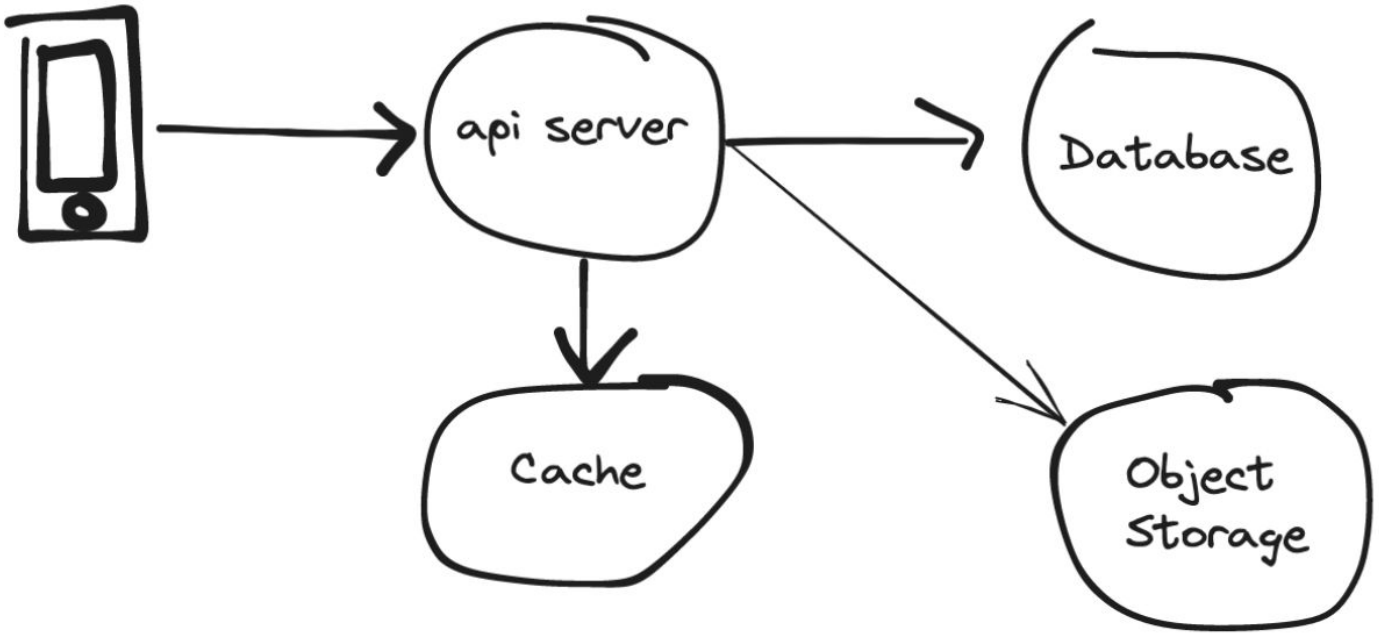
백엔드 시스템 아키텍처

Part 4.

백엔드 시스템 아키텍처

백엔드 시스템이 어떤 아키텍처 구조를 가지고 있는지,
어떤 컴포넌트를 사용하는지, 그리고 각 컴포넌트별로
사용할 수 있는 솔루션에 대해서 살펴본다.

가장 기본적인 아키텍처



API 서버

프로그래밍 언어

- Java (Spring Boot)
- Node.js (Express,Next.js)
- GoLang
- RubyOnRails
- Python (Django,Flask,FastAPI)

런타임

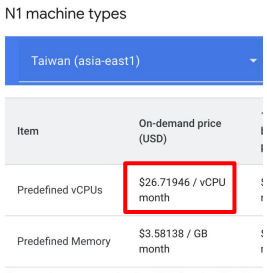
- VM (GCE, EC2)
- Container + VM
- Serverless
  - CloudRun
  - Function
  - Appengine (Standard,Flex)
  - AWS Lambda
- Kubernetes
  (Standard vs AutoPilot)
- Firebase hosting

- 오토스케일링
- 콜드 스타트
- AMD,Intel

CPU 성능

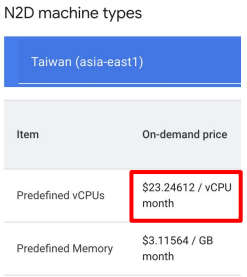
N1 standard VMs

Machine Type	CPU Platform	Operating system	vCPUs	Coremark Score
n1-standard-1	Skylake	ubuntu2204	1	20,060
n1-standard-2	Skylake	ubuntu2204	2	26,293
n1-standard-4	Skylake	ubuntu2204	4	52,091



N2D standard VMs

Machine Type	CPU Platform	Operating system	vCPUs	Coremark Score
n2d-standard-2	Milan	ubuntu2204	2	41,092
n2d-standard-4	Milan	ubuntu2204	4	80,098



출처 :
https://cloud.google.com/compute/docs/coremark-scores-of-vm-instances#n2d_standard_vms

API서버

로드 밸런서

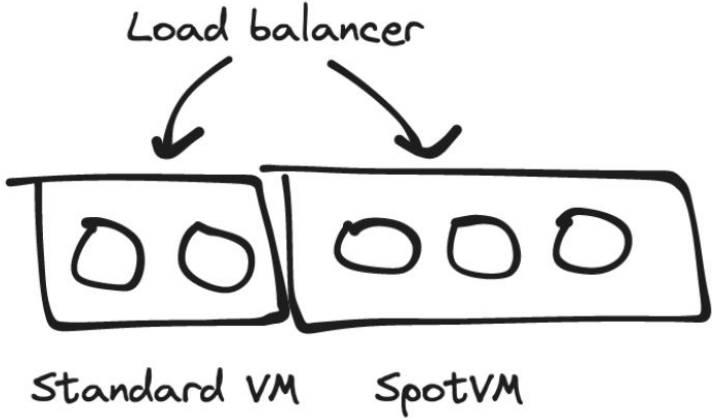
- L4/L7 로드 밸런서
- 소프트웨어 로드밸런서 (nginx,haproxy,envoy etc)
- 웹캐시
- API 게이트 웨이 고려
- 네트워크 (ADN) 고려

