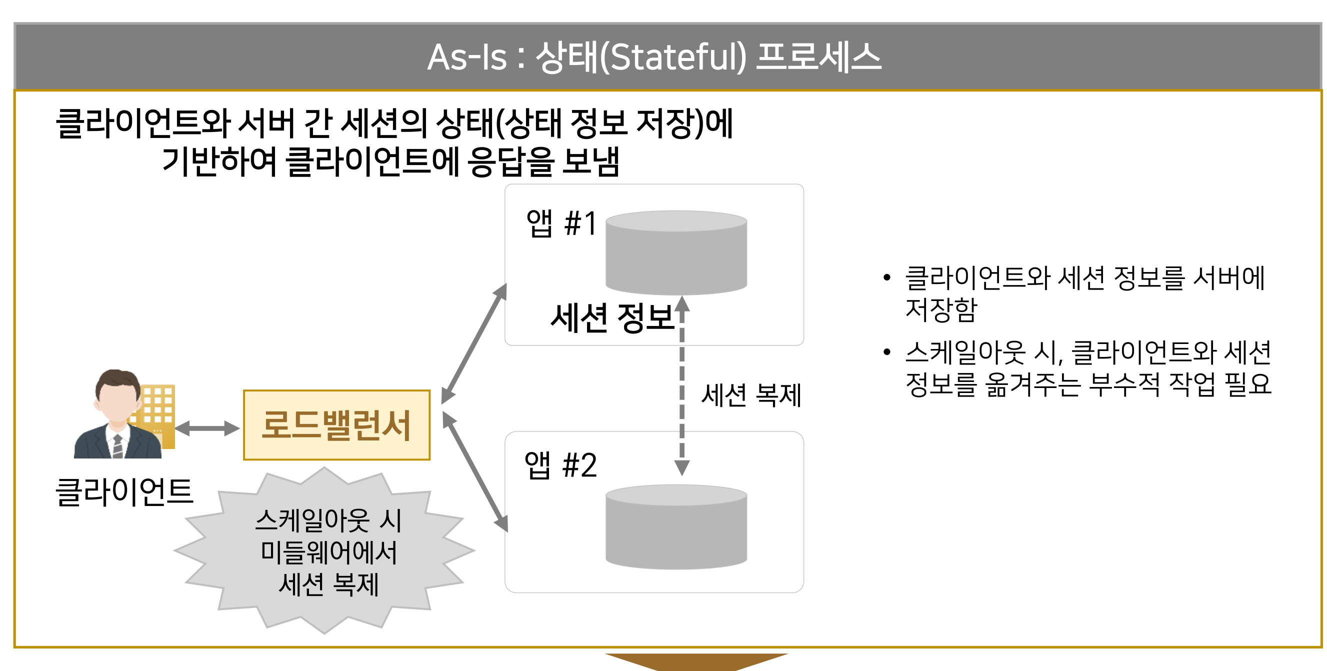
08 클라우드 네이티브 정보시스템 설계 단계

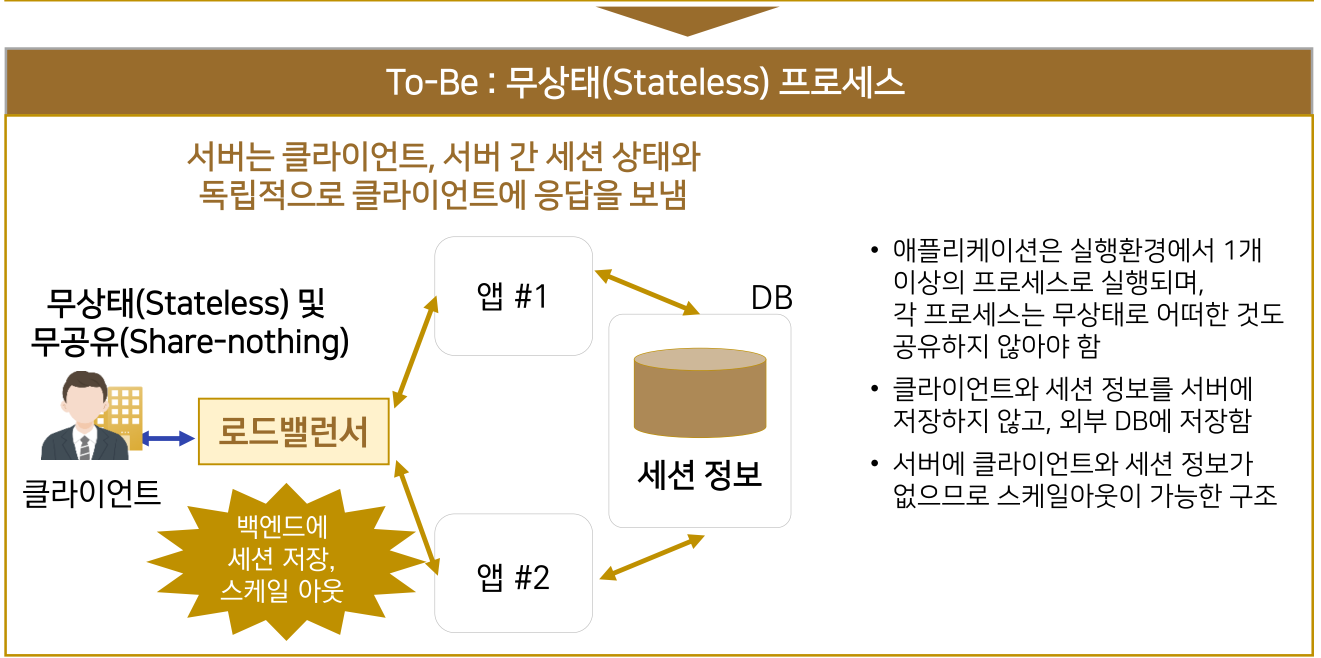
8.3 12가지 요소 기반 개발 원칙

8.3.7 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 구조이다.**

[그림 3-37] 무상태 원칙





8.3 12가지 요소 기반 개발 원칙

8.3.8 포트 바인딩

- **포트 바인딩은 메시지를 송수신하는 위치와 방법을 결정하는 구성 정보를 의미하며, 배포된 애플리케이션을 타 애플리케이션에서 접근할 수 있도록 포트 바인딩을 통해 서비스를 공개한다. (서비스 공개 및 접근성 제공)**

[그림 3-38] 포트 바인딩 원칙

As-Is : 온프레미스 환경의 앱 포트 수동 정의

포트 충돌이 되지 않도록 배포 시 수동 관리

서버

사용자 요청

80(포트)

사용자

Haproxy

81 (호스트) App1

82 (호스트) App 2

83 (호스트) App 3

- 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

서버(호스트 머신)

사용자 요청

80(포트)

사용자

라우터

18001 (호스트)

18002 (호스트)

18003 (호스트)

80 (컨테이너) App 1 (서비스)

80 (컨테이너) App 2 (서비스)

80 (컨테이너) App 3 (서비스)

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

8.3 12가지 요소 기반 개발 원칙

8.3.9 동시성

- 모든 일을 처리하는 하나의 프로세스 대신 기능별로 분리된 마이크로 프로세스를 실행하며, 프로세스 모델을 기반으로 수직적(Scale-up) 및 수평적(Scale-out) 확장성을 제공한다.
- 그리고 클라우드 서비스들은 규모에 따른 확장성이 기본적으로 제공되어야 한다.

[그림 3-39] 동시성 원칙

As-Is : 모든 일을 처리하는 하나의 프로세스

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

사용자 A
사용자 B
로드밸런서
APP #1~2
APP#1 (웹/API/ Batch)
APP#2 (웹/API/ Batch)

- 하드웨어 또는 소프트웨어 기반 부하분산 (L4)
- 전통적인 웹로직과 같은 미들웨어 기반으로 웹, API, 배치 애플리케이션을 통합하여 배포하는 방식
- 특정 유형의 프로세스 스케일아웃이 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

웹 #1~3
API #1~3
Batch (#1~3)
로드밸런서
웹서버
로드밸런서
MSA #1
MSA #2
MSA #3
로드밸런서
Batch Job
데이터 스토리지
Job

- 소프트웨어 기반 L4, L7 기능 제공
- 프로세스 유형의 확장, 워크로드의 다양성, 마이크로서비스 지원
- 동시성을 지원하는 경우에는 해당 요청사항을 충족하도록 애플리케이션 유형별로 수평적 확장이 가능함
- 비즈니스 서비스에 대한 부담으로 병목현상 발생 시 해당 계층을 독립적으로 확장 가능

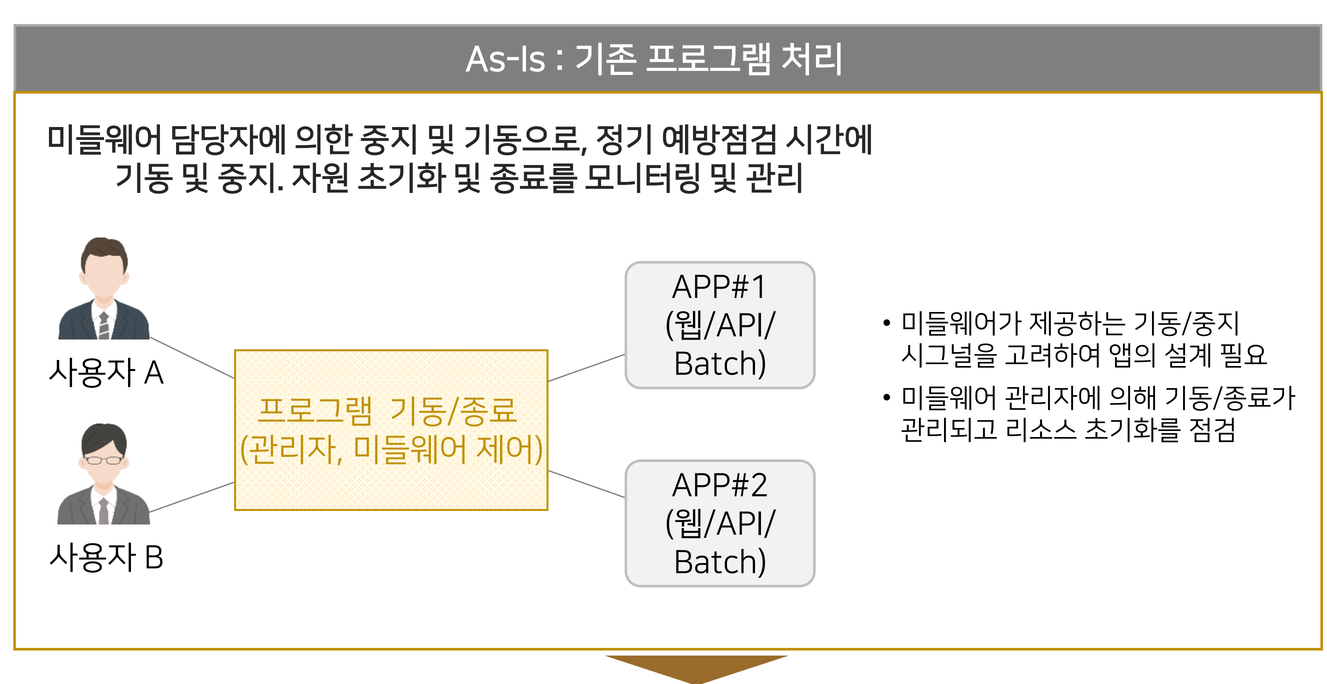
08 클라우드 네이티브 정보시스템 설계 단계

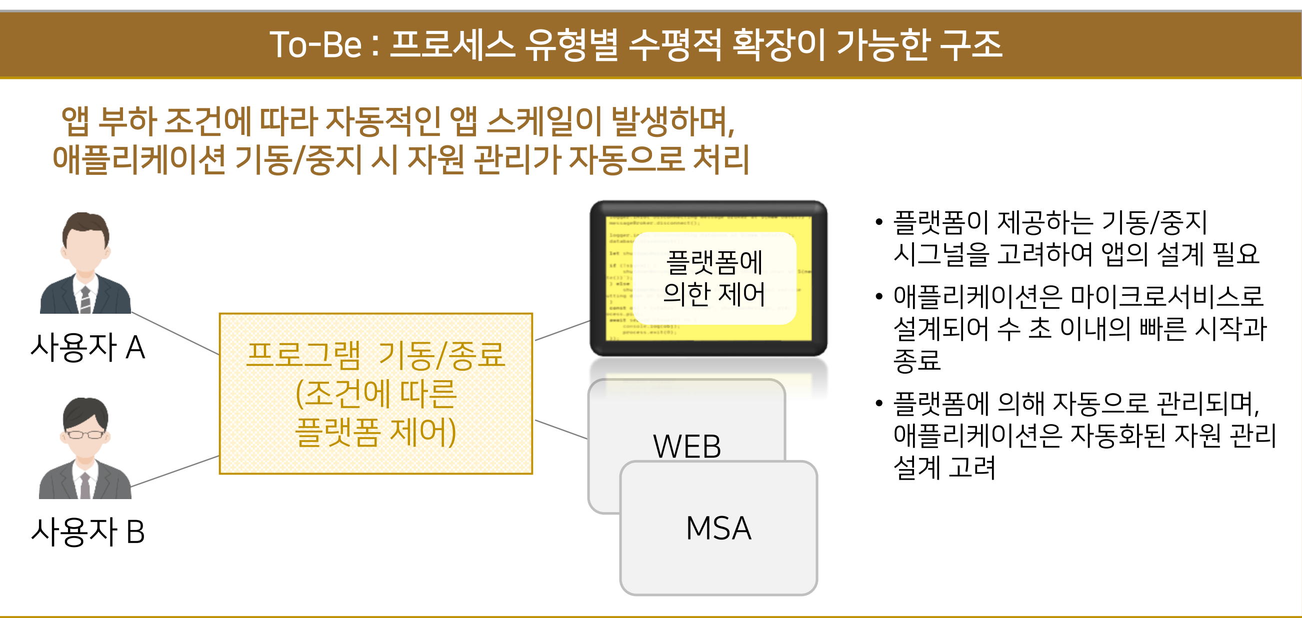
8.3 12가지 요소 기반 개발 원칙

8.3.10 폐기 가능성

- **MSA는 각종 요청에 의해 스케일 업/다운이 빈번히 발생하므로 프로세스는 셧다운 신호를 받았을 때 정상 종료해야 한다. 빠른 시작(Start-up)을 통해 릴리즈 작업과 확장 작업이 민첩하게 이뤄지고, 정상종료 (Graceful Shutdown)에 의해 관련된 모든 리소스가 올바르게 종료됨으로써 안정성을 확보할 수 있다.**

[그림 3-40] 폐기 가능성 원칙





08 클라우드 네이티브 정보시스템 설계 단계

8.3 12가지 요소 기반 개발 원칙

8.3.11 개발/운영 환경 일치

- **가능한 한 동일한 개발/검증/운영 환경을 유지하여 장애 최소화 및 지속적 배포가 가능하게 한다.**
- **개발 환경과 운영 환경의 백엔드 서비스(DB, 메시지큐, 캐시 등)가 상이한 경우, 개발 환경에서 테스트가 완료되었더라도 운영계에서 오류를 일으킬 수 있으므로 개발, 검증, 운영 환경의 일치가 필요하다.**

[그림 3-41] 개발/운영 환경 일치 원칙

As-Is : 개발과 운영환경의 차이

개발과 운영환경의 차이 존재

개발자 A
코드베이스
상이한 환경
긴 배포 간격
개발자 A
개발계
≠
개발자 A
검증계
≠
운영자 B
운영계

- 코드 개발자와 코드 배포자 상이
  - 운영자가 운영계 배포를 담당하므로 코드 개발자와 코드 배포자가 다름
- 긴 배포 간격
  - 개발자가 코드 작성 후 개발계, 검증계, 운영계 환경에 반영하기까지 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 : 개발과 운영환경의 일치

개발, 테스트(검증), 운영 환경을 최대한 동일하게 유지

개발자 A
코드베이스
동일한 환경
짧은 배포 간격
개발자 A
개발계
=
개발자 A
검증계
=
개발자 B
운영계

- 코드 개발자와 코드 배포자 일치
  - 운영자 아니라 코드 개발자가 직접 배포
- 짧은 배포 간격
  - 개발자가 코드 작성 후 개발계, 검증계, 운영계 환경에 반영하기까지 수 분에서 수 시간 소요
- 개발 환경과 운영 환경의 차이를 최소화하여 운영 환경의 예기치 않은 오류 및 다운타임의 위험성 감소

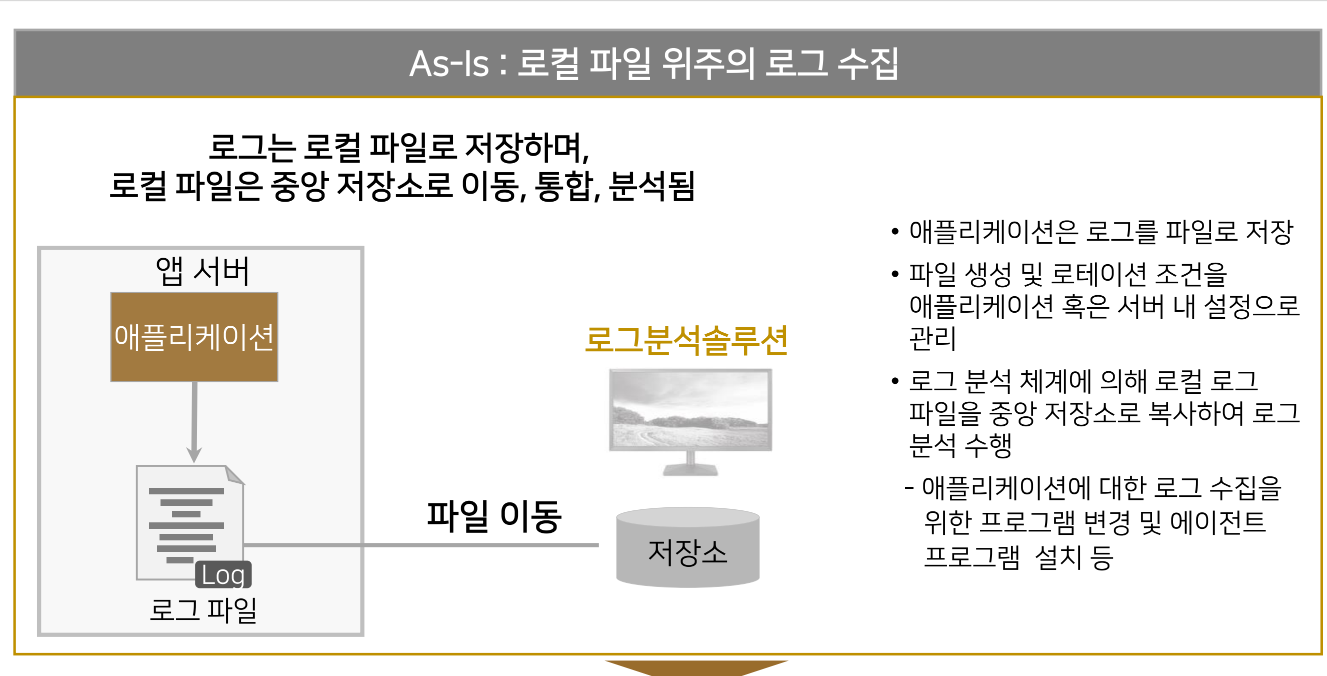
08 클라우드 네이티브 정보시스템 설계 단계

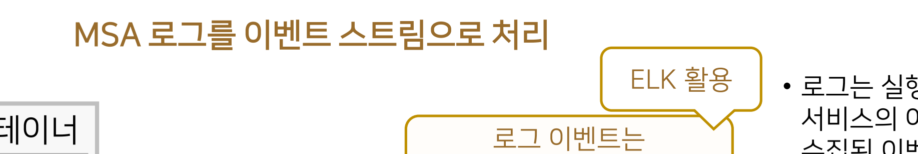
8.3 12가지 요소 기반 개발 원칙

8.3.12 로그

- **애플리케이션 로그는 파일이 아닌 로그 스트림 형태로 표준 출력하며, 플랫폼에 의해 로그 스트림이 통합되고 관리된다. 즉, 애플리케이션은 로그를 작성하거나 로컬 로그 파일을 관리하지 않는다.**
- **이벤트 로그 스트림으로 관리함으로써 애플리케이션의 배포와 플랫폼 간 이식이 용이하다.**

[그림 3-42] 로그 원칙



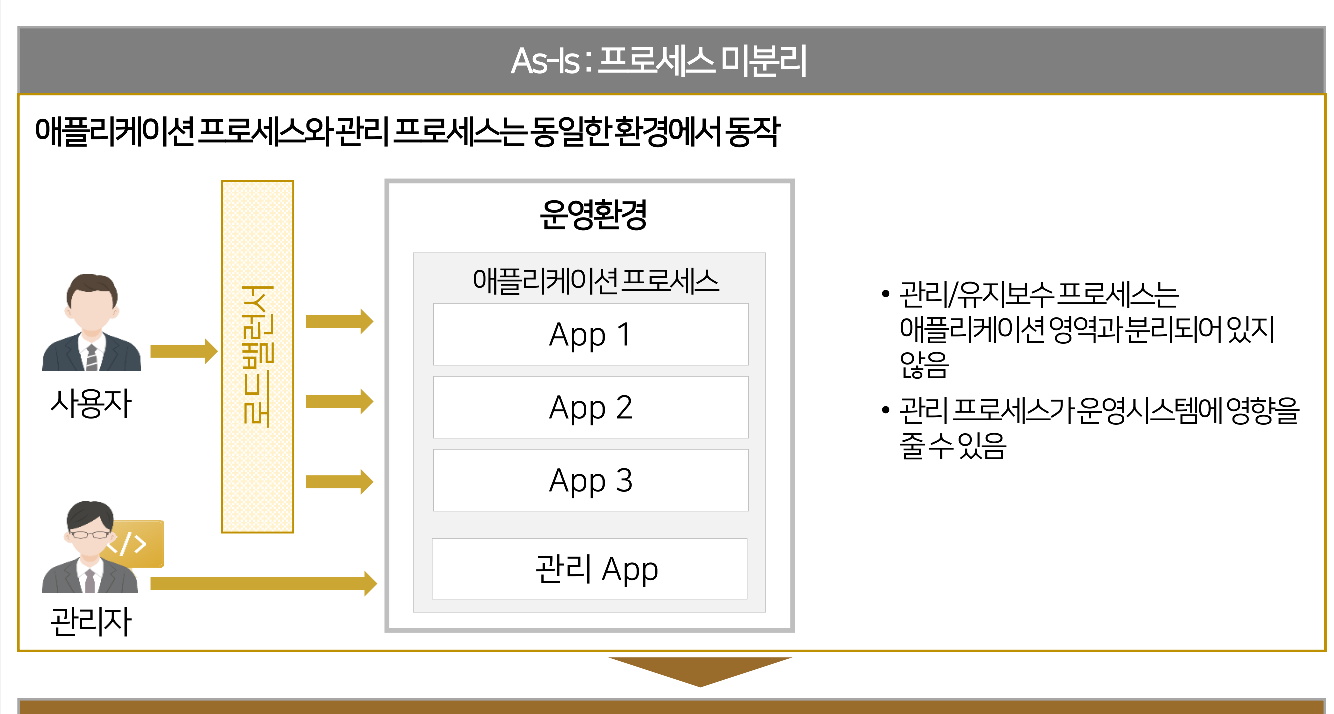


8.3 12가지 요소 기반 개발 원칙

8.3.13 관리 프로세스

- **관리/유지보수 프로세스를 일회성 프로세스로 애플리케이션 프로세스와 분리되어 실행되지만, 애플리케이션과 동일한 빌드/릴리즈/배포 사이클로 실행된다.**

[그림 3-43] 관리 프로세스 원칙



클라우드 네이티브 정보시스템 구축을 위한

개발자 안내서

09

클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발· 테스트·운영 환경 구성
9.2 스프링 부트 기반 마이크로서비스 제작
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축
9.4 컨테이너 기반 마이크로서비스 빌드·배포

클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발·테스트·운영 환경 구성

9.1.1 개발·테스트·운영 환경 구성 원칙

- **설계 단계에서 식별한 마이크로서비스와 아키텍처를 구현하기 위한 개발·테스트·운영 환경을 구성한다.**
- **12가지 요소의 개발/운영 환경 일치 원칙에 따라 개발·테스트·운영 환경을 최대한 동일하게 구성하여 클라우드 네이티브 애플리케이션 서비스 제공 시 발생할 수 있는 오류 및 장애를 감소시키고, 시스템의 안정성을 확보하도록 한다.**
- **마이크로서비스 개발을 지원하고, 개발자의 로컬 PC 개발환경에서도 분산처리를 구축하기 위해 스프링 클라우드를 적용하여 환경을 구성한다.**

[그림 9-1] 스프링 클라우드 기반의 개발·테스트·운영 환경 구성

개발자 A
개발자 B
개발자 C
깃 저장소
빌드
릴리즈
배포
개발 환경
테스트 환경
운영 환경
개발, 테스트, 운영 환경의 일치를 통해 지속적인 배포 가능

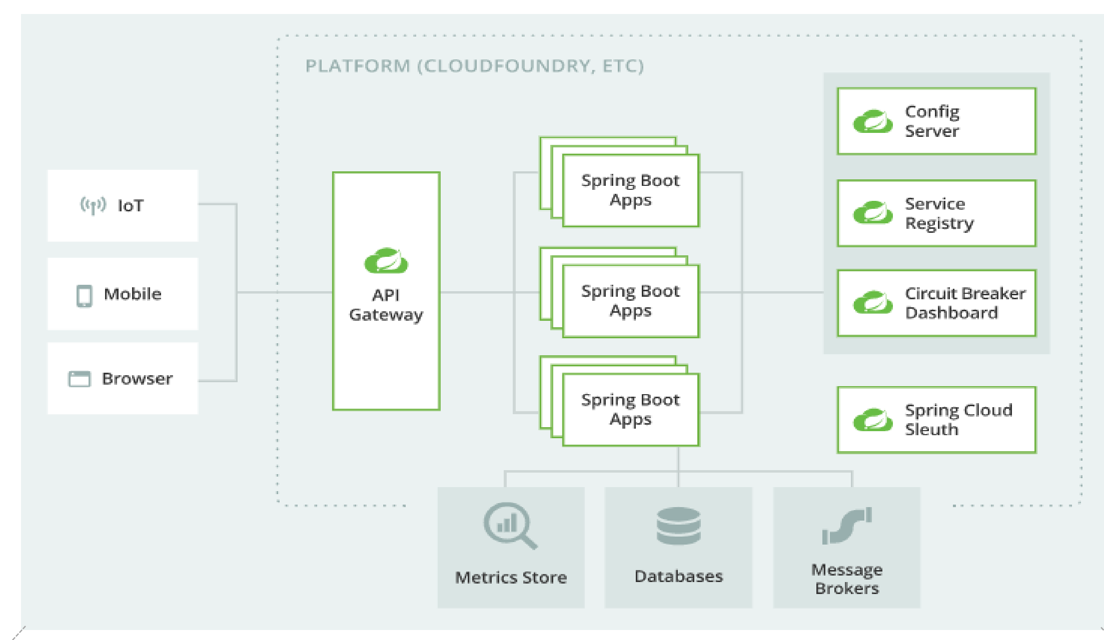
클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발·테스트·운영 환경 구성

9.1.2 스프링 클라우드 환경 구성

- **스프링 클라우드는 마이크로서비스의 개발, 배포, 운영에 필요한 아키텍처를 쉽게 구성할 수 있고, MSA를 지원하는 스프링 부트 기반 프레임워크이다.**
- **스프링 클라우드는 클라우드 네이티브 패턴 중 일부에 집중하여 다양한 라이브러리를 제공하는 스프링 부트를 확장하며, 분산 시스템에서 일반적인 패턴을 빠르게 조합할 수 있도록 해 준다.**

[그림 9-2] 스프링 클라우드의 구성 및 주요 특징



주요 특징

- 분산 및 버전 설정 관리(Distributed/versioned configuration)
- 서비스 등록 및 조회(Service registration and discovery)
- 라우팅(Routing)
- 서비스 대 서비스 호출(Service-to-service calls)
- 서비스 분산 로딩(Load balancing)
- 서비스 간 호출 분리, 서킷 브레이커(Circuit breakers)
- 글로벌 락(Global locks)
- 리더십 선출 및 클러스터 상태(Leadership election and cluster state)
- 분산 메시징(Distributed messaging)

