문제1. A고객사 시스템의 채널 확대 및 사용자 증가에 따라 발생하는 인증 및 세션 관리 문제 해결을 위한 인증 방식 개선 방안

인증 방식 개선 방안

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **분류** | **사용되는 경우** | **설명** | **장점** | **단점** |
| 멀티 팩터 인증(MFA) | 높은 보안이 요구되는 경우 | 단순한 ID/PW 인증 대신, 추가적인 인증 요소(예: SMS 코드, 이메일 인증, 생체인식)를 도입 | 보안 수준이 크게 향상되어 비인가 접근을 방지할 수 있음 | 사용자 경험이 다소 불편해질 수 있음 |
| SSO | 여러 채널과 시스템에 접근할 때 | 하나의 인증으로 여러 시스템에 접근할 수 있도록 하는 방식 | 사용자 경험을 개선하고, 비밀번호 관리의 복잡성을 줄임 | 초기 구현 및 설정에 많은 시간과 비용이 소요될 수 있음 |
| OAuth 2.0 및 OpenID Connect | 외부 인증 서버와의 연동이 필요한 경우 | 표준화된 프로토콜을 사용하여 사용자 인증 및 권한 부여를 관리 | 보안이 강화되고, 타 시스템과의 연동이 용이함 | 구현 복잡도가 높고, 외부 서비스 의존성이 발생할 수 있음 |
| 분산 세션 스토리지 | 서버 간 세션 동기화가 필요한 경우 | Redis, Memcached와 같은 분산 세션 스토리지를 활용하여 세션 데이터를 관리 | 세션 데이터를 중앙에서 관리하여 서버 간 세션 동기화 문제를 해결하고, 확장성이 높아짐 | 분산 환경에서의 설정 및 관리가 복잡할 수 있음 |
| JWT | 서버 부하를 줄이고 확장성을 높여야 하는 경우 | 클라이언트가 JWT 토큰을 통해 인증 상태를 유지 | 서버 부하를 줄이고, 확장성이 향상됨 | 토큰 탈취 시 보안 위협이 될 수 있으며, 토큰 만료 및 갱신 관리가 필요함 |

‘

## Azure 기반 OAuth, JWT 토큰, Redis를 활용한 CSRF 처리 및 이중 로그인 방지 시스템 설계서

**1. 개발 적용 부분**

* **OAuth 연동**:
  + Azure AD와 애플리케이션 간의 OAuth 2.0/OpenID Connect 연동 구현
  + authorization code 획득 및 access token, refresh token 발급 로직 구현
* **JWT 토큰 생성 및 검증**:
  + access token 기반 사용자 정보 확인 및 JWT 토큰 생성 로직 구현
  + JWT 토큰 서명 및 검증 기능 구현
  + JWT 토큰 만료 및 갱신 메커니즘 구현
* **Redis 연동**:
  + Azure Cache for Redis와 애플리케이션 연동 구현
  + JWT 토큰, CSRF 토큰, 사용자 세션 정보 저장 및 조회 로직 구현
* **CSRF 방지**:
  + CSRF 토큰 생성 및 클라이언트 쿠키 저장 로직 구현
  + 모든 요청에 대한 CSRF 토큰 검증 로직 구현
* **이중 로그인 방지**:
  + Redis 기반 사용자 세션 관리 로직 구현
  + 동시 로그인 제한 또는 기존 세션 만료 로직 구현 (애플리케이션 정책에 따라 결정)
* **Azure Key Vault 연동 (선택 사항)**:
  + JWT 토큰 서명에 사용되는 비밀 키를 Azure Key Vault에 안전하게 저장 및 관리하는 로직 구현

**2. 필요 DB 구성**

별도의 DB는 필요하지 않지만, Redis에 다음과 같은 정보를 저장합니다.

* **JWT 토큰**: 사용자 식별자를 key로 하여 JWT 토큰 저장
* **CSRF 토큰**: 사용자 식별자를 key로 하여 CSRF 토큰 저장
* **사용자 세션 정보**: 사용자 식별자를 key로 하여 로그인 시간, IP 주소 등 세션 정보 저장 (이중 로그인 방지에 활용)

**3. 시스템 아키텍처 다이어그램**

+-------------------+ +--------------+ +----------------+ +-------------+

| Client | | Azure AD | | Application | | Redis |

+-------------------+ +--------------+ +----------------+ +-------------+

| | | |

|--- OAuth 요청 ----->| | |

| |--- 인증 및 토큰 발급 -->| |

| | |--- JWT 생성 및 저장 -->|

|<--- JWT, CSRF 토큰 ---| | |

| | |--- CSRF 토큰 저장 -->|

| | | |

|--- 요청 (JWT, CSRF) -->| | |

| | |--- 토큰 검증 및 처리 -->|

| | | |

|<--- 응답 -------------| | |

| | | |

|--- 로그아웃 요청 ----->| | |

| | |--- 토큰 삭제 --------|

|<--- 응답 -------------| | |

+-------------------+ +--------------+ +----------------+ +-------------+

**4. 추가 고려 사항**

* **보안**:
  + HTTPS 사용
  + Redis 데이터 암호화
  + JWT 토큰 서명 키 안전하게 관리 (Azure Key Vault 활용)
  + 정기적인 보안 취약점 점검 및 업데이트
* **성능**:
  + Redis 캐싱 전략 최적화
  + 부하 테스트를 통한 성능 병목 지점 파악 및 개선
* **확장성**:
  + Azure App Service 자동 확장 기능 활용
  + Redis 클러스터 구성 (필요 시)
* **모니터링 및 로깅**:
  + Azure Monitor 활용하여 애플리케이션 성능 및 로그 모니터링
  + 에러 및 예외 상황 로깅 및 알림 설정

