가상 면접 사례로 배우는 대규모 시스템 설계 기초

적은 수의 사용자를 지원하는 시스템을 몇백만 이상의 대규모 사용자를 지원하는 시스템으로 확장, 발전 시켜볼 것이다.

1. 단일 서버

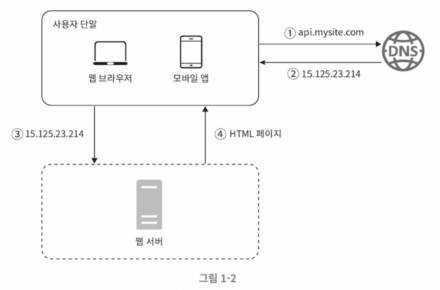


그림 1-2

1. 사용자가 사용하는 앱/웹 단말에서 도메인으로 웹사이트 접속을 시도
2. DNS가 도메인에 맞는 ip 주소 반환
3. ip 주소를 통해 웹서버에 요청
4. HTML, JSON 데이터를 반환

2. 데이터베이스

사용자가 늘어나면서, 트래픽 처리 용도와 분리하여 데이터베이스를 따로 두어야한다.

- 장점
  - 계층 분리를 통해 독립적으로 확장이 가능해진다.
- 데이터베이스 선택
  - RDBMS

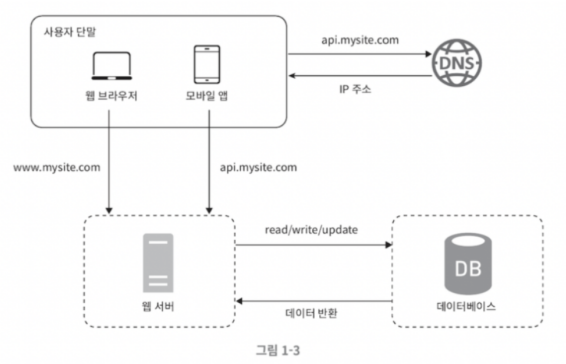


그림 1-3

- ex) MySQL, PostgreSQL
- Row, Column 구성되어 정형화 되어 있는 데이터를 저장
- 관계에 따라 join

- NoSQL
  - ex) Redis, MongoDB, DynamoDB
  - key-value / graph / column / document
  - 아주 낮은 지연시간 요구
  - 비정형 데이터
  - 많은 양의 데이터 저장

3. Scale up VS Scale out

Scale up

- 수직적 규모 확장 → 서버의 성능을 끌어 올리는 방법
- 하드웨어 자원 (CPU, RAM) 은 무한대로 증설할 수 없다.
- 자동 복구, 다중화 방안을 제시하지 않음 → 하나의 서버를 사용중이기에 서버 장애시 웹/앱 이 완전히 정지 된다.

Scale out

- 수평적 규모 확장 → 서버를 추가하여, 서버의 수를 늘리는 방법
- 로드 밸런서를 통해 서버간의 부하를 분산해야한다.

로드 밸런서

Scalue out을 통해 생긴 여러 웹 서버들에게 부하를 고르게 분산하는 역할 수행

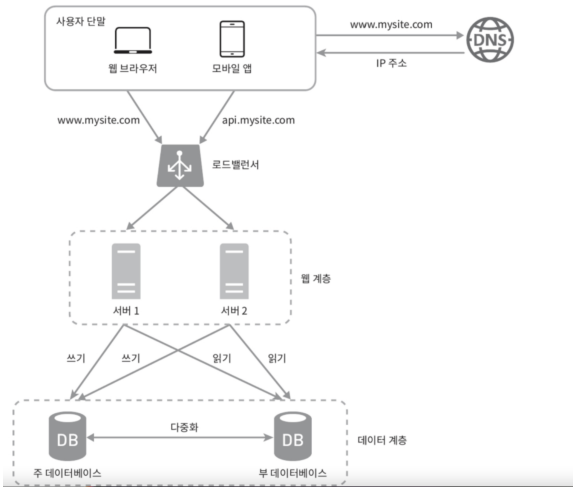
- 사용자 단말 (웹 / 앱) 에서 로드밸런서의 Public ip 주소로 접속