[출처 : spring.io]

클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발·테스트·운영 환경 구성

9.1.2 스프링 클라우드 환경 구성

[표 9-1] 스프링 클라우드의 컴포넌트

서비스	설명	컴포넌트
API 게이트웨이 (API Gateway)	• 마이크로서비스에 대한 API 관리 및 모니터링 서비스	Zuul(Spring Cloud Netflix)
설정 (Config)	• 별도의 통합된 설정 관리 서비스 제공을 통해 독립적 서비스 제공	Spring Config
서비스 디스커버리 (Service Discovery)	• 서비스에 대한 물리적 위치 정보 대신 논리적 서비스 위치 정보 제공	Eureka(Spring Cloud Netflix)
서킷 브레이커 (Circuit Breaker)	• 서비스 간 호출 시 문제가 있는 서비스에 대한 차단 지원	Hystrix(Spring Cloud Netflix)
이벤트 버스 (Event Bus)	• 분산 메시징 지원을 위한 서비스 연계 지원	Spring Cloud Bus (AMQP & RabbitMQ)
클라이언트 로드밸런싱 (Client Load Balancing)	• 서비스 호출 시 분산 형태로 호출할 수 있는 클라이언트 적용 서비스 라이브러리	Ribbon(Spring Cloud Netflix)
서비스 라우터 (Service Router)	• 서비스 호출 시 서비스 라우팅을 통해 실제 서비스에 위치 정보 제공	Zuul(Spring Cloud Netflix)
클러스터 (Cluster)	• 서비스 레벨의 클러스터를 지원하기 위한 서비스 제공	Spring Cloud Cluster
보안 (Security)	• 로드밸런스 환경에서의 OAuth2 인증 지원 서비스	Spring Cloud Security
폴리글랏(Polyglot) 지원	• non-JVM 프로그래밍 언어 지원을 위한 서비스	Spring Cloud Sidecar
쿠버네티스 지원	• 스프링 클라우드 애플리케이션을 위한 쿠버네티스 • 디스커버리와 ConfigMaps 지원 서비스	Spring Cloud Kubernetes

클라우드 네이티브 정보시스템 구현·이행 단계

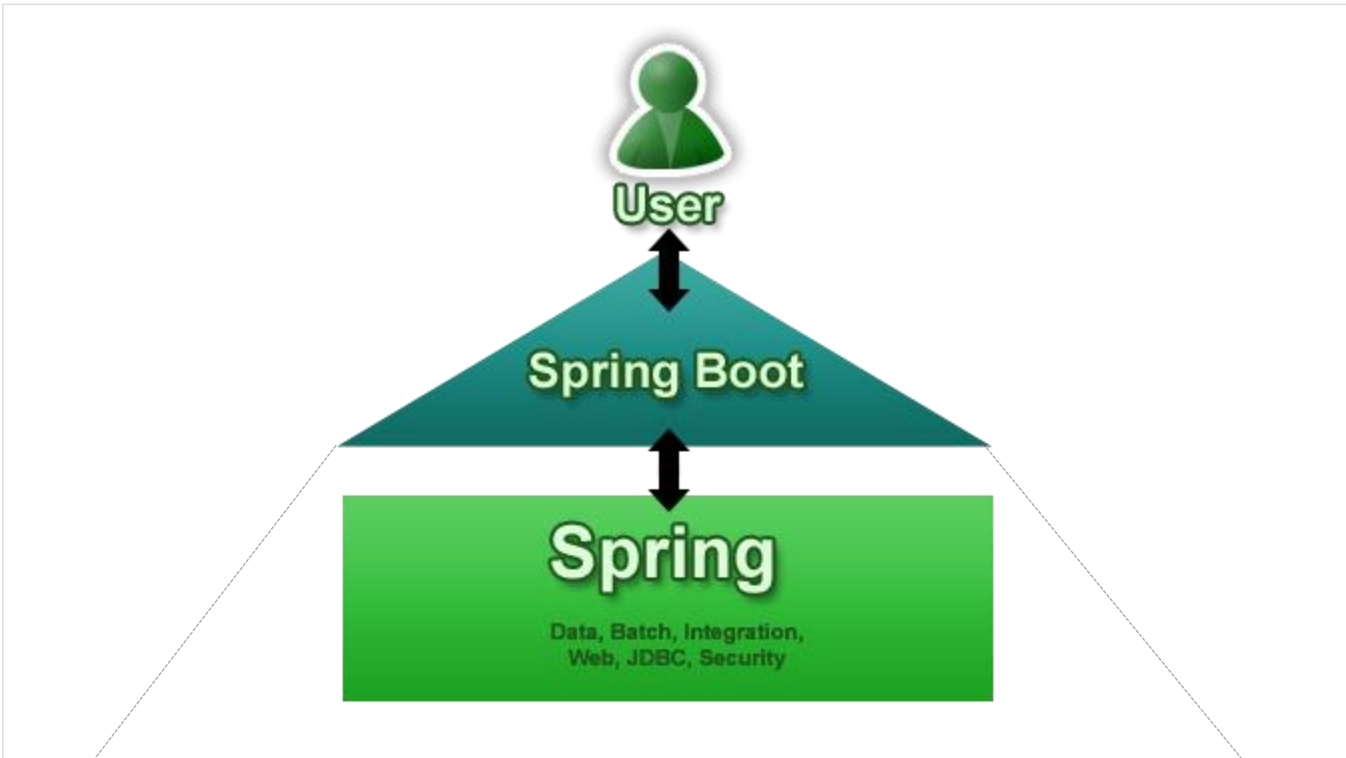
9.1 개발·테스트·운영 환경 구성

9.1.3 스프링 부트 환경 구성

9.1.3.1 스프링 부트 개요

- **스프링 부트는 자바를 기반으로 한 웹 애플리케이션 프레임워크이다. 기존의 스프링 프레임워크 기반 프로젝트를 복잡한 설정 없이 쉽고 빠르게 설정하도록 만들어 주는 라이브러리이다.**
- **스프링 부트는 자동 설정 기능을 제공하고, 개발에 필요한 모든 의존성(Dependency)을 호환되는 버전으로 관리한다. 의존성을 간단하게 설정할 수 있도록 스프링 부트 스타터를 제공한다.**

[그림 9-3] 스프링 부트의 구조 및 특징



특징

- 단독 실행이 가능한(Stand-alone) 스프링 애플리케이션 생성
- 내장형 Tomcat, Jetty, undertow 지원(WAR 파일을 직접 배포할 필요가 없음)
- 빌드 구성을 단순화하기 위해 독자적인 '스타터(Starter)' 의존성을 제공함
- 필요시 스프링과 3rd 파티 라이브러리를 자동으로 구성
- 통계(Metrics), 헬스체크(Health check), 외부 설정과 같은 프로덕션 준비 기능 제공
- 코드 생성 및 XML 구성 불필요

[출처 : spring.io]

클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발·테스트·운영 환경 구성

9.1.3 스프링 부트 환경 구성

9.1.3.2 스프링 부트의 주요 기능

[표 9-2] 스프링 부트 구성 요소의 주요 기능

구분	주요 기능	
스프링 부트 스타터 (Spring Boot Starters)	• Web MVC, JDBC, ORM과 같은 스프링 프로젝트 모듈 • 스프링 애플리케이션에서 스타터의 추가만으로 필요한 라이브러리가 포함되도록 함 • 예를 들어 Restful 서비스 발행을 목적으로 Spring Web MVC를 사용하기 위해서 Spring-boot-Starter-web 의존성을 추가하기만 하면 됨 • 스타터는 의존성의 개수를 줄여줄 뿐만 아니라 빌드하고자 하는 특별한 기능을 추가해 줌	• Spring-boot-starter-web은 아래 의존성을 자동으로 추가함 - spring-web-*.jar - spring-webmvc-*.jar - tomcat-*.jar - Jackson-databind-*.jar
자동 설정 (Automatic Configuration)	• 애플리케이션 기능에 대한 자동 설정 제공 • 오토스케일링은 classpath를 이용 • 예를 들어, JPA Starter(spring-boot-starter-data-jpa)를 의존성에 추가하면 JPA와 관련된 설정을 자동으로 시도함 • 먼저 classpath에서 설정을 하고, 설정할 수 없으면 애플리케이션의 설정 정보에서 찾음	• 의존성을 추가하지 않으면 스프링 부트는 자동으로 설정할 수 없음 • 의존성 관리 도구(Dependency Management Tool)가 필요한 종속성 제시 • Maven, Gradle, Ant/Ivy 지원
스프링 부트 액추에이터 (Spring Boot Actuator)	• 스프링 애플리케이션을 모니터링하기 위한 프로덕션 레벨의 특징 제공 • http의 엔드포인트나 JMX로 모니터링 가능	• 애플리케이션 콘텍스트의 상세한 설정 • 자동으로 설정 관리 • 모든 환경 변수, 시스템 속성, 설정 속성, 코맨드 라인 매개변수 • 메모리 사용량, GC(Garbage Collection), Web Request, 데이터 리소스 사용량 등의 다양한 통계 정보 • 최근 처리된 http 요청의 추적 • 쓰레드의 상태 정보

[출처 : 스프링 부트 강의 - 1-1강 Spring Boot 개요, https://www.youtube.com/watch?v=MFT2s6ijTws]

9.1 개발·테스트·운영 환경 구성

9.1.3 스프링 부트 환경 구성

9.1.3.3 스프링 부트의 애플리케이션 스타터

- **스프링 부트 스타터는 애플리케이션에 포함할수 있는 편리한 의존성 관리의 집합이다. 샘플 코드와 복사-붙여넣기의 의존성 관리를 거치지 않고도 필요한 모든 스프링 및 관련 기술을 한 번에 관리할 수 있다. 예를 들어, 데이터베이스 액세스를 위한 스프링 및 JPA를 사용하려면 "spring-boot-starter-data-jpa"프로젝트를 의존성에 포함시켜 사용할 수 있다.**
- **스타터에는 프로젝트를 신속하게 시작하고 실행하는 데 필요한 많은 의존성이 포함되어 있으며 일관되게 지원·관리되는 의존성 세트를 제공한다.**
- **스프링 부트는 애플리케이션 스타터, 프로덕션 스타터 및 테크니컬 스타터를 제공한다.**

[표 9-3] 스프링 부트 애플리케이션 스타터

이름	설명
spring-boot-starter	• 자동 구성 지원, 로깅 및 YAML을 포함한 핵심 스타터
spring-boot-starter-activemq	• Apache ActiveMQ를 사용한 JMS 메시징 스타터
spring-boot-starter-amqp	• Spring AMQP 및 RabbitMQ 사용을 위한 스타터
spring-boot-starter-aop	• Spring AOP 및 AspectJ를 이용한 Aspect 지향 프로그래밍 스타터
spring-boot-starter-artemis	• Apache Artemis를 사용한 JMS 메시징 스타터
spring-boot-starter-batch	• 스프링 배치 사용을 위한 스타터
spring-boot-starter-cache	• Spring Framework의 캐싱 지원 사용을 위한 스타터
spring-boot-starter-data-cassandra	• Cassandra 분산 데이터베이스 및 Spring Data Cassandra 사용을 위한 스타터
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 사용하기 위한 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	• Hibernate와 함께 Spring Data JPA를 사용하기 위한 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

9.1 개발·테스트·운영 환경 구성

9.1.3 스프링 부트 환경 구성

9.1.3.3 스프링 부트의 애플리케이션 스타터

[표 9-4] 스프링 부트 애플리케이션 스타터

이름	설명
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 반응형 및 Lettuce 클라이언트와 함께 Redis 키-값 데이터 저장소를 사용하기 위한 스타터
spring-boot-starter-data-rest	• Spring Data REST를 사용하여 REST를 통해 Spring Data 저장소를 노출하기 위한 스타터
spring-boot-starter-data-solr	• Spring Data Solr와 함께 Apache Solr 검색 플랫폼을 사용하기 위한 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 HATEOAS로 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 JDBC를 사용하기 위한 스타터
spring-boot-starter-jersey	• JAX-RS 및 Jersey를 사용하여 RESTful 웹 애플리케이션을 빌드하기 위한 스타터(대안 spring-boot-starter-web)
spring-boot-starter-jooq	• jOOQ를 사용하여 SQL 데이터베이스에 액세스하기 위한 스타터(spring-boot-starter-data-jpa 또는 이에 대한 대안 spring- boot-starter-jdbc)
spring-boot-starter-JSON	• JSON을 읽고 쓰는 스타터
spring-boot-starter-jta-atomikos	• Atomikos를 사용한 JTA 트랜잭션 스타터
spring-boot-starter-jta-bitronix	• Bitronix를 사용한 JTA 트랜잭션 스타터(2.3.0 부터 사용되지 않음)
spring-boot-starter-mail	• Java Mail 및 스프링 프레임워크의 이메일 전송 지원을 위한 스타터
spring-boot-starter-mustache	• 콧수염보기를 사용하여 웹 애플리케이션을 빌드하기 위한 스타터
spring-boot-starter-oauth2-client	• Spring Security의 OAuth2/OpenID Connect 클라이언트 기능을 사용하기 위한 스타터
spring-boot-starter-oauth2-resource-server	• Spring Security의 OAuth2 리소스 서버 기능을 사용하기 위한 스타터
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 Mockito를 포함한 라이브러리로 스프링 부트 애플리케이션을 테스트하기 위한 스타터

클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발·테스트·운영 환경 구성

9.1.3 스프링 부트 환경 구성

9.1.3.3 스프링 부트의 애플리케이션 스타터

[표 9-5] 스프링 부트 애플리케이션 스타터

이름	설명
spring-boot-starter-thymeleaf	• Thymeleaf 보기를 사용하여 MVC 웹 애플리케이션 빌드를 위한 스타터
spring-boot-starter-validation	• Hibernate Validator와 함께 Java Bean Validation을 사용하기 위한 스타터
spring-boot-starter-web	• Spring MVC를 사용하는 RESTful 애플리케이션을 포함한 웹 구축을 위한 스타터(Tomcat을 기본 내장 컨테이너로 사용)
spring-boot-starter-web-services	• 스프링 웹 서비스 사용을 위한 스타터
spring-boot-starter-webflux	• 스프링 프레임워크의 Reactive Web 지원을 사용하여 WebFlux 애플리케이션 구축을 위한 스타터
spring-boot-starter-websocket	• 스프링 프레임워크의 WebSocket 지원을 사용하여 WebSocket 애플리케이션 구축을 위한 스타터

[표 9-6] 스프링 부트 프로덕션 스타터

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

[표 9-7] 스프링 부트 프로덕션 스타터

이름	설명
spring-boot-starter-jetty	• 내장 서블릿 컨테이너로 Jetty를 사용하기 위한 스타터(대안 spring-boot-starter-tomcat)
spring-boot-starter-log4j2	• 로깅을 위해 Log4j2를 사용하기 위한 스타터(대안 spring-boot-starter-logging)
spring-boot-starter-logging	• Logback을 이용한 기본 로깅 스타터
spring-boot-starter-reactor-netty	• 임베디드 Reactive HTTP 서버로 Reactor Netty를 사용하기 위한 스타터
spring-boot-starter-tomcat	• 임베디드 서블릿 컨테이너로 Tomcat을 사용하기 위한 스타터에 의해 사용되는 기본 서블릿 컨테이너 스타터(spring-boot-starter-web)
spring-boot-starter-undertow	• Undertow를 임베디드 서블릿 컨테이너로 사용하기 위한 스타터(대안 spring-boot-starter-tomcat)

클라우드 네이티브 정보시스템 구현·이행 단계

9.1 개발·테스트·운영 환경 구성

9.1.4 마이크로서비스 구현을 위한 오픈소스 설치

- **스프링 클라우드와 스프링 부트를 활용하여 개발·테스트·운영 환경을 구성하기 위해 서비스 개발과 관련된 오픈소스 소프트웨어를 설치한다.**
- **간단한 예제 테스트를 위해 설치한 오픈소스 소프트웨어는 다음과 같다.**

[표 9-8] 개발 · 테스트 · 운영 환경 구성을 위한 오픈소스 설치 내역

SW 명	버전	비고
OpenJDK	1.8	
eGovframework	4.0.0	• 전자정부 표준프레임워크
spring-boot-starter	2.2.6.RELEASE	• 스프링 부트
spring-boot-starter-test	2.2.6.RELEASE	• 스프링 부트
spring-boot-test	2.2.6.RELEASE	• 스프링 부트
spring-boot-starter-web	2.2.6.RELEASE	• 스프링 부트
Spring Framework	5.2.5 RELEASE	• 스프링 프레임워크
Spring swagger	2.9.2	

09 클라우드 네이티브 정보시스템 구현·이행 단계

9.2 스프링 부트 기반 마이크로서비스 개발

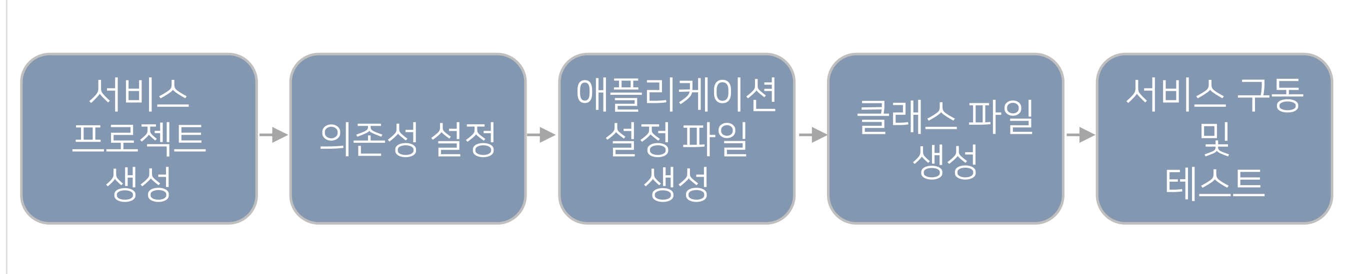
9.2.1 개요

- **클라우드 네이티브 정보시스템 구현 과정을 소개하기 위해 '표준프레임워크 MSA 적용 개발 가이드 V 1.2.0'를 참고하여 마이크로서비스 개발 과정을 설명하고자 한다.**
- **예제 마이크로서비스는 화면의 디스플레이를 담당하는 카탈로그 서비스와 실제 정보는 담고 있는 고객 서비스로 구성된다.**

[그림 9-4] 예제 마이크로서비스 구성도

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

[그림 9-5] 예제 마이크로서비스 제작 과정



9.2 스프링 부트 기반 마이크로서비스 개발

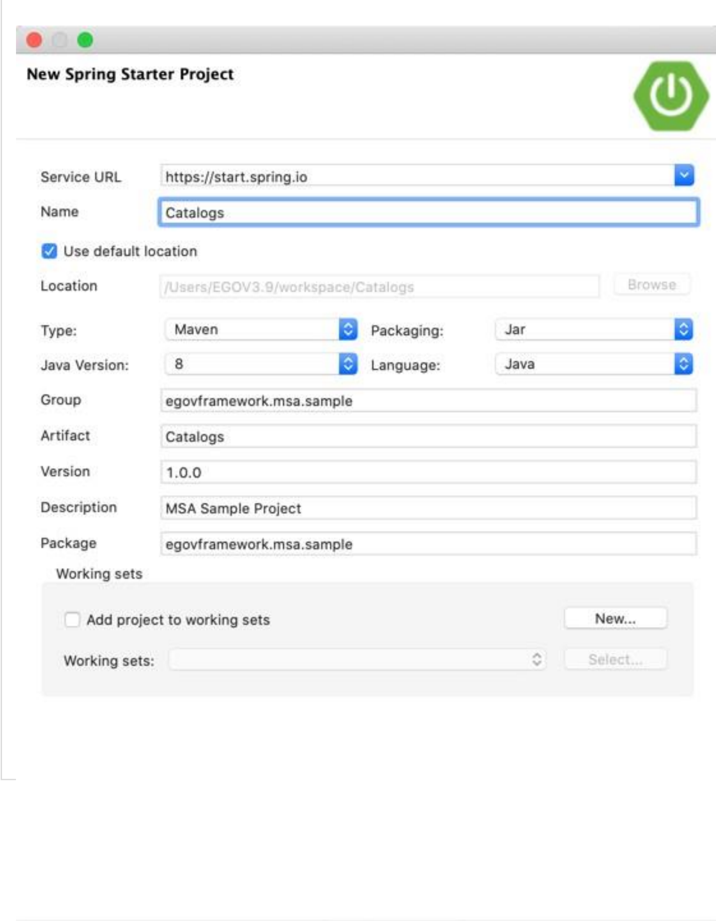
9.2.2 카탈로그 서비스 개발

9.2.2.1 카탈로그 서비스 프로젝트 생성

- **화면 레이어를 담당하는 카탈로그 서비스를 전자정부 표준프레임워크 개발 환경을 활용하여 생성한다. (표준프레임워크 V3.10 기준)**

[그림 9-6] 카탈로그 서비스 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
- 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
  (이 안내서에서는 의존 관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.2 카탈로그 서비스 개발

9.2.2.1 카탈로그 서비스 프로젝트 생성

- **카탈로그 서비스의 디렉토리 구조를 다음과 같이 설정한다.**

[표 9-9] 카탈로그 프로젝트 디렉토리 구조 설정

SW 명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	리소스 파일	스프링 부트 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	

9.2.2.2 카탈로그 서비스의 의존성 설정

- **카탈로그 서비스의 의존성을 Pom.xml 파일에 다음과 같이 설정한다.**

[그림9-7] 카탈로그 서비스의 의존성 추가 – 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>

    <properties>
        <java.version>1.8</java.version>
        <org.egovframe.rte.version>4.0.0</org.egovframe.rte.version>
        <spring.cloud.version>2.2.5.RELEASE</spring.cloud.version>
    </properties>
```

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.2 카탈로그 서비스 개발

9.2.2.2 카탈로그 서비스의 의존성 설정

[그림 9-8] 카탈로그 서비스의 의존성 추가

```
<repositories>
    <repository>
        <id>mvn2s</id>
        <url>https://repo1.maven.org/maven2/</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>egovframe</id>
        <url>http://maven.egovframe.go.kr/maven/</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>

<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

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.2 카탈로그 서비스 개발

9.2.2.3 애플리케이션 설정 파일 생성

- 스프링 부트 및 프레임워크의 의존성을 추가한 후 애플리케이션 설정을 위한 application.yml 파일을 생성한다. application.yml 파일은 /src/main/resources 디렉토리에 위치하며, yml(yaml) 파일 대신 properties 형태의 파일을 사용할 수도 있다.
- Application.yml 파일 소스에 카탈로그 서비스의 이름과 contextPath 및 접속 포트를 설정한다.

[그림 9-9] application.yml 파일 소스 내용

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

9.2.2.4 각 클래스 파일 작성

- MSA 애플리케이션의 구조 파일들을 모두 작성한 후, 실제 로직을 담고 있는 각각의 클래스 파일들을 작성한다.

[그림 9-10] CatalogsApplication.java 파일 작성

```
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);
    }

}
```

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.2 카탈로그 서비스 개발

9.2.2.4 각 클래스 파일 작성

[그림 9-11] CatalogsController.java 파일 작성

```
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);
    }

}
```

[그림 9-12] CustomerApiService.java 파일 작성

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

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

[그림 9-13] CustomerApiServiceImpl.java 파일 작성

```
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;
    }

}
```

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.2 카탈로그 서비스 개발

9.2.2.5 카탈로그 서비스 구동 테스트

- **카탈로그의 CatalogsApplication.java 파일을 자바 애플리케이션으로 실행하면, 아래와 같이 스프링 부트를 통하여 임베디드 톰캣(Embedded Tomcat)으로 구동되는 것을 확인할 수 있다.**

[그림 9-14] 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
```

- **웹 브라우저를 실행하고 URL : http://localhost:8081/catalogs/customerinfo/1234 customerid가 1234인 내용을 확인할 수 있다.**

[그림 9-15] 카탈로그 서비스 테스트 결과 확인

localhost:8081/catalogs/custom × +

localhost:8081/catalogs/customerinfo/1234

[Customer id = 1234 at 1594991787610 1234]

클라우드 네이티브 정보시스템 구현·이행 단계

9.2 스프링 부트 기반 마이크로서비스 개발

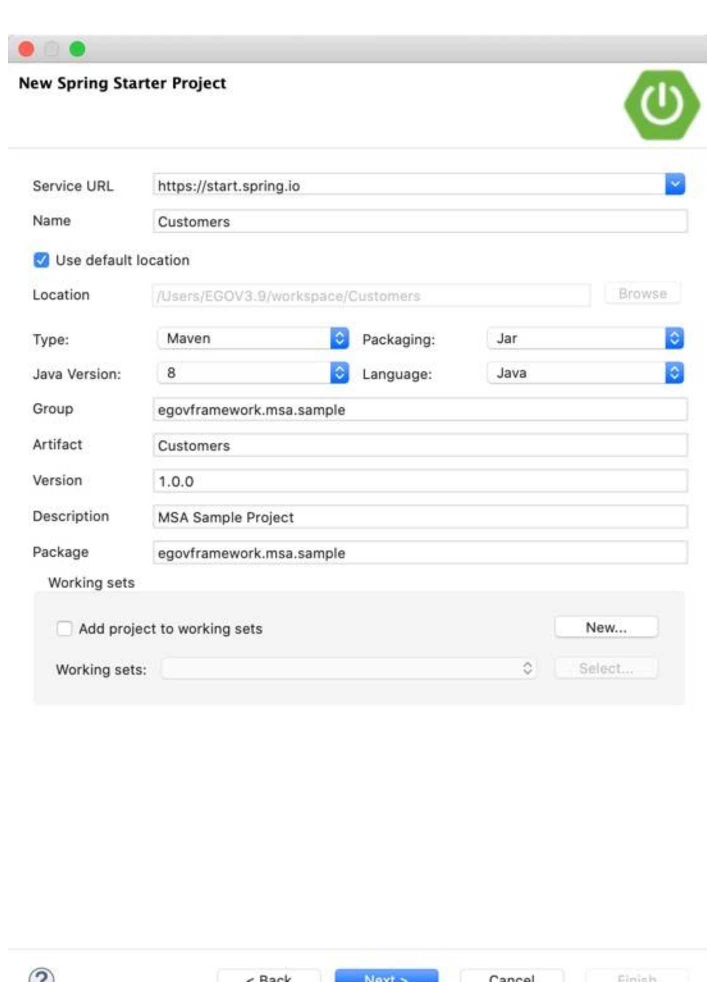
9.2.3 고객 서비스 개발

9.2.3.1 고객 서비스 프로젝트 생성

- **고객 서비스를 전자정부 표준프레임워크 개발환경을 활용하여 생성한다. (표준프레임워크 V3.10 기준)**

[그림 9-16] 고객 서비스 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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 : Customers**
- Version : 1.0.0
- Description : MSA Sample Project
  Group Id : egovframework.msa.sample

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
  - 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
   (이 안내서에서는 의존관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.3 고객 서비스 개발

9.2.3.1 고객 서비스 프로젝트 생성

- **고객 서비스의 디렉토리 구조를 다음과 같이 설정한다.**

[표 9-10] 고객 프로젝트 디렉토리 구조 설정

SW 명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	Resource 파일	스프링 부트 설정 파일
CustomerApplication.java	egovframework.msa.sample	클래스 파일	애플리케이션 구동 파일
CustomerController.java	egovframework.msa.sample.controller	컨트롤러 클래스 파일	Restful API Controller

9.2.3.2 고객 서비스의 의존성 설정

- **고객 서비스의 의존성을 Pom.xml 파일에 다음과 같이 설정한다.**

[그림 9-17] 고객서비스의 의존성 추가–Pom.xml

```
<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

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.3 고객 서비스 개발

9.2.3.3 애플리케이션 설정 파일 생성

- 스프링 부트 및 프레임워크의 의존성을 추가한 후, 애플리케이션 설정을 위한 application.yml 파일을 생성한다. application.yml 파일은 /src/main/resources 디렉토리에 위치하며, yml(yaml) 파일 대신 properties 형태의 파일을 사용할 수도 있다.
- Application.yml 파일 소스에 고객 서비스의 이름과 contextPath 및 접속 포트를 설정한다.

[그림 9-18] application.yml 파일 소스 내용

```
server: port:
  8082
spring:
  application:
    name: customer
```

9.2.3.4 각 클래스 파일 작성

- MSA 애플리케이션의 구조 파일들을 모두 작성한 후, 실제 로직을 담고 있는 각각의 클래스 파일들을 작성한다.