**테스트 시나리오 및 케이스**

**성공 시나리오**

1. **정상 로그인**:
   * 사용자가 OAuth 제공자를 통해 정상적으로 로그인
   * 애플리케이션은 Key Vault에서 비밀 키를 가져와 JWT 토큰 생성 및 Redis 저장
   * 클라이언트는 JWT 토큰과 CSRF 토큰을 이용하여 정상적으로 요청 및 응답
2. **JWT 토큰 갱신**:
   * JWT 토큰 만료 후, 클라이언트는 refresh token을 이용하여 새로운 access token 발급
   * 애플리케이션은 Key Vault에서 비밀 키를 가져와 새로운 JWT 토큰 생성 및 Redis 저장
   * 클라이언트는 새로운 JWT 토큰과 CSRF 토큰을 이용하여 정상적으로 요청 및 응답
3. **로그아웃**:
   * 사용자 로그아웃 시, Redis에서 해당 사용자의 JWT 토큰 및 CSRF 토큰 삭제

**실패 시나리오**

1. **잘못된 OAuth 인증 정보**:
   * 사용자가 OAuth 제공자에게 잘못된 인증 정보 제공
   * OAuth 인증 실패 및 에러 응답
2. **Key Vault 접근 권한 오류**:
   * 애플리케이션에 Key Vault 비밀 읽기 권한이 없을 경우
   * JWT 토큰 생성 또는 검증 실패 및 에러 응답
3. **잘못된 JWT 토큰**:
   * 클라이언트가 변조된 JWT 토큰 전송
   * JWT 토큰 검증 실패 및 에러 응답 (401 Unauthorized)
4. **CSRF 토큰 불일치**:
   * 클라이언트가 잘못된 CSRF 토큰 전송
   * CSRF 토큰 검증 실패 및 에러 응답 (403 Forbidden)
5. **이중 로그인 시도**:
   * 이미 로그인된 사용자가 다른 곳에서 로그인 시도
   * 기존 세션 만료 또는 새로운 세션 거부 (애플리케이션 정책에 따라 결정) 및 에러 응답

**테스트 케이스**

* **OAuth 인증**:
  + 정상적인 로그인
  + 잘못된 인증 정보 입력
  + OAuth 제공자 오류 발생
* **JWT 토큰**:
  + 정상적인 토큰 생성 및 검증
  + 만료된 토큰 갱신
  + 변조된 토큰 검증
* **CSRF 토큰**:
  + 정상적인 토큰 생성 및 검증
  + 잘못된 토큰 검증
* **이중 로그인**:
  + 동일 사용자의 다중 로그인 시도
  + 기존 세션 만료 또는 새로운 세션 거부
* **Key Vault**:
  + Key Vault 접근 권한 확인
  + Key Vault 오류 발생 시 처리

## Azure 기반 OAuth, JWT 토큰, 분산 DB 활용 CSRF 처리 및 이중 로그인 방지 시스템 설계서

**1. 개발 적용 부분**

* **OAuth 연동**:
  + Azure AD와 애플리케이션 간의 OAuth 2.0/OpenID Connect 연동 구현
  + authorization code 획득 및 access token, refresh token 발급 로직 구현
* **JWT 토큰 생성 및 검증**:
  + access token 기반 사용자 정보 확인 및 JWT 토큰 생성 로직 구현
  + JWT 토큰 서명 및 검증 기능 구현
  + JWT 토큰 만료 및 갱신 메커니즘 구현
* **분산 DB 연동**:
  + Azure Cosmos DB 또는 다른 분산 데이터베이스와 애플리케이션 연동 구현
  + JWT 토큰, CSRF 토큰, 사용자 세션 정보 저장 및 조회 로직 구현 (Redis 대체)
* **CSRF 방지**:
  + CSRF 토큰 생성 및 클라이언트 쿠키 저장 로직 구현
  + 모든 요청에 대한 CSRF 토큰 검증 로직 구현
* **이중 로그인 방지**:
  + 분산 DB 기반 사용자 세션 관리 로직 구현
  + 동시 로그인 제한 또는 기존 세션 만료 로직 구현 (애플리케이션 정책에 따라 결정)
* **Azure Key Vault 연동 (선택 사항)**:
  + JWT 토큰 서명에 사용되는 비밀 키를 Azure Key Vault에 안전하게 저장 및 관리하는 로직 구현

