1장 : 사용자 수에 따른 규모 확장성

난이도	⭐⭐⭐⭐
학습날짜	@2025년 12월 3일

챕터 개요

해당 챕터는 수백만 사용자를 지원하는 시스템을 설계하기 위해서 점진적으로 시스템 설계를 확장하는 방법에 대해 소개합니다.
이 과정에서 각 기술의 등장배경 및 개념부터 트레이드오프, 가용성 등이 소개됩니다.

개념

먼저, 아키텍처를 구성하며 기본적으로 알아야 하는 개념들에 대해 알아보겠습니다.

가용성

가용성은 비기능적 요구사항 (Non-Functional Requirement)으로 "**시스템이 필요한 시점에 정상적으로 작동할 수 있는 능력**"을 의미합니다.

- **고가용성 (High Availability)**를 지향하는 것이 대규모 시스템의 핵심입니다.

💡 **가용성**은 다양한 기준 수치가 존재합니다.

가용성 수치	연간 허용 중단 시간	시스템 특징 및 용도
99%	3일 15시간	비즈니스 핵심이 아닌 소규모 / 테스트 시스템
99.9%	8시간 45분	일반적인 웹 서비스의 최소 기준
99.99%	**52분 36초**	**대규모 상용 서비스 (이커머스, SNS)**
99.999%	5분 15초	미션 크리티컬 시스템 (금융, 의료, 통신)

고가용성을 지향하는 것이 바람직하며, 고가용성 (HA)가 지켜지는 서비스는 다음 성질을 지닙니다.

- 고장나도 바로 복구해서 서비스를 지속할 수 있다.
- 장애 상황에도 서비스를 지속할 수 있다.

수평적 규모 확장 (Scale Out) vs 수직적 규모 확장 (Scale Up)

- 수평적 규모 확장 : **Scale Out**이라고 부르며, 추가 서버를 두어 성능을 개선하는 행위를 의미한다.
- 수직적 규모 확장 : **Scale Up**이라고 부르며, 사용하고 있는 서버에 **고사양 자원** (더 좋은 CPU, 더 많은 RAM 등)을 추가하는 행위를 의미합니다.

💡 수직적 규모 확장은 대체적으로 **한계**가 있습니다.

- ~~한 대의 서버에 CPU나 메모리를~~ **~~무한대로 증설할 수 없다.~~**
  인프라 비용 : 예산 100만
  - `t3.infinite` : 10000만 * 1
    - CPU 속도 좋음
  - `t3.small` : 10만 * 10, 5개만 → 50만 + Auto Scaling 30만 = 80만, DB +2 ⇒ 100만
    - CPU 속도 느림
  - `t3.micro` : 1만

  CPU, 메모리 속도, 비용 ↔ 최적의 선택 / 맥북 air 1024 + pro 512
  - 관리 포인트
  - Scale Out을 했을 때 **JVM 불필요한 요소**들이 올라간다.
    - 비슷한 맥락: MSA (JVM n개) ↔ Monolithic (JVM 1)
  - Scale Out 장점
    - 병렬로 처리 가능하다.
      - ex) 스레드 풀이 n개 된다.

  Q: 현업에서 선택을 어떻게 하나?
  - AI) “Scale Up이 비싸다, 효율이 안나온다”

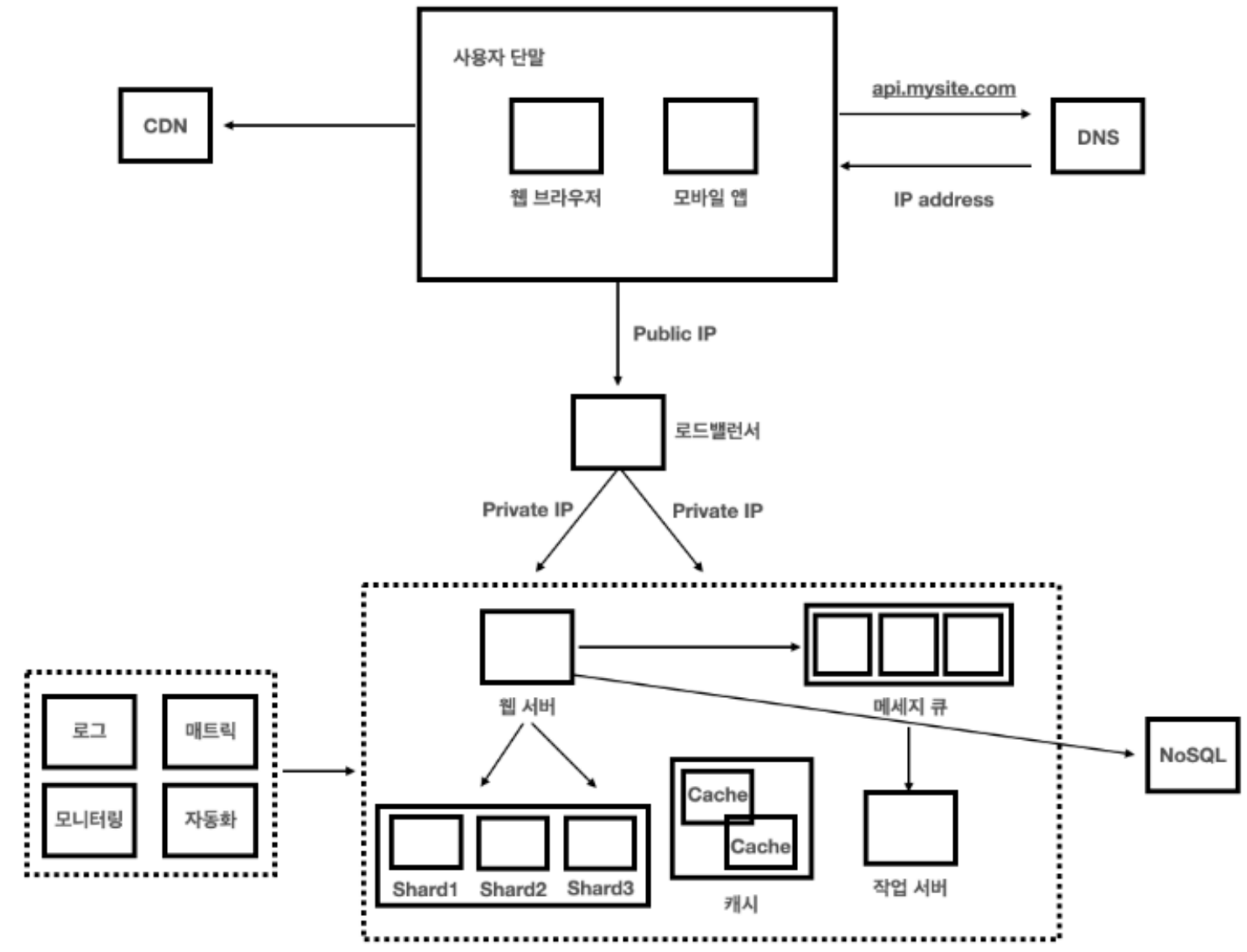
  Q: Scale Up에 대한 장점을 찾아보면 끝.

  측정 단위
  - RPS : 초당 처리하는 요청(Request) 개수
    - TPS : 초당 트랜잭션(Transaction) 처리량
      - QPS : 초당 쿼리(Query) 처리량

- 장애에 대한 자동복구 방안, 다중화 방안을 제시하지 않으므로, **서버에 장애가 발생하면 서비스가 완전히 중단**된다. 이러한 중단 현상을 **SPOF** (Single Point Of Failure)라 부른다.

아키텍처 구성

수백만 사용자를 고려하여 구성된 **최종 아키텍처를 토대로 해당 챕터의 내용을 정리**해보겠습니다.



최종 아키텍처

요청이 처리되는 과정

1. 사용자 단말 요청 발생
2. DNS 조회
3. CDN을 통한 정적 콘텐츠 처리
4. 로드밸런서의 Public IP로 요청 전송
5. Private IP를 통한 웹 서버로 전달
6. 캐시 확인 및 Cache Hit시 데이터 반환
7. 데이터베이스 조회 및 처리 / 비동기 작업 처리
8. 최종 응답

아래 모든 과정에 대해 필요에 따라 로그 수집, 메트릭 수집, 모니터링 지표 수집, 자동화가 실행된다.