[그림 9-19] CustomerApplication.java 파일 작성

```
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 CustomersApplication {

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

}
```

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.3 고객 서비스 개발

9.2.3.4 각 클래스 파일 작성

[그림 9-20] CustomerController.java 파일 작성

```
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);
    }

}
```

클라우드 네이티브 정보시스템 구현·이행 단계

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.3 고객 서비스 개발

9.2.3.5 고객 서비스 구동 테스트

- **고객의 CustomerApplication.java 파일을 자바 애플리케이션으로 실행하면, 아래와 같이 스프링 부트를 통하여 임베디드 톰캣으로 구동되는 것을 확인할 수 있다.**

[그림 9-21] 고객 서비스 구동 및 테스트

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

2020-10-19 16:16:04.188    INFO 50318 --- [           main] e.msa.sample.CustomersApplication
: Starting CustomersApplication on jeonghunhuiui-iMac.local with PID 50318
(/Users/EGOV3.9/workspace/Customers/target/classes started by hhjeong in
/Users/EGOV3.9/workspace/Customers)
2020-10-19 16:16:04.190    INFO 50318 --- [           main] e.msa.sample.CustomersApplication
: No active profile set, falling back to default profiles: default
2020-10-19 16:16:04.865    INFO 50318 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat initialized with port(s): 8082 (http)
2020-10-19 16:16:04.873    INFO 50318 --- [           main] o.apache.catalina.core.StandardService
: Starting service [Tomcat]
2020-10-19 16:16:04.873    INFO 50318 --- [           main] org.apache.catalina.core.StandardEngine
: Starting Servlet engine: [Apache Tomcat/9.0.38]
2020-10-19 16:16:04.937    INFO 50318 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring embedded WebApplicationContext
2020-10-19 16:16:04.937    INFO 50318 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 713 ms
2020-10-19 16:16:05.077    INFO 50318 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor
: Initializing ExecutorService 'applicationTaskExecutor'
2020-10-19 16:16:05.205    INFO 50318 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat started on port(s): 8082 (http) with context path ''
2020-10-19 16:16:05.213    INFO 50318 --- [           main] e.msa.sample.CustomersApplication
: Started CustomersApplication in 1.532 seconds (JVM running for 1.782)
2020-10-19 16:16:10.795    INFO 50318 --- [nio-8082-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-10-19 16:16:10.796    INFO 50318 --- [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet
: Initializing Servlet 'dispatcherServlet'
2020-10-19 16:16:10.799    INFO 50318 --- [nio-8082-exec-1] o.s.web.servlet.DispatcherServlet
: Completed initialization in 3 ms 202x-0x-xx 22:49:26.150    INFO 10020 ---
[nio-8080-exec- 1] o.s.web.servlet.DispatcherServlet       : Initializing Servlet
'dispatcherServlet' 202x-0x-xx 22:49:26.157   INFO 10020 --- [nio-8080-exec-1]
o.s.web.servlet.DispatcherServlet          : Completed initialization in 6 ms
```

- **웹 브라우저를 실행하고 URL : http://localhost:8082/customers/1234 면 Customer id가 1234인 내용을 확인할 수 있다.**

[그림 9-22] 고객 서비스 테스트 결과 확인

localhost:8082/customers/1234 × +

← → C ⓘ localhost:8082/customers/1234

[Customer id = 1234 at 1594994185969]

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.4 카탈로그와 고객 서비스 연동 및 테스트

9.2.4.1 카탈로그에 RestTemplate 적용

- **카탈로그와 고객 서비스의 구현 및 테스트 수행 후 두 서비스를 연동한다. 고객서비스는 서비스가 호출되는 서비스로서 별도의 변경 내용은 없다. 그러나 카탈로그에서는 고객 서비스를 호출하기 위해 별도의 RestTemplate을 적용하여 서비스를 호출하도록 변경한다.**
- **서비스를 호출하여 JSON 형태의 결과를 받기 위해 스프링에서 제공하는 RestTemplate을 카탈로그 서비스에 아래와 같이 변경한다.**

[그림 9-23] CatalogsApplication.java 파일 수정

```
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.web.client.RestTemplate;

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

    @Bean
     public RestTemplate restTemplate() {
         return new RestTemplate();
     }

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

}
```

[그림 9-24] CustomerApiServiceImpl.java 파일 수정

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import egovframework.msa.sample.service.CustomerApiService;
```

클라우드 네이티브 정보시스템 구현·이행 단계

9.2 스프링 부트 기반 마이크로서비스 개발

9.2.4 카탈로그와 고객 서비스 연동 및 테스트

9.2.4.1 카탈로그에 RestTemplate 적용

[그림 9-25] CustomerApiServiceImpl.java 파일 수정

```
@Service
public class CustomerApiServiceImpl implements CustomerApiService {

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public String getCustomerDetail(String customerId) {
        return restTemplate.getForObject("http://localhost:8082/customers/" + customerId,
String.class);
    }

}
```

9.2.4.2 연동 테스트 실행 및 테스트

- **연동을 위한 수정사항을 모두 반영한 후, 두 서비스를 각각 실행하고 테스트를 통하여 정상 작동을 확인한다.**
- **각각의 서비스를 실행하고, 카탈로그 서비스를 호출하여 두 서비스가 연동되었는지 확인한다.**

  - 고객 서비스 구동 → 카탈로그 서비스 구동 → 테스트 URL 호출

  (URL : http://localhost:8081/catalogs/customerinfo/1234)
- **두 서비스를 [illegible]**

[그림 9-26] 연동 테스트 결과

localhost:8081/catalogs/custom × +
localhost:8081/catalogs/customerinfo/1234

[Customer id = 1234 at 1594995434324 [Customer id = 1234 at 1594995434297]]

클라우드 네이티브 정보시스템 구현·이행 단계

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.1 스프링 클라우드 주요 컴포넌트

- **예제로 제작된 화면 서비스와 고객 서비스를 위한 마이크로서비스 아키텍처를 구성하기 위해 스프링 클라우드의 컴포넌트를 활용한다.**
- **제작된 마이크로서비스가 유연하게 동작하기 위하여 스프링 클라우드에 제공하는 넷플릭스 OSS인, 줄(Zuul), 설정(Config), 유레카(Eureka), 리본(Ribbon), 히스트릭스 대시보드(Hystrix Dashboard) 등 에코시스템을 이용한다.**

[그림 9-27] 스프링 클라우드 기반 에코시스템 아키텍처

클라이언트

설정 서비스
유레카

폴리글랏 지원
사이드카

API
게이트웨이
줄

화면 서비스
리본
히스트릭스

고객 서비스
리본
히스트릭스

서비스
디스커버리
유레카

로그 분석
ELK 스택

통계 및
모니터링
프로메테우스
그라파나

분산 추적
쓸루스
집킨

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

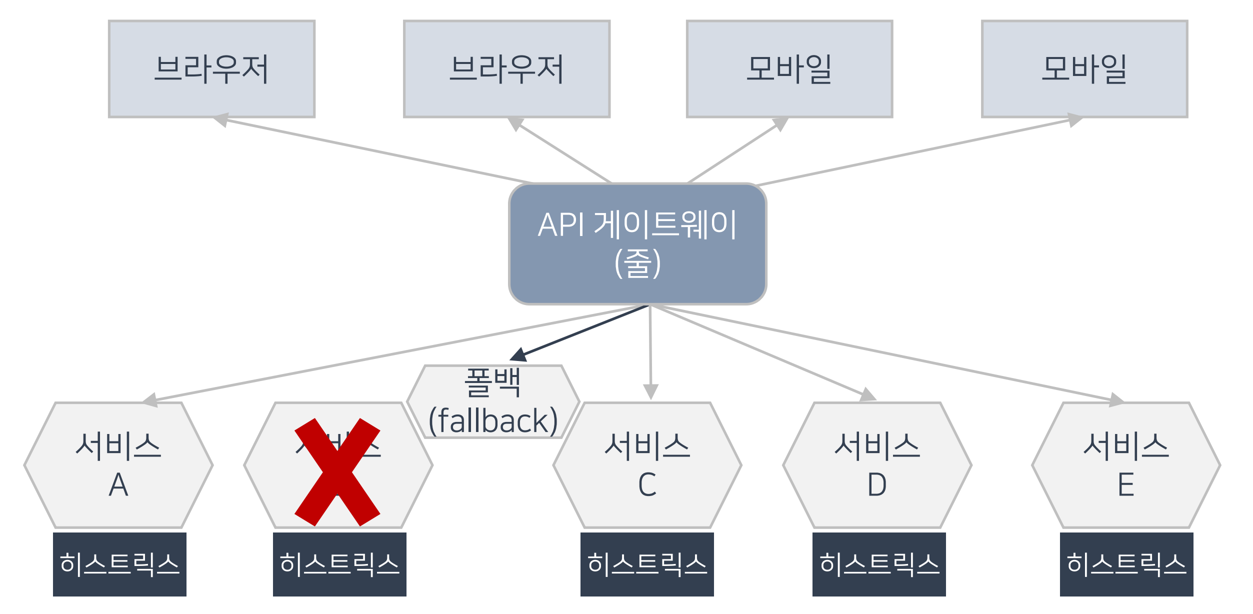
9.3.2 서킷 브레이커 – 히스트릭스(Hystrix)

9.3.2.1 역할 및 주요 기능

- **히스트릭스는 분산 환경을 위한 장애 및 지연 내성(Latency and Fault Tolerance)을 갖도록 도와주는 서킷 브레이커 라이브러리이다.**
- **기존의 모놀리식 아키텍처에서는 고려되지 않던 모듈 간 또는 메서드 간 호출 실패가 마이크로서비스 아키텍처에서는 발생할 수 있으므로 이를 미연에 방지할 수 있는 서킷 브레이커가 필수적이다.**

[그림 9-28] 히스트릭스의 역할 및 주요 기능

히스트릭스는 서비스 사이에서 서킷 브레이커 역할을 수행(서비스 장애 시 등록된 폴백 서비스 수행)



구분	주요 기능
장애 및 지연 내성	• 분산 환경에서 한 개의 서비스가 실패하는 경우 해당 서비스의 실패로 의존성이 있는 타 서비스까지 장애가 전파될 수 있음 • 외부에서 호출하는 서비스를 히스트릭스로 래핑(Wrapping)하면 실패가 전파되는 것을 폴백(Fallbacks)을 활용하여 미연에 방지하고 빠르게 복구할 수 있도록 도와줌 • 각 서비스의 히스트릭스(HystrixCommand)는 서킷 브레이커 패턴으로 외부에 영향을 받지 않도록 쓰레드와 세마포어 방식으로 분리되어 있음
실시간 구동 모니터링	• 히스트릭스를 통한 시간 모니터링과 설정 변경 지원 • 서비스와 설정 값의 변경이 어떻게 시스템에 적용되는지 대시보드를 통해서 확인 가능
병행성	• 병렬 실행 기능을 제공하며 여러 설정 값에 대한 변경을 지원하고, 내부적으로 중복되는 요청 처리를 줄이기 위하여 Request Caching과 일괄 처리를 위한 Request Collapsing 기능을 제공

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.2 서킷 브레이커 - 히스트릭스(Hystrix)

9.3.2.2 히스트릭스 라이브러리의 적용 예시

- **히스트릭스를 각 서비스를 호출하는 서비스인 카탈로그 서비스에 적용해 본다. 본 예제는 고객 서비스에서 호출한 API에 에러가 발생하거나 지연(1 초 이상)되는 경우 별도의 폴백 메소드를 실행하여 장애의 전파를 방지한다.**

[그림 9-29] Pom.xml에 히스트릭스 라이브러리 추가

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>${spring.cloud.version}</version>
</dependency>
```

[그림 9-30] CatalogsApplication.java에 @EnableCircuitBreaker 어노테이션[1] 추가

```
…
@ComponentScan("egovframework.*")
@EnableCircuitBreaker
@SpringBootApplication
public class CatalogsApplication {
…
}
```

- **고객 서비스가 에러 또는 지연될 경우 곧바로 폴백 메소드를 호출하여 에러 전파를 방지하도록 한다.**

[그림 9-31] CustomerApiServiceImpl.java에 @HystrixCommand 어노테이션 추가, 폴백 메소드 추가 및 작성

```
@Override
@HystrixCommand(fallbackMethod = "getCustomerDetailFallback")
public String getCustomerDetail(String customerId) {
    return restTemplate.getForObject("http://localhost:8082/customers/" + customerId,
String.class);
}

public String getCustomerDetailFallback(String customerId, Throwable ex) {
    System.out.println("Error:" + ex.getMessage());
    return "고객정보 조회가 지연되고 있습니다.";
}
```

1) 어노테이션(Annotation) : 주석이라는 뜻으로 자바에서 사용될 때 코드 사이에 주석처럼 쓰여서 특별한 의미, 기능을 수행하도록 하는 기술임

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.2 서킷 브레이커 - 히스트릭스(Hystrix)

9.3.2.2 히스트릭스 라이브러리의 적용 예시

- **원활한 테스트를 위하여 고객 서비스에 강제로 오류(Exception)을 발생하도록 한다. 고객 서비스의 CustomerController.java를 아래와 같이 수정한다.**

[그림 9-32] 테스트를 위한 CustomerController.java 수정

```
@GetMapping("/{customerId}")
public String getCustomerDetail(@PathVariable String customerId) {
    throw new RuntimeException("I/O Exception");
    //System.out.println("request customerId :" + customerId);
    //return "[Customer id = " + customerId + " at " + System.currentTimeMillis() + "]";
}
```

- **각각의 카탈로그 서비스와 고객 서비스를 실행하고, 아래와 같이 테스트 URL에 접속한다.**

  - 고객 서비스 구동 → 카탈로그 서비스 구동 → 테스트 URL 호출

  (URL : http://localhost:8081/catalogs/customerinfo/1234

[그림 9-33] 히스트릭스 구동 테스트

```
localhost:8081/catalogs/customerinfo/1234

[Customer id = 1234 at 1595004318761 고객정보 조회가 지연되고 있습니다.]
```

클라우드 네이티브 정보시스템 구현·이행 단계

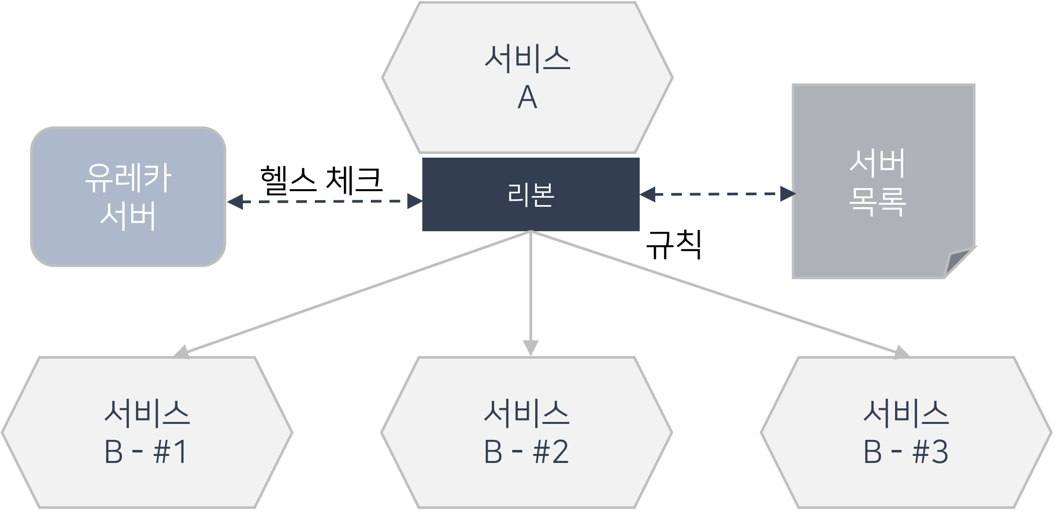
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.3 클라이언트 로드밸런서 – 리본(Ribbon)

9.3.3.1 역할 및 주요 기능

- **리본은 클라이언트에 탑재할 수 있는 소프트웨어 기반의 로드밸런서이다.**
- **일반적으로 서버 사이드에서는 하드웨어적인 L4 스위치를 사용하지만, 마이크로서비스 아키텍처에서는 소프트웨어적으로 구현된 클라이언트 측 로드밸런서를 주로 사용한다.**
- **로빈은 분산 처리 방법으로 여러 서버에 대한 라운드 로빈 방식의 부하분산 기능을 제공한다.**

[그림 9-34] 리본의 역할 및 주요 기능



구분	주요 기능
서버 목록	• 로드밸런싱 대상 목록 • 호스트 이름과 포트 번호 대신 서버목록의 서비스(서버) 이름으로 서비스를 호출할 수 있음 • 해당 서비스(서버)의 주소는 리본 설정 application.yml에 포함되어 있어야 함
로드밸런싱 규칙	• 기본은 라운드 로빈 방식 • 3회 연속 실패 시 일정 시간 동안 해당 서버로 로드밸런싱되지 않도록 설정 가능함
헬스 체크	• 유레카 서버에서 해당 서비스가 정상인지 확인 • 리본은 ping에 실패한 서비스(서버)를 임시로 로드밸런싱에서 제외함

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.3 클라이언트 로드밸런서 - 리본(Ribbon)

9.3.3.2 리본 라이브러리의 적용 예시

▪ **각 서비스를 호출하는 서비스인 카탈로그 서비스에 리본을 적용하도록 한다.**

[그림 9-35] Pom.xml에 리본 라이브러리 추가

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
    <version>${spring.cloud.version}</version>
</dependency>
```

[그림 9-36] CatalogsApplication.java의 RestTemplate에 @LoadBalanced 어노테이션 추가

```
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.web.client.RestTemplate;

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

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.3 클라이언트 로드밸런서 - 리본(Ribbon)

9.3.3.2 리본 라이브러리의 적용 예시

- **CustomerApiServiceImpl.java의 고객 서비스 호출 URL 주소를 명시적으로 변경한다.**

  (URL : localhost:8082 -> customer)

[그림 9-37] CustomerApiServiceImpl.java의 고객 서비스 호출 URL 변경

```
@Override
@HystrixCommand(fallbackMethod = "getCustomerDetailFallback")
public String getCustomerDetail(String customerId) {
    return restTemplate.getForObject("http://customer/customers/" + customerId,
String.class);
}
```

- **Application.yml에서 리본 설정을 추가한다. 설정 값의 앞부분인 customer가 명시적인 서비스명이 된다.**

[그림 9-38] Application.yml에 Ribbon 설정 추가

```
customer:
  ribbon:
    listOfServers: localhost:8082
```

- **각각의 카탈로그 서비스와 고객 서비스를 실행하고, 아래와 같이 테스트 URL에 접속한다.**

  - 고객 서비스 구동 → 카탈로그 서비스 구동 → 테스트 URL 호출

  (URL : http://localhost:8081/catalogs/customerinfo/1234

[그림 9-39] 히스트릭스 구동 테스트

```
localhost:8081/catalogs/customerinfo/1234

[Customer id = 1234 at 1595006861032 [Customer id = 1234 at 1595006860997] ]
```

09 클라우드 네이티브 정보시스템 구현·이행 단계

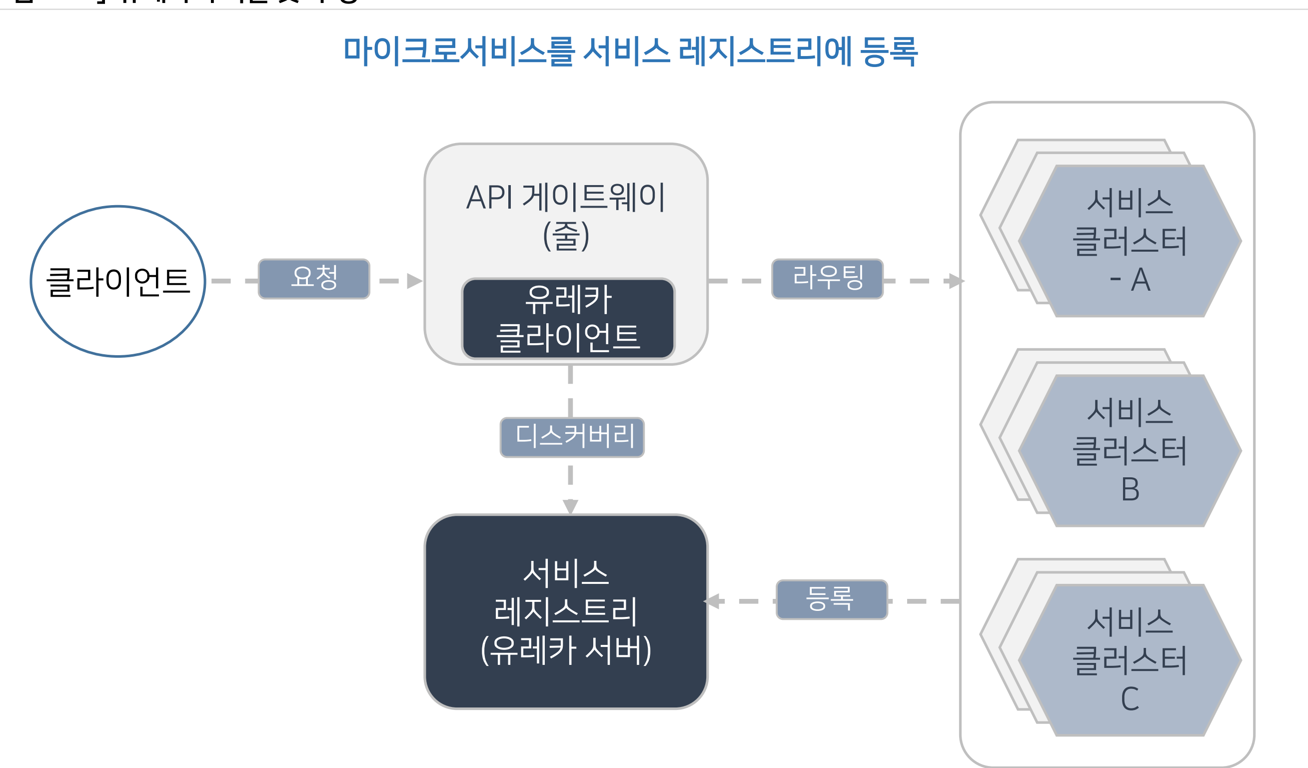
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.4 서비스 레지스트리 – 유레카(Eureka)

9.3.4.1 역할 및 주요 기능

- **유레카는 마이크로서비스 아키텍처의 장점 중 하나인 동적인 서비스 증설 및 축소를 위하여 필수적으로 필요한 서비스의 자가 등록, 탐색 및 부하분산에 사용될 수 있는 라이브러리이다.**
- **유레카는 마이크로서비스들의 정보를 레지스트리 서버에 등록할 수 있는 기능을 제공한다.**

[그림 9-40] 유레카의 역할 및 구성 요소



구분	주요 기능
유레카 서버	• 유레카 클라이언트에 해당하는 마이크로서비스들의 상태 정보가 등록되어 있는 레지스트리 서버
유레카 클라이언트	• 서비스가 시작될 때 Eureka 서버에 자신의 정보를 등록하고 이후 주기적으로 자신의 헬스체크(health check)를 통해 가용상태를 알리며, 일정 횟수 이상의 ping이 확인되지 않으면 유레카 서버에서 해당 서비스를 제외시킴

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

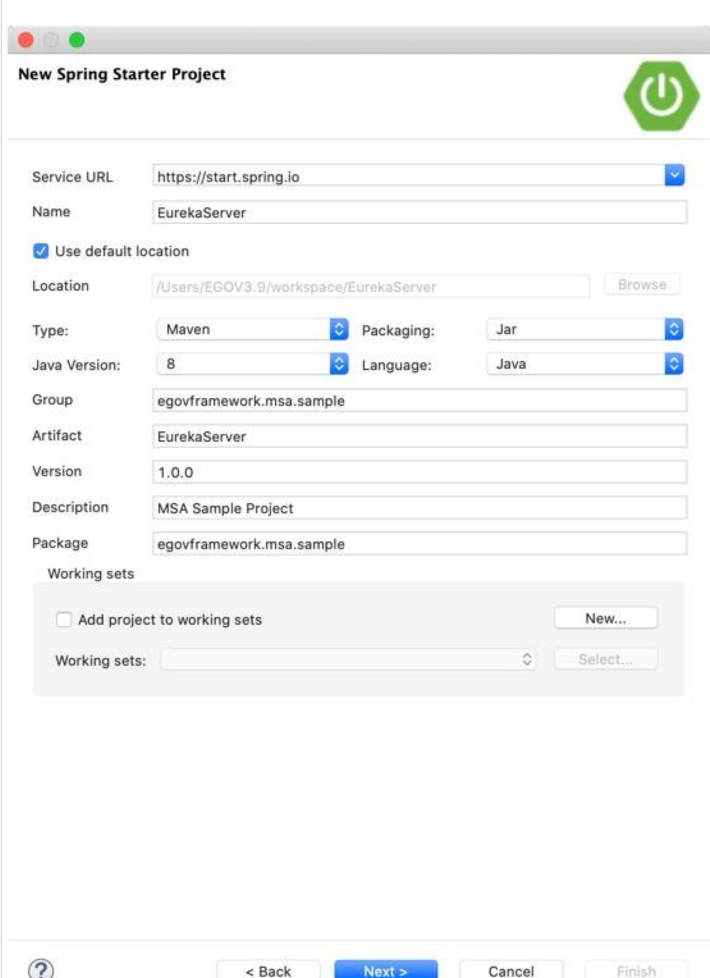
9.3.4 서비스 레지스트리 - 유레카(Eureka)

9.3.4.2 유레카 서버 서비스 작성 예시

- **전자정부 표준프레임워크 개발 환경을 활용하여 유레카 서버를 생성한다. (표준프레임워크 V3.10 기준)**

[그림 9-41] 유레카 서버 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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 : EurekaServer**
- Version : 1.0.0
- Description : MSA Sample Project Group Id : egovframework.msa.sample

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
 - 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
 (이 안내서에서는 의존 관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.4 서비스 레지스트리 - 유레카(Eureka)

9.3.4.2 유레카 서버 서비스 작성 예시

- **유레카 서버 서비스의 디렉토리 구조를 다음과 같이 설정한다.**

[표 9-11] Eureka 서버 서비스 프로젝트 디렉토리 구조 설정

SW명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	리소스 파일	스프링부트 설정 파일
EurekaServerApplication.java	egovframework.msa.sample.eureka	클래스 파일	애플리케이션 구동 파일

- **유레카 서버는 서비스 레지스트리 이외의 별도의 작업이 없으므로 모든 eGovframework 라이브러리를 제거하고 아래와 같이 유레카 서버 라이브러리만 등록한다.**

[그림 9-42] 유레카 서버 서비스의 의존성 추가 – 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>EurekaServer</artifactId>
    <version>1.0.0</version>
    <name>EurekaServer</name>
    <description>MSA Sample Project</description>

    <properties>
        <java.version>1.8</java.version>
        <spring.cloud.version>2.2.5.RELEASE</spring.cloud.version>
    </properties>

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

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.4 서비스 레지스트리 - 유레카(Eureka)

9.3.4.2 유레카 서버 서비스 작성 예시

[그림 9-43] 유레카 서버 서비스의 의존성 추가 – Pom.xml

```
        <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.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
            <version>${spring.cloud.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

- EurekaServerApplication 클래스에 @SpringBootApplication과 @EnableEurekaServer 어노테이션을 추가한다.