WAF

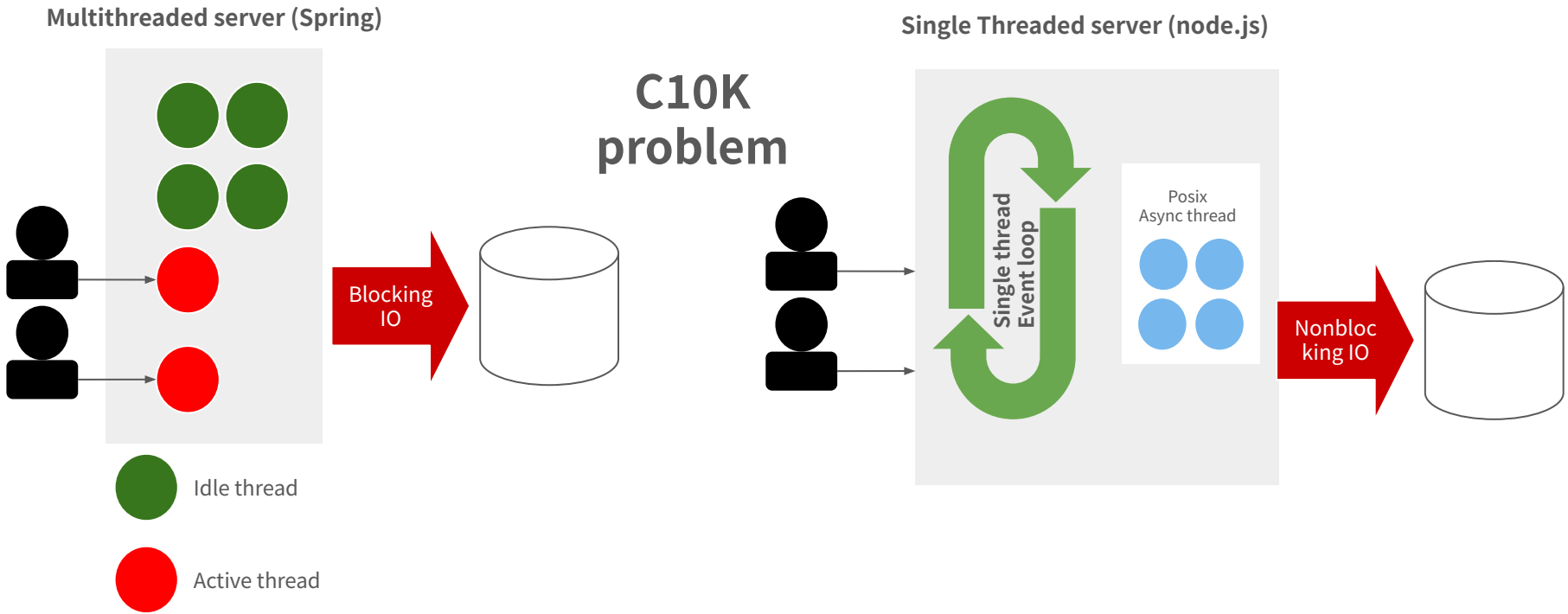
웹 방화벽

Spot VM을 이용한 비용 절감 방안

- API 서버는 stateless 이기 때문에, 리스타트시에 큰 문제가 없음
- 비용 절감을 위해서 일반 VM과 Spot VM을 혼용해서 사용하는 방안 고려



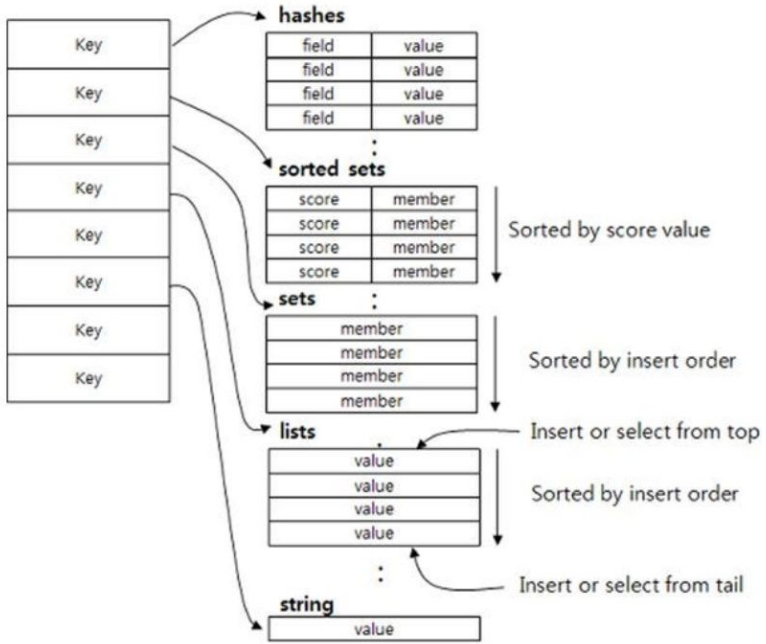
멀티스레드 vs 싱글스레드



캐쉬 서버

캐쉬 서버

- Memcached : 단순 KV 스토어
- Redis
  - 다양한 데이터 타입, 메시지 큐 모델 지원
  - 클러스터링



Redis data model

캐싱 기법

- **TTL**
  - 데이터를 캐시에 저장할때 일정 시간이 지나면 캐시에서 데이터를 삭제함
  - 장점 : 실시간성이 덜 중요한 데이터에 적합하며, 캐시 오버플로우 방지
  - 단점 : 데이터가 업데이트 되었을때, 캐시 데이터와 불일치가 발생할 수 있음.
- **Lazy Loading**
  - 데이터를 캐시에서 조회했을때, 없을 경우 데이터베이스에서 캐시로 데이터를 로딩
  - 장점 : 자주 사용되는 데이터만 캐시에 남음
  - 단점 : 처음 요청에는 반드시 DB에 접근해야 하기 때문에, 캐시 미스율이 높음

캐싱 기법

- **Write through caching**
  - 데이터 변경시, 캐시에 먼저 반영하고 데이터베이스에 변경 내용을 반영
  - 장점 : 캐시와 데이터베이스의 내용이 일관됨
  - 단점 : 데이터 변경 작업이 많을 경우, 캐시 업데이트가 많기 때문에 성능 저하 가능성이 있음
- **Write back**
  - 캐시에 먼저 데이터를 기록하고, 일정 시간후에, DB에 저장하는 방식
  - 장점 : 쓰기 작업이 매우 빠름
  - 단점 : 데이터의 불일치가 발생, 장애 발생시 캐시의 데이터가 유실될 수 있음

