호텔 예약 시스템

▼ 1️⃣ 설계 범위 확정

📝 질문거리

기능 요구사항

- 시스템 규모
- 예약 대금 지불 방식
- 예약 방식
- 예약 취소 가능 여부
- 초과 예약 가능 여부
- 객실 가격 변동

비기능 요구사항

- 높은 수준의 동시성 지원
- 적절한 지연 시간

🧵 개략적 규모 추정

- 5000개 호텔, 100만 개 객실
- 평균 70% 객실이 사용중, 평균 투숙 기간 3일
- 일일 예상 예약 건수 = 약 24만 건
- 초당 예약 건수(TPS) = 약 3건
- QPS 단계 별 10% 사용자만 다음 단계로 진행한다고 가정
    - 호텔/객실 상세 페이지(조회): **300**
    - 예약 상세 정보 페이지(조회): **30**
    - 객실 예약 페이지(트랜잭션): **3**

▼ 2️⃣ 개략적 설계

🎨 API 설계

호텔 관련 API

API	설명
GET /v1/hotels/:id	호텔 상세 정보 반환
POST /v1/hotels	신규 호텔 추가
PUT /v1/hotels/:id	호텔 정보 갱신
DELETE /v1/hotels/:id	호텔 정보 삭제

객실 관련 API

API	설명
GET /v1/hotels/:id/rooms/:id	객실 상세 정보 반환
POST /v1/hotels/:id/rooms	신규 객실 추가
PUT /v1/hotels/:id/rooms/:id	객실 정보 갱신
DELETE /v1/hotels/:id/roms/:id	객실 정보 삭제

예약 관련 API

API	설명
GET /v1/reservations	로그인한 사용자의 예약 이력 반환
GET /v1/reservations/:id	예약 상세 정보 반환
POST /v1/reservations	신규 예약
DELETE /v1/reservations/:id	예약 취소

🎨 데이터 모델

질의 목록

1. 호텔 상세 정보 확인
2. 지정된 날짜 범위에 사용 가능한 객실 유형 확인
3. 예약 정보 기록
4. 예약 내역 또는 과거 예약 이력 정보 조회

🎨 설계안

우리는 RDB를 선택할 것이다 ...

Why RDB?

- RDB는 읽기 빈도 > 쓰기 빈도인 작업 흐름을 잘 지원한다. NoSQL은 대체로 쓰기 연산에 최적화됨
- RDB는 ACID 속성을 보장한다.
- RDB는 데이터의 구조를 명확하게 표현하고 엔티티 간 관계를 안정적으로 지원한다.

예약 상태

결제 대기	→ 취소	
	→ 결제 완료	→ 환불 완료
	→ 승인 실패	

구성 요소

- **사용자**: 객실을 예약하는 당사자
- **관리자**: 고객 환불 / 예약 취소 / 객실 정보 갱신 등의 관리 작업을 수행하는 호텔 직원
- **CDN**: JS, image, 동영상, HTML 등 정적 컨텐츠를 캐시하여 웹사이트 로드 성능 개선
- **공개 API 게이트웨이**: 처리율 제한, 인증, 엔드포인트 기반 요청 전달 등의 기능 지원
- **내부 API**: 호텔 직원만 사용 가능한 API
- **호텔 서비스**: 호텔 / 객실 정보 제공 (정적)
- **요금 서비스**: 어떤 날에 어떤 요금을 받아야 하는지 정보 제공
- **예약 서비스**: 객실 예약 프로세스, 객실 정보 갱신 역할
- **결제 서비스**: 고객 결제 프로세스, 예약 상태 변경
- **호텔 관리 서비스**: 호텔 직원만 사용 가능한 예약 관리 기능

> 💡 예약 서비스는 총 객실 요금을 계산하기 위해 요금 서비스에 질의해야 한다. 이처럼 시스템의 서비스 간 통신에는 `gRPC`와 같은 고성능 RPC 프레임워크를 사용하기도 한다.

▼ 3️⃣ 상세 설계

데이터 모델

호텔 객실 예약은 **특정 객실**(101호)이 아닌 **특정 객실 유형**(스위트룸)을 예약함

→ roomID 관련 인자를 roomTypeId로 변경한다.

스키마

호텔 서비스

hotel
hotel_id name address location

room
room_id room_type_id floor number hotel_id name is_available

요금 서비스

room_type_rate
hotel_id **date** rate

투숙객 서비스

room_type_rate
guest_id file_name last_name email

예약 서비스

room_type_inventory
hotel_id **room_type_id** **date** total_inventory total_reserved

reservation
reservation_id hotel_id room_type_id start_date end_date status guest_id

저장 용량 추정

- 저장할 레코드 수
  - 5000개 호텔 * 20개 객실 유형 * 2년 * 365일 = **약 7300만 개**
  - 단일 서버로 처리 가능하지만 SPOF 문제를 피하기 위해 복제가 필요하다.

> 💡 **예약 데이터가 단일 데이터베이스에 담기에 너무 크다면?**
>
> 1) 과거 이력을 아카이빙하거나 냉동 저장소로 옮긴다.
> 2) 데이터베이스를 샤딩한다.
> `hash(hotel_id) % number_of_servers`

동시성 문제

이중 예약을 어떻게 방지할 것인가?