요청 처리 순서에 따라 사용되는 구성 요소들의 특징과 성질을 알아보겠습니다.

DNS

데이터 센터

서비스의 규모가 커짐에 따라, 글로벌 서비스로 발디딤하는 경우 전 세계 어디서든 쾌적하게 사용할 수 있기 위해서는 서버를 하나의 지역에만 두는 것이 아닌, **지리적 라우팅** (GeoDNS Routing 또는 geo-routing)을 통해 가장 가까운 **"데이터 센터"**로 안내되도록 할 수 있습니다.

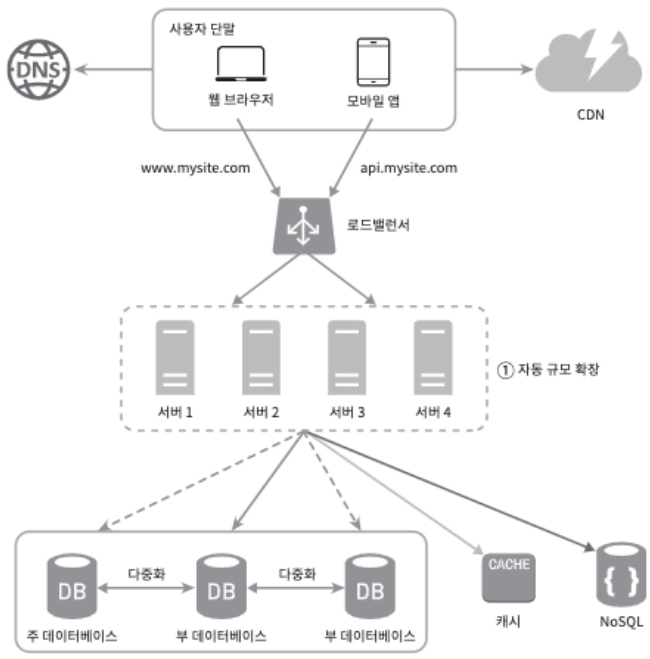
이때, 기존의 DNS와는 달리, 사용자를 가장 가까운 데이터센터로 안내하기 위해 사용되는 서비스가 **geoDNS 서비스**이다.

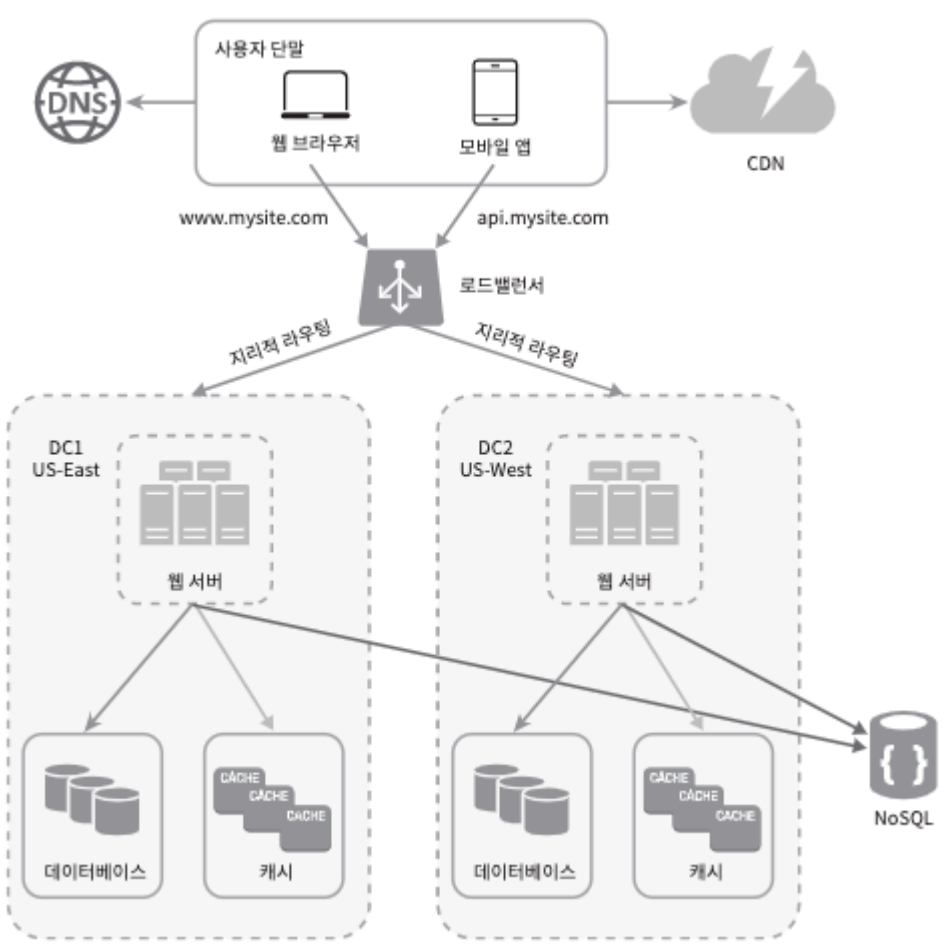
> **삼성 SDS**의 데이터 센터에 대한 설명
>
> 기업이나 기관이 대량 데이터 저장, 처리, 관리, 배포를 위해 **대규모 IT 인프라(서버, 스토리지, 네트워크 장비 등)를 집약해 운영 및 관리**하는 핵심 시설

💡 클라우드와 데이터센터의 차이는 무엇인가요?

- 클라우드는 데이터센터 위에서 가상화 기술과 자동화 플랫폼을 통해 서비스를 제공하는 형태이다.
  - 데이터센터를 `DB`라고 생각할 수 있으며, 클라우드는 `@Service` 라고 생각하면 된다.
  - AWS, Azure, Google Cloud 등 클라우드 사업자들은 지역별로 분산된 하이퍼스케일 데이터센터를 운영하며, 이를 통해 고가용성, 확장성을 보장하는 서비스를 제공한다.
- 기업은 자체 데이터센터를 운영하거나, 클라우드 사업자의 데이터센터를 활용할 수 있다.
  - 하이브리드 전략으로, 자체 데이터센터와 퍼블릭 클라우드를 결합하여 유연성을 높이는 사례도 많다.

왼쪽과 같은 형태의 아키텍처 구조를 오른쪽의 데이터 센터를 활용한 지리적 라우팅 방식으로 구현하는 것이다.





CDN

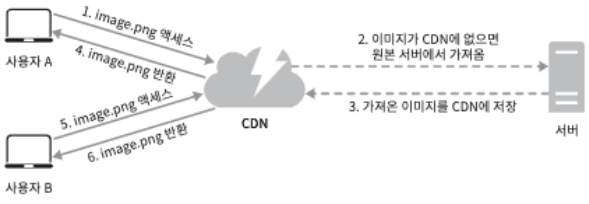
CDN은 정적 컨텐츠의 전송 속도를 개선하기 위해 사용되며, **지리적으로 분산**된 서버의 네트워크이다.

이미지, 비디오, CSS, Javascript 파일 등을 캐싱할 수 있다.

각 캐싱 값은 **TTL (Time to Live)**를 가지며 시간이 끝날 때까지 캐시됩니다.

- TTL은 데이터의 특성을 고려하여 적절한 값으로 설정되어야 합니다.
  - TTL이 너무 크면 데이터의 신선도가 떨어진다
  - TTL이 너무 작으면 원본 서버에 빈번히 접속하게 되어서 성능 (응답 속도, DB Connection Pool 관리 등)에 좋지 않다.

CDN은 다음과 같은 방법으로 **데이터를 캐싱**합니다.



사용자가 CDN 서버에 처음으로 콘텐츠를 요청하는 상황

1. **클라이언트 요청:** 사용자가 CDN 서버(가장 가까운 CDN 서버)에 콘텐츠를 요청
2. **CDN 확인:** CDN 에지 서버는 캐시를 확인하지만, 해당 콘텐츠가 없으므로 **캐시 미스(Cache Miss)**가 발생
3. **원본 서버 요청:** CDN 에지 서버는 요청을 **원본 서버**(사용자님의 웹 서버)로 전달
4. **원본 서버 응답:** 원본 서버는 요청된 데이터(예: JSON 응답, 이미지 파일)와 함께 다음과 같은 **캐싱 지시 헤더**를 CDN에게 전달

   ```
   HTTP/1.1 200 OK
   Cache-Control: public, max-age=3600  ← 핵심!
   Content-Type: application/json
   ```

5. **캐싱 및 전달 (동시 발생):**
   - CDN은 이 응답의 `Cache-Control` 헤더를 확인합니다. `max-age=3600`이 있다면, **이 데이터를 1시간(3600초) 동안 저장하겠다**고 결정
   - CDN은 해당 데이터를 로컬 캐시에 **저장(Caching)**하는 동시에, 클라이언트에게 응답을 **전달**

💡 CDN에 캐싱하는 방법

HTTP 응답 헤더를 통해 CDN에 캐싱 여부를 저장합니다.

