전자 지갑

🔥 고객이 지갑에 돈을 넣어두고 필요할 때 사용하도록 하는 결제 플랫폼의 전자 지갑 서비스를 설계하자

▼ 1️⃣ 설계 범위 확정

메인 기능 : 두 전자 지갑 사이의 이체

- 1,000,000 TPS
- 정확성 요건 : 데이터베이스가 제공하는 트랜잭션 보증
- 처음부터 데이터를 재생하여 언제든지 과거 잔액을 재구성할 수 있는 시스템을 만들어야 함
- 가용성 요건 : 99.99%

개략적 추정

노드 당 천 TPS를 지원할 수 있다고 가정한다면, **백만 TPS를 위해 1000개의 데이터베이스가 필요**하겠군 !

> 🦹‍♀️ 이체 명령은 두 번의 연산(A 계좌에서 인출, B 계좌에 입금)이 필요하기 때문에 **2000개**가 필요
>
> 🤔 그건 좀 ...
>
> **목표. 단일 노드가 처리할 수 있는 트랜잭션 수를 늘리자 !**

▼ 2️⃣ 개략적 설계

API 설계

POST /v1/wallet/balance_transfer

한 지갑에서 다른 지갑으로 자금을 이체한다.

Request Body

- **from_account**(string): 돈을 인출할 계좌

- **to_account**(string): 돈을 이체할 계좌
- **amount**(string): 이체할 금액
- **currency**(string): 통화 단위
- **transaction_id**(uuid): 중복 제거에 사용할 ID

Response Example

```
{
    "status": "success",
    "transaction_id": "01589980-2664-11ec-9621-0242ac130002
```

인메모리 샤딩

`사용자-잔액` 관계를 나타내기 위한 자료 구조는 **키-값 저장소 !**

레디스

한 대로 100만 TPS 처리는 어렵다.

- 클러스터 구성
- 사용자 계정을 모든 노드에 균등 분산 (파티셔닝, 샤딩)

주키퍼

모든 레디스 노드의 파티션 수 및 주소를 한 곳에 저장

지갑 서비스

1. 이체 명령의 수신
2. 이체 명령의 유효성 검증
3. 명령이 유효할 경우 이체에 관계된 두 계정의 잔액 갱신

🌊 작업 흐름

- 계정 잔액은 여러 레디스 노드에 분산
- 주키퍼는 샤딩 정보 관리에 사용
- 지갑 서비스는 무상태 서비스

- 주키퍼에 샤딩 정보를 질의하여 특정 클라이언트의 정보를 담은 레디스 노드를 찾고 잔액을 갱신

→ 첫 번째 업데이트 후 두 번째 업데이트 완료 전 지갑 서비스 노드가 다운된다면? 😱

→ **원자적 트랜잭션**으로 연산이 실행되어야 한다 !

분산 트랜잭션

데이터베이스 샤딩

서로 다른 두 개 저장소 노드를 갱신하는 연산을 원자적으로 수행하려면?

1. **각 레디스 노드를 트랜잭션을 지원하는 관계형 데이터베이스 노드로 교체**
    - 한 이체 명령이 서로 다른 두 데이터베이스 서버에 있는 계정 두 개를 업데이트 한다면 동시에 처리된다는 보장이 없음
2. **2단계 커밋(2PC)**
    a. 지갑 서비스는 정상적으로 여러 데이터베이스에 읽기/쓰기 작업 수행 (락을 걸고)
    b. 애플리케이션이 트랜잭션을 커밋하려 할 때 지갑 서비스는 데이터베이스에 트랜잭션 준비를 요청
        i. 모든 데이터베이스가 정상 응답하면 지갑 서비스는 모든 데이터베이스에 트랜잭션 커밋 요청
        ii. 하나라도 정상 응답하지 못하면 지갑 서비스는 모든 데이터베이스에 트랜잭션 중단 요청

