사용자 증가에 따른 인증 관리 체계의 분담을 줄이기 위한 방안을 고안하고 설계하시오
전제사항 : 각 서비스는 고객 증가 또는 인증 서비스의 장애 요인에 영향을 받지 않아야 한다

1. 분산 인증 아키텍처 설계
a) 마이크로서비스 기반 인증 서비스
독립적인 인증 서비스: 인증 서비스를 별도의 마이크로서비스로 분리하여 관리합니다. 이렇게 하면 다른 서비스와 독립적으로 확장 가능하며, 인증 서비스의 장애가 다른 서비스에 영향을 미치지 않는 다
b) OAuth2 및 OpenID Connect
OAuth2와 OpenID Connect 표준을 기반으로 인증 및 권한 부여를 관리합니다. 이는 인증 서비스의 부하를 줄이고, 사용자 인증을 보다 안전하고 효율적으로 처리할 수 있습니다.

2. 토큰 기반 인증 관리
a) JWT (JSON Web Token) 사용
JWT 기반 인증: 인증 후 발급된 JWT 토큰을 사용하여 각 서비스에 접근하도록 합니다. JWT는 서버에 상태를 저장할 필요 없이 사용자의 상태를 관리할 수 있어 서버 부하를 줄일 수 있습니다.
토큰의 수명 관리: 토큰의 만료 시간(Expiration Time)을 설정하여, 토큰이 일정 시간 후에 자동으로 만료되도록 합니다. 이로 인해 재인증 과정을 통해 보안성을 높일 수 있습니다.
b) 리프레시 토큰
리프레시 토큰 사용: 만료된 JWT 토큰을 갱신하기 위해 리프레시 토큰을 사용합니다. 이는 사용자가 장기간 로그인 상태를 유지하면서도, 보안성을 유지할 수 있도록 돕습니다.

3. 캐싱 및 로드 밸런싱
a) 인증 요청 캐싱
인증 요청의 캐싱: Redis와 같은 인메모리 데이터베이스를 사용하여 최근 인증된 사용자의 세션을 캐싱합니다. 이는 인증 서비스의 부하를 줄이고, 사용자의 인증 속도를 향상시킵니다.
b) 로드 밸런싱
로드 밸런서: 인증 요청을 여러 인증 서버에 분산시키기 위해 로드 밸런서를 도입합니다. 이는 특정 서버에 부하가 집중되는 것을 방지하고, 시스템의 안정성을 높입니다.

4. 장애에 대비한 복구 전략
a) 다중 인증 서버 구성
다중 인증 서버: 인증 서비스를 여러 서버에 분산하여 배포합니다. 특정 인증 서버에 장애가 발생하더라도 다른 서버가 이를 대체하여 서비스 연속성을 유지할 수 있습니다.
b) 자동 복구 및 장애 대응
자동 복구: 인증 서비스에 장애가 발생하면 자동으로 문제를 감지하고, 복구 작업을 수행하는 시스템을 도입합니다.
서킷 브레이커: 인증 서비스에 문제가 발생할 경우, 서킷 브레이커 패턴을 적용하여 요청을 차단하고, 백업 서비스 또는 캐시된 데이터를 사용해 서비스 중단을 최소화합니다.

5. 탈중앙화 인증 방식 도입
a) 분산 아이덴티티 (Decentralized Identity)
분산 아이덴티티 솔루션: 블록체인 기술을 활용한 분산 아이덴티티 시스템을 도입하여 사용자 인증 데이터를 중앙 서버에 의존하지 않고 분산하여 관리합니다.
이 방식은 특정 서버에 인증 요청이 집중되는 것을 방지하고, 보안성을 향상시킵니다.
b) 페더레이션 인증
페더레이션 인증 (Federated Authentication): 사용자가 한 번 인증되면, 여러 서비스에 접근할 수 있도록 SSO(Single Sign-On)를 구현합니다. 이 방식은 인증 횟수를 줄이고, 사용자 경험을 개선

6. 모니터링 및 실시간 대응
a) 실시간 모니터링
모니터링 시스템: Prometheus, Grafana 등을 사용하여 인증 서비스의 상태를 실시간으로 모니터링하고, 이상 상황 발생 시 알림을 통해 빠르게 대응할 수 있도록 합니다.
b) 알림 및 자동 대응 시스템
자동 대응 시스템: 장애가 발생했을 때 자동으로 장애를 처리하고, 복구하는 시스템을 도입합니다. 예를 들어, AWS Lambda 등을 활용해 특정 조건이 발생하면 자동으로 새로운 인증 서버를 생성하도록 구성할 수 있습니다.

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

OAuth 2.0 및 OIDC(OpenID Connect) 프로토콜

이제는 너무나도 익숙한 소셜 로그인은 유저와 기업 모두에게 매력적인 인증 방법입니다. 유저는 간편하게 로그인할 수 있고 기업은 신규 유저의 가입 장벽을 낮추고 신뢰성 있는 타기업에게 인증의 책임을 미룰 수 있죠. 소셜 로그인 구현을 위해 가장 많이 쓰이고 있는 프로토콜은 OAuth 2.0과 OIDC가 있습니다.
이 두 프로토콜이 어떻게 인증 및 인가를 부여하는지 알아봅시다.

■ OAuth 2.0

위임 권한부여를 위한 표준 프로토콜인 OAuth는 사용자가 비밀번호를 제공하지 않고 서드파티 어플리케이션에게 접근 권한을 부여할 수 있게 해줍니다.
2010년 IETF에서 OAuth 1.0 공식 표준안이 RFC 5849로 발표되었으며, 현재는 OAuth 2.0 (RFC 6749, RFC 6750)이 많이 쓰이고 있습니다.

※ 위임 권한부여 (Delegated Authorization)
. 서드파티 어플리케이션이 사용자의 데이터에 접근하도록 허가해 주는 것.
. 서드파티에게 아이디/비밀번호를 주기보다는 주로 OAuth를 통해 위임 권한부여를 함.

■ 용어 정리

OAuth 2.0 의 로직 흐름을 이해하기 위해 몇 가지 용어를 알아야 합니다.

1. Client: 사용자의 데이터에 접근하고 싶어 하는 어플리케이션.
2. Resource Owner: 클라이언트 어플리케이션이 접근하길 원하는 데이터의 사용자 또는 소유자.
3. Resource Server: 클라이언트 어플리케이션이 접근하길 원하는 데이터를 저장하고 있는 서버.
4. Authorization Server: 사용자로부터 권한을 부여받아 클라이언트가 사용자의 데이터에 접근할 권한을 부여해 주는 권한부여 서버.
5. Access Token: 리소스 서버의 사용자 데이터에 접근하기 위해 클라이언트가 사용할 수 있는 유일한 키

■ OAuth 2.0 요청을 위한 파라미터

Client가 Authorization Server에 요청을 보낼 때 주로 다음과 같은 설정값을 Query String을 통해 전달합니다.

1. response_type: Authorization Server로부터 받길 원하는 응답의 타입 (code, token 등).
2. scope: Client가 Resource Server에서 접근하고 싶은 리소스 리스트
3. client_id: OAuth 세팅을 할 때, Authorization Server에 의해 제공. Client가 누구인지 알아내기 위해 사용.
4. client_secret: Authorization Server에 의해 제공. Code와 client_secret을 가지고 Access Token 받게 됨.
5. redirect_url: Authorization Server에게 OAuth 플로우가 끝나면 어디로 보내줄지에 대해 알려주는 역할

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

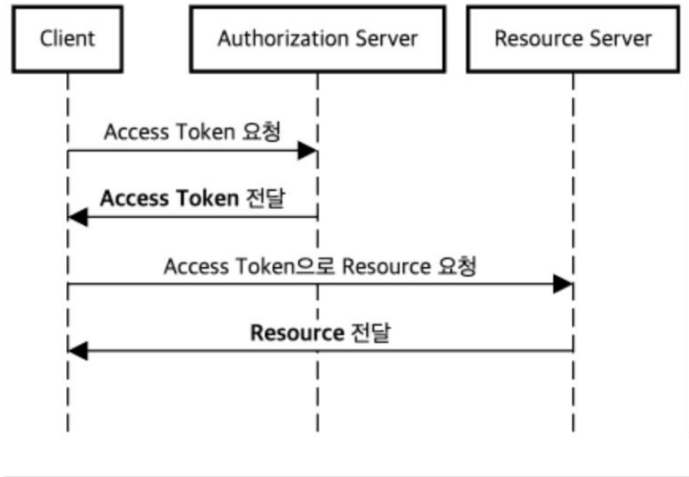
■ 인증방식과 동작 흐름

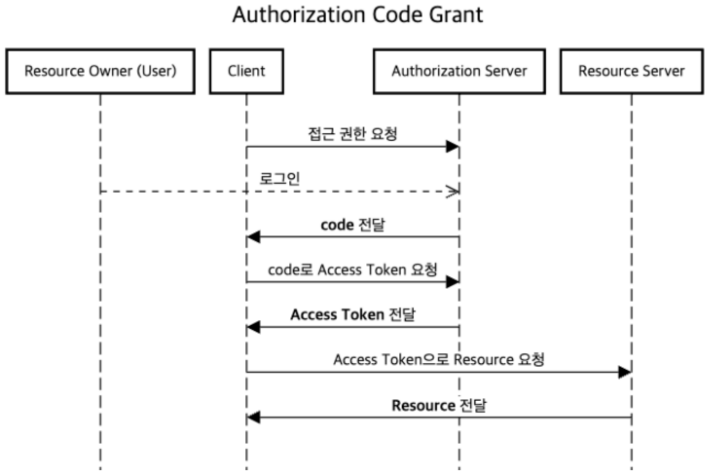
1. 권한 부여 코드 승인 방식 (Authorization Code Grant)
자체 생성한 Authorization Code를 전달하는 방식으로 OAuth2.0에서 가장 기본이 되는 방식.
response_type=code, grant_type=authoration_code 등 형식으로 요청.
Authoration Server가 Redirect 시 엑세스 토큰을 전달하면 브라우저에 토큰이 바로 노출되기 때문에
프론트앤드에서 code를 받아서 서버로 전달하면 서버에서 엑세스 토큰을 요청하는 방식.
Code를 Access Token으로 변환할 때 client_secret이 필요. 결국, 백엔드 서버에서만 필요.
백엔드 사이에서만 토큰이 이동하여 비교적 안전한 방식

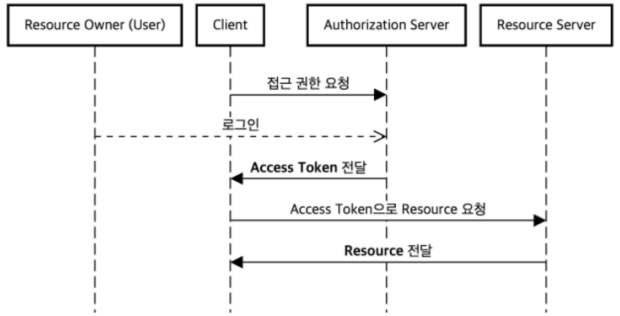
2. 암묵적 승인 방식 (Implicit Grant)
자격 증명을 안전하게 저장하기 힘든 클라이언트 사이드에서 OAuth2.0 인증에 최적화된 방식.
response_type=token 등 형식으로 요청.
Autorization Code 발급 없이 바로 Access Token 발급되기 때문에 만료 기간이 짧아야 함.
절차가 비교적 간단하지만 Access Token이 URI를 통해 전달되어 보안에 취약

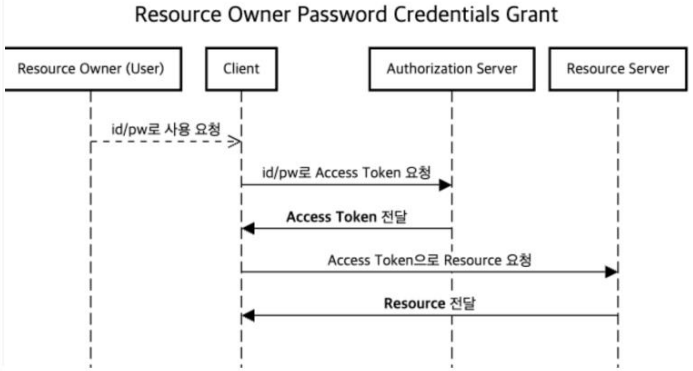
3. 자원 소유자 자격 증명 방식 (Resource Owner Password Credentials Grant)
Authorization Server, Resource Server, Client가 모두 같은 시스템에 속해 있을 때만 사용 가능.
ID, Password로만 Access Token을 발급받는 방식.
grant_type=password 형식으로 요청

4.클라이언트 자격 증명 방식 (Client Credentials Grant)
클라이언트의 자격 증명만으로 Access Token을 획득하는 방식.
User가 아닌 Client에 대한 인가가 필요할 때 사용.
즉, Client에 대해 리소스 접근 권한이 설정된 경우 사용.
자격 증명을 안전하게 보관할 수 있는 클라이언트에서만 사용.