**2. 필요 DB 구성**

* **Azure Cosmos DB 또는 다른 분산 데이터베이스**:
  + JWT 토큰 저장: 사용자 식별자를 Partition Key로 하여 JWT 토큰 저장
  + CSRF 토큰 저장: 사용자 식별자를 Partition Key로 하여 CSRF 토큰 저장
  + 사용자 세션 정보 저장: 사용자 식별자를 Partition Key로 하여 로그인 시간, IP 주소 등 세션 정보 저장 (이중 로그인 방지에 활용)

**3. 시스템 아키텍처 다이어그램**

+-------------------+ +--------------+ +----------------+ +-----------------+

| Client | | Azure AD | | Application | | Distributed DB |

+-------------------+ +--------------+ +----------------+ +-----------------+

| | | |

|--- OAuth 요청 ----->| | |

| |--- 인증 및 토큰 발급 -->| |

| | |--- JWT 생성 및 저장 -->|

|<--- JWT, CSRF 토큰 ---| | |

| | |--- CSRF 토큰 저장 -->|

| | | |

|--- 요청 (JWT, CSRF) -->| | |

| | |--- 토큰 검증 및 처리 -->|

| | | |

|<--- 응답 -------------| | |

| | | |

|--- 로그아웃 요청 ----->| | |

| | |--- 토큰 삭제 --------|

|<--- 응답 -------------| | |

+-------------------+ +--------------+ +----------------+ +-----------------+

**Redis 성능 병목 해결**: 분산 DB를 적용하여 Redis의 성능 병목 문제를 해결하고, 글로벌 확장 시에도 안정적인 성능과 가용성을 확보할 수 있습니다.

**성공적인 시스템 구축을 위해서는 분산 DB의 특성을 충분히 이해하고, 데이터 모델링, 성능 튜닝, 글로벌 확장 등 다양한 측면을 고려하여 설계 및 개발해야 합니다.**

## Azure 기반 OAuth, JWT 토큰, 분산 DB 활용 CSRF 처리 및 이중 로그인 방지 시스템 설계서 (데이터 모델링 및 일관성 추가)

**1. 분산 DB 데이터 모델링**

* **사용자 정보 (User)**
  + id: 사용자 고유 식별자 (Partition Key)
  + username: 사용자 이름
  + email: 이메일 주소
  + other\_user\_data: 기타 사용자 정보 (필요에 따라 추가)
* **JWT 토큰 (JwtToken)**
  + id: JWT 토큰 고유 식별자
  + user\_id: 사용자 식별자 (Partition Key)
  + token: JWT 토큰 값
  + expiration\_time: 토큰 만료 시간
* **CSRF 토큰 (CsrfToken)**
  + id: CSRF 토큰 고유 식별자
  + user\_id: 사용자 식별자 (Partition Key)
  + token: CSRF 토큰 값
* **사용자 세션 (UserSession)**
  + id: 세션 고유 식별자
  + user\_id: 사용자 식별자 (Partition Key)
  + login\_time: 로그인 시간
  + ip\_address: 로그인 IP 주소
  + other\_session\_data: 기타 세션 정보 (필요에 따라 추가)

**2. 데이터 일관성**

* **강력한 일관성 (Strong Consistency)**: 모든 읽기 요청은 가장 최근에 쓰여진 데이터 반환. 이중 로그인 방지 및 중요 데이터 처리에 적합.
* **세션 일관성 (Session Consistency)**: 단일 클라이언트 세션 내에서 읽기 요청은 해당 세션에서 쓰여진 데이터 반영. 사용자 세션 관리에 적합.
* **결과적 일관성 (Eventual Consistency)**: 모든 복제본에 데이터가 결국 전파됨. 높은 가용성 필요하지만 데이터 일관성 요구 낮은 경우 적합.

**3. DB 설계 고려 사항**

* **Partition Key 선택**: 데이터 모델의 Partition Key는 데이터 분산 및 쿼리 성능에 큰 영향. 사용자 식별자 (user\_id)를 Partition Key로 사용하여 사용자 관련 데이터를 효율적으로 관리.
* **인덱싱**: 자주 사용되는 쿼리에 대한 인덱스 생성하여 쿼리 성능 향상. 예: JwtToken 컬렉션에서 expiration\_time 필드에 인덱스 생성하여 만료된 토큰 검색 효율성 증대.
* **TTL (Time-to-Live)**: JwtToken 및 CsrfToken 컬렉션에 TTL 설정하여 만료된 토큰 자동 삭제. 저장 공간 효율성 및 관리 편의성 향상.
* **데이터 모델 확장성**: 향후 추가될 수 있는 데이터 및 기능 고려하여 데이터 모델 설계. 유연한 스키마 구조를 통해 데이터 모델 변경 용이성 확보.