    단점
    - 저수준 방안, 준비 단계를 실행하려면 트랜잭션 실행 방식을 변경해야 함
    - 이기종 데이터베이스 간 2PC 실행을 위해 모든 데이터베이스가 X/OPEN XA 표준을 만족해야 함
    - 다른 노드의 메시지를 기다리는 동안 락이 오랫동안 잠겨 성능이 좋지 않음 (SPOF 위험)
3. **TC/C(보상 기반 분산 트랜잭션)**
    a. 지갑 서비스는 모든 데이터베이스에 트랜잭션에 필요한 자원 예약을 요청

b. 지갑 서비스는 모든 데이터베이스로부터 회신 받음

   i. 모두 정상 응답하면 모든 데이터베이스에 작업 확인을 요청 **(시도-확정 절차)**

   ii. 하나라도 정상 응답하지 못하면 모든 데이터베이스에 작업 취소 요청 **(시도-취소 절차)**

> 🤔 **2PC vs TC/C ?**
>
> **2PC**
> 의 두 단계는 한 트랜잭션, **TC/C**의 각 단계는 별도 트랜잭션 !
> TC/C는 실행 취소 절차를 비즈니스 로직으로 구현하여 고수준 해법임 (데이터베이스에 구애 X)
> 단점은 비즈니스 로직에서 세부 사항을 관리하고 복잡성을 처리해야 함

> 🤔 **TC/C 실행 도중에 지갑 서비스가 다시 시작된다면?**
>
> TC/C의 진행 상황, 각 단계 상태 정보를 트랜잭션 데이터베이스에 저장하자 !
> 분산 트랜잭션 ID, 내용, 각 데이터베이스에 대한 단계값, 플래그 등 ...
> 일반적으로 돈을 인출할 지갑의 계정이 있는 데이터베이스에 둔다.

4. **사가**

   a. 모든 연산은 순서대로 정렬

   b. 각 연산은 자기 데이터베이스에 독립 트랜잭션으로 실행

   c. 연산은 순서대로 실행

   d. 연산이 실패하면 전체 프로세스는 실패한 연산부터 역순으로 보상 트랜잭션을 통해 롤백

연산 실행 순서 정하기

- 분산 조율: 사가 트랜잭션에 관련된 모든 서비스가 다른 서비스의 이벤트를 구독하여 작업 수행
- 중앙 집중형 조율: 하나의 조정자가 모든 서비스가 올바른 순서로 실행되도록 조율

🤔 **TC/C vs 사가 ?**

둘 다 애플리케이션 수준의 분산 트랜잭션.
TC/C는 취소 단계에서, 사가는 롤백 단계에서 보상 트랜잭션이 실행됨
TC/C는 작업 실행 순서가 임의로 지정되므로 병렬 실행이 가능

→ 지연 시간 요구사항이 없다면 둘 다 사용 가능.
msa에서 흔히 하는대로 한다면 사가
→
지연 시간에 민감하다면 TC/C가 낫다.

이벤트 소싱

1. 특정 시점의 계정 잔액을 알 수 있나요?
2. 과거 및 현재 계정 잔액이 정확한지 어떻게 알 수 있나요?
3. 코드 변경 후에도 시스템 로직이 올바른지 어떻게 검증하나요?

용어

명령

- 외부에서 전달된 의도가 명확한 요청
- FIFO 큐에 저장되어 순서를 가짐

이벤트

- 명령의 유효성을 검사한 후 유효한 명령은 반드시 이행하고 결과를 남김
- 검증된 사실, 실행이 끝난 상태
- 특징
  - 하나의 명령으로 여러 이벤트가 만들어질 수 있다.
  - 이벤트 생성 과정에는 무작위성이 개입될 수 있어서 같은 명령에 항상 동일한 이벤트가 만들어진다는 보장은 없다.

상태

- 이벤트가 적용될 때 변경되는 내용
- 키-값 저장소를 사용

상태 기계

- 이벤트 소싱 프로세스를 구동
    - 명령의 유효성을 검사하고 이벤트를 생성
    - 이벤트를 적용하여 상태를 갱신
- 무작위성 내포 불가, 결정론적으로 동작해야 함

지갑 서비스 예시

1. 명령(=요청)을 FIFO 큐(카프카)에 기록
2. 상태(=계정 잔액)은 관계형 데이터베이스에 저장
3. 상태 기계는 명령을 큐에 들어간 순서대로 확인
4. 명령 하나를 읽을 때마다 계정에 충분한 잔액이 있는지 확인

재현성

이벤트를 처음부터 다시 재생하면 과거 잔액 상태는 언제든 재구성할 수 있다!

