대용량 아키텍쳐 설계

대용량 아키텍쳐 설계 패턴

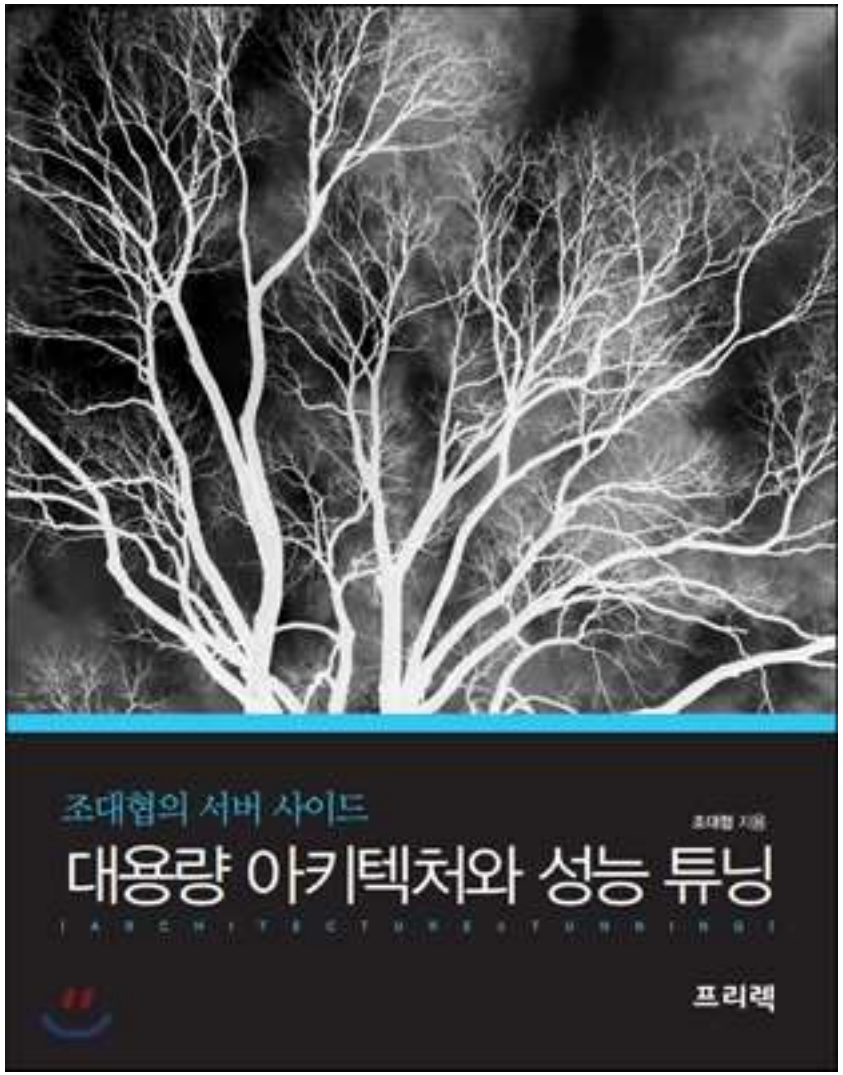
발표자 소개

조 대 협 본명: 조병욱

nhn Microsoft bea ORACLE SAMSUNG Pikicast

- 회원 13만명 온라인 개발자 커뮤니티 자바스터디(www.javastudy.co.kr) 운영자.. (기억의 저편..)
- 한국 자바 개발자 협회 부회장,서버사이드 아키텍트 그룹 운영자
- 벤쳐 개발자
- BEA 웹로직 기술 지원 엔지니어
- 장애 진단, 성능 튜닝
- NHN 잠깐
- 오라클 컨설턴트 (SOA,EAI,ALM,Enterprise 2.0,대용량 분산 시스템)
- MS APAC 클라우드 수석 아키텍트
- 프리렌서 (잘나가는 사장님)
- 삼성전자 무선 사업부 B2B팀 Chief Architect
- 전 피키캐스트 CTO

오늘 강의 내용은



조대협의 서버사이드

“대용량 아키텍쳐와 성능 튜닝”

#목표

"대용량 백앤드 시스템의 일반적인 구조와 설계 패턴을 이해한다."

어려울 수 있습니다만, 구체적인 내용을 이해하기 보다는 어떤 구성 요소가 있는지 전체 구조를 이해한후에, 실제 설계시 찾아볼 수 있도록 합니다.

#0

개발 트렌드의 변화

요즘 실리콘 밸리에서는

잘하는 것 부터 시작해서 발전

스스로 공부,스스로 개발,내재화

소규모 조직 [10~20명]

대세는 스크립트 언어

대우 받는 개발자

유연한 고용 시장

메뚜기!!

3년 넘으면 바보

한국은 배신자

클라우드 컴퓨팅

빠른 시장 진입,저비용,누구나 서비스

젊은 개발자 (한국은 접는 개발자)

협업

[SNS,오픈소스,GitHub]

개발자의 변신은 무죄

CI to CD

STAR UP

새로운 능력..!! 잉여력

요즘 실리콘 밸리에서는

대세는 수퍼 개발자!!

- 3가지 관점에서 트랜드를 분석

소프트웨어 개발 중심의 변화

시대별 기술의 변화

개발 방식의 변화

소프트웨어 개발 중심의 변화

시대별로 소프트웨어를 개발하는 이유가 어떻게 변화 했을까?

우리가 지금 쓰는 기술들은 왜?

엔터프라이즈 시대

기술 변화의 주체

벤더

인터넷,SNS의 시대

서비스 사업자
대단한 님들!!

모바일 시대

개발자!!
(스타트업,앱개발자)
→ 당신도 할 수 있다

시대별 기술의 변화

엔터프라이즈 시대	인터넷,SNS의 시대	모바일 시대
EJB, JAVA, SERVLET/JSP 들어는 보셨나요? 벤더 : “이게 최신 기술임!!. 교육 부터 다 해드림!! 나만 믿으세요” $$$	Spring,Hibernate,MySQL ,Struts,Ruby On Rails,PHP .. 서비스사 : 제품은 비싸요. 우리가 만든거 가져다 쓰세요!! 대신 책임은 니꺼!	JavaScript, Cloud,node.js, Ruby on Rails,NoSQL 개발자:어려워서 못 써 먹겠음. 차라리 만듬..!!
먹고 살 걱정 없음!!	공부해야 함.. 압박!! 조금씩 먹고 살기 힘들어짐 (버틸만 함-그래도 대세는 있음)	기술 변화도 심함 이제는 생존의 문제!! 명퇴가 눈앞에..

시대별 기술 변화

엔터프라이즈 시대	인터넷,SNS의 시대	모바일 시대
웹,4GL	웹	안드로이드,IOS
		HTML 5
벤더 소프트웨어	오픈소스	스크립트 언어!!
		API
		오픈소스
ORACLE	MySQL	NoSQL
UNIX 서버	X86 서버	클라우드
안정성,미션 크리티컬 업무 위주	대체제 성격 새로운 기술 흐름을 만드는 전환기	쉽고 빠른 개발 기술의 홍수

시대별 기술자

- **왜 이런 현상이 생기는가? 쉬운 기술이 주목 받는 이유**

스타트업. 앱 개발

조합 1.
기획자 창업
디자이너 - 기획자 여자 친구
개발자 – 기획자의 친구 (디자이너를 사랑함)

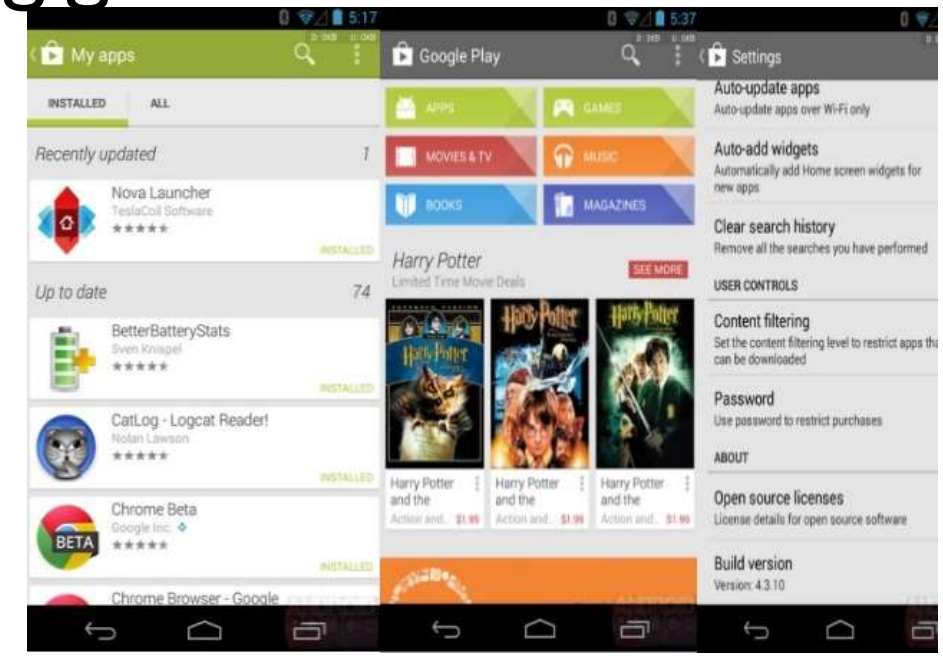
조합 2.
디자이너 창업
기획자 - 디자이너의 여자 친구
개발자 – 디자이너의 친구 (기획자를 사랑함)

개발자 1인 혼자 다 해야함 (클라이언트, 서버, 하드웨어 인프라)
배우기 쉬워야 함

돈 떨어지기전에 시장 출시해야함
빨리 배울 수 있어야함

시장 현황

수많은 앱들 / 치열한 경쟁



빠른 출시, 빠른 업데이트가 필요

요구 되는 기술

빠른 출시, 빠른 업데이트가 필요

쉬운 기술 필요

낮은 비용

클라우드 컴퓨팅

스크립트 언어

운영 효율화

Devops

자동화

클라우드 컴퓨팅

- 클라우드 컴퓨팅이 가져다준 변화
  - 저비용으로 시작 가능
  - 무제한적 리소스
  - 지역/시간 제약에서 자유로워짐
  - 쉽다. 하드웨어 인프라에 대한 작업이 없어짐

클라우드 컴퓨팅

- 고려해야 하는 것들

클라우드 컴퓨팅의 장점	설계시 고려 사항
빠른 시장 진입	느려요 IO Performance
운영비 절감	싸지 않아요
초기 투자비 절감	기존 솔루션이 안돌아요
유연한 자원 사용 (Auto Scale Out)	장애가 납니다. 아주 잘!! (멀티 데이타 센타 설계)

Devops의 등장

Devops = Development + Operation (개발 + 운영)

개발과 운영 조직을 하나로 묶어서 원할한 의사 소통
빠른 반영, 빠른 피드백 반영 가능

자동화

- 흔한(평범한) 개발 환경 시나리오
  - 개발자가 아침에 출근해서
  - 이클립스를 키고, 소스코드를 Git에서 Check Out 한후
  - JIRA를 통해서 오늘 할당된 작업을 확인 한후에 코딩을 하고
  - PC에서 Junit등을 이용하여 단위테스트등을 모두 끝 마치고
  - 코드를 Git에 Commit하면
  - Jekins에서 코드 변경을 감지하여, 자동으로 Check Out해서 mvn을 이용해서 컴파일 하고, 테스트 서버에 배포해서 단위 테스트를 모두 수행하고, 코드의 라인커버리지를 분석하여 리포팅 한다.
  - 팀장은 빌드가 완료되었음을 확인하고, 단위 테스트 100% 완료 및 라인 커버리지 80% 완료를 확인한다.
  - 릴리즈 날짜가 오면, 배포 엔지니어는 별도의 작업 없이 Jenkins에서 빌드된 그날 WAR를 확인하고, Fabric으로 된 배포 스크립트를 수행하면, 자동으로 개발,QA 환경으로 배포가 되고, 환경별로 필요한 resource 파일들이 자동으로 customization해서 배포가 완료된 후, Junit 기반의 단위 테스트, SOAP UI 기반의 REST API 테스트, Seleniuem 기반의 UI 테스트까지 자동으로 완료 한다. 만약에 배포나 테스트가 실패하면, 이전 버전으로 자동 롤백한다.