Redis Cache 사용 예제 (Flask & MySQL)

```
# Lazy loading with TTL
@app.route('/users/<string:userid>', methods=['GET'])
def get_user(userid):
    try:
        # Redis에서 캐시된 데이터 확인
        user_info = redis_client.get(userid)

        if user_info:
            # Redis에 있으면 JSON을 디코딩해서 리턴
            return jsonify(json.loads(user_info)), 200

        # Redis에 없으면 MySQL에서 데이터 가져오기
        user_info = get_user_from_db(userid)

        if user_info:
            # 가져온 데이터를 Redis에 저장 (캐싱), TTL을 180초로 정의
            redis_client.set(userid, json.dumps(user_info), ex=180)
            return jsonify(user_info), 200
        else:
            # 사용자가 없을 경우
            return jsonify({"error": "User not found"}), 404
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

데이터 베이스

데이터 베이스 솔루션 분류

RDBMS vs NoSQL

RDBMS

- 테이블간의 관계가 있는 경우 유용.
- 가장 범용적으로 사용됨
- 일관성 보장이 됨 (중요한 트랜잭션에 유용)

NoSQL

- 단순하지만 대용량 데이터에 유리
- 일관성 보장이 안될 수 있음
- 비정형 데이터 (JSON 등) 지원 가능

설치형 vs 매니지드

설치형

- 관리가 까다로움
- On-Prem등 다양한 인프라에 설치가능
- 커스터마이제이션이 자유로움
- 다양한 인프라 (NVME Disk, Infiniband)등을 이용하여 성능을 극대화 할 수 있음

매지니드

- 관리가 필요 없음. 특히 분산환경이나 HA 등 복잡한 환경일 수록 관리 부담이 줄어 듬
- 공급자마다 내부 아키텍처를 변경하여 성능과 용량을 추가 제공하는 경우가 있음 (AWS 오로라, 구글 Alloy DB)
- 가격이 상대적으로 높음

오픈소스 vs 유료

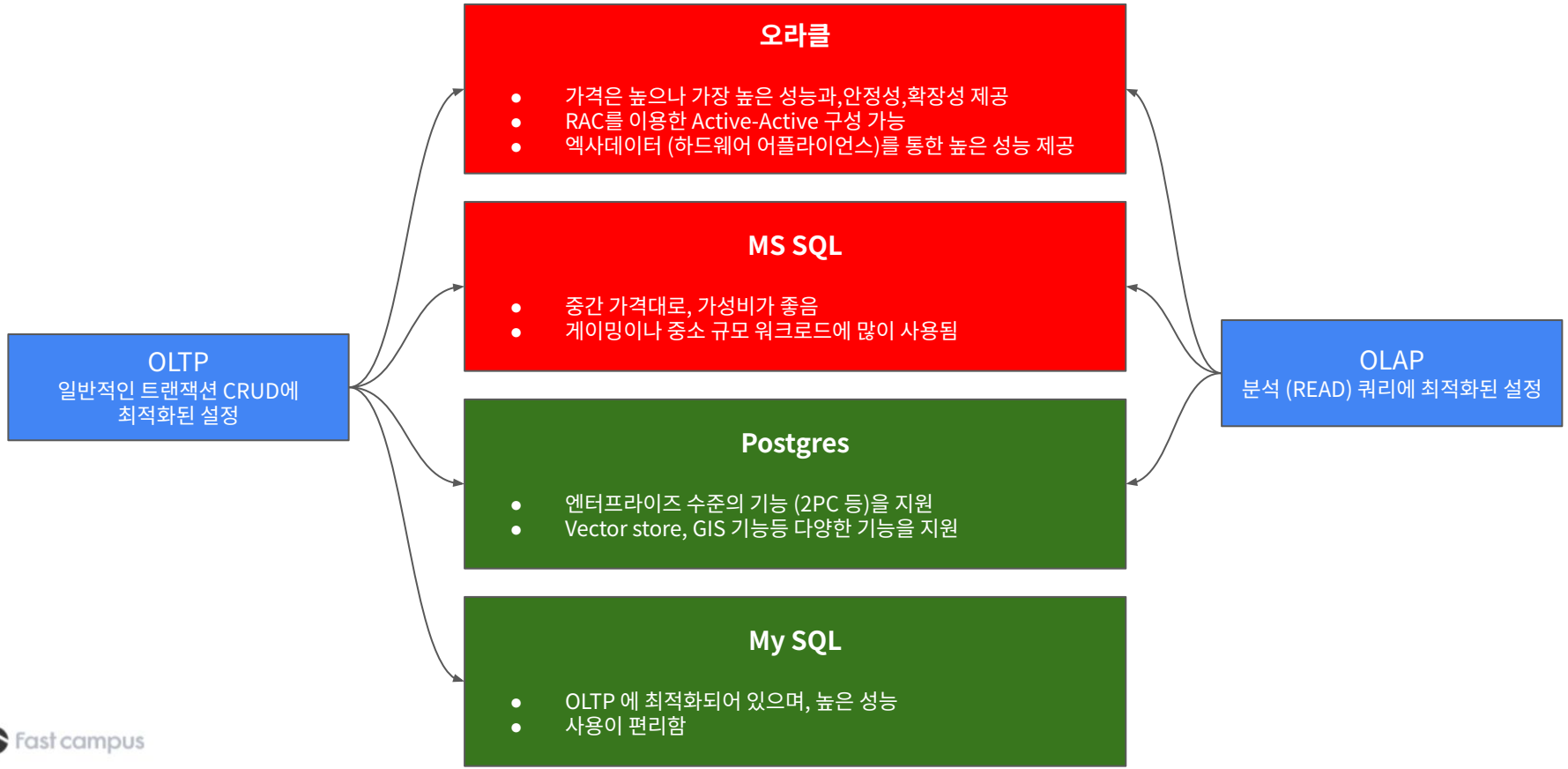
오픈소스

- 저비용이나 기술 지원에 어려움이 많음

유료

- 기술지원이 용이함
- 최적화되거나 차별화된 기능이 많음 (예 Oracle RAC, 엑사 데이터)

데이터 베이스 (RDBMS)



오픈소스 데이터 베이스 (RDBMS)

오픈소스 RDBMS

Postgres

- 엔터프라이즈 수준의 기능 (2PC 등)을 지원
- Vector store, GIS 기능등 다양한 기능을 지원

My SQL

- OLTP 에 최적화되어 있으며, 높은 성능
- 사용이 편리함

클라우드 서비스 형태

커스터마이즈드 서비스

- OSS 서비스와 사용법과 문법은 동일
- 내부 엔진구조를 재설계하여, OSS와 완전히 다른 구조를 가지면서 더 높은 성능과 확장성 제공
- 더 비쌈
- Google AlloyDB (OSS Postgres대비 4배빠름), AWS Aurora DB (MySQL)

오픈소스 매니지드 서비스

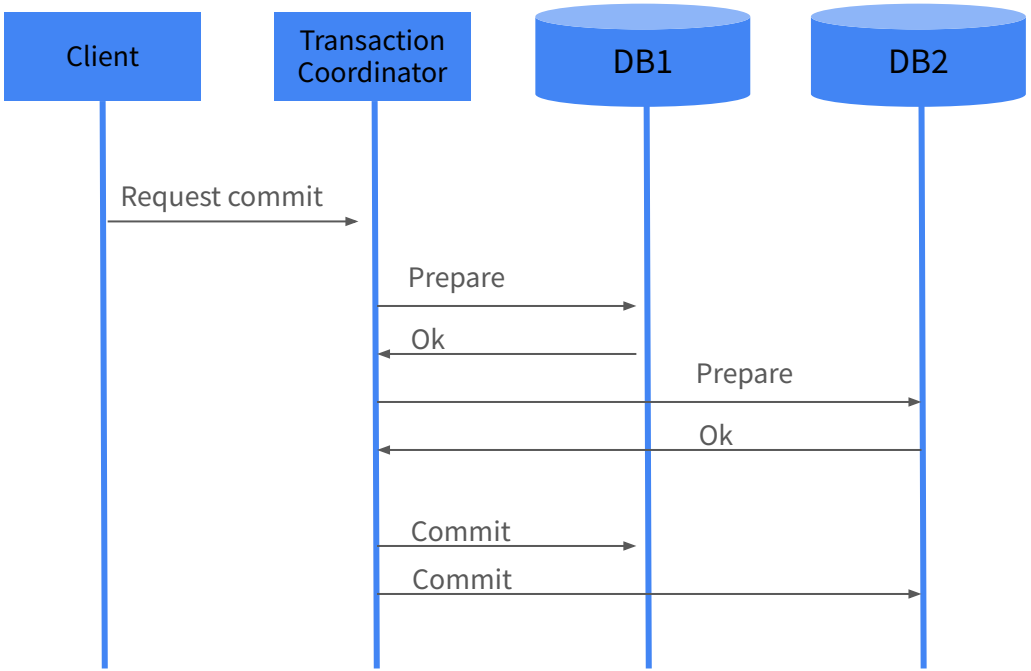
- 오픈소스 RDBMS를 클라우드에서 제공함
- 솔루션의 특징이 오픈소스 RDBMS와 동일함
- Google CloudSQL (MySQL,Postgres), AWS RDS (MySQL,Postgres)

분산 트랜잭션

“ 결제 DB와 주문 DB가 분리되어 있을때, 결재는 성공했는데, 주문이 실패했다면 ?”

- 각각 다른 트랜잭션 이기 때문에, 결재 레코드는 성공으로 저장되고, 주문은 생성되지 않음.
- 이 두개의 트랜잭션으로 묶어서, 같이 성공 처리하거나 실패시 같이 롤백을 해줘야함
- XA (eXtended Architecture)라는 스펙을 지원 하는 트렌젝션 서비스를 사용
- 2PC 을 사용

분산 트랜잭션 (XA-2PC)



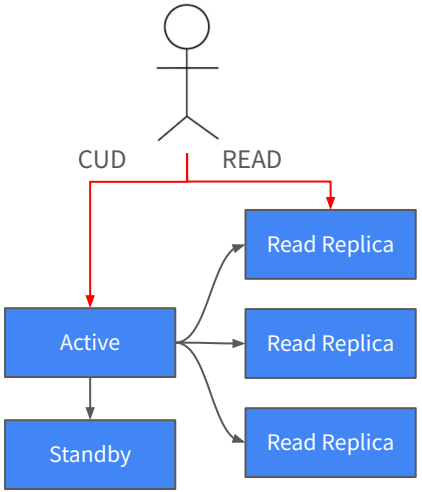
- 두개 이상의 리소스(데이터베이스, 메시지 Q) 에 대해서 트랜잭션 처리
- 리소스가 XA (eXtended Architecture) 프로토콜을 지원해야 함
- 트랜잭션 관리 시스템이 필요함 (자바의 경우 JTS/JTA를 지원하는 WebLogic, JBoss 등)

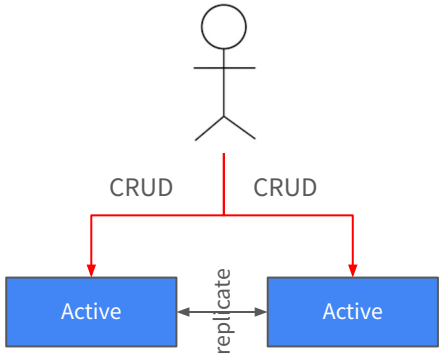
자바에서 분산 트랜잭션 구현 예제