- 여러 서버 중 하나로 부하를 분산
- 보안을 위해 서버간 통신에 Private ip 사용

데이터베이스 다중화

데이터 계층에도 다중화가 필요

- master(write) - slave(read) 구조를 통해 다중화
  - 대부분 읽기 작업이 많기 때문에 slave를 여러대 두게된다.
- 장점
  - 성능 상승 ← 읽기 연산 요청은 분산되기 때문에
  - 안정성 ← 하나의 데이터베이스가 파괴되어도 다중화시켜 놓았기 때문에 데이터 보존
  - 가용성 ← 하나의 데이터베이스가 파괴되어도 다른 서버로 서비스 지속 가능



4. 캐시

자주 참조되는 데이터를 빠르게 처리할 수 있도록 캐시를 사용한다.

캐시 계층

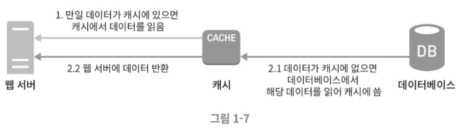


그림 1-7

- 데이터베이스도 계층을 분리하여 훨씬 빠르게 처리 가능
- 캐시 계층, 데이터베이스 계층을 독립적으로 확장시킬 수 있다.
- 이미지는 read-through 캐시 전략

캐시 사용 시 유의할 점

- 데이터 갱신은 자주 일어나지 않지만, 참조는 빈번하게 일어나는 상황에 고려
- 캐시는 휘발성이다.
- 만료 시간 정책을 고려하자
  - 만료 시간이 긴 경우 : 실시간성 떨어짐
  - 만료 시간이 짧은 경우 : 데이터베이스 접근이 많아짐
- 일관성을 고려하자
  - 원본과 캐시 데이터의 불일치 문제가 발생 할 수 있다.
- 장애 대응
  - SPOF를 피하기 위해 캐시 서버도 분산이 필요하다
- 캐시 메모리 크기가 너무 작으면 eviction이 자주 발생해 성능이 저하된다.
- eviction 정책 고려
  - LRU, LFU, FIFO 등 상황에 따라 맞는 정책 사용

5. CDN

정적 컨텐츠(이미지, 비디오, 파일 등)를 캐시할 수 있다.

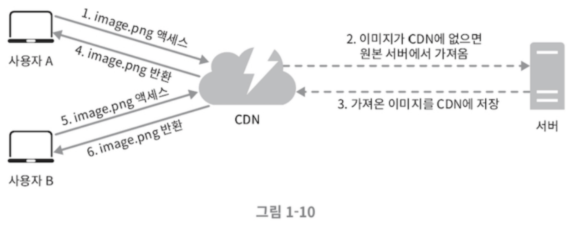


그림 1-10

read-through 캐시 전략을 사용한 CDN의 모습이다.

CDN 사용시 고려해야 할 사항

- 비용
  - 주로 3 사업자(AWS)에 의해 운영되는 경우가 많으며, CDN을 통하는 데이터 전송 양에 따라 요금을 내는 경우가 많음 → 적용할 데이터 고려
- 만료 시간 정책
- 컨텐츠 무효화 방법

- CDN에서 제공하는 API
- 버저닝을 통해 다른 버전의 컨텐츠 제공

6. 무상태(StateIss) 웹 계층

Scale out을 위해서는 상태 정보를 웹 계층에서 제거하는 것이 좋다. → 상태 정보를 RDB, NoSQL 에 저장

상태정보 의존적인 아키텍쳐

- 서버에서 상태 정보를 보관하고 있기 때문에 같은 클라이언트는 같은 서버로 요청을 해야 한다.
- 고정 세션 (sticky session)을 사용할 수 있지만 로드밸런서의 부담을 주게 된다.
  - 서버를 추가, 제거 하기에도 까다로워 진다.