스크립트 언어

Ruby on Rails, Python,PHP

대세는 node.js

아키텍쳐 스타일의 변화

먹고 살기 좋던 시절	요즘
엔터프라이즈 시대	모바일 시대
한정된 용량	대용량
중앙 집중형	분산형
RMI,WebService	REST OPEN API
고가용성(HA)	Resilience
트랜젝션 보장	장애 허용
목표 성능	한계 성능

#1

대용량 아키텍쳐 설계 패턴

레퍼런스

- SOA
  - SOA Design Pattern (Thomas Erl) – 좋은지 잘 모르겠음. 유명하니까.
  - Applied SOA – Michael Rosen – 추천
  - Enterprise SOA – Dirk Krafzig – 옛날 책이지만 추천
  - Enterprise integration Pattern – Gregor Hohpe 연동 패턴 잘 설명됨
- 사이트
  - HighScalability.com
  - http://aosabook.org/en/distsys.html
  - http://martinfowler.com/articles/microservices.html

대용량 분산 시스템 디자인 패턴

- 대용량 분산 시스템도 디자인 패턴이 있고 대부분의 설계가 비슷함
- 여기서는 자주 사용되는 공통되는 패턴을 소개함
- 계속해서 변화되기 때문에 꾸준이 패턴을 공부하는 것이 필요함

분산 아키텍쳐 디자인 패턴

- 서비스 지향적
- Redudant & Resilience
- 파티셔닝
- Query Off Loading
- 캐슁
- CDN & ADN
- 로깅
- 비동기 패턴

디자인 PRINCIPALS

- 디자인시 다음과 같은 항목을 고려해서 디자인 해야 함
  - 가용성 (Availability)
  - 성능 (Performance)
  - 확장성 (Scalability)
  - 안정성 (Reliability)
  - 관리성(Manageability)
  - 비용 (Cost)
- 우선 순위를 정하는 것도 도움이 됨

서비스 지향적인 접근 방법 (Service Oriented Approach)

- Loosely coupled
- 기능을 API로 제공
- 표준 API
- 공통 서비스
- 컴포넌트화

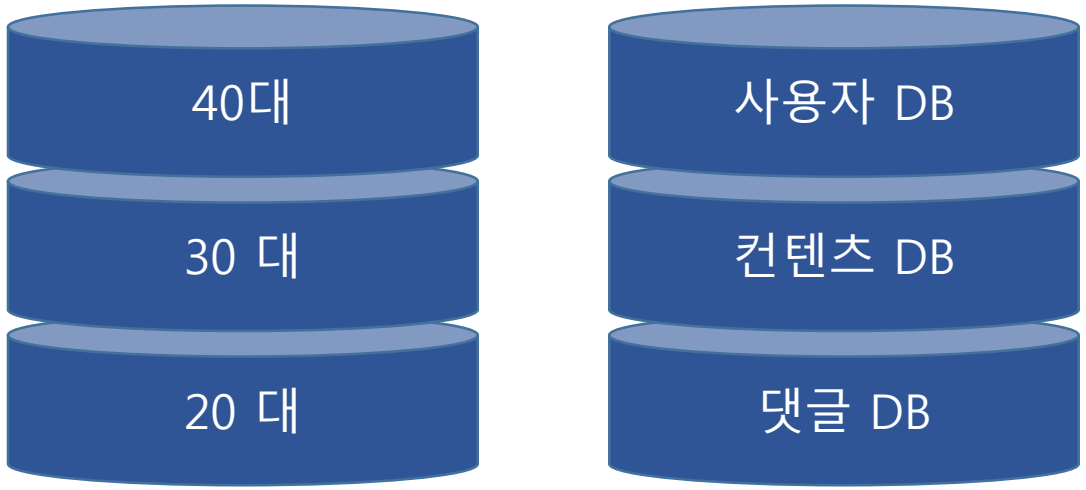
Redundant vs Resiliency

- Redundant (엔터프라이즈 서비스에 유리)
  - 이중화
  - 비싼 고가용성 서버, 클러스터링, 엔터프라이즈
  - 트렌젝션을 깨지지 않고 보장
- Resilience (대세, B2C 서비스에 유리)
  - 장애가 나면 빠르게 복구
  - X86 Commodity 하드웨어, Shared Nothing, B2C
  - 트렌젝션이 깨짐

  왜?) 대용량 서비스에서 비용을 낮추다 보니, 장애가 남. 장애가 나는 것을 전제로 하고, 고 가용에 들어가는 비용을 낮춤

파티셔닝 (샤딩/SHARDING)

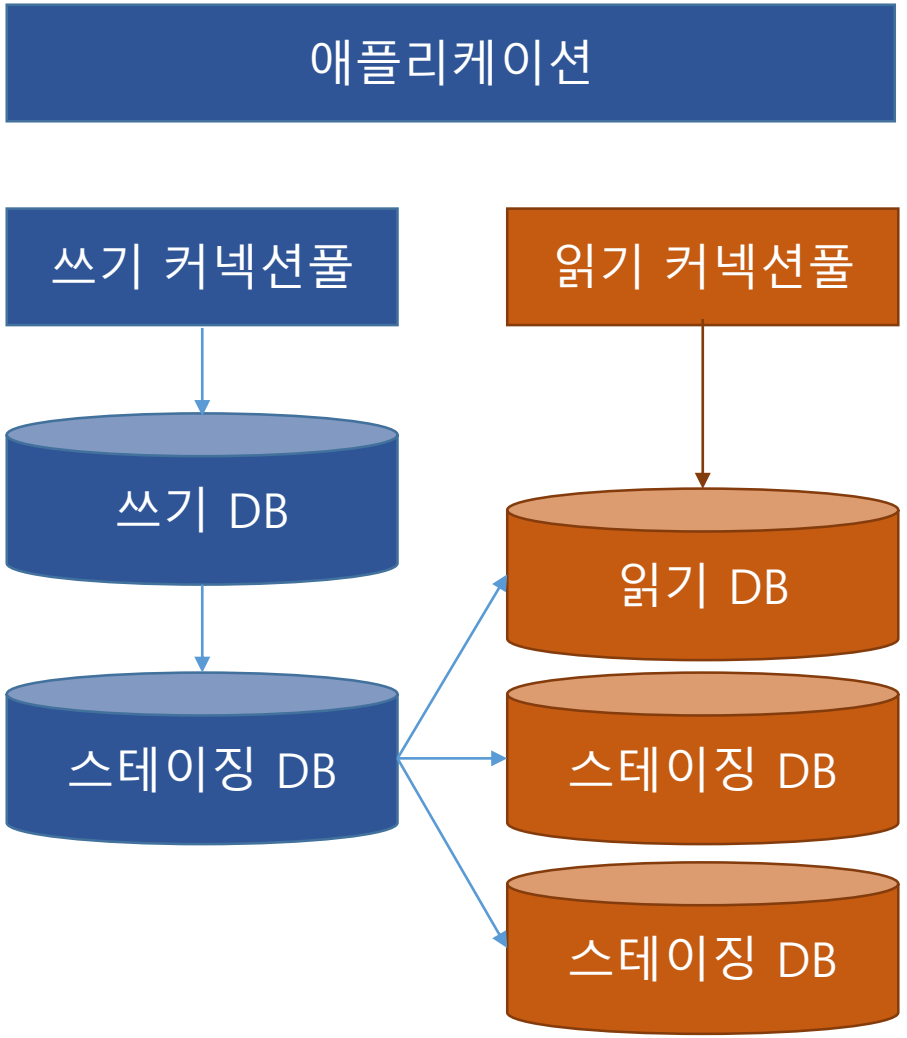
- 데이타를 여러 DB에 나눠서 자ㅓ장
- 방식
  - 수평적 샤딩(Horizontal Sharding)
  - 수직적 샤딩(Vertical Sharding)
- 데이타 쏠림에 주의
- 검색이 어려움. (별도의 Index 서버 고려)
- 일반적으로 애플리케이션에서 분산 처리 (솔루션 차원에서 지원하기도함.)



수직적 샤딩 수평적 샤딩

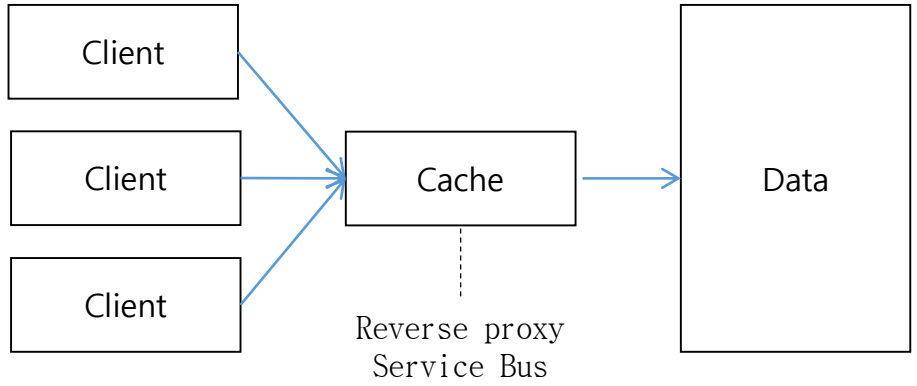
쿼리 오프 로딩

- 읽기와 쓰기를 분리
  - 일반적으로 읽기:쓰기 비율 = 80:20
  - Master node : 쓰기 중심
  - Salve node : 읽기 중심 (무한 확장 가능)
  - 중간에 스테이징 DB를 놓는 방법을 고려
  - 애플리케이션에서 분리 되서 구현현되어야 함

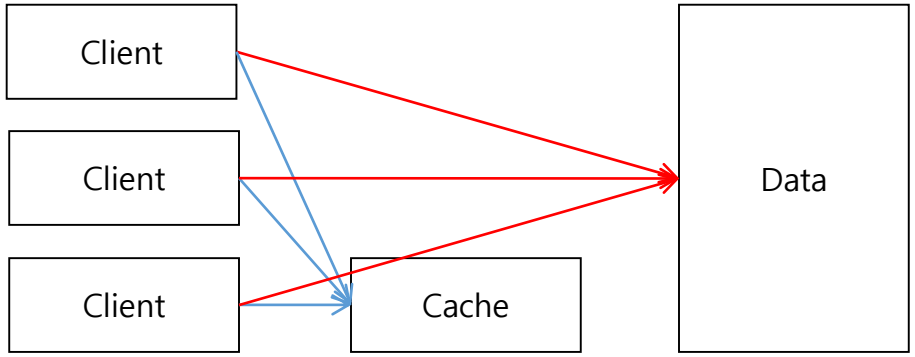


캐싱

- 중앙 집중형 캐쉬

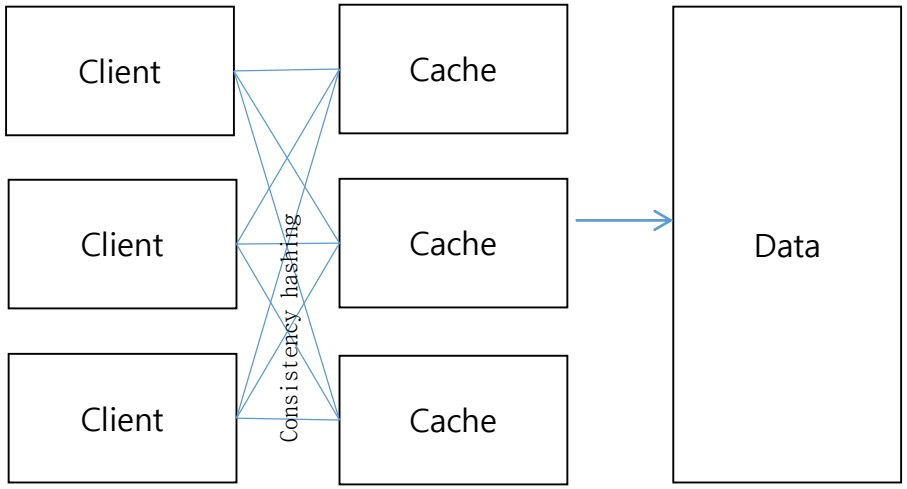


Pass through

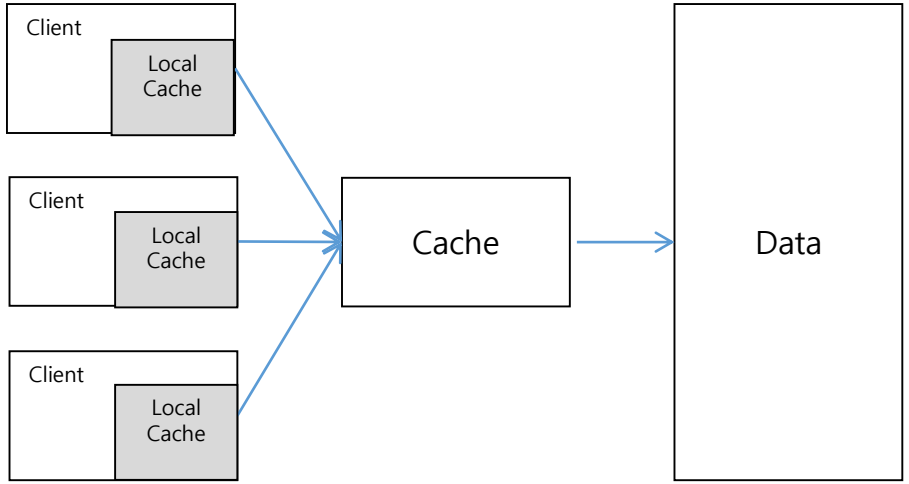


Referal

- 분산형 캐쉬



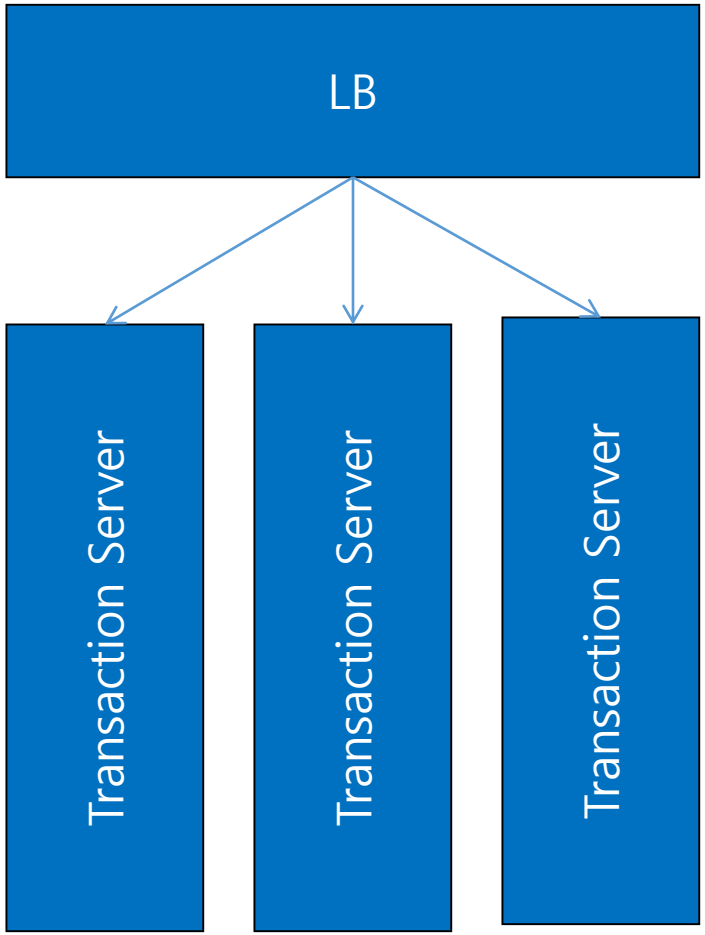
- 분산형 캐쉬



부하분산 (로드 밸런싱)