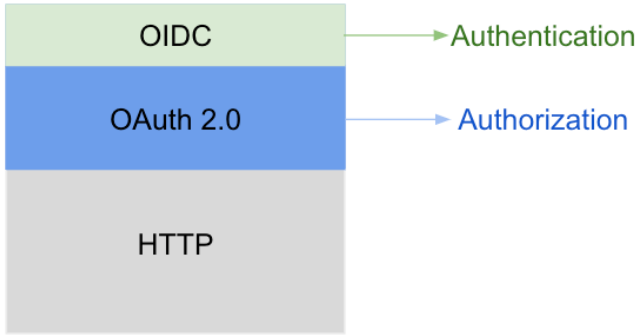








MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)



■ OpenID Connect 프로토콜

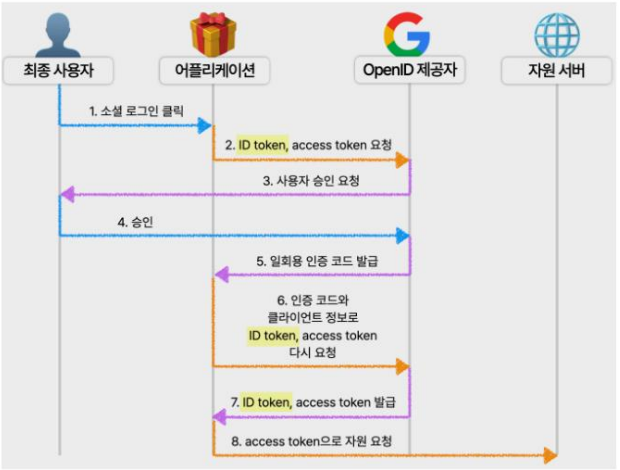
OIDC는 OAuth 2.0을 기반으로 만들어진 유저의 인증(Authentication)을 위한 프로토콜 입니다.
OIDC는 OAuth 2.0을 확장하여 인증 방식을 표준화 합니다.
OpenID를 관리하는 OpenID Foundation에서 정의한 OpenID의 개념은 다음과 같습니다.

OpenID Connect 1.0은 OAuth 2.0 프로토콜 위에서 동작하는 간단한 ID 레이어입니다.
이를 통해 클라이언트는 인증 서버에서 수행한 인증을 기반으로 최종 사용자의 신원을 확인할 수 있을 뿐만
아니라, 최종 사용자에 대한 기본 프로필 정보를 상호 운용 가능하며 REST와 유사한 방식으로 얻을 수 있습니다.
OpenID Connect를 사용하면 웹 기반, 모바일, 자바스크립트 클라이언트 등을 포함한 모든 유형의 클라이언트에서
인증 세션과 최종 사용자에 대한 정보를 요청하고 받을 수 있습니다. 스펙은 확장 가능하므로 필요에 따라 참가자들에게 ID 데이터 암호화, OpenID 제공자 확인, 로그아웃
등을 이용할 수 있습니다.

■ 인증 방식과 동작 흐름

기존 Oauth 2.0의 동작 흐름과 거의 유사하며 ID token을 추가 발급한다는 차이점이 있습니다.
추가로, OpenID 문서를 읽다 보면 IDP, RP라는 용어가 등장하는데 각각 다음과 같습니다.

- IDP (IDentity Provider): Google, Apple 같은 간편 로그인 서비스 제공사 (OpenID 제공자)
- RP (Relying Party): 사용자를 인증하기 위해 IDP에 의존하는 주체 (어플리케이션)



■ OAuth 2.0 과 OIDC의 차이점

1. 스코프 (Scopes)

OAuth 2.0에서는 제공자가 원하는 대로 요청 범위를 설정 가능하여 유연한 사용이 가능했지만 상호 운용에
취약했습니다. OIDC는 요청범위를 profile, email, address, phone으로 표준화했습니다.

2. 클레임 (claims)

OIDC에서 ID토큰의 payload는 claims라고 알려진 필드들을 포함합니다. OIDC는 이런 클레임들을 표준화했습니다.
1. iss: 토큰 발행자, 2. sub: 사용자의 유니크한 식별자, 3. email: 사용자 이메일, 4.iat: 토큰 발행 시간 (Unix time), 5.exp: 토

3. 사용자 정보 요청 엔드포인트 통일

OIDC는 사용자가 요청하는 엔드포인트도 표준화했습니다. 예를 들어 /userinfo를 통해 사용자 메타데이터 정보를 검증합니다.

4. ID token

OIDC의 동작 흐름에서 가장 눈에 띄는 차이가 ID token의 유무입니다. ID token은 JWT로 생성이 되어 Payload 내부에 클레임을 포함합니다.
즉, ID token을 복호화하여 사용자 정보를 얻을 수 있습니다. OAuth 2.0에서 액세스 토큰을 얻고 다시 사용자 정보를 요청하는 것보다 네트워크 통신 비용이 절감됩니다.

※ OAuth2: 인가 프로토콜로서, 인증된 사용자에게 자원 접근 권한을 부여하는 것에 중점을 둡니다.
※ OIDC: OAuth 2.0을 확장하여, 사용자의 인증 정보를 안전하게 전달하는 인증 프로토콜입니다.

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

Stateless 한 프로토콜 : HTTP

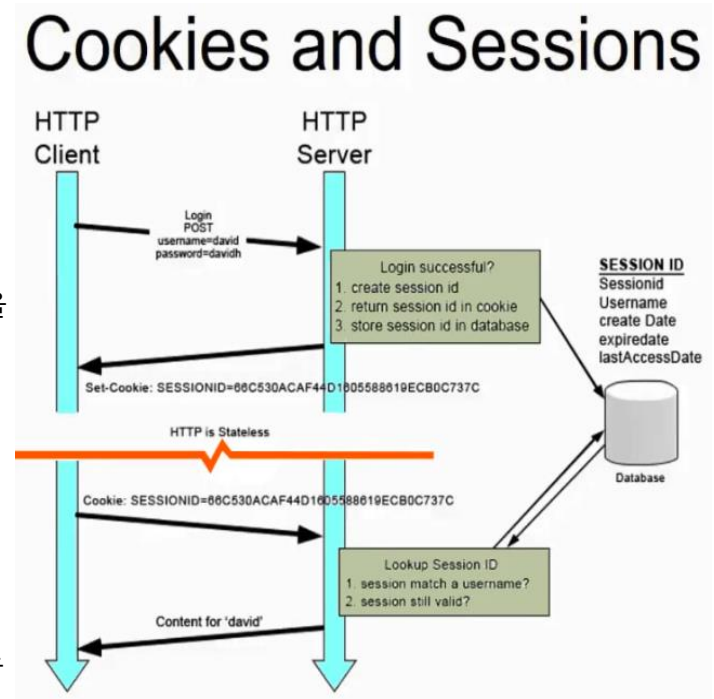
우선 HTTP의 프로토콜 상태에 알아보자. HTTP 는 stateless 한 특성 때문에 각 통신의 상태는 저장되지 않는다. 하지만 서비스에서는 어떤 유저가 기능을 사용하는지 특정할 수 있어야 하는데 이를 위해서 세션(Session) 혹은 토큰(Token)이 사용된다.
유저가 로그 인을 시도할 때 서버 상에서 일치하는 유저 정보를 찾았다면 인증 확인의 표시로 세션이나 토큰을 발급해준다.
세션과 토큰의 가장 큰 차이점은 세션은 서버에 저장된다는 것이고, 토큰은 클라이언트 측에서만 저장된다는 것이다.

쿠키(Cookie) + 세션(Session)

쿠키에 ID, PW와 같은 중요한 정보가 아닌, 인증을 위한 별개의 정보를 세션 저장소에 저장하고, 클라이언트는 세션을 쿠키에 담아 서버에 요청한다.
서버는 세션 저장소에 있는 세션과 일치하는지 즉, 유효한 세션인지 확인 후 적절한 응답을 보내준다.

■ 동작과정

1. 클라이언트가 ID/PW로 서버에 로그인
2. ID/PW로 인증 후, 사용자를 식별한 특정 유니크한 세션 ID를 만들어 마치 자물쇠처럼 서버의 세션 저장소에 저장
3. 세션 ID를 특정한 형태로 클라이언트에 다시 반환
4. 이후 사용자 인증이 필요한 정보를 요청할 때마다 세션 ID를 쿠키에 담아 서버에 함께 전달
5. 인증이 필요한 API일 경우, 서버는 세션 ID가 세션 저장소에 저장된 것인지 즉 유효한 세션인지 확인
   유효한 세션이라면, 인증 완료 후 적절한 응답을 보내준다. 없다면 401 에러 반환



■ 문제점

1. 세션 ID, Cookie 등이 탈취된다면 세션 저장소를 전부 지워 해결 가능하지만, 탈취당하지 않은 정상적인 사용자도 모두 재 인증을 해야하는 상황이 발생한다.
2. 무엇보다 HTTP의 가장 큰 특성 중 하나인 stateless한 특성을 위배하게 된다. stateless 특성은 서버에서는 클라이언트의 상태를 저장하지 않아야 하지만 세션 저장소라는 곳에서 클라이언트의 상태를 저장하게 되므로 stateful 한 상태가 된다.
   - 위의 내용이 문제가 되는 이유는 확장성에 있다. 1번 서버에서 로그인한 사용자가 다른 2번 서버로 요청하게 되면 2번 서버에서는 세션이 저장되어 있지 않아 유효하지 않은 세션으로 인식된다는 것이다.
   - 이런 문제를 해결하기 위해 세션 저장소를 별도로 외부에 두는 것이 가장 일반적인 방식이다. Redis가 세션 저장소로 사용

■ 다중 서버환경에서 세션 불일치 문제점 해결 방안

1. 세션 클러스터링 (Session Clustering)
   세션 클러스터링으로 서버 간 로그인 정보가 담긴 세션을 공유하는 방법이 있지만, 실제 서비스와 관련없는 인프라적인 작업으로 서버 리소스를 많이 쓰게되는 단점이 있다.
   전체적인 서버 규모가 크지 않다면 나쁘지 않지만, MSA로 잘게 쪼개져 수십 수백개의 서버로 이루어진다면 단점이 극명하게 나타날 것이다.
   세션 클러스터링에는 방법이 여러 가지다. "WAS 구성", "외부에 세션 서버 구축",

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ 스티키 세션 (Sticky Session)

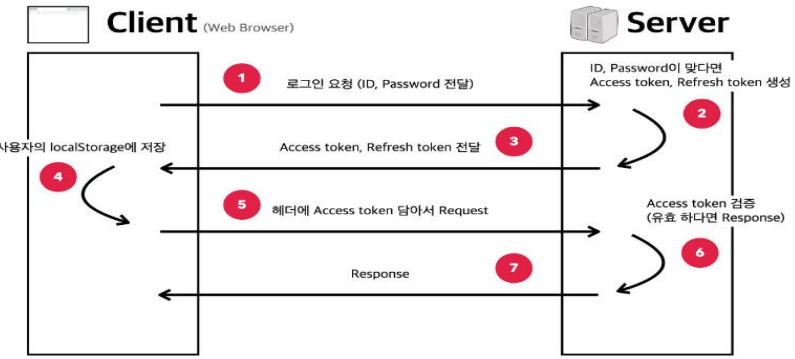
스케일 아웃 시 여러 서버에 세션 정보를 복사할 필요 없도록 특정 세션을 처음 처리한 서버에게 이후 같은 세션의 요청을 같은 서버가 처리하도록 하는 방식이다.
A 사용자가 A 서버에게 요청했다면, A사용자의 요청은 모두 A서버에서 처리하는 방식이다. 이 방법의 문제점은 각 서버가 균일하게 요청을 처리할 수 없다는 점에 있다.
즉, 특정 서버에만 요청이 몰리는 상황이 발생할 수 있다. 부하가 균일하게 분산되지 않는다.

■ 세션 스토리지 분리

이 방식은 세션 스토리지를 외부 서버로 분리하는 방식이다. 이 때 사용되는 세션 스토리지 서버로 일반적인 Disk-Based DB (Mysql, PostgreSQL, MongoDB 등)을 사용할 수 있지만, 입출력이 잦은 세션 특성 상 I/O 성능이 느린 데이터베이스는 사용하기에 적합하지 않다. 따라서 세션을 저장하는 저장소로는 In-Memory DB를 사용하는 것이 일반적이다. In-Memory DB 중 어떤 DBMS 를 사용하는 것이 좋을까? 세션 데이터는 Key-Value로 구성되어 있다.
따라서 세션을 저장할 때는 대표적인 Key-Value DB인 Redis와 Memcached 를 사용한다.

토큰 JWT(Json web token)

JWT 토큰 방식은 웹표준(RFC 7519)로서 두 개체에서 JSON 객체를 사용하여 가볍고 자가수용적인(self-contained) 방식으로 정보를 안정성 있게 전달한다. (자가수용적이라는 의미는 JWT 안에 인증에 필요한 모든 정보를 자체적으로 지니고 있다



■ 동작과정

1. 사용자는 클라이언트에서 ID/PW를 통해 로그인을 요청한다.
2. 유효한 ID/PW라면, Access token & Refresh token을 발급한다.
3. 클라이언트는 전달 받은 토큰들은 localStorage에 저장한다.
4. 클라이언트는 헤더에 Access token을 담아 서버에 요청한다.
5. 서버에서는 Access token을 검증하고, 응답을 클라이언트로 보낸다.
   - Access token이 유효하지 않다면 Refersh token으로 Access token을 재발급한 뒤, access token을 리턴해준다.

■ JWT 구조

1. 헤더 (Header) : 헤더는 두 가지 정보를 가진다. "typ - 토큰의 타입(JWT)" , "alg - 해싱 알고리즘" (Signature 를 해싱하기 위한 알고리즘 지정)
2. 내용 (Payload) : Payload에는 토큰에 담을 정보들이 존재하고, 여기에 담는 정보의 한 조각을 클레임(claim) 이라고 한다.
   - 클레임은 키 값 형태로 존재한다. 클레임의 종류는 등록된(registered) 클레임, 공개(public) 클레임, 비공개(private) 클레임들이 있다.
3. 서명 (Signature) : Signature 이란 토큰을 인코딩 하거나 유효성 검증을 할 때 사용하는 고유한 암호화 코드이다.
   - 서명 생성 과정
     . 헤더와 페이로드 값을 각각 BASE64 로 인코딩
     . 위에서 인코딩한 값을 비밀 키를 이용해 헤더에서 정의한 알고리즘으로 해싱
     . 위에서 해싱한 값을 다시 BASE64 로 인코딩

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ **JWT 의 장점**

1. 인증에 필요한 정보가 토큰에 있기에 별도의 저장소가 필요 없다.
   - 하지만, 보안성을 높이기 위해 Refresh Token을 사용하는 경우 별도의 저장소에 저장하면서 사용하는 경우도 있긴 하다.
2. Cookie와 Session 사용 시 문제점이었던 stateful 한 특성을 JWT 토큰 사용 시에는 stateless 하게 가져갈 수 있다. 즉, 서버는 클라이언트의 상태를 가질 필요가 없다.
3. HTTP 헤더에 넣어서 쉽게 전달 가능하다.
4. 확장성에 용이하다. MSA 환경에 적용하기 편하다.

