대용량 아키텍쳐 설계

마이크로서비스 아키텍쳐

#목표

"근래에 유행하는 백앤드 아키텍쳐 스타일인 MSA에 대해서 알아본다."

시작하기에 앞서서

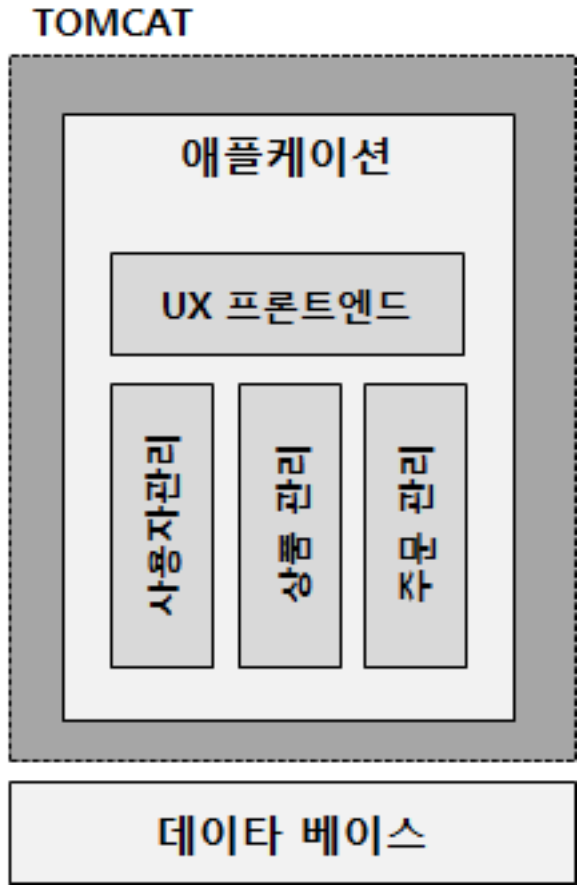
- 아쉽게도 “마이크로서비스 아키텍쳐”를 권장하지는 않겠습니다.
  - 실력이 전체적으로 좋은 팀만이 운영이 가능
  - 좋은 커뮤니케이션이 전제되어야 함
  - 서비스 개발에 유용 (SI 나 프로젝트식의 만들고 빠지는 식에는 안좋음)
  - 장애 처리 , 테스트 어려움

#1

기본 개념의 이해

전통적인 아키텍쳐 스타일

- 모노리틱 아키텍쳐 (통서버)
  - 하나의 서버에 모든 비지니스 로직이 들어가 있는 형태
  - 하나의 중앙 집중화된 데이타 베이스에 모든 데이타가 저장됨



전통적인 아키텍쳐 스타일의 장단점

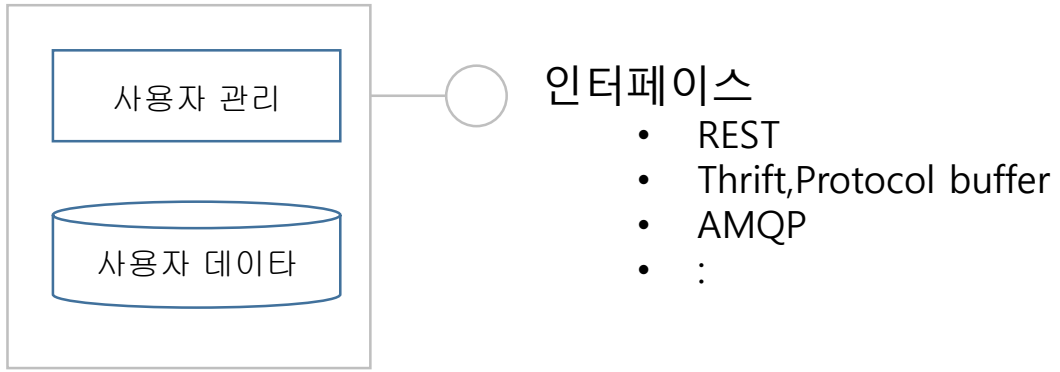
- 단점
  - 여러개의 기술을 혼용해서 사용하기 어려움 (node.js , Ruby, Python etc)
  - 배포 및 재기동 시간이 오래 걸림
  - 수정이 용이하지 않음 (타 컴포넌트 의존성)
- 장점
  - 기술 단일화
  - 관리 용이성

마이크로서비스 아키텍쳐란?

- 시스템을 여러개의 독립된 서비스로 나눠서, 이 서비스를 조합함으로서 기능을 제공하는 아키텍쳐 디자인 패턴
- SOA의 경량화 버전 (실패한다면 실패하는 이유도 같음)

서비스란?

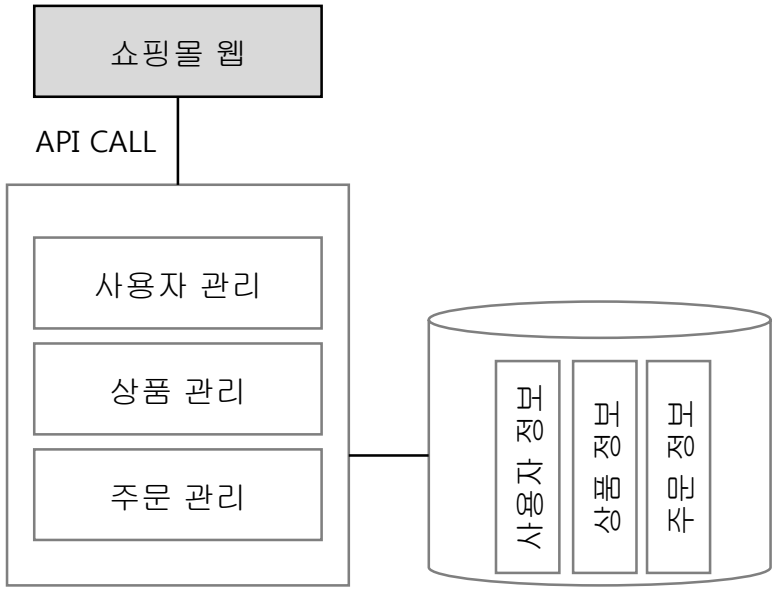
- 단일된 기능 묶음으로 개발된 서비스 컴포넌트
- REST API등을 통하여 기능을 제공
- 데이타를 공유하지 않고 독립적으로 가공 저장