```
            userTransaction = new UserTransactionImp();
            userTransaction.begin();

            // 첫 번째 데이터소스 설정 (예: Oracle)
            AtomikosDataSourceBean dataSource1 = new AtomikosDataSourceBean();
            dataSource1.setUniqueResourceName("OracleDB");
            dataSource1.setXaDataSourceClassName("oracle.jdbc.xa.client.OracleXADataSource");
            :
            // 두 번째 데이터소스 설정 (예: PostgreSQL)
            AtomikosDataSourceBean dataSource2 = new AtomikosDataSourceBean();
            :
            // 첫 번째 데이터베이스 작업
            PreparedStatement stmt1 = connection1.prepareStatement(
                "INSERT INTO accounts (account_id, balance) VALUES (?, ?)");
             :
            // 두 번째 데이터베이스 작업
            PreparedStatement stmt2 = connection2.prepareStatement(
                "INSERT INTO transactions (transaction_id, amount) VALUES (?, ?)");
            // 트랜잭션 커밋
            userTransaction.commit();
             :
        } catch (Exception e) {
            // 오류 발생 시 롤백
            try {
                if (userTransaction != null) {
                    userTransaction.rollback();
```

RDBMS 고가용성 및 스케일링 구조





Active Stand by 구조

- 고가용성 (High Availability)를 지원하기 위한 구조로, Active(Master)노드가 다운 되었을때 Stand by 노드가 마스터 노드로 격상되어서 서비스를 정상적으로 제공함
- 사용하지 않는 Standby 노드를 하나 항상 유지해야 함.

Master Slave 구조

- 확장성을 지원하기 위한 구조로 Master 노드에는 쓰기,업데이트등만 하고, Read Replica(Slave)노드에는 읽기만 함

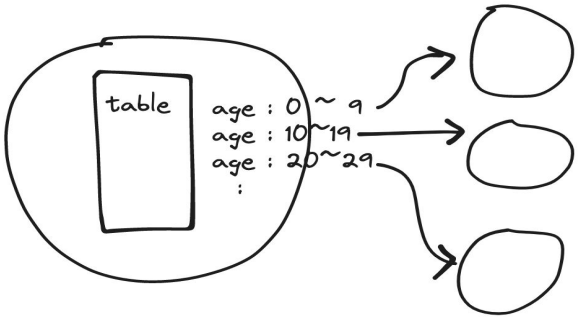
Active Active 구조

- 고가용성과 확장성을 동시에 구현하는 구조로, 여러개의 노드에 동시 쓰기,업데이트,삭제,읽기가 가능함
- 노드간에 트렌젝션을 일관성있게 계속 복제해야 하기 때문에 구현이 어려움
- Oracle RAC, MySQL Galera Cluster 등과 같은 추가 솔루션이 필요함

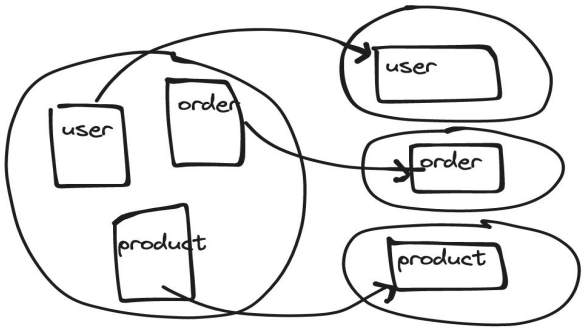
데이터 베이스 샤딩

- 데이터의 전체 양이 방대하여 단일 데이터베이스에 저장하기 어려운 경우, 데이터를 분리된 데이터베이스에 저장할 수 있다.
- 몇 년 전만 해도 동일한 데이터베이스 유형(예: 샤딩된 여러 MySQL)을 사용하는 방식이 자주 활용되었지만, 현재는 서로 다른 데이터베이스로 데이터를 분리하는 방식이 일반적이다(예: 메타데이터는 MySQL, 대규모 데이터(제품, 주문 정보 등)는 NoSQL).