1. `Cache-Control` 헤더:
   - `max-age=3600`: 3600초(1시간) 동안 캐시를 유효하게 유지합니다.
   - `no-store`: 절대 캐시하지 마세요. (민감한 데이터에 필수)
   - `public` / `private`: CDN(공용 캐시)이 캐시할 수 있는지 결정합니다. 개인 정보가 포함된 응답에는 `private`을 사용합니다.
2. `Expires` 헤더:
   - 캐시가 만료되는 정확한 날짜와 시간을 지정합니다.

💡 동적 컨텐츠 캐싱이란? API

요청 경로 (request path), 질의 문자열 (query string), 쿠키 (cookie), 요청 헤더 (request header) 등의 정보에 기반하여 HTML 페이지, 요청에 대한 응답을 캐시하는 방법

- 사용자별, 요청별로 달라질 수 있는 컨텐츠
  - API 응답, 개인화된 페이지, 검색 결과 등
- 예시: `/api/user/profile`, `/search?q=keyword`
  - image javascript HTML **GET** `/api/v1/image-jiho-profile`

💡 CDN에 캐싱된 값을 무효화하는 방법

Revert 전략을 매우 중요합니다.
왜냐하면 CDN에 올바르지 못한 값이 저장된다면 해당 CDN에 접속하는 모든 사용자는 부적절한 데이터를 전송받을 수 있기 때문입니다.

- 사용자의 인증세션이 CDN에 실수로 캐싱되어 버리면 개인정보 유출 및 서비스 신뢰도 하락으로 이어진다.

1. Purge (완전 삭제) - 가장 빠름

```
# Cloudflare 예시
curl -X POST "https://api.cloudflare.com/client/v4/zones/{zone_id}/purge_cache" \
  -H "Authorization: Bearer {api_token}" \
  -H "Content-Type: application/json" \
  --data '{
    "files": [
      "https://example.com/api/user/dashboard"
    ]
  }'

# AWS CloudFront 예시
aws cloudfront create-invalidation \
  --distribution-id EDFDVBD6EXAMPLE \
  --paths "/api/user/*" "/api/session/*"
```

- **즉각적**: 보통 수초~수분 내 완료
- **정확함**: 특정 URL만 타겟팅
- **추적 가능**: Invalidation ID로 진행상황 모니터링

2. Purge All (전체 삭제) - 긴급 상황

```
# 전체 캐시 삭제
curl -X POST "https://api.cloudflare.com/client/v4/zones/{zone_id}/purge_cache" \
  -H "Authorization: Bearer {api_token}" \
  -H "Content-Type: application/json" \
  --data '{"purge_everything":true}'
```

언제 사용하는가:

- 세션 정보가 광범위하게 캐싱된 경우
- 영향 범위를 특정할 수 없는 경우
- 즉각적인 조치가 필요한 보안 이슈

3. Cache Tag 기반 무효화 - 고급 기법

```
// 컨텐츠 태깅
res.set('Cache-Tag', 'user-data, user-123, sensitive');

// 태그별 무효화
purgeByTag('sensitive'); // 해당 태그가 붙은 모든 캐시 삭제
```

캐시 메모리가 꽉찬 경우

먼저 CDN의 메모리 구조를 이해해야 합니다:

```
CDN Edge Server (PoP - Point of Presence)
├─ Hot Cache (메모리): 수 GB - 수십 GB
│   └─ 가장 자주 접근되는 컨텐츠
├─ Warm Cache (SSD): 수백 GB - 수 TB
│   └─ 최근 접근된 컨텐츠
└─ Cold Storage (Origin): 무제한
    └─ 원본 데이터
```

대부분의 CDN에서는 **LRU 알고리즘**을 사용합니다.

1. **Temporal Locality(시간적 지역성)**: 최근 요청된 컨텐츠는 다시 요청될 가능성 높음
2. **구현이 단순**: 링크드 리스트 + 해시맵으로 O(1) 구현 가능
3. **예측 가능**: 동작이 직관적이고 디버깅 용이

특수한 경우 LFU (Least Frequently Used)를 사용하기도 합니다.

- 롱테일 컨텐츠가 많은 경우 (예: 동영상 플랫폼)
- 인기 컨텐츠와 비인기 컨텐츠가 명확히 구분되는 경우

단점

- Cold Start 문제 : 새로운 컨텐츠가 즉시 인기를 얻어도 제거될 수 있음
- **복잡도**: LRU보다 구현이 복잡함

Size-Aware LRU 등 다양한 방식이 존재합니다.

[실제 CDN 제공업체]

```
Cloudflare:
- LRU 기반
- 자동 Tiered Caching (메모리 → SSD)
- 객체당 최대 512MB

AWS CloudFront:
- LRU 변형 (정확한 알고리즘 비공개)
- Origin Shield로 추가 캐싱 계층
- 객체당 최대 30GB

Fastly:
- LRU 기반
- Instant Purge (실시간 무효화)
- 커스터마이징 가능한 eviction 정책
```

CDN과 Redis의 차이

의문점 : CDN을 사용할 바에는 Redis를 사용하면 되지 않나?

데이터베이스

[RDBMS, NoSQL]

RDBMS는 관계형 데이터베이스 관리 시스템이라고 하며, 자료를 테이블과 열, 칼럼으로 표현하며 SQL을 사용하여 여러 테이블에 있는 데이터를 그 관계에 따라 조인하여 합칠 수 있습니다.

- MySQL, 오라클 데이터베이스, PostgreSQL 등이 있습니다.

NoSQL은 비관계형 데이터베이스로 키-값(key-value) 저장소, 그래프(graph) 저장소, 칼럼(column) 저장소, 문서(document) 저장소가 있습니다.

비관계형 데이터베이스는 일반적으로 조인 연산을 지원하지 않습니다.

- Amazon DynamoDB, CouchDB, Neo4j, Cassandra, HBase 등이 있습니다.

NoSQL은 RDBMS에 비해

- 낮은 응답 지연시간이 요구된다.
- 데이터가 비정형 데이터이다.
    - 데이터를 직렬화하거나, 역직렬화 할 수만 있으면 된다.

[데이터베이스 다중화 (Master-Slave 관계)]

데이터베이스 다중화를 통해 부하를 분산할 수 있다.

많이 사용되는 방식이 Master-Slave 방식이며 다음과 같이 Master와 Slave를 두어 역할을 분리하여 사용한다.