■ JWT의 단점

1. 거의 모든 요청에 토큰이 포함되므로 트래픽 크기에 영향을 미칠 수 있다.
2. 토큰에 정보가 많아져 토큰의 크기가 커지면 네트워크에 부하를 줄 수 있다.
3. 페이로드는 암호화된 게 아니라 BASE64 로 인코딩된 것이므로, 중간에 토큰을 탈취하면 페이로드의 데이터를 모두 볼 수 있다.
4. 따라서 페이로드에는 중요 정보를 담아선 안된다.

■ JWT의 암호화 방식

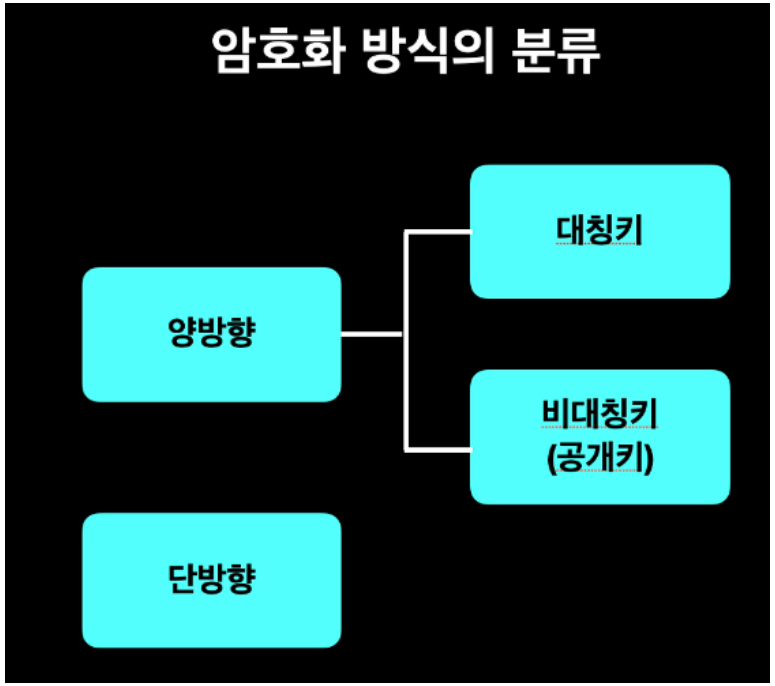
JWT 토큰 생성 시, JWT 헤더와 페이로드 정보를 인코딩하고, 둘을 합친 문자열을 비밀 키로 서명한다.
이 때 대칭키 암호화, 비대칭키 암호화 방식을 사용할 수 있다.

대칭키 암호화 - 암호화, 복호화 키가 같으면 대칭키 암호화 방식이라고 한다.

. 같은 키를 사용해 암호화, 복호화를 수행하기 때문에 속도가 빠르다.
. 대표적으로 HMAC 암호화 알고리즘이 있다.
. HS256, HS384, HS512 .... 가 이에 해당하고, 뒤 숫자는 secret key의 최소 바이트 크기를 의미한다.
. 기본적으로 단방향 암호화 알고리즘인 SHA-256 과 함께 쓰인다.
. 값에 SHA-256를 적용해서 해싱 후 private key(== secret key , 대칭키 역할)로 암호화 한다.
. private key를 알고있는 서버만 Signature 유효성 검증이 가능하다. 즉 JWT를 복호화 할 수 있다.

비대칭키 암호화 - 암호화, 복호화 키가 다르면 비대칭키 암호화 방식이라고 한다.

. 다른 키를 사용해 암호화, 복호화를 수행하기 때문에 속도가 느리지만, 대칭키 암호화에 비해 안전하다.
. 대표적으로 RSA 암호화 알고리즘이 있다.
. 마찬가지로, SHA-256 단방향 암호화 알고리즘과 함께 쓰인다.
. 값에 SHA-256 을 적용해서 해싱 후 비밀키(private key)로 암호화한다.
  그리고 공개키(public key) 는 공개적으로 제공한다. 어떠한 서버든 이 공개키를 통해 JWT를 복호화할 수 있다.



MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

대칭키 암호화 vs 비대칭키 암호화

대칭키 암호화 방식 같은 경우, private key를 모르는 서버는 JWT의 유효성을 검증할 수 없다.
반대로 비대칭키 암호화 방식은 private key를 몰라도 public key를 통해 복화 할 수 있기 때문에 JWT의 유효성을 검증할 수 있다.

대칭키 암호화 방식에서의 인증서버 구축하기

1. 인증 서버가 클라이언트에게 JWT를 발급
2. 클라이언트는 JWT와 함께 애플리케이션 서버에 요청
3. 애플리케이션 서버는 인증 서버의 private key를 모르므로 JWT를 검증할 수 없음
   각 애플리케이션 서버에 인증서버의 private key를 넣어놓으면 되긴 한다.
   하지만 MSA 환경에서 수많은 애플리케이션 서버가 존재하는데, scale-out 할때마다 매번 private key를 넣어줘야 한다.

비대칭키 암호화 방식에서의 인증서버 구축하기

1. 인증 서버가 클라이언트에게 JWT를 발급
2. 클라이언트는 JWT와 함께 애플리케이션 서버에 요청
3. 애플리케이션 서버는 인증 서버의 public Key를 통해 JWT를 검증할 수 있음
   각 애플리케이션 서버에 일일히 key를 넣어줄 필요가 없다. public key가 공개되어 있기 때문이다.

API Gateway가 존재한다면?

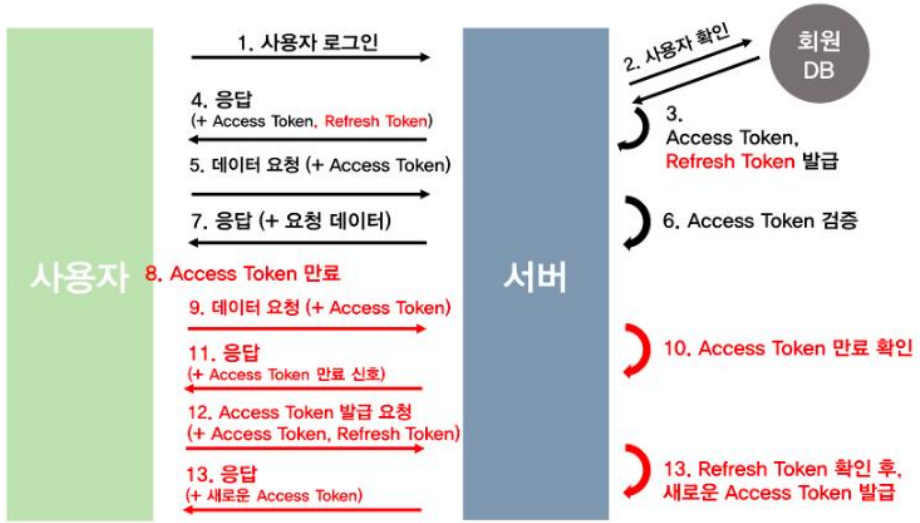
비대칭키 암호화 방식을 사용하면 매번 각 서버에서 필터나 인터셉터를 통해 JWT에 대한 검증을 수행할 것이다
하지만 API Gateway가 존재하면 API GW에서만 검증하면 된다.
API GW에서 public key를 통해 검증해도 되지만, 대칭키 방식을 사용해도 API GW에만 private key를 넣어주면 되므로 대칭키 방식의 문제점도 딱히 드러나지 않는다.

간단히 결론을 내려보면 **API GW가 없다면 비대칭키 암호화 방식을 사용하는게 좋고, API GW가 존재한다면 어떤 방식을 쓰든 상관없을** 것 같다는 생각이 든다.

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ **Refresh Token**

Refresh Token은 토큰이 탈취당할 경우를 대비해 사용되는 것이다. Access Token 만으로 공격자가 요청하는 것인지 정상적인 클라이언트가 요청하는 것인지 알 수 없기 때문이다. Access Token은 언제든지 탈취될 수 있다고 가정하기 때문에 Access Token에는 중요한 정보를 담으면 안된다. 따라서 Access Token의 유효기간을 짧게 설정하고, Refresh Token의 유효기간을 길게 설정한다. 물론 Access Token의 유효기간 동안에는 공격에 노출되어 있지만, 피해를 최소화하기 위한 방법이다.



■ **동작과정**

1. Access Token이 탈취됐을 때 대비를 위해 Refresh Token 개념을 도입했다. 그런데 Access Token과 Refresh Token 모두 클라이언트에 저장되면 같이 탈취되는거 아닌가? 라는 생각이든다.
2. 그래서 Access Token을 로컬 스토리지 또는 세션 스토리지에 저장하고, Refresh Token은 쿠키에 저장하고 보안 옵션들(HTTP Only, Secure Cookies)을 활성화 한다.
3. 물론 Refresh Token은 서버에도 저장돼있어야 한다.

■ **Refresh Token만 탈취되면?**

1. 공격자는 탈취한 Refresh Token 으로 계속 Access Token을 생성해서 정상적인 사용자처럼 서버에 계속 요청할 수 있다.
2. 이를 대비해서 서버에서 추가 검증 로직으로 방어해야 한다.
   - DB에 사용자와 Access Token, Refresh Token 들을 매핑하여 저장한다.
   - 정상적인 유저의 Access Token이 만료된 경우
   - Access Token과 Refresh Token을 서버로 보내서 새 Access Token을 요청한다 → 서버에서는 DB에 저장된Access Token, Refresh Token쌍과 클라이언트에서 보낸 토큰 쌍들을 비교한다 → 일치하면 새 Access Token토큰을 발급해준다.
   - 공격자가 Refresh Toekn을 탈취한 경우
     . 공격자가 탈취한 Refresh Token으로 새 Access Token 생성 요청 → Access Token이 없이 요청하면 공격으로 간주 → 서버에서 Access Token , Refresh Token 폐기

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ JWT 와 Session 방식 비교

1. 사이즈

세션의 경우, Cookie 헤더에 세션 ID만 실어 보내면 되므로, 트래픽을 적게 사용한다. 하지만, JWT는 사용자 인증 정보와 토큰의 발급시각, 만료시각, 토큰의 ID 등 담겨 있는 정보가 세션 ID에 비해 비대하므로 세션 방식보다 훨씬 더 많은 네트워크 트래픽을 사용한다.
그에 비해 세션 ID는 단 6바이트. 50배가 넘는 트래픽 비효율이다.

2. 안정성과 보안성

. 세션의 경우, 모든 인증 정보를 서버에서 관리하기 때문에 보안 측면에서 조금 더 유리하다. 설령 세션 ID가 해커에게 탈취된다고 하더라도, 서버 측에서 해당 세션을 무효 처리하면 된다.
. 토큰의 경우는 그렇지 않다. 토큰은 서버가 트래킹하지 않고, 클라이언트가 모든 인증정보를 가지고 있다. 따라서 토큰이 한 번 해커에게 탈취당하면 세션과 비교했을 때 조금 복잡한 방식(위의 내용 참고)으로 해킹을 막아야한다.
. 또한 JWT 특성 상 토큰에 실린 Payload가 별도로 암호화 되어있지 않으므로, 누구나 내용을 확인할 수 있다. 따라서 Payload에는 민감한 데이터를 실을 수 없다.
. 하지만 세션과 같은 경우에는 모든 데이터가 서버에 저장되기 때문에 아무나 함부로 열람할 수 없기에 저장할 수 있는 데이터에 제한이 없다.

3. 확장성

. 그럼에도 불구하고 최근 모던 웹 어플리케이션이 토큰 기반 인증을 사용하는 이유가 바로 이 확장성이다.
. 일반적으로 웹 어플리케이션의 서버 확장 방식은 수평 확장을 사용한다. 즉, 한 대가 아닌 여러 대의 서버가 요청을 처리하게 된다.
이때 별도의 작업을 해주지 않는다면, 세션 기반 인증 방식은 세션 불일치 문제를 겪게 된다.
이를 해결하기 위해서 Sticky Session, Session Clustering, 세션 스토리지 외부 분리 등의 작업을 해주어야 한다.
. 하지만, 토큰 기반 인증 방식의 경우 서버가 직접 인증 방식을 저장하지 않고, 클라이언트가 저장하는 방식을 취하기 때문에
이런 세션 불일치 문제로부터 자유롭다. 이런 특징으로 토큰 기반 인증 방식은 HTTP의 비상태성(Stateless)를 그대로 활용할 수 있고, 따라서 높은 확장성을 가질 수 있다.

4. 서버 부담

. 확장성과 어느 정도 이어지는 내용이다. 세션 기반 인증 방식은 서비스가 세션 데이터를 직접 저장하고, 관리한다. 따라서 세션 데이터의 양이 많아지면 많아질수록 서버의 부담이 증가할 것 이다.
. 하지만 토큰 기반 인증 방식은 서버 대신, 클라이언트가 인증 데이터를 직접 가지고 있다. 따라서 유저의 수가 얼마나 되던 서버의 부담이 증가하지 않는다.
따라서 서버의 부담 측면에서는 세션 기반 인증 방식보다는 토큰 기반 인증 방식이 조금 더 유리함을 알 수 있다.