- 만약 고객이 RDBMS(예: 여러 MySQL 샤드)를 사용하고 있다면, 특별히 여러 MySQL을 사용할 이유가 없다면 가장 큰 MySQL을 NoSQL로 마이그레이션하는 것이 좋다.

Vertical sharding
(Not popular)



Horizontal sharding
(more popular)



NoSQL

Not Only SQL

- RDBMS와 다르게 SQL을 사용하지 않는 DBMS의 총칭
- 특징
  - 스키마리스
  - 수평성 확장성
  - 다양한 데이터 모델 (문서, KV,Graph)
- 주로 대규모 데이터와 빠른 읽기 쓰기

NoSQL

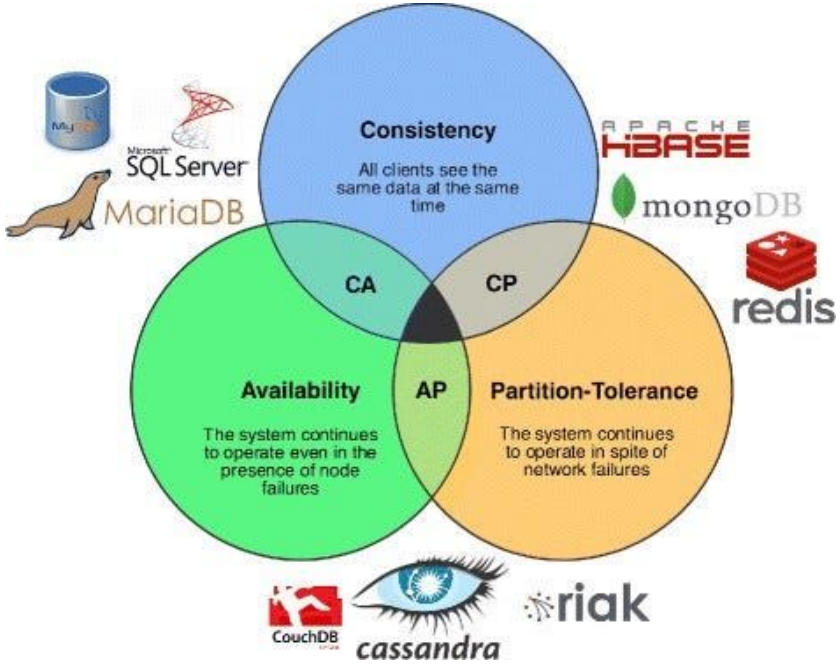
NOSQL 분류

		설명	제품
1	KV 스토어	• 키에 대한 밸류를 저장하는 형태 • 소팅된 키를 사용하는 경우가 많음 • put/get, 커서 이동 정도의 쿼리만 지원 (Join,sorting, where 등은 지원하지 않음)	• HBase • Cassandra • Google Cloud BigTable • AWS Dynamo
2	도큐먼트 DB	• JSON 문서를 통으로 저장 가능 • Where,join 등 복잡한 쿼리를 일부 지원	• MongoDB • Google FireStore
3	그래프 DB	• 그래프 구조의 데이터 베이스 • 노드와 엣지를 통해서 관계를 표현하는데 강함 (노드:배우,영화 엣지 : 출연함, 감독함 등)	• Neo4J • AWS neptune

CAP 이론

분산 데이터 저장소가 다음 세 가지 보장 중 두 가지만 제공할 수 있다는 이론

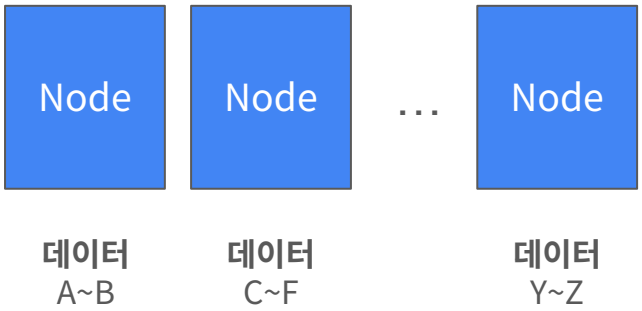
- **일관성 (Consistency)**
  시스템 내의 모든 노드가 동일한 데이터를 반환하는 상태를 의미한다. 즉, 한 노드에서 데이터가 변경되면 모든 다른 노드에 그 변경 사항이 즉시 반영된다.
- **가용성 (Availability)**
  시스템의 일부 노드가 실패하더라도 서비스가 계속 제공될 수 있도록 보장하는 것이다. 즉, 요청에 대해 항상 응답을 반환하는 시스템이다.
- **파티션 허용 (Partition-Tolerance)**
  시스템의 일부 노드가 실패하더라도 서비스가 계속 제공될 수 있도록 보장하는 것이다. 즉, 요청에 대해 항상 응답을 반환하는 시스템이다
- Google Cloud Spanner : 3 가지를 모두 만족



출처 : https://thecustomizewindows.com/

NoSQL 아키텍처

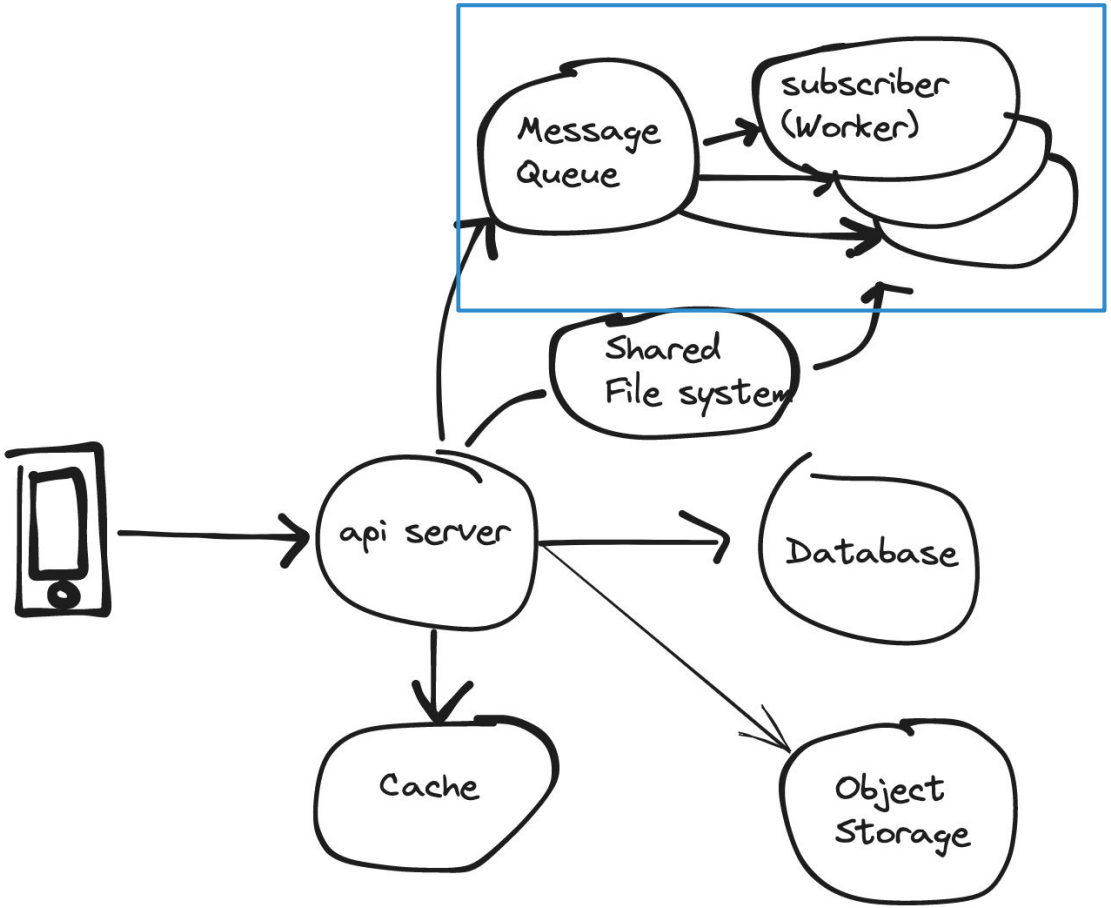
- 일반적으로 분산형 구조를 갖는다.
- 수평적 샤딩을 이용하여, 각 노드마다 하나의 테이블을 분리된 구간별로 데이터를 저장하고 처리한다.
- 특정 키 영역이 자주 액세스 되는 경우 (나이) 이 영역을 핫키라고 한다. → 성능 저하 요인



비동기 아키텍처

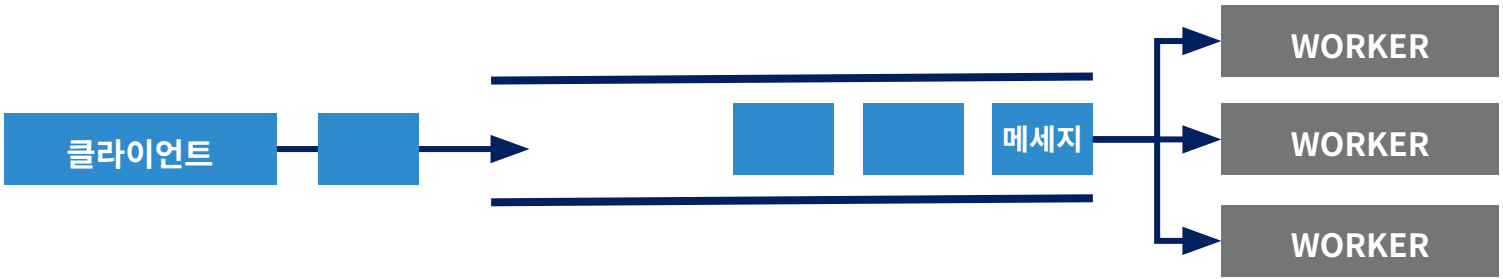
비동기 처리

비동기 아키텍처


Message Queue
subscriber (Worker)
Shared File system
api server
Database
Cache
Object Storage

비동기 처리

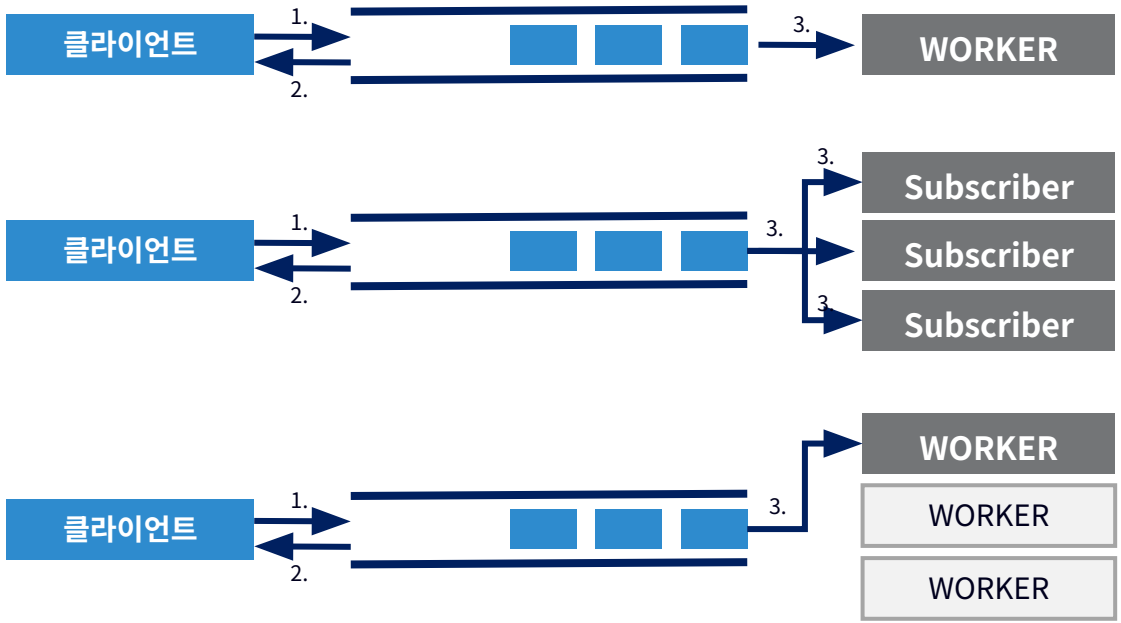
- 메세지 큐를 기반 (MQ, RABBIT MQ,KAFKA, SQS etc)
- 응답을 기다리지 않고 바로 리턴
- 큐 뒤에 다수의 워커가 메세지를 읽어서 처리 (워커수를 조정하여 대용량 처리가 용이)



비동기 처리 패턴

비동기 처리 메시지 패턴 (MEP)

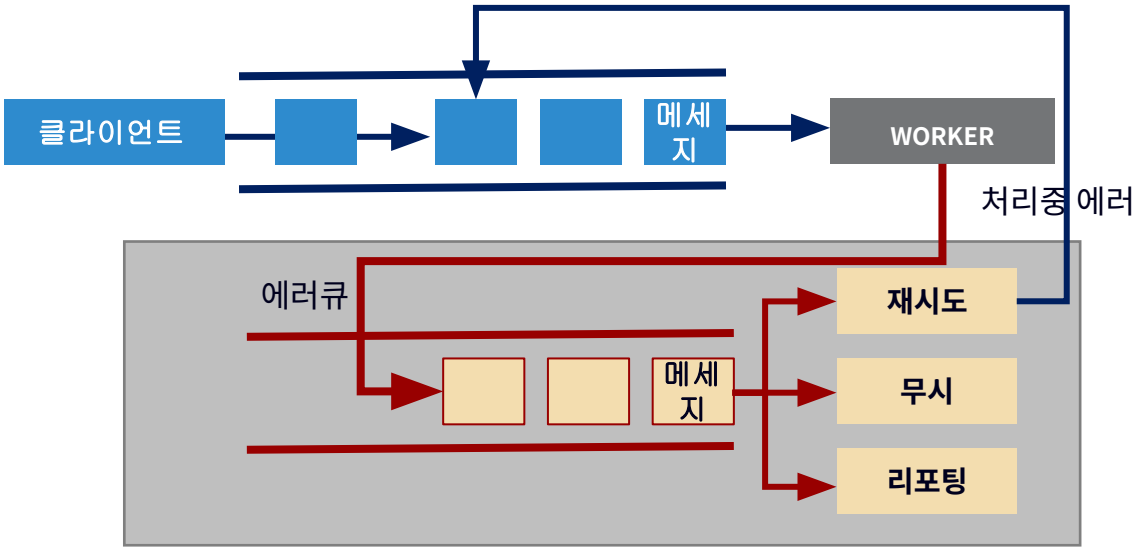
- FIRE & FORGET
- PUBLISH & SUBSCRIBE
- ROUTING