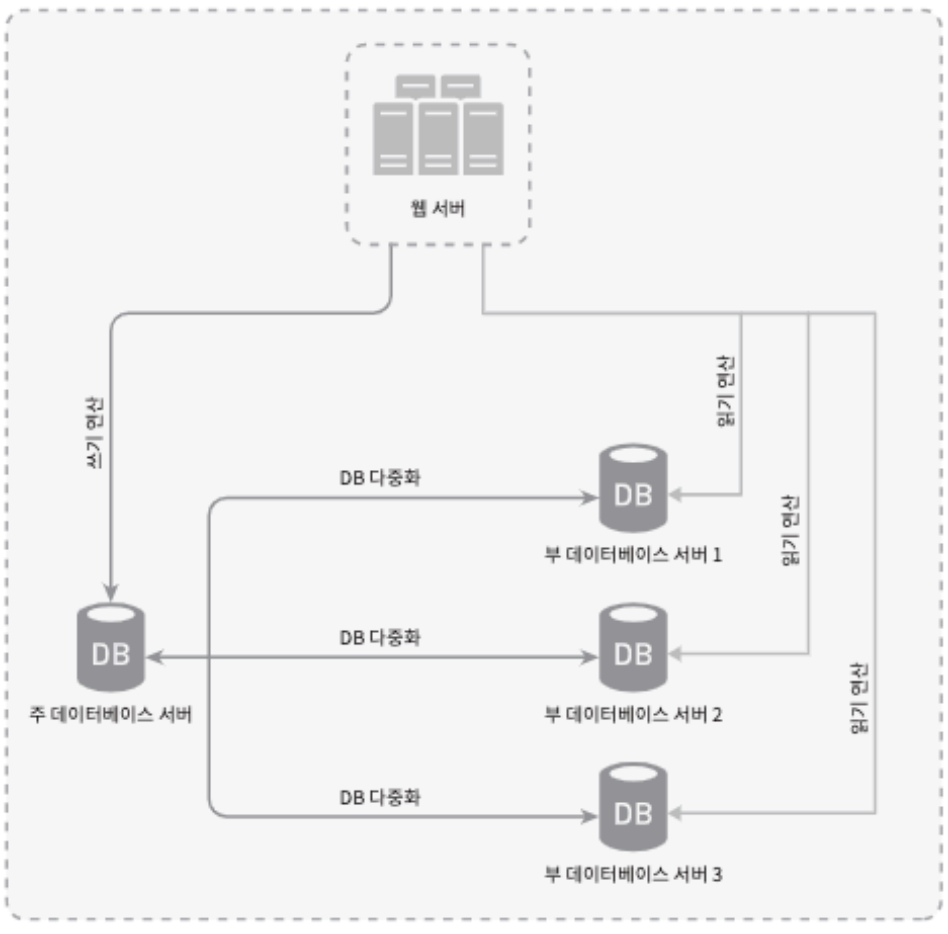
- 주 (Master) : 쓰기 연산 실행
    - insert, delete, update 등
- 부 (Slave) : 주 DB로부터 사본을 전달받으며, 읽기 연산만을 지원한다.

이렇게 구성하는 이유는 다음과 같다

1. **성능**

   읽기 연산이 모두 부 데이터베이스 서버로 분산되어, 병렬로 처리될 수 있는 질의의 수가 늘어나므로, 성능이 개선된다
2. **안정성, 가용성**

   다중화된 서버로 인해 데이터를 보존시킬 수 있으며, 하나의 DB에 문제가 발생해도 다른 서버로 서비스를 유지시킬 수 있다.



[샤딩 (Sharding)]

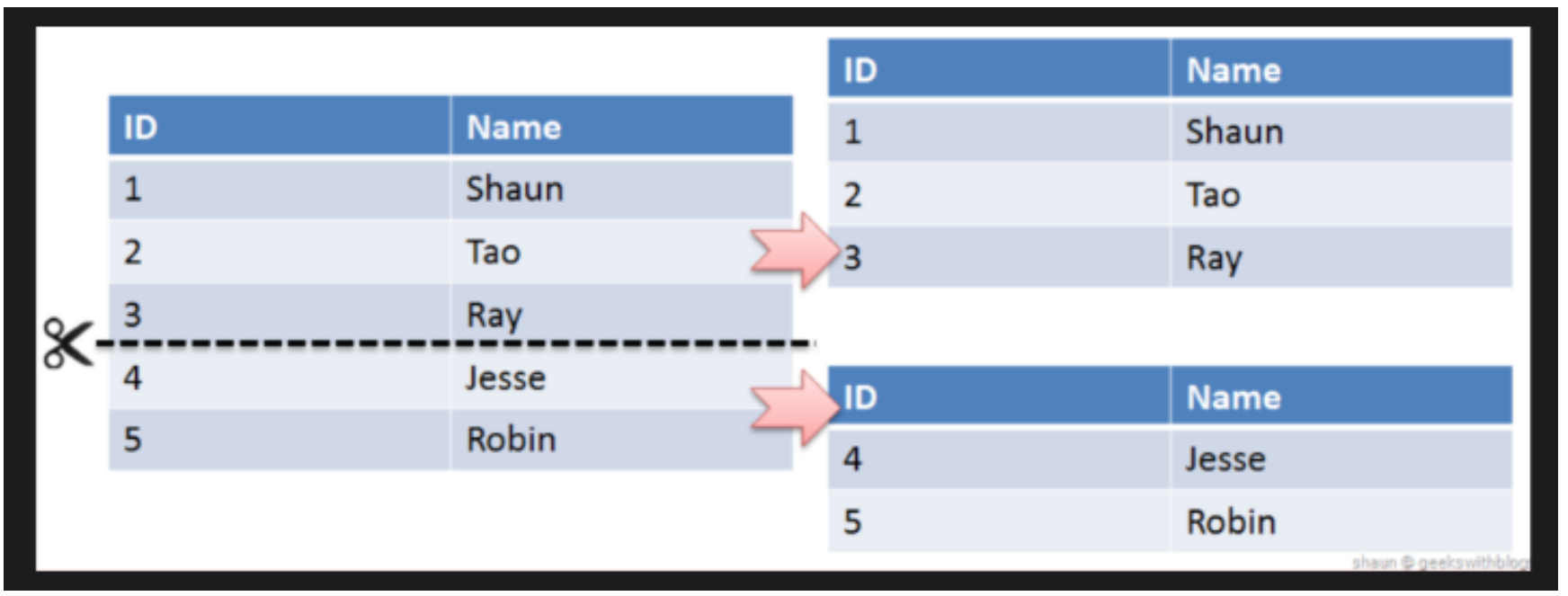
데이터베이스의 수평적 확장을 할 때 사용하는 방법이며, 대규모 데이터베이스를 작은 단위인 샤드 (shard)로 분할하는 기술을 일컫는다.

모든 샤드는 같은 스키마를 사용하지만, **샤드에 보관되는 데이터 사이에는 중복된 데이터가 없다**는 특징을 지니고 있다.

샤딩 전략을 구현할 때는 **샤딩 키(Sharding Key)**를 어떻게 정하는지가 중요합니다.

> 샤딩 키는 파티셔닝 키라고도 불린다.

- 데이터를 **고르게 분할** 할 수 있도록 하는게 가장 중요하다.
- 유명인사의 경우, 핫스팟 키 (hotspot key) 문제라고 하여 특정 샤드에 질의가 집중되어 서버에 과부하가 걸리는 문제가 발생할 수 있다.
- 하나의 DB를 여러 개의 샤드로 쪼개고 나면, 여러 샤드에 걸친 데이터를 조인하기가 힘들어진다.



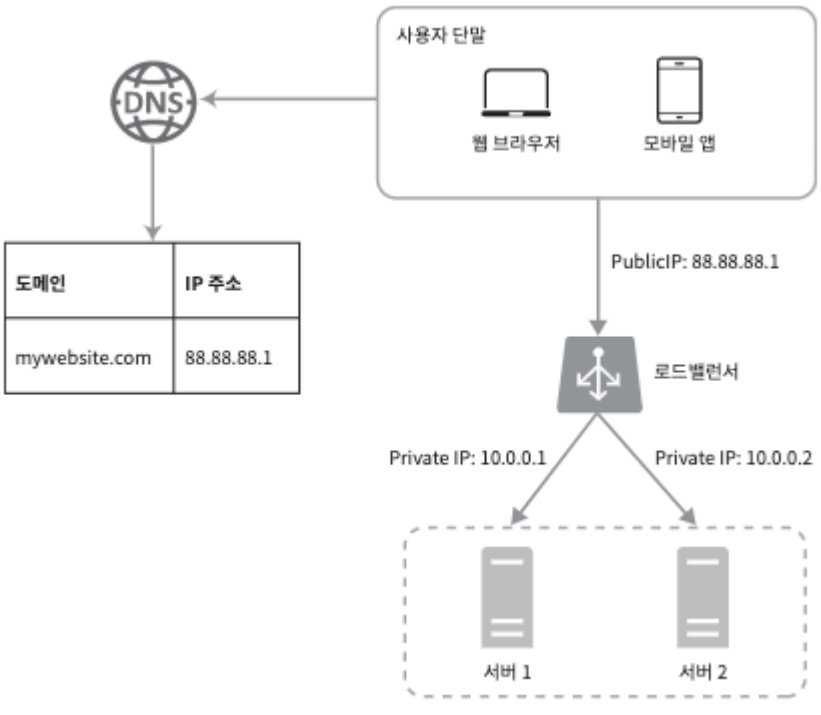
수평 단편화의 예시 (Horizontal Partitioning)

💡 TODO : 샤딩을 하는 방법 → 수평적 / 수직적

로드밸런서

로드밸런서는 부하 분산 집합 (load balancing set)에 속한 웹 서버들에게 **트래픽 부하를 고르게 분산하는 역할**을 수행합니다.