4. Session 과 Token 의 차이점 정리

stateful과 stateless 하다는 측면에서 차이점이 존재한다. 이는 토큰이 탈취됐을 때, 서버에서 능동적으로 이에 대응하여 토큰을 폐기 처리할 수 있냐 없느냐에 따라 직결
Session 방식은 로그인한 유저의 세션 개수를 제한할 수 있다는 점에서도 차이가 있다.
JWT의 등장 배경을 살펴보면, 보안이 뛰어나서가 아니라, 마이크로 서비스 아키텍처(MSA)가 도입되면서 주목 받기 시작한 방식이다.
아래 사진처럼 수천 수만가지의 서버 to 서버 통신이 이루어지는 아키텍처에서 중앙화된 사용자 식별 저장소를 통해 각 API 요청을 인증처리 해야한다면.. 인증 서버만 수백대가 필요할 것이다. 그렇다고 아무리 내부 서버 끼리의 통신이라고 인증을 제외할 순 없으니 JWT를 통해 인증을 진행하는 것이다.
추가로 앱과 웹을 모두 서비스하는 서버인 경우 웹에서는 Session을 이용하고 앱에서는 토큰을 이용하는 방식으로 별개의 인증방식을 가져가는게 아니라,
두 환경 모두 토큰을 기반으로 인증하여 환경에 구애받지 않고 동일한 API를 이용할 수 있다.

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ 토근 기반 인증 절차

1. 서비스 사용자는 자신의 신원을 증명하기 위해 인증 서비스로부터 토큰을 발급 받는 다.
2. 사용자는 발급받은 토큰을 서비스 요청 명령에 첨부하여 서비스에게 전달 한다.
3. 명령을 수신한 서비스는 첨부된 토큰을 확인하여 인증 절차를 수행하고 사용자인증정보 확인

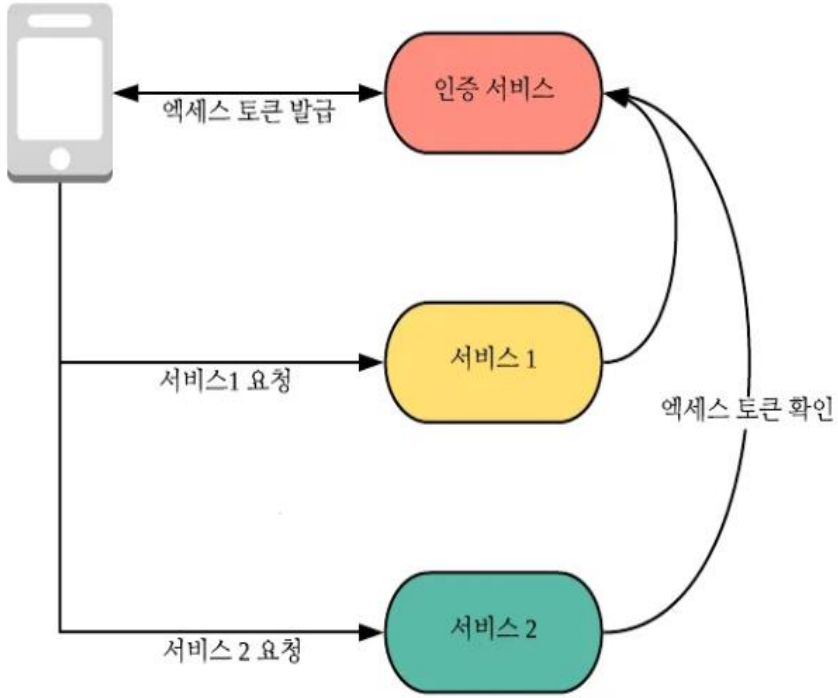
■ 엑세스(Access Token) 인증

엑세스 토큰은 사용자를 특정하기 위해 인증 서비스가 랜덤하게 생성된 고유 식별자로,
엑세스 토큰 자체로는 어떠한 정보도 담고 있지 않다.
그러나 인증 서비스는 토큰 발급 시 생성된 엑세스 토큰과 사용자 인증 정보를
매핑하여 저장함으로써 엑세스 토큰을 key 로 하여 사용자의 인증 정보를 확인 할 수 있다

엑세스 토큰 인증 절차

1. 사용자는 엑세스 토큰을 첨부하여 서비스에게 명령을 요청한다.
2. 서비스는 엑세스 토큰을 인증 서비스에게 전달하여 사용자 인증 정보 확인 요청한다.
3. 인증 서비스는 수신된 엑세스 토큰에 매핑된 사용자 인증 정보를 서비스에게 전달한다
4. 서비스는 인증 서비스로부터 전달받은 사용자 인증 정보를 기반으로 요청을 처리한다

서비스는 요청된 모든 명령 처리를 위해 인증 서비스에 질의 해야하며 그에 따라
강한 의존성을 갖게 된다. 이러한 상황에서 인증 서비스에 장애가 발생한다면
전체 서비스로 장애가 전파 될 것 이다.



MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ JWT(Json Web Token) 인증

엑세스 토큰과 다르게 JWT 는 토큰 스스로가 사용자 인증 정보를 갖고 있다. 인증 서비스는 토큰 발급 시 Base64 로 인코딩된 Json 형식의 사용자 인증 정보와 해당 인증 정보의 위변조 검증을 위한 시그니처(Signiture) 를 토큰에 첨부하여 함께 제공한다. 첨부된 시그니처는 인증 서비스가 발급한 공개키(public key) 를 통해 검증 가능하다.

JWT 시그니처 검증 목적은 사용자 인증 정보 해석이 아닌 첨부된 인증 정보의 위변조가 없음을 확인하기 위함이다. 사용자 인증 정보는 Base64 를 통해 쉽게 decode 가능하기 때문에 사용자의 개인정보 등 보안이 필요한 성격의 정보는 JWT 에 포함하면 안된다.

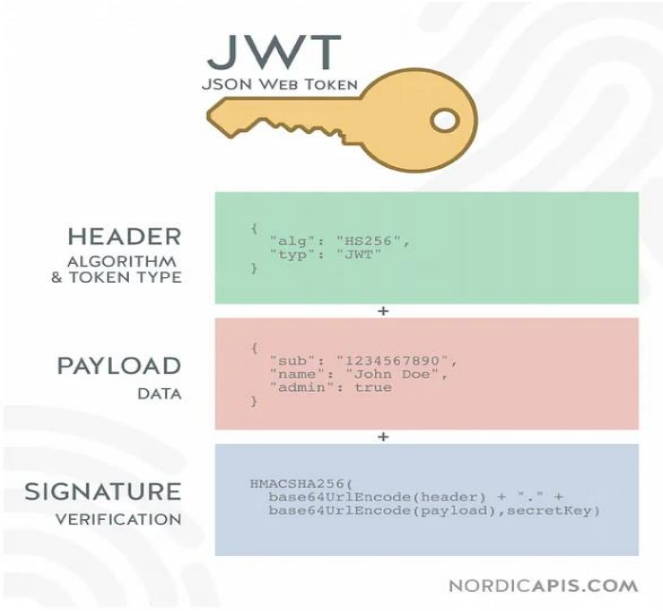
JWT 검증 절차
1. 사용자는 JWT를 첨부하여 서비스에게 명령을 요청한다.
2. 서비스는 미리 발급받은 공개키를 사용해 JWP 의 유효성을 검증한다.
3. 서비스는 유효성이 검증된 JWT 를 Base64 를 이용하여 사용자 인증 정보를 디코드한다.
4. 서비스는 디코드된 사용자 인증 정보를 기반으로 요청 처리한다.

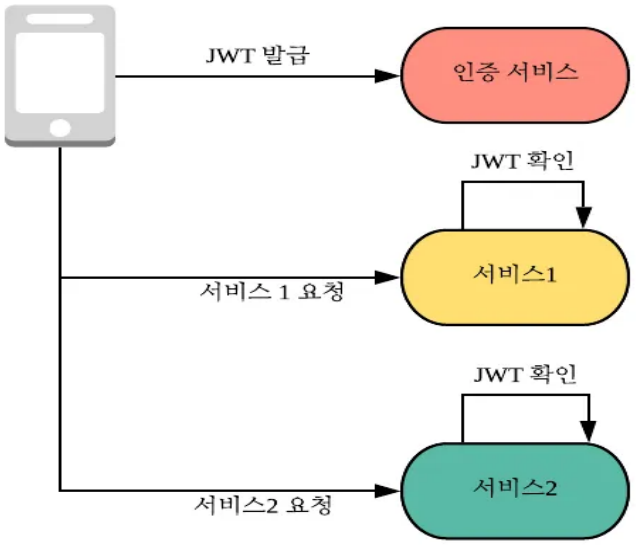
토큰 스스로가 사용자 정보를 포함하고 있어 서비스들은 인증 서비스와 별도의 의존성을 갖지 않게 되었다. 반면 이러한 특성으로 발생할 수 있는 문제점 또한 있다.

JWT 문제점
발급된 토큰 관리의 어려움, 엑세스 토큰의 경우 중앙에서 관리하여 각 서비스들이 인증 서비스에 질의해야 하는 반면, JWT 는 한 번 발급된 토큰은 분산되어 의존성 없이 스스로 인증되기 때문에 관리에 어려움이 있다.

1. 접근제어
   엑세스 토큰은 인증 서비스가 저장된 토큰 정보를 삭제해서 토큰 사용을 불가능하게 하지만, JWT 는 토큰에 명시된 만료 시간 전까지는 제어가 불가능하다.
2. 변경된 정보 적용
   엑세스 토큰은 토큰에 매핑된 정보를 수정하여 즉시 변경 적용이 가능하지만 JWT 는 접근제어와 마찬가지로 JWT 에 명시된 만료 시간 전까지 잘못된 정보로 서비스에 접근할 수 있다.

이러한 관리의 어려움으로 JWT 는 가능한 토큰 만료 시간을 짧게 가져가는 것이 좋다 토큰 만료 시간이 짧아 질수록 토큰 갱신을 위해 발생되는 비용이 커지게 되서 적절한 시간 선택이 필요하다.

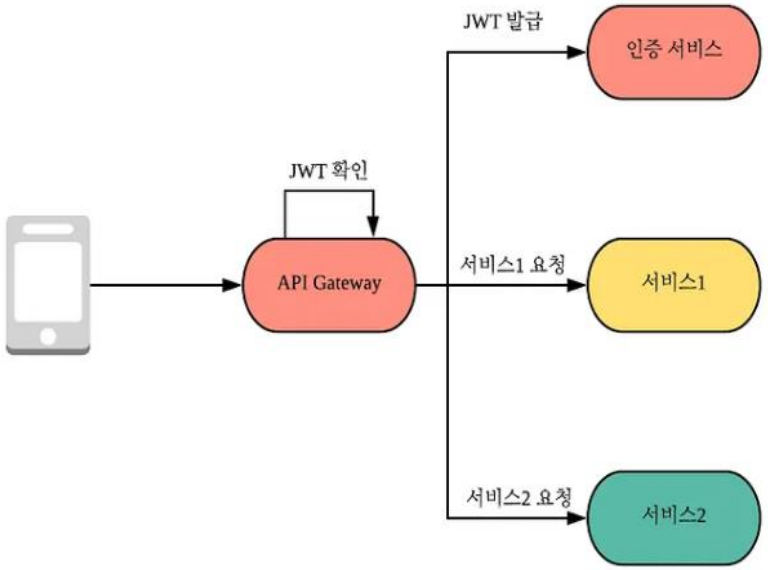




MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ API Gateway를 활용한 공통 인증

MSA에서 API Gateway는 여러 분리된 서비스 환경에서 사용자에게 하나의 엔드포인트를 제공하기 위해 사용되는 패턴이다. API Gateway는 서비스 최전방에 위치하며, 모든 사용자 요청은 API Gateway를 통해서만 각 서비스에 접근 가능하게 된다. 따라서 API Gateway는 서비스 전체에 미들웨어 계층으로써 공통된 로직을 처리할 수 있으며 인증도 API Gateway가 처리할 수 있는 공통 로직 중 하나이다.



API Gateway 인증 절차
1. 사용자는 JWT를 첨부하여 API Gateway를 통해 서비스에게 명령을 요청한다.
2. API Gateway는 요청된 명령을 수신하여 JWT 인증 로직을 수행한다.
3. 인증된 사용자 정보를 요청에 추가로 첨부하여 뒷단 서비스에게 전달한다.
4. 서비스는 API Gateway가 첨부한 사용자 인증 정보를 기반으로 요청을 처리한다.

앞 단에 API Gateway가 공통된 인증 절차를 수행하면서 뒷단의 서비스들은 인증 방식 으로 부터 완전히 독립되어 인증 서비스와 의존성이 사라지게 되었다. 이후 인증 절차에 어떠한 변화에도 다른 서비스들은 영향이 없을 것이다. 예를 들어 JWT의 인증 방식이 앞서 설명한 엑세스 토큰으로 변경되어도 API Gateway가 동일한 사용자 인증 정보를 반환할 수 있다면 뒷 단의 다른 서비스들은 어떠한 변경 사항도 없을 것이다.

모든 서비스의 요청은 API Gateway를 통해 각 서비스에 전달된다. 이러한 구조로 API Gateway는 API Gateway 하나의 장애로 서비스 전체가 먹통이 되는 SPOF(Single Point of Failure)가 될 수 있다. 이러한 상황을 대비하기 위해 API Gateway 도입 시에는 철저한 이중화가 준비되어야 한다.

■ 인증 캐시

토큰을 이용한 사용자 인증 결과는 가변성이 크지 않은 데이터이다. 방금 막 유효성이 확인된 토큰이 다음 요청에 결과가 변경되어 있을 가능성을 크지 않다.
이러한 데이터 성격으로 토큰을 키로 하여 사용자 인증 데이터를 캐시 하여 재사용할 수 있다.
재사용된 캐시 데이터는 아래의 이점을 갖는다.

1. 통신 오버해드 감소
   캐시 데이터를 재사용함으로써 반복되는 인증 오버해드(엑세스 토큰: 통신 오버해드, JWT: 시그니처 검증 연산 오버해드)를 줄일 수 있다.
2. 장애 전파 최소화
   인증 서비스 장애 발생 시에도 캐시 데이터를 사용함으로써 장애 영향도를 최소화할 수 있다.
   그러나 캐시의 사용은 상황에 따라 이미 내용이 변경되어 유효하지 않은 데이터를 참조 할 수도 있음을 뜻한다.
   따라서 각 서비스 성격과 정책에 따라 알맞는 캐시 만료 시간이 설정되어야 하며,
   일부 인증의 유효성 판단이 크리티컬(Critical)한 경우, 캐시 사용을 배제해야 한다.