**분산 DB 적용 및 데이터 모델링, 일관성 고려를 통해 시스템의 성능, 확장성, 가용성을 향상시키고, 글로벌 서비스 운영에 필요한 데이터 관리 기능을 효과적으로 구현할 수 있습니다.**

A고객사 비즈니스 유연성과 성능 관점에서 상품을 관리하기 위한 데이터 모델을 새롭게 설계하고 설계 사유 제시

A고객사의 비즈니스 유연성과 성능 관점에서 상품을 관리하기 위한 데이터 모델을 새롭게 설계하기 위해서는 데이터 모델링 방법론 중에서 정규화(Normalization), 비정규화(Denormalization), 스타 스키마(Star Schema), 스노우플레이크 스키마(Snowflake Schema), EAV(Entity-Attribute-Value) 모델, 도메인 주도 설계(DDD) 중 가장 적절한 것을 선택해야 합니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **분류** | **사용되는 경우** | **설명** | **장점** | **단점** |
| **정규화(Normalization)** | 데이터 중복을 최소화하고 무결성을 유지할 때 | 데이터의 중복을 제거하고 각 데이터 요소를 논리적으로 분리하는 방법론 | 데이터 무결성 유지, 데이터 중복 최소화, 데이터베이스의 저장 공간 효율성 향상 | JOIN 연산이 많아져 쿼리 성능이 저하될 수 있음 |
| **비정규화(Denormalization)** | 쿼리 성능을 최적화하고 읽기 속도를 높일 때 | 정규화된 데이터를 다시 통합하여 중복을 허용하는 방법론 | 쿼리 성능 향상, 읽기 속도 증가, 복잡한 JOIN 연산 최소화 | 데이터 중복으로 인해 저장 공간 증가, 데이터 무결성 유지 어려움 |
| **스타 스키마(Star Schema)** | 데이터 웨어하우징과 분석을 위해 설계할 때 | 중심에 사실 테이블을 두고 주변에 차원 테이블을 배치하는 모델 | 쿼리 성능 최적화, 분석 및 보고서 작성에 용이, 단순하고 직관적인 구조 | 데이터 중복 가능성, 사실 테이블에 많은 데이터가 집중될 경우 성능 저하 가능 |
| **스노우플레이크 스키마(Snowflake Schema)** | 복잡한 데이터 분석과 다차원적 데이터를 처리할 때 | 스타 스키마를 확장하여 차원 테이블을 더 세분화한 모델 | 데이터 중복 최소화, 정규화로 인한 저장 공간 효율성 향상 | 복잡한 구조로 인해 쿼리 성능 저하 가능, 데이터베이스 설계 및 유지보수 복잡성 증가 |
| **EAV(Entity-Attribute-Value) 모델** | 유연한 속성 관리를 필요로 할 때 | 속성의 종류와 개수가 동적으로 변하는 경우 사용하는 모델 | 매우 유연한 데이터 구조, 속성 추가 및 변경이 용이 | 쿼리가 복잡하고 비효율적일 수 있음, 데이터 무결성 및 성능 문제 발생 가능 |
| **도메인 주도 설계(DDD, Domain-Driven Design)** | 복잡한 비즈니스 로직을 캡슐화할 때 | 도메인 모델을 중심으로 비즈니스 로직을 설계하는 방법론 | 복잡한 비즈니스 요구사항을 잘 반영, 유지보수 용이, 높은 재사용성 | 초기 설계가 복잡하고 비용이 많이 들 수 있음, 도메인 전문가와의 긴밀한 협업 필요 |

## A고객사 상품 관리 데이터 모델 설계 (유연성 및 성능 중심)

**1. 개요**

A고객사의 상품 관리 시스템은 다양한 상품 유형, 속성, 옵션 등을 효율적으로 관리해야 합니다. 이를 위해 유연성과 성능을 고려한 데이터 모델을 설계합니다.

**2. 핵심 엔터티 및 관계**

* **상품 (Product)**: 상품의 기본 정보를 저장합니다. (상품ID, 상품명, 상품유형, 등록일 등)
* **상품 유형 (ProductType)**: 상품 유형을 정의합니다. (상품 유형ID, 유형명 등)
* **상품 속성 (ProductAttribute)**: 상품 속성을 정의합니다. (속성ID, 속성명, 데이터 유형 등)
* **상품 속성 값 (ProductAttributeValue)**: 상품별 속성 값을 저장합니다. (상품ID, 속성ID, 속성 값)
* **상품 옵션 (ProductOption)**: 상품 옵션을 정의합니다. (옵션ID, 옵션명 등)
* **상품 옵션 값 (ProductOptionValue)**: 상품별 옵션 값을 저장합니다. (상품ID, 옵션ID, 옵션 값)
* **상품 이미지 (ProductImage)**: 상품 이미지를 저장합니다. (이미지ID, 상품ID, 이미지 URL 등)
* **상품 카테고리 (ProductCategory)**: 상품 카테고리를 정의합니다. (카테고리ID, 카테고리명 등)
* **상품-카테고리 관계 (ProductCategoryRelation)**: 상품과 카테고리의 관계를 정의합니다. (상품ID, 카테고리ID)