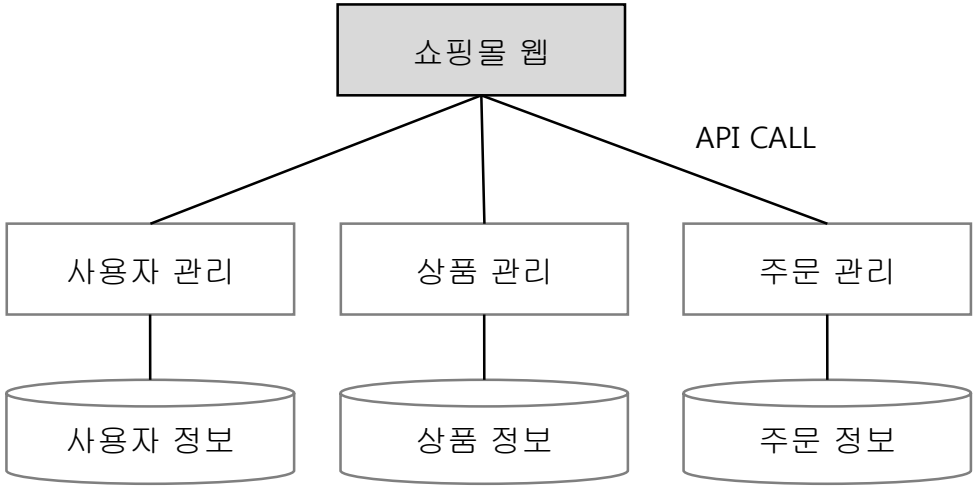
서비스 조합

- 하나의 기능을 구현 하는데, 여러개의 서비스를 조합하여 기능을 제공

  예) 주문 하기 : 사용자 정보 조회, 상품 정보 조회, 신규 주문 생성



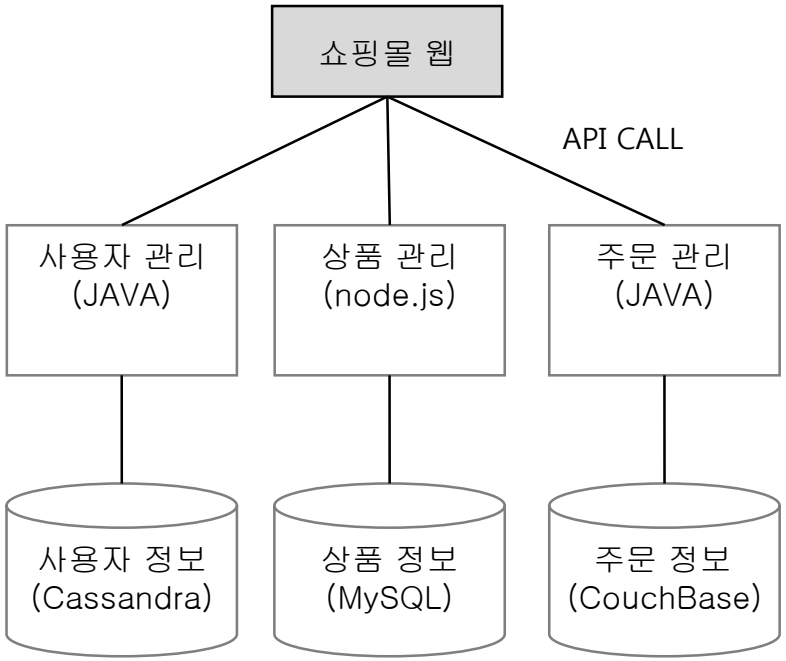
모노리틱
아키텍쳐



마이크로 서비스
아키텍쳐

기술 스택

- 마이크로 서비스 아키텍쳐는 서비스 별로 다른 기술 스택을 사용할 수 있음



장점일까?

- 운영 관점에서 여러가지 기술을 동시에 다뤄야 함
  (Devops – You build, You run)
- 사람이 떠나면 보수 불가
  (Product not a project)

단점일까?

- 적절한 기술을 적절하게 배치 가능
  - 복잡한 데이타 RDBMS, 양이 많은 고속 데이타 NoSQL
  - C10K NoSQL, 빠른 개발 스크립트 언어, 튼튼한 시스템은 자바 …

팀 모델

- 컨웨이의 법칙
  - 뼈저리게 느낌

“소프트웨어의 구조는 그 소프트웨어를 만드는 팀의 구조와 일치한다.”

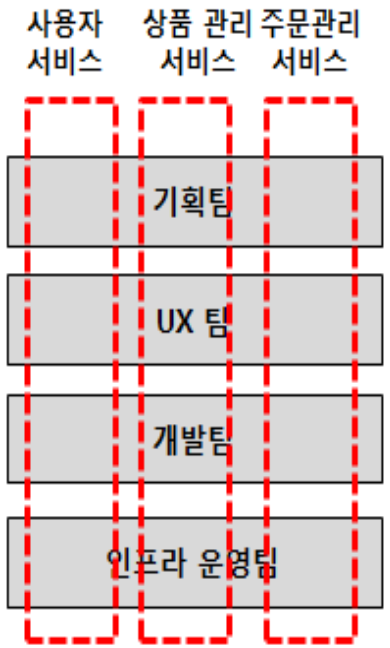
설계 백날 잘해야 소용없음
제대로 된 팀 구조를 만드는 것이 설계 (그 다음은 알아서 됨 ?)

- 친한팀 컴포넌트끼리는 개발이 잘됨. 그러다 보니 그쪽으로 기능이 몰림
- 잘하는 팀한테 자꾸 중요 기능이 몰림

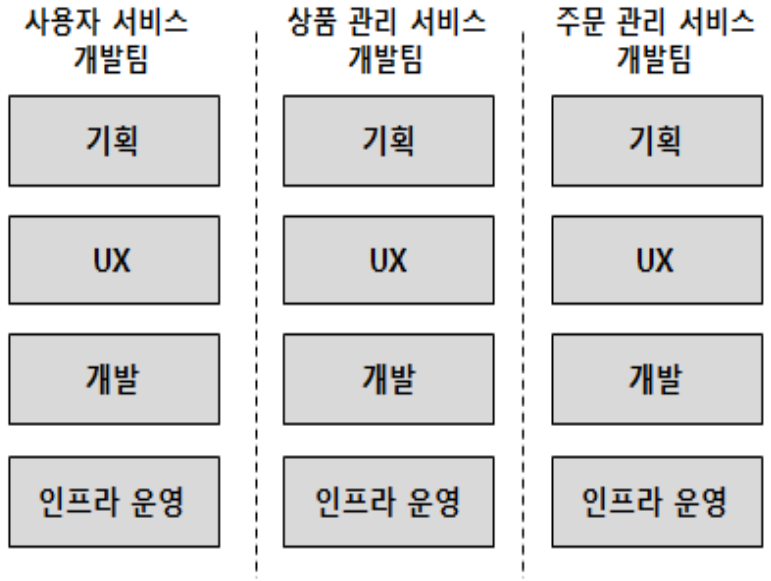
서비스 컴포넌트들이 균등하게 디자인 되지 않음

팀모델

- 분산형 거버넌스 (각 팀이 알아서. 적합한 기술, 스스로 하기)
- You build & You run!!
- Self Organized Team
- Product vs Project
- Cross functional team
- Alignment (소통!!)