MSA 구조(MSA) 의 인증(Authentication) 및 인가(Authorization)

■ 결론

지금까지 MSA 에서 사용될 수 있는 사용자 인증 전략을 살펴 보았다.
소개된 전략의 핵심은 서비스 전반으로 사용될 인증 서비스 기능과 다른 서비스의 의존성을 줄이는 것이다.
앞서 설명한 내용을 요약하면 아래와 같다.

JWT 를 적용하여 인증 서비스와 의존성 없이 각 서비스가 스스로 사용자 인증을 수행할 수 있도록 하자

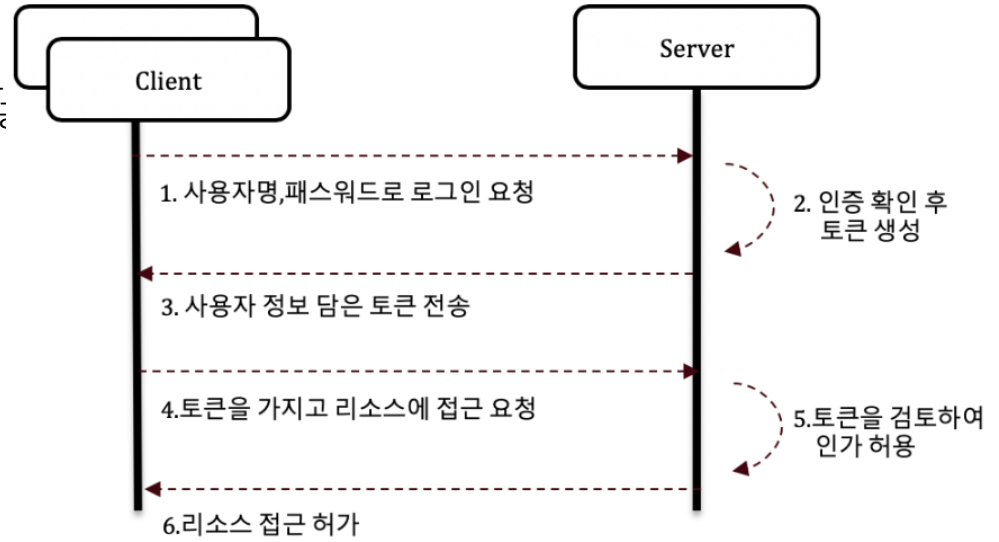
API Gateway 에서 공통 인증 절차를 수행하여 각 서비스와 인증 절차를 추상화 하자.

인증캐시를 사용하여 반복된 인증 절차를 줄이자.

MSA 인증/인가 패턴
여러 개의 MSA에 대한 인증/인가 등 접근 제어는 어떻게 구현해야 할까?
각 서비스가 모두 인증/인가를 중복으로 구현한다면 비효율적이다. 따라서 MSA 인증/인가를 처리하기 위해서는 일반적으로 다음과 같은 패턴들을 활용한다.

1. 중앙 집중 식 세션 관리
기존 **모노리스 방식**에서 가장 많이 사용했던 방식은 **서버 세션에 사용자의 로그인 정보 및 권한 정보를 저장하고 이를 통해 어플리케이션에 인증/인가를 판단하는 것이다**.
그렇지만 MSA 는 사용량에 따라 수시로 수평 확장할 수 있고 로드 밸런싱이 되기 때문에 세션 데이터가 손실될 수 있다.
따라서 마이크로비스는 각자의 서비스에 세션을 저장하지 않고 공유 저장소에 세션을 저장하고 모든 서비스가 동일한 사용자 데이터를 얻게 한다.
보통 **세션 저장소로 레디스(Redis), 맴캐쉬드(Memcached)를 사용**한다.

2. 클라이언트 토큰
세션은 서버의 중앙에 저장되고 토큰은 사용자의 브라우저에 저장된다. 토큰은 사용자이 신원정보를 가지고 있고
를 할 수 있다.JWT(Json Web Token)은 토큰 형식을 정의하고 암호화 하며 다양한 언어에 라이브러리를 제공하는
음 다이어그램과 같다.
브라우저가 서버에 로그인이름과 패스워드로 인증을 요청한다.
서버는 인증 후 토큰을 생성하고 브라우저에 토큰에 사용자 정보의 인증/인가 정보를 포함에 전송한다.
브라우저는 서버 리소스 요청 시 토큰을 같이 보낸다. 서버의 서비스는 토큰 정보를 확인후 자원 접근을 허가한다.



B2B 고객을 위한 제로 트러스트 인증 방식에 대한 프로세스를 설계하시오

제로 트러스트(Zero Trust) 인증 방식은 "절대 신뢰하지 말고 항상 검증하라"는 원칙에 기반하여 보안을 강화하는 접근 방식입니다.
이는 특히 B2B 환경에서 중요한 데이터와 시스템을 보호하는 데 매우 효과적입니다.

1. 사용자 및 기기 식별
- **다중 인증(Multi-Factor Authentication, MFA)**:
  사용자는 비밀번호 외에도 추가적인 인증 요소를 통해 본인임을 증명해야 합니다. 예를 들어, OTP(One-Time Password), 생체 인식(지문, 얼굴 인식), 인증 앱 등을 활용합니
- **기기 인증(Device Authentication)**:
  사용자가 접근하는 기기가 사전 등록된 신뢰할 수 있는 기기인지 확인합니다. 기기 인증에는 디지털 인증서, TPM(신뢰 플랫폼 모듈), MDM(모바일 장치 관리) 솔루션 등을 사용할 수 있습니다.

2. 동작 기반 인증 및 위험 평가
- **사용자 행동 분석(User Behavior Analytics, UBA)**:
  AI와 머신러닝을 활용하여 사용자의 행동 패턴을 분석합니다. 평소와 다른 비정상적인 활동이 감지되면 추가 인증이나 접근 제한이 필요할 수 있습니다.
  예: 사용자가 평소와 다른 위치에서 접근을 시도하거나, 예상치 못한 시간대에 접속하려 할 때 경고가 발생합니다.
- **위험 기반 인증(Risk-Based Authentication, RBA)**:
  접속 시도의 위험 수준을 평가하여 인증 강도를 동적으로 조절합니다. 예를 들어, 고위험 상황에서는 추가적인 인증을 요구하고, 저위험 상황에서는 간소화된 인증 절차를 허용합니다.

3. 최소 권한 원칙(Least Privilege Access)
- **역할 기반 접근 제어(Role-Based Access Control, RBAC)**:
  사용자는 자신의 역할에 맞는 최소한의 권한만 부여받습니다. 특정 데이터나 애플리케이션에 접근하기 위해 반드시 필요한 경우에만 권한을 요청하고, 이를 승인받아야 합니다.
- **정책 기반 접근 제어(Policy-Based Access Control, PBAC)**:
  특정 규칙이나 조건에 따라 접근을 제어하는 방식입니다. 예를 들어, 특정 IP 주소에서만 접근이 허용되거나, 특정 시간대에만 시스템에 접속할 수 있습니다.

4. 실시간 모니터링 및 지속적인 검증
- **실시간 모니터링 및 로깅**:
  모든 접속 시도와 활동을 실시간으로 모니터링하고 기록합니다. 보안 이상 징후가 포착되면 즉시 대응할 수 있도록 설정합니다.
- **연속적 인증(Continuous Authentication)**:
  사용자가 세션 동안 지속적으로 인증 상태를 유지할 수 있도록 합니다. 이는 특정 시간 간격으로 사용자의 신원을 재검증하거나, 행동 패턴을 분석하여 신뢰할 수 있는 사용자임을 지속적으로 확인

5. 데이터 암호화 및 보호
- **데이터 암호화(Encryption)**:
  전송 중이거나 저장된 모든 데이터는 암호화됩니다. SSL/TLS를 사용하여 네트워크 통신을 보호하고, 데이터베이스와 파일 시스템의 데이터를 암호화합니다.
- **데이터 분류 및 레이블링**:
  민감한 데이터는 분류되고, 중요도에 따라 레이블이 지정됩니다. 각 레벨에 따른 접근 정책이 적용되며, 높은 보안 등급이 요구되는 데이터는 더욱 강화된 보호 조치를 취합니다.

6. 적응형 접근 제어
- **동적 접근 정책(Dynamic Access Policy)**:
  시스템이 실시간으로 사용자의 상태와 환경을 평가하고, 그에 따라 접근 정책을 자동으로 조정합니다. 예를 들어, 보안 위협이 증가하면 자동으로 접근 권한을 축소하거나, 더 강력한 인증 절차를 요구
- **상황 인식(Context-Aware Security)**:
  사용자, 기기, 위치, 시간, 네트워크 상태 등 다양한 상황적 요소를 고려하여 접근을 제어합니다. 이러한 접근 방식은 제로 트러스트 원칙을 강화하는 데 필수적입니다.

7. 위협 탐지 및 대응
- **침입 탐지 시스템(IDS) 및 침입 방지 시스템(IPS)**: 네트워크 및 애플리케이션 레벨에서 실시간으로 위협을 감지하고, 이에 대한 자동 대응 조치를 수행합니다.
- **위협 인텔리전스(Threat Intelligence)**: 외부 위협 정보를 수집하고 분석하여, 시스템이 최신 위협에 대응할 수 있도록 합니다. 이를 통해, 새로운 공격 벡터나 취약점이 발견되었을 때 빠르게 대응

8. 정기적인 보안 검토 및 감사
- **보안 감사(Security Audits)**: 정기적으로 시스템 보안 상태를 검토하고, 정책의 준수 여부를 평가합니다. 보안 감사는 잠재적인 취약점을 파악하고, 개선 사항을 도출하는 데 중요합니다.
- **침투 테스트(Penetration Testing)**: 실제 공격 시나리오를 기반으로 시스템의 보안을 테스트하여, 제로 트러스트 아키텍처의 실효성을 검증합니다.

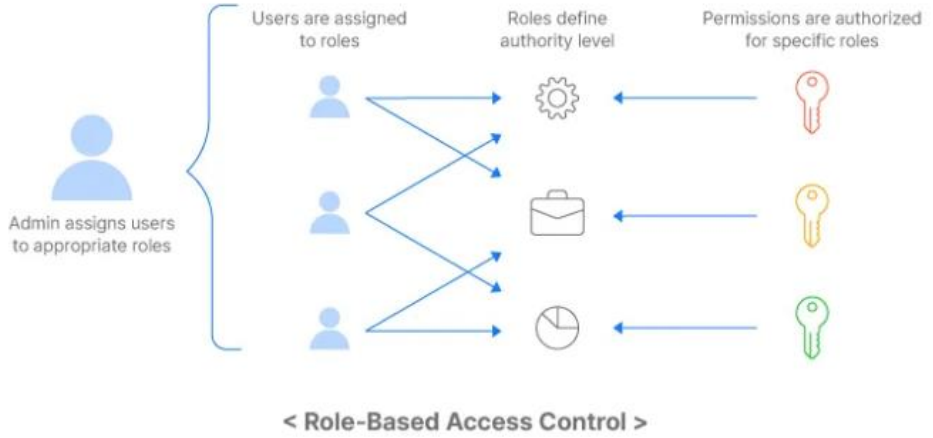
이러한 제로 트러스트 인증 프로세스는 B2B 고객이 접근하는 모든 자산에 대해 지속적인 신뢰 검증과 보안 보호를 제공하며, 잠재적인 보안 위협을 최소화합니다.

역할 기반 엑세스 제어(Role Based Access Control, RBAC)란?

역할 기반 *엑세스 제어(RBAC)는 정보 시스템 및 네트워크 보안에서 사용되는 중요한 접근 제어 모델 중 하나입니다
이 모델은 사용자 역할, 작업 및 권한을 중심으로 구성되며, 시스템에 대한 접근을 조직화하고 제어하기 위해 사용
RBAC는 각 사용자에게 하나 이상의 *"역할"을 할당하고 각 역할에 서로 다른 권한을 부여하여 이를 수행합니다.
RBAC는 단일 소프트웨어 애플리케이션 또는 여러 애플리케이션에 적용할 수 있습니다.

*엑세스 제어 : 사이버 보안에서 액세스 제어는 사용자가 수행할 수 있는 작업과 볼 수 있는 데이터를 제한하고 제어하는 도구를 지칭합니다. 스마트폰 잠금을 해제하기 위해 비밀번호를 입력하는 것도 액세스 제어의 기본적인 예입니다. 비밀번호를 아는 사람만 전화기의 파일과 애플리케이션에 액세스할 수 있습니다.

*역할 : RBAC에서는 역할에 대한 보다 기술적인 정의가 있습니다. 회사 시스템 내에서 사용하기 위해 명확하게 정의된 기능 또는 권한의 집합입니다. 각 내부 사용자에게는 적어도 하나의 역할이 할당되며 일부는 여러 역할을 가질 수 있습니다.



< Role-Based Access Control >

RBAC의 필요성 및 장점
- **필요성 :** 보안을 강화하고 정보 시스템의 접근을 효과적으로 관리하여 데이터 보호와 규정 준수를 달성합니다.
- **보안 강화 :** 역할 기반 접근 제어는 무단 액세스를 제한하고 중요 데이터를 보호하는데 도움을 줍니다.
- **규정 준수 :** 규제 및 법률을 준수하며 감사 추적을 용이하게 하여 조직의 규정 준수를 지원합니다.
- **관리 효율성 :** 사용자 및 권한을 중앙에서 관리하여 운영 및 비용을 최적화합니다.
- **업무 효율성 :** 역할 기반 접근 제어는 사용자가 필요한 권한을 간단하게 얻도록 도와 업무 효율성을 높입니다.

RBAC모델의 권한 방식 3가지

1. 핵심 RBAC

핵심 모델은 RBAC의 모든 단일 구성 요소를 정교하게 만드는 것입니다. 각 역할부터 각 권한까지 모든 것이 이 모델을 통해 지정됩니다. 따라서 이는 다른 2가지 유형의 RBAC의 기반이 될 뿐만 아니라 사용자 액세스 권한을 관리하기 위한 독립형 방법으로도 작동할 수 있습니다.