[그림 9-44] EurekaServerApplication.java 작성

```
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.4 서비스 레지스트리 - 유레카(Eureka)

9.3.4.2 유레카 서버 서비스 작성 예시

[그림 9-45] EurekaServerApplication.java 작성

```
public class EurekaServerApplication {

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

}
```

▪ **유레카 서버의 포트와 서버이름을 설정한다**

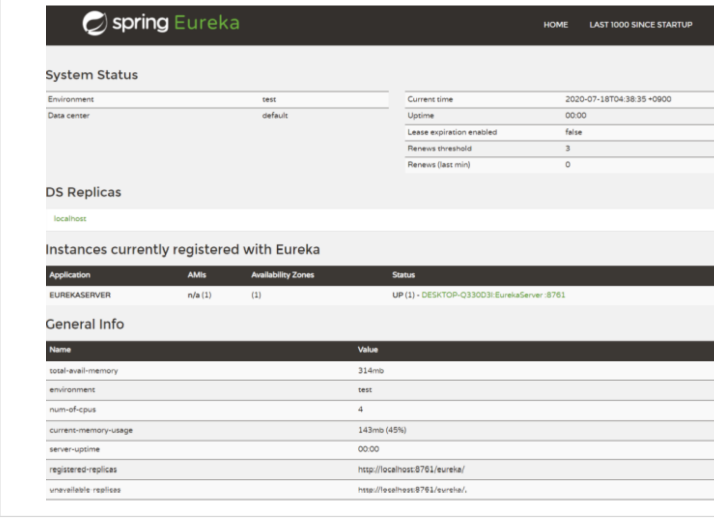
[그림 9-46] EurekaServer의 Application.yml 작성

```
server:
  port: 8761

spring:
  application:
    name: EurekaServer
```

▪ **유레카 서버의 구동 및 테스트를 위해 EurekaServerApplication.java 파일을 java application으로 실행하고 아래의 URL로 접속한다.**

(URL : http://localhost:8761/

[그림 9-47] EurekaServer 현황 조회



- 유레카 서버가 정상적으로 작동하면 유레카 서버의 현황을 볼 수 있는 대시보드 화면이 나타남
- 등록된 서비스와 현황을 조회할 수도 있음
- 코드 생성 및 XML 구성 불필요

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.4 서비스 레지스트리 - 유레카(Eureka)

9.3.4.3 유레카 클라이언트 서비스 작성 예시 – 카탈로그 서비스

- 기존의 카탈로그 서비스를 유레카 클라이언트로 적용하여 유레카 서버에 서비스를 등록할 수 있도록 작성한다.

[그림 9-48] 카탈로그 서비스에 유레카 클라이언트 라이브러리 적용

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>${spring.cloud.version}</version>
</dependency>
```

[그림 9-49] CatalogsApplication.java에 @EnableEurekaClient 추가

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

- 리본의 listOfServers 목록은 유레카 적용으로 불필요하여 주석 처리한다.

[그림 9-50] application.yml 파일 수정

```
server:
  port: 8081

spring:
  application:
    name: catalog

#customer:
  #ribbon:
    #listOfServers: localhost:8082

eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka  # default address
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.4 서비스 레지스트리 - 유레카(Eureka)

9.3.4.4 유레카 클라이언트 서비스 작성 예시 – 고객 서비스

- 기존의 고객 서비스를 유레카 클라이언트로 적용하여 Eureka 서버에 서비스를 등록할 수 있도록 작성한다.

[그림 9-51] 고객 서비스에 유레카 클라이언트 라이브러리 적용

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>${spring.cloud.version}</version>
</dependency>
```

[그림 9-52] CatalogsApplication.java에 @EnableEurekaClient 추가

```
@ComponentScan("egovframework.*")
@EnableEurekaClient
@SpringBootApplication
public class CustomersApplication {
...
}
```

[그림 9-53] application.yml 파일 수정

```
server:
  port: 8082

spring:
  application:
    name: customer

eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka  # default address
```

[그림 9-54] 유레카 서버에 등록된 서비스 확인

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 192.168.0.172:catalog:8081
CUSTOMER	n/a (1)	(1)	UP (1) - 192.168.0.172:customer:8082
EUREKASERVER	n/a (1)	(1)	UP (1) - 192.168.0.172:EurekaServer:8761

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

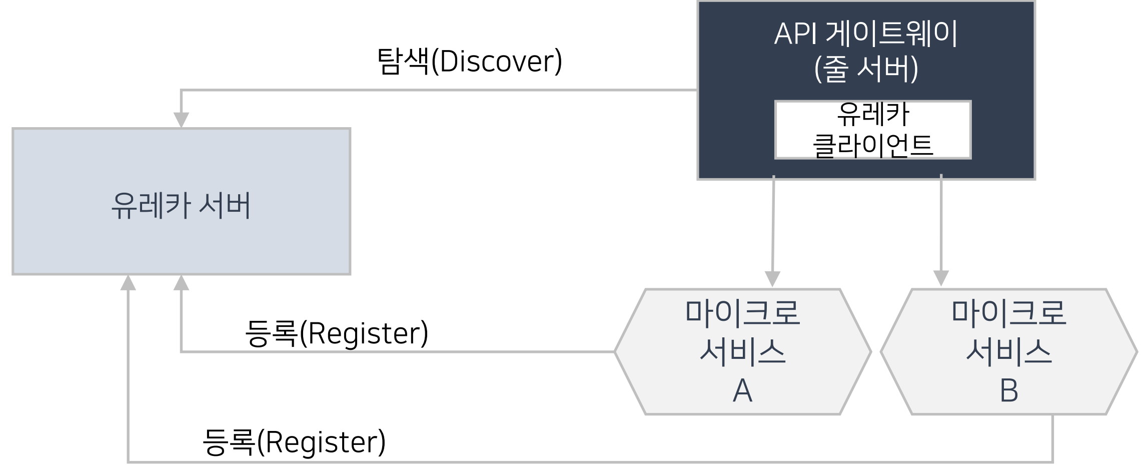
9.3.5 API 게이트웨이 – 줄(Zuul)

9.3.5.1 역할 및 주요 기능

- **API 게이트웨이란 모든 클라이언트 요청에 대한 엔드포인트를 통합하는 서비스로 마치 프록시 서버처럼 동작하며, 인증 및 권한, 모니터링, 로깅 등의 추가적인 기능도 지원한다.**
- **마이크로서비스 아키텍처에서는 도메인별로 하나 이상의 서비스(서버)가 존재하며, 한 서비스에 한 개 이상의 서버가 존재할 수 있으므로 사용자(클라이언트) 입장에서는 다수의 엔드포인트를 알아야 한다. 엔드포인트가 변경될 경우도 있으므로 이러한 문제 해결을 위해 하나로 통합할 수 있는 API 게이트웨이가 필요하다.**

[그림 9-55] 줄 서버의 역할 및 주요 기능

줄 서버는 단일 엔드포인트 역할을 하며, 이를 위해 유레카, 히스트릭스, 리본 등 여러 기능 내장



구분	주요 기능
인증 및 보안	• 클라이언트 요청 시, 각 리소스에 대한 인증 요구 사항을 식별하고 이를 충족하지 못한 경우 해당 요청을 거부하는 기능 제공
모니터링	• 모든 트래픽이 지나기 때문에 의미 있는 데이터와 지표를 수집할 수 있는 기능 제공
동적 라우팅	• 필요에 따라 즉시 원하는 백엔드 클러스터로 트래픽을 동적으로 라우팅하는 기능 제공
부하 테스트	• 성능 측정을 위하여 신규 서비스 또는 백엔드에 트래픽을 점진적으로 증가하는 방식으로 부하를 유발할 수 있는 기능 제공
트래픽 드랍	• 각 요청에 대해 용량을 할당하고, 제한된 이상의 요청이 발생한 경우 이를 제한할 수 있는 기능 제공
정적 응답 처리	• 특정 요청에 대해서는 백엔드로 트래픽을 보내는 대신 즉시 API 게이트웨이에서 응답을 처리하는 기능 제공

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

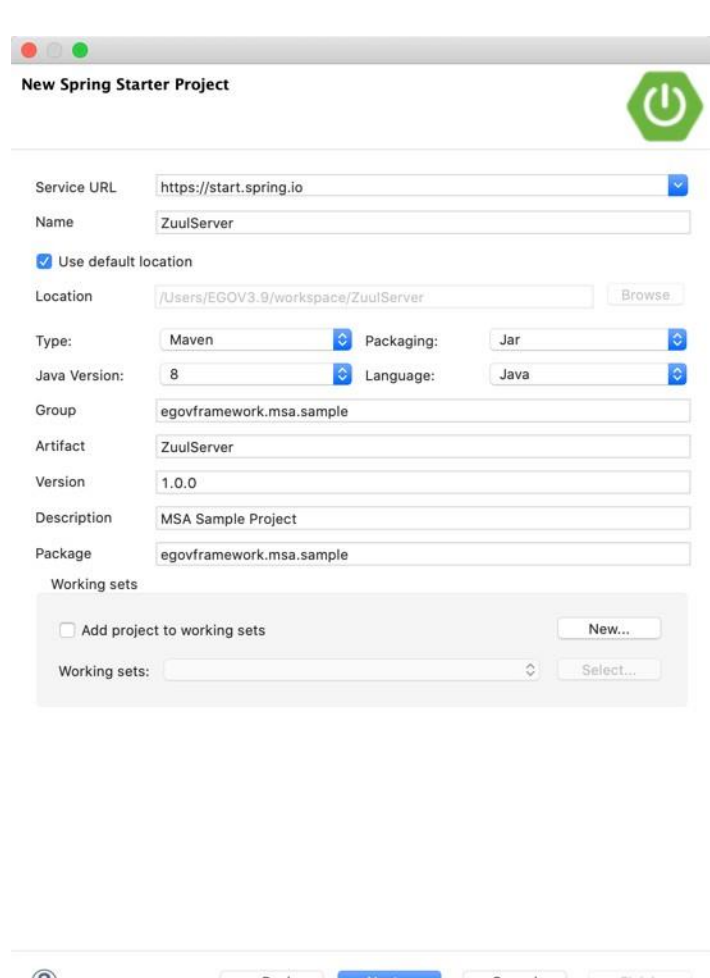
9.3.5 API 게이트웨이 - 줄(Zuul)

9.3.5.2 줄 서버 서비스 작성 예시

- **전자정부 표준프레임워크 개발환경을 활용하여 줄 서버를 생성한다. (표준프레임워크 V3.10 기준)**

[그림 9-56] 줄 서버 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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 : ZuulServer**
- Version : 1.0.0
- Description : MSA Sample Project
  Group Id : egovframework.msa.sample

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
- 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
  (이 안내서에서는 의존 관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.5 API 게이트웨이 - 줄(Zuul)

9.3.5.2 줄 서버 서비스 작성 예시

- **줄 서버 서비스의 디렉토리 구조를 다음과 같이 설정한다.**

[표 9-12] Eureka 서버 서비스 프로젝트 디렉토리 구조 설정

SW 명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	리소스 파일	스프링 부트 설정 파일
ZuulServerApplication.java	egovframework.msa.sample.eureka	클래스 파일	애플리케이션 구동 파일

- **줄 서버는 API 게이트웨이 역할 이외의 별도의 작업이 없으므로 모든 eGovframework 라이브러리를 제거하고 아래와 같이 줄, 유레카 클라이언트, Spring-retry 라이브러리를 등록한다.**

[그림 9-57] 줄 서버 서비스의 의존성 추가-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>ZuulServer</artifactId>
    <version>1.0.0</version>
    <name>ZuulServer</name>
    <description>MSA Sample Project</description>

    <properties>
        <java.version>1.8</java.version>
        <spring.cloud.version>2.2.5.RELEASE</spring.cloud.version>
    </properties>

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

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

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.5 API 게이트웨이 - 줄(Zuul)

9.3.5.2 줄 서버 서비스 작성 예시

[그림 9-58] 줄 서버 서비스의 의존성 추가 – Pom.xml

```
                <exclusions>
                    <exclusion>
                        <groupId>org.junit.vintage</groupId>
                        <artifactId>junit-vintage-engine</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
                <version>${spring.cloud.version}</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
                <version>${spring.cloud.version}</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.retry</groupId>
                <artifactId>spring-retry</artifactId>
            </dependency>
        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
            </plugins>
        </build>

    </project>
```

- ZuulServerApplication 클래스에 @SpringBootApplication, @EnableZuulProxy 와 @EnableDiscoveryClient 어노테이션을 추가한다.
- @EnableDiscoveryClient는 @EnableEurekaClient와 동일하게 작동하지만, @EnableEurekaClient는 Eureka 서버일 경우만 작동하고, @EnableDiscoveryClient는 Eureka 뿐 아니라 Consul, Zookeeper도 지원한다. 여기서는 @EnableDiscoveryClient를 사용하도록 한다.

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.5 API 게이트웨이 - 줄(Zuul)

9.3.5.2 줄 서버 서비스 작성 예시

[그림 9-59] ZuulServerApplication 클래스에 @EnableZuulProxy와 @EnableDiscoveryClient 어노테이션 추가

```
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ZuulServerApplication {

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

[그림 9-60] ZuulServer의 Application.yml 파일 작성

```
spring:
  application:
    name: zuul

server:
  port: 8080

zuul:
  routes:
    catalog:
      path: /catalogs/**
      serviceId: catalog
      stripPrefix: false
    customer:
      path: /customers/**
      serviceId: customer

eureka:
  instance:
    non-secure-port: ${server.port}
    prefer-ip-address: true
  client:
    service-url:
     defaultZone: http://localhost:8761/eureka
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.5 API 게이트웨이 - 줄(Zuul)

9.3.5.2 줄 서버 서비스 작성 예시

- **줄 서버의 구동 이전에 카탈로그, 고객, 유레카 서버 모두를 실행하고 정상 작동을 확인한다.**
- **이후 줄 서버를 기동하고 유레카 페이지에서 줄이 정상적으로 등록되었는지 확인한다.**

[그림 9-61] 줄 서버의 구동 및 테스트

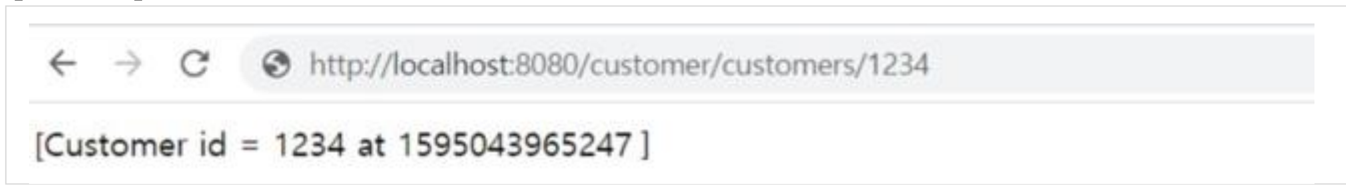
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 192.168.0.172:catalog:8081
CUSTOMER	n/a (1)	(1)	UP (1) - 192.168.0.172:customer:8082
EUREKASERVER	n/a (1)	(1)	UP (1) - 192.168.0.172:EurekaServer:8761
ZUUL	n/a (1)	(1)	UP (1) - 192.168.0.172:zuul:8080

- **정상적으로 구동이 확인되었으면 아래의 URL을 통해 결과를 확인한다. URL 모두 API 게이트웨이인 localhost:8080을 통하여 테스트를 진행한다.**

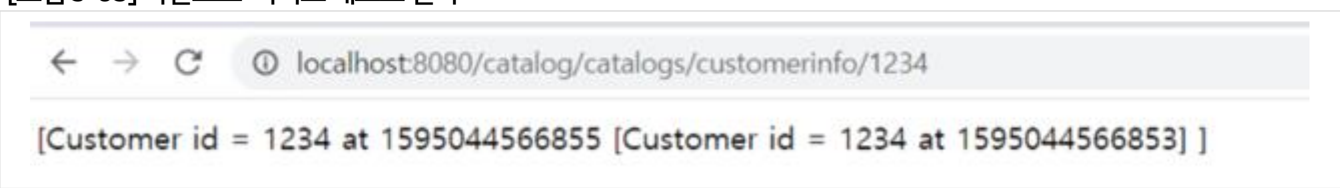
(고객 서비스 URL : http://localhost:8080/customer/customers/1234)

[그림 9-62] 고객 서비스 테스트 결과



(카탈로그 서비스 URL : http://localhost:8080/catalog/catalogs/customerinfo/1234)

[그림 9-63] 카탈로그 서비스 테스트 결과



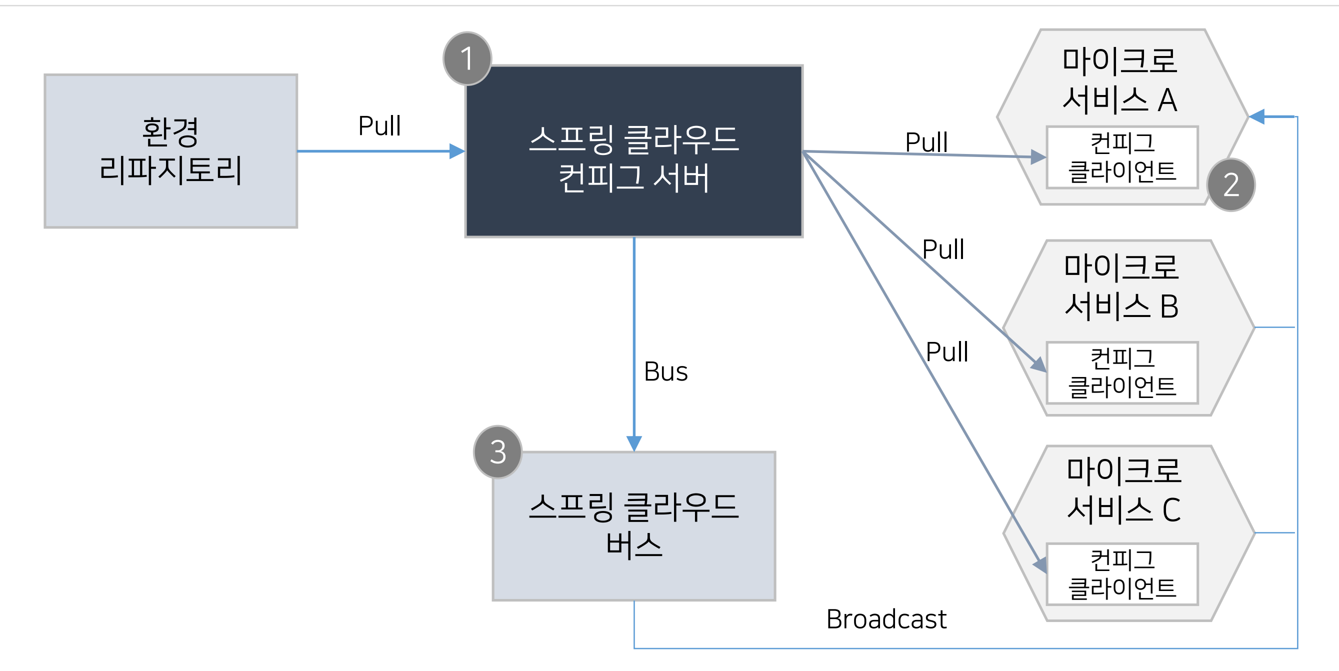
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.1 역할 및 주요 기능

- **컨피그 서버는 분산 시스템에서 애플리케이션의 환경 설정 정보 특히 서비스, 비즈니스 로직과 연관성 있는 정보들을 애플리케이션과 분리해 외부에서 관리하도록 환경 설정을 한다.**
- **스프링 컨피그 서버를 통해 수많은 애플리케이션들의 환경 설정 속성 정보를 중앙에서 관리할 수 있다. 여기서 환경 설정 속성 정보란 DB 접속 정보나 미들웨어(연계 서버) 접속 정보, 애플리케이션을 구성하는 각종 메타데이터들을 말한다. 이런 속성 값들은 보통 application.properties 또는 application.yml 파일에 기록하여 사용한다. 이러한 정보들을 마이크로서비스들이 제각각 따로 관리하게 될 경우 환경 설정 값이 변경되면 전체 마이크로서비스들을 다시 빌드해야 한다. 마이크로서비스의 개수가 많으면 많을수록 변경사항을 적용하는 데 많은 비용이 발생하게 된다.**

[그림 9-64] 컨피그 서버의 역할 및 주요 기능



구분	주요 기능
1 컨피그 서버	• 컨피그 서버가 구동될 때 환경 리파지토리(Environment Repository)에서 설정 내용을 가져옴 • 환경 리파지토리는 VCS, 파일 시스템, DB로 구성 가능(깃, SVN 등)
2 컨피그 클라이언트	• 각 마이크로서비스들은 서비스가 구동될 때 컨피그 서버에서 설정 내용을 내려 받아 애플리케이션이 초기화되고 구동
3 스프링 클라우드 버스	• 서비스 운영 중에 설정파일을 변경 해야 할 경우에는 스프링 클라우드 버스를 이용하여 모든 마이크로서비스의 환경설정을 업데이트할 수 있음(RabbitMQ 또는 카프카) • 이 메시지 브로커를 이용하여 각 서비스들에게 컨피그 변경사항을 전파함

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

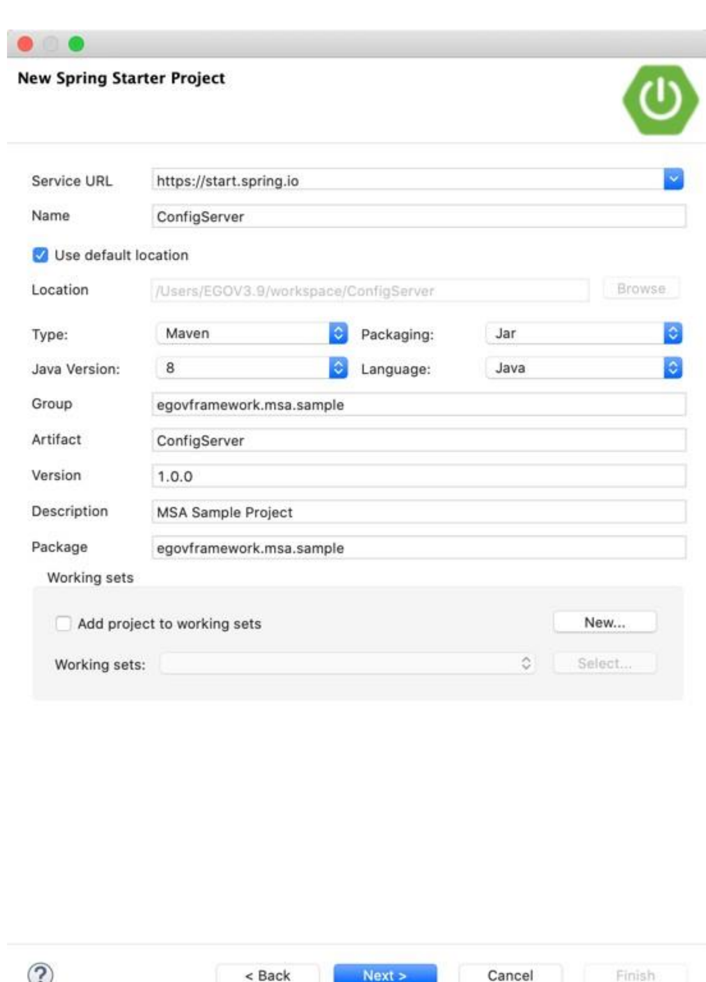
9.3.6 설정 서버 – 컨피그(Config)

9.3.6.2 컨피그 서버 설정 및 테스트

- **전자정부 표준프레임워크 개발환경을 활용하여 컨피그 서버를 생성한다. (표준프레임워크 V3.10 기준)**

[그림 9-65] 컨피그 서버 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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 : ConfigServer**
- Version : 1.0.0
- Description : MSA Sample Project
  Group Id : egovframework.msa.sample

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
- 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
  (이 안내서에서는 의존 관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.2 컨피그 서버 설정 및 테스트

- **컨피그 서버 서비스의 디렉토리 구조를 다음과 같이 설정한다.**

[표 9-13] 컨피그 서버 프로젝트 디렉토리 구조 설정

SW 명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	리소스 파일	스프링 부트 설정 파일
templateSimple-dev.yml templatePortal-dev.yml templateEnterprise-dev.yml	/src/main/resources/configurationreposi tory	리소스 환경 파일	애플리케이션 환경 설정 파일
ConfigServerApplication.java	egovframework.msa.sample	클래스 파일	애플리케이션 구동 파일

- **컨피그 서버는 환경변수 외부화 역할 이외의 별도의 작업이 없으므로 모든 egovframework 라이브러리를 제거하고 아래와 같이 컨피그 서버, 시큐리티 그리고 웹 라이브러리를 등록한다.**

[그림 9-66] 컨피그 서버 환경변수 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>ZuulServer</artifactId>
    <version>1.0.0</version>
    <name>ZuulServer</name>
    <description>MSA Sample Project</description>

    <properties>
        <java.version>1.8</java.version>
        <spring.cloud.version>2.2.5.RELEASE</spring.cloud.version>
    </properties>

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

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

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.2 컨피그 서버 설정 및 테스트

[그림 9-67] Config서버 환경변수 설정 – Pom.xml

```
            <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>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-server</artifactId>
            <version>${spring.cloud.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

- ConfigServerApplication 클래스에 @SpringBootApplication, @EnableConfigServer 어노테이션을 추가한다.

[그림 9-68] ConfigServerApplication.java 작성

```
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.2 컨피그 서버 설정 및 테스트

- **컨피그 서버의 관련 내용을 아래와 같이 설정한다. 환경 리파지토리는 깃, SVN, 파일 시스템, DB 등으로 구성이 가능하지만, 본 안내서에서는 파일 시스템을 이용하여 작성한다.**

[그림 9-69] 컨피그 서버의 application.yml 파일 작성

```
server:
  port: 8888

spring:
  application:
    name:
      -templateSimple
      -templatePortal
      -templateEnterprise

  profiles:
    active: native

  cloud:
    config:
      server:
        native:
          search-locations: classpath:configuration-repository/
```

- **templateSimple-dev.yml, templatePortal-dev.yml, templateEnterprise-dev.yml 등 환경 파일을 작성한다.**

[그림 9-70] 환경 파일 작성

1. templateSimple-dev.yml

```
config:
  profile: sht
  message: templateSimple(dev)

Globals:
  DbType: mysql
  DriverClassName: com.mysql.jdbc.Driver
  Url: jdbc:mysql://127.0.0.1:3306/sht
  UserName: com
  Password: com01
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.2 컨피그 서버 설정 및 테스트

[그림 9-72] 환경 파일 작성

2. TemplatePortal-dev.yml

```yaml
config:
  profile: pst
  message: templatePortal(dev)

Globals:
  DbType: mysql
  DriverClassName: com.mysql.jdbc.Driver
  Url: jdbc:mysql://127.0.0.1:3306/pst
  UserName: com
  Password: com01
```

3. TemplateEnterprise-dev.yml

```yaml
config:
  profile: ebt
  message: templateEnterprise(dev)

Globals:
  DbType: mysql
  DriverClassName: com.mysql.jdbc.Driver
  Url: jdbc:mysql://127.0.0.1:3306/ebt
  UserName: com
  Password: com01
```

- **ConfigServerApplication.java를 Run As > Java Application 으로 실행한다. 정상적으로 구동이 확인되었으면 http://localhost:8888/actuator에 접속하여 서버가 정상으로 작동하는지 확인한다.**

[그림 9-71] ConfigServer의 구동 및 테스트

localhost:8888/actuator
localhost:9265/message
localhost:8888/actuator

{"_links":{"self":{"href":"http://localhost:8888/actuator","templated":false},"health-path":
{"href":"http://localhost:8888/actuator/health/{*path}","templated":true},"health":
{"href":"http://localhost:8888/actuator/health","templated":false},"info":
{"href":"http://localhost:8888/actuator/info","templated":false}}}

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

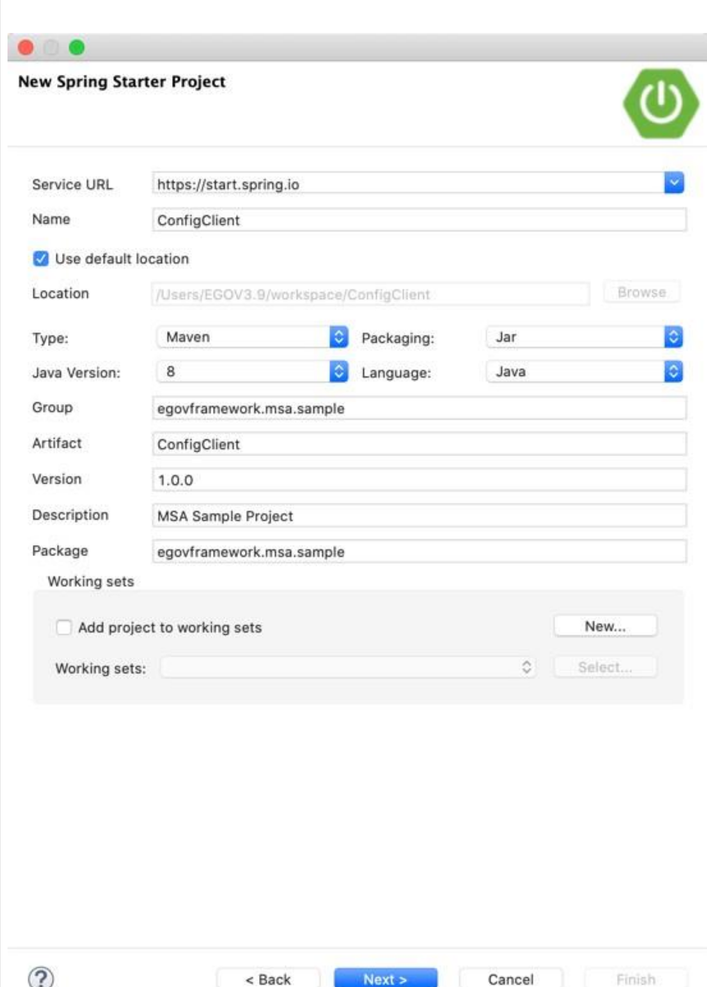
9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

- **전자정부 표준프레임워크 개발환경을 활용하여 컨피그 클라이언트를 생성한다. (표준프레임워크 V3.10 기준)**

[그림 9-73] 컨피그 클라이언트 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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 : ConfigClient**
- Version : 1.0.0
- Description : MSA Sample Project
  Group Id : egovframework.msa.sample

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
 - 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
 (이 안내서에서는 의존관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

- **컨피그 클라이언트 서비스의 디렉토리 구조를 다음과 같이 설정한다.**

[표 9-14] 컨피그 서버 프로젝트 디렉토리 구조 설정

SW 명	패키지명	유형	비고
Pom.xml	/		의존성 관리 파일
Application.yml	src/main/resources	Resource 파일	스프링 부트 설정 파일
bootstrap.yml	/src/main/resources	Resource 파일	스프링 부트 설정 파일(application.yml 보다 우선순위)
ConfigClientApplication.java	egovframework.msa.sample	클래스 파일	애플리케이션 구동 파일
ConfigClientController. java	egovframework.msa.sample.controller	클래스 파일	테스트용 컨트롤러 파일

- **Pom.xml을 아래와 같이 설정한다.**

[그림 9-74] pom.xml 설정