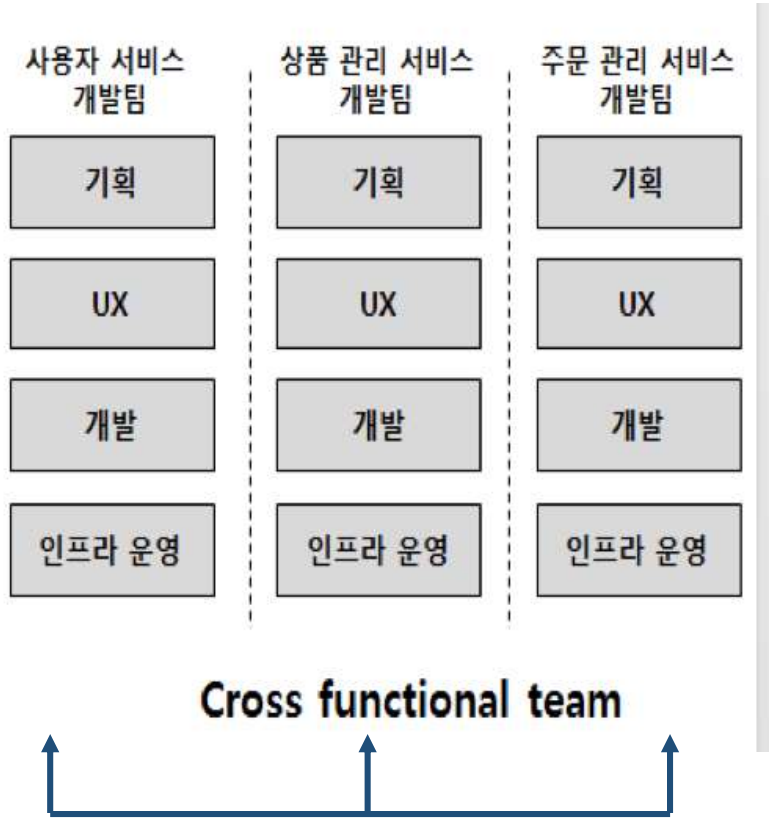
역할 중심의 개발팀



Cross functional team

팀 모델

- 팀간 조율이 핵심
  - 새로운 조율자 ROLE이 요구됨



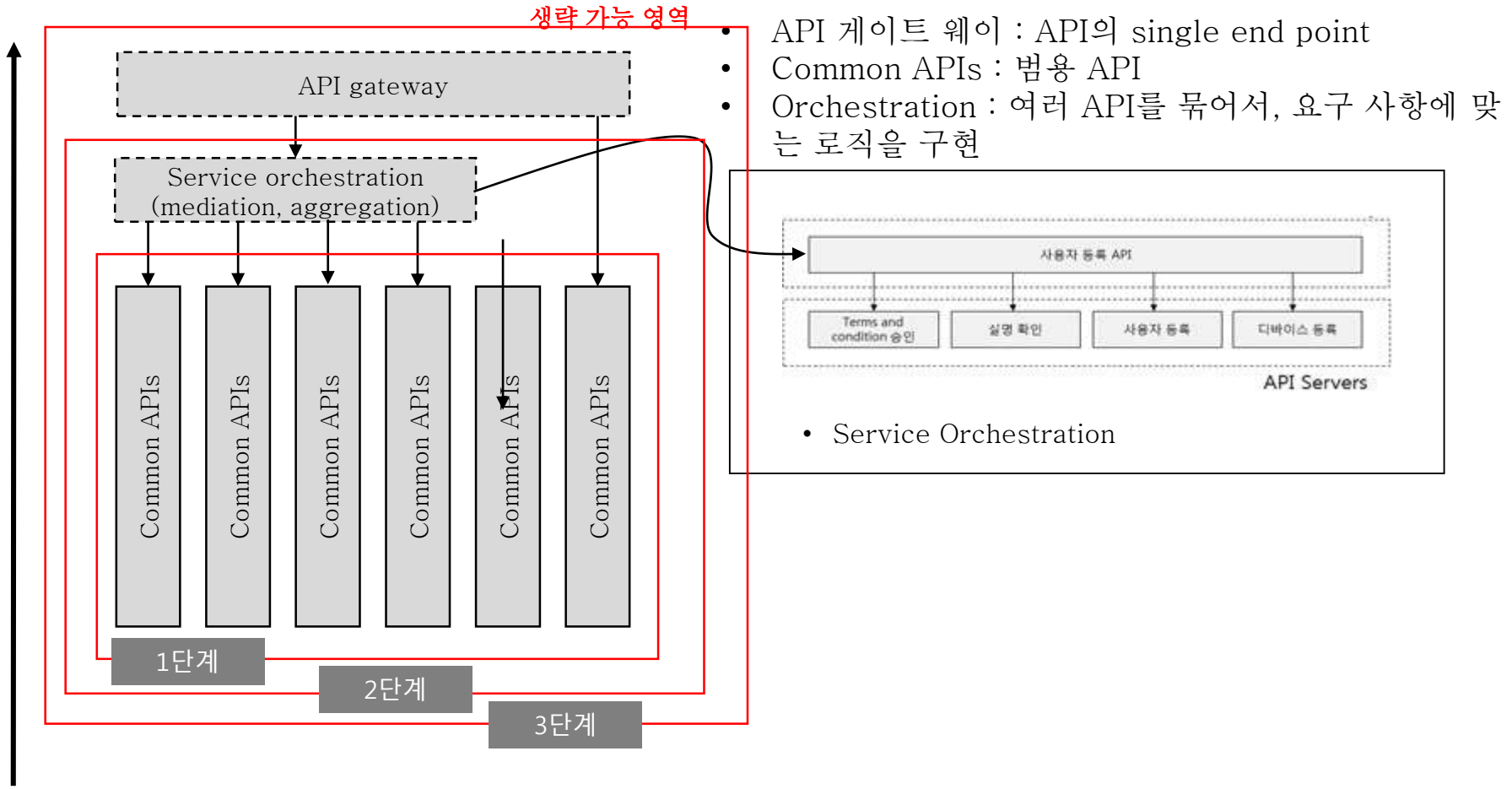
프로그램 메니져 : 팀간 일정 조율
치프 아키텍트 : 서비스간 흐름 정의, 표준 정의, 에러 추적/처리 방법 정의

#2

MSA 설계 패턴

마이크로 서비스 아키텍쳐 토폴로지

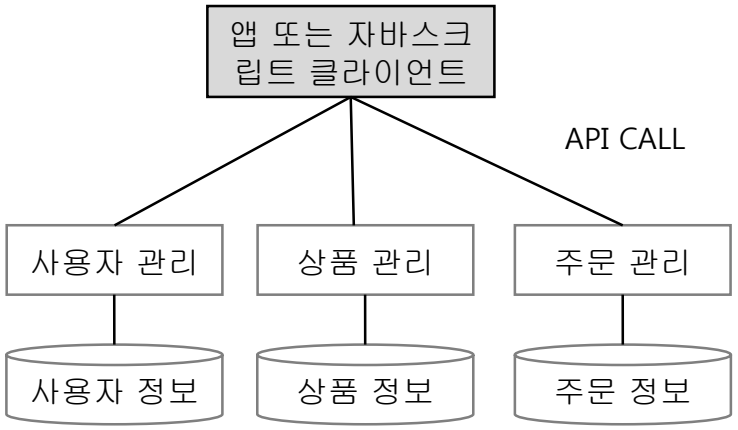
- 일반적인 마이크로 서비스 아키텍쳐 스택 구조



규모가 커질 수 록 추가되는 계층들 (오케스트레이션, API 게이트 웨이)