- 로드밸런싱 알고리즘을 통해 부하 분산
  - Round Robin, Weighted Round Robin, Least Connections, Least Response Time (최소 응답 시간), IP Hash (세션 어피니티)
- Health Check를 통해 서버 상태 확인
- 세션 지속성을 통해 동일한 클라이언트의 요청을 동일 서버로 유지



로드밸런서는 **공개 IP (공인 IP)**를 지니고 있으며, 이를 통해 외부와의 통신을 진행하며 로드밸런서와 서버간의 통신은 내부 IP (사설 IP)를 통해 진행됩니다.

- 사설 IP로는 외부로 통신할 수 없다.
    - 서버의 IP를 감추는 효과를 제공하기도 한다.

로드밸런서의 종류로는

1. **L4 로드밸런서 (NLB/TCP LB):**
    - **사용 시점:** HTTP/S 헤더를 분석할 필요 없이 **고속의 패킷 전달**이 중요할 때.
    - **예시:** 게임 서버, 실시간 스트리밍, 대규모 데이터베이스 연결(Connection Pool) 등 순수한 TCP/UDP 트래픽을 처리할 때.
    - **장점:** L7보다 처리 속도가 매우 빠르고 지연 시간이 짧습니다.
2. **L7 로드밸런서 (ALB/HTTP LB):**
    - **사용 시점:** HTTP/S 요청의 내용을 기반으로 **지능적인 라우팅**이 필요할 때.
    - **예시:**
        - **경로 기반 라우팅:** `/api` 요청은 API 서버로, `/images` 요청은 이미지 서버로 분리.
        - **호스트 기반 라우팅:** `api.example.com` 요청은 A 서버 그룹으로, `www.example.com` 요청은 B 서버 그룹으로 분리.
        - **SSL/TLS 종료:** 클라이언트와의 암호화 통신을 LB에서 끝내고(Decryption), 백엔드 서버는 부담을 덜고 평문(HTTP)으로 통신하게 할 때.

▼ 자세히 보기

시나리오 1: L7 로드밸런서 (Application Load Balancer, ALB)

L7 로드밸런서는 OSI Layer7의 어플리케이션 계층에서 동작하는 로드밸런서입니다.

→ 부하 혹은 트래픽을 균형있게 나눠주는 역할을 수행합니다.

네트워크 구성:

```
 인터넷
  ↓
[공인 IP: 1.2.3.4]
  ↓
┌──────────────────────┐
│ ALB (L7 LB)     │
│ - SSL 종료       │
│ - HTTP 라우팅     │
└──────────────────────┘
```

```
   ↓ ↓ ↓
[사설 IP: 10.0.1.1] [10.0.1.2] [10.0.1.3]
   ↓ ↓ ↓
 [서버1] [서버2] [서버3]
```

패킷 흐름 (ALB의 Proxy 동작):

구분	클라이언트 → ALB	ALB → 백엔드 서버
출발지 IP:Port	203.0.113.50:54321 (클라이언트 IP 유지)	10.0.0.10:38291 (**ALB 사설 IP**로 변경)
목적지 IP:Port	1.2.3.4:443 (ALB 공인 IP 유지)	10.0.1.1:8080 (**서버 사설 IP**로 변경)
원본 IP 확인	-	`X-Forwarded-For` 헤더를 통해 원본 클라이언트 IP 전달

ALB는 NAT 역할을 하는가?

답: 부분적으로 YES, 하지만 엄밀히는 **Proxy**

이유:

1. **연결 종료 및 재생성**: ALB는 클라이언트 연결을 완전히 종료하고 백엔드로 새로운 연결을 생성합니다.
2. **출발지 IP 변환**: 트래픽이 ALB의 사설 IP로 변환되므로 **SNAT (Source NAT) 역할**을 수행합니다.
3. **NAT와의 차이: NAT는 L3/L4 계층에서만 IP를 변환하지만, ALB는 L7(HTTP)에서 메시지 내용을 이해하고 조작(헤더 추가)하므로 프록시(Proxy)라고 부르는 것이 더 정확합니다.**

시나리오 2: L4 로드밸런서 (Network Load Balancer, NLB)

모드 1: DSR (Direct Server Return) 모드

패킷 흐름:

- **요청 시:** NLB는 목적지 IP만 서버의 사설 IP로 변경하고 **출발지 IP(클라이언트 IP)는 그대로 유지**한 채 서버로 전달합니다. (DNAT만 수행)
- **응답 시:** 서버가 **NLB를 거치지 않고** 클라이언트에게 직접 응답합니다.

NLB (DSR 모드)는 NAT 역할을 하는가?

답: NO, DNAT(Destination NAT)만 수행

이유:

- **출발지 IP 유지**: 클라이언트 IP가 그대로 서버까지 전달되어 서버 로그에 클라이언트 원본 IP가 남습니다.
- **응답 직접 반환**: 서버가 공인 IP를 루프백 인터페이스에 바인딩하여 NLB의 도움 없이 직접 응답합니다.

모드 2: Proxy 모드 (NAT 수행)

패킷 흐름:

구분	클라이언트 → NLB	NLB → 서버1
출발지 IP:Port	203.0.113.50:54321	10.0.0.10:38291 (**NLB 사설 IP**로 변경) **(SNAT)**
목적지 IP:Port	1.2.3.4:80	10.0.1.1:80 (**서버 사설 IP**로 변경) **(DNAT)**
응답 흐름	-	서버 → NLB → 클라이언트 (양방향 NLB 통과)

NLB (Proxy 모드)는 NAT 역할을 하는가?

답: YES, 완전한 NAT

이유:

- **SNAT (Source NAT)**: 클라이언트 IP를 NLB 사설 IP로 변환합니다.
- **DNAT (Destination NAT)**: 공인 IP를 서버 사설 IP로 변환합니다.
- **상태 추적**: 연결 상태를 추적하여 요청과 응답 모두를 정확하게 역변환(Reverse NAT)합니다.

시나리오 3: NAT Gateway + 로드밸런서 분리 (AWS 일반 구성)

실제 클라우드 프로덕션 환경에서는 LB와 NAT 역할을 분리합니다.

구성 개요:

- **ALB/NLB (Public Subnet): 인바운드(Inbound)** 트래픽 처리 (클라이언트 → 서버)
- **NAT Gateway (Public Subnet): 아웃바운드(Outbound)** 트래픽 처리 (서버 → 외부 인터넷)
- **서버 (Private Subnet):** 사설 IP만 보유

역할	인바운드 (외부 → 서버)	아웃바운드 (서버 → 외부)
담당 장치	**ALB 또는 NLB**	**NAT Gateway**
수행 기능	Proxy 또는 DNAT/SNAT (로드밸런싱)	**완벽한 SNAT** (사설 IP → 공인 IP 변환)

트래픽 플로우 (아웃바운드 예시):

1. **서버1** (10.0.1.1)이 외부 API 호출을 시도합니다.
2. 라우팅 테이블에 의해 트래픽이 **NAT Gateway** (10.0.0.5)로 전송됩니다.
3. NAT Gateway가 출발지 IP를 **NAT GW의 공인 IP** (예: 54.230.1.1)로 변환합니다. **(SNAT 수행)**
4. 변환된 트래픽이 Internet Gateway를 통해 외부 서비스로 나갑니다.

로드밸런서와 리버스 프록시의 관계

의문점 : ALB가 로드밸런서이면 리버스 프록시는 아닌가? 로드밸런서이면서 리버스 프록시인것 같은데 둘을 같이 사용하거나 어느 하나를 선택해서 사용하는 경우가 있나?

먼저, 로드밸런서와 리버스 프록시에 대해 확실히 정리해야 합니다.

- 로드밸런서 : 트래픽을 여러 **서버에 분산시키는 목적**을 지닌 시스템
- 리버스 프록시 : 클라이언트 요청을 받아서 백엔드 서버로 **전달하는 중개자**

두 개념을 자세히 살펴보면, A or B의 대척점을 지닌 개념이 아닌 비슷하지만 다른 역할을 수행하는 것을 볼 수 있습니다.