이벤트 리스트는 *불변*이고 **상태 기계 로직**은 *결정론적*이기 때문에 이벤트 이력을 재생하여 만든 상태는 항상 동일하다.

```
1. 특정 시점의 계정 잔액을 알 수 있나요?
- 시작부터 계정 잔액을 알고 싶은 시점까지 이벤트를 재생하면 가능

2. 과거 및 현재 계정 잔액이 정확한지 어떻게 알 수 있나요?
- 이벤트 이력에서 계정 잔액을 다시 계산하면 정확한지 확인 가능

3. 코드 변경 후에도 시스템 로직이 올바른지 어떻게 검증하나요?
- 새로운 코드에 동일한 이벤트 이력을 입력으로 주고 같은 결과가 나오는지
```

감사 가능 시스템이어야 한다는 요건 때문에 **이벤트 소싱이 지갑 서비스 구현의 실질적인 솔루션**이 되는 경우가 많다.

명령-질의 책임 분리(CQRS)

클라이언트가 계정 잔액을 알도록 할 방법이 없을까?

1. 상태 이력 데이터베이스의 읽기 전용 사본을 생성하여 외부와 공유한다.
2. 이벤트 소싱을 사용하여 이벤트를 수신하는 외부 주체가 직접 상태를 재구축하도록 한다.

상태 기록을 담당하는 상태 기계는 하나, 읽기 전용 상태 기계는 여러 개

읽기 전용 상태 기계는...

- 상태 뷰를 만들어 질의에 이용한다.
- 이벤트 큐에서 다양한 상태 표현을 도출할 수 있다.
- 결과적 일관성을 보장한다.

▼ 3️⃣ 상세 설계

고성능 이벤트 소싱

파일 기반 명령/이벤트 목록

1. `명령`과 `이벤트`를 로컬 디스크에 저장하자!

   → 네트워크를 통한 전송 시간을 피할 수 있다.

   - 이벤트 목록: 추가 연산만 가능한 자료 구조에 저장 (순차적 쓰기 연산 → 매우 빠름)
2. 최근 명령과 이벤트는 메모리에 캐시하자!
3. mmap
   - 최적화 구현에 유용
   - 로컬 디스크에 쓰는 동시에 최근 데이터는 메모리에 자동으로 캐시 (실행 속도 향상)

파일 기반 상태

`상태 정보`도 로컬 디스크에 저장하자!

- 파일 기반 로컬 관계형 데이터베이스 (SQLite)
- 📍 로컬 파일 기반 키-값 저장소 (RocksDB)

- 쓰기 작업에 최적화된 자료 구조(LSM)를 사용함

스냅샷

모든 것이 파일 기반이라면 과거 특정 시점의 상태를 파일에 저장해서 시간을 절약하자!

- 변경 불가능
- 거대한 이진 파일 → HDFS와 같은 객체 저장소에 저장
- 컴퓨터 하드웨어의 I/O 처리량을 최대로 활용 가능

신뢰할 수 있는 고성능 이벤트 소싱

신뢰성 ?

시스템 신뢰성 문제 == 데이터 신뢰성 문제

1. 파일 기반 명령
2. ⭐ 파일 기반 이벤트
3. 파일 기반 상태
4. 상태 스냅샷

합의

높은 안정성을 위해 이벤트 목록을 여러 노드에 복제하자.

1. 데이터 손실 없는 복제
2. 로그 파일 내 데이터의 상대적 순서는 모든 노드에 동일할 것

→ 합의 기반 복제 방안이 적합하다 ! 모든 노드가 동일한 이벤트 목록에 합의하도록 보장

💡 **래프트 알고리즘**

노드의 절반 이상이 온라인 상태면 그 모두에 보관된 append-only 리스트는 같은 데이터를 가짐
- 노드는 세 가지 역할을 가질 수 있음

1. 리더 (최대 1개): 외부 명령을 수신하고 클러스터 노드 간 데이터를 안정적으로 복제
2. 후보
3. 팔로워

래프트 알고리즘을 사용하면 과반수 노드가 작동하는 한 시스템은 안정적일 수 있다 !

분산 이벤트 소싱

아직 해결하지 못한 문제들 ...