**3. 설계 사유**

* **유연성 확보**
  + **EAV(Entity-Attribute-Value) 모델 활용**: 상품 속성 및 옵션을 EAV 모델로 관리하여 새로운 속성이나 옵션 추가 시 데이터베이스 스키마 변경 없이 유연하게 대응 가능합니다.
  + **상품 유형**: 다양한 상품 유형을 정의하여 각 유형별 특징을 효과적으로 관리할 수 있습니다.
  + **상품-카테고리 관계**: 다대다 관계를 통해 상품을 여러 카테고리에 할당하거나 카테고리를 계층적으로 구성하여 유연한 상품 분류 체계를 구축할 수 있습니다.
* **성능 향상**
  + **인덱스**: 자주 사용되는 검색 조건에 대한 인덱스를 생성하여 조회 성능을 향상시킵니다. (예: 상품ID, 상품명, 카테고리ID 등)
  + **캐싱**: 자주 조회되는 상품 정보를 캐싱하여 데이터베이스 접근 횟수를 줄이고 응답 속도를 향상시킵니다.
  + **데이터베이스 튜닝**: 데이터베이스 설정 및 쿼리 튜닝을 통해 성능을 최적화합니다.

**4. 추가 고려 사항**

* **데이터 무결성**: 외래 키 제약 조건 등을 활용하여 데이터 무결성을 보장합니다.
* **확장성**: 향후 추가될 기능이나 데이터 증가를 고려하여 확장 가능한 구조로 설계합니다.
* **보안**: 중요 데이터에 대한 접근 제어 등 보안 조치를 적용합니다.

**5. 결론**

제안된 데이터 모델은 EAV 모델, 상품 유형, 상품-카테고리 관계 등을 활용하여 유연성과 성능을 모두 고려한 상품 관리 시스템 구축을 위한 기반을 제공합니다.

**설계 고려 사항**

* **유연성**: 다양한 상품 유형, 구독 모델, 프로모션을 지원할 수 있도록 유연한 구조로 설계합니다.
* **확장성**: 향후 새로운 기능 추가 및 데이터 증가에 대비하여 확장 가능한 구조로 설계합니다.
* **성능**: 대용량 데이터 처리 및 빠른 조회를 위해 인덱스 및 캐싱 전략을 적용합니다.
* **무결성**: 외래 키 제약 조건 등을 활용하여 데이터 무결성을 보장합니다.

**- 예약 처리 프로세스와 데이터 모델에서 발생하고 있는 동시성 이슈의 해결 방안**

다음은 예약 처리 프로세스와 데이터 모델에서 발생하는 동시성 이슈의 해결 방안을 분류, 사용되는 경우, 설명, 장점, 단점으로 정리한 표입니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **분류** | **사용되는 경우** | **설명** | **장점** | **단점** |
| **낙관적 잠금(Optimistic Locking)** | 동시 업데이트가 드물 때 | 데이터 읽을 때 버전을 기록하고 업데이트 시 버전을 비교하여 충돌을 감지 | 충돌이 거의 없을 때 성능이 우수, 잠금으로 인한 병목 현상 없음 | 충돌이 발생하면 롤백 필요, 높은 충돌 발생 시 성능 저하 |
| **비관적 잠금(Pessimistic Locking)** | 동시 업데이트가 빈번할 때 | 데이터 읽기 시 즉시 잠금 설정, 다른 트랜잭션이 접근하지 못하게 함 | 데이터 충돌 완전 방지, 일관성 유지에 유리 | 잠금으로 인한 병목 현상 발생 가능, 동시성 감소 |
| **타임스탬프 기반 접근(Timestamp-based Concurrency Control)** | 다수의 트랜잭션이 있을 때 | 각 트랜잭션에 타임스탬프를 부여하여 순서를 관리, 트랜잭션 간의 순서 충돌을 방지 | 높은 일관성 유지, 충돌 시 빠른 검출 가능 | 타임스탬프 관리의 복잡성 증가, 순서 관리 오버헤드 |
| **MVCC(Multi-Version Concurrency Control)** | 읽기 및 쓰기 작업이 많을 때 | 데이터의 여러 버전을 유지하여 읽기 작업이 쓰기 작업을 방해하지 않도록 함 | 동시성 매우 우수, 읽기 작업의 지연 없음 | 저장 공간의 증가, 버전 관리의 복잡성 증가 |
| **분산 잠금(Distributed Locking)** | 분산 시스템에서 동시성 제어할 때 | 여러 노드 간의 일관성을 유지하기 위해 분산 락 매니저를 통해 잠금 관리 | 분산 환경에서 데이터 일관성 유지, 확장성 우수 | 복잡한 설정과 관리 필요, 네트워크 지연에 따른 성능 저하 가능 |
| **애플리케이션 수준 잠금(Application-level Locking)** | 특정 비즈니스 로직에 잠금 필요할 때 | 애플리케이션 코드 내에서 동시성을 제어, 특정 리소스에 대한 잠금을 직접 구현 | 구현의 유연성, 특정 요구사항에 맞춘 잠금 제어 가능 | 구현 복잡성 증가, 개발자의 실수로 인한 잠금 문제 발생 가능 |
| **데드락 방지(Deadlock Prevention)** | 데드락 발생 가능성이 있을 때 | 데드락 발생 가능성을 사전에 방지하는 알고리즘을 사용, 예: 타임아웃 설정, 자원 요청 순서 정의 | 데드락 방지로 시스템 안정성 증가 | 성능 저하 가능, 특정 상황에서 과도한 자원 낭비 가능 |
| **잠금 시간 제한(Lock Timeout)** | 잠금으로 인한 무한 대기를 방지할 때 | 잠금 시간 제한을 설정하여 일정 시간 이후 잠금이 해제되도록 함 | 무한 대기 방지, 시스템의 유연성 증가 | 잠금 해제로 인한 데이터 일관성 문제 발생 가능 |