ALB는 **로드밸런서이면서 동시에 리버스 프록시**로 사용할 수 있습니다.
왜냐하면, 리버스 프록시 구현 방식을 사용하여 로드밸런싱 목적을 구현하기 때문입니다.

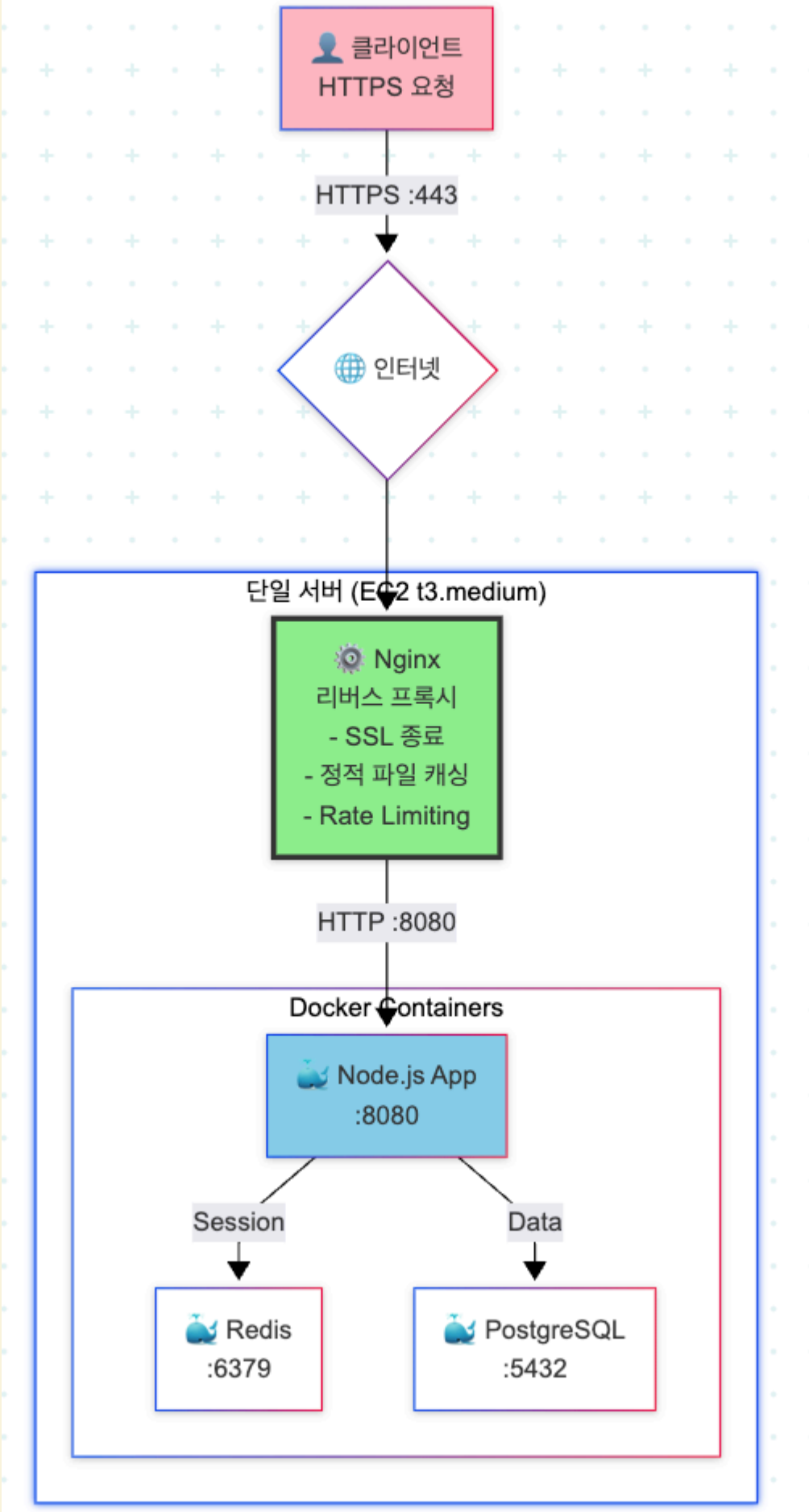
로드밸런서와 리버스 프록시라는 개념에 혼동을 가지지 말고, **부하를 분산하는 여러가지 시나리오**를 통해 아키텍처를 구성하는 방법에 대해 알아보자.

1. Nginx를 단일 백엔드 서버 앞에 두는 경우: Nginx는 리버스 프록시 역할만 수행

자주 사용하는 방식은 아니지만 장점이 존재하기는 합니다. 서버가 한대 뿐인 작은 서비스를 운영하는 경우 유용하게 사용할 수 있습니다.
Nginx 역할 : ✅ 리버스 프록시, ❌ 로드밸런서

- 벡엔드 서버를 직접 노출시키지 않을 수 있습니다.
- 3-Way HandShake를 담당하는 HTTP (TLS) 처리를 NGINX에 위임하고, 애플리케이션 서버는 HTTPS에 비해 비교적 가벼운 HTTP만 처리하도록 할 수 있습니다.
- 정적 파일을 Nginx에서 서빙하도록 하여 트래픽을 분산할 수 있습니다.

```
http {
    upstream backend {
        server 10.0.1.1:8080;  # 단 하나의 서버만!
    }

    server {
        listen 80;

        # SSL 종료
        listen 443 ssl;
        ssl_certificate /etc/nginx/cert.pem;
        ssl_certificate_key /etc/nginx/key.pem;

        location / {
            # 정적 파일 캐싱
            proxy_cache my_cache;

            # 헤더 추가
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            # 요청 전달
            proxy_pass http://backend;
        }
    }
}
```