비동기 에러 처리 패턴

비동기 처리에서 실패시 처리 패턴

- 에러호스피탈
- 보통 재시도 후 안되면 사람에게 리포팅해서 차후 재처리
- 메시지 유실이 되면 안되는 엔터프라이즈 시스템에서 많이 사용



에러 호스피텔 (ERROR HOSPITAL)

메시지큐

메시지큐 종류

01	확장 가능한 분산큐	• Kafka • Google Pub/Sub
02	확장이 불가능한 큐	• Active MQ • Redis • Rabbit MQ
03	XA 기반 트랜잭션 지원	• IBM MQ • JMS • Active MQ
04	가장 일반적으로 사용되는 큐	• Redis • Kafka • Rabbit MQ

- 메시지큐들은 대부분 메모리에 메시지를 저장한다.
  그래서, 확장 불가능 큐의 경우, 큐가 다 차면 OOM으로 죽는 경우가 많다.
  많은 메시지가 있는 경우, 이 경우에는 Disk나 외부 DB에 메시지를 저장하는 시스템도 있다.

메시지큐 디자인시 주의사항

01	**순차 보장** 메시지를 클라이언트가 보낸 순서대로 도착해야 한다.	• 메시지를 동일한 큐로 보내고, 단일 소비자를 사용하는 방식으로 처리
02	**메시지 중복 처리** 네트워크 문제나, retry 과정에서 동일한 메시지가 전달되는 경우	• 메시지 ID 를 기반으로 Consumer에서 기록을 유지해서 비교 • Consumer에서 idempotent를 유지하도록 설계 (ex update)
03	**메시지 유실방지** 비정상종료,네트워크문제, 큐의 TTL timeout등으로 메시지가 유실되는 문제	• Persistent disk또는 DBMS에 메시지를 저장 • ACK를 사용하여 메시지가 성공 처리되었음을 저장 • 에러큐를 이용하여, 비정상 메시지에 대한 처리
04	**메시지 크기** 큰 메시지는 성능 저하와 OOM을 유발함	• 메시지는 간결하게 유지 • 대용량 데이터는 별도의 스토리지에 저장 (NFS,S3 etc)하고 메타 정보만 큐에 저장
05	**TTL 설정** 오래된 메시지가 시스템에 남아서 메모리와 디스크를 낭비하는 것을 방지	• 메시지큐의 TTL을 적절히 설정 • 만료된 메시지를 에러큐로 보내거나, 삭제

공유 파일 시스템

- 보통 메시지는 메시지 큐를 통해서 전달한다.
- 메시지가 큰 경우 (이미지 파일,비디오), 메타 정보만 메시지 큐를 통해서 전달하고, 메시지 바디는 별도의 공유 파일 시스템을 통해서 전달한다. (메시지 메타 정보에 파일 경로 저장)
- HPC, 머신러닝에도 사용됨

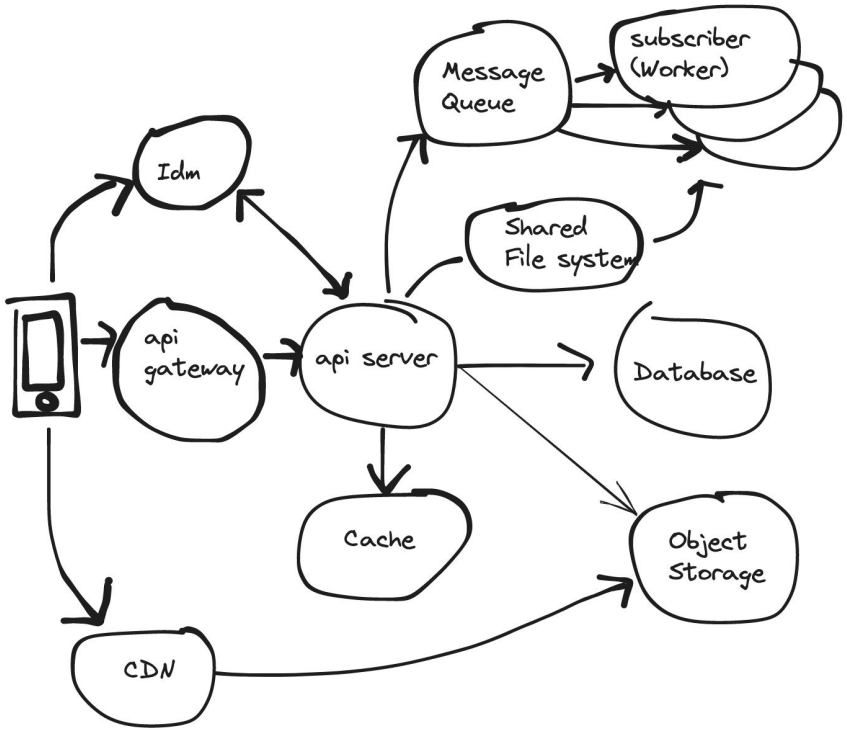
여러개의 워커들이 파일을 공유하는 공간

공유 파일 시스템

		장단점	사용 용도
1	NFS	• 장점 : Linux 배포판에 기본 탑재됨, 사용이 편리함 • 단점 : 확장성 부족, 성능 저하	• 소규모 네트워크 파일 공유 • 소규모 머신러닝, HPC
2	NFS appliance	• 장점 : OSS NFS에 비하여 빠른 성능 • 단점 : 확장성 부족 (100~500TB),가격이 매우 높음	• HPC, 일부 DB 등 다양한 시나리오 사용 가능
3	Blob (AWS S3, GCP GCS etc)	• 장점 : 대용량, 저비용 • 단점 : 느림 (SSD Local 캐쉬로 어느정도 해결 가능)	• 대용량 파일 저장 • 대규모 머신러닝 학습 및 HPC
4	병렬 스토리지 (DAOS, Lustre)	• 장점 : 대용량, 매우 빠름 • 단점 : 러닝 커브가 매우 높고, 운영이 어려움	• 대규모 머신러닝 학습 및 HPC

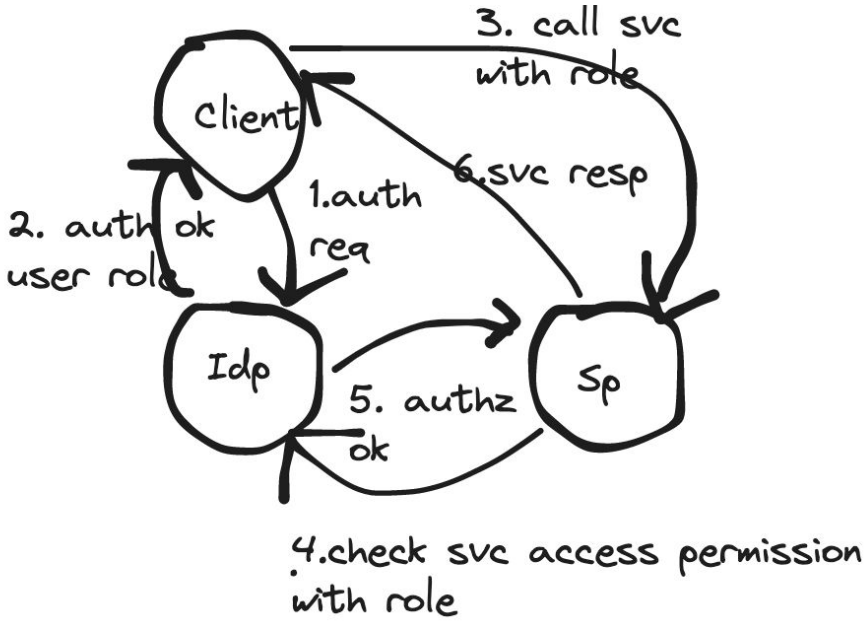
엔터프라이즈 아키텍처

엔터프라이즈 수준의 아키텍처



계정 관리 시스템 (IDM)

- IDM (Identity management system)은 사용자의 계정을 관리하고, 사용자에 대한 인증(Authentication)과 인가(Authorization)을 수행한다.
