# CP1. 신뢰할 수 있고 확장 가능하며 유지보수 하기 쉬운 애플리케이션



# Part 1. 데이터 시스템의 기초

## 1. 신뢰할 수 있고 확장 가능하며 유지보수 하기 쉬운 애플리케이션

### ▼ 개요

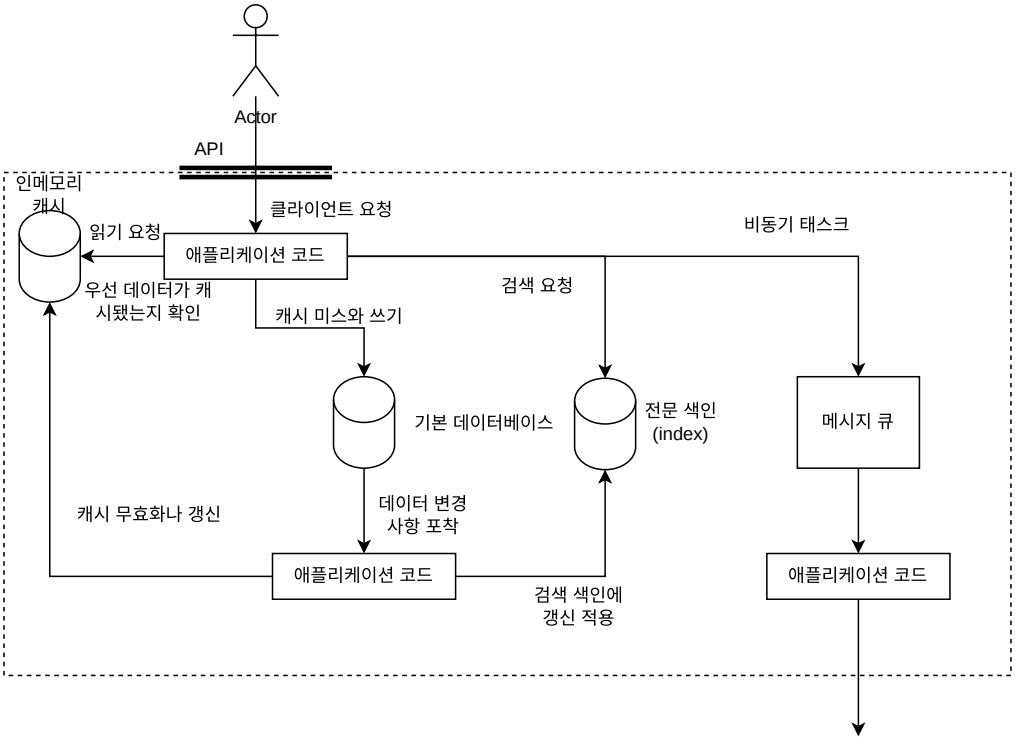
- 오늘날 애플리케이션은 **계산 중심** 보다 **데이터 중심** 적이다.
  이러한 애플리케이션의 경우 CPU 성능은 애플리케이션을 제한하는 요소가 아니며,
  더 큰 문제는
  **데이터의 양**, **데이터의 복잡도**, **데이터의 변화 속도** 이다.
- 일반적인 애플리케이션의 공통된 기능 (데이터 시스템이 추상화가 잘 되었기 때문)
  - Database
  - Cache (읽기 속도 향상)
  - Search Index (데이터 검색 필터링)
  - Stream Processing (비동기 처리를 위한 다른 Process 메시지 보내기)
  - Batch Processing (주기적으로 대량의 누적된 데이터를 분석)
- **중요한 건, 애플리케이션을 만들 때 어떤 도구와 어떤 접근 방식이 가장 적합한지 판단할 수 있어야 한다.**

### ▼ 데이터 시스템

- 구현 방식이 각각 다 다른데, 왜 **데이터 시스템**이라는 포괄적 용어를 사용할까?
  1. 새로운 도구들은 다양항 사용 사례에 최적화됐기 때문에 더 이상 전통적인 분류에 딱 들어맞지 않는다. (분류가 흐려지고 있음.)
     - 메시지 큐로 사용하는 datastore 인 레디스(Redis)

- DB 처럼 지속성을 보장하는 메시지 큐인 아파치 카프카(Apache Kafak)