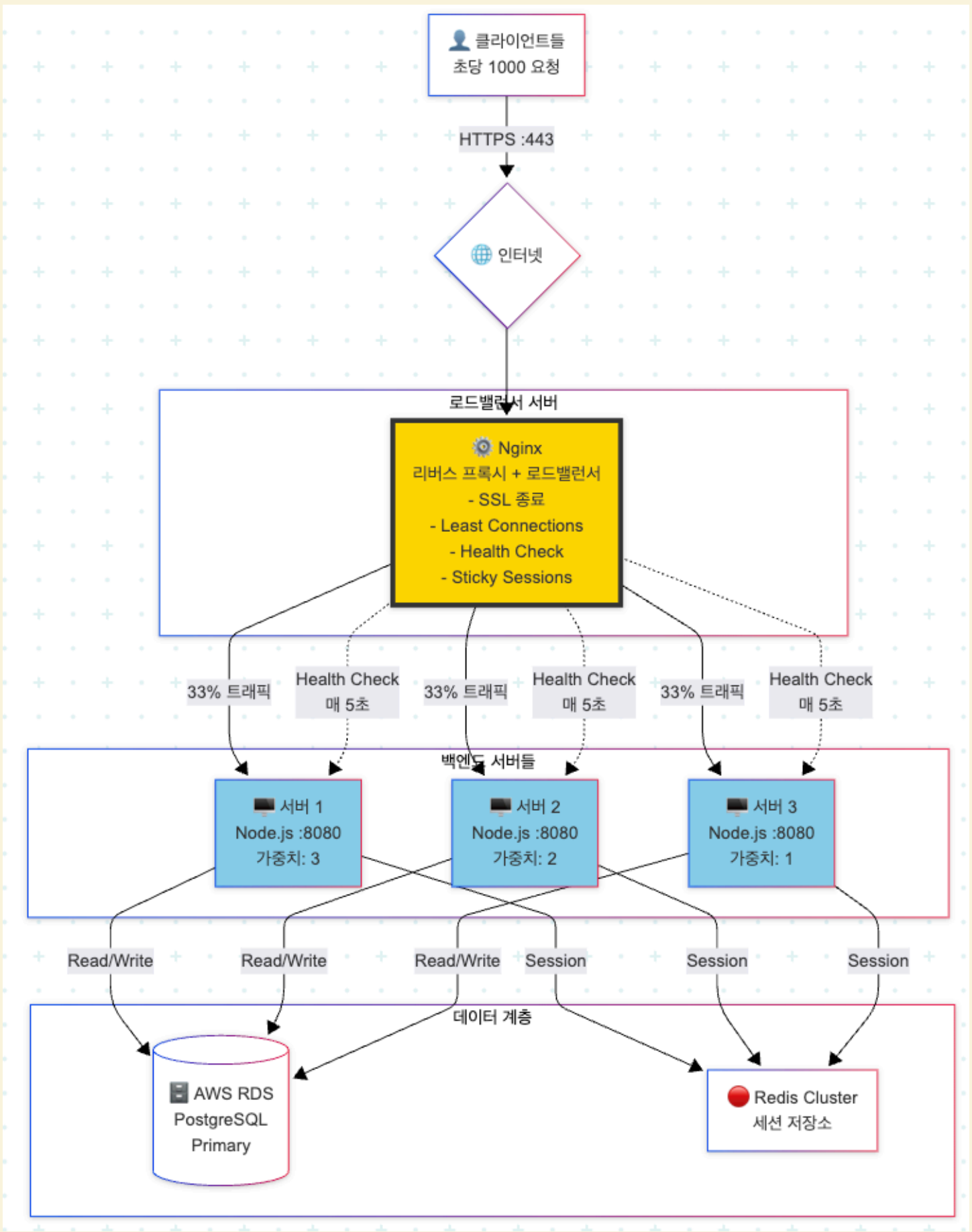


2. Nginx를 여러 백엔드 서버 앞에 두는 경우: Nginx는 로드밸런서 + 리버스 프록시 역할 수행

가장 흔하게 사용하는 방법으로, 여러 대의 서버를 운영하는 경우 사용됩니다.
Nginx 역할 : ✅ 리버스 프록시, ✅ 로드밸런서

```
http {
    upstream backend {
        # 여러 서버로 부하 분산
        server 10.0.1.1:8080 weight=3;
        server 10.0.1.2:8080 weight=2;
        server 10.0.1.3:8080 weight=1;

        # 로드밸런싱 알고리즘
        least_conn;

        # Health check
        server 10.0.1.1:8080 max_fails=3 fail_timeout=30s;
    }
```

```
    server {
        listen 80;

        location / {
            # 리버스 프록시 설정
            proxy_pass http://backend;

            # 로드밸런서 특화 설정
            proxy_next_upstream error timeout http_500;  # 장애 시 다음 서버로
        }
    }
}
```



3. L4 로드밸런서와 Nginx를 동시에 사용 하는 경우 : 대규모 엔터프라이즈

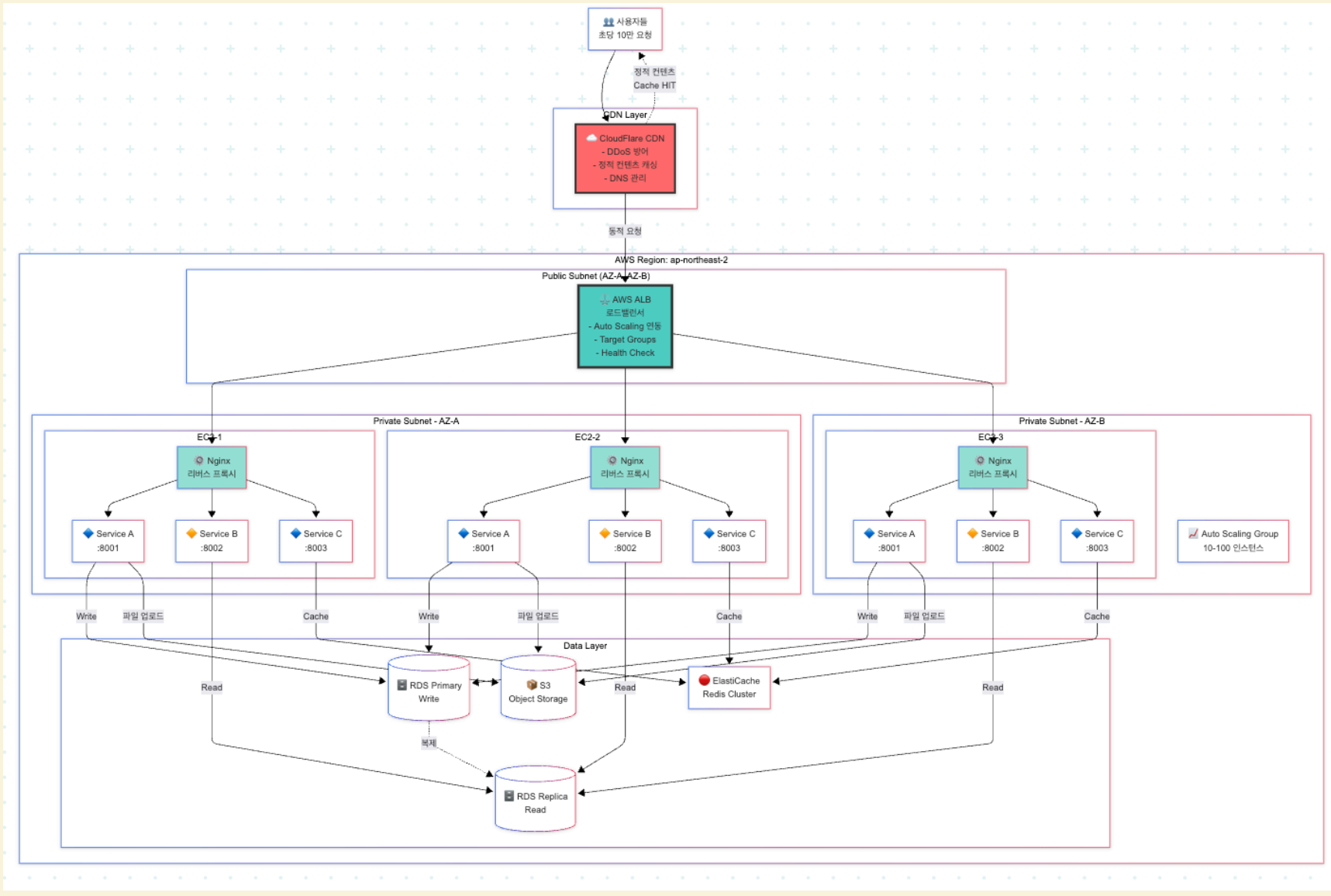
서비스 규모가 커짐에 따라, AWS ALB와 같이 로드밸런서 역할만을 수행하는 서비스와 각 서버에 Nginx를 통해 ㅐ 리버스 프록시 역할을 수행하도록 구성합니다.

AWS ALB의 역할

- **오토 스케일링**
- Target Group 관리
- 헬스 체크

각 서버의 Nginx 역할

- 리버스 프록시
- 마이크로서비스 라우팅
- 서비스별 인증/인



캐시

캐시 계층은 데이터를 잠시 보관해두는 곳으로 데이터베이스보다 훨씬 빠른 응답 속도를 지니고 있습니다.

다양한 캐시 전략이 존재하는데, 캐시할 데이터의 종류 / 크기 / 액세스 팬턴 등에 맞는 적절한 캐시 전략을 선택하면 됩니다.

캐시를 사용할 때는 다음 내용들을 고려해보아야 합니다.

- 어떤 상황에 쓰는 것이 바람직한가?
- 어떤 데이터를 캐시할까?
- 어떻게 만료시킬까?
- 일관성을 어떻게 유지시킬까? (데이터의 정합성)
- 캐시의 장애에는 어떻게 대응할까?
- 캐시 메모리의 크기는 어떻게 측정할까?
- 데이터 방출 정책은 어떻게 설정할까?