- 로드밸런싱
  - 알고리즘
    - Hash
    - Round Robin

    ※ Sticky Session (Timeout 주의)
  - L4,L7,Reverse Proxy(HAProxy, Nginx), ELB
- 글로벌 로드 밸런싱
  - Dynamic : DNS approach (Amazon Route 53)
  - Static : Look up & pinning (*)
    - CDC
    - Regional info
    - 복제할 필요가 없음.
      (비행기 타고 날라가도 같은 데이타 센터에)

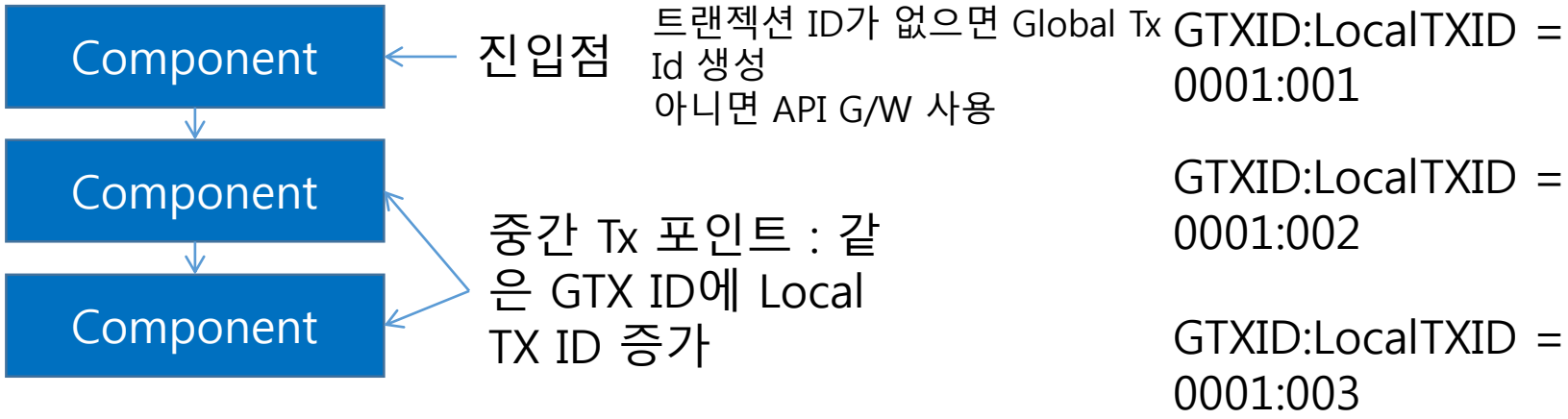


CDN & ADN

- CDN
  - 정적 컨텐츠를 지역적으로 분산된 EDGE NODE에 배포
  - 지능형 기술 보유 (CSS,HTML 압축, WebP 변환등)
  - CloudFlare (무료 CDN)
- ADN
  - 압축 전송 : 리버베드 (Riverbed)
  - 전용망 서비스 : 아카마이, AWS 클라우드 커넥스
  - 클라우드 리전에 프록시 서버를 넣어도 유사 효과

로깅

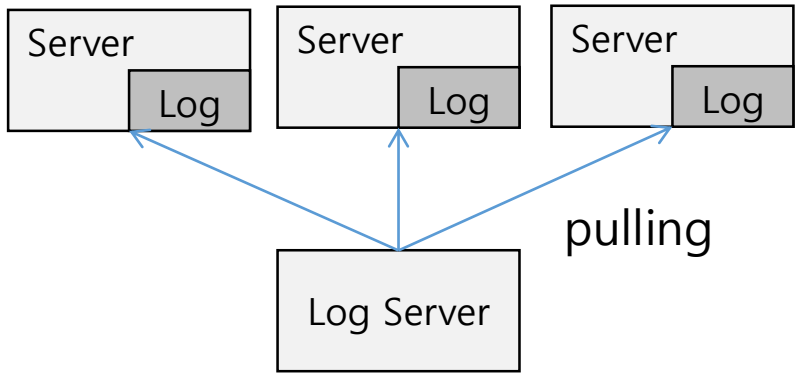
- 글로벌 트렌젝션과 로컬 트렌젝션

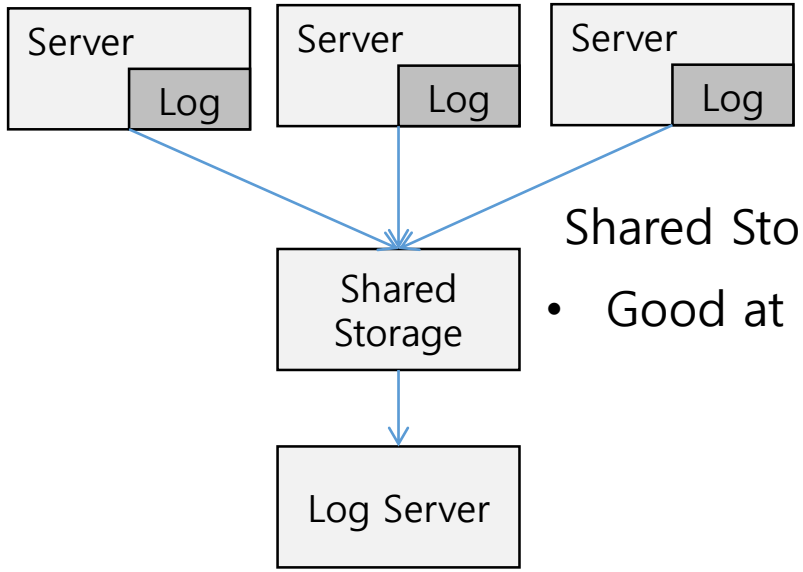


- TX ID Propagation
  - GTX ID : Header에 넘겨서 전달
  - Local TX ID : Thread Local 에 넘겨서, Local Tx내에 Propagation
- APM 이나 로깅 서비스를 사용하는 것도 좋음
  - Newrelic, Scout (오픈소스), 제니퍼

분산 로깅

- 로깅 방식
  - Pulling
  - Shared storage
- 로깅 계층
  - 시스템 로깅 (ELK etc)
  - 이벤트 로깅 (Sentry)
  - 모바일 로깅 (Flurry)

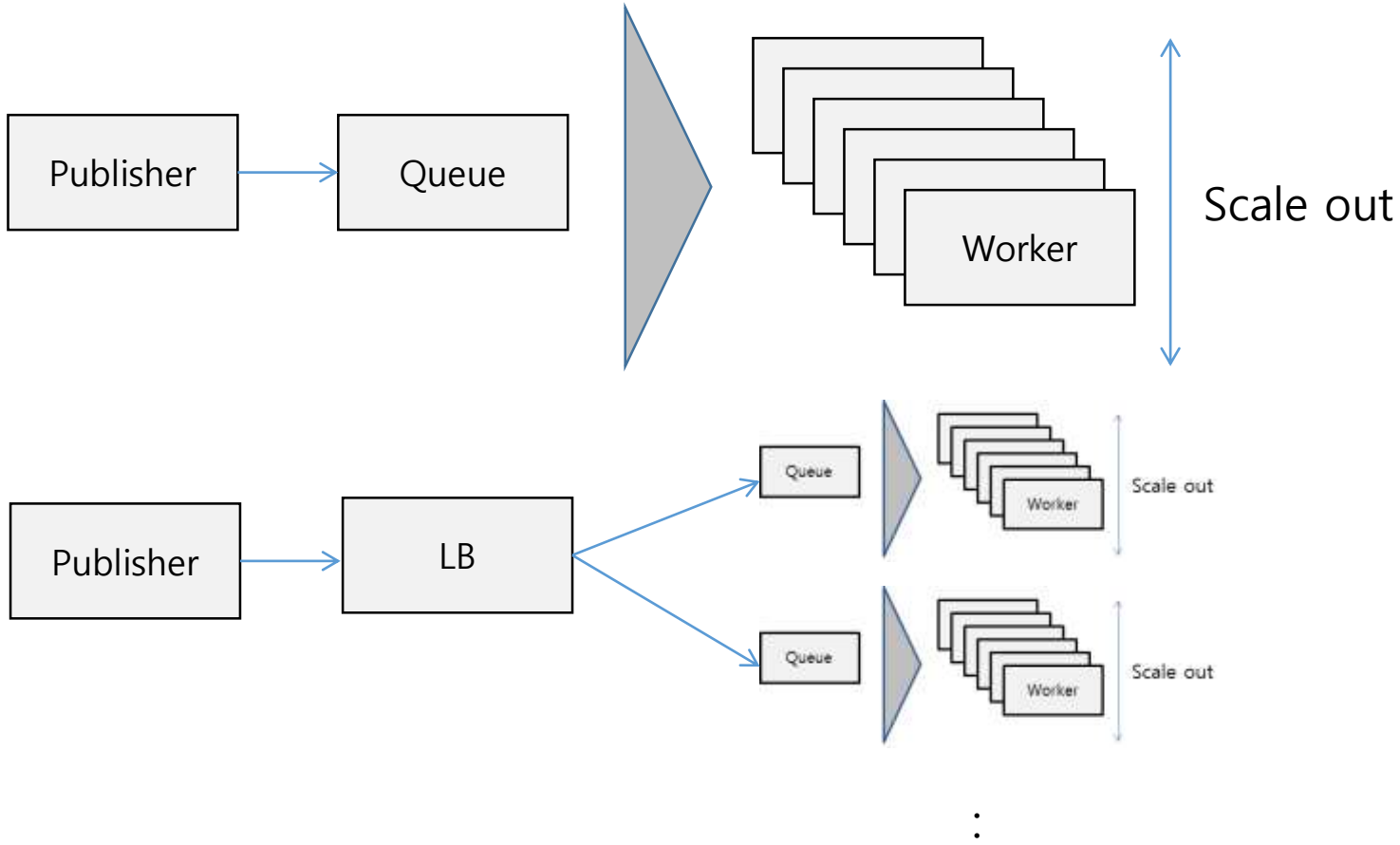




- Good at auto-scale out/in

비동기 패턴

- 대용량 트렌젝션 처리에 유리함
  - 큐 자체에 대한 파티셔닝 (또는 대용량 큐 eg. 카프카)을 고려



#2

대용량 시스템

레퍼런스 아키텍쳐

소프트웨어 개발 트렌드의 변화

- 대규모/긴기간 에서 소규모/단기간 **(스타트업)**
- 빠르고 잦은 릴리즈 **(애자일)**
- 고객의 VOC를 수용 **(빅데이타,SNS)**
- 개발과 운영을 통합 **(DEVOPS)**
- 열심히 일하는 것으로 감당 안됨 **(자동화)**
- 스페샬 리스트에서 제너럴 리스트 **(수퍼엔지니어)**
- 대용량 글로벌 스케일
- 오픈소스
- 구글링,STACKOVERFLOW,블로그,GITHUB

소프트웨어 아키텍쳐의 변화

- 아키텍쳐 스타일의 변화

동기식,중앙 집중형,고가용성 클러스터링

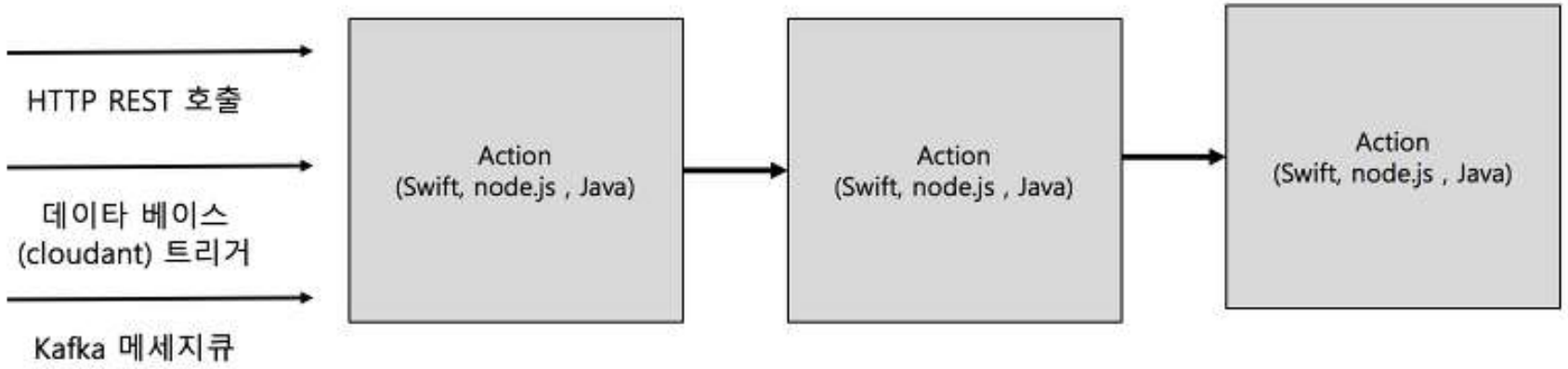
- HTML,Servlet/JSP
- EJB,Spring
- RDBMS
- Data center

비동기식, 분산형, Resilience Shared Nothing

- HTML5,AngularJS
- REST,WebSocket
- Node.JS,IMDG
- NoSQL,Sharding
- Cloud computing