SAGA와 2단계 커밋(2PC)은 분산 시스템에서 트랜잭션 관리 및 동시성 제어를 위한 두 가지 주요 방법입니다. 이 둘을 설명하고, 이를 표로 정리해 보겠습니다.

### SAGA (Saga)

**SAGA**는 분산 트랜잭션 관리 기법으로, 긴 시간에 걸쳐 실행되는 분산 트랜잭션을 여러 개의 작은 트랜잭션으로 나누고, 각 트랜잭션이 성공할 때마다 다음 트랜잭션을 수행하며, 실패 시 보상 트랜잭션을 통해 이전 상태로 되돌리는 방식입니다.

### 2단계 커밋 (Two-Phase Commit, 2PC)

\*\*2단계 커밋(2PC)\*\*는 분산 트랜잭션 관리 기법으로, 트랜잭션을 참여자들 간에 일관되게 적용하기 위해 두 단계로 커밋을 수행합니다. 첫 번째 단계에서는 모든 참여자들이 준비 상태를 확인하고, 두 번째 단계에서는 모든 참여자가 준비 상태일 경우 커밋을 확정하는 방식입니다.

### SAGA와 2PC의 비교

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **분류** | **사용되는 경우** | **설명** | **장점** | **단점** |
| **SAGA** | 장시간 트랜잭션이나 높은 분산 환경 | 여러 개의 작은 트랜잭션으로 나누어 수행, 실패 시 보상 트랜잭션을 통해 이전 상태로 되돌림 | 비동기적 처리 가능, 높은 확장성, 롱 런닝 트랜잭션에 적합 | 보상 트랜잭션 필요, 복잡한 롤백 로직 구현 필요 |
| **2단계 커밋(2PC)** | 강한 일관성이 요구되는 경우 | 트랜잭션을 두 단계로 나누어 준비 상태 확인 후 커밋을 확정 | 강한 일관성 보장, 모든 참여자가 일관된 상태 유지 | 높은 지연 시간, 장애 발생 시 복구 복잡성 증가 |

### SAGA 설명

* **사용되는 경우**: 장시간 트랜잭션이나 높은 분산 환경에서 유용.
* **설명**: 분산 트랜잭션을 여러 개의 작은 트랜잭션으로 나누어 수행하고, 각 트랜잭션이 실패할 경우 보상 트랜잭션을 통해 이전 상태로 되돌리는 방식.
* **장점**: 비동기적 처리 가능, 높은 확장성, 롱 런닝 트랜잭션에 적합.
* **단점**: 보상 트랜잭션이 필요하며, 복잡한 롤백 로직 구현이 필요.

### 2단계 커밋(2PC) 설명

* **사용되는 경우**: 강한 일관성이 요구되는 경우에 유용.
* **설명**: 분산 트랜잭션을 두 단계로 나누어 수행. 첫 단계에서 모든 참여자들의 준비 상태를 확인하고, 두 번째 단계에서 준비가 완료되면 커밋을 확정.
* **장점**: 강한 일관성 보장, 모든 참여자가 일관된 상태를 유지.
* **단점**: 높은 지연 시간 발생 가능, 장애 발생 시 복구 복잡성 증가.

이 두 가지 방법론은 분산 시스템에서 트랜잭션 일관성을 유지하고 동시성 이슈를 해결하기 위한 중요한 도구입니다. SAGA는 비동기 처리와 높은 확장성을 요구하는 환경에 적합하며, 2PC는 강한 일관성이 필요한 경우에 유용합

## Meata-X 시스템 상세 설계 (빌링, 재고 관리, 동시성 이슈 처리 포함)

**1. 시스템 아키텍처**

* **프레젠테이션 계층**
  + 사용자 인터페이스 (웹, 모바일 앱)
  + 상품 정보 조회, 구독 신청/변경/해지, 결제, 마이페이지 등 사용자 기능 제공
* **애플리케이션 계층**
  + API 서버: RESTful API 제공, 사용자 요청 처리 및 데이터 액세스
  + 백그라운드 작업: 빌링 처리, 정산 처리, 재고 관리, 데이터 분석 등 주기적 또는 비동기 작업 수행
  + 외부 시스템 연동: 결제 시스템 (PG), 빌링 시스템, 재고 관리 시스템 (ERP) 등과 데이터 교환
* **데이터 계층**
  + 관계형 데이터베이스 (RDBMS): 상품, 구독, 고객, 프로모션, 위약금, 정산, 빌링, 주문, 주문상세 정보 저장
  + NoSQL 데이터베이스: 상품/구독/프로모션 변경 이력, 혜택 내용 등 유연한 데이터 저장
  + Azure Cosmos DB: 글로벌 분산 및 멀티 마스터 복제를 통한 높은 가용성 및 확장성 확보
  + Azure Cache for Redis: 자주 사용되는 데이터 캐싱 및 분산 락 기능 제공