메세지 큐

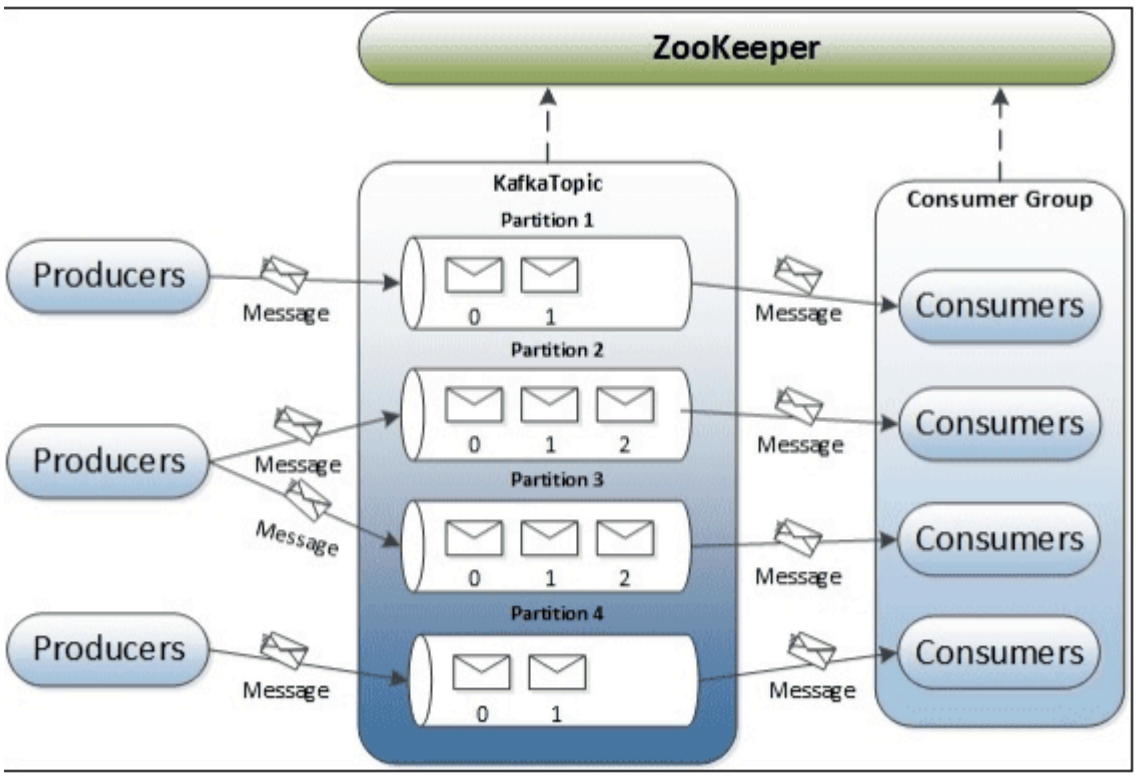
메세지 큐는 메세지 혹은 데이터의 무손실을 보장하는, 비동기 통신을 지원하는 서비스입니다.

메세지의 버퍼 역할을 수행하며 비동기적으로 메세지를 전송합니다.

메세지 큐를 사용하면, **서비스 또는 서버 간 결합이 느슨**해져서, 규모 확장성이 보장되어야 하는 안정적 애플리케이션을 구성하기에 좋습니다.

메세지 큐 아키텍처

- 생산자 또는 발행자 (Producer / Consumer) 형태로 메세지 큐를 활용한다.



로그, 메트릭, 자동화

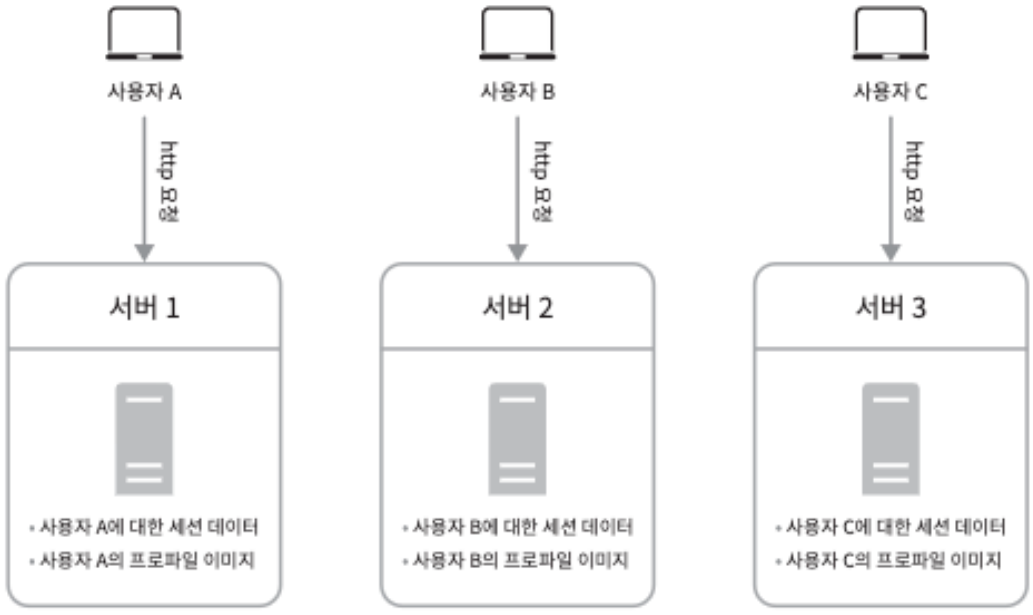
서비스와 관련된 지표를 수집하는 방법입니다.

- 로그 : 시스템 오류나 문제, 에러들을 뜻합니다.
- 메트릭 : Host (CPU, 메모리, 디스크 I/O)와 관련된 메트릭, 종합 (DB 계층의 성능, 캐시 계층의 성능) 메트릭, 비즈니스 (DAU, 수익, 재방문률) 메트릭 등이 존재합니다.
- 자동화 : 생산성을 높이기 위한 도구들을 뜻하며, CI/CD가 대표적입니다.

상태 아키텍처

상태 아키텍처를 설계하는 경우는 HTTP의 무상태성 (Staleless)를 위반하는 행위로, 대부분의 아키텍처에서 채택하고 있지 않으므로 생략한다.

>> 너무 당연한 내용이라서기도 함



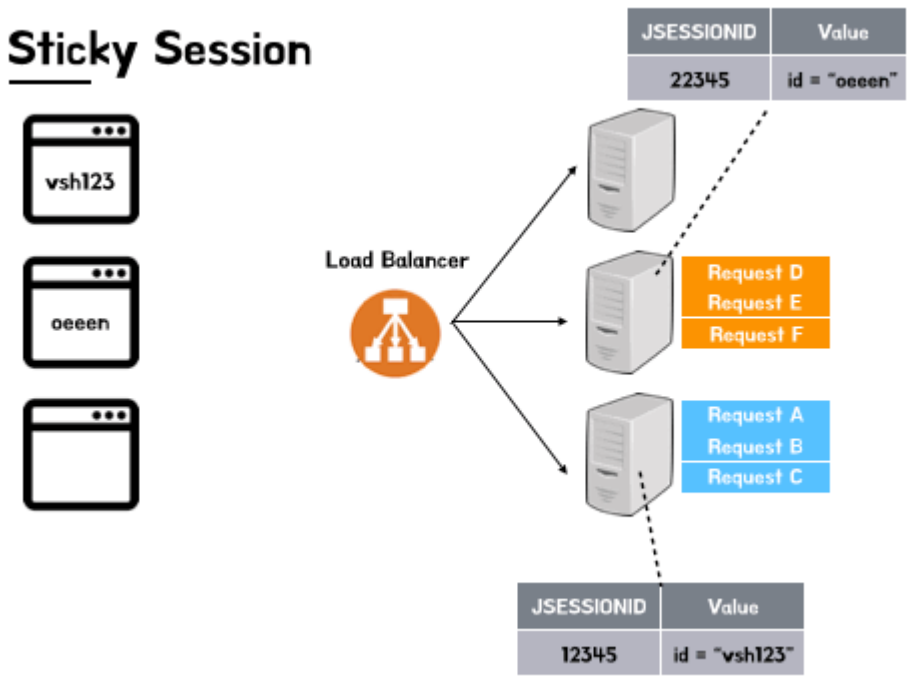
(유)상태 아키텍처

[고정 세션 (Sticky-Session)]

만약 무상태 아키텍처를 따르면서 세션 정보를 저장하여, 특정 클라이언트의 모든 요청을 항상 동일한 백엔드 서버로 보내기 위해서는 어떻게 해야할까?

바로 고정 세션 방식을 로드밸런서에 제공하여 사용 가능하다.

다만, 부하를 분산하기 위한 목적인 로드밸런서에 불필요한 부담을 주며, 로드밸런서 뒷단에 서버를 추가하거나 제거하기도 어려워진다.



부록

이미지 저장소 (Cloud Storage)

💡 로컬 이미지 저장소가 아닌, 클라우드 저장소를 사용하는 근본적인 이유

💡 이미지 저장소에 장애가 발생하는 경우에 대해 어떻게 대응해야 할지

참고 문헌

- 11번 부록 : 넷플릭스가 여러 데이터센터를 어떻게 데이터를 다중화하는지
- 14번 부록 : NoSQL의 다양한 활용 사례