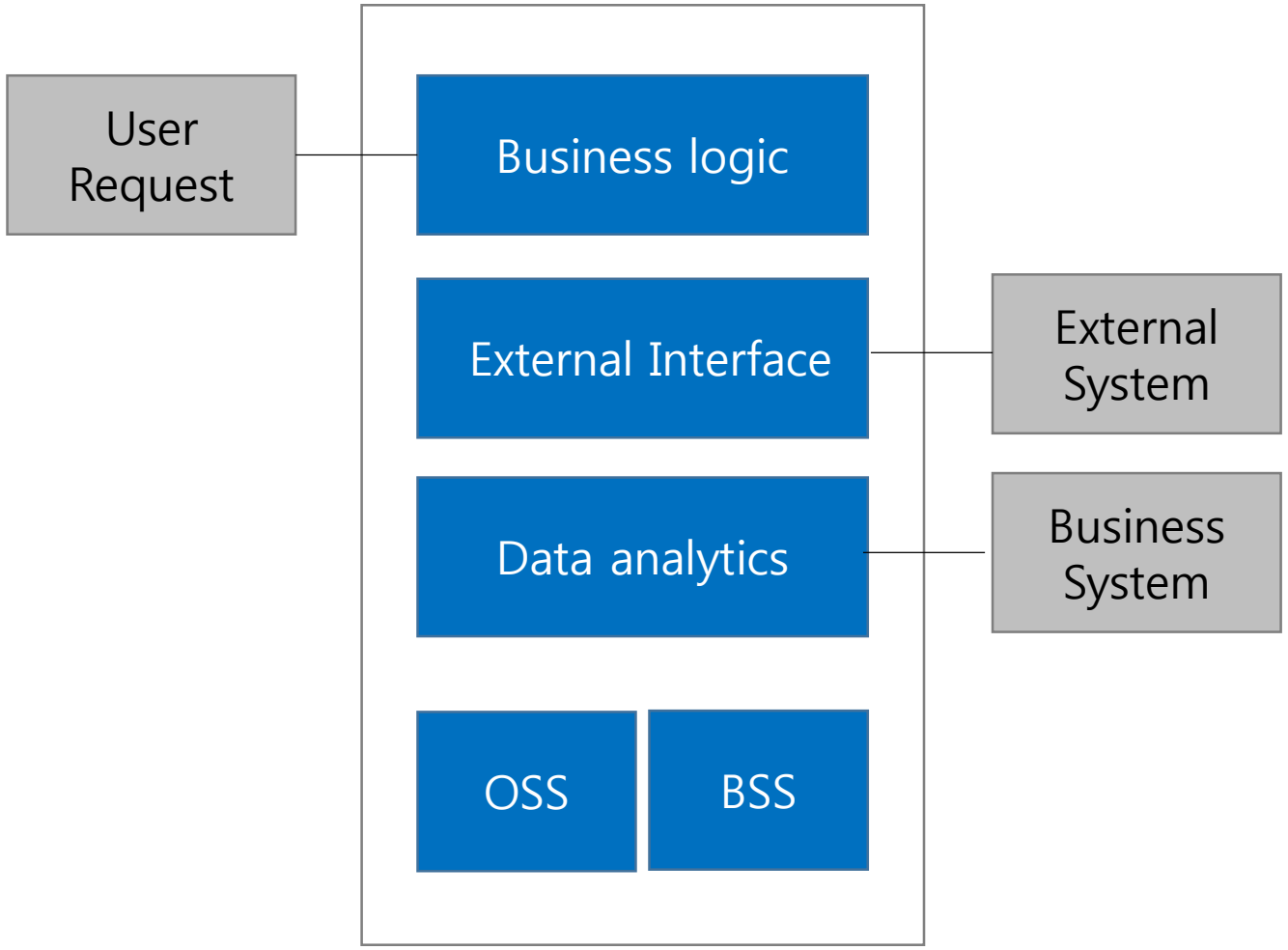
소프트웨어 아키텍쳐의 변화

- 다음 아키텍쳐 스타일 예상
  - PaaS 기반의 조합형 아키텍쳐
  - 서버리스(Serverless) 컴퓨팅
    - 클라우드 인프라 프레임웍이 되는 아키텍쳐
    - AWS Lambda, Azure Function, IBM Bluemix open whisk, Google Cloud function



  - 인공지능 & 데이타 분석 API
    - Google Cloud Vision API, Google Machine Learning, Microsoft Machine Learning API etc
    - 빅데이타 분석 플랫폼 (Yahoo Flurry, Google Analytics etc)

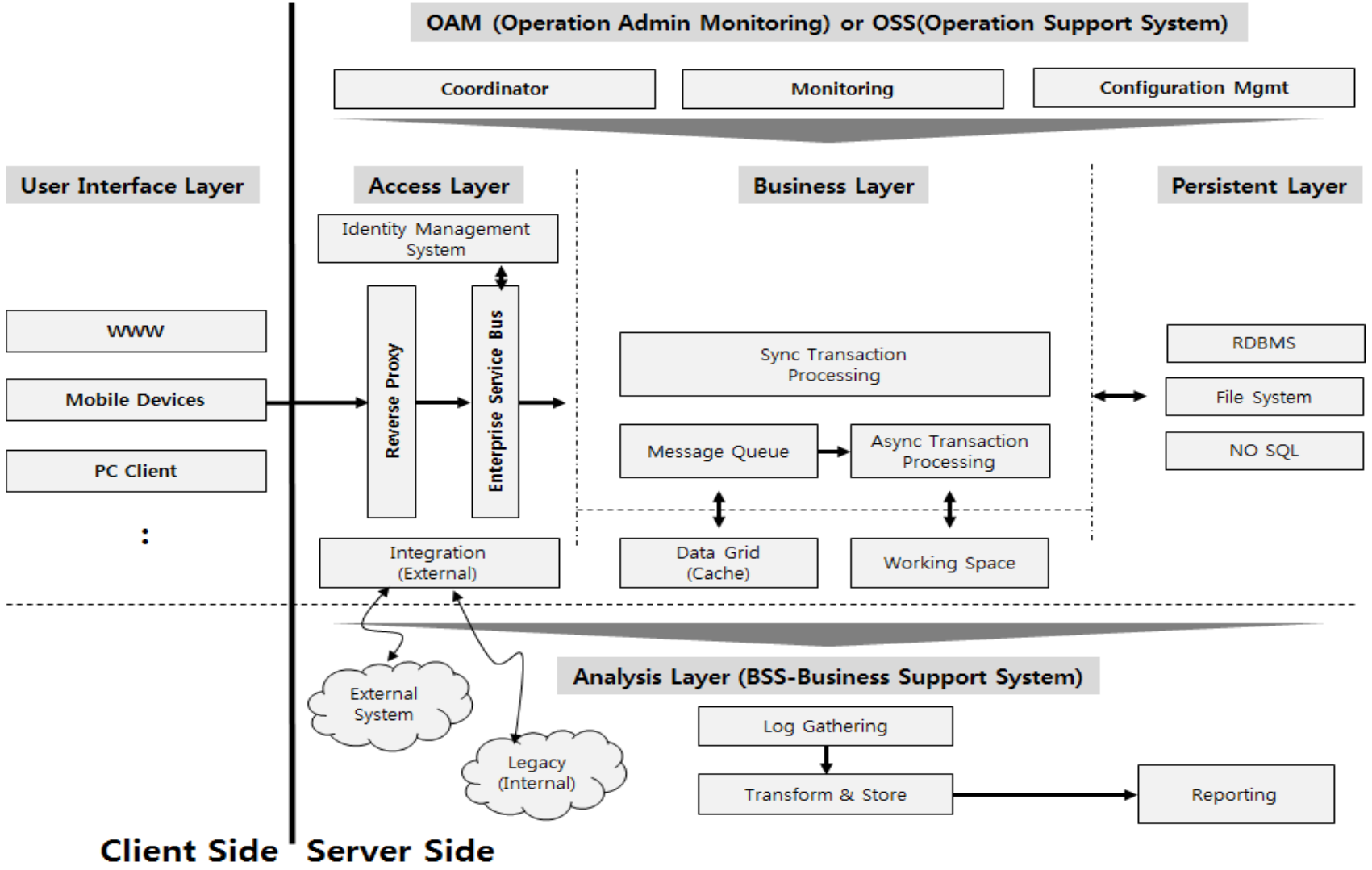
일반적인 시스템의 구조



- Business Logic :
  일반적인 트렌젝션 처리 (서비스)
- External Interface :
  대외 연계
- Data analytics :
  데이타 수집 및 분석/리포트 생성
- OSS (Operation Support System) :
  Tech Ops, Biz Ops
- BSS (Business Support System) :
  리포트, PO 관리

대용량 분산 시스템 아키텍쳐

- SOA 기반의 대용량 분산 아키텍쳐

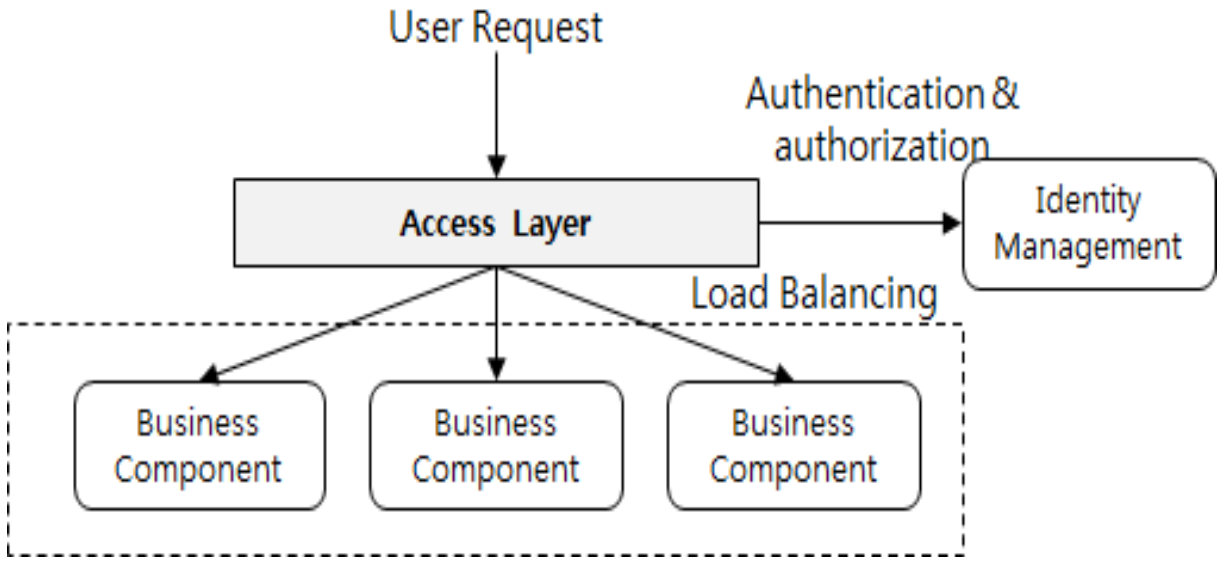


대용량 분산 시스템 아키텍쳐

- SOA 기반의 대용량 분산 아키텍쳐
  - ACCESS LAYER
    - 사용자로 부터 들어오는 요청에 대한 관문
    - 외부 시스템으로의 연계
  - BUSINESS LAYER
    - 요청에 대한 처리를 하는 주요 비지니스 로직
  - PERSISTENT LAYER
    - BUSINESS LAYER에 의해서 처리되는 또는 처리된 데이타를 저장

ACCESS LAYER

- 사용자 API에 대한 END POINT (접점)



REVERSE PROXY

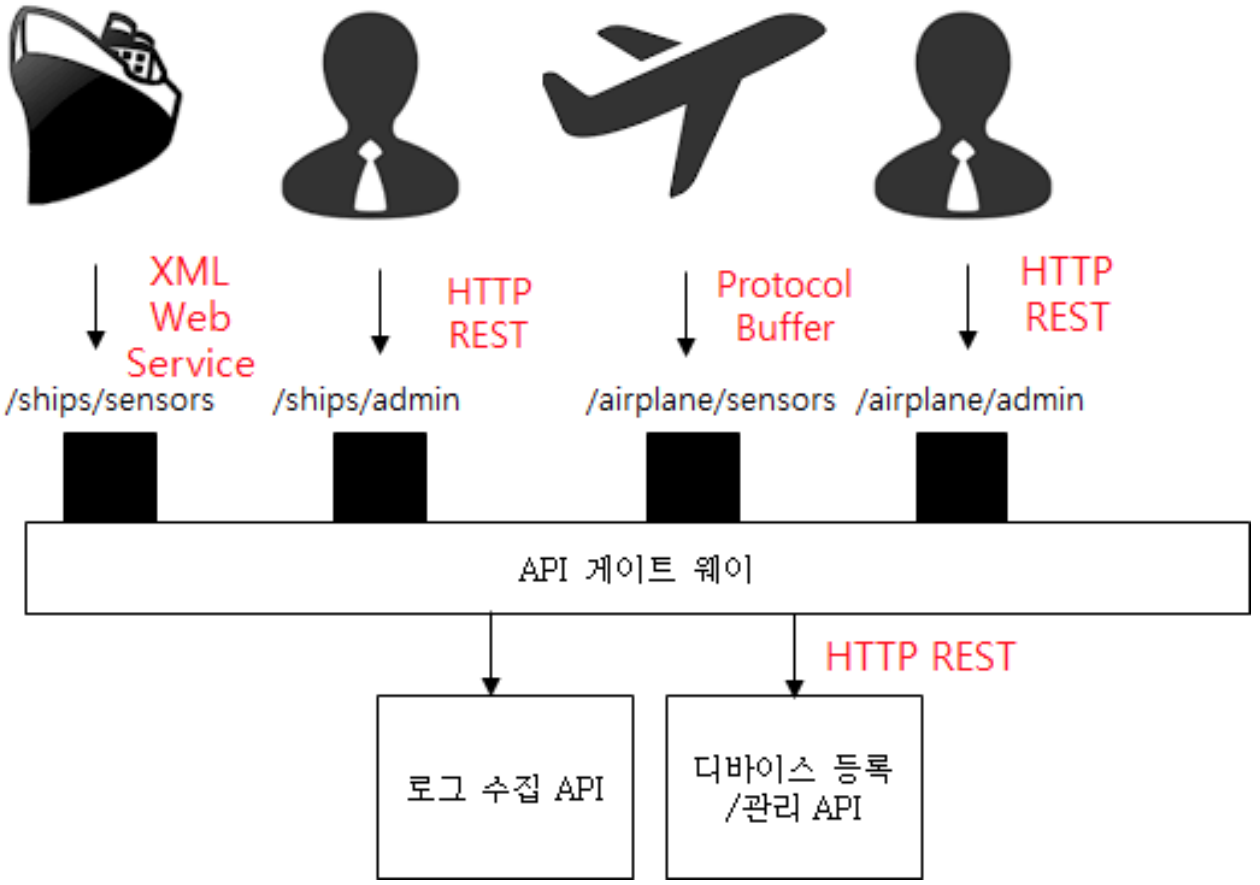
- 부하분산(로그 밸런싱)
- SSL
- IP 블록킹
- 로깅
- 패킷 압축

(NGINX,APACHE,ELB,HAPROXY ..)