- **Idp & Sp**
  - Idp (Identity provider)
    사용자 정보를 저장하고, 이에 대한 인증을 수행함. (예시 구글 인증, 카카오 인증 등)
  - Sp(Service provider)
    인증이 된후, 서비스를 제공하는 시스템
- **분류**
  - 사내 인증 시스템
    다양한 시스템에 대한 지원이 가능해야 하며, 보통 이메일 시스템을 Idp로 사용함
    Microsoft Active Directory, Google Gmail
  - 대외 인증 시스템
    대용량 사용자를 지원할 수 있어야 함.
    RDBMS로 보통 구현



계정 관리 시스템 디자인시 주의 사항

보안

- 최소 권한
- 다단계 인증 (MFA)
- RBAC (Role based authorization)
- 계정 생명 주기 (생성 및 **폐기,**비밀번호 업데이트)

확장성과 유연성

- 사용자수나 애플리케이션 종류 증가에 따른 확장
- 다양한 프로토콜 (SAML,OAuth 2, Open ID) 및 레거시 지원
- SSO 지원
- 모바일 친화 환경 (BYOD 지원등)

규제 준수

- GDPR (유럽 개인 정보 보호법), ISMS(한국 정보 보안 규정) 준수
- 데이터 보호 및 암호화

초반에 설계를 잘해놓지 않으면, 한 회사내에 여러개의 계정 시스템이 존재하는 경우가 발생할 수 있음.

IDM 솔루션

		설명
1	WSO2 Identity Server	• 상용, 오픈소스 모두 지원 • IDM에 필요한 거의 모든 기능을 지원함으로써 공부하기 매우 좋음
2	KeyCloak	• 클라우드,온프렘 모두 지원 • SSO,MFA,소셜 로그인 지원 • 사용자 친화적 콘솔
3	Apache syncope	• Ldap,AD등 디렉토리 서비스 연동 • 라이프사이클 관리 기능

오픈소스

		설명
1	Microsoft AD	• MS 생태계에 최적화됨 (윈도우) • 온프렘 지원
2	Okta	• 클라우드 기반,엔터프라이즈레벨 • 광범위한 애플리케이션 지원 • 가격이 비싸고, 온프렘에 제약
3	OneLogin	• 클라우드 기반, 중소 기업 레벨 • 광범위한 애플리케이션 지원 • 사용이 간단.

상용 솔루션

모바일 페더레이션

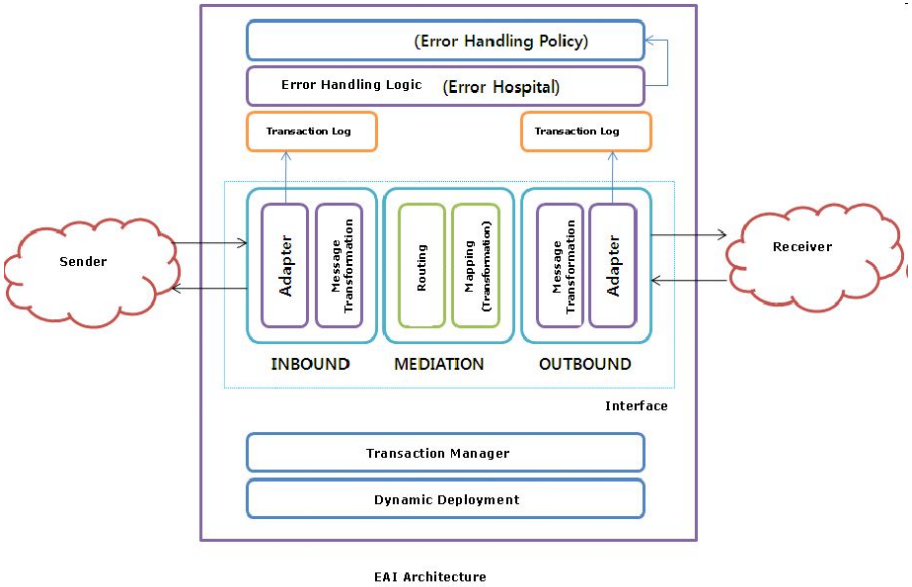
구글이 제공하는 사용자 인증 SDK

- Facebook, Twitter, Google 등 SNS 인증 제공 (카카오, 네이버 플러그인)
- IOS, Android,Web, Unity 등 지원
- JWT 토큰 발급 지원
- 빠르게 모바일, 웹 인증 구현에 유리함

출처 : https://firebase.google.com/docs/auth

EAI (엔터프라이즈 애플리케이션 통합)

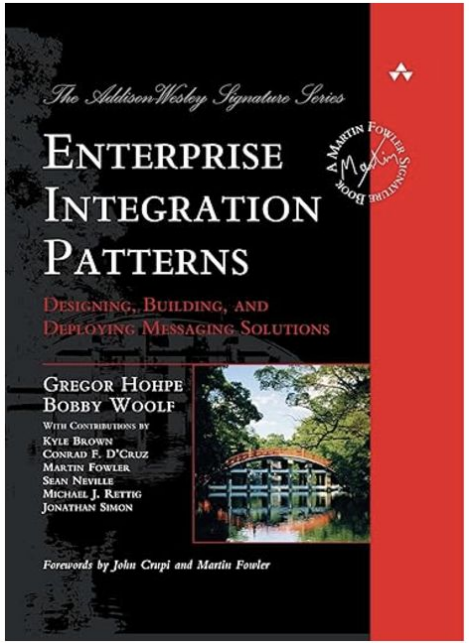
- **서로다른 시스템간의 연계가 필요한 경우**
  - 마이크로 서비스는 REST 기반 연계이기 때문에 별도의 EAI가 필요 없음
  - 메인프레임, CRM, ERP, TP Monitor등 레거시 시스템과의 연동에 많이 사용함. (다른 프로토콜)
  - Saas 기반의 서비스 연동은 EAI 보다는 Saas 기반의 Integration 솔루션을 사용 (Fivetran)
- **vs ETL**
  - ETL은 데이터베이스간 연결이라면 EAI는 애플리케이션간 API 연동
  - 조금더 실시간성에 가까움



EAI Architecture

EAI (엔터프라이즈 애플리케이션 통합)

- **구현**
  - 한국 대기업 기준 500~3000개의 통합 인터페이스를 구현함
  - 기술적인 통합보다는, 송신 부서와 수신 부서와의 인터페이스 협의 (주기, 데이터 포맷, 에러 처리 방식) 협의가 더 중요함
- **연동 패턴**
  - https://www.enterpriseintegrationpatterns.com/