서비스 오케스트레이션 계층의 활용

클라이언트에서 직접 서비스를 조합 하는 방식

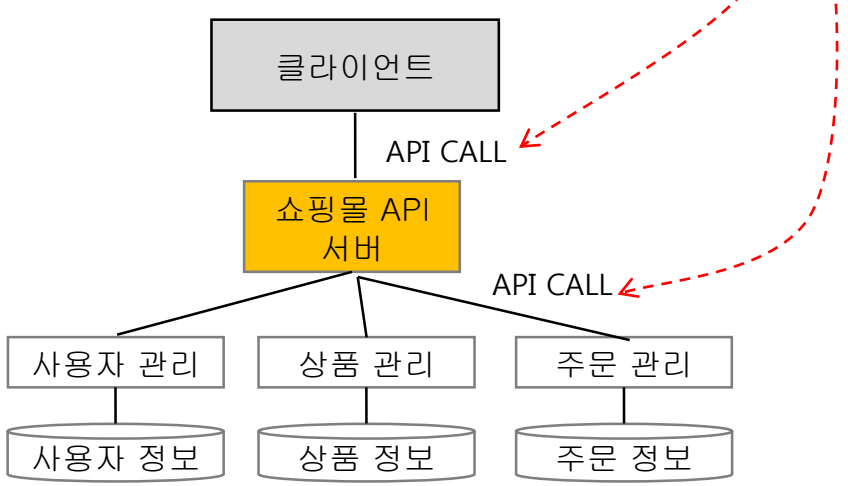


- API가 범용적으로 잘 짜야 있어야함
- 클라이언트 팀이 모든 컴포넌트팀과 조율이 필요

규모가 크지 않은 구조에서 효과적

별도의 조합 계층(Orchestration or Mediation)을 넣는 방식

다른 프로토콜 사용 가능
ex)내부 PB, 외부 REST

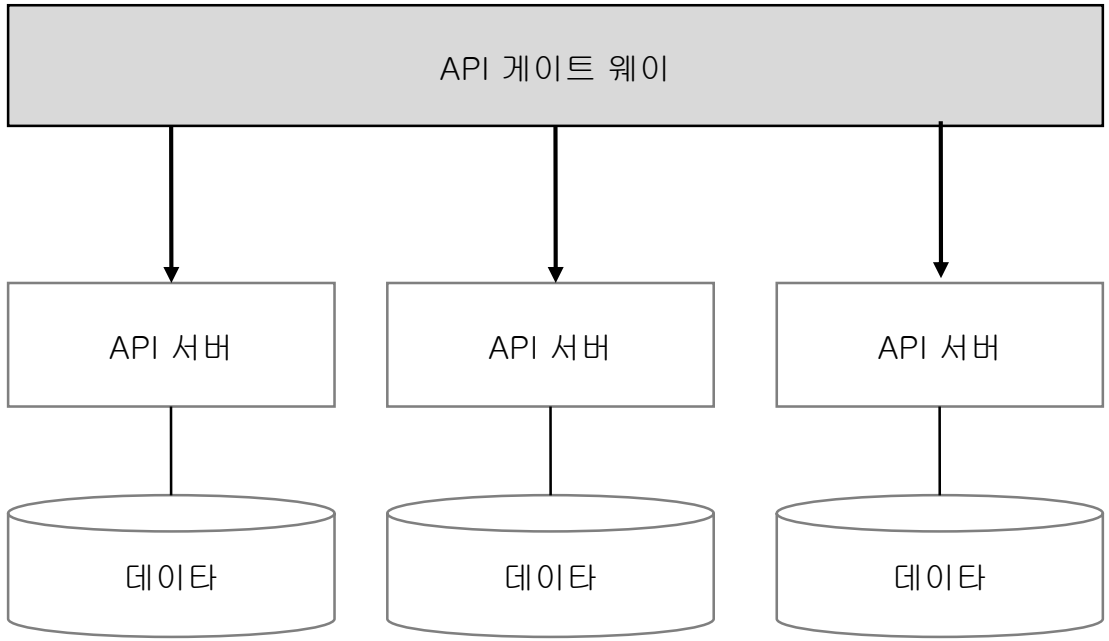


- 중간에 API를 커스터마이제이션 또는 조합하는 계층을 별도로 둠
- 클라이언트팀은 조합 API 개발팀과 커뮤니케이션만 하면 됨
- 클라이언트 요구 사항에 기민하게 대처
- 그러나 계층은 하나 더 늘어남 (성능,디버깅,배포)

일정 규모 이상. 특히 클라이언트가 여러개인 구조에 효과적

API 게이트 웨이

- 클라이언트와 API 서버 앞에서 일종의 프록시 처럼 위치 하여 다양한 기능을 수행함
  - API 인증/인가
  - 로깅
  - 라우팅
  - 메시지 변환
  - 메시지 프로토콜 변환
  - :

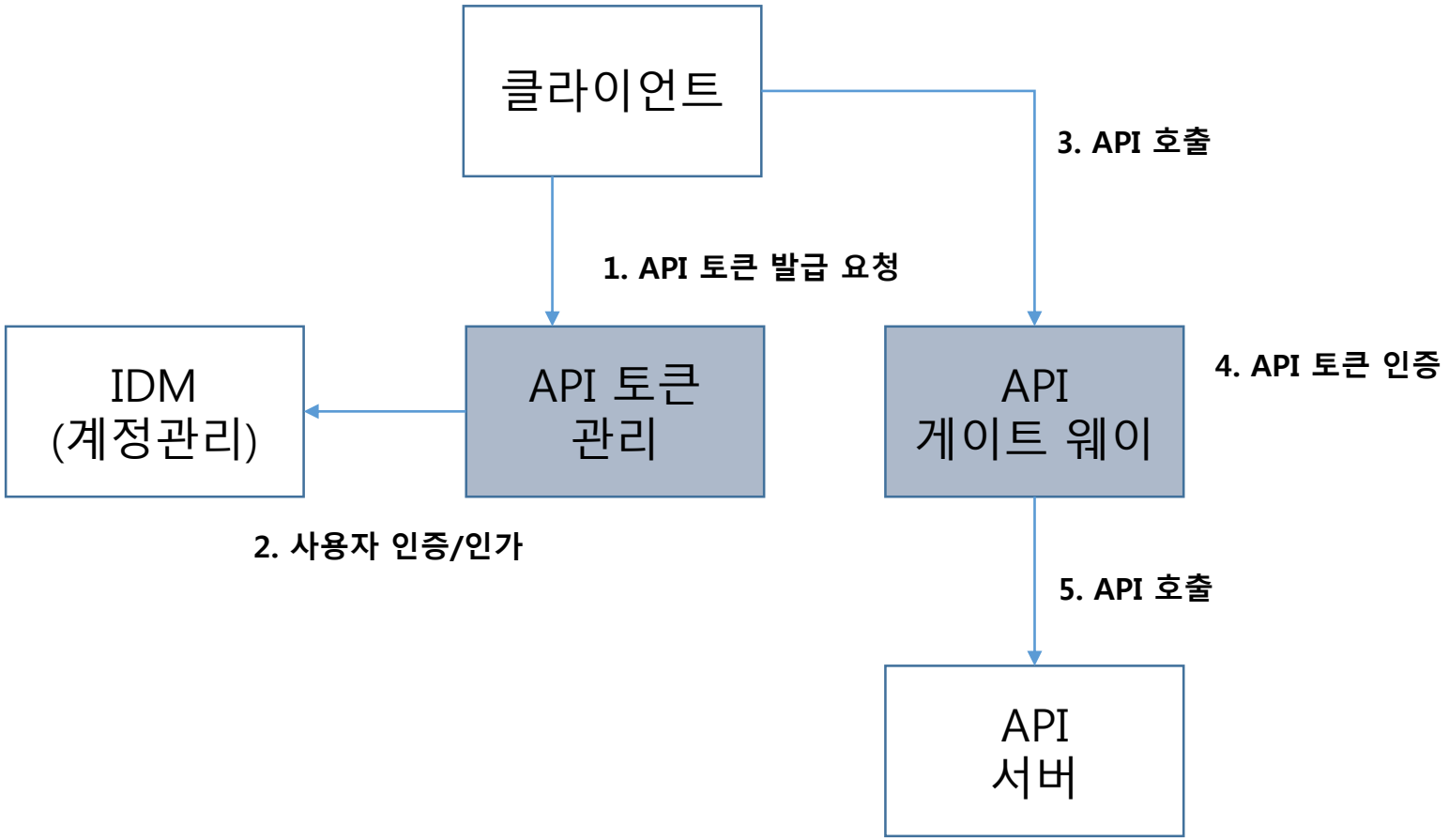


API 게이트 웨이

- SOA ESB (Enterprise Service Bus)의 단순화 버전
- 있으면 좋음. 없어도 됨
- 만들 수 있는 실력있으면, 쓰는게 좋음
- 잘못 쓰면 망하는 지름길