ACCESS LAYER

- API 게이트 웨이 (양날의검)
  - API 에 대한 버스(백본) 역할
  - CROSS CUTTING CONCERN 처리 (사용자 인증, 인가, 로깅 처리)
  - 메디에이션 (Mediation)
    - 메세지 라우팅
    - 메세지 변환
    - 메세지 포맷 또는 프로토콜 변환
    - MEP 변환 (동기 ←→비동기)
    - QoS 관린
    - 오케스트레이션

(Strongloop, Tyk.io, APIgee, CA Layer 7)



API 게이트웨이에 대해서는 MSA 부분에서 자세하게 다룸

ACCESS LAYER

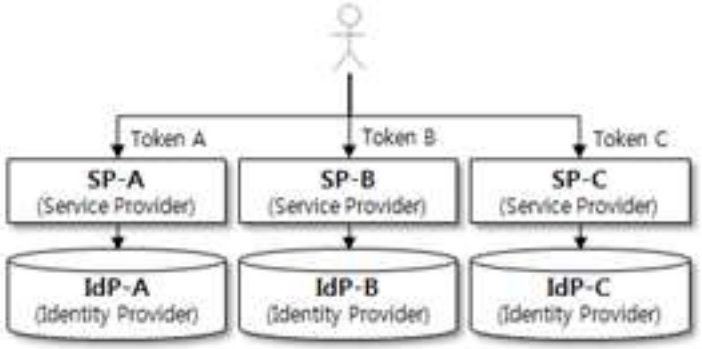
- IDM (Identity management system : 계정 관리 시스템)
  - 사용자 계정 관리 : 사용자 계정 및 사용자 프로파일 정보 저장 및 관리
  - 접근 관리: Authentication, Authorization, Audit
  - 계정 생명 주기 관리 : Identity lifecycle management
  - 페데레이션 : 여러 계정 시스템을 묶어서 연동 (FACEBOOK 계정 로그인, 구글 계정 로그인 등)
  - 프로비저닝 : 계정 정보를 다른 시스템으로 복사
  - SINGLE SIGN ON (SSO)

- 이거 하나로만으로도 하루종일 이야기할 수 있음
- 보통 잘 모르고 시작했다가 나중에 폭망하는 컴포넌트.
  (회사에 몇개의 IDM이 있는건 보통인 경우가 많음)
- WSO2 Identity manager를 보고 개념 잡는것을 추천

ACCESS LAYER

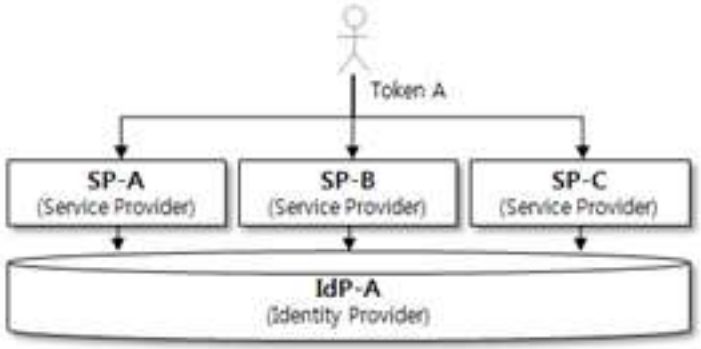
- 토폴로지별 IDM 구축 모델

독립형 계정 관리 모델 (Isolated IDM Model)



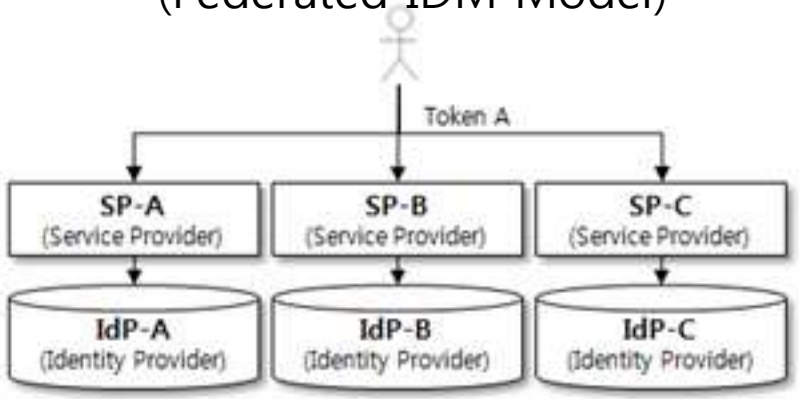
- 시스템별 다른 계정으로 접근
- 시스템별 각자 사용자 정보 저장

중앙 집중형 계정 관리 모델 (Centralized IDM Model)



- 시스템별 같은 계정으로 접근
- 사용자 정보를 중앙에 저장

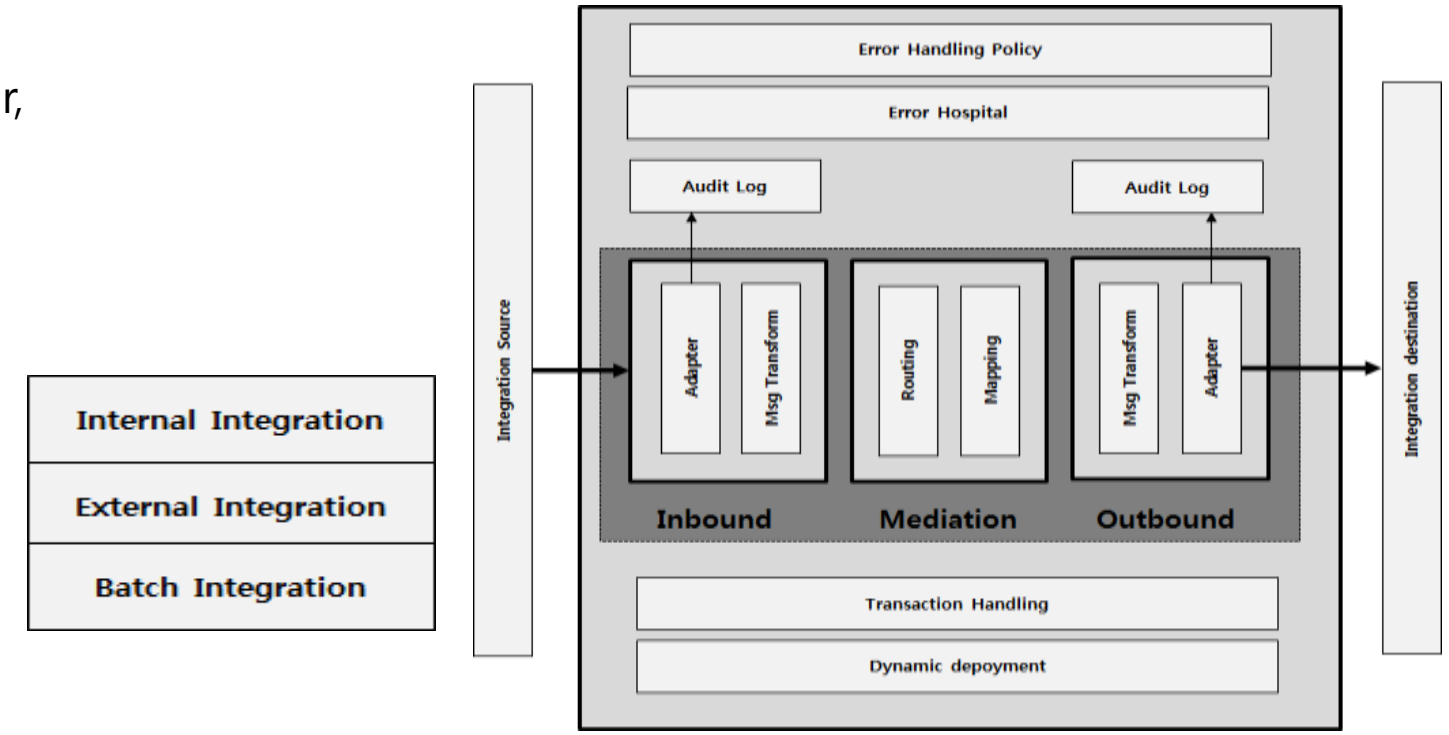
연합형 계정 관리 모델 (Federated IDM Model)



- 시스템별 같은 계정으로 접근
- 시스템별 각자 사용자 정보 저장

ACCESS LAYER

- INTEGRATION (시스템 연계)
  - Integration Type
    - API Integration
    - Native Adapter – JCA, Mecator, TopLink ,WTC
    - Data replication – ETL/CDC
  - Consideration
    - Logging (Audit)
    - Retry
      - Ignore
      - Notification
      - Retry
      - Manual Handling



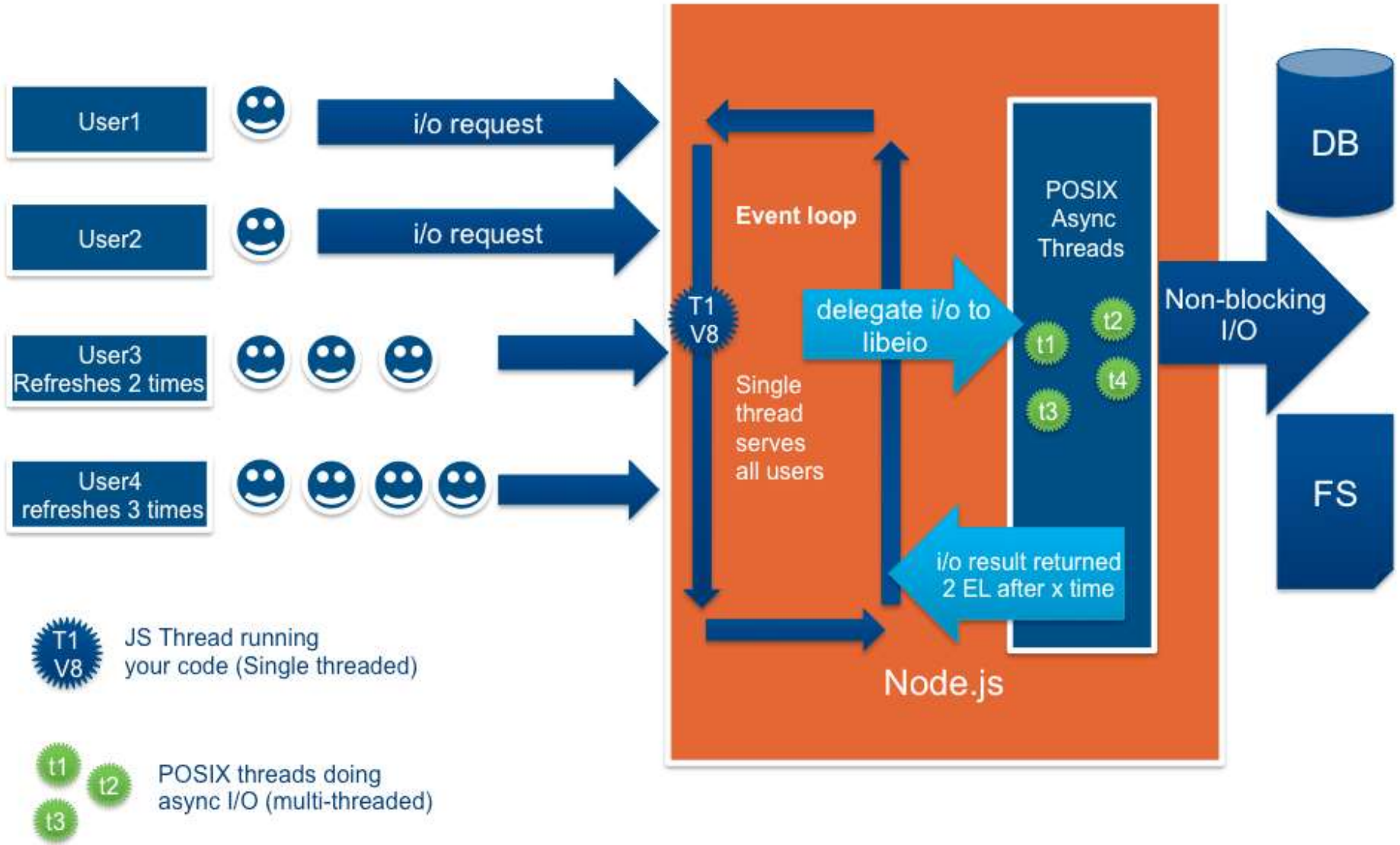
EAI (Enterprise Application Integration) 패턴을 참고 하는 것이 좋음
Apache Camel 프레임웍 참고

BUSINESS LAYER

- 트렌젝션 처리 (동기)
  - 우리가 일반적으로 생각하는 웹서버,애플리케이션 서버
  - 서버에 요청이 들어오면 응답을 주는 구조
  - SHARED NOTHING, STATELESS 구조로 설계하는 것이 좋음
    - 중앙에 데이타 그리드 (레드스등)을 사용하거나
    - 또는 상태 정보를 JWT 토큰등을 이용하여 유지
  - 무거운 트렌젝션 처리, 동시에 적은 사용자만 처리 가능
    - 톰캣과 같은 전통적인 애플리케이션 서버
    - 일반적으로 멀티 쓰레드로 작동
  - 가벼운 트렌젝션 처리, 동시에 많은 사용자 처리 가능 (C10K)
    - 싱글 쓰레드 모델
    - VERTE.X, NODE.JS, NGINX