같은 사용자가 예약 버튼을 여러 번 누른다.

두 개 예약이 만들어진다.

클라이언트 측 구현

- 클라이언트에서 요청을 전송하면 예약 버튼을 비활성화한다.
  - 자바스크립트 비활성화시 우회가 가능해서 안정적이지 않다.
- 예약 요청에 멱등 키(reservation_id)를 추가하여 멱등 API로 만든다.
  - 1) 예약 주문 생성 (전역적 유일성을 보증하는 ID 생성)
  - 2) 예약 주문서 표시
  - 3) 예약 제출 (동시성 문제 발생 시 유일성 조건 위반)

여러 사용자가 같은 객실을 동시에 예약한다.

트랜잭션 격리 수준이 가장 높은 수준(serializable)이 아닌 상황이라고 가정

비관적 락

레코드를 갱신하려고 하는 순간 즉시 락을 걸어 동시 업데이트를 방지

SELECT ~ FOR UPDATE 실행 시 SELECT가 반환한 레코드에 락이 걸림 (MySQL)

- 장점
  - 변경 중이거나 변경이 끝난 데이터를 갱신하는 일을 막음

    - 구현이 쉬움
    - 데이터 충돌이 심한 경우 유용
- 단점
    - 교착 상태 발생 가능성
    - 확장성이 낮음

👍🏻 **낙관적 락**

버전 또는 타임스탬프로 레코드를 관리하여 갱신할 때마다 버전 유효성을 검사

- 장점
    - 비관적 락보다 빠름 락을 안 거니까
- 단점
    - 동시성 수준이 아주 높으면 성능이 급격히 나빠짐

데이터베이스 제약 조건

제약 조건을 걸어 레코드를 갱신할 때마다 제약 조건 검사, 낙관적 락과 유사

- 장점
    - 구현이 쉬움
- 단점
    - 동시성 수준이 아주 높으면 성능이 급격히 나빠짐
    - 제약 조건을 허용하지 않는 데이터베이스 존재

→ 예약 QPS가 일반적으로 높지 않기 때문에 호텔 예약 시스템에는 낙관적 락을 사용하도록 하자.

시스템의 규모 확장

호텔 예약 시스템이 유명한 웹과 연동되어야 한다면 QPS를 어떻게 감당할까요?

- 병목 가능 요소 이해
    - 무상태 서비스 → 서버를 추가하는 것으로 성능 문제 **해결 가능**
    - 데이터베이스 → 단순 서버 추가로 **해결 불가**

데이터베이스 샤딩

서버를 여러 대 두고 각각에 데이터 일부만 보관하자

어떻게 나눌 것인가?

대부분의 질의가 hotel_id를 필터링 조건으로 사용하므로 hotel_id를 샤딩 조건으로 사용하자

16개 샤드로 데이터베이스 부하 분산 시 QPS 30000인 경우, **각 샤드의 QPS는 1875**

캐시

과거의 데이터는 중요하지 않다

TTL

- 데이터 보관 시 낡은 데이터는 자동으로 소멸되도록 하자.
- **레디스**: TTL, LRU 캐시 교체 정책으로 메모리 최적화 가능
    - 잔여 객실 캐시 (key: hotelID_roomTypeID_날짜, value: 호텔 ID, 객실 유형 ID, 객실 수)

데이터베이스와의 일관성 유지

객실 데이터 변화를 데이터베이스에 먼저 반영하므로 캐시에 최신 데이터가 없을 가능성

데이터베이스가 최종적으로 잔여 객실 확인을 하면 문제가 되지 않는다.

MSA에서의 데이터 일관성 문제

서비스 별로 데이터베이스가 갖춰져 있는 경우 데이터 일관성 문제가 생긴다.

> 하나의 원자적 연산이 여러 데이터베이스에 걸쳐 실행되기 때문

2PC

- 2단계 커밋
- 분산 트랜잭션 환경에서 원자적 실행을 보증함
- 트랜잭션 코디네이터가 각 서버가 트랜잭션이 가능한 상황인지 응답을 받은 후 커밋
- 분산 트랜잭션을 지원하지 않는 데이터베이스(noSQL) 환경에서 사용 불가
- 트랜잭션 코디네이터 - 서버 간 응답 확인 때문에 지연 발생 가능성

사가 패턴

- 각 서비스에 국지적으로 발생하는 트랜잭션을 하나로 엮은 것

- 각각의 트랜잭션이 완료되면 다음 트랜잭션을 실행 (보통 메시지 큐 사용)
- 트랜잭션 실패 시 롤백 트랜잭션도 동일하게 수행
- **결과적 일관성** 보장

▼ 4️⃣ 마무리

1. 경쟁 조건이 발생할 수 있는 시나리오
    - 비관적 락 / 낙관적 락 / 데이터베이스 제약 조건
2. 시스템 규모 확장을 위한 전략
    - 데이터베이스 샤딩
    - 레디스 캐시
3. 데이터 일관성 문제
    - 마이크로서비스 간 데이터 불일치 해결을 위해 사용되는 메커니즘은 전체 설계를 복잡하게 만든다.

      → 복잡성이 그만한 가치가 있을까?