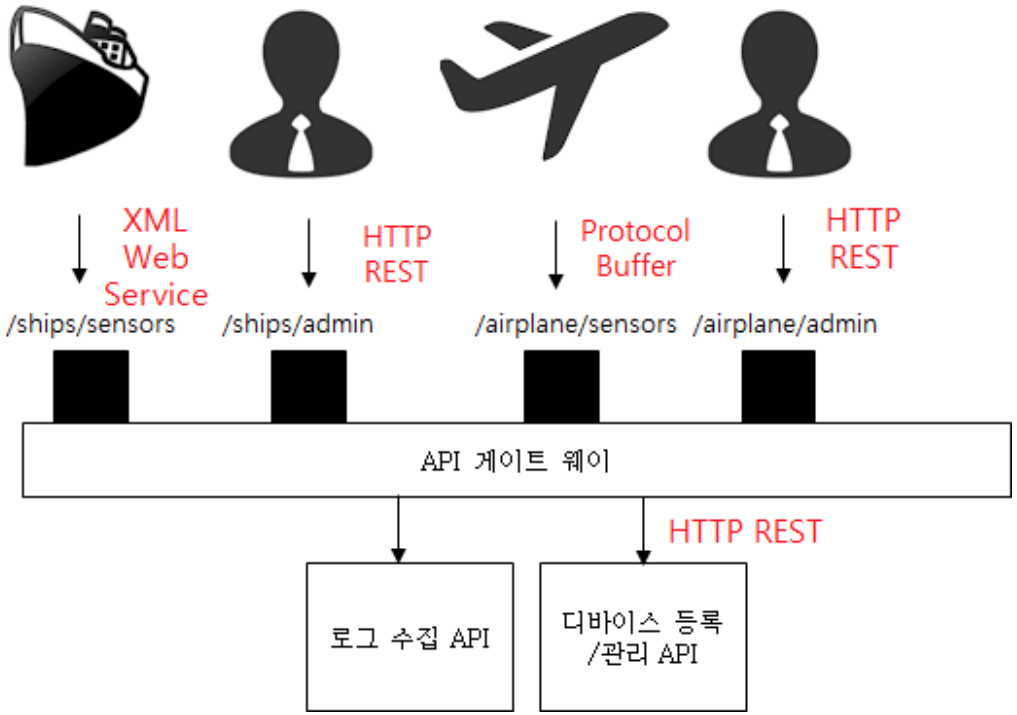
API 게이트웨이를 이용한 설계 패턴 #1

- 인증,인가의 단일화



API 게이트웨이를 이용한 설계 패턴 #2

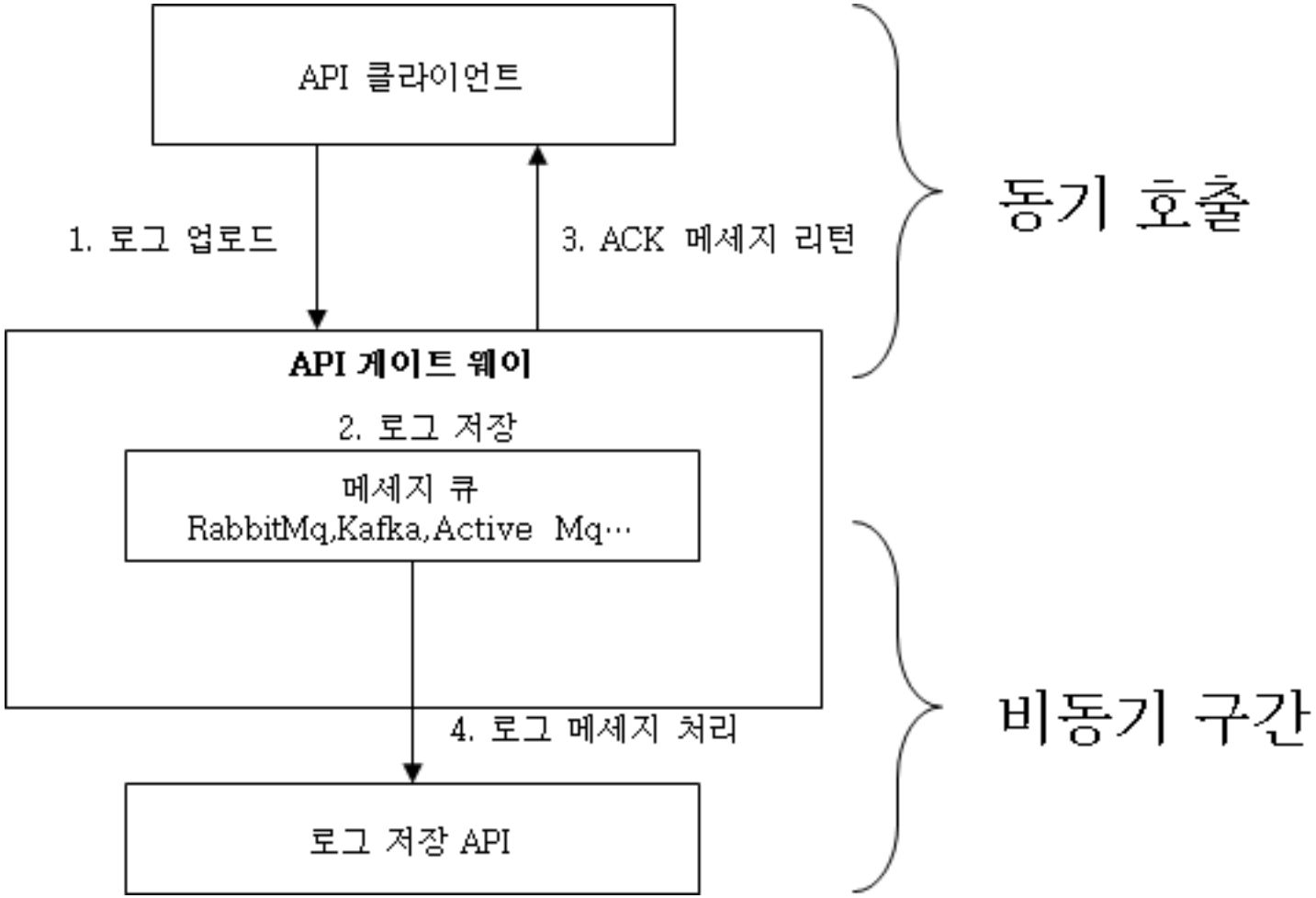
- 멀티 앤드포인트와 멀티 프로토콜 제공



<그림. 다양한 디바이스로 부터 정보를 수집하는 IOT시스템에 타입별 앤드 포인트 제공하는 예제>

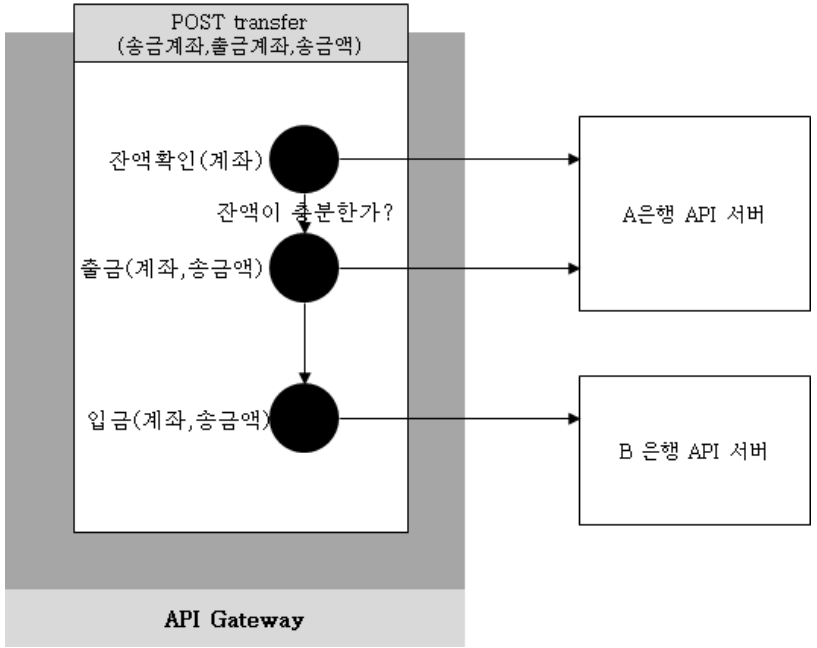