1. 전자 지갑 업데이트 결과를 즉시 받고 싶지만 CQRS 시스템에서는 느릴 수 있다.
2. 단일 래프트 그룹의 용량은 제한되어 있어 일정 규모 이상에서는 샤딩 및 분산 트랜잭션 구현이 필요하다.

풀 vs 푸시

- 풀 모델
  - 외부 사용자가 읽기 전용 상태 기계에서 주기적으로 실행 상태를 읽음
  - 실시간이 아님
  - 읽는 주기가 너무 짧으면 지갑 서비스에 과부하

  외부 사용자 ↔ 리버스 프록시 ↔ 이벤트 소싱 노드로 개선 가능
- 푸시 모델
  - 읽기 전용 상태 기계가 특별한 로직을 가질 수 있다
    - 읽기 전용 상태 기계가 이벤트를 수신하자마자 실행 상태를 역방향 프록시에 푸시하도록 한다.

실시간 같기도 🤔

분산 트랜잭션

1. 사용자 A가 사가 조정자에게 분산 트랜잭션을 보낸다.
2. 사가 조정자는 단계별 상태 테이블에 레코드를 생성하여 트랜잭션 상태를 추적한다.
3. 사가 조정자는 작업 순서를 검토한 후 출금을 먼저 처리하기로 결정한다.
4. 조정자는 출금 명령을 계정 A 정보가 들어있는 파티션 1로 보낸다.
5. 파티션 1의 래프트 리더는 출금 명령을 수신하고 명령 목록에 저장한다.
6. 명령의 유효성을 검사하여 유효하면 이벤트로 변환한다. (래프트 합의 알고리즘)
7. 이벤트가 동기화되면 파티션 1의 이벤트 소싱 프레임워크가 CQRS를 사용하여 데이터를 읽기 경로로 동기화한다.
8. 읽기 경로는 상태 및 실행 상태를 재구성한다.
9. 파티션 1의 읽기 경로는 이벤트 소싱 프레임워크를 호출한 사가 조정자에게 상태를 푸시한다.
10. 사가 조정자는 파티션 1에서 성공 상태를 수신한다.
11. 사가 조정자는 단계별 상태 테이블에 파티션 1의 작업이 성공했음을 나타내는 레코드를 생성한다.
12. 첫 번째 작업이 성공했으므로 사가 조정자는 두 번째 작업인 입금을 실행한다.
13. 조정자는 계정 C의 정보가 포함된 파티션 2에 입금 명령을 보낸다.
14. 파티션 2의 래프트 리더가 입금 명령을 수신하여 명령 목록에 저장한다.
15. 유효한 명령으로 확인되면 이벤트로 변환된다. (래프트 합의 알고리즘)
16. 이벤트가 동기화되면 파티션 2의 이벤트 소싱 프레임워크는 CQRS를 사용하여 데이터를 읽기 경로로 동기화한다.
17. 읽기 경로는 상태 및 실행 상태를 재구성한다.
18. 파티션 2의 읽기 경로는 이벤트 소싱 프레임워크를 호출한 사가 조정자에게 상태를 푸시한다.
19. 사가 조정자는 파티션 2에서 성공 상태를 수신한다.
20. 사가 조정자는 단계별 상태 테이블에 파티션 2의 작업이 성공했음을 나타내는 레코드를 생성한다.

21. 모든 작업이 성공하고 분산 트랜잭션이 완료되며 사가 조정자는 호출자에게 결과를 응답한다.

▼ 4️⃣ 마무리

1. 레디스 같은 인메모리 키-값 저장소를 사용하는 솔루션
    - 데이터 내구성 부족
2. 인메모리 캐시를 트랜잭션 데이터베이스로 변경
    - 2PC, TC/C, 사가, ...
    - 데이터 감사가 어려움
3. 이벤트 소싱
    a. 외부 데이터베이스와 큐
        - 성능이 좋지 않음
    b. 명령, 이벤트, 상태 데이터를 로컬 파일 시스템에 저장
        - SPOF
    c. 래프트 합의 알고리즘
        - 이벤트 목록을 여러 노드에 복제
        - 시스템 안정성을 높임
4. CQRS
5. 리버스 프록시
    - 외부 사용자에게 비동기 이벤트 소싱 프레임워크를 동기식 프레임워크로 제공하기 위해