```
<?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>ConfigClient</artifactId>
    <version>1.0.0</version>
    <name>ConfigClient</name>
    <description>MSA Sample Project</description>

    <properties>
        <java.version>1.8</java.version>
        <spring.cloud.version>2.2.5.RELEASE</spring.cloud.version>
    </properties>

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

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

[그림 9-75] pom.xml 설정

```
        <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>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
            <version>${spring.cloud.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

- **ConfigClientApplication 클래스에 @SpringBootApplication 어노테이션을 추가한다.**

[그림 9-76] ConfigClientApplication.java 작성

```java
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ConfigClientApplication {

    public static void main(String[] args) {

        String profile = System.getProperty("spring.profiles.active");
        if (profile == null) {
            System.setProperty("spring.profiles.active", "dev");
        }

        SpringApplication.run(ConfigClientApplication.class, args);

    }

}
```

- **ConfigClientController 클래스에 @RestController와 @RefreshScope 어노테이션을 추가한다. @RefreshScope 어노테이션을 추가해 주어야 컨피그 서버의 Config 정보를 갱신(refresh)할 수 있다.**

[그림 9-77] ConfigClientController.java 작성

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

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RefreshScope
public class ConfigClientController {

    @Value("${config.profile}")
    private String profile;

    @Value("${config.message}")
    private String message;

    @GetMapping("/config/profile")
    public String profile() {
        return profile;
    }

    @GetMapping("/config/message")
    public String message() {
        return message;
    }
}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

- **컨피그 클라이언트의 Bootstrap.yml과 Application.yml 파일을 다음과 같이 작성한다.**

[그림 9-78] 컨피그 클라이언트의 bootstrap.yml과 application.yml 작성

1. bootstrap.yml 작성

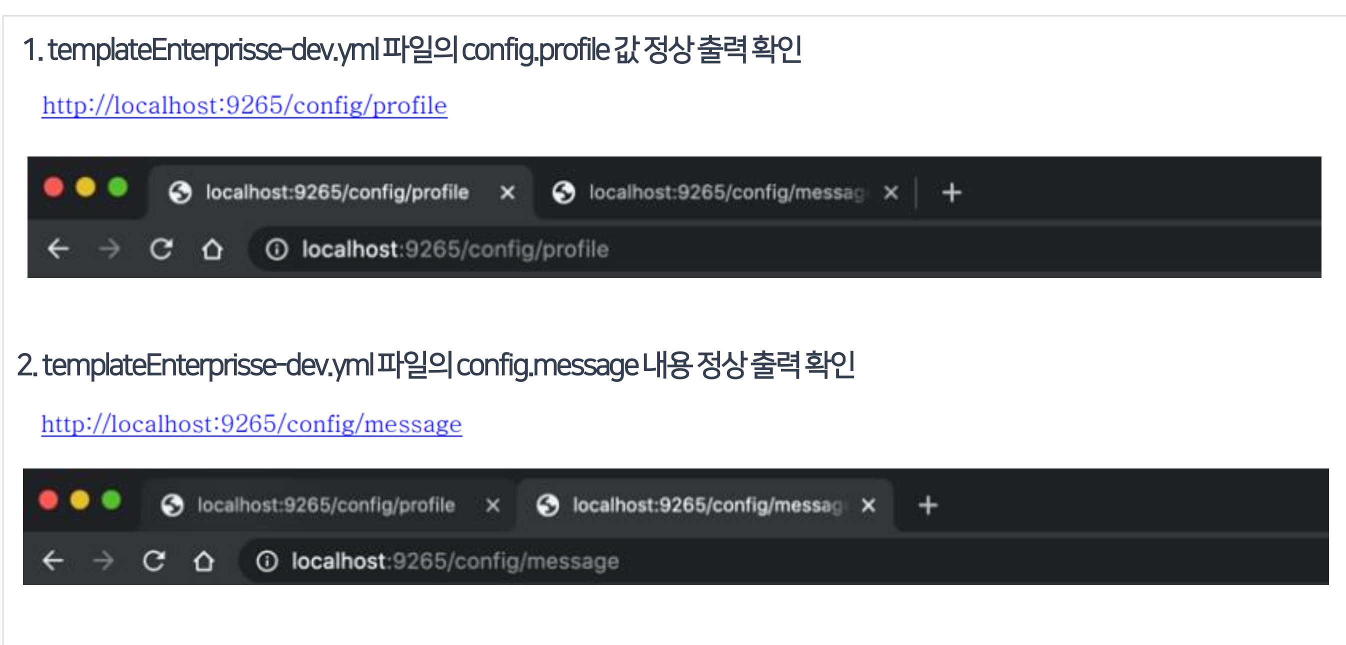
```
server:
  port: 9265

spring:
  application:
    name: templateEnterprise  #서비스ID (Config클라이언트가 어떤 서비스를 조회할 것인지 매핑)

  cloud:
    config:
      uri: http://localhost:8888    #Config서버 URL
```

2. application.yml 작성

```
management:
  endpoints:
    web:
      exposure:
        include: ['env', 'refresh'] #엔드포인트, ex) http://localhost:9265/actuator/refresh (POST)
```

- **ConfigClientApplication.java를 Run As > Java Application으로 실행한다. 정상적으로 구동이 확인되었으면 아래의 URL을 통하여 결과를 확인한다.**

[그림 9-79] 컨피그 클라이언트의 구동 및 테스트



9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

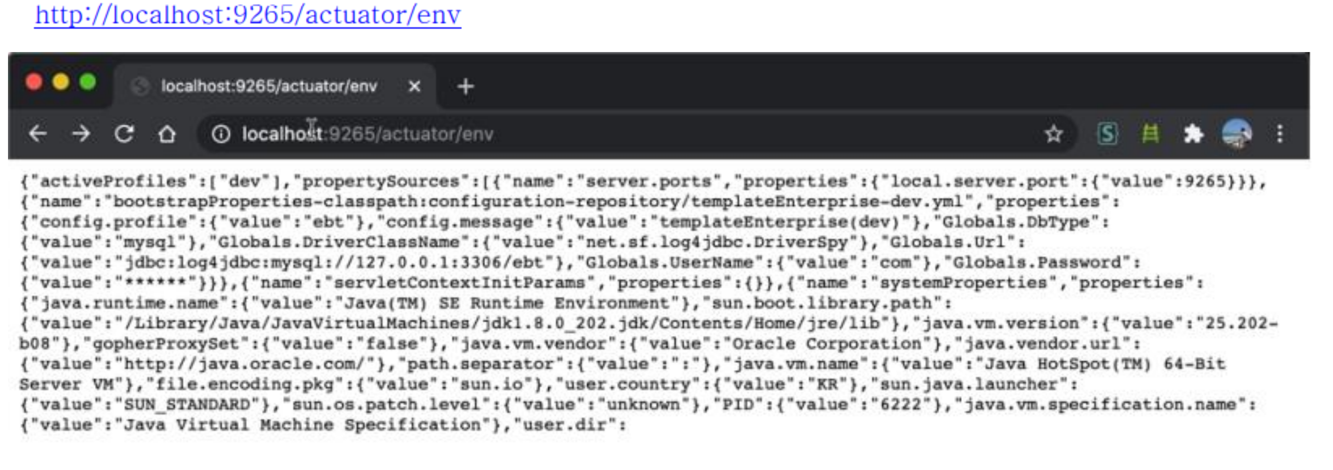
9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

[그림 9-80] 컨피그 클라이언트의 구동 및 테스트

3. 컨피그 서버의 모든 환경설정 값들의 JSON 형식의 문자열 출력 확인

http://localhost:9265/actuator/env



- **컨피그 서버의 다음의 환경 파일에서 config.profile 속성 정보를 'ebt'에서 'ebt2'로 변경한다.**

[그림 9-81] templateEnterprise-dev.yml 작성

```
config:
  profile: ebt2  # ebt 에서 ebt2로 변경
  message: templateEnterprise(dev)

Globals:
  DbType: mysql
  DriverClassName: com.mysql.jdbc.Driver
  Url: jdbc:mysql://127.0.0.1:3306/ebt
  UserName: com
  Password: com01
```

- **컨피그 클라이언트의 설정(속성) 정보를 갱신하는 요청을 보내도록 한다. 아래의 URL을 POST로 요청하여 변경된 속성 정보가 출력되는지 확인한다. (POST로 요청)**

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.6 설정 서버 – 컨피그(Config)

9.3.6.3 컨피그 클라이언트 설정 및 테스트

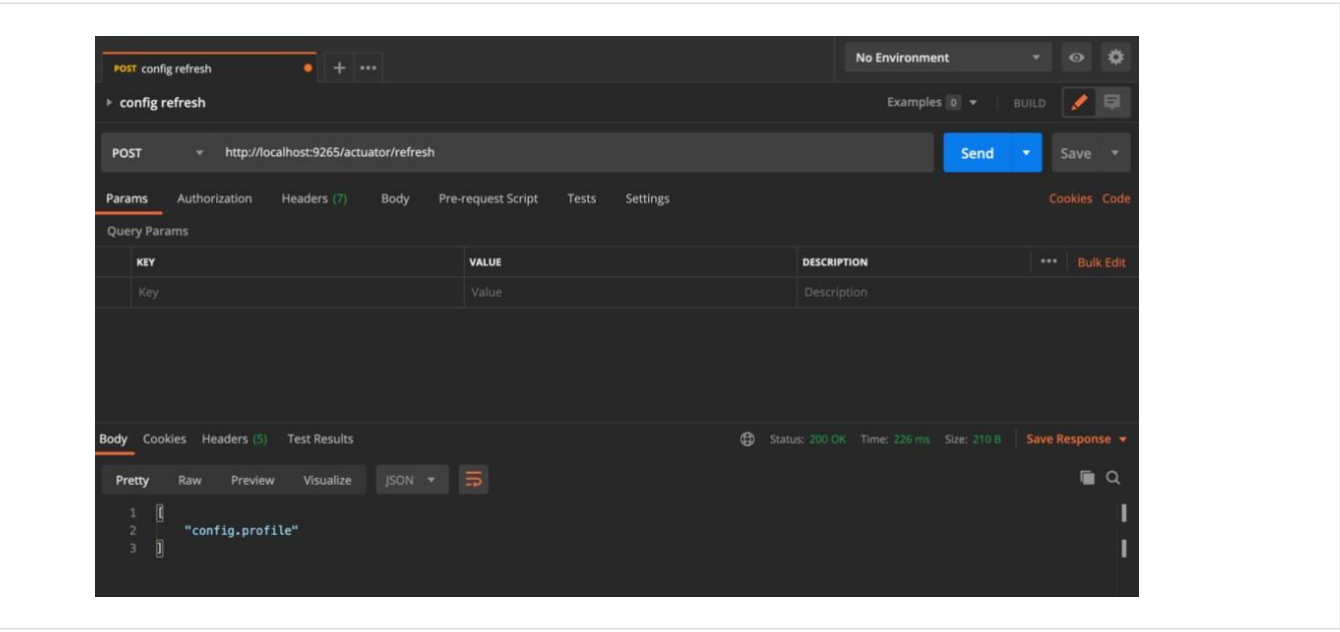
- **리눅스, 맥(Mac)의 경우 curl-X POST http://localhost:9265/actuator/refresh로 요청하여 결과를 확인한다. 변경된 속성 정보가 JSON 문자열로 출력된다.**

[그림 9-82] 리눅스, 맥의 경우 JSON 문자열 출력

```
> curl -X POST http://localhost:9265/actuator/refresh
["config.profile"]
```

- **그 외 SoapUI, Postman, JMeter 등의 REST API 테스트 도구를 이용하여 POST 요청을 할 수 있고, 요청 결과를 확인할 수 있다. (아래 그림 예시 참조)**
- **REST API 테스트 도구(Postman)를 이용하여 설정(속성) 정보를 갱신하도록 요청을 보내고, 결과를 확인하는 그림이다. (변경된 속성 정보가 JSON 문자열로 출력된다.)**

[그림 9-83] Rest PI 테스트 요청 결과 확인



- **위 요청을 보내고 정상적으로 변경이 된 것을 확인한 후 http://localhost:9265/config/profile로 접속해 보면 변경된 속성 정보가 출력되는 것을 확인할 수 있다.**

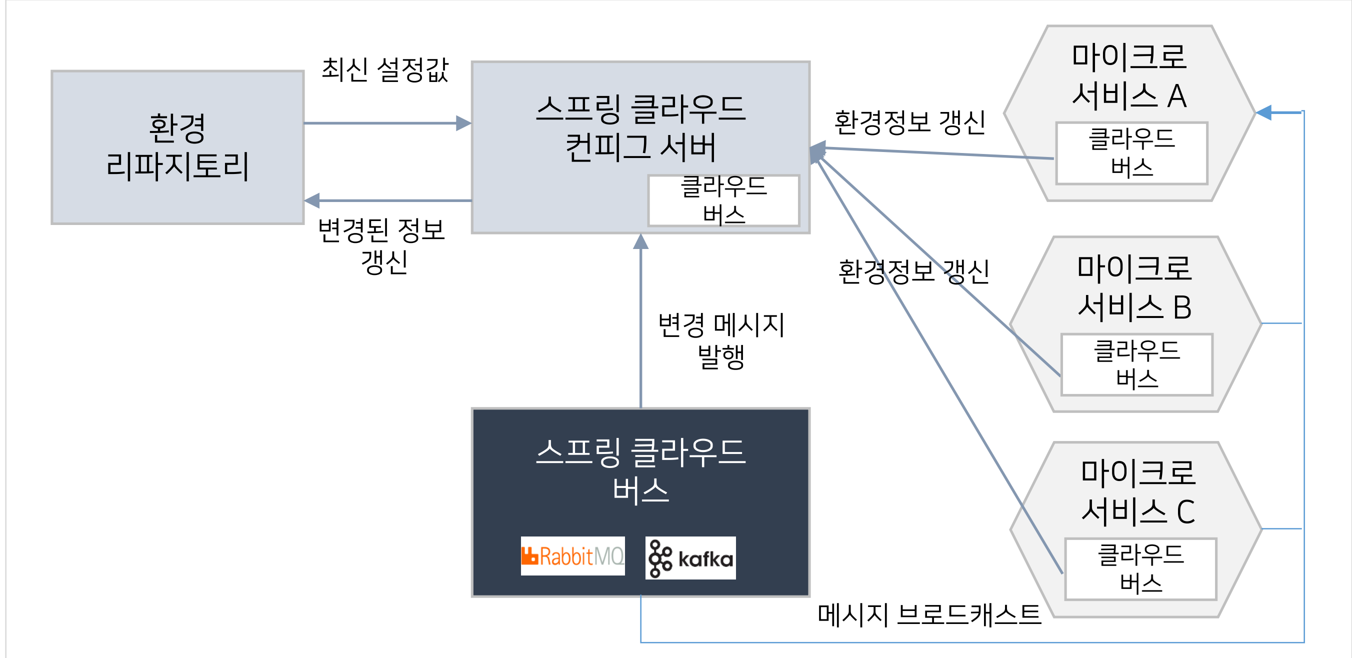
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.7 스프링 클라우드 버스(Bus)

9.3.7.1 역할 및 주요 기능

- **컨피그 서버에서 마이크로서비스들에게 갱신(Refresh)를 수행하라는 메시지를 전송해야 한다. 이 메시지를 전송해 주는 컴포넌트를 메시지 브로커라고 부르며, RabbitMQ, Kafka 등이 있다.**
- **스프링 클라우드 버스는 분산 시스템에 존재하는 노드들을 경량 메시지 브로커(RabbitMQ, Kafka 등)와 연결하는 역할을 한다.**
- **구성 변경과 같은 상태 변경, 기타 관리 등을 브로드캐스트하는 데 사용이 가능하다.**
- **컨피그 서버의 설정 값이 변경될 경우, 마이크로서비스들이 변경된 설정 값을 갱신하기 위해 마이크로 서비스를 재시작하거나, 컨피그 클라이언트를 갱신해줘야 한다. 하지만 마이크로서비스의 개수가 많을 경우 각각의 서비스들을 모두 수동으로 재시작하거나 갱신하기는 쉽지 않다.**

[그림 9-84] 스프링 클라우드 버스와 RabbitMQ의 역할 및 주요 기능



RabbitMQ

- 소용량 데이터용
- 메시지 브로커
- 메시지 전달 보장, 시스템 간 메시지 전달
- 브로커, 소비자 중심

kafka

- 대용량 데이터용
- 이벤트 처리
- Pub/sub, Topic에 메시지 전달
- 생산자 중심

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.7 스프링 클라우드 버스(Bus)

9.3.7.2 RabbitMQ 설치 및 구동

- RabbitMQ를 설치하는 방법은 각 OS 환경에 따라 다양한 방법이 존재한다. 본 안내서에서는 각 환경에 따른 RabbitMQ 설치방법에 대해서는 생략한다.
- RabbitMQ의 자세한 설치방법은 아래의 공식 설치 가이드를 참조하도록 한다.

  - RabbitMQ 다운로드 및 설치 가이드 : https://www.rabbitmq.com/download.html

  - Windows : https://www.rabbitmq.com/install-windows.html

  - MacOS : https://www.rabbitmq.com/install-homebrew.html

  - Linux (Debian/Ubuntu) : https://www.rabbitmq.com/install-debian.html

- 이하는 RabbitMQ를 설치한 후 RabbitMQ를 실행한 상태라는 가정하에 설명한다. RabbitMQ를 설치한 후 실행시키면 아래 주소로 관리 페이지에 접속이 가능하다.

  - RabbitMQ 관리 URL : http://localhost:15672/

- 위 주소로 접속을 해 보면 RabbitMQ 로그인 화면이 출력되는데, 로컬에서 접속 시 guest/guest로 로그인 가능하다. (초기 로그인 후 관리자 계정을 생성해서 사용하길 권장한다.)

[그림 9-85] RabbitMQ 로그인

- 스프링 클라우드 버스(RabbitMQ)를 이용하기 위해서 클라이언트 애플리케이션을 수정해야 한다.
- 컨피그 클라이언트 애플리케이션의 pom.xml 파일에 아래의 의존 라이브러리를 추가한다. RabbitMQ는 AMQP(Advanced Message Queuing Protocal)을 구현한 서비스이므로 spring-cloud-starter-bus-amqp 라이브러리를 추가해 준다.

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.7 스프링 클라우드 버스(Bus)

9.3.7.2 RabbitMQ 설치 및 구동

[그림 9-86] 컨피그 클라이언트 수정을 위한 pom.xml 수정

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
  <version>2.2.3.RELEASE</version>
</dependency>
```

- **컨피그 클라이언트 애플리케이션의 application.yml 파일을 아래와 같이 수정한다. 설치한 RabbitMQ의 정보를 입력하고 엔드포인트를 "*"로 변경한다.**

[그림 9-87] application.yml 파일 수정

```
spring:
  rabbitmq:
    host: localhost          #rabbitmq
    호스트 port: 5672         #rabbitmq
    서비스 포트 username: guest    #rabbitmq
    사용자명 password: guest $rabbitmq
    비밀번호

management:
  endpoints:
    web:
      exposure:
        include: "*" #엔드포인트, ex) http://localhost:9265/actuator/bus-refresh (POST)
```

- **버스(Bus) 테스트를 하기 위하여 컨피그 클라이언트 애플리케이션을 최소 2개 이상 실행해 준다. 여러 개의 컨피그 클라이언트 애플리케이션을 실행하기 위해서는 서버 포트를 변경해서 실행해 주어야 한다. 아래의 bootstrap.yml 파일의 서버 포트를 변경하여 2개 이상 띄워준다. (예 : 9265, 9266...).**

[그림 9-88] 컨피그 클라이언트–버스 테스트(bootstrap.yml 수정 후 실행)

```
server:
  port: 9265  #서버포트

spring:
  application:
    name: templateEnterprise        #서비스ID (Config Client가 어떤 서비스를 조회할지의 매핑)

  cloud:
    config:
      uri: http://localhost:8888           #Config서버 URL
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.7 스프링 클라우드 버스(Bus)

9.3.7.2 RabbitMQ 설치 및 구동

- **컨피그 클라이언트 애플리케이션을 실행(재실행)한 후 RabbitMQ 관리페이지에 접속해 보면 현재 연결된 컨피그 클라이언트 정보와 springCloudBus 토픽이 추가된 것을 확인할 수 있다.**

  - RabbitMQ 관리 URL : http://localhost:15672/

[그림 9-89] RabbitQ 화면에서 현재 연결된 클라이언트 수 및 정보 확인

1. 연결된 클라이언트 수 확인

Global counts: Connections: 2 | Channels: 2 | Exchanges: 8 | Queues: 2 | Consumers: 2

2. 연결된 클라이언트 정보 확인

Connections — All connections (2)

Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
127.0.0.1:65327 rabbitConnectionFactory#3468ee6e:0	guest	running	○	AMQP 0-9-1	1	0 B/s	0 B/s
127.0.0.1:65348 rabbitConnectionFactory#6cd98a05:0	guest	running	○	AMQP 0-9-1	1	0 B/s	0 B/s

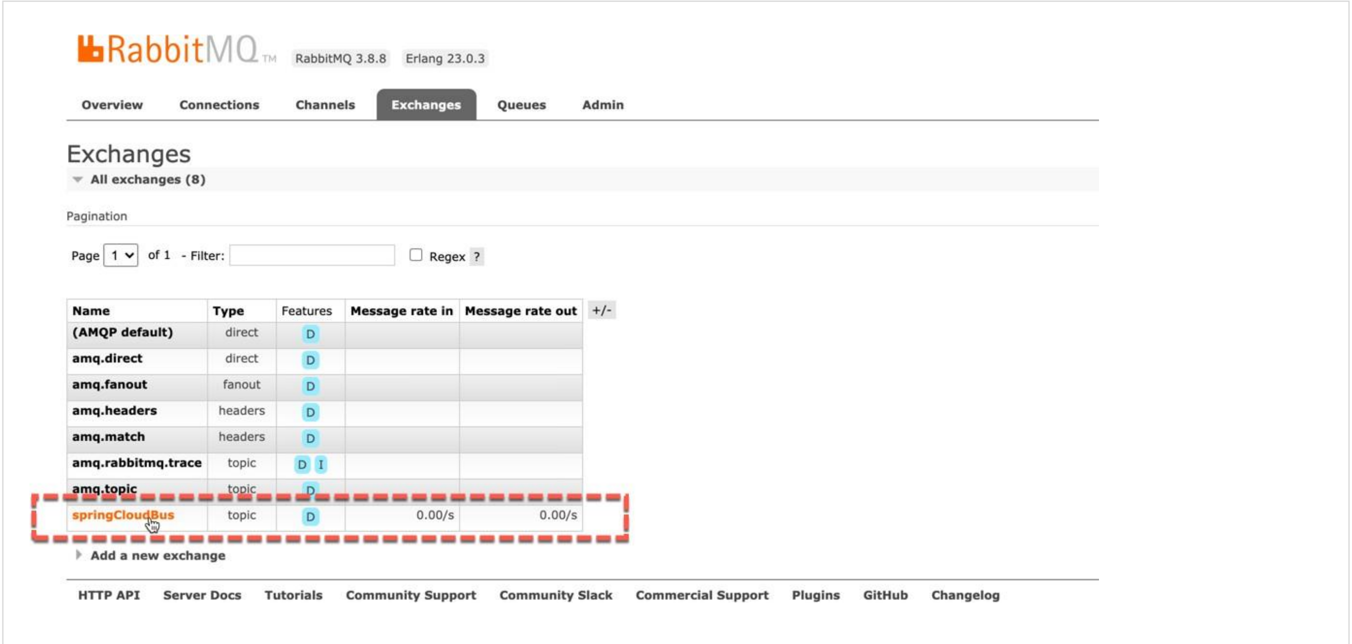
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.7 스프링 클라우드 버스(Bus)

9.3.7.2 RabbitMQ 설치 및 구동

- **Exchanges 화면에서 springCloudBus 토픽이 추가된 것을 확인할 수 있다. RabbitMQ 관리페이지에서 아래의 항목들이 확인되었다면 스프링 클라우드 버스를 이용할 준비가 완료된 것이다.**

[그림 9-90] springCloudBus 토픽 추가 확인



- **springCloudBus 테스트를 하기 위해 컨피그 서버의 templateEnterprise-dev.yml 파일의 config.profile 값을 변경한다.**

[그림 9-91] springCloudBus 테스트를 위한 config.profile 값 변경

templateEnterprise-dev.yml

```
1
2 config:
3   profile: ebt
4   message: templateEnterprise(dev)
5
6 Globals:
7   DbType: mysql
8   DriverClassName: com.mysql.jdbc.Driver
9   Url: jdbc:mysql://127.0.0.1:3306/ebt
10  UserName: com
11  Password: com01
12
13
```

→

templateEnterprise-dev.yml

```
1
2 config:
3   profile: ebt-changed
4   message: templateEnterprise(dev)-changed
5
6 Globals:
7   DbType: mysql
8   DriverClassName: com.mysql.jdbc.Driver
9   Url: jdbc:mysql://127.0.0.1:3306/ebt
10  UserName: com
11  Password: com01
12
13
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.7 스프링 클라우드 버스(Bus)

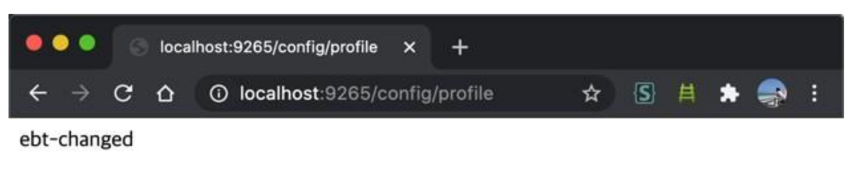
9.3.7.2 RabbitMQ 설치 및 구동

- **컨피그 서버를 재시작한다.**
- **실행되어 있는 컨피그 클라이언트의 bus-refresh 엔드포인트를 호출한다.(POST로 호출)**

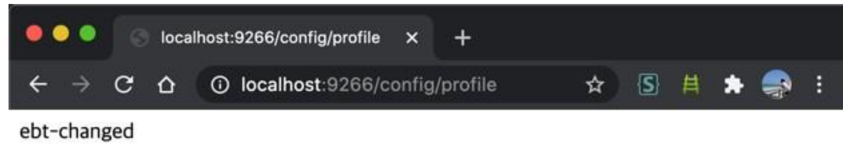
  - 실행 명령어 : curl-X POST http://localhost:9265/actuator/bus-refresh
- **bus-refresh를 클라이언트 한 곳만 호출하면 자동으로 모든 클라이언트 서비스들에게 전파된다.**
- **실행되어 있는 컨피그 클라이언트들의 config.profile 값을 확인한다.**