API 게이트웨이를 이용한 설계 패턴 #2

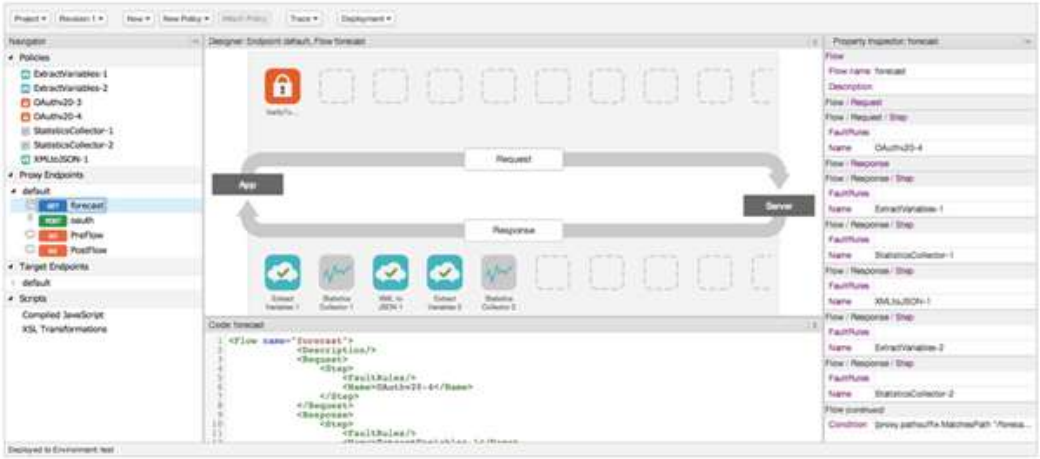
- MEP (Message Exchange Pattern) 변화



API 게이트웨이를 이용한 설계 패턴 #3

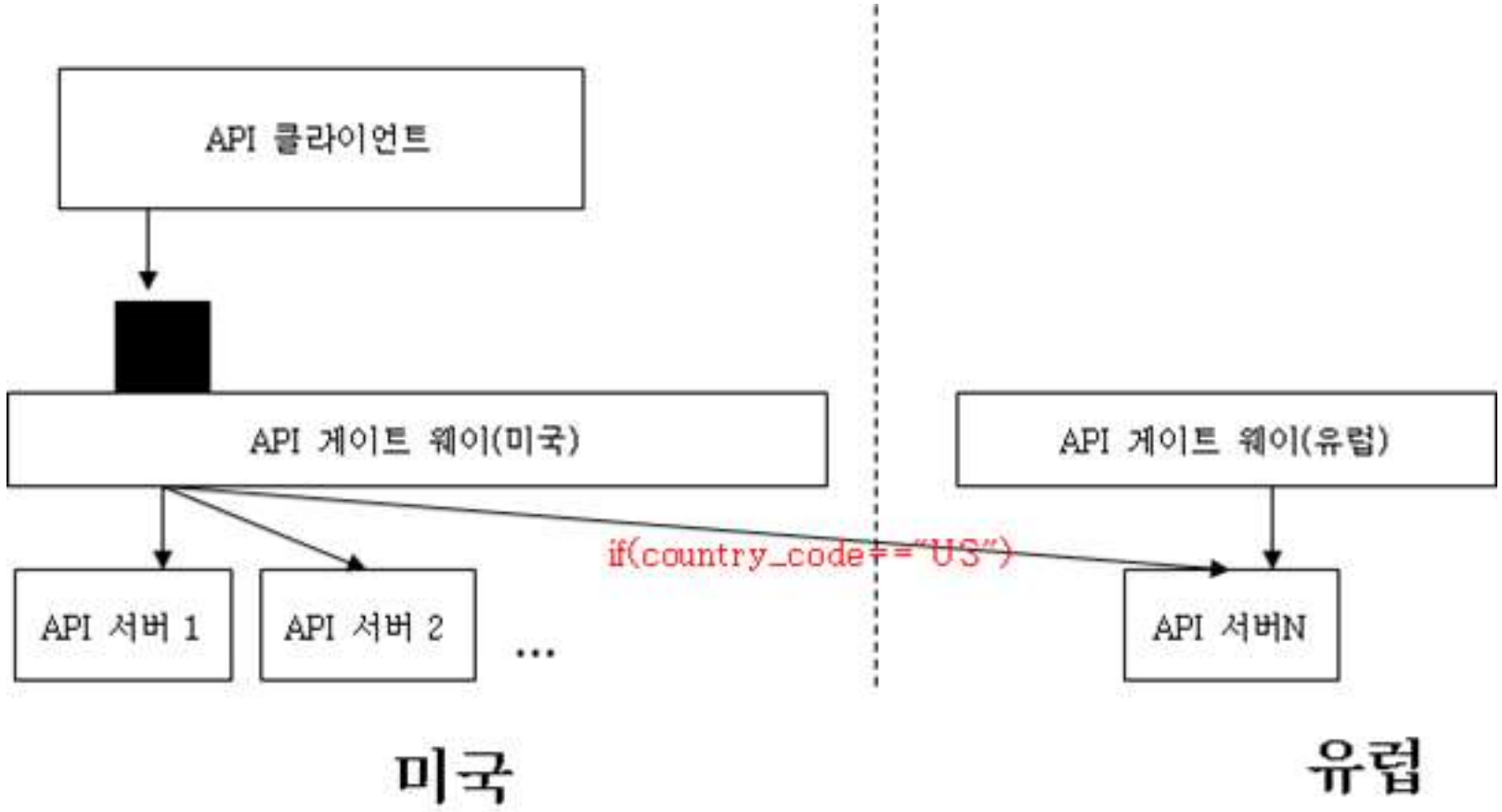
- 오케스트레이션
  - ESB 기반 SOA 프로젝트가 실패한 대부분의 원인 (안하는게 좋음)
  - 오케스트레이션 서버를 별도로 두는게 좋음