2. 애플리케이션이 단일 도구로는 더 이상 데이터 처리와 저장을 모두 만족할 수 없이 과도하고 광범위한 요구사항을 갖고 있다. 대신 작업은 단일 도구에서 효율적으로 수행할 수 있는 태스크로 나누고 다양한 도구들은 애플리케이션 코드를 이용해 서로 연결한다.
    - Main DB 와 분리된 애플리케이션 관리 캐시 계층 (멤캐시디(Memcached)) 이나 엘라스틱서치(Elasticsearch)나 솔라(Solr) 같은 전문(full text) 검색 서버의 경우 메인 DB와 동기화된 캐시나 색인을 유지하는 것은 보통 애플리케이션 코드의 책임이다.
    - 그림) 다양한 구성 요소를 결합한 데이터 시스템 아키텍쳐의 예




- 서비스 제공을 위해 각 도구를 결합할 때 서비스 인터페이스나 애플리케이션 프로그래밍 인터페이스(API)는 보통 클라이언트가 모르게 구현 세부 사항을 숨긴다. 기본적으로 좀 더 작은 범용 구성 요소들로 새롭고 특수한 목적의 데이터 시스템을 만든다. 복합 데이터 시스템은 외부 클라이언트가 일관된 결과를 볼 수 있게끔 쓰기에서 캐시를 올바르게 무효화하거나 업데이트 하는 등의 특정 보장 기능을 제공할 수 있다.
- 데이터 시스템이나 서비스를 설계할 때 까다로운 문제가 많이 생긴다.
    1. 내부적으로 문제가 있어도 데이터를 정확하고 완전하게 유지하려면 어떻게 해야 할까?
    2. 시스템의 일부 성능이 저하되더라도 클라이언트에 일관되게 좋은 성능을 어떻게 제공할 수 있을까?
    3. 부하 증가를 다루기 위해 어떻게 규모를 확장할까?
    4. 서비스를 위해 좋은 API는 어떤 모습인가?
- 애플리케이션 시스템 설계시 고민해야 하는 점

- 관련자의 기술 숙련도
- 기존 시스템의 의존성
- 전달 시간 척도
- 다양한 종류의 위험에 대한 조직의 내성
- 규제 제약 등

## ▼ 신뢰성

- 정의 (아래 내용이 정상 동작 못한다고 가정하고 생각해보자 - 얼마나 중요한지 알게된다.)
  - 하드웨어나 소프트웨어 결함, 심지어 인적 오류 같은 역경에 직면하더라도 **시스템은 지속적으로 올바르게 동작**해야 한다.
  - 애플리케이션은 사용자가 **기대한 기능을 수행**한다.
  - 시스템은 **사용자가 범한 실수나 예상치 못한 소프트웨어 사용법을 허용**할 수 있다.
  - 시스템 **성능은 예상된 부하와 데이터 양에서 필수적인 사용 사례를 충분히 만족**한다.
  - 시스템은 **허가되지 않은 접근과 오남용을 방지**한다.
- 개요
  - IT 용어로 잘 못될 수 있는 일을 **결함(fault)**로 부른다.
  - 결함을 예측하고 대처할 수 있는 시스템을 **내결함성(fault-tolerant) 또는 탄력성(resilient)**이라 부른다.
  - 모든 케이스에 대한 내성은 실현 불가능함으로, **특정 유형**에 대한 결함 내성만을 일컫는다.
  - 결함은 **장애(failure)**와 동일하지 않다.
  - 일반적으로 **결함**은 사양에서 벗어난 시스템의 한 구성 요소로 정의되지만, **장애**는 사용자에게 필요한 서비스를 제공하지 못하고 시스템 전체가 멈춘 경우다.
  - 결함 확률을 0으로 줄이는 것은 불가능하다. 따라서 대개 결함으로 인해 장애가 발생하지 않게끔 내결함성 구조를 설계하는 것이 가장 좋다.
  - 내결함성을 높이는 방법
    - 내결함성 시스템에서 경고 없이 개별 프로세스를 무작위로 죽이는 것과 같이 고의적으로 결함을 일으켜 결함률을 증가시키는 방법은 납득할 만하다.
      이유는, 고의적으로 결함을 유도함으로써 내결함성 시스템을 지속적으로 훈련하고 테스트해서 결함이 자연적으로 발생했을 때 올바르게 처리할 수 있다는 자신감을 높인다. (카오스 몽키)