**2. 핵심 기능 상세 설계**

* **상품 관리**
  + 상품 등록: 상품 정보 (이름, 유형, 가격 등) 및 재고 정보 등록, 상품 이미지 업로드
  + 상품 수정: 상품 정보 및 재고 정보 수정, 상품 이미지 변경
  + 상품 삭제: 상품 삭제 (단, 관련 구독 정보 존재 시 삭제 제한)
  + 상품 조회: 상품 목록 조회, 상품 상세 정보 조회, 필터링 및 검색 기능 제공
* **구독 관리**
  + 구독 신청: 상품 선택, 약정 기간 선택, 프로모션 적용, 결제 정보 입력
  + 구독 변경: 약정 기간 변경, 상품 변경 (추가/삭제), 결제 정보 변경
  + 구독 해지: 해지 신청, 위약금 계산 및 안내, 잔여 기간 서비스 제공
  + 구독 갱신: 약정 기간 만료 시 자동 갱신 또는 수동 갱신 선택
  + 구독 조회: 현재 구독 중인 상품 목록, 구독 상세 정보, 결제 내역 조회
* **프로모션 관리**
  + 프로모션 등록: 프로모션 정보 (이름, 기간, 대상 상품, 혜택 내용) 등록
  + 프로모션 수정: 프로모션 정보 수정
  + 프로모션 삭제: 프로모션 삭제 (단, 적용 중인 구독 존재 시 삭제 제한)
  + 프로모션 조회: 프로모션 목록 조회, 프로모션 상세 정보 조회
* **위약금 관리**
  + 위약금 계산: 구독 해지 시 약정 기간, 사용 기간, 잔여 기간 등을 기반으로 위약금 계산
  + 위약금 부과: 계산된 위약금 정보 저장 및 고객에게 안내
  + 위약금 조회: 위약금 내역 조회
* **정산 관리**
  + 정산 정보 관리: 구독 정보, 결제 정보, 프로모션 정보 등을 기반으로 정산 정보 생성 및 관리
  + 정산 처리: 정산 정보를 기반으로 정산 금액 계산 및 정산 상태 업데이트
  + 정산 조회: 정산 내역 조회
* **빌링 관리**
  + 청구서 생성: 구독 정보, 빌링 주기 등을 기반으로 청구서 생성
  + 결제 처리: 결제 시스템 연동하여 결제 처리, 빌링 상태 업데이트
  + 미납 관리: 미납 발생 시 알림 발송, 연체료 부과 등 미납 관리 기능 제공
* **고객 관리**
  + 고객 정보 관리: 고객 정보 등록, 수정, 삭제, 조회
  + 구독 이력 조회: 고객별 구독 이력 조회
* **재고 관리**
  + 입고 관리: 상품 입고 시 재고 정보 업데이트, 입고 내역 기록
  + 출고 관리: 상품 판매 또는 구독 시 재고 정보 업데이트, 출고 내역 기록
  + 재고 현황 조회: 상품별 재고 현황 조회, 안전 재고량 설정 및 알림 기능 제공
  + 재고 부족 알림: 안전 재고량 미만 시 관리자에게 알림 발송
* **보고 및 분석**
  + 매출 분석: 기간별, 상품별, 구독 모델별 매출 분석
  + 상품별 판매 현황: 상품별 판매량, 매출액, 인기 상품 분석
  + 고객 행동 분석: 고객별 구독 패턴, 이용 현황 분석
  + 맞춤형 보고서 생성 기능 제공

**3. 동시성 이슈 처리 상세 설계**

* **재고 초과 판매 방지**
  + **낙관적 잠금**: 상품 재고 업데이트 시 ETag 또는 타임스탬프 활용하여 충돌 감지 및 처리
  + **재고 선점**: 상품 선택 시 Redis를 활용하여 재고 선점, 결제 완료 시 실제 재고 차감, 실패 시 선점 해제
  + **Azure Cosmos DB**: 멀티 마스터 복제 및 충돌 해결 기능 활용하여 재고 데이터 일관성 유지
* **데이터 불일치 방지**
  + **데이터베이스 트랜잭션**: 여러 데이터베이스 작업을 하나의 논리적 단위로 묶어 처리, ACID 특성 보장
  + **Azure Cosmos DB**: 낙관적 동시성 제어 및 충돌 해결 기능 활용하여 데이터 일관성 유지
  + **SAGA 패턴**: 분산 트랜잭션 처리, 로컬 트랜잭션 및 보상 트랜잭션을 통해 데이터 일관성 유지

**4. 결론**

본 상세 설계는 Meata-X 시스템의 핵심 기능 및 동시성 이슈 처리 방안을 제시합니다. 실제 구현 시에는 추가적인 요구사항 및 시스템 환경을 고려하여 더욱 상세한 설계 및 구현이 필요하며, 시스템 모니터링 및 성능 개선을 위한 지속적인 노력이 필요합니다.