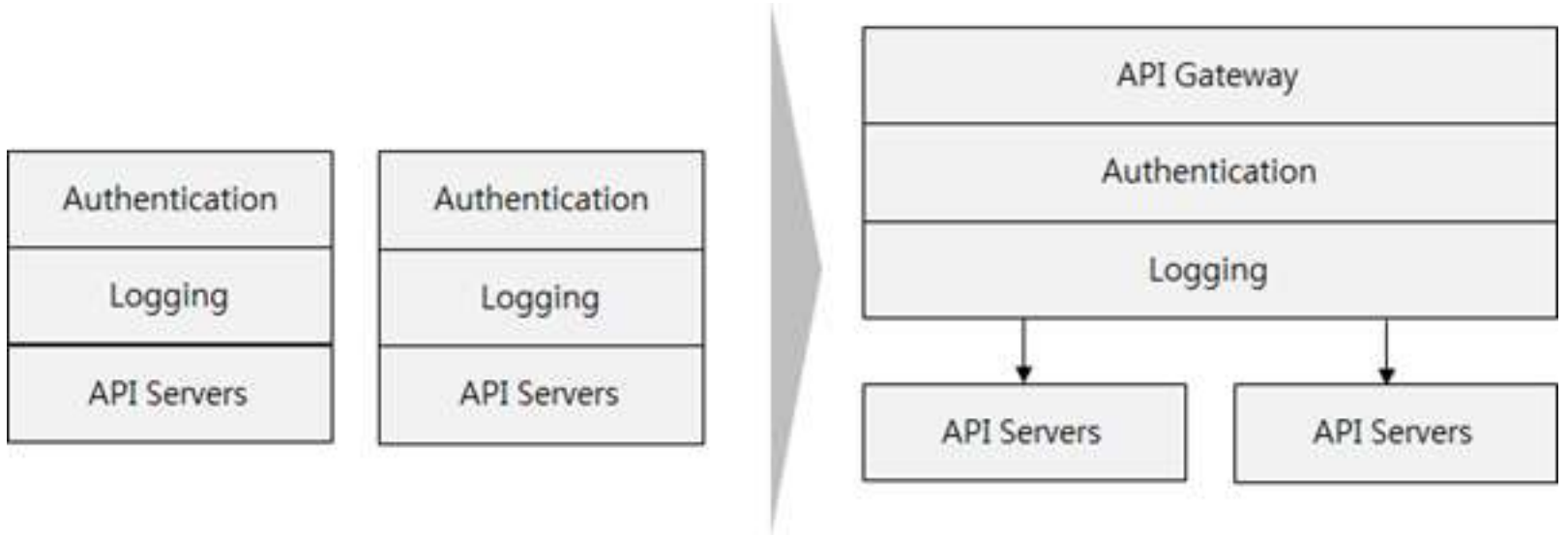
API 게이트웨이를 이용한 설계 패턴 #4

- 메세지 기반 라우팅
  - 글로벌 배포 시스템에 유용함
  - 멀티 버전 시스템 (레거시 업그레이드)에 유용



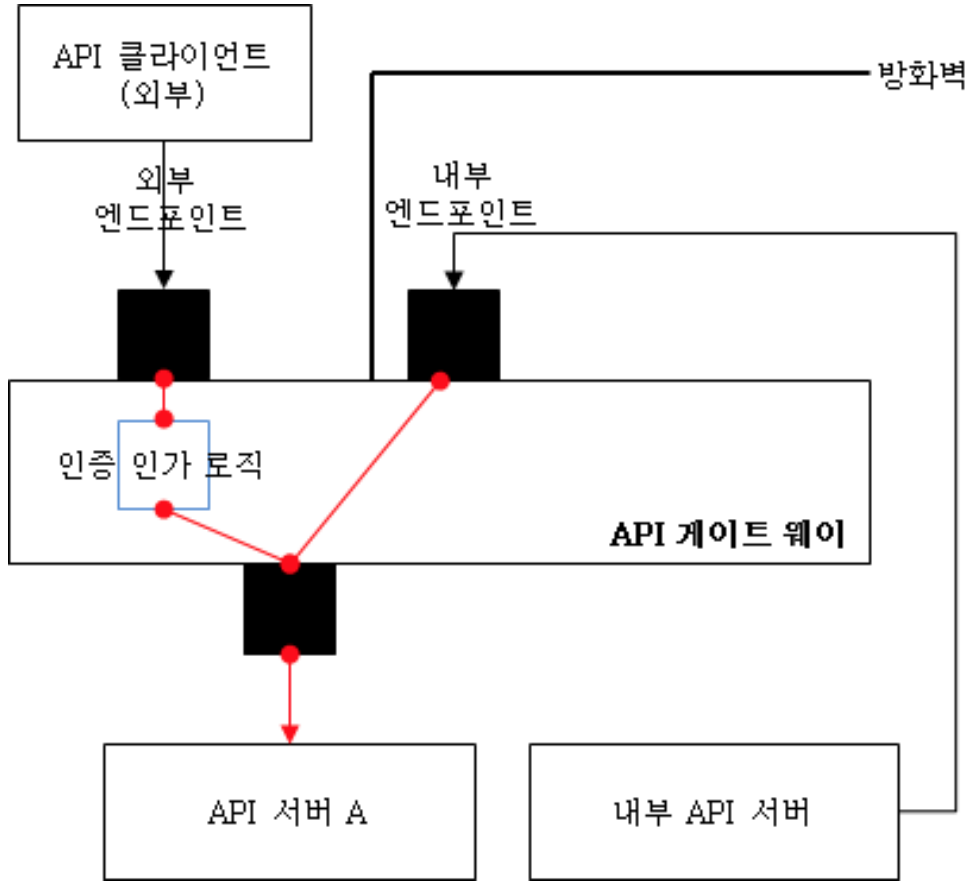
API 게이트웨이를 이용한 설계 패턴 #5

- Cross Cutting Concern (공통 기능) 처리
  - 인증,로깅등 공통 기능 처리
  - API 개발팀은 비지니스 로직에 집중할 수 있도록 해줌



