데이터 시스템에 대한 생각

- 왜 "데이터 시스템"이라고 포괄적으로 묶어 생각할까?
    - 특정 분류와 역할에 맞게 사용하기 어려움. 분류간 경계가 흐려지고 있음
- 하나의 애플리케이션을 단일 도구로 데이터 처리 저장 모두 만족하기 어려움
    - 대신 테스크 별 데이터베이스를 선택하기도 함.

신뢰성

- 올바르게 동작함 <-> 잘못될 수 있는 일 : 결함
- 결함을 예측하고 대처하는 시스템 : 내결함성 or 탄력성
- 결함 (버그, 에러 수준) != 장애 (사용자가 서비스 이용 못하는것)
- 고의적으로 많은 결함을 일으켜 오류 처리를 다양화 (넷플릭스의 카오스 몽키)
    - 일부러 시스템을 다운시킴
    - AWS fault injection service https://aws.amazon.com/ko/fis/
    - 관측 가시성 (모니터링 할 수 있는 대시보드 등)

하드웨어 결함

- 하드웨어 구성 요소에 중복을 추가하는 방법이 일반적.
- 예비 전원 디젤 발전기, 핫스왑 가능한 CPU, 이중 전원 디바이스 등

소프트웨어 오류

- 소프트웨어의 체계적 오류 문제는 신속한 해결책이 없다
- 주의깊게 생각하기, 빈틈없는 테스트, 프로세스 격리, 프로세스 재시작 허용, 프로덕션 환경에서 시스템 동작의 측정, 모니터링, 분석하기 등 작은 일들이 문제해결에 도움을 줌

인적 오류

- 잘 설계하기
- 실수 많이하는 부분의 분리 / 비 프로덕션 샌드박스 제공
- 철저한 테스트 (자동 테스트, 코너 케이스)
- 롤백 롤아웃 잘 하도록 만들기
- 모니터링 대책 마련
- 실습, 조작교육 하기

확장성

- 부하 증가에 대처하는 시스템 능력 (scailability)
    - 대기열을 어떻게 두고 있는지, 부하 대처 능력 같은 개념.

부하 기술하기

- 부하 매개변수로 현재 부하를 기술.
  - 웹 서버의 초당 요청 수
  - 데이터베이스의 읽기 대 쓰기 비율
  - 대화방의 동시 활성 사용자
  - 캐시 적중률 등

성능 기술하기

- 시스템 성능 수치
  - 지연 시간 (전송되는데 걸리는 시간)
  - 처리량 (초당 처리 가능한 레코드 수나 일정 크기의 데이터 집합)
  - 온라인의 경우 응답 시간 (클라이언트 요청 및 응답 시간)
    - SLO, SLA를 준수. 안그럼 환불 요청도 가능
- 응답시간
  - 분위수가 높은 이상치로 성능을 판단할 수 있음
  - 꼬리 지연 시간은 고객 사용 경험에 직접 영향을 미침
  - 테스트용 부하의 경우 응답 시간과 독립적으로 요청을 지속해서 보내야 함. 기존 환경과 동일하게~

부하 대응 접근 방식 (1)

- 용량 확장 (수직 확장) == 강력한 장비로 이동
- 규모 확장 (수평 확장) == 낮은 사양 장비에 부하를 분산
  - 분산하는 아키텍쳐를 **비공유 아키텍처**라고 부른다
- 탄력적으로 운용 (범용적인건 없다)
  - 읽기의 양, 쓰기의 양, 저장할 데이터의 양, 데이터 복잡도, 응답시간 요구사항, 접근 패턴 등에 맞춰서 ...
  - 예시) AWS에서 연설한 내용.
    BTS 접속 시간에 맞춰서 lambda 이용해서 이벤트 발생 시 노드 발란싱 이용해서 탄력적으로 운용

부하 대응 접근 방식 (2)

- 데이터베이스의 다중화 : 날라갈 수 있는 것들에 대한 방어 방식~
  - 클러스터링 : 저장소는 하나 DB 서버는 다중화. -> 예전 방식. 돈 많은 데나 씀. (은행, 증권사 : 안정성) 단일 고정점을 가지는 서비스. 저장소가 고장나면 답없음.
    - Active-Active 두대 서버 돌림 (릴레이션 ? 로드 밸런스가 중간에 있고 쉬고 있는)
    - Active-Standby 하나만 돌리다가 고장나면 쉬던거 돌려주기
      - 콜드 스텐바이 : 꺼져있는거 키기
      - 핫 스텐바이 : active된 서버에 응답 시간 계속 체크(하트비트) - 통신 계속 시도
  - 래플리케이션 : 저장소, DB 둘 다 다중화0
  - 샤딩 : 저장소, DB 모든 자원 분리. 장애 발생시 커버링 구성 필요. (좁은 범위 안에서 분할) 예시. 구글의 경우 엄청 많은 데이터를 가지고 있기 때문에 지역별 분할 한국에서 검색하면 한국 데이터베이스에서 가져오고, ... == 종분할 (테이블을 가로로 자르는 것)
    - 횡분할 : 논리적인 파티셔닝이 될 수 있고, 물리적인 파티셔닝이 될 수 있음 보통 도메인 단위로 분할됨.

유지보수성

- 소프트웨어 시스템 설계 원칙 세가지
  - 운용성 : 운영의 편리함 만들기 (자동화 시스템 등)
  - 단순성 : 복잡도 관리
    - 를 제거하기 위한 최상의 도구는 추상화다.
  - 발전성 : 변화를 쉽게 만들기
    - 애자일하게 (테스트 주도 개발, 리팩토링)

풀잎스쿨 첨언

- Amazon Aurora Serverless : 자동화된 용량 확장 및 축소 -> 비용을 가시화 하는 솔루션
  - 성능 모니터링 (performance insight) 지표 확인하려면 계속 메모리 잡아먹고 있음.. 비용 결코 적게 들지 않음
  - AWS는 비용이 너무 많이듬. 클라우드 비용 줄이는 방식 (서버리스 비싸...)
- 데이터 센터와 환경 https://www.youtube.com/watch?v=UTRBVPvzt9w&t=3839s
  - Database Reliability Engineering (DBRE)
- 모니터링 필요
  - 내부 배포 플랫폼 이용. 프로젝트별로 쓰는 리소스 볼 수 있게...
- 데이터베이스에서 외래키를 이용할 때,
  - 근래에는 RDBMS를 많이 사용하고, 관계형 DB에서는 무결성 검증 외래키를 수정하고 연결된 사항에 대해서는 반영하지 않도록 함. 해당 키를 바탕으로 Join해서 값을 찾는 방식을 사용.