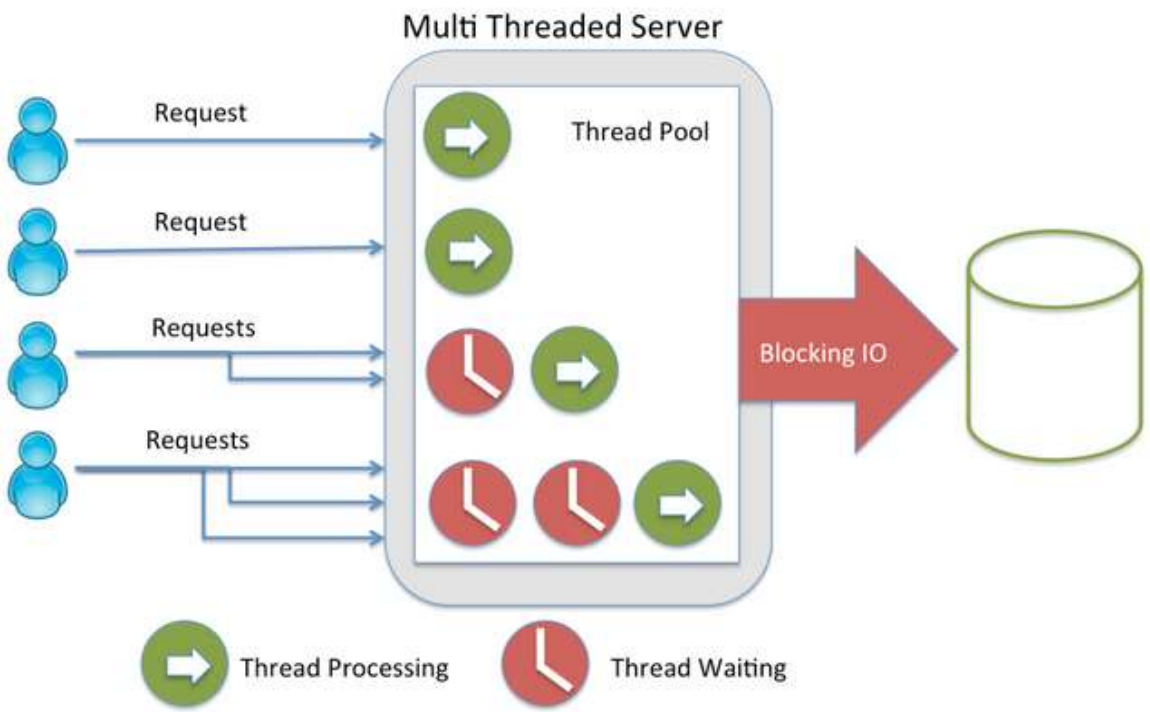
BUSINESS LAYER

- 트렌젝션 처리 (싱글 쓰레드 모델)

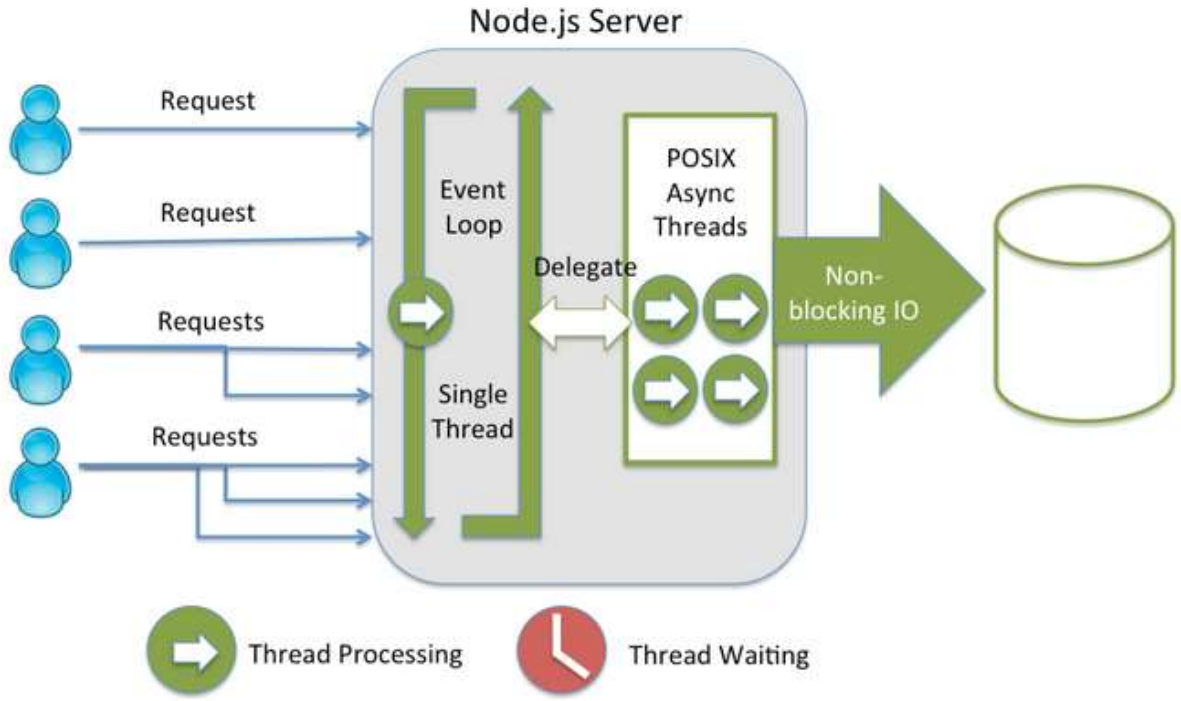


BUSINESS LAYER

트렌젝션 처리 (멀티 쓰레드 모델)



트렌젝션 처리 (싱글 쓰레드 모델)



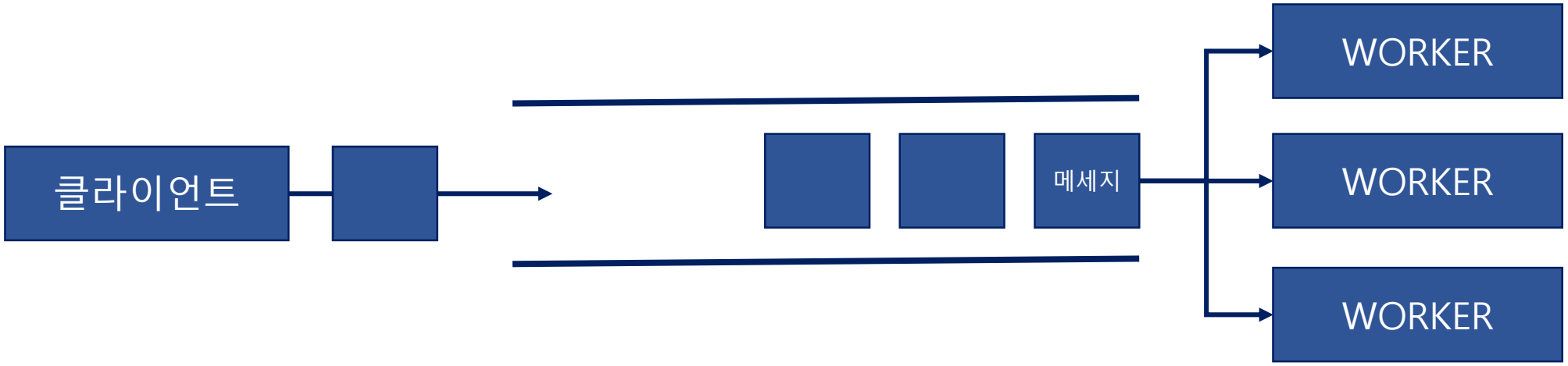
http://strongloop.com/strongblog/node-js-is-faster-than-java/

BUSINESS LAYER

- 분산 트렌젝션 처리
  - 엔터프라이즈 시스템 : XA 기반 2 PHASE-COMMIT - JTS 등 트렌젝션 코디네이터 필요
  - B2C 서비스 시스템 : 보상 트렌젝션 (Compensation Transaction)

BUSINESS LAYER

- 비동기 처리
  - 메세지 큐를 기반 (MQ, RABBIT MQ,KAFKA, SQS etc)
  - 응답을 기다리지 않고 바로 리턴
  - 큐 뒤에 다수의 워커가 메세지를 읽어서 처리 (워커수를 조정하여 대용량 처리가 용이)

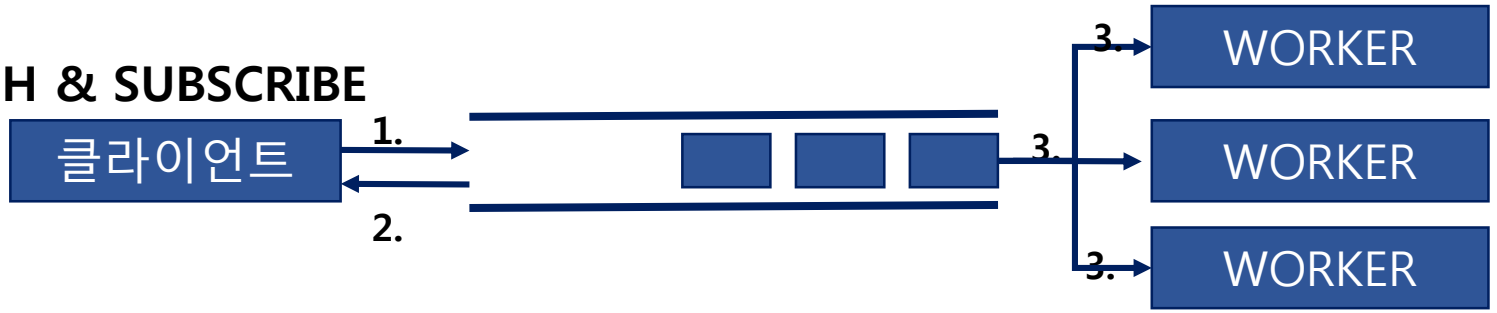


BUSINESS LAYER

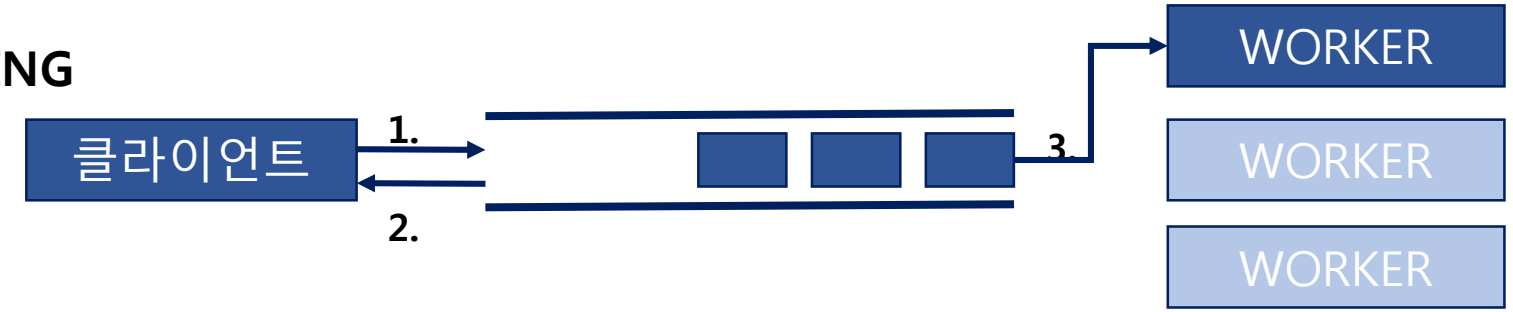
- 비동기 처리 메세지 패턴 (MEP)