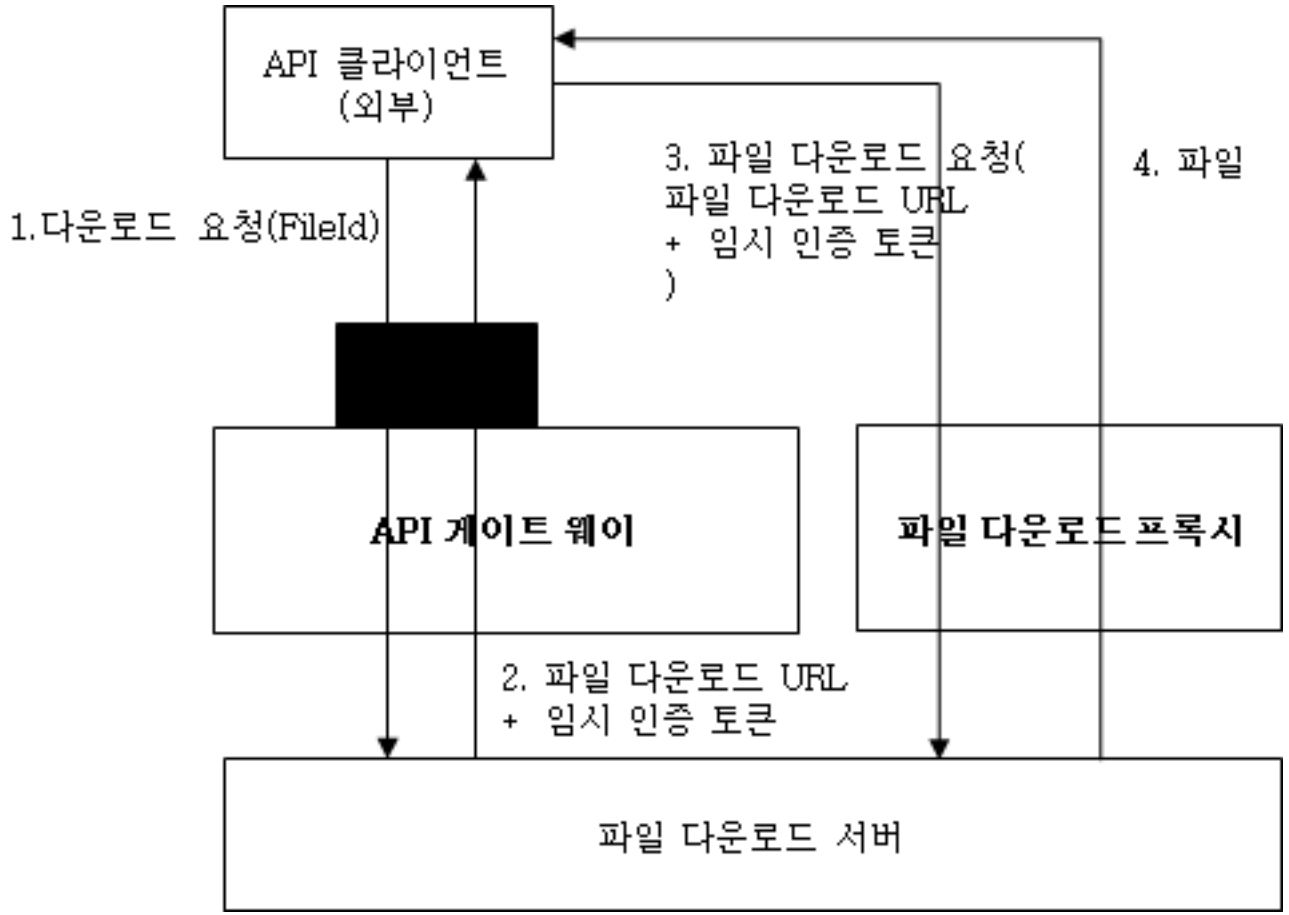
API 게이트웨이를 이용한 설계 패턴 #6

- 다중 API 게이트 웨이 패턴
  - 내부,외부용 API 게이트웨이 분리



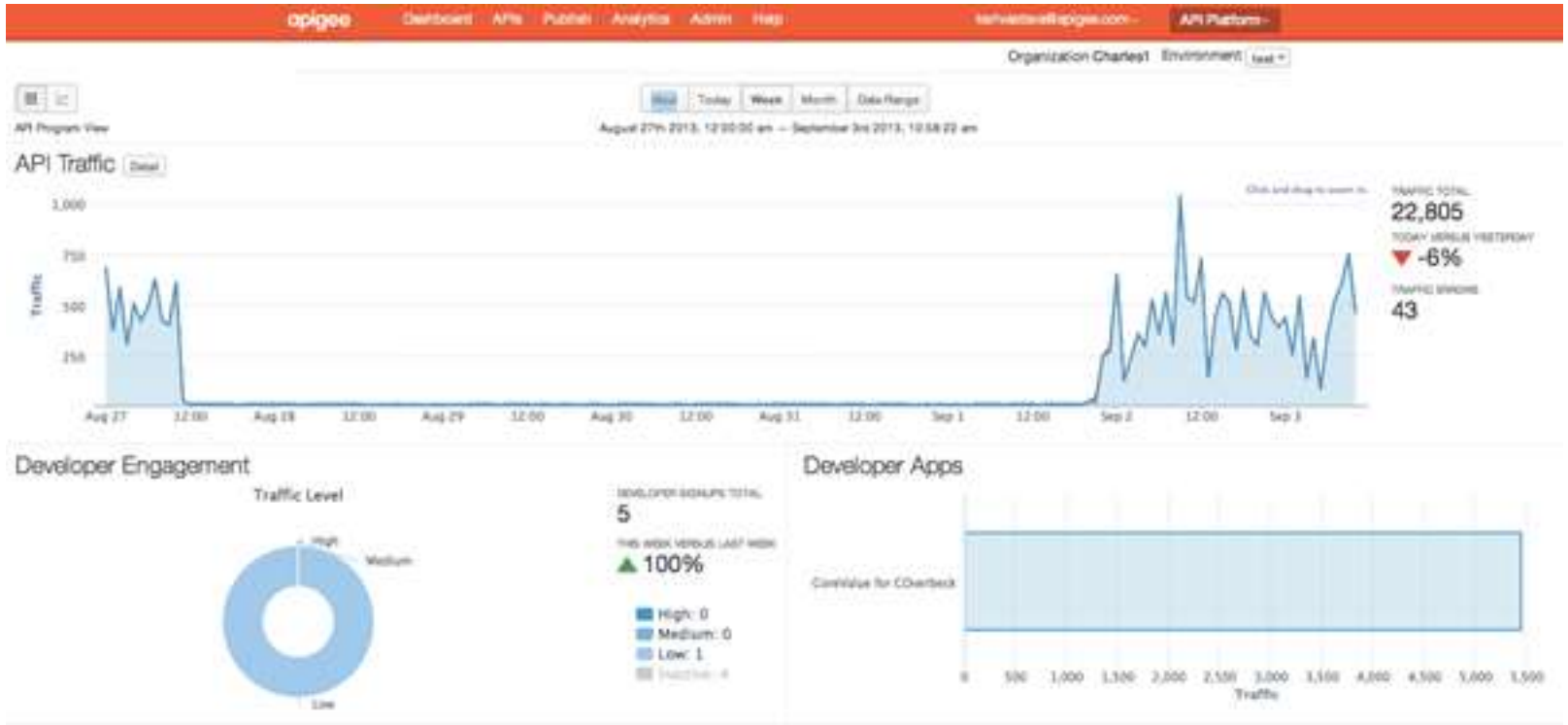
API 게이트웨이를 이용한 설계 패턴 #7

- API 호출용 엔드포인트와 스트리밍 (파일)용 엔드포인트 분리



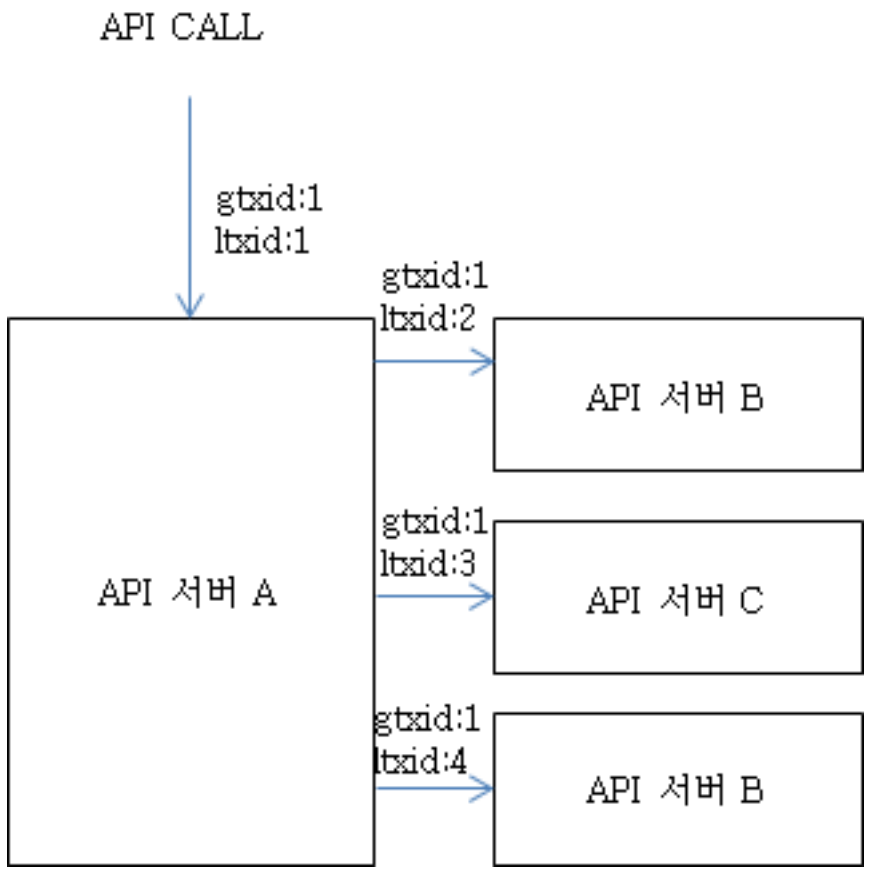
API 게이트웨이를 이용한 설계 패턴 #6

- 비 기능 요소 제어
  - QoS 제어
  - Metering & Charging (상용 API 서비스 과금)
  - API 모니터링



분산 트렌젝션의 추적

- 여러개의 서비스 컴포넌트를 조합하여 움직이는 트렌젝션에 대한 추적 디자인 패턴
- **원리 (XA 분산 트렌젝션과 유사)**
  - 초기 API에서 GTXID와 LTXID를 생성
  - 서버를 넘어갈때 마다 같은 GTXID를 사용, LTXID는 하나씩 증가
- **구현시**
  - 서버간에는 HTTP 헤더로 TXID 전달
  - 서버내에서는 Thread Local (Java)등의 컨택스트 변수 활용
    → 초기 표준 설계가 중요함



Beyond API 게이트 웨이

- API 플랫폼
  - API 게이트 웨이
  - API 포탈 (웹)
    - API 스펙 문서
    - 샌드 박스 제공
- 제품군
  - 설치형
    - WSO2 (오픈소스)
    - CA Layer 7 (상용)
    - Tyk.io (오픈소스)
    - IBM Strong Loop (오픈소스)
  - 클라우드형
    - AWS API GW
    - apigee

#3

정리

마이크로 서비스 아키텍쳐 문제점

- 장애 진단
- 테스팅
- 표준 관리
- 팀의 역량에 따른 일정 및 품질 문제
- 트렌젝션 관리
- 서비스간 코디네이션 (Chief Architect, Program manager의 역할)
- 서비스간 일정 관리

마이크로서비스 아키텍쳐와 같이 가는 개념들

- Self Organized Team
- Agile based interaction model
- Devops (Development & Operation)
- CD (Continuous Delivery)
- Product not a project

마이크로서비스 아키텍쳐

- 전체적인 기술 역량이 뛰어날 것
- Project 보다는 Product 기반의 서비스 조직에 유리
- 일정 규모 이상의 팀이 되어야 유리함
- 팀간 커뮤니케이션과 조직 구조 (Self organized team) 세팅이 전제
- API 게이트 웨이는 양날의 검
- 왠만하면 하지 마시고... 모노리틱과 MSA 중간 정도에서 타협을
  - 통제된 기술.
  - 너무 잘게 나누지 말고. 업무 단위 정도로 인터페이스 정해가면서.

End of document