### ▼ 하드웨어 결함 (해결책이 있는 결함 유형)

- Disk 고장, RAM 결함, 정전 등
- 최근 이슈) 2023년도 판교 데이터 센터 화재로 인한 카카오 장애
- 결함 해결 방법

- **하드웨어 구성 요소에 중복을 추가하는 방법 (온프레미스)**
    - 클라우드 서비스들은 가상 장비 인스턴스가 별도의 경고 없이 사용할 수 없게 되는 상황이 상당히 일반적임, 단일 장비 신뢰성 보다 유연성과 탄력성을 우선적으로 처리하게끔 설계되었기 때문
- Disk 는 RAID(여러 개의 디스크를 하나로 묶어 하나의 논리적 디스크로 작동) 로 구성,
  서버는 이중 디바이스와 hot-swap(컴퓨터 시스템에 있어서 전원을 끄거나 시스템을 중지시키는 행위 없이 장치를 교체해서 사용이 가능한 기능) 가능한 CPU 등
- 구성 요소 하나가 죽으면 고장 난 구성 요소가 교체되는 동안 중복된 구성 요소를 대신 사용할 수 있다. 이 방식은 하드웨어 문제로 장애가 발생하는 것을 완전히 막을 수는 없지만 이해하기 쉽고 보통 수년 간 장비가 중단되지 않고 계속 동작할 수 있게 한다.
- 즉, 다중 장비 중복은 HA 보장이 필수적인 애플리케이션에만 필요했다.
- 하지만, 데이터 양과 애플리케이션의 계산 요구가 늘어나면서 더 많은 애플리케이션이 많은 수의 장비를 사용하게 됐고 이와 비례해 하드웨어 결함율도 증가했다.
- 따라서, 소프트웨어 내결함성 기술을 사용하거나 하드웨어 중복성을 추가해 전체 장비의 손실을 견딜 수 있는 시스템으로 옮겨지고 있다. (운영상 이점 - 장비를 부팅해야 하는 경우, 단일 서버는 계속된 중단 시간이 필요하지만, 장비 장애를 견딜 수 있는 시스템은 한 번에 한 노드씩 패치 할 수 있다.)

**▼ 소프트웨어 결함 (해결책이 있는 결함 유형)**

- 시스템 내 체계적 오류
    - 이 결함은 예상하기가 더 어렵고 노드 간 상관관계 때문에 상관관계 없는 하드웨어 결함보다 오히려 시스템 오류를 더욱 많이 유발하는 경향이 있다.
        - 잘못된 특정 입력이 있을 대, 모든 애플리케이션 서버 인스턴스가 죽는 소프트웨어 버그
            - 리눅스 커널의 버그로 인해 많은 애플리케이션이 일제히 멈춰버린 2012년 6월 30일 윤초
        - CPU 시간, 메모리, 디스크 공간, 네트워크 대역폭 처럼 공유 자원을 과도하게 사용하는 일부 프로세스
        - 시스템 속도가 느려져 반응이 없거나 잘못된 응답을 반환하는 서비스
        - 한 구성 요소의 작은 결함이 다른 구성 요소의 결함을 야기하고 차례차례 더 많은 결함이 발생하는 연쇄 장애
    - 결함 해결 방법
        - 소프트웨어 결함의 체계적 오류 문제는 신속한 해결책이 없다.
        - 시스템의 가정과 상호작용에 대해 주의 깊게 생각하기, 빈틈없는 테스트, 프로세스 격리, 죽은 프로세스의 재시작 허용, 프로덕션 환경에서 시스템 동작의 측정, 모니터링, 분석하기 등 과 같은 여러 작은 일이 문제 해결에 도움을 준다.