[그림 9-92] 컨피그 클라이언트의 config.profile 값 확인

1. ConfiClient1 테스트 URL : http://localhost:9265/config/profile



2. ConfiClient2 테스트 URL : http://localhost:9266/config/profile



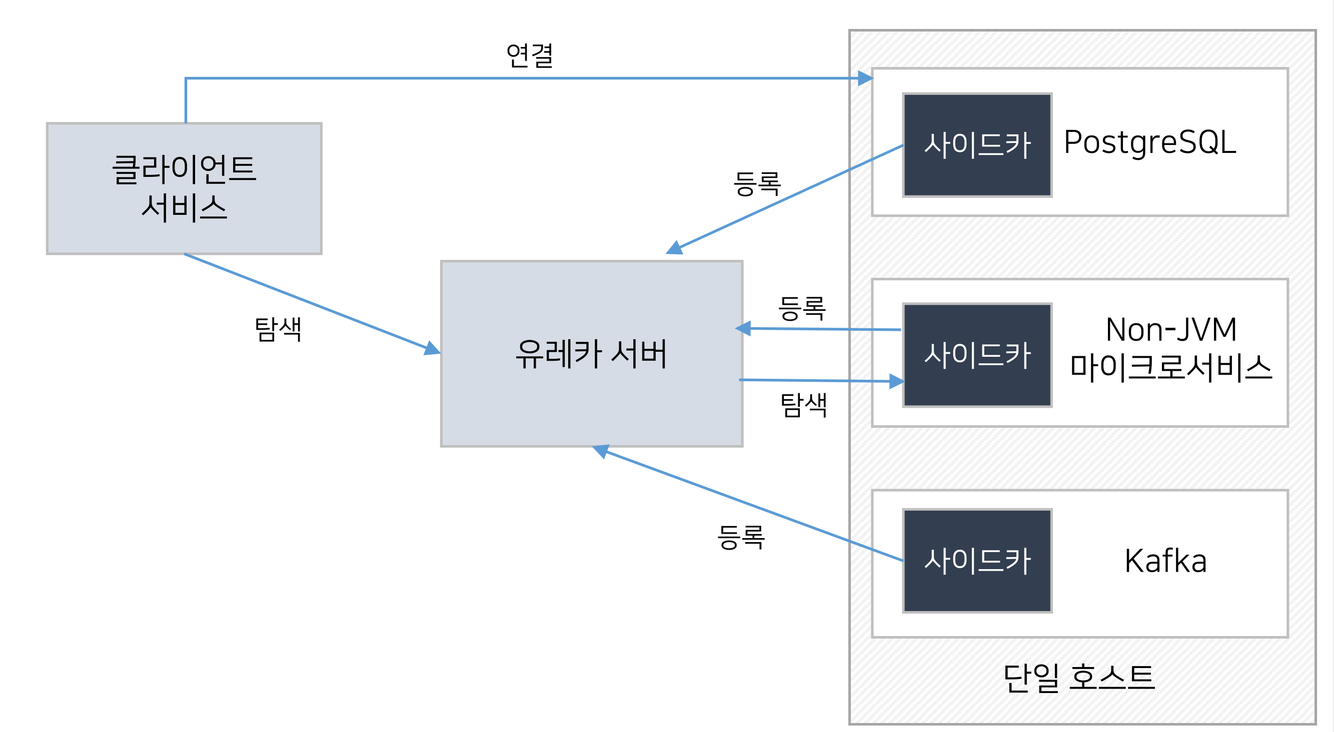
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.8 폴리글랏 지원 – 사이드카(Sidecar)

9.3.8.1 역할 및 주요 기능

- **스프링 클라우드는 스프링 기반으로 작성된 애플리케이션만 동작이 가능하다. 비 JVM 애플리케이션 (Python, Go, Ruby, C#, PHP 등 Java 이외의 언어로 작성된 애플리케이션)을 스프링 클라우드와 엮어서 동작하도록 하려면 스프링 클라우드 사이드카를 사용하면 가능하다.**
- **일반적으로 마이크로서비스와 같은 분산 시스템에서 하나의 호스트에서 동작하는 이 두 개의 메인과 보조 애플리케이션은 서로 다른 언어나 프레임워크 기반으로 동작한다. 그리고 보조 애플리케이션의 역할은 전체 시스템에서 사용하는 공통 모듈과 동일한 언어나 프레임워크로 구성한다.**

[그림 9-93] 스프링 클라우드 사이드카의 역할 및 주요 기능



- 위의 사이드카 패턴은 서비스 디스커버리인 유레카와 함께 동작하는 방식임
- PostgreSQL, Kafka, JVM이 아닌 애플케이션들은 JVM 기반으로 Eureka 클라이언트와 함께 동작하는 사이드카 애플리케이션을 가짐
- 유레카 클라이언트는 메인 애플리케이션이 아니라 사이드카에서 동작하며, 다른 호스트에서 동작하는 유레카 서버와 다른 클라이언트에 대한 정보를 공유하거나 자신이 구동하고 있는 애플리케이션의 정보를 제공함(메인 애플리케이션은 사이드카와 통신하며 플랫폼에서 제공하는 정보를 얻거나 내보낼 수 있음)
- 사이드카 패턴이 폴리글랏을 처리하는 핵심 기능임

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

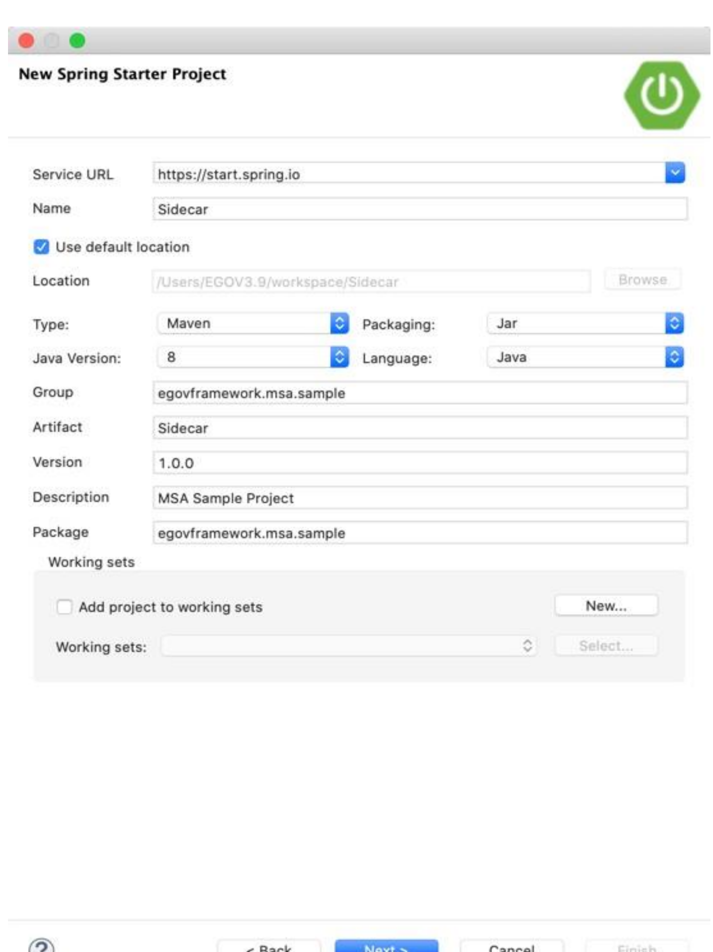
9.3.8 폴리글랏 지원 – 사이드카(Sidecar)

9.3.8.2 사이드카 설정

- **사이드카(Sidecar) 작성을 위해 컨피그 클라이언트 애플리케이션을 전자정부 표준프레임워크 개발환경을 활용하여 생성한다. (표준 프레임워크 V3.10 기준)**

[그림 9-94] Sidecar 프로젝트 생성

① New > Project > Spring Boot > Spring Starter Project를 선택 후 아래와 같이 입력한 후 Next를 선택한다.



- 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 : Sidecar**
- Version : 1.0.0
- Description : MSA Sample Project
  Group Id : egovframework.msa.sample

② Next > Finish 또는 Finish를 바로 선택하여 프로젝트를 생성한다.
- 다음 단계는 프로젝트의 의존성(Dependency)을 추가하는 단계인데 여기서는 선택하지 않는다.
  (이 안내서에서는 의존관계를 pom.xml에 직접 등록하는 방법으로 진행한다.)

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.8 폴리글랏 지원 – 사이드카(Sidecar)

9.3.8.2 사이드카 설정

- **Pom.xml을 아래와 같이 설정한다.**

[그림 9-95] 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>Sidecar</artifactId>
    <version>1.0.0</version>
    <name>Sidecar</name>
    <description>MSA Sample Project</description>

    <properties>
        <java.version>1.8</java.version>
        <spring.cloud.version>2.2.5.RELEASE</spring.cloud.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </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.cloud</groupId>
            <artifactId>spring-cloud-netflix-sidecar</artifactId>
            <version>${spring.cloud.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.8 폴리글랏 지원 – 사이드카(Sidecar)

9.3.8.2 사이드카 설정

- **SidecarApplication 클래스에 @SpringBootApplication과 @EnableSidecar 어노테이션을 추가한다. 이 어노테이션은 내부적으로 @EnableCircuitBreaker, @EnableDiscoveryClient, 그리고 @EnableZuulProxy를 포함하고 있다.**

[그림 9-96] SidecarApplication.java 작성

```
package egovframework.msa.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.sidecar.EnableSidecar;

@SpringBootApplication
@EnableSidecar
public class SidecarApplication {

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

}
```

- **사이드카 서버의 포트와 서버 이름을 설정한다.**

[그림 9-97] Sidecar의 Application.yml 파일 작성

```
server:
  port: 5678

spring:
  application:
    name: sidecar-nonJVM

sidecar:
  port: 5000
  health-uri: http://localhost:5000/health.json
  hostname: localhost   #sidecar와 비 JVM 애플리케이션은 동일 호스트에 있어야 함
```

- **비 JVM 애플리케이션의 http://localhost:5000/health.JSON을 요청하면, 비 JVM 애플리케이션에서 다음과 같은 상태를 반환하여 사이드카가 인식할 수 있도록 한다.**

[그림 9-98] 비 JVM 애플리케이션의 상태 설정

```
{
  "status": "UP"
}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.8 폴리글랏 지원 – 사이드카(Sidecar)

9.3.8.2 사이드카 설정

- **예를 들어 다음과 같은 파이썬 애플리케이션을 개발한다고 가정하자.**

[그림 9-99] 파이썬 애플리케이션 예시

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, World!"

@app.route("/health.json")
def health():
    return jsonify({"status": "UP"}), 200

if __name__ == "__main__":
    app.run(debug=True)
```

- **위 파이썬으로 작성한 프로그램을 다음과 같이 실행한다.**

[그림 9-100] 파이썬 프로그램 실행

```
⌥⌘1                                              hhjeong: python app.py

❯ python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 493-028-346
```

- **아래와 같이 서비스 상태 체크 요청을 하면 작동할 것이다.**

[그림 9-101] 서비스 상태 체크 요청

```
❯ curl -X GET http://localhost:5000/health.json
{
  "status": "UP"
}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.8 폴리글랏 지원 – 사이드카(Sidecar)

9.3.8.2 사이드카 설정

- **비 JVM 애플리케이션을 구동(health-check 활성)하고, 사이드카 애플리케이션을 실행하면, 유레카 서버에 아래와 같이 Sidecar-NonJVM이라는 애플리케이션이 등록된 것을 확인할 수 있다.**

[그림 9-102] 유레카 서버에서 Sidecar-NonJVM 애플리케이션 확인

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 192.168.100.158:catalog:8081
CUSTOMER	n/a (1)	(1)	UP (1) - 192.168.100.158:customer:8082
EUREKASERVER	n/a (1)	(1)	UP (1) - 192.168.100.158:EurekaServer:8761
SIDECAR-NONJVM	n/a (1)	(1)	UP (1) - 192.168.100.158:sidecar-nonJVM:5678
ZUUL	n/a (1)	(1)	UP (1) - 192.168.100.158:zuul:8080

클라우드 네이티브 정보시스템 구현·이행 단계

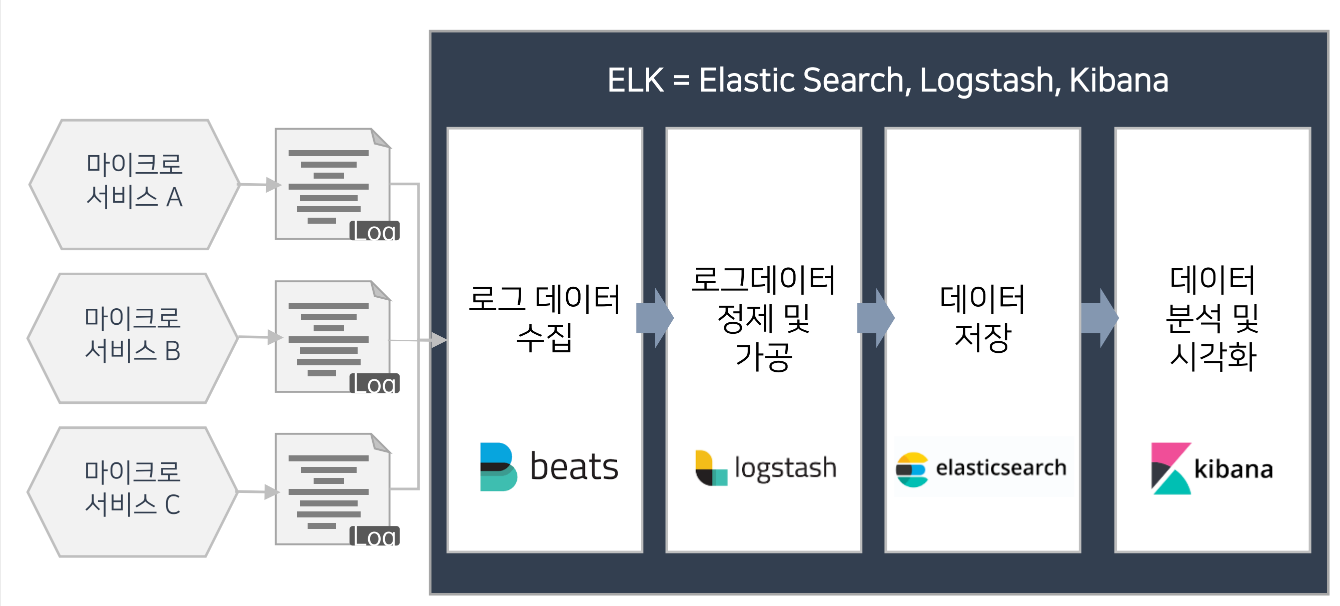
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana)

9.3.9.1 역할 및 주요 기능

- **분산 환경에서 각각의 마이크로서비스에 의해 생성된 로그들을 한 곳으로 취합하여 효율적으로 분석하기 위해 ELK 스택을 사용한다. 각 마이크로서비스에서 생성된 로그 파일들은 비츠(Beats)와 로그스태시 (Logstash)에서 수집하고 가공하여 일래스틱서치(Elasticsearch)로 전송하면 일래스틱서치의 데이터를 기반으로 키바나(Kibana)는 데이터를 분석하고 시각화하여 제공한다.**
- **ELK 스택의 각각의 역할 및 기능은 다음과 같다.**

[그림 9-103] ELK 스택의 역할 및 주요 기능



구분	주요 기능
비츠(Beats)	• 로그 데이터를 수집하는 경량화된 모듈 • 로그 파일, 메트릭, 네트워크 데이터, 윈도우 이벤트 로그, 가동시간 모니터링 정보 등 수립
로그스태시 (Logstash)	• 오픈소스 서버 측 데이터 처리 파이프라인으로, 다양한 소스에서 동시에 데이터를 수집하고 변환하여 로그스태시 보관소로 보냄 • 수집할 로그를 선정해서, 지정된 대상 서버(일래스틱서치)에 인덱싱하여 전송하는 역할을 담당
일래스틱서치 (ElasticSearch)	• Lucene 기반으로 개발한 분산 검색엔진 기반의 NoSQL 데이터베이스로, 로그스태시를 통해 수신된 데이터를 저장소에 저장하는 역할을 담당 • 데이터를 중심부에 저장하여 예상되는 항목을 검색하고 예상치 못한 항목을 밝혀낼 수 있음 • 정형, 비정형, 위치정보, 메트릭 등 원하는 방법으로 다양한 유형의 검색을 수행하고 결합할 수 있음
Kibana	• 일래스틱서치를 위한 시각화 및 관리 도구로서, 애플리케이션 로그를 모니터링할 수 있음 • 시각화를 담당하는 HTML + 자바스크립트 엔진의 개념

클라우드 네이티브 정보시스템 구현·이행 단계

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana)

9.3.9.2 일래스틱서치 설치 및 구동

- **일래스틱서치를 설치하기 위해 https://www.elastic.co/kr/downloads/elasticsearch에서 압축파일을 다운로드하여 압축을 푼 후 ./bin/elasticsearch 또는 .₩bin₩elasticsearch.bat를 실행한다. (Ctrl+C로 중지할 수 있음)**

[그림 9-104] Elastic Search 설치 및 구동

1. Elasticsearch 설치

```
[2021-02-11T18:21:40,727][WARN ][o.e.b.BootstrapChecks    ] [MacBook-Air.local] the default discovery settings are unsuit
able for production use; at least one of [discovery.seed_hosts, discovery.seed_providers, cluster.initial_master_nodes] m
ust be configured
[2021-02-11T18:21:40,731][INFO ][o.e.c.c.Coordinator      ] [MacBook-Air.local] cluster UUID [TrP4sL7CQN-cyrY_wSJ9rA]
[2021-02-11T18:21:40,744][INFO ][o.e.c.c.ClusterBootstrapService] [MacBook-Air.local] no discovery configuration found, w
ill perform best-effort cluster bootstrapping after [3s] unless existing master is discovered
[2021-02-11T18:21:40,989][INFO ][o.e.c.s.MasterService    ] [MacBook-Air.local] elected-as-master ([1] nodes joined)[{Mac
Book-Air.local}{R9Tm8EaiRfyVM-Jvol1uVA}{MYJb16wXSzaHK6QUDmvyKA}{127.0.0.1}{127.0.0.1:9300}{cdhilmrstw}{ml.machine_memory=
8589934592, xpack.installed=true, transform.node=true, ml.max_open_jobs=20, ml.max_jvm_size=4294967296} elect leader, _BE
COME_MASTER_TASK_, _FINISH_ELECTION_], term: 2, version: 76, delta: master node changed {previous [], current [{MacBook-A
ir.local}{R9Tm8EaiRfyVM-Jvol1uVA}{MYJb16wXSzaHK6QUDmvyKA}{127.0.0.1}{127.0.0.1:9300}{cdhilmrstw}{ml.machine_memory=858993
4592, xpack.installed=true, transform.node=true, ml.max_open_jobs=20, ml.max_jvm_size=4294967296}]}
[2021-02-11T18:21:41,141][INFO ][o.e.c.s.ClusterApplierService] [MacBook-Air.local] master node changed {previous [], cur
rent [{MacBook-Air.local}{R9Tm8EaiRfyVM-Jvol1uVA}{MYJb16wXSzaHK6QUDmvyKA}{127.0.0.1}{127.0.0.1:9300}{cdhilmrstw}{ml.machi
ne_memory=8589934592, xpack.installed=true, transform.node=true, ml.max_open_jobs=20, ml.max_jvm_size=4294967296}]}, term
: 2, version: 76, reason: Publication{term=2, version=76}
[2021-02-11T18:21:41,180][INFO ][o.e.h.AbstractHttpServerTransport] [MacBook-Air.local] publish_address {127.0.0.1:9200},
 bound_addresses {[::1]:9200}, {127.0.0.1:9200}
[2021-02-11T18:21:41,184][INFO ][o.e.n.Node               ] [MacBook-Air.local] started
[2021-02-11T18:21:41,546][INFO ][o.e.l.LicenseService     ] [MacBook-Air.local] license [24e9d692-6341-4667-b5b5-9db3e994
07fd] mode [basic] - valid
[2021-02-11T18:21:41,548][INFO ][o.e.x.s.s.SecurityStatusChangeListener] [MacBook-Air.local] Active license is now [BASIC
]; Security is disabled
[2021-02-11T18:21:41,553][INFO ][o.e.g.GatewayService     ] [MacBook-Air.local] recovered [6] indices into cluster_state
[2021-02-11T18:21:43,653][INFO ][o.e.c.r.a.AllocationService] [MacBook-Air.local] Cluster health status changed from [RED
] to [GREEN] (reason: [shards started [[.ds-ilm-history-5-2021.02.11-000001][0]]]).
```

2. Elasticsearch 구동

localhost:9200

localhost:9200


```
{
  "name" : "MacBook-Air.local",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "TrP4sL7CQN-cyrY_wSJ9rA",
  "version" : {
    "number" : "7.11.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "8ced7813d6f16d2ef30792e2fcde3e755795ee04",
    "build_date" : "2021-02-08T22:44:01.320463Z",
    "build_snapshot" : false,
    "lucene_version" : "8.7.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana)

9.3.9.3 키바나 설치 및 로깅 설정

- **키바나를 설치하기 위해 https://www.elastic.co/kr/downloads/kibana에서 압축파일을 다운로드하여 압축을 푼다.**
- **./config/kibana.yml 파일에서 elasticsearch.hots 부분의 주석을 풀고 저장한 후 ./bin/Elasticsearch 또는 .₩bin₩kibana.bat를 실행한다. (Ctrl+C 로 중지할 수 있음) 브라우저에서 localhost:5601로 확인할 수 있다.**

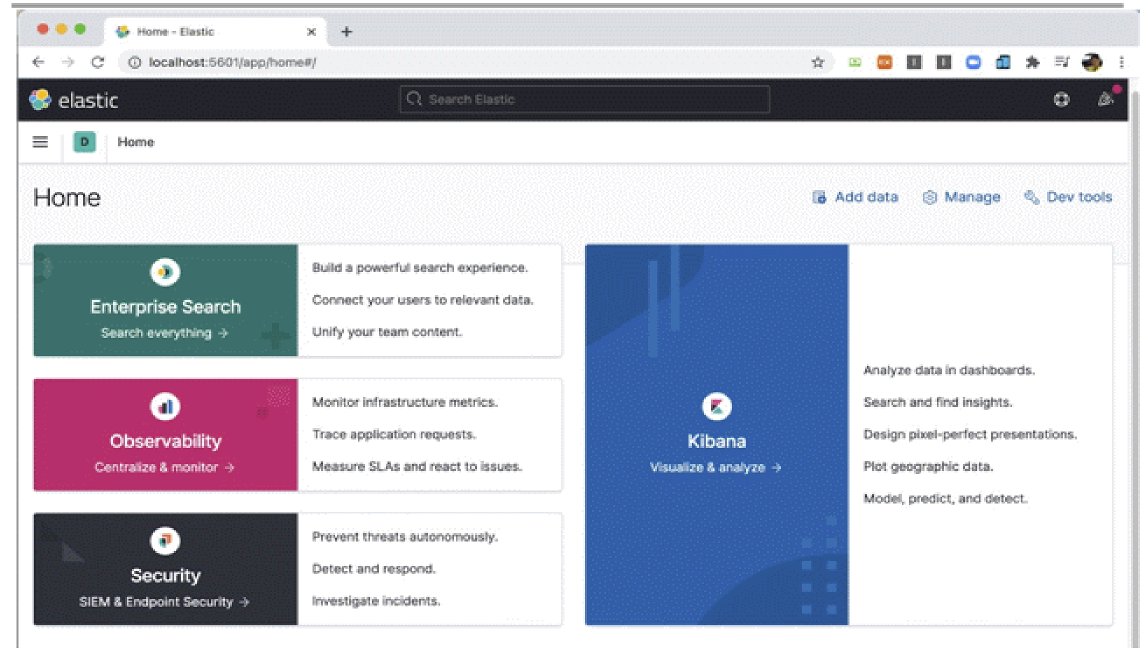
[그림 9-105] 키바나 설치 및 구동

1. 키바나 설치

```
]
  log   [18:22:01.658] [info][plugins][watcher] Your basic license does not support watcher. Please upgrade your license.
  log   [18:22:01.662] [info][crossClusterReplication][plugins] Your basic license does not support crossClusterReplicati
on. Please upgrade your license.
  log   [18:22:01.662] [info][kibana-monitoring][monitoring][monitoring][plugins] Starting monitoring stats collection
  log   [18:22:02.241] [info][listening] Server running at http://localhost:5601
  log   [18:22:02.
  log   [18:22:03.                                                                    rwin-x86_64/.chromium/chr
omium-ef768c9-darw                                                                    eporting/chromium/headles
s_shell-darwin_x64
```

```
# The Kibana server's name.  This is used for display purposes.
#server.name: "your-hostname"

# The URLs of the Elasticsearch instances to use for all your queries.
elasticsearch.hosts: ["http://localhost:9200"]
```

2. 키바나 설치 확인



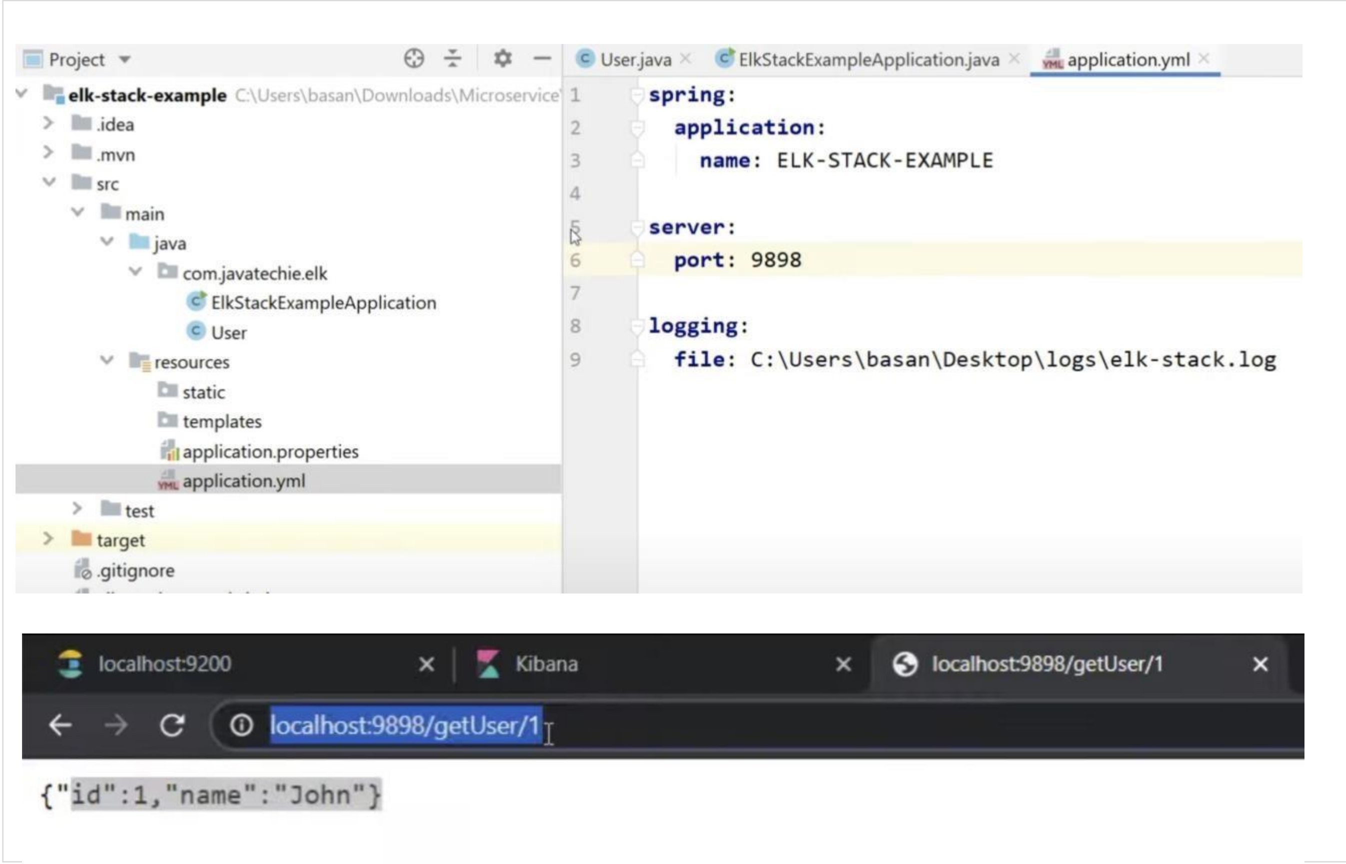
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana)