FIRE & FORGET

PUBLISH & SUBSCRIBE

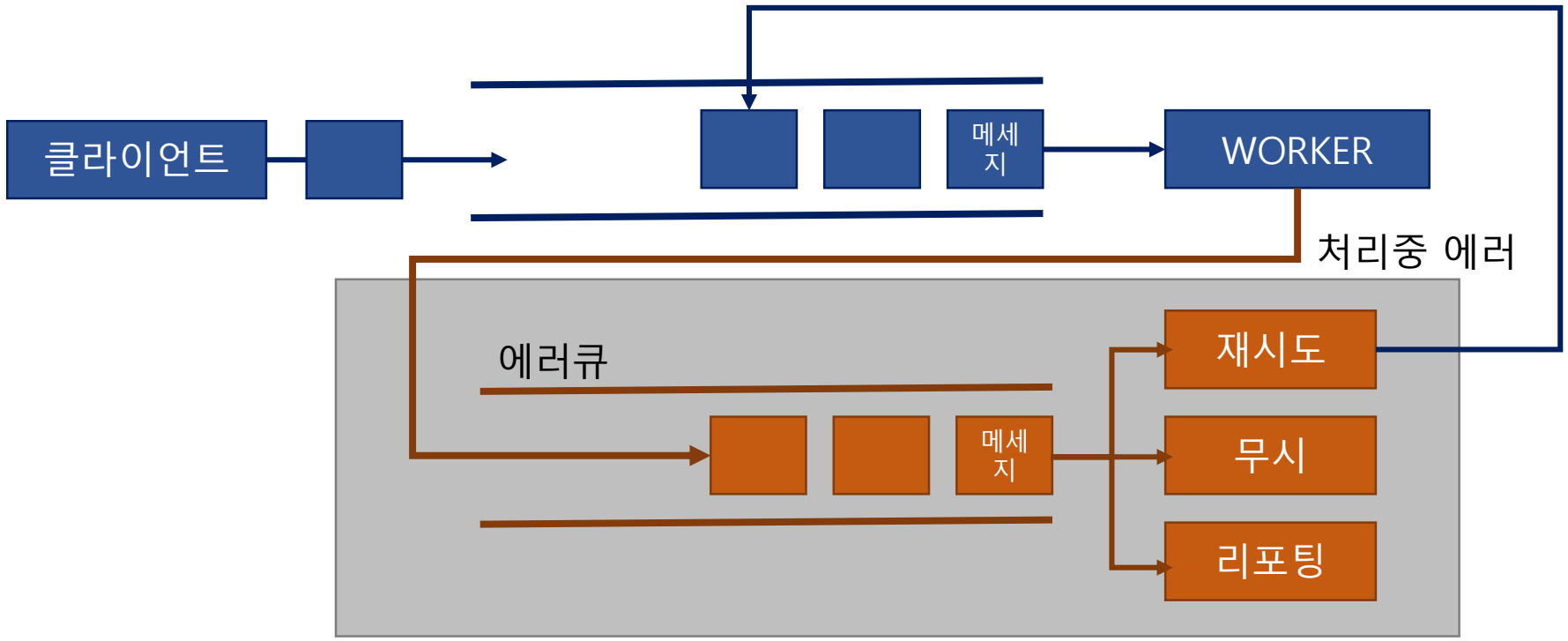


ROUTING



BUSINESS LAYER

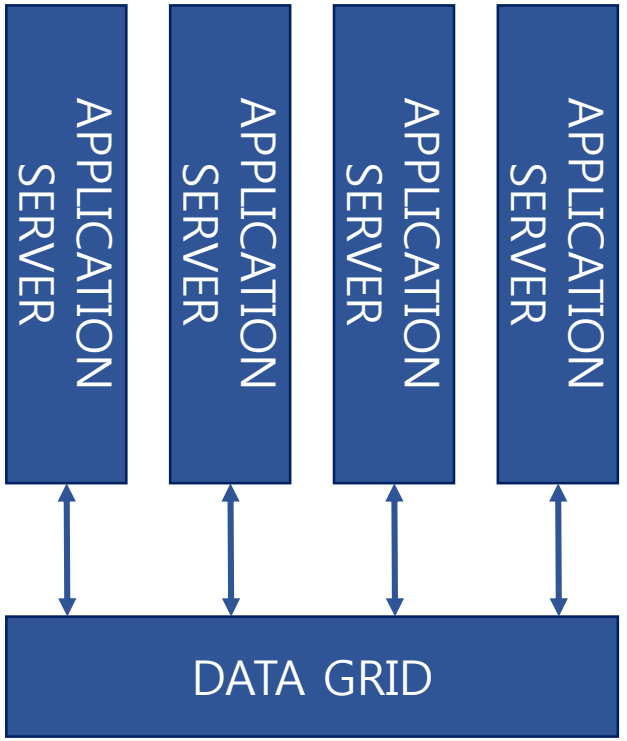
- 비동기 에러 처리
  - 재처리 (RETRY) : AGING 필요
  - 무시
  - 리포팅 (수동 처리)



에러 호스피텔 (ERROR HOSPITAL)

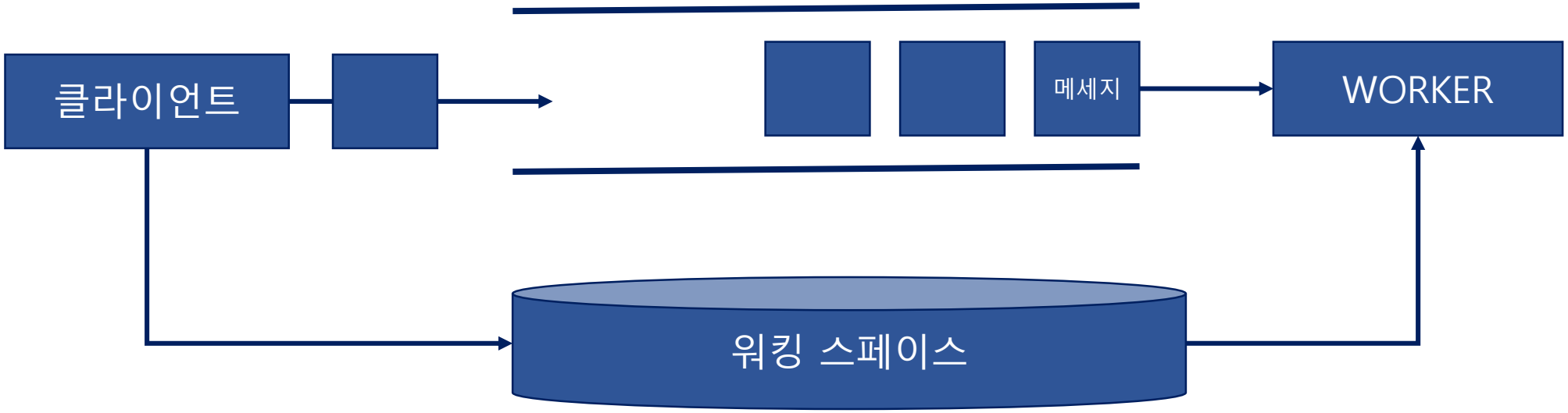
BUSINESS LAYER

- 데이타 그리드
  - 클러스터링된 중앙 공유 메모리
    - 세션 및 상태(STATE) 정보 저장 및 공유
    - REDIS, MEMCACHED 와 같은 메모리 그리드 솔루션을 이용하여 구현됨
    - 자가 클러스터링 기능이 중요



BUSINESS LAYER

- 워킹스페이스 (WORKING SPACE)
  - 작업을 위해서 임시로 파일을 저장하는 곳
  - 파일 업로드, 인코딩등
  - 로컬 파일 시스템을 사용하거나 HA를 대비하여 마운트 가능한 파일 시스템 사용 (NFS, GLUSTER FS)



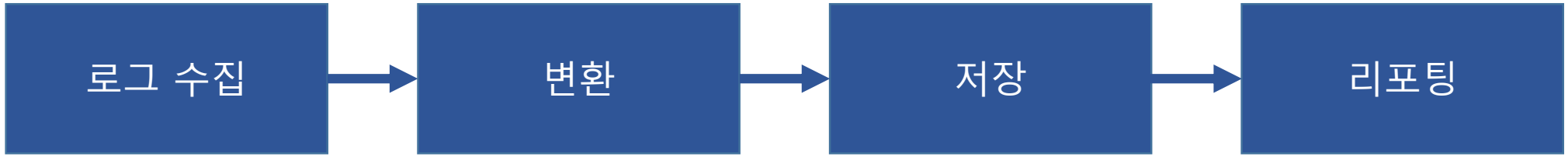
동영상 파일 인코딩에서 워킹 스페이스 활용

PERSISTENCE LAYER

- RDBMS
- NOSQL
  - COLUMN DB (CASSANDRA, HBASE)
  - DOCUMENT DB (MONGODB, COUCHBASE,RIAK)
  - GRAPH DB (NEO4J)
- 파일 시스템
  - 일반 파일 시스템
  - OBJECT STORAGE (AWS S3, AZURE BLOB, SWIFT etc)

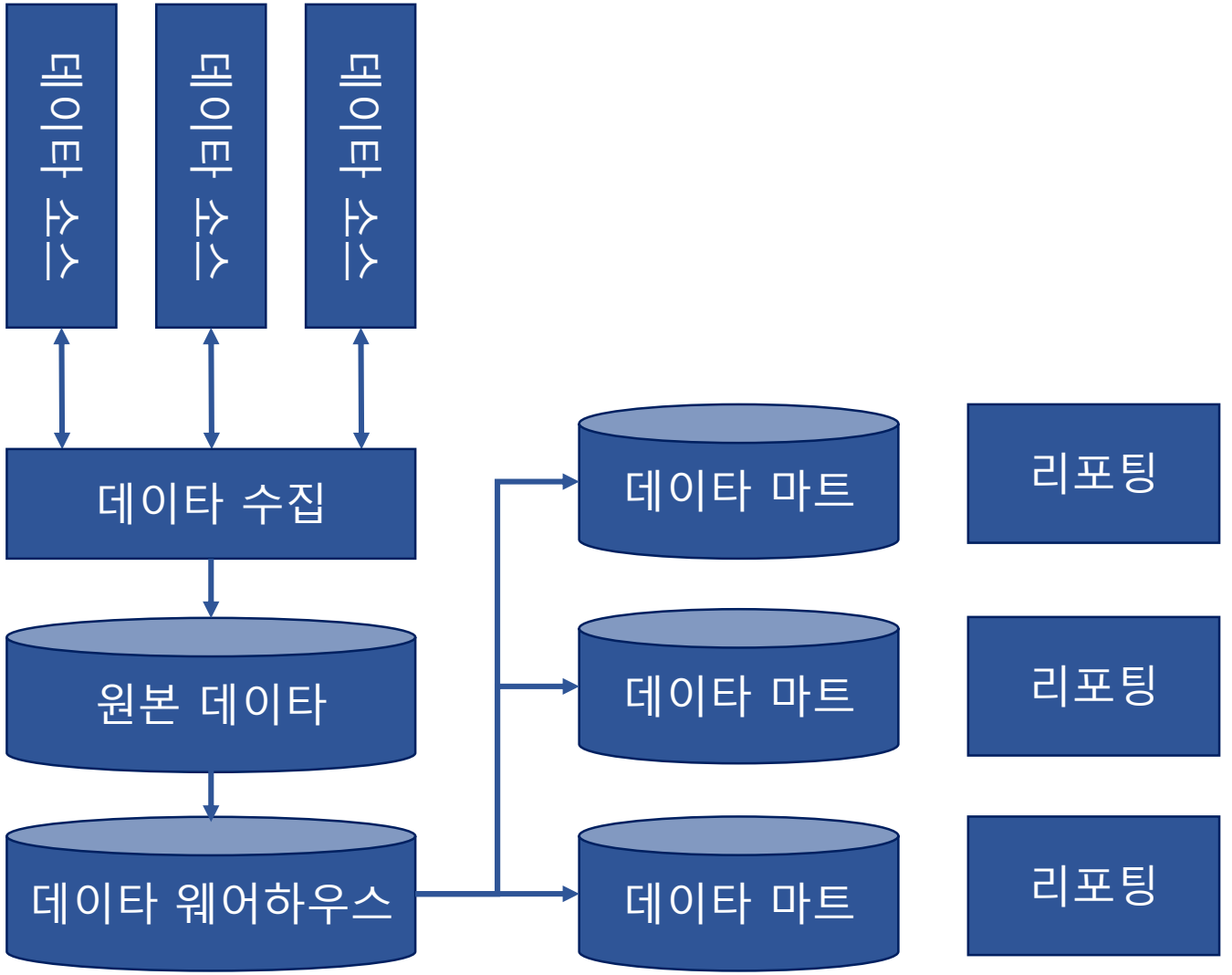
ANALYTICS LAYER

- 데이타 분석 및 리포팅
  - 단순 리포팅 → 인사이트 발견 → 예측
  - 리포팅
    - 리포트 생성 (엑셀)
    - 대쉬 보드 (웹)
    - AD-HOC 쿼리



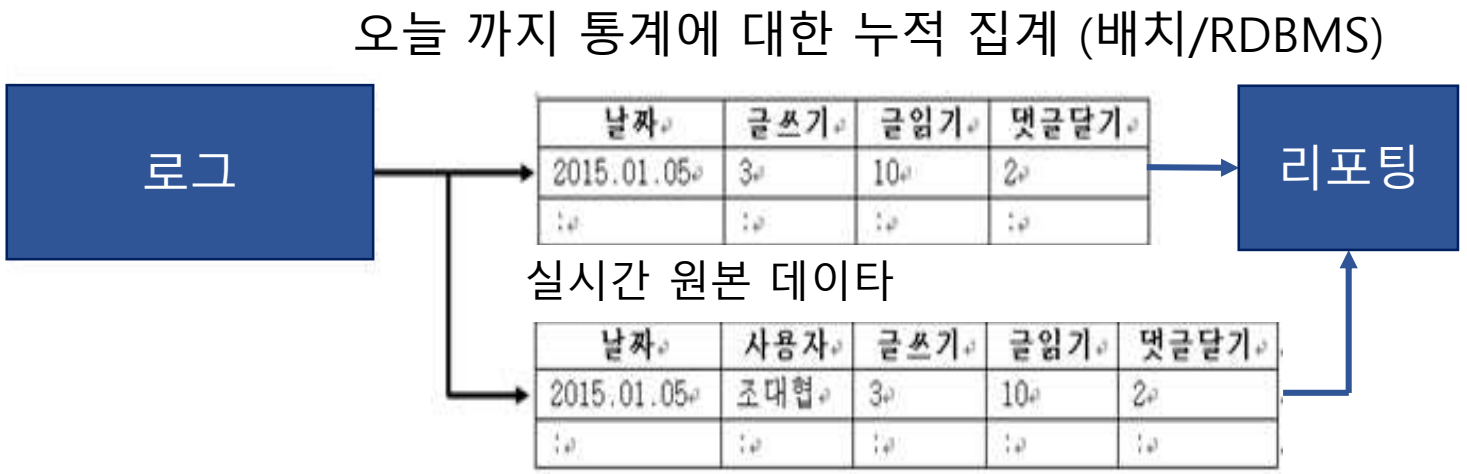
ANALYTICS LAYER

- 전통적인 OLAP 방식의 분석 방법
  - ETL, 로그 수집 솔루션을 이용하여 정재된 데이타를 데이타 웨어하우스에 중앙 저장
  - 부서별 필요한 데이타가 있는 데이타 마트에 데이타를 옮기고 쿼리 수행
  - 실시간 보다는 배치로 수행하는 것이 일반적
  - 빅데이타?
    하드웨어 어플라이언스. ORACLE 엑사 데이타 $$$$