▼ **인적 오류 (해결책이 있는 결함 유형)**

- 사람은 소프트웨어 시스템을 설계하고 구축하며, 운영자로서 시스템을 계속 운영한다. 이들이 최선의 의도를 갖고 있어도 사람은 실수하기 마련이다.
    - 책에 나온 정보로는 전체 결함율의 주축이 운영자의 사용 미숙 이다.
- 결함 해결 방법
    - 오류 가능성을 최소화하는 방향으로 시스템 설계해라
        - 잘 설계된 추상화, API, 관리 인터페이스를 사용하면 옳은 일은 쉽게 하고, 잘못된 일은 막을 수 있다.
        - 사람이 가장 많이 실수하는 장소에서 사람의 실수로 장애가 발생할 수 있는 부분을 분리하라.
        - 단위 테스트부터 전체 시스템 통합 테스트와 수동 테스트까지 모든 수준에서 철저하게 테스트하라.
        - 장애 발생의 영향을 최소화하기 위해 인적 오류를 빠르고 쉽게 복구할 수 있게 하라.
        - 성능 지표와 오류율 같은 상세하고 명확한 모니터링 대책을 마련해라.

- 백엔드 Java, Spring boot 도구
    - 부하 테스트
        - JMETER
    - 모니터링
        - **마이크로미터**는 자바 애플리케이션의 메트릭을 수집하는 도구
        - **프로메테우스**는 이러한 메트릭 데이터를 수집, 저장하고 쿼리하는 시스템
        - **그라파나**는 이를 시각적으로 표현하여 모니터링하는 도구입니다.
        - 이들을 함께 사용하면 효과적인 시스템 모니터링 및 디버깅이 가능합니다

## ▼ 확장성

- 정의
    - 확장성은 증가한 부하에 대처하는 시스템 능력을 설명하는 데 사용하는 용어이다.
        - 시스템이 특정 방식으로 커지면 이에 대처하기 위한 선택은 무엇인가?
        - 추가 부하를 다루기 위해 계산 자원을 어떻게 투입할까?
- 개요
    - 시스템이 현재 안정적으로 동작한다고 해서 미래에도 안정적으로 동작한다는 보장은 없다.
    - 성능 저하를 유발하는 흔한 이유 중 하나는 부하 증가다.
        - 시스템은 과거에 처리했던 양보다 더 많은 데이터를 처리하고 있을지 모른다.

- 부하 기술하기
  - 부하는 **부하 매개변수**라 부르는 몇 개의 숫자로 나타낼 수 있다.
    가장 적합한 부하 매개변수 선택은 시스템 설계에 따라 달라진다.
    - 부하 매개변수는 웹 서버의 초당 요청 수, 데이터베이스의 읽기 대 쓰기 비율, 대화방의 동시 활성 사용자, 캐시 적중률 등이 될 수 있다.
    - 부하 매개변수는 평균적인 경우가 중요할 수도 있고, 소수의 극단적인 경우가 병목 현상의 원인일 수 있다.
  - ex) 트위터 트윗 write, read 예
    - 트위터는 주로 read 트윗이 더 많다. 즉 읽기가 더 부하를 일으킨 대상임으로, 최대한 읽기에 대한 시간을 줄이고자, 공간적 확보를 통해 write 작업시 follow 수에 따른 만큼 follow 계정 타임 라인에 데이터를 넣어 write 시 작업을 조금 더 부하를 주고 read 때 부하를 줄이는 방식으로 선택되었다.
    - 하지만, 평균 트윗 수는 75명 팔로워 이지만, 일부 사용자는 팔로워가 3천 만 명이 넘는다. 이때 write 작업시 3000만에게 write 를 한다는 것이다… 이를 위해 트위터는 사용자가 팔로우한 유명인의 트윗은 별도로 가져와 (read 시 Query)로 읽는 시점에 홈 타임 라인에 합친다.
- 성능 기술하기
  - 시스템 부하를 기술하면 부하가 증가할 때 어떤 일이 일어나는지 조사할 수 있다.
    아래 내용 모두 성능 수치가 필요하다.
    - 부하 매개변수를 증가시키고 시스템 자원(CPU, 메모리, Network 대역폭 등)은 변경하지 않고 유지하면 시스템 성능은 어떻게 영향을 받을까?
    - 부하 매개변수를 증가시켰을 때 성능이 변하지 않고 유지되길 원한다면 얼마나 많은 자원을 늘려야 할까?
  - 시스템 성능
    - 하둡 같은 일관 처리 시스템은 보통 **처리량(throughput)** (초당 처리할 수 있는 레코드 수나 일정 크기의 데이터 집합으로 작업을 수행할 때 걸리는 전체 시간)이 중요
    - 온라인 시스템는 서비스 **응답 시간(response time)** (클라이언트가 요청하고 응답을 받는 시간)이 중요
      - 응답 시간은 매번 다를 수 있음으로, **측정 가능한 분포**로 생각해야 한다.
        - 대부분의 요청은 꽤 빠르지만, 가끔 꽤 오래 걸리는 **특이 값(outlier)**이 있다.
        - 위 트윗 예시처럼 특정 요청에만 데이터가 오래 걸리는 것일 수도 있지만, 백그라운드 프로세스의 컨텍스트 스위치, 네트워크 패킷 손실과 TCP 재전송, 가비지 컬렉션 휴지, 디스크에서 읽기를 강제하는 페이지 디폴트, 서버 랙의 기계적인 진동이나 다른 원인으로 추가 지연이 생길 수 있음
      - 평균 응답 시간