9.3.9.3 키바나 설치 및 로깅 설정

- **마이크로서비스 로그가 특정 폴더로 쌓이도록 코딩한다.**

[그림 9-106] 마이크로서비스 로깅 설정



9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana)

9.3.9.4 로그스태시 설치 및 로깅 설정

- **로그스태시를 설치하기 위해 https://www.elastic.co/kr/downloads/logstash에서 압축파일을 다운로드하여 압축을 푼 후 logstash.conf 파일을 다음과 같이 생성하여 ./bin/logstash.conf로 복사한다. Logstash.conf에서 로그 Input(Source)과 Output(Target)을 지정한다.**
- **그리고 https://www.elastic.co/kr/downloads/logstash에서 압축파일을 다운로드하여 압축을 푼 후 ./bin/logstash -f logstash.conf 또는 .₩bin₩logstash -f logstash.conf 를 실행한다.**

[그림 9-107] Logstash.conf 설정 수정 및 실행

1. Logstash.conf 설정 수정

new 1 | youtube.txt | new 4 | new 11 | test.gc | new 3 | new 2 | new 8 | new 5 | logstash.conf

```
input {
  file {
    path => "C:\Users\basan\Desktop\logs\elk-stack.log"
    start_position => "beginning"
  }
}


output {

  stdout {
    codec => rubydebug
  }

  # Sending properly parsed log events to elasticsearch
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}
```

2. Logstash.conf 실행

```
[2020-04-05T21:26:15,135][WARN ][logstash.outputs.elasticsearch][main] Restored connection to ES instance {:url=>"http://localhost:9200/"}
[2020-04-05T21:26:15,183][INFO ][logstash.outputs.elasticsearch][main] ES Output version determined {:es_version=>7}
[2020-04-05T21:26:15,190][WARN ][logstash.outputs.elasticsearch][main] Detected a 6.x and above cluster: the `type` event field won't be used to determine th
 document _type {:es_version=>7}
[2020-04-05T21:26:15,253][INFO ][logstash.outputs.elasticsearch][main] New Elasticsearch output {:class=>"LogStash::Outputs::ElasticSearch", :hosts=>["//loca
host:9200"]}
[2020-04-05T21:26:15,333][INFO ][logstash.outputs.elasticsearch][main] Using default mapping template
[2020-04-05T21:26:15,389][WARN ][org.logstash.instrument.metrics.gauge.LazyDelegatingGauge][main] A gauge metric of an unknown type (org.jruby.specialized.Ru
yArrayOneObject) has been created for key: cluster_uuids. This may result in invalid serialization.  It is recommended to log an issue to the responsible dev
loper/development team.
[2020-04-05T21:26:15,399][INFO ][logstash.javapipeline    ][main] Starting pipeline {:pipeline_id=>"main", "pipeline.workers"=>8, "pipeline.batch.size"=>125,
"pipeline.batch.delay"=>50, "pipeline.max_inflight"=>1000, "pipeline.sources"=>["F:/Softwares/ELK/logstash-7.6.2/bin/logstash.conf"], :thread=>"#<Thread:0x6b
4cb9 run>"}
[2020-04-05T21:26:15,451][INFO ][logstash.outputs.elasticsearch][main] Attempting to install template {:manage_template=>{"index_patterns"=>"logstash-*", "ve
sion"=>60001, "settings"=>{"index.refresh_interval"=>"5s", "number_of_shards"=>1, "index.lifecycle.name"=>"logstash-policy", "index.lifecycle.rollover_alias"
>"logstash"}, "mappings"=>{"dynamic_templates"=>[{"message_field"=>{"path_match"=>"message", "match_mapping_type"=>"string", "mapping"=>{"type"=>"text", "nor
s"=>false}}}, {"string_fields"=>{"match"=>"*", "match_mapping_type"=>"string", "mapping"=>{"type"=>"text", "norms"=>false, "fields"=>{"keyword"=>{"type"=>"ke
word", "ignore_above"=>256}}}}}], "properties"=>{"@timestamp"=>{"type"=>"date"}, "@version"=>{"type"=>"keyword"}, "geoip"=>{"dynamic"=>true, "properties"=>{"
p"=>{"type"=>"ip"}, "location"=>{"type"=>"geo_point"}, "latitude"=>{"type"=>"half_float"}, "longitude"=>{"type"=>"half_float"}}}}}}}
[2020-04-05T21:26:17,073][INFO ][logstash.inputs.file     ][main] No sincedb_path set, generating one based on the "path" setting {:sincedb_path=>"F:/Softwar
s/ELK/logstash-7.6.2/data/plugins/inputs/file/.sincedb_e6beb8e7fcec57e3fd8b4bc00476fc57", :path=>["C:/Users/basan/Desktop/logs/elk-stack.log"]}
[2020-04-05T21:26:17,113][INFO ][logstash.javapipeline    ][main] Pipeline started {"pipeline.id"=>"main"}
[2020-04-05T21:26:17,186][INFO ][filewatch.observingtail  ][main] START, creating Discoverer, Watch with file and sincedb collections
[2020-04-05T21:26:17,193][INFO ][logstash.agent           ] Pipelines running {:count=>1, :running_pipelines=>[:main], :non_running_pipelines=>[]}
[2020-04-05T21:26:17,546][INFO ][logstash.agent           ] Successfully started Logstash API endpoint {:port=>9600}
```

09 클라우드 네이티브 정보시스템 구현·이행 단계

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

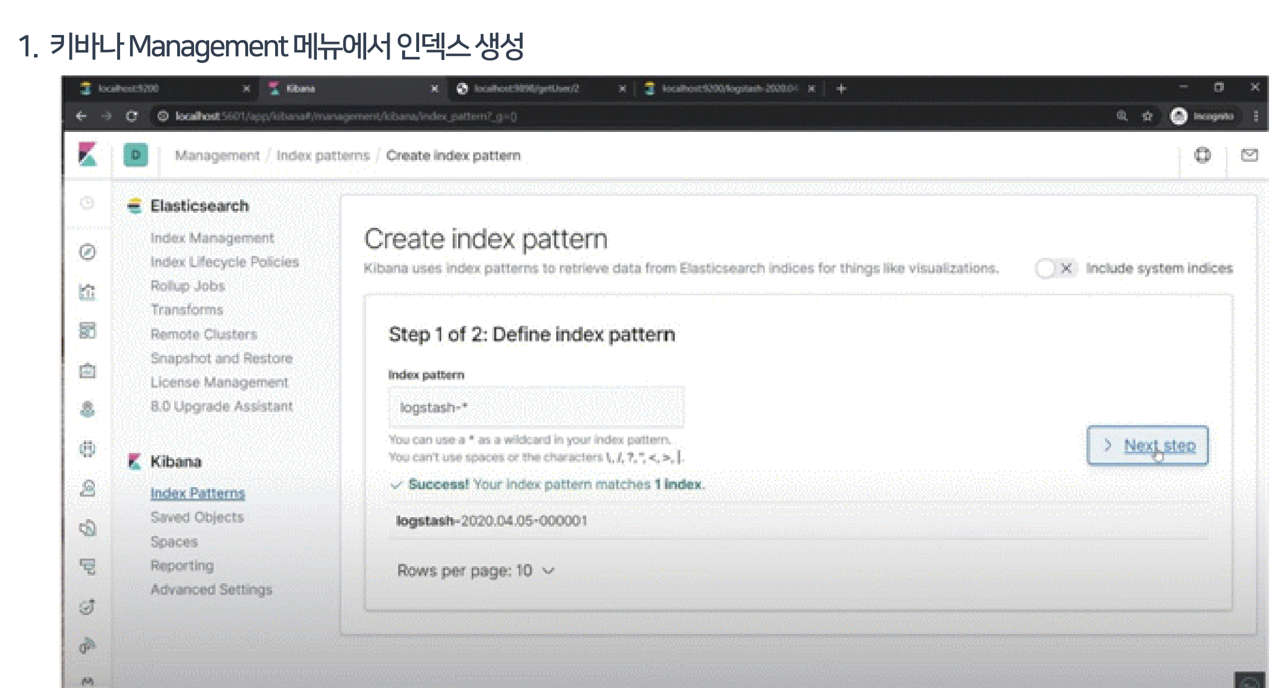
9.3.9 중앙집중식 로깅 – ELK(Elastic Search, Logstash, Kibana)

9.3.9.4 로그스태시 설치 및 로깅 설정

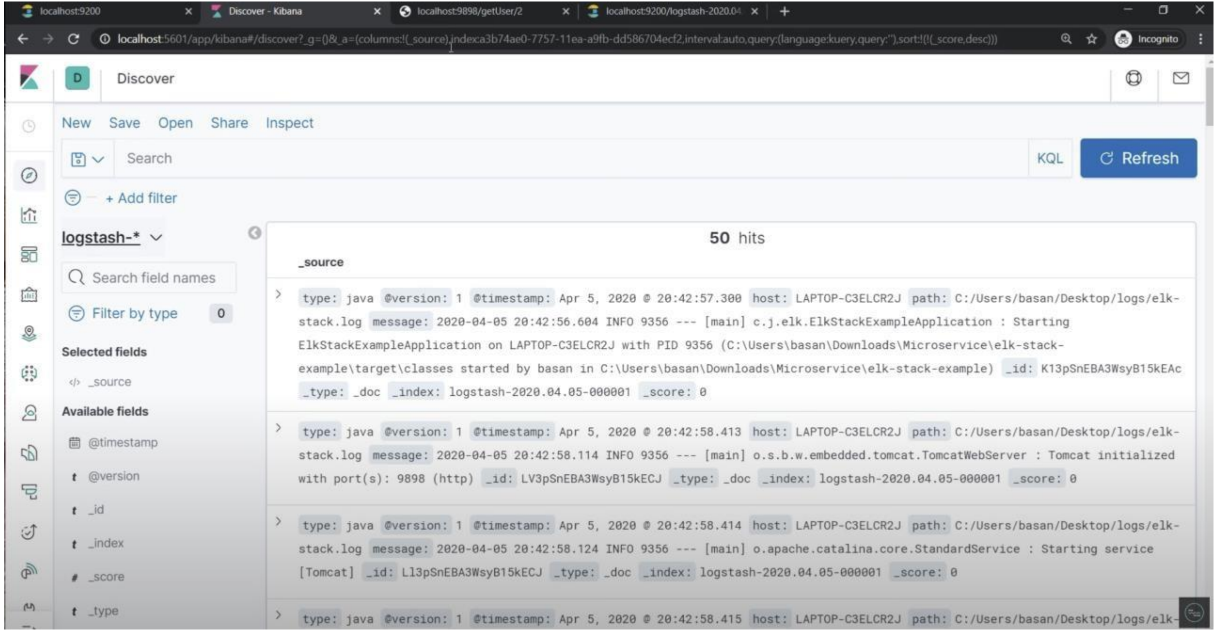
- **키바나의 Management 메뉴에서 Index를 생성한다.**

[그림 9-108] 키바나에서 Index 생성 후 로그 확인

1. 키바나 Management 메뉴에서 인덱스 생성



2. 키바나 Discover 메뉴에서 로그 확인



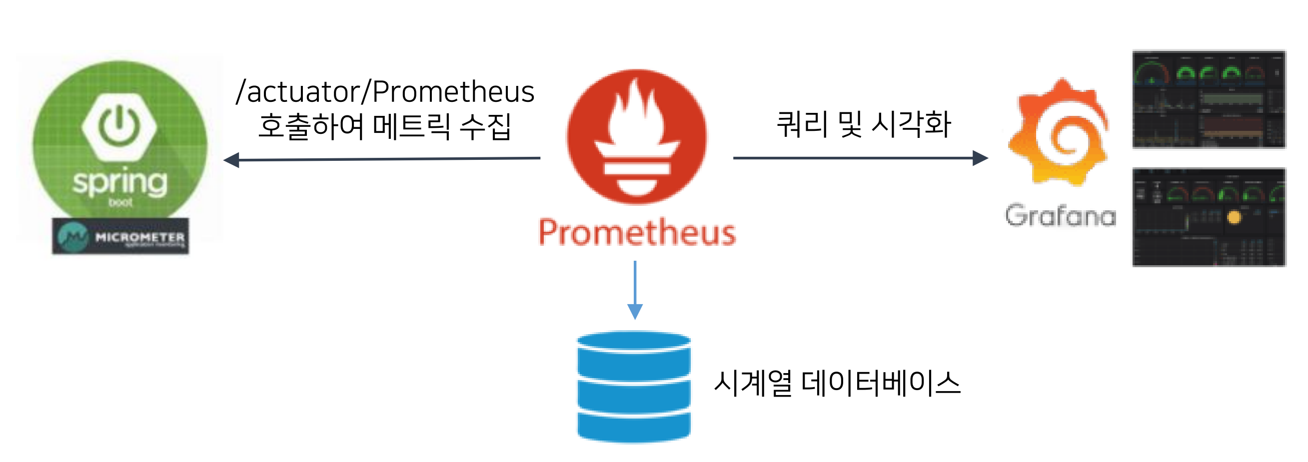
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.1 역할 및 주요 기능

- **시스템의 성능과 최적화 작동 여부를 확인 및 유지관리하기 위해 애플리케이션의 메트릭(Metric, 통계) 정보와 상태를 모니터링할 필요가 있다.**
- **스프링 부트 액추에이터는 마이크로미터(Micrometer)를 사용하여 애플리케이션의 메트릭 정보를 제공한다. 마이크로미터를 통해 노출된 애플리케이션의 메트릭 정보를 프로메테우스(Prometheus)를 사용하여 저장하고, 그라파나(Grafana)를 활용하여 메트릭 정보를 그래프로 시각화한다.**

[그림 9-109] 마이크로미터, 프로메테우스, 그라파나를 활용한 통계 수집 및 모니터링



구분	주요 기능
마이크로미터 (Micrometer)	• JVM 기반의 애플리케이션의 메트릭을 제공 • 스프링 부트 2.0부터 마이크로미터를 지원
프로메테우스 (Prometheus)	• 메트릭을 수집, 저장하고 모니터링할 수 있는 오픈소스 도구 • 시계열 데이터베이스에 실시간 메트릭을 저장하고 쿼리를 사용하여 메트릭을 조회할 수 있음
그라파나 (Grafana)	• 데이터 시각화, 모니터링 및 분석을 위한 오픈소스 도구 • 시계열 매트릭 데이터를 시각화하는 데 가장 최적화된 대시보드를 제공 • 사용자는 그라파나에서 패널(panel)을 사용해 설정된 기간 동안 특정 메트릭을 나타내는 대시보드를 만들 수 있음

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.2 메트릭 수집 및 모니터링 환경 설정 절차

- **Pom.xml에 다음과 같이 의존성을 추가한다.**

[그림 9-110] pom.xml에 의존성 추가

```
<!-- 메트릭을 Endpoint로 노출 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!-- Micrometer core dependecy  -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
</dependency>
<!-- Micrometer Prometheus registry  -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

- **Application.properties에 모든 정보를 노출하려면 다음과 같이 설정한다. management.server.port를 설정하지 않으면, 애플리케이션과 동일한 8080이 기본이 된다.**

[그림 9-111] Application.properties에 포트 설정

```
management.endpoint.metrics.enabled=true
management.endpoints.web.exposure.include=*
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
```

- **management.endpoints.web.exposure.include를 사용하여 활성화하고자 하는 엔드포인트들의 id를 작성할 수 있다. Management.endpoints.web.base-path와 management.endpoints.web.path- mapping.<id>값을 수정하여, 특정 id의 엔드포인트 경로를 수정할 수 있다. 또한 외부 도메인에서 액추에이터 정보 요청을 허용할 수 있다.**

[그림 9-112] endpoint 활성화 id 작성 및 actuator 정보 요청 허용

1. 엔드포인트 활성화 id 작성

```
management.endpoints.web.exposure.include=prometheus,health,info,metric
```

2. 액추에이터 정보 요청 허용

```
management.endpoints.web.base-path=/monitor
management.endpoints.web.path-mapping.health=healthcheck
management.endpoints.web.cors.allowed-origins=http://other-domain.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.2 메트릭 수집 및 모니터링 환경 설정 절차

- **애플리케이션을 실행하고, http://localhost:8080/actuator/health로 서비스가 실행 중이면 다음과 같은 측정값을 제공한다. 상태(Status)가 'UP'이면 건강하다는 의미이다.**

[그림 9-113] 서비스의 측정값 확인

```
▶ http GET "http://localhost:8080/actuator/health"
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/vnd.spring-boot.actuator.v3+json
Date: Wed, 21 Oct 2020 18:11:35 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "status": "UP"
}
```

- **http://localhost:8080/actuator를 통해 액추에이터가 제공하는 엔드포인트들을 확인할 수 있으며, 프로메테우스 메트릭(Prometheus Metric) 노출 엔드포인트가 존재하는지 확인한다.**

[그림 9-114] 프로메테우스 메트릭 노출 엔드포인트 존재 확인

```
,"prometheus":{"href":"http://localhost:8080/actuator/prometheus","templated":false},"
```

- **http://localhost:8080/actuator/prometheus로 프로메테우스로 수집할 메트릭 정보가 노출되는지 확인 할 수 있다.**

[그림 9-115] Prometheus Metric 노출 endpoint 존재 확인

```
# HELP system_cpu_usage The "recent cpu usage" for the whole system
# TYPE system_cpu_usage gauge
system_cpu_usage 0.08039035510719317
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed fc
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes{area="heap",id="PS Survivor Space",} 1.1010048E7
jvm_memory_committed_bytes{area="heap",id="PS Old Gen",} 1.89792256E8
jvm_memory_committed_bytes{area="heap",id="PS Eden Space",} 1.92413696E8
jvm_memory_committed_bytes{area="nonheap",id="Metaspace",} 4.063232E7
jvm_memory_committed_bytes{area="nonheap",id="Code Cache",} 8847360.0
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space",} 5767168.0
# HELP logback_events_total Number of error level events that made it to the logs
# TYPE logback_events_total counter
logback_events_total{level="warn",} 0.0
logback_events_total{level="debug",} 0.0
logback_events_total{level="error",} 0.0
logback_events_total{level="trace",} 0.0
logback_events_total{level="info",} 7.0
# HELP tomcat_servlet_request_max_seconds
# TYPE tomcat_servlet_request_max_seconds gauge
tomcat_servlet_request_max_seconds{name="default",} 0.0
tomcat_servlet_request_max_seconds{name="dispatcherServlet",} 0.083
# HELP tomcat_threads_current_threads
# TYPE tomcat_threads_current_threads gauge
tomcat_threads_current_threads{name="http-nio-8080",} 10.0
# HELP tomcat_sessions_rejected_sessions_total
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.2 메트릭 수집 및 모니터링 환경 설정 절차

- **http://localhost:8080/actuator/metrics에서 프로메테우스로 수집할 memory, heap, prosessors, threads, classes, thread pools 등의 메트릭정보가 노출되는지 확인할 수 있다.**

[그림 9-116] Prometheus 수집할 Metric 정보 확인

```
@Service
public class LoginServiceImpl {

    private final CounterService counterService;

    public LoginServiceImpl(CounterService counterService) {
        this.counterService = counterService;
    }

    public boolean login(String userName, char[] password) {
        boolean success;
        if (userName.equals("admin") && "secret".toCharArray().equals(passwor
d)) {
            counterService.increment("counter.login.success");
            success = true;
        }
        else {
            counterService.increment("counter.login.failure");
            success = false;
        }
        return success;
    }
}
```

```
{
    ...
    "counter.login.success" : 105,
    "counter.login.failure" : 12,
    ...
}
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.2 메트릭 수집 및 모니터링 환경 설정 절차

▪ **액추에이터에서 제공할 수 있는 엔드포인트들은 다음과 같다.**

[표 9-15] 액추에이터의 엔드포인트

엔드포인트 ID	설명
auditevents	• 현 애플리케이션에 대한 Audit Event 정보를 표시하고, AuditEvenRepository bean을 필요로 함
beans	• 애플리케이션의 모든 bean의 목록을 표시
caches	• 캐시를 표시
conditions	• 구성 및 자동 구성 클래스에서의 조건과 일치 또는 불일치 이유를 표시
configroups	• @ConfigurationProperties의 집합 목록을 표시
env	• Spring의 ConfigurableEnvironment의 환경 변수 값을 표시 • /env/{name}을 통해 특정 변수의 값을 조회
flyway	• 적용된 Flyway 데이터베이스 마이그레이션을 표시하고, 하나 이상의 Flyway bean을 필요로 함
health	• 애플리케이션의 상태(health) 정보를 표시
httptrace	• HTTP trace 정보를 표시하고, HttpTraceRepository bean을 필요로 함
info	• 애플리케이션의 정보를 표시
integrationgraph	• Spring의 통합 그래프를 표시하며, spring-integration-core 의존성이 필요함
loggers	• 애플리케이션의 loggers 구성을 표시하고 수정함
liquibase	• 적용된 모든 Liquibase 데이터베이스 마이그레이션을 표시하고, 하나 이상의 Liquibase bean을 필요로 함
metrics	• 애플리케이션의 메트릭 정보를 표시(heap, class, gc, thread 등) • /metrics/{name}을 통해 개별 메트릭을 확인
mappings	• @RequestMapping 경로와 컨트롤러 매핑 정보를 표시
scheduledtasks	• 애플리케이션의 예약된 태스크를 표시
sessions	• 스프링 세션 저장소에서 사용자 세션을 검색 및 삭제할 수 있음
shutdown	• 애플리케이션을 안전하게 종료할 수 있고, 기본값은 disabled임
startup	• ApplicationStartup에서 수집한 시작단계 데이터를 표시함
threaddump	• Thread dump를 수행
heapdump	• 힙 덤프 파일을 반환
jolokia	• Jolokia가 클래스 경로에 있을 때 HTTP를 통해 JMX 빈을 노출
logfile	• 로그파일의 내용을 리턴
prometheus	• Prometheus 서버로 부터 스크랩될 수 있는 형태로 메트릭 정보를 노출 • Micrometer-registry-Prometheus 종속성이 필요함

[참고 : https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html]

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.3 프로메테우스 설치 및 설정

- **프로메테우스를 설치하기 위해 https://Prometheus.io/download/에서 다운로드 후 압축을 해제한다.**
- **프로메테우스가 액추에이터 엔드포인트에 접근할 수 있도록 Prometheus.yml에 아래와 같이 추가한다. Scrape_interva은 액추에이터에 접근하여 데이터를 가져오는 간격이다. Scrape_interval: 15s로 설정하면, 15초마다 프로메테우스가 액추에이터에 접근하여 정보를 가져온다. Targets 속성은 프로메테우스가 바라보는 서버 주소이다.**

[그림 9-118] Prometheus.yml 에 Scrape interval 설정

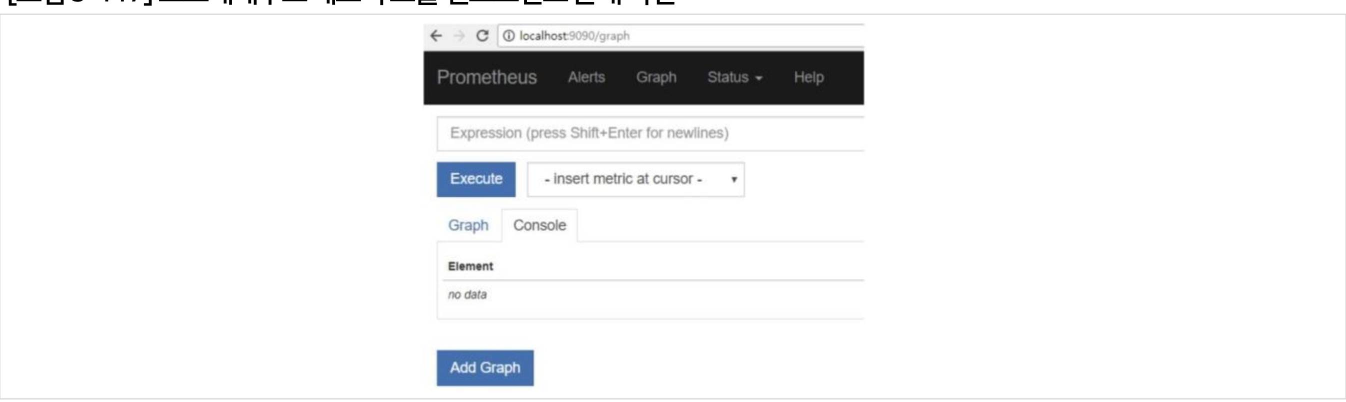
```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'cAdvisor'
    scrape_interval: 15s
    static_configs:
    - targets: ['cadvisor:8080']

...

  - job_name: '...'
    scrape_interval: 15s
    metrics_path: '/actuator/prometheus'
    static_configs:
    - targets: ['[container_name or ip]:[port]']
```

- **윈도우라면, Prometheus.exe를 실행하고, 리눅스의 경우는 다음과 같이 실행한다.**

  ./Prometheus --config.file="Prometheus.yml"

- **웹브라우저에서 http://localhost:9090/graph로 확인한다.**

[그림 9-117] 프로메테우스 메트릭 노출 엔드포인트 존재 확인



9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.4 그라파나를 이용한 프로메테우스 시각화

- **다음 URL을 참고하여 그라파나를 다운로드 및 설치한다.**

  https://grafana.com/docs/grafana/latest/installation/windows/

  s://grafana.com/docs/grafana/latest/installation/mac/

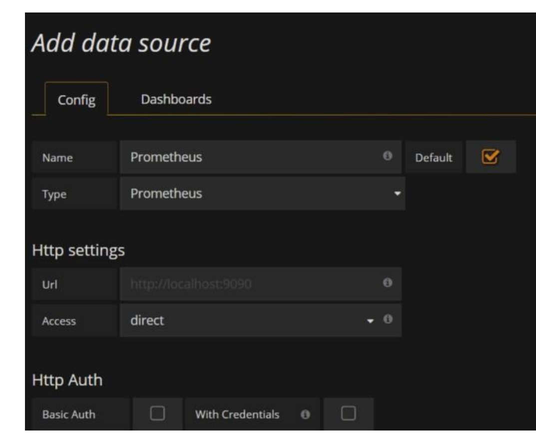
- **그라파나를 실행하고 접속한다.**

  http://localhost:3000/에 접속한다.