## Meata-X 시스템 동시성 이슈 처리 설계 개념 및 상세 설명

**1. 동시성 이슈 개념**

* **정의**: 다수의 사용자 또는 프로세스가 동시에 시스템 자원에 접근하여 작업을 수행할 때 발생하는 문제 상황
* **Meata-X 시스템 예시**:
  + **재고 초과 판매**: 여러 사용자가 동시에 같은 상품을 구매하려 할 때, 실제 재고보다 더 많은 상품이 판매되는 현상
  + **데이터 불일치**: 여러 사용자가 동시에 같은 데이터를 수정하려 할 때, 데이터의 일관성이 깨지고 잘못된 값으로 업데이트되는 현상

**2. 해결 방안 상세 설명**

* **Azure 서비스 활용**
  + **Azure Cosmos DB**
    - **글로벌 분산 및 멀티 마스터 복제**: 여러 지역에서 동시에 데이터 쓰기 가능, 충돌 감지 및 해결 기능 내장
    - **낙관적 동시성 제어**: ETag (Entity Tag)를 사용하여 데이터 버전 관리 및 충돌 감지, 충돌 발생 시 사용자에게 재시도 또는 충돌 해결 방법 제시
  + **Azure SQL Database**
    - **비관적 동시성 제어**: 잠금(Lock)을 사용하여 동시 접근 제어, 특정 데이터에 대한 접근을 제한하여 데이터 일관성 유지
    - **트랜잭션**: ACID (원자성, 일관성, 격리성, 지속성) 특성을 보장하여 데이터베이스 작업의 안정성 확보
  + **Azure Cache for Redis**
    - **분산 락**: Redlock 알고리즘을 사용하여 여러 Redis 노드 간 동시 접근 제어, 데이터 경합 방지
    - **트랜잭션**: Redis 트랜잭션 기능을 활용하여 여러 Redis 명령을 하나의 논리적 단위로 묶어 처리, 데이터 일관성 유지
* **기타 해결 방안**
  + **큐 기반 비동기 처리 (Azure Queue Storage, Azure Service Bus)**
    - 사용자 요청을 큐에 저장하고 순차적으로 처리하여 동시 접근 제어
    - 시스템 부하 분산 및 안정성 향상
  + **분산 락 (Azure Blob Lease, Azure Service Bus Topic Subscription)**
    - 여러 인스턴스 간 동시 접근 제어, 데이터 일관성 유지
  + **낙관적 동시성 제어 (ETag, 타임스탬프)**
    - 데이터 버전 관리 및 충돌 감지, 데이터베이스 솔루션과 함께 사용
* **SAGA 패턴**
  + **분산 트랜잭션 처리**: 여러 개의 로컬 트랜잭션으로 분할하여 처리, 각 로컬 트랜잭션은 독립적으로 실행
  + **보상 트랜잭션**: 로컬 트랜잭션 실패 시 이전 트랜잭션들의 결과를 원래 상태로 되돌리는 작업 수행
  + **장점**: 높은 가용성, 유연성, 확장성
  + **단점**: 복잡성, 성능 저하 가능성
  + **Azure Functions, Azure Logic Apps, Azure Durable Functions**: 로컬 트랜잭션 및 보상 트랜잭션 조율 및 관리
  + **Azure Event Grid, Azure Service Bus**: 서비스 간 이벤트 통신 및 데이터 동기화
* **2PC vs SAGA 패턴 비교**
  + **2PC**: 강한 일관성, 구현 및 관리 용이, 잠금으로 인한 성능 저하 및 코디네이터 장애 시 전체 시스템 블록 가능성
  + **SAGA**: 최종 일관성, 복잡한 보상 트랜잭션 설계, 높은 가용성 및 확장성, 여러 로컬 트랜잭션 실행으로 인한 성능 저하 가능성

**3. 결론**

Meata-X 시스템의 동시성 이슈 처리를 위해 다양한 기술 및 패턴을 적용할 수 있습니다. 각 방안은 장단점을 가지고 있으며, 시스템 요구사항, 데이터 특성, 성능 및 비용 등을 고려하여 최적의 방안을 선택해야 합니다.

* 데이터 일관성 및 무결성: 2PC, Azure SQL Database, Cosmos DB
* 높은 가용성 및 확장성: SAGA, Cosmos DB, Queue, 분산 락
* 실시간 처리 및 빠른 응답 속도: Cache for Redis

실제 시스템 구현 시에는 여러 방안을 조합하여 사용하거나, 특정 기능에 따라 다른 방안을 적용하는 등 유연하게 대응하는 것이 중요합니다. 또한, 시스템 모니터링 및 성능 분석을 통해 지속적인 개선 노력을 기울여야 합니다.