- **핵심 RBAC의 기본규칙**
  - **역할 할당 :** 사용자는 주체에게 역할이 할당된 경우에만 권한을 행사할 수 있습니다.
  - **역할 기반 권한 부여 :** 사용자의 역할에 권한이 부여되어 사용자가 권한이 부여된 역할만 수행할 수 있도록 합니다.
  - **권한 승인 :** 사용자는 역할 할당 및 권한 부여에 따라 권한이 부여된 경우 특정 권한을 행사할 수 있습니다.

2. 계층적 RBAC

RBAC 모델의 변형으로, 높은 수준의 역할에 속한 사용자에게는 더 많은 권한과 보안이 제공되고, 낮은 수준의 역할에 속한 사용자에게는 제한된 권한이 부여됩니다.
이로써 보안을 강화하고 비용을 절감하는 데 도움이 됩니다.

역할 기반 엑세스 제어(Role Based Access Control, RBAC)란?

3. 제한된 RBAC

RBAC 배포의 책임 또는 임무는 이 표준을 통해 지정됩니다. 구현 또는 직무 분리(SD)는 요구 사항에 따라 정적이거나 동적일 수 있습니다.

- **정적 모델(Static Model — SSD) :** 상호 배타적인 역할이 동일한 개인에게 부여되지 않도록 방지합니 다. 예를 들어, 전자 상거래 플랫폼에서 구매자와 판매자 역할은 상호 배타적이며 사용자는 하나의 역할 만 가질 수 있습니다. 이로써 역할 간 권한 충돌을 방지하고 사용자에게 엄격한 역할을 할당합니다.
- **동적 모델(Dynamic Model — DSD) :** 사용자가 제한없이 여러 권한을 가질 수 있지만, 동일한 세션에서는 동시에 사용하는 것은 불가능합니다. 사용자는 특정 권한을 실제로 사용하려면 추가 승인이 필요합니다. 이는 더 유연한 권한 관리를 가능하게 합니다.

RBAC 모델의 선택은 조직의 보안 및 역할 관리 요구에 따라 다를 수 있으며, 이러한 다양한 방식을 활용하여 사용자 액세스를 효과적으로 관리할 수 있습니다.

RBAC의 예시

조직은 역할 또는 그룹별로 사용자를 지정할 수 있고, 역할 그룹에 사용자를 추가한다는 것은 새 사용자가 해당 특정 그룹의 모든 권한에 액세스할 수 있음을 의미합니다.
조직은 관리자, 업무별 최종 사용자 또는 게스트를 포함하도록 역할을 분할할 수 있습니다.
예를 들면 다음과 같은 역할이 있습니다 :

- 조직의 소프트웨어 엔지니어링 도구(GitHub, Docker 및 Jenkins)에 대한 역할과 액세스 권한이 부여된 **소프트웨어 엔지니어**
- 조직의 마케팅 도구(이메일 마케팅 목록, Google Analytics 또는 소셜 미디어 프로필)에 대한 역할과 액세스 권한이 부여된 **마케팅 담당자**
- 조직의 HR 관련 도구(ADP, Oracle Cloud Human Capital Management 또는 Paycor)에 대한 역할과 액세스 권한이 부여된 **인사 담당자**

소프트웨어 엔지니어는 같은 회사의 HR 또는 마케팅 담당자가 보유하고 있는 도구나 파일에 액세스할 수 없지만 작업을 완료하는 데 필요한 도구에는 액세스할 수 있습니다.
마찬가지로 마케팅 담당자는 자신의 역할에 따라 필요한 도구에 액세스할 수 있지만 HR 또는 소프트웨어 엔지니어링 도구에는 액세스할 수 없습니다.

RBAC vs. ABAC(Attribute-Based Access Control)

RBAC(역할 기반 액세스 제어)와 ABAC(속성 기반 액세스 제어)는 모두 액세스 제어 방법이지만 접근 방식이 다릅니다. **RBAC**는 사용자의 역할에 따라 액세스 권한을 부여하는 반면,

ABAC는 다음 범주의 조합을 기반으로 액세스를 제어합니다.

- **사용자 속성 :** 사용자 이름, 국적, 조직, ID, 역할 및 보안 허가가 포함될 수 있습니다 .
- **자원 속성 :** 액세스 되는 개체의 소유자, 이름 및 데이터 생성 날짜를 설명할 수 있습니다.
- **작업 속성 :** 액세스 중인 시스템 또는 애플리케이션과 관련된 작업을 설명합니다.
- **환경적 속성 :** 액세스 위치, 액세스 시간 및 위협 수준이 포함될 수 있습니다.

ABAC는 특정 속성을 추가하여 권한 부여 옵션을 기하급수적으로 증가시켜 RBAC에 비해 다른 수준의 제어를 구현합니다. RBAC보다 훨씬 더 유연하지만 이러한 유연성을 적절하게 구현 및 관리하지 않을 경우 위험을 증가시킬 수 있습니다.

< Attribute-Based Access Control >

통신사 K의 사내 관리 시스템에 RBAC를 적용하기 위한 데이터베이스(역할, 사용자, 권한) 테이블을 설계하고 ERD를 작성하세요.힌트**: 사용자 테이블(Users), 역할 테이블(Roles), 권한 테이블(Permissions)과 사용자-역할(UserRoles), 역할-권한(RolePermissions) 관계 테이블을 설계하세요. ERD를 통해 각 테이블 간 관계를 시각화하세요.**

RBAC(Role-Based Access Control)은 역할 기반 접근 제어를 통해 사용자가 특정 역할에 따라 시스템 내에서 수행할 수 있는 작업을 제어하는 방법입니다.
이를 적용하기 위해 필요한 기본적인 데이터베이스 테이블은 다음과 같습니다:

1. Users 테이블: 시스템의 사용자 정보를 저장.
2. Roles 테이블: 시스템 내의 역할 정보를 저장.
3. Permissions 테이블: 시스템에서 수행할 수 있는 권한(작업) 정보를 저장.
4. UserRoles 테이블: 사용자와 역할 간의 매핑 정보를 저장 (Many-to-Many 관계).
5. RolePermissions 테이블: 역할과 권한 간의 매핑 정보를 저장 (Many-to-Many 관계).

1. 테이블 설계

a) Users 테이블

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(100) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

user_id: 사용자 ID, 기본 키.
username: 사용자 이름, 시스템에서 고유해야 함.
password: 사용자 비밀번호 (해싱된 형태로 저장).
email: 사용자 이메일 주소, 고유해야 함.
created_at, updated_at: 사용자 생성 및 업데이트 타임스탬프.

b) Roles 테이블

```
CREATE TABLE Roles (
    role_id INT PRIMARY KEY AUTO_INCREMENT,
    role_name VARCHAR(50) NOT NULL UNIQUE,
    description VARCHAR(255)
);
```

role_id: 역할 ID, 기본 키.
role_name: 역할 이름 (예: Admin, Manager, User), 고유해야 함.
description: 역할에 대한 설명.

c) Permissions 테이블

```
CREATE TABLE Permissions (
    permission_id INT PRIMARY KEY AUTO_INCREMENT,
    permission_name VARCHAR(50) NOT NULL UNIQUE,
    description VARCHAR(255)
);
```

permission_id: 권한 ID, 기본 키.
permission_name: 권한 이름 (예: ViewDashboard, EditUser), 고유해야 함.
description: 권한에 대한 설명.

d) UserRoles 테이블

```
CREATE TABLE UserRoles (
    user_id INT,
    role_id INT,
    PRIMARY KEY (user_id, role_id),
    FOREIGN KEY (user_id) REFERENCES Users(user_id),
    FOREIGN KEY (role_id) REFERENCES Roles(role_id)
);
```

user_id: Users 테이블의 사용자 ID, 외래 키.
role_id: Roles 테이블의 역할 ID, 외래 키.
PRIMARY KEY (user_id, role_id): 복합 기본 키로 사용자의 역할을 고유하게 관리.

e) RolePermissions 테이블

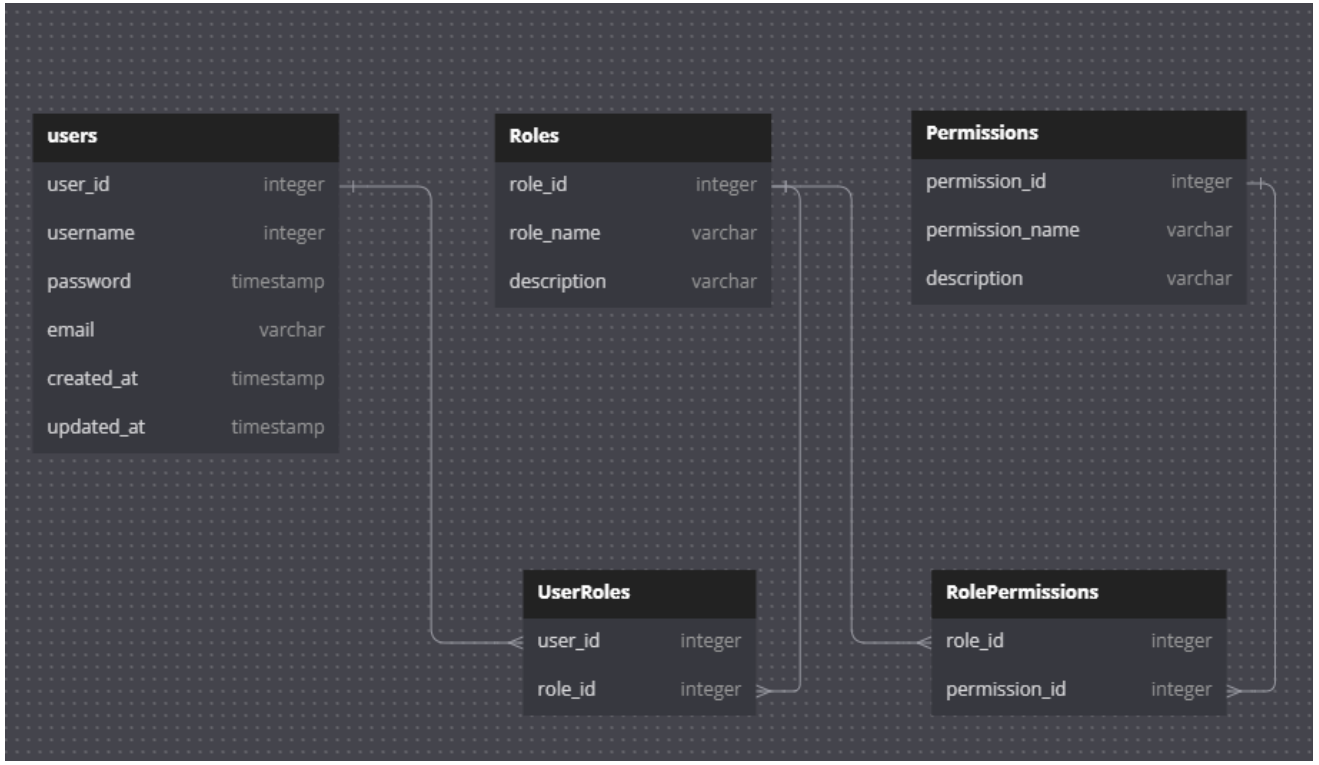
```
CREATE TABLE RolePermissions (
    role_id INT,
    permission_id INT,
    PRIMARY KEY (role_id, permission_id),
    FOREIGN KEY (role_id) REFERENCES Roles(role_id),
    FOREIGN KEY (permission_id) REFERENCES Permissions(permission_id)
);
```

role_id: Roles 테이블의 역할 ID, 외래 키.
permission_id: Permissions 테이블의 권한 ID, 외래 키.
PRIMARY KEY (role_id, permission_id): 복합 기본 키로 역할의 권한을 고유하게 관리.

통신사 K의 사내 관리 시스템에 RBAC를 적용하기 위한 데이터베이스(역할, 사용자, 권한) 테이블을 설계하고 ERD를 작성하세요.힌트**: 사용자 테이블(Users), 역할 테이블(Roles), 권한 테이블(Permissions)과 사용자-역할(UserRoles), 역할-권한(RolePermissions) 관계 테이블을 설계하세요. ERD를 통해 각 테이블 간 관계를 시각화하세요.**

2. ERD (Entity-Relationship Diagram) 작성

ERD를 통해 각 테이블 간의 관계를 시각적으로 표현합니다:



plaintext

```
+------------+          +-------------+          +--------------+
|   Users    |          |    Roles    |          |  Permissions |
+------------+          +-------------+          +--------------+
| user_id PK |<-------+ | role_id PK  |<-------+ | permission_id PK |
| username   |        | | role_name   |        | | permission_name  |
| password   |        | | description |        | | description      |
| email      |        | +-------------+        | +--------------+
+------------+        |                        |
                      |                        |
    +-----------------+|       +----------------+|
    |                  |       |                 |
+---------+   +--------------------+   +--------------+
|UserRoles|   | RolePermissions    |   | UserRoles    |
+---------+   +--------------------+   +--------------+
|user_id FK   | role_id FK         |   | role_id FK   |
|role_id FK   | permission_id FK   |   | permission_id FK  |
+-------------+--------------------+   +--------------+
```

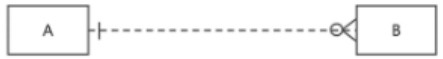
3. ERD 설명

Users와 Roles는 Many-to-Many 관계로, 하나의 사용자는 여러 역할을 가질 수 있고, 하나의 역할은 여러 사용자에게 할당될 수 있습니다. 이를 UserRoles 중간 테이블이 연결합니다.
Roles와 Permissions도 Many-to-Many 관계로, 하나의 역할은 여러 권한을 가질 수 있으며, 하나의 권한은 여러 역할에 할당될 수 있습니다. 이를 RolePermissions 중간 테이블이 연결합니다.
ERD에서는 각 테이블과 관계를 시각화하여, 각 엔터티 간의 관계를 쉽게 이해할 수 있도록 구성하였습니다.