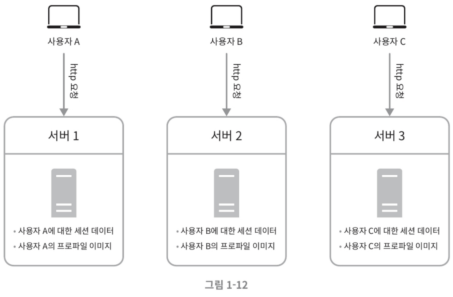


그림 1-12

무상태 아키텍쳐

- HTTP 요청은 어떤 웹 서버로도 전달 될 수 있다.
- 상태 정보가 필요할 경우 공유 저장소로 부터 데이터를 가져와 처리
- 단순한 구조, 안정적이고 규모 확장도 용이하다.

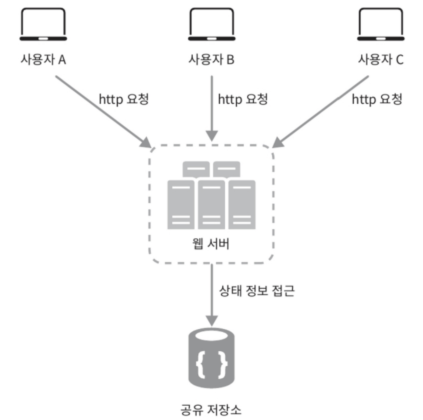


그림 1-13

7. 데이터 센터

사용자는 지리적 라우팅 (geoDNS-routing)을 통해 가까운 데이터 센터로 안내된다.

- 데이터 센터 적용시 해결해야할 기술적 난제
  - 트래픽 우회 → 올바른 데이터 센터로 트래픽을 보내는 방법 (geoDNS)
  - 데이터 동기화 → 데이터 센터별 데이터베이스의 서로 다른 데이터 동기화

메시지 큐

메시지의 무손실을 보장하는 컴포넌트

- 서비스간 결합을 느슨하게 하여 규모 확장을 용이하게 하고 애플리케이션이 서로 영향을 받지 않아 안정적으로 서비스 가능
- 버퍼 역할을 하며 비동기적으로 데이터를 전송한다.
- Producer : 메시지 발송 / Consumer : 메시지 처리

8. 로그, 메트릭, 자동화

규모가 커질 수록 로그, 메트릭, 자동화 도입이 필수적이다.

- 로그
  - 에러 로그 모니터링은 중요한 작업
  - 여러 컴포넌트를 오가는 요청에 로그를 하나의 서비스로 모아주는 도구를 활용하는 것이 좋다. (data Dog)
- 메트릭
  - 사업 현황에 유용한 정보를 얻고, 시스템 상태 파악을 쉽게 할 수 있다.
    - 호스트 단위 메트릭 : CPU, 메모리, 디스크 I/O
    - 종합 메트릭 : 데이터베이스 계층의 성능, 캐시 계층의 성능
- 자동화
  - 시스템이 커지고 생산성을 높이기 위해서는 자동화 도구를 활용해야 한다.

9. 데이터베이스의 규모 확장

Scale out

- 데이터베이스 크기 증가 (고성능 자원 사용)

Scale up

- 샤딩을 통해 데이터 분산
- 샤드 단위로 데이터를 분할하여 보관한다.
- 샤딩의 문제점
    - 데이터의 재 샤딩 → 샤드를 추가할 때 샤드 키를 다시 계산하여 재배치 해야한다.
        - 안정 해시 기법을 활용해 문제 해결
    - 유명인사 문제
        - 하나의 샤드의 질의가 집중되어 서버에 과부하가 발생하는 문제
    - 조인
        - 데이터가 샤드 서버로 쪼개졌기 때문에 데이터를 조인하기 힘들다.