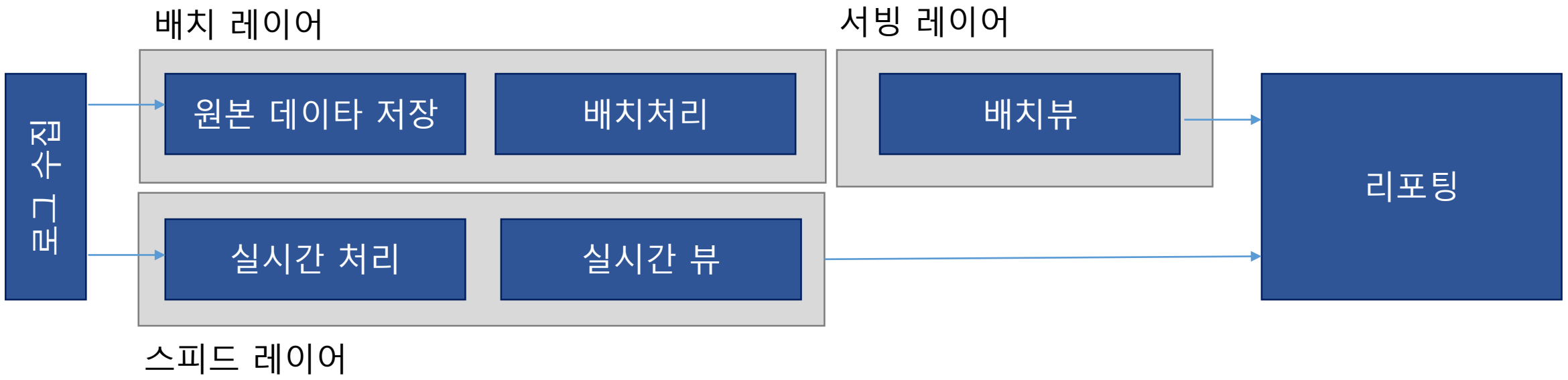
ANALYTICS LAYER

- 람다 아키텍쳐
  - 실시간성 보장
  - 전날까지의 데이타는 배치로 집계값 저장, 실시간 데이타는 원본 데이타를 고속 데이타 저장소에 저장하여 조회



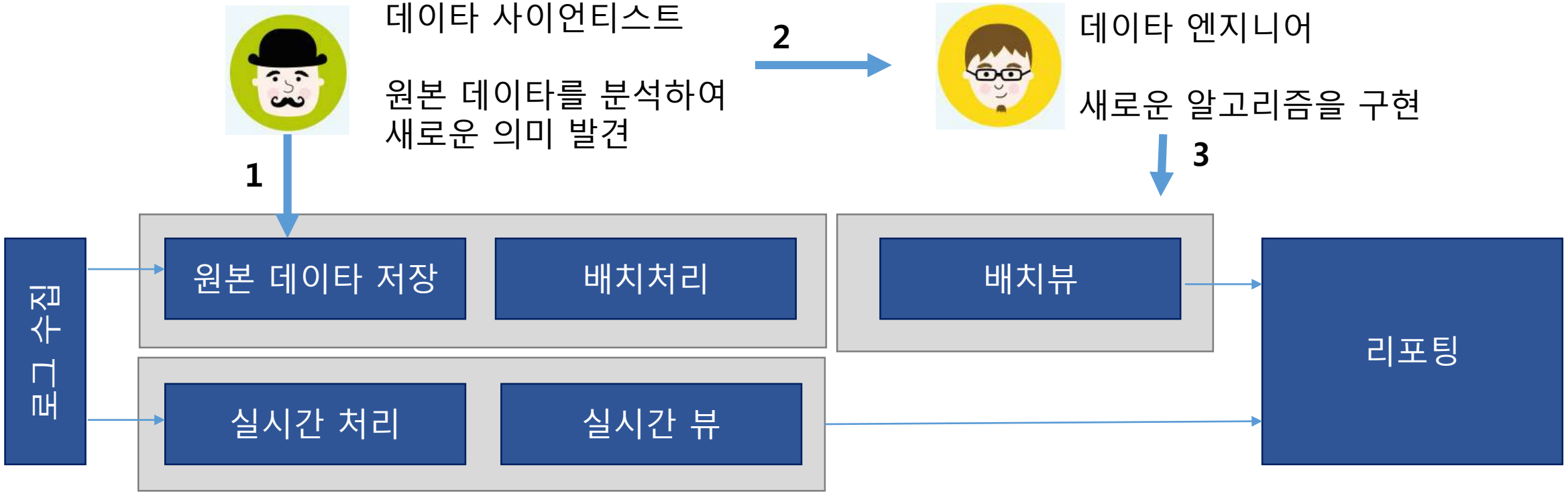
ANALYTICS LAYER

- 람다 아키텍쳐 구성
  - 원본 데이타 저장소 : S3, HDFS
  - 배치 처리 : Map & Reduce
  - 배치 뷰 : AWS Redshift, Hbase, Elastic search
  - 실시간 처리 : Storm, Spark
  - 실시간 뷰 : Redis, RDBMS



ANALYTICS LAYER

- 람다 아키텍쳐 구성



OAM (OPERAION ADMIN & MONITORING) 시스템 운영 관리

- CMDB
  - 애플리케이션의 CONFIG나 메타 정보 저장소
  - ZOOKEEPER,RDBMS
- CONFIGURATION MANAGEMENT
  - 솔루션의 설정 정보를 관리하여, 솔루션 배포나 수정시 일괄 통제
  - CHEF,PUPPET
- 배포 시스템
  - RPM, FABRIC,ANCIBLE
- 모니터링
  - 시스템 대쉬 보드 : NAGIOS
  - 히스토리 모니터링 : CACTI, GANGLIA,ZZABIX
  - APM : JENNIFER, SCOUT
  - 클라우드 모니터링 : NEWRELIC
- 로그 수집 시스템
  - ELK (Elastic search + Logstash + Kibana)
  - 에러 이벤트 핸들링 : Sentry
  - 클라우드 서비스 : LogEntries

글로벌 지원 시스템

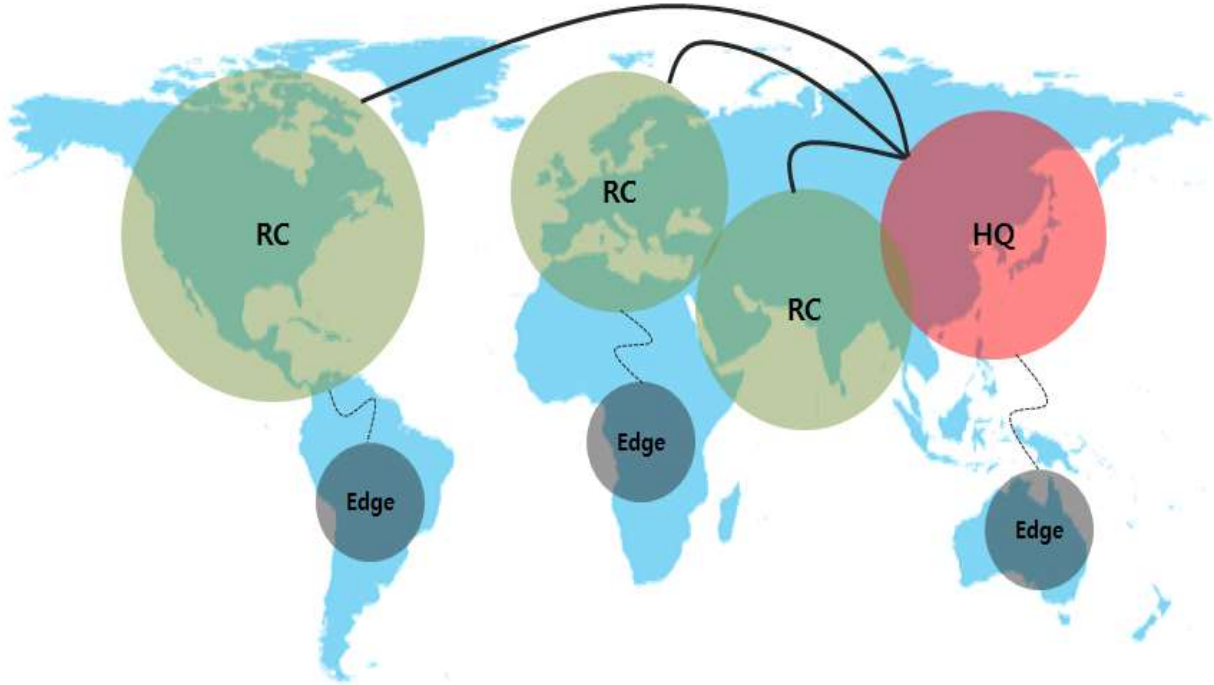
- 고려 사항
  - 레귤레이션 (REGULATION) : 법률 (데이타 위치와 이동, 개인 정보 보호, 결재)
  - 지역별 기술 차이 : 클라우드 리전간 기능 차이 존재
  - 네트웍 레이턴시, 3G/4G 보급율, 모바일 요금제
  - 데이타 센간 데이타 복제
    - DB 복제 : CDC, ETL
    - API 복제 : API 중복 호출
  - GLOBALIZATION (국제화) – 다국어 지원
  - LOCALIZATION (지역화) – 한국은 카톡 로그인, 국가별 멀티 태넌트 시스템

글로벌 지원 시스템

- 위치 선정
  - 법적 이슈 및 세금 (미국, 중국, 유럽 순으로 유리)
  - 네트워크 속도 (더블린, 미서부, 일본)
  - 인력 수급의 용이성
  - 세제 혜택
  - 가격
  - PROCUREMENT (서버 구매등)

글로벌 지원 시스템

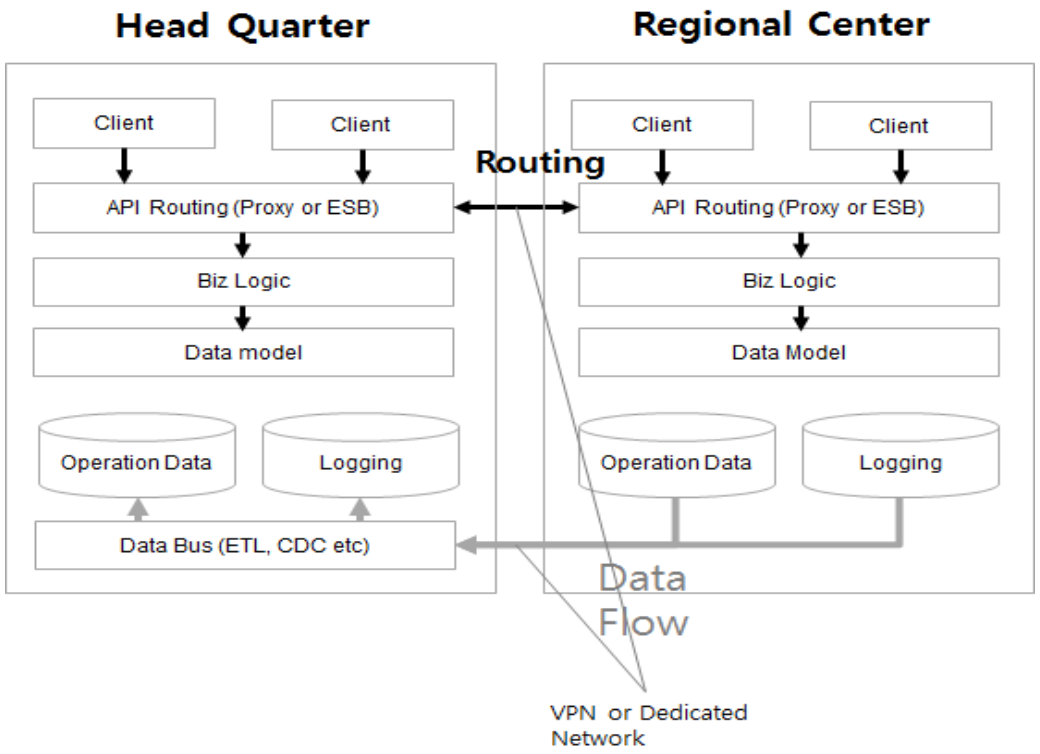
- 구성 방식
  - Master center / Regional center
  - Master / Master center
- 서비스 Look up : 주로 데이타 복제 가능 여부에 따라 디자인
  - 가까운 곳 우선 (데이타 센터간 동기화가 잘 되어 있을 경우 – 주로 근거리 또는 전용망)
  - 특정 데이타 센터 지정 방식 ※ Global Load Balancer 디자인이 관건



사실 요즘 망이 좋아서, 대충 아시아는 일본, 글로벌 서비스는 미국에 넣으면 됨

글로벌 지원 시스템

- Request 라우팅
- 데이타 센터간 복제



클라우드 컴퓨팅

- 퍼브릭 클라우드
  - IaaS,PaaS,SaaS
  - 결코 싸지 않음
  - 같은 서비스라도 지역별로 가격이 다름
  - 생각보다 장애 많이 남 (99.95%)
  - IO가 함정
  - CPU Core가 2000년대 초 Xeon 수준 (요즘 CPU 수준이 아닌 가상화됨)
  - 비쌈!!
  - Infra Service, Fundamental Service
  - IaaS : Amazon(갑), MS Azure, Google Compute engine, IBM Layer 7
  - PaaS : Heroku, Google App Engine, Azure, MongoLab, Cloudant, MS Directory Service
  - 저가 ? Digital Ocean

클라우드 컴퓨팅

- 스타트업 중심의 아키텍쳐 변화 **"만들기 보다는 가져다 조합해 쓴다!!"**
  - 인원이 적으니 어쩔 수 없이 DEVOPS = 되도록이면 managed service
  - Heroku, Google App engine 등의 PaaS 유행
  - 전문 managed service를 이용한 짜 붙이기
    - RedisLabs, Compose IO와 같은 미들웨어 서비스
    - OneSignal, Zencoder와 같은 플랫폼 서비스
- 서버리스 컴퓨팅의 시작
- AI API 활용 (Azure ML, Cloud Vision)
- 데이타 분석 플랫폼의 서비스화 (Flurry, GA 등)

End of document