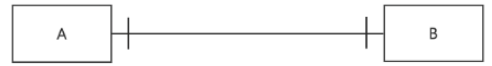
결론

이 데이터베이스 설계를 통해 통신사 K의 사내 관리 시스템에 RBAC(Role-Based Access Control)를 효율적으로 적용할 수 있습니다. 사용자, 역할, 권한 간의 관계를 명확히 정의하여, 관리자는 역할에 따라 권한을 쉽게 부여하고 관리할 수 있으며, 사용자 접근 제어를 체계적으로 관리할 수 있습니다.

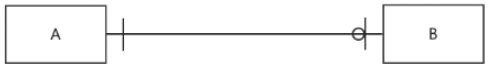
ERD (Entity-Relationship Diagram)

기호	의미
사각형	엔터티
타원	0개
해쉬 마크	1개
까마귀 발	2개 이상 (n)
실선	Identifying 부모테이블의 기본 키를 자식테이블의 기본 키로 사용  A가 없으면 B가 존재할 수 없는 관계
점선	Non-Identifying  A가 없어도 B가 존재할 수 있는 관계

관계 (Relationship)



하나의 A는 하나의 B로 구성되어 있다.

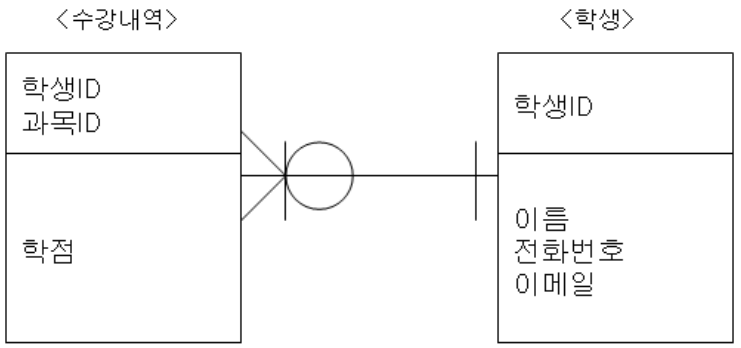


하나의 A는 0 또는 하나의 B로 구성되어 있다.

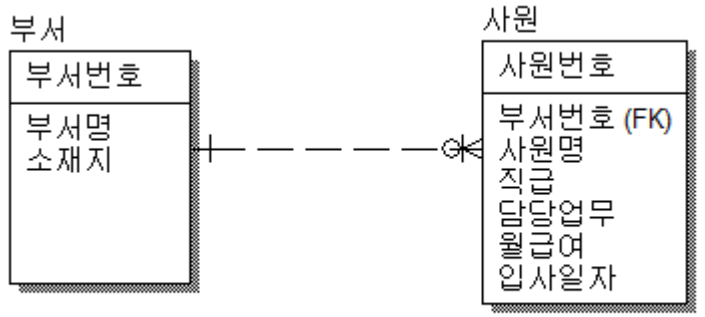
하나의 A는 두개 이상의 B로 구성되어 있다.

하나의 A는 하나 이상의 B로 구성되어 있다.

하나의 A는 0, 1, 또는 그 이상의 B로 구성되어 있다.



- **부모** 테이블은 학생 테이블이다.
- **자식** 테이블은 수강내역 테이블이다.
- **부모 테이블의 PK를 자식 테이블에서 PK로 사용**하고 있다.
- 학생 한 명은 0~N 개의 수강내역을 가진다.
- 수강내역은 하나의 학생을 가진다.
- **수강내역 테이블은 학생 테이블의 PK인 [학생ID]를 FK로 가진다.**



- **부모** 테이블은 부서 테이블이다.
- **자식** 테이블은 사원 테이블이다.
- 자식 테이블이 **부모 테이블의 PK를 가지고 있지만 이를 PK로 사용하지는 않는다.**
- 하나의 부서는 0~N 명의 사원을 가질 수 있다.
- **사원 테이블은 부서 테이블의 PK인 부서번호를 FK로 가진다.**

중복 로그인 방지

■ Spring Security 구조, 흐름 그리고 역할 알아보기

1. Spring Security란?

스프링 시큐리티는 인증(Authentication) ,권한(Authorize) 부여 및 보호 기능을 제공하는 프레임워크다.

Java / Java EE 프레임워크

개발을 하면서 보안 분야는 시간이 많이 소요되는 활동들 중 하나다. Spring Security를 사용함으로써 짜여진 내부 로직을 통해 인증, 권한 확인에 필요한 기능과 옵션들을 제공한다.

2. 인증(Authentication) , 인가(Authorization)

인증과 인가란 무엇일까? 보통 인증 절차를 거친 후 인가 절차를 진행한다.

- 인증: 해당 사용자가 본인이 맞는지를 확인하는 절차.
- 인가: 인증된 사용자가 요청된 자원에 접근 가능한가를 결정하는 절차

3. 인증 방식

1. credential 방식: username, password를 이용하는 방식
2. 이중 인증(two factor 인증): 사용자가 입력한 개인 정보를 인증 후, 다른 인증 체계(예: 물리적인 카드)를 이용하여 두가지의 조합으로 인증하는 방식이다.
3. 하드웨어 인증: 자동차 키와 같은 방식

이중 Spring Security는 credential 기반의 인증을 취합니다.

. principal: 아이디 (username), credential: 비밀번호 (password)

특정 자원에 대한 접근을 제어하기 위해서는 **권한**을 가지게 된다.

특정 권한을 얻기 위해서는 유저는 인증정보(Authentication)가 필요하고 관리자는 해당 정보를 참고해 권한을 인가(Authorization)한다.

보편적으로 username - password 패턴의 인증방식을 거치기 때문에 **스프링 시큐리티는 principal - credential 패턴**을 가지게 된다.

4. Spring Security의 특징

- Filter를 기반으로 동작한다. : Spring MVC와 분리되어 관리하고 동작할 수 있다.
- Bean으로 설정할 수 있다. : Spring Security 3.2부터는 XML설정을 하지 않아도 된다.

중복 로그인 방지

■ Spring Security 구조, 흐름 그리고 역할 알아보기

5. Spring Security Architecture(구조)

1. Http Request 수신 : 사용자가 로그인 정보와 함께 인증 요청을 한다.

2. 유저 자격을 기반으로 인증토큰 생성 : AuthenticationFilter가 요청을 가로채고, 가로챈 정보를 통해 UsernamePasswordAuthenticationToken의 인증용 객체를 생성한다.

3. FIlter를 통해 AuthenticationToken을 AuthenticationManager로 위임
AuthenticationManager의 구현체인 ProviderManager에게 생성한 UsernamePasswordToken 객체를 전달한다.

4. AuthenticationProvider의 목록으로 인증을 시도
AutenticationManger는 등록된 AuthenticationProvider들을 조회하며 인증을 요구한다.

5. UserDetailsService의 요구
실제 데이터베이스에서 사용자 인증정보를 가져오는 UserDetailsService에 사용자 정보를 넘겨준다.

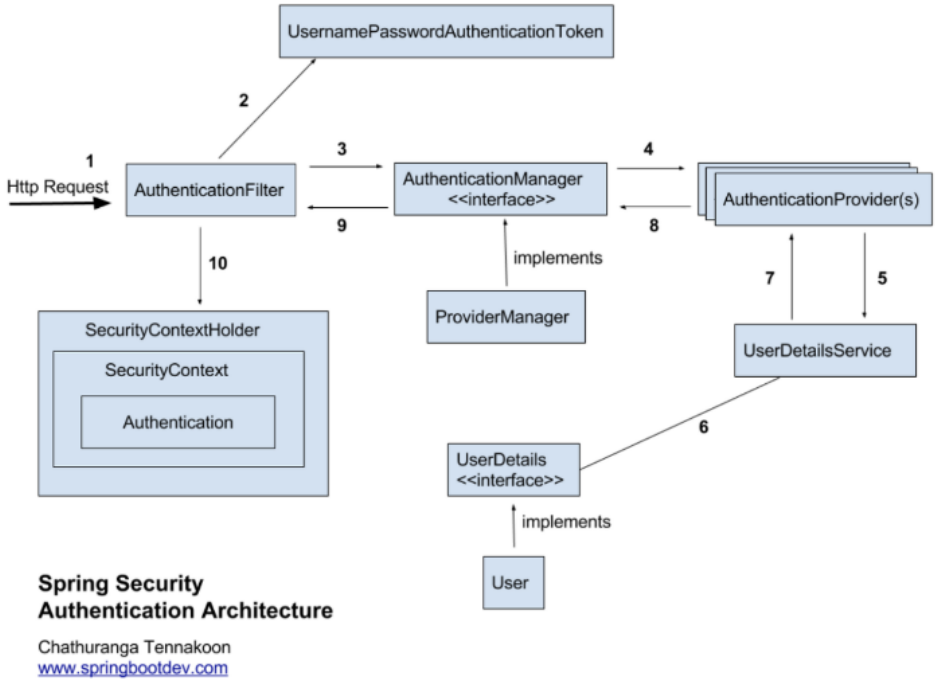
6. UserDetails를 이용해 User객체에 대한 정보 탐색
넘겨받은 사용자 정보를 통해 데이터베이스에서 찾아낸 사용자 정보인 UserDetails 객체를 만든다.

7. User 객체의 정보들을 UserDetails가 UserDetailsService(LoginService)로 전달
AuthenticaitonProvider들은 UserDetails를 넘겨받고 사용자 정보를 비교한다.

8. 인증 객체 or AuthenticationException
인증이 완료가되면 권한 등의 사용자 정보를 담은 Authentication 객체를 반환한다.

9. 인증 끝
다시 최초의 AuthenticationFilter에 Authentication 객체가 반환된다.

10. SecurityContext에 인증 객체를 설정
Authentication 객체를 Security Context에 저장한다.



최종적으로는 SecurityContextHolder는 세션 영역에 있는 SecurityContext에 Authentication 객체를 저장한다. 사용자 정보를 저장한다는 것은 스프링 시큐리티가 전통적인 세션-쿠키 기반의 인증 방식을 사용한다는 것을 의미한다.

중복 로그인 방지

■ 사용자의 중복 요청 방지하기(feat. 멱등키, Redis)

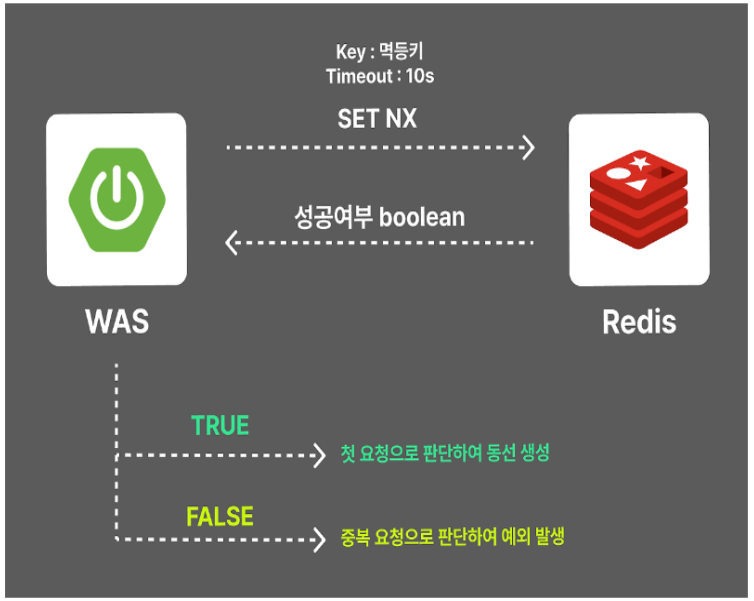
1. 중복 요청을 어떻게 식별할까? (멱등키)

먼저 중복 요청을 방지하기 위해서는 중복 요청에 대해서 '멱등성'을 보장해야 했습니다.
멱등성이란, 첫 번째 수행을 한 뒤 여러 차례 적용해도 결과를 변경시키지 않는 작업 또는 기능의 속성을 뜻합니다.
즉, 멱등한 작업의 결과는 한 번 수행하든 여러 번 수행하든 같습니다.
멱등성을 보장해서 여러 번의 중복 요청이 와도 처음 요청 시에만 동선을 생성하고, 그 후에는 생성한 결과가 변하지 않게 해야합니다.
여기서 멱등성을 보장하는 시간은 서버의 최대 응답 지연 시간을 고려해서 설정했습니다.
최대 응답 지연 시간 동안의 들어오는 요청은 모두 중복 요청이라고 판단했습니다.
그래서 저는 최대 10초 동안 응답 지연이 있을 수 있다고 고려해서 멱등성 보장 시간을 10초로 결정했습니다.
그리고 요청이 중복되었음을 판단할 요청의 식별자, 즉 멱등키가 필요했습니다.
해당 식별자는 현재 비스니스에서 '1명의 사용자의 같은 날짜 동선 생성 요청'을 식별해야 했습니다.
그래서 멱등키는 다음과 같은 요소가 들어가도록 결정하게 되었습니다.

addRoute(동선 생성 메소드 이름)
memberId(사용자 식별자)
date(동선 날짜)
이후 구현이 어떻게 되든 우선 중복 요청을 식별하는 멱등키는 위의 요소를 포함하여 설계를 진행했습니다.
이렇게 처음 요청에서 멱등키를 생성하고, 요청마다 들어오는 멱등키를 체크하여 중복 요청을 방지할 수 있습니다.