  - 복잡하고 다양한 패턴이 있으나, 단순한 패턴이 좋음 (실업무에서는 API나 파일, DB 테이블 ETL 식이 더 많음)
- **솔루션**
  - OSS : Apache Camel 또는 REST API, 파일 방식
  - 상용 : Tibco, Webmethod



인터페이스 정의서 예시

프로젝트명	통합재무관리시스템
단 계	설계
작 성 자	AppsTech
작 성 일	2024.02.01

인터페이스 정의서 목록

필수기재항
선택기재
AppS팀 기재

일련번호	인터페이스 ID	인터페이스 업무 유형 ID	시스템 구분				업무명	인터페이스 명	데이터 순차성 여부	선행처리요건	후행처리요건	처리내역 설명	예외처리/특이 사항	운영 상태	데이터유형		발생 유형	전송방식	Direction	발생회수/기준	전송회수/발생회수 기준	I/F Data Size (Record건수)	Record 건수 기준	Record 당 size(Byte)	EAI 인터페이스 유형	
			송신(Send/Reply)		수신(Receive/Request)										Send/Request	Receive/Reply									메시지플로우코드	어댑팅유형코드
			기관명	시스템명	시스템명	기관명																				
1			통합재무구축	통합재무시스템	MIS	한국회사	물류관리	물류세부항목	N					개발중	I/F Table	I/F Table	Near-Real		단방향	일	2	1,000	회	1000		
2			한국회사	MIS	통합재무시스템	통합재무구축팀	물류관리	물류합산항목	N					개발중	I/F Table	I/F Table	Near-Real		단방향	일	3	1,000	회	1000		

CDN

- 정적 컨텐츠 (이미지, 스크립트, 바이너리, 비디오) 등을 사용자와 가까운 곳에 복사해놓고, 가까운 곳에서 서비스를 함으로써, 시스템의 레이턴시를 줄임
- Akami가 선두 (라임라이트, CDNetworks 등), 클라우드 CDN
- 비용이 매우 많이 나옴.
- 멀티 CDN으로 장애에 대비 필요. 비용에 따라서 트래픽을 유동적으로 정의할 수 있도록 설정. (사용자가 접속할때 마다 CDN Primary 주소를 내리는 방식)
- 콘텐츠 압축 (Jpeg >> WebP 30%)

글로벌 배포 시스템

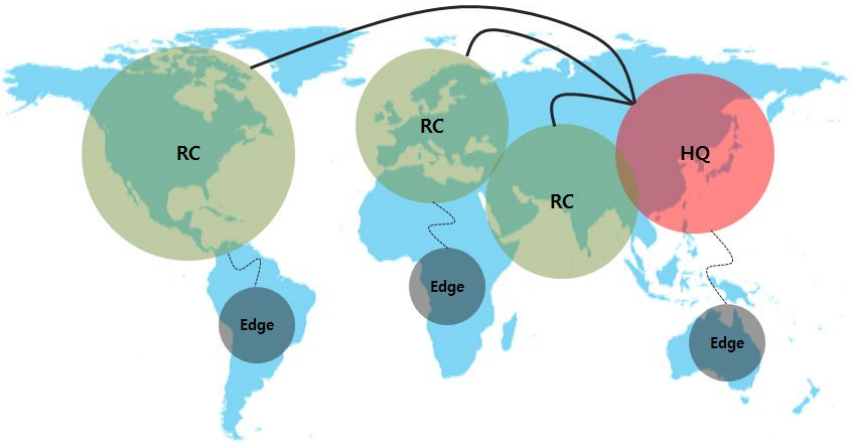
글로벌 스케일 시스템 디자인시 고려 사항

위치 선정

- 법적인 요건 (GDPR, Standard Contractual Clauses, SCCs)
- 지적 재산권, 라이센스 (넷플릭스 컨텐츠)
- 레이턴시,세제 혜택, 비용
- US, EU, Asia, China 보통 이렇게 4개 리전을 사용.
- **L10, I18N**

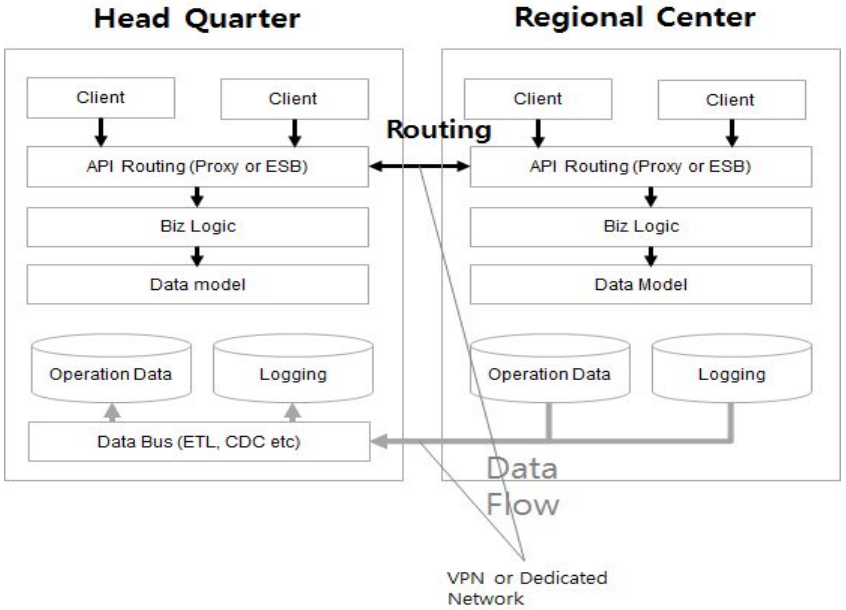
글로벌 스케일 시스템 아키텍처

- 구성 방식
  - **비대칭** : Master(HQ) / Regional 센터
    - Master : 모든 인프라 + 운영 + 분석 + 머신러닝 학습
    - Regional : 운영 (서빙 API), 데이터 분석 시스템 (국지적)
  - **대칭** : Master / Master 방식
- 서비스 Look up
  - 주로 가까운 데이터에 접속하도록 함. (글로벌 로드 밸런서)
  - 사용자 프로필 (국적)에 따라 리전 지정



글로벌 스케일 시스템 아키텍처

- 글로벌 시스템간 라우팅
  - 공용 서비스 이외에, 대부분의 경우 Region 간에 라우팅이 불필요함
  - 한국 사용자가 미국에 출장가서 네이버 접속해도 빠름
- 데이터 복제
  - 법적 규제에 따라 필요한 데이터만 복제
  - 실시간 데이터 복제가 필요한 경우는 적음
  - Google Spanner 등 클라우드 서비스의 경우 글로벌 복제를 지원할 수 있음



감사합니다.