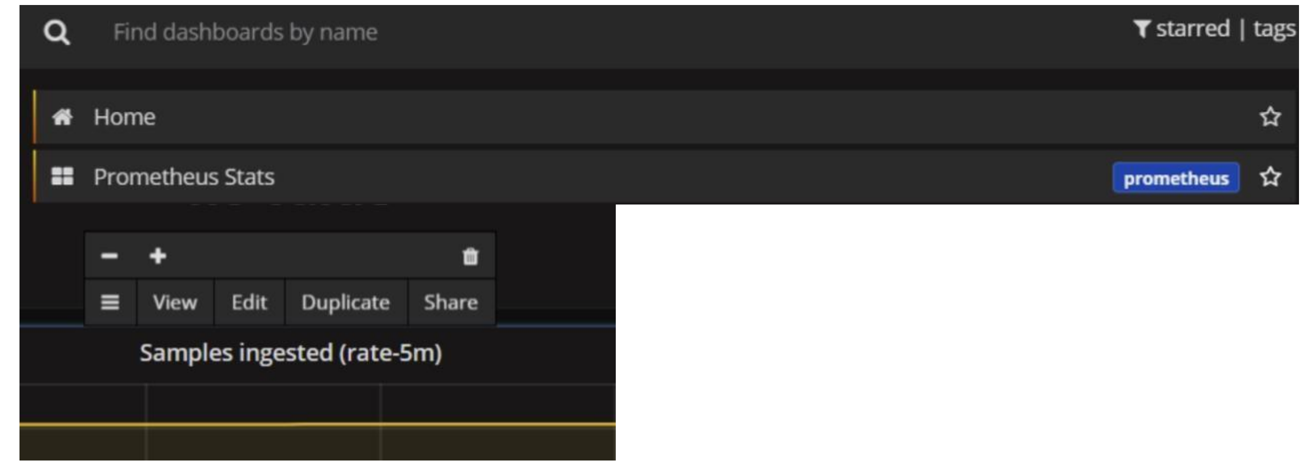
- **프로메테우스 설정을 위해 Add data source에서 프로메테우스를 추가한다. 그리고 추가된 프로메테우스를 클릭하여 그래프를 설정한다.**

[그림 9-119] 프로메테우스 설정 추가 및 그래프 설정

1. 프로메테우스 설정 추가



2. 그래프 설정



9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.10 중앙집중식 메트릭 – 프로메테우스 & 그라파나(Prometheus, Grafana)

9.3.10.4 그라파나를 이용한 프로메테우스 시각화

- **다음은 프로메테우스를 통한 시각화의 예시이다.**

[그림 9-120] 프로메테우스를 통한 시각화

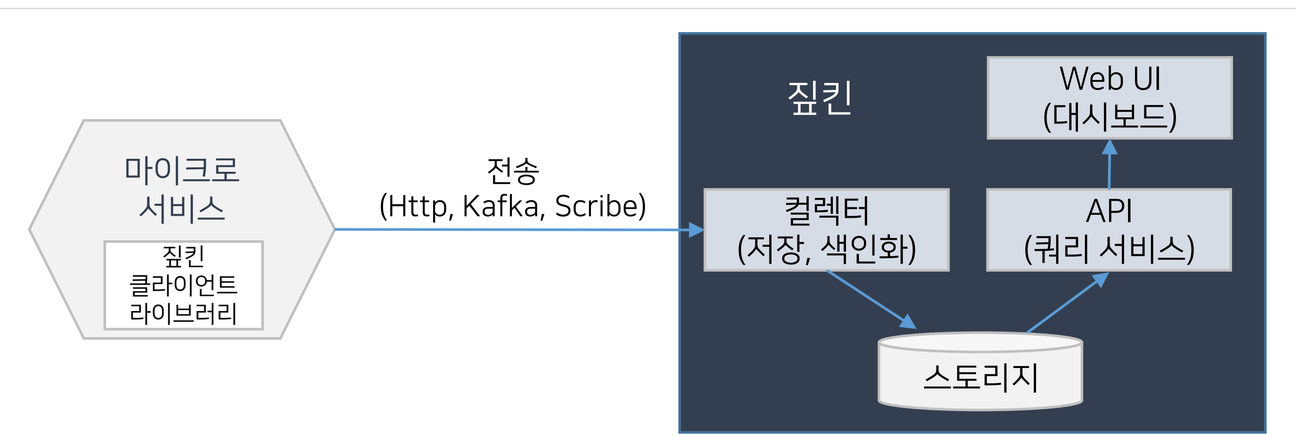
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.11 분산 서비스 로그 추적-쓸루스 & 짚킨(Sleuth, Zipkin)

9.3.11.1 역할 및 주요 기능

- **마이크로서비스의 중요한 도전과제는 이슈를 디버깅하고 모니터링하는 능력이다. 각 마이크로서비스는 독립적인 환경에서 실행되기 때문에 DB와 로그 파일과 같은 리소스를 공유할 수 없기 때문이다. 분산 환경에서의 서비스 간 병목이 발생할 때 기존 방식의 모니터링으로는 추적이 불가능하기 때문에 분산 환경의 트래픽 추적이 필요하다.**
- **스프링 부트와 스프링 클라우드 환경에서 분산 추적을 위해 활용되는 도구는 스프링 클라우드 쓸루스(Sleuth)와 짚킨(Zipkin)이다.**

[그림 9-121] 짚킨 구조도



- 짚킨 클라이언트 라이브러리 : 마이크로서비스에서 트레이스 정보 수집
- 컬렉터(Collector) : 전송된 트레이스 정보 유효성 검증, 검색을 위한 저장 및 색인화
- API(Zipkin Query Service) : 스토리지에 저장된 트레이스 정보를 검색하기 위한 JSON API
- Web UI : 트레이스 정보를 대시보드의 형태로 제공

[그림 9-122] 스프링 클라우드 쓸루스 개념

- 스프링 클라우드 쓸루스는 스프링에서 공식적으로 지원하는 짚킨 클라이언트 라이브러리로 스프링과의 연동이 쉬우며, 호출되는 서비스에 Trace(추적) ID와 Span(구간) ID를 부여
- Trace ID는 클라이언트 호출의 시작부터 끝날 때까지 동일한 ID로 처리되며, Span ID는 마이크로서비스 당 1개의 ID가 부여됨
- Trace ID와 Span ID를 활용하여 클라이언트 호출을 쉽게 추적함

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.11 분산 서비스 로그 추적- 쓸루스 & 집킨(Sleuth, Zipkin)

9.3.11.2 집킨 환경 설정

- **zipkin.io 최신 도커이미지를 다음과 같이 실행할 수 있다.**

```
$ docker run -d -p 9411:9411 openzipkin/zipkin
```

- **집킨 서버를 mvnrepository.com/artifact/io.zipkin.java/zipkin-server에서 직접 다운로드 받을 수 있다. 다운로드 받은 집킨 서버를 다음과 같이 기동한다.**

```
$ java -jar zipkin-server-2.12.9-exec.jar
```

- **http://localhost:9411/로 접속하면 아래와 같이 집킨 대시보드를 볼 수 있다.**

[그림 9-123] 집킨 대시보드

```
chrisyoon  ~/Downloads  java -jar zipkin-server-2.12.9-exec.jar

:: Powered by Spring Boot ::
```

- **http://localhost:9411/로 접속하면 아래와 같이 집킨 대시보드를 볼 수 있다.**

[그림 9-124] 마이크로서비스들의 pom.xml에 의존성 설정

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.11 분산 서비스 로그 추적-쓸루스 & 짚킨(Sleuth, Zipkin)

9.3.11.2 짚킨 환경 설정

- **마이크로서비스들의 application.xml에는 다음과 같이 짚킨 서버의 기본 base-url을 지정한다.**

[그림 9-125] Zipkin 서버의 base URL 지정

```
spring:
  zipkin:
    base-url: http://localhost:9411/
```

- **마이크로서비스들을 호출하면, 다음과 같이 로그에 Service Name, Trace Id, Span Id, Export Flag 를 확인할 수 있다. A 마이크로서비스에서 B 마이크로서비스를 호출하는 구조라 할 때 A 마이크로서비스를 호출하면 Trace Id와 Span Id는 동일하며, B 마이크로서비스 로그에서는 Trace Id는 동일하지만, Span Id는 마이크로서비스별로 다름을 확인할 수 있다.**
- **마이크로서비스 A와 B의 로그를 다음과 같이 확인한다.**

[그림 9-126] 마이크로서비스의 로그 확인

마이크로서비스 A의 로그

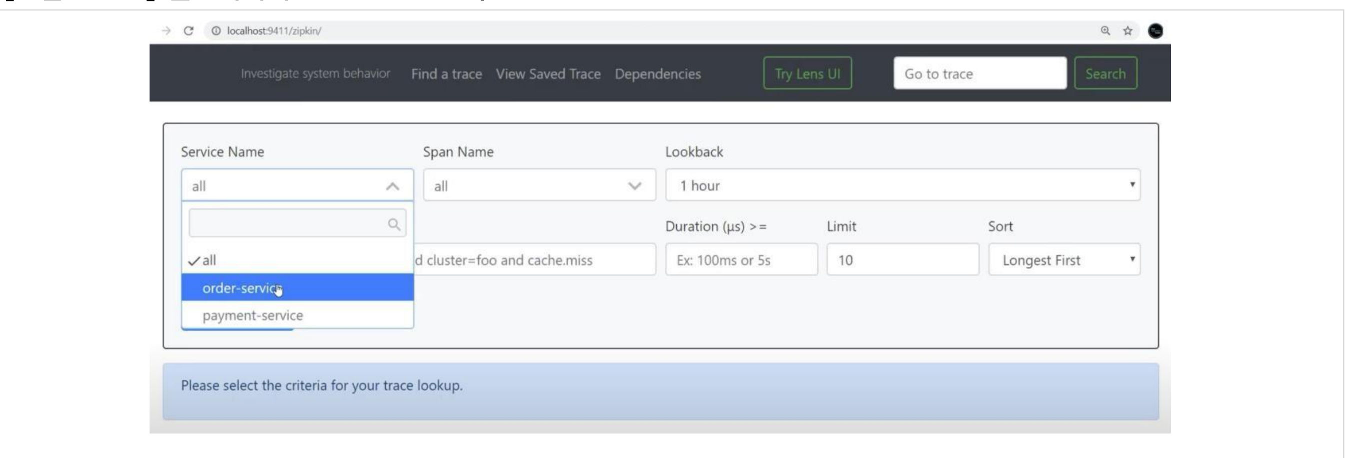
```
2020-04-20 17:15:42.052  INFO [ORDER-SERVICE,a3b13aeb479fdebf,a3b13aeb479fdebf,true] 14036 ---
2020-04-20 17:15:42.062  INFO [ORDER-SERVICE,a3b13aeb479fdebf,a3b13aeb479fdebf,true] 14036 ---
```

마이크로서비스 B의 로그

```
2020-04-20 17:15:42.057  INFO [PAYMENT-SERVICE,a3b13aeb479fdebf,45c1505517ebca81,true] 15632
```

- **짚킨 서버의 Service Name에 위의 마이크로서비스명을 확인한다.**

[그림 9-127] 짚킨서버의 Service Name 확인



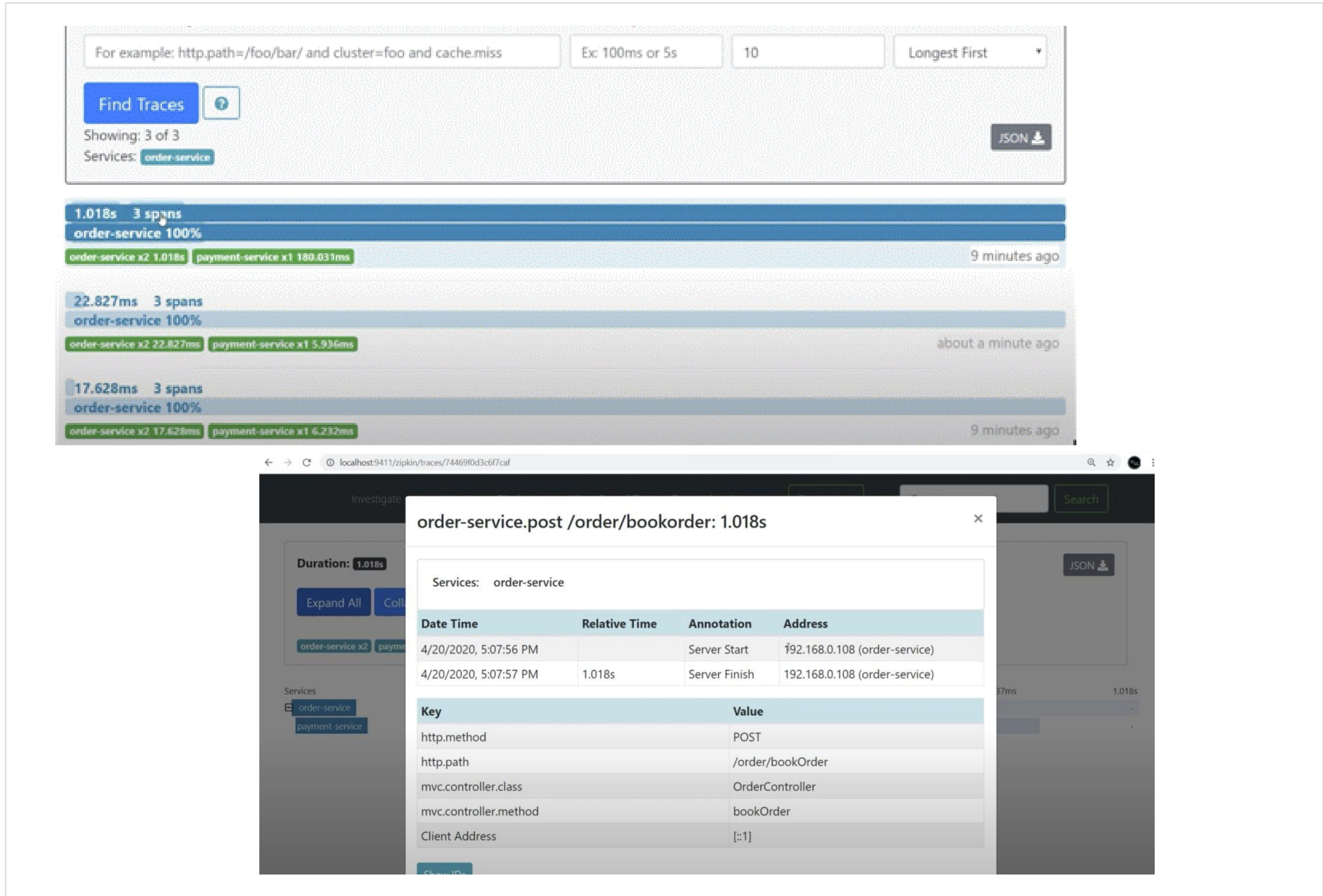
9.3 스프링 클라우드 기반 마이크로서비스 아키텍처 구축

9.3.11 분산 서비스 로그 추적- 슬루스 & 집킨(Sleuth, Zipkin)

9.3.11.2 집킨 환경 설정

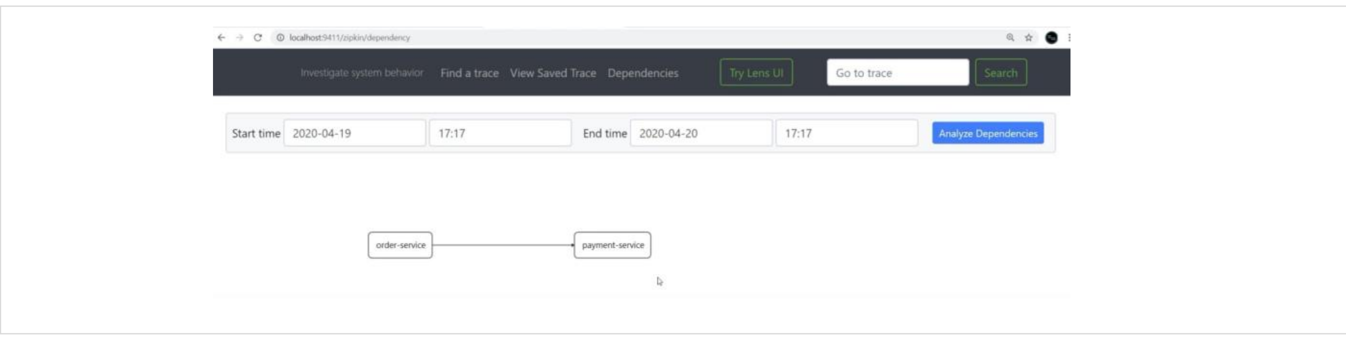
- **Service Name을 선택한 후 [Find Traces] 버튼을 클릭하면, 다음과 같이 호출 관계를 추적할 수 있게 된다. Spans 링크를 클릭하면 서비스별 호출 이력을 알 수 있다.**

[그림 9-128] 호출 관계 확인



- **Dependencies 메뉴에서 마이크로서비스별 의존 관계를 표시해 준다.**

[그림 9-129] 마이크로서비스별 의존 관계 표시



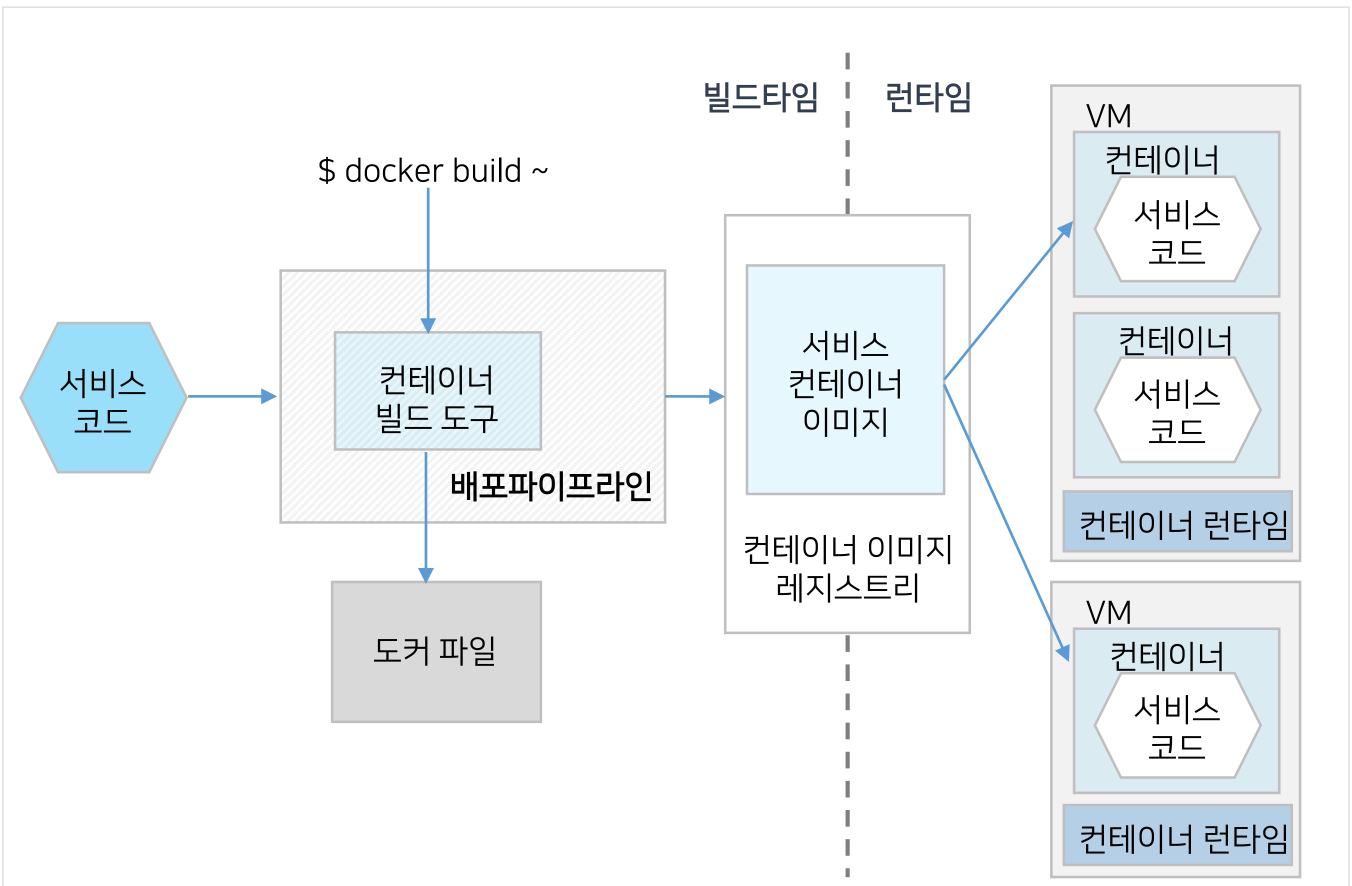
클라우드 네이티브 정보시스템 구현·이행 단계

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.1 도커 컨테이너 기반 빌드·배포

- **클라우드 네이티브 환경에서는 OS 수준에서 가상화한 컨테이너를 통해 배포를 하게 된다. 컨테이너는 다른 컨테이너와 격리된 환경에서 1개 이상의 서비스를 실행한다.**
- **컨테이너를 이용한 서비스 배포 과정은 다음과 같다.**
  - **빌드 타임에 컨테이너 이미지 빌드 도구로 서비스 코드 및 이미지 디스크립션(Description)을 읽고 컨테이너 이미지를 생성한 후 레지스트리에 보관**
  - **런타임에는 레지스트리에서 컨테이너 이미지를 가져와 컨테이너를 생성**

[그림 9-130] 컨테이너를 이용한 배포 과정



- 컨테이너 이미지로 패키징된 서비스는 레지스트리에 저장함
- 저장된 이미지를 토대로 런타임에 여러 컨테이너가 인스턴스를 만들어 서비스를 구성함
- 컨테이너는 대부분 가상머신에서 실행되고 하나의 VM이 여러 컨테이너를 실행함

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.2 쿠버네티스 기반 배포

- **도커 컴포즈는 개발 및 테스트 환경에서 유용한 도구이지만, 운영 환경에서 컨테이너로 묶은 서비스를 확실하게 실행하기 위해 좀 더 정교한 쿠버네티스와 같은 도커 오케스트레이션 프레임워크를 활용할 수 있다.**

[그림 9-131] 쿠버네티스 아키텍처

쿠버네티스 마스터
스케줄러
Kubectl CLI
API서버
컨트롤러 관리
개발자
배포 파이프라인
etcd
사용자
쿠버네티스 노드
큐블릿
큐브 프록시
파드
서비스 코드
쿠버네티스 노드
큐블릿
큐브 프록시
파드
서비스 코드

- 쿠버네티스 클러스터는 클러스터를 관리하는 마스터와 서비스를 실행하는 노드로 구성
- 개발자, 배포 파이프라인은 API 서버를 통해 쿠버네티스와 상호 작용
- 애플리케이션 컨테이너는 노드에서 실행되며, 각 노드는 애플리케이션 컨테이너를 관리하는 큐블릿과 애플리케이션 요청을 파드로 라우팅하는 큐브 프록시를 실행

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.1 도커 이미지 생성

- **도커 파일을 생성하고, 도커 이미지를 빌드하는 과정은 다음과 같다.**

[그림 9-132] 도커 파일 생성 및 도커 이미지 빌드

구분	설명	예시
도커 파일 생성	FROM • 도커 이미지의 바탕이 될 베이스 이미지를 지정 • FROM에서 받아오는 도커 이미지는 기본적으로 도커 허브 레지스트리에서 참조 • 각 이미지의 버전을 구분하는 식별자로 태그가 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	$ vi Dockerfile FROM openjdk:8-jre-alpine RUN mkdir /apps COPY hello-0.0.1-SNAPSHOT.jar /apps/hello.jar CMD ["java","-jar","/apps/hello.jar"]
도커 이미지 빌드	• docker build 명령으로 빌드 • -t 옵션으로 이미지명[:태그명]을 지정 • 도커 파일 위치를 지정 • docker images 명령으로 생성 이미지를 조회	$ docker build –t exam/hello:latest . $ docker images

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.2 도커 컨테이너 실행

- **도커 컨테이너를 실행하고, 포트를 포워딩하고, 도커 컨테이너를 조회하고 종료하는 과정은 다음과 같다.**

[그림 9-133] 도커 컨테이너 실행

단계	설명	명령

도커 컨테이너 실행	• docker run 명령으로 실행
	• 백그라운드 처리시 -d 옵션으로 실행
		$ docker run exam/hello:latest
		$ docker rm $(docker ps -aq) -f
		$ docker run -d exam/hello:latest

포트 포워딩	• 컨테이너 외부의 요청을 컨테이너 내부로 전달 • 포트 포워딩 설정 시 -p 옵션으로 실행 • -p 호스트포트:컨테이너포트	$ docker run -d -p 8080:8080 -- name hello exam/hello:latest
도커 컨테이너 조회	• docker ps 명령으로 실행한 도커 컨테이너의 상태 조회	$ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORT NAMES
도커 컨테이너 종료	• docker stop 명령으로 실행 • 종료할 컨테이너ID를 지정	$ docker stop $(docker ps --filter “name=hello” -q)

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.3 도커 이미지 등록

- **도커 파일을 작성한 후 도커 레지스트리에 접속하고, 도커 이미지를 빌드하여 생성한 이미지를 도커 허브에 등록한다.**

[그림 9-134] 도커 이미지 등록

단계	설명	명령어
도커 파일 작성	• 작업 경로에서 디렉토리 1개 생성 후 해당 디렉토리에 도커 파일을 생성	
도커 레지스트리 접속	• 도커 레지스트리에 접속함	$ docker login https://레지스트리주소 Username Password
도커 이미지 빌드	• 생성할 이미지 이름을 지정하고 이미지를 빌드함 • 빌드 명령어를 실행하면 도커 파일에 정의한 내용이 한 줄 한 줄 실행되는 걸 볼 수 있음	$ docker build –t 레지스트리주소/test-paas-edu04/hello:latest
도커 이미지 등록	• 생성한 도커 이미지를 도커 허브에 올리는 과정 • 도커 허브 홈페이지에 회원 가입이 선행되어야 함	$ docker push 레지스트리주소/test-paas-edu04/hello:latest

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.4 파드 배포

- **디플로이먼트는 쿠버네티스가 무상태(stateless) 앱을 배포할 때 사용하는 가장 기본적인 컨트롤러이다. 최근에는 디플로이먼트로 파드 배포를 수행한다.**

[그림 9-135] 파드 배포

단계	내용

파드 배포를 위한 Deployment .yml 작성	$ vi deployment.yml
	apiVersion: apps/v1
	kind: Deployment
	metadata:
	  name: hello
	  labels:
	    application: hello
	spec:
	  replicas: 2
	  selector:
	    matchLabels:
	      name: hello
	  template:
	    metadata:
	      labels:
	        name: hello
	    spec:
	      containers:
	        - name: hello
	          image: default-route-openshift-image-registry.apps.dmzsd.kbstar.local/test-paas-edu04/hello:latest
	          imagePullPolicy: Always
	          ports:
	            - containerPort: 8080
	              protocol: TCP
	      imagePullSecrets:
	      - name: regcred
	      restartPolicy: Always

Deployment .yml 적용	$ kubectl get pods –n test-paas-edu04 $ kubectl get deployments –n test-paas-edu04
생성된 파드 조회	$ kubectl get pods –n test-paas-edu04 $ kubectl get deployments –n test-paas-edu04

디플로이먼트 조회	$ kubectl get pods -n test-paas-edu04
	NAME READY STATUS RESTARTS AGE
	hello-857fcf74b4-8d8bh 1/1 Running 0 56s
	hello-857fcf74b4-tnqwl 1/1 Running 0 56s
	$ kubectl get deployments -n test-paas-edu04
	NAME READY UP-TO-DATE AVAILABLE AGE
	hello 2/2 2 2 76s

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.5 서비스 배포

- **파드 라우팅을 위한 서비스 작성 및 배포는 다음과 같다.**

[그림 9-136] 서비스 배포

파드 라우팅을 위한 service.yml 작성

```
$ vi service.yml

apiVersion: v1
kind: Service
metadata:
  name: hello
  labels:
     application: hello
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    name: hello
  type: ClusterIP
```

service.yml 적용

```
$ kubectl apply -f service.yml –n test-paas-edu04 service/hello created
```

생성된 서비스 조회

```
$ kubectl get services -n test-paas-edu04
NAME    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
hello   ClusterIP   172.30.236.217   <none>        8080/TCP   90s
```

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.6 잉그레스 배포

- **외부에서의 요청을 전달하기 위한 잉그레스를 작성하고 전달하기 위한 과정은 다음과 같다.**

[그림 9-137] 잉그레스 배포

단계	내용

Ingress.yml 작성	$ vi service.yml
	
	apiVersion: v1
	kind: Service
	metadata:
	  name: hello
	  labels:
	    application: hello
	spec:
	  ports:
	  - port: 8080
	    protocol: TCP
	    targetPort: 8080
	  selector:
	    name: hello
	  type: ClusterIP

Ingress.yml 적용	$ kubectl apply -f service.yml –n test-paas-edu04 service/hello created
잉그레스 조회	$ kubectl get ingress -n test-paas-edu04 NAME HOSTS ADDRESS PORTS AGE hello hello.kbstar.local 10.1.3.10,10.1.3.11,10.1.3.12 80 49s ※ 이스티오 적용된 오픈쉬프트는 적용 안됨

9.4 컨테이너 기반 마이크로서비스 빌드·배포

9.4.3 도커와 쿠버네티스를 활용한 배포

9.4.3.6 잉그레스 배포

- **외부에서의 요청을 전달하기 위한 잉그레스를 작성하고 전달하기 위한 과정은 다음과 같다.**

[그림 9-138] 잉그레스 배포

Ingress 주소 미존재 시 파드 접근

```
$ kubectl port-forward pod/$(kubectl get pod | grep hello- | head -n 1 | awk '{print
$1}') 8080:8080
```

※ 오픈쉬프트 이스티오 환경에서 port-forwar를 통해 파드에 접근할 수 있음

Curl 영령으로 URL 요청 확인

```
$ curl localhost:8080

<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <meta name="description" content="">
   <meta name="author" content="">
………
```

hosts에 IP 및 Ingress 설정 Host명 등록

```
$ C:₩Windows₩System32₩drivers₩etc
10.180.60.12   hello.kbstar.local
```

※ Ingress에 설정된 host URL로 거래 요청을 위하여 DNS 설정 대신 hosts에 IP 및 Ingress 설정 Host명 등록(일반적인 경우)

클라우드 네이티브 정보시스템 구축을 위한
개발자 안내서

발행일
2021년 12월 30일 발행

발행처
한국지능정보사회진흥원

담당팀
디지털정부기반지원팀