        - 산술 평균 n 값이 주어지면 `(a1 + a2 + a3… + an) / n`
          하지만, 이는 좋은 지표가 아니다. 얼마나 많은 사용자가 지연을 경험했는지 모름으로,
          **백분위를 사용한 지표가 좋다.**
    - **백분위**
        - **서비스 수준 목표(SLO), 서비스 수준 협약서(SLA)**에 사용되는 중요 지표임
        - 중앙값
            - 백분위로 응답 시간 목록을 sort 후 중간 지점
            - 사용자가 보통 얼마나 오랫동안 기다려야 하는지 알고 싶은지
        - 특이 값
            - 백분위로 95 분위, 99 분위, 99.9분위가 일반적으로 특이 값을 확인하기 좋다.
        - **꼬리 지연 시간(tail latency)**
            - 최상위 백분위 응답 시간이라고 하고 서비스의 중요한 사용자 경험에 직접 영향을 주기 때문에 중요하다.
                - AWS 에서는 꼬리 지연 시간을 99.9 분위로 취급하는 데, 보통 응답 시간이 가장 느린 요청을 경험한 고객들이 많은 구매를 하여 가장 많은 데이터를 갖고 있는 유저이다. (VIP 라는 뜻)
        - 큐 대기 지연은 높은 백분위에서 응답 시간의 상당 부분을 차지한다.
            - 서버는 병렬로 소수의 작업만 처리할 수 있기 때문에 (ex) CPU 코어 수) 소수의 느린 요청 처리만으로도 후속 요청 처리가 지체된다.
              이러한 현상을
              **선두 차단**이라고 한다.
- 부하 대응 접근 방식
    - **아키텍처**
        - 부하 규모의 자리 수가 바뀔 때 마다 검토해야한다.
        - 결정하는 요소로는 **읽기의 양, 쓰기의 양, 저장할 데이터의 양, 데이터 복잡도, 응답 시간 요구사항, 접근 패턴** 등이 있다.
        - 용량 확장(scaling up), 수직 확장(vertical scaling)
            - 더 좋은 스펙의 장비로 이동
        - 규모 확장(scaling out), 수평 확장(horizontal scaling)
            - 다수의 낮은 장비에 부하를 분산 - 비공유 아키텍처
            - 상태

        - 다수의 장비는 상태 비저장(stateless) 서비스는 간단하다. (비상태 - Jwt token)
        - 다수의 장비의 상태 유지(stateful) 서비스를 분산 설치하는 일은 많은 복잡도가 추가적으로 발생한다. (상태 - Session 공유를 위한 세션 클러스팅 - 요즘은 Redis, Memcached를 사용)
    - 일부 시스템은 탄력적이다.
        - 부하를 감지하면 컴퓨팅 자원을 자동으로 추가할 수 있다. 그렇지 않은 시스템은 수동으로 확장해야 한다.
        - 탄력적인 시스템은 부하를 예측할 수 없을 만큼 높은 경우 유용하지만, 수동으로 확장하는 시스템이 더 간단하고 운영상 예상치 못한 일이 더 적다.
    - **애플리케이션에 적합한 확장성을 갖춘 아키텍처는 주요 동작이 무엇이고, 잘 하지 않는 동작이 무엇인지에 대한 가정을 바탕으로 구축한다. 이 가정은 곧 부하 매개변수가 된다.**
- 트랜드
    - 상태 비저장 서비스로 배포하며 수평 확장이 되도록 하는 시스템을 추구로 한다.
    - 이전 서비스들의 한에서만 수직 확장을 하려고 한다.

## ▼ 유지보수성