2 Redis Set을 사용한 중복 방지 구현

멱등키를 저장하는 Redis의 자료 구조로는 Set을 선택했습니다.
Redis의 명령 중 SET을 사용하고 NX를 옵션으로 전달하여 명령을 수행할 것입니다.
(Redis 2.6.12 이전에는 SETNX 명령이 별도로 있었지만, Deprecated 되어 SET 명령어에 NX를 옵션으로 전달해야 합니다.)
SET : key-value 형태인 Set 자료 구조의 key에 해당하는 value를 지정하는 명령어
NX 옵션 : Not Exist의 약자로, Key가 존재하지 않을 때만 value를 지정하는 명령어
해당 명령어를 사용한 Flow는 다음과 같습니다.
1. 사용자의 요청이 들어오면 멱등성이 보장되는 멱등키를 생성한다.
2. 해당 멱등키를 Redis에 Set 자료 구조로 SET NX를 통해 저장한다. (Spring Data Redis의 RedisTemplate 사용)
3. 멱등키에 해당하는 키가 존재하지 않으면 만료 시간이 10초로 멱등키를 저장하고 Application에 true를 반환한다.
4. 멱등키에 해당하는 키가 이미 존재한다면 저장하지 않고 Application에 false를 반환한다.
5. SET NX 명령어 성공 여부에 따라 분기 처리로 로직을 수행한다.
   True : 첫 요청으로 판단하여 동선을 생성한다.
   False : 중복된 요청으로 판단하여 예외를 발생시킨다.



인증방식의 전환에 따른 방안을 제시하고 프로세스를 설계하시오

인증방식을 기존의 방식에서 새로운 방식으로 전환하는 것은 보안성 강화와 사용자 경험 개선을 위한 중요한 결정입니다. 이러한 전환은 단계적이고 신중하게 수행되어야 하며, 사용자와 시스템에 미치는 영향을 최소화해야 합니다.

1. 인증방식 전환의 필요성 분석 및 선택

a) 기존 인증방식 평가

- **보안성 분석**: 현재 사용 중인 인증방식의 보안 수준을 평가합니다. 예를 들어, 단순 비밀번호 기반 인증이더라도 취약한 비밀번호, 피싱 공격, 브루트포스 공격 등에 대한 취약성을 분석합니다.
- **사용자 경험(UX) 평가**: 기존 인증방식이 사용자에게 제공하는 경험을 평가합니다. 예를 들어, 인증 절차가 복잡하거나 시간이 많이 걸리는지, 또는 사용자가 자주 비밀번호를 잊는 문제가 있는지 확인

b) 새로운 인증방식 선택

- **강화된 보안성**: 다중 인증(MFA), 생체 인식(Fingerprint, Face ID), OAuth2, OpenID Connect, 또는 패스워드리스 인증(Passwordless Authentication)과 같은 강력한 보안 방식을 선택합니다.
- **사용자 편의성**: 사용자가 쉽게 접근하고 사용할 수 있는 인증방식을 선택합니다. 예를 들어, OTP(One-Time Password) 또는 FIDO(패스워드리스 인증)를 활용할 수 있습니다.

2. 전환 전략 수립

a) 단계적 전환

- **병행 운영 단계(Parallel Operation)**: 기존 인증방식과 새로운 인증방식을 일정 기간 동안 병행 운영합니다. 이는 사용자가 새로운 인증방식에 적응할 시간을 제공하며, 시스템 안정성을 검증
  예: 사용자가 기존 비밀번호 인증을 통해 로그인한 후, 새로운 인증방식(예: OTP, 생체 인식)으로 전환하도록 안내.
- **사용자 그룹별 전환**: 모든 사용자에게 한꺼번에 전환을 요구하지 않고, 특정 사용자 그룹부터 전환을 시작합니다. 이는 전환 과정에서 발생할 수 있는 문제를 최소화할 수 있습니다.
  예: 내부 직원부터 새로운 인증방식으로 전환하고, 이후 외부 사용자를 점진적으로 전환.

b) 사용자 교육 및 지원

- **가이드 제공**: 새로운 인증방식 사용법에 대한 가이드라인, FAQ, 튜토리얼 비디오 등을 제공하여 사용자가 전환 과정에서 어려움을 겪지 않도록 합니다.
  예: 새로운 인증 방식이 생체 인식일 경우, 기기에서 생체 인식을 설정하는 방법에 대한 가이드 제공.
- **헬프데스크 지원 강화**: 전환 과정에서 발생할 수 있는 문제를 신속히 해결하기 위해 헬프데스크를 강화합니다. 사용자 문의에 대한 실시간 지원을 제공하여 문제 발생 시 빠르게 대응할 수 있도록 합니다.

3. 시스템 및 데이터 준비

a) 시스템 통합 및 테스트

- **시스템 통합**: 새로운 인증방식이 기존 시스템과 완벽하게 통합되도록 합니다. 이는 인증 서버, 사용자 데이터베이스, 애플리케이션 서버 간의 원활한 연동을 의미합니다.
- **전환 전 테스트**: 새롭게 도입되는 인증방식을 다양한 환경(브라우저, 운영체제, 기기)에서 충분히 테스트하여 모든 사용자가 원활하게 이용할 수 있도록 합니다.
  예: 다양한 모바일 기기와 웹 브라우저에서 OTP 또는 생체 인식 인증이 제대로 동작하는지 검증.

b) 데이터 마이그레이션

- **사용자 데이터 준비**: 새로운 인증방식에 필요한 사용자 데이터를 준비합니다. 예를 들어, 생체 인식 정보를 저장하거나 OTP 발송을 위한 연락처 정보를 수집합니다.
- **데이터 암호화 및 보호**: 전환 과정에서 사용자의 민감한 데이터를 보호하기 위해 강력한 암호화 방식을 사용하고, 데이터 전송 시 SSL/TLS를 통해 보안을 강화합니다.

4. 모니터링 및 피드백 수집

a) 실시간 모니터링

- **전환 상태 모니터링**: 인증 전환 과정에서 발생하는 모든 로그인 시도와 오류를 실시간으로 모니터링하여 문제 발생 시 즉각 대응할 수 있도록 합니다.
  예: 실패한 로그인 시도의 로그를 분석하여 새로운 인증방식에서 발생할 수 있는 문제를 식별.

b) 사용자 피드백 수집

- **피드백 루프 설정**: 전환 과정 중 사용자로부터 피드백을 수집하여 불편 사항을 파악하고, 이를 바탕으로 인증방식을 개선합니다.
  예: 로그인 후 설문조사를 통해 사용자가 새로운 인증방식에 대해 어떻게 느끼는지 조사.

5. 전환 완료 및 유지 관리

a) 전환 완료 단계

- **기존 인증방식 중단**: 병행 운영 단계가 종료되면 기존 인증방식을 완전히 중단하고, 모든 사용자가 새로운 인증방식을 사용하도록 합니다.
- **최종 데이터 검토**: 새로운 인증방식에 대한 최종 데이터를 검토하여, 모든 사용자가 문제없이 전환되었는지 확인합니다.

b) 지속적인 유지보수 및 업데이트

- **보안 업데이트**: 새로운 인증방식에 대한 최신 보안 패치와 업데이트를 주기적으로 적용하여 보안성을 유지합니다.
- **지속적인 모니터링**: 사용자가 새로운 인증방식을 지속적으로 원활히 사용할 수 있도록 모니터링을 계속하며, 필요 시 추가적인 개선 작업을 수행합니다.

결론

인증방식의 전환에 따른 방안을 제시하고 프로세스를 설계하시오

A고객사의 인증 방식 개선을 위해 다음과 같은 방안을 제안하고 프로세스를 설계해 드리겠습니다:

1. 토큰 기반 인증 방식으로 전환

기존의 세션 기반 인증에서 JWT(JSON Web Token) 기반 인증으로 전환합니다. 이는 다음과 같은 이점을 제공합니다:

- 서버의 부하 감소: 세션 저장소가 필요 없어 서버 확장성이 향상됩니다.
- 다중 서버 환경에서의 용이성: 토큰이 클라이언트 측에서 관리되어 서버 간 세션 공유 문제가 해결됩니다.
- 모바일 앱 및 다양한 클라이언트 지원 용이성

2. 리프레시 토큰 도입

- 액세스 토큰의 유효 기간을 짧게 설정하고, 리프레시 토큰을 통해 새로운 액세스 토큰을 발급받는 방식을 도입합니다:
- 액세스 토큰 유효 기간: 15분
- 리프레시 토큰 유효 기간: 30일 (재발급 시 갱신)

3. 중복 로그인 방지 메커니즘

사용자별로 고유한 리프레시 토큰을 관리하여 중복 로그인을 방지합니다:

- 새로운 로그인 시 기존 리프레시 토큰 무효화
- 리프레시 토큰 데이터베이스에서 사용자당 하나의 유효한 토큰만 유지

4. 개선된 인증 프로세스 설계

a) 로그인 프로세스:
1. 사용자가 아이디와 비밀번호로 로그인 요청
2. 서버에서 자격 증명 확인
3. 유효한 경우, 액세스 토큰과 리프레시 토큰 생성
4. 기존 리프레시 토큰이 있다면 무효화하고 새로운 토큰 저장
5. 클라이언트에 토큰 전송

b) 인증된 요청 처리:
1. 클라이언트가 액세스 토큰과 함께 요청
2. 서버에서 토큰 유효성 검증
3. 유효한 경우 요청 처리, 그렇지 않으면 401 Unauthorized 응답

c) 토큰 갱신 프로세스:
1. 액세스 토큰 만료 시 클라이언트가 리프레시 토큰으로 갱신 요청
2. 서버에서 리프레시 토큰 유효성 검증
3. 유효한 경우 새로운 액세스 토큰 발급, 그렇지 않으면 재로그인 요구

d) 로그아웃 프로세스:
1. 클라이언트에서 로그아웃 요청
2. 서버에서 해당 사용자의 리프레시 토큰 무효화
3. 클라이언트에 로그아웃 성공 응답
4. 보안 강화 방안
   . HTTPS 사용 필수
   . 토큰 암호화 및 서명 검증
   . 액세스 토큰에 최소한의 정보만 포함 (사용자 ID, 권한 등)
   . 주기적인 리프레시 토큰 순환 (예: 7일마다)

이러한 개선된 인증 방식은 A고객사의 채널 확대 및 사용자 증가에 따른 인증 및 세션 관리 문제를 효과적으로 해결할 수 있을 것입니다. 시스템의 확장성이 향상되고, 보안성도 강화되며, 다양한 클라이언트 환경에서도 일관된 사용자 경험을 제공할 수 있습니다.

인증 불일치 또는 Session 만료에 따른 처리에 대한 UI 처리 방안을 설계하시오

인증 불일치 또는 세션 만료는 사용자 경험에 큰 영향을 미칠 수 있습니다. 따라서 이러한 상황이 발생했을 때 사용자에게 명확한 안내와 사용성을 보장하는 UI 처리 방안을 마련하는 것이 중요합니다.

1. 세션 만료 처리

a) 세션 만료 전 사용자 알림

- **세션 만료 경고 메시지**: 세션이 만료되기 몇 분 전(예: 5분 전)에 사용자에게 세션이 곧 만료될 것임을 알려주는 경고 메시지를 표시합니다.
  - **UI 요소**: 모달 창 또는 화면 상단에 바 형태의 알림 배너를 사용합니다.
  - **내용**: "세션이 5분 후 만료됩니다. 계속 사용하시려면 '연장' 버튼을 클릭하세요."
  - **액션**: 연장 버튼을 클릭하면 세션이 연장되고, 알림이 사라집니다.

b) 세션 만료 후 처리

- **세션 만료 알림 모달**: 사용자가 세션 만료 후 페이지 내에서 활동을 시도할 때, 즉시 세션 만료 알림 모달 창을 표시합니다.
  - **UI 요소**: 모달 창
  - **내용**: "세션이 만료되었습니다. 계속 사용하시려면 다시 로그인해주세요."
  - **액션**: 로그인 버튼을 클릭하면 로그인 페이지로 리디렉션됩니다. 이전에 작업하던 내용을 복구할 수 있는 옵션을 제공할 수 있습니다.
- **자동 로그아웃 후 리디렉션**: 세션이 만료되면 자동으로 로그아웃 처리되고, 로그인 페이지로 리디렉션됩니다.
  - **UI 요소**: 로그인 페이지로의 자동 리디렉션
  - **내용**: 로그인 페이지에 "세션이 만료되었습니다. 다시 로그인해주세요."라는 메시지를 표시합니다.
  - **액션**: 로그인 후, 사용자가 이전에 있던 페이지나 작업을 이어서 할 수 있도록 리디렉션합니다.

2. 인증 불일치 처리

a) 인증 불일치 발생 시 알림

- **즉각적인 오류 메시지**: 사용자가 잘못된 자격 증명(예: 비밀번호, 2FA 코드)을 입력했을 때, 즉시 오류 메시지를 표시합니다.
  - **UI 요소**: 입력 필드 아래 또는 모달 창으로 오류 메시지 표시
  - **내용**: "입력하신 비밀번호가 일치하지 않습니다. 다시 시도해주세요."
  - **액션**: 입력 필드가 강조되며, 사용자가 쉽게 다시 입력할 수 있도록 처리합니다. 일정 횟수 이상 오류 발생 시 CAPTCHA 또는 추가 보안 절차를 요구할 수 있습니다.

b) 인증이 필요할 때

- **재인증 요구 모달**: 사용자가 중요 작업(예: 결제, 민감한 데이터 접근 등)을 수행할 때 인증이 필요할 경우, 재인증을 요구하는 모달 창을 표시합니다.
  - **UI 요소**: 모달 창
  - **내용**: "이 작업을 수행하기 위해 다시 인증이 필요합니다. 비밀번호를 입력해주세요."
  - **액션**: 비밀번호 입력 후, 인증이 성공하면 작업을 계속 진행합니다. 인증 실패 시, 오류 메시지와 함께 다시 입력할 수 있도록 합니다.

3. 일관된 사용자 경험 제공

a) 일관된 디자인 및 메시지

- **통일된 UI 스타일**: 세션 만료나 인증 불일치 상황에서 표시되는 모든 메시지와 알림은 일관된 디자인과 언어를 사용하여 혼란을 최소화합니다.
  - **명확한 가이드 제공**: 사용자에게 필요한 조치를 명확하게 안내하고, 쉽게 이해할 수 있도록 간결한 언어로 설명합니다.

b) 접근성 및 반응성 고려

- **모든 기기에서 동일한 사용자 경험 제공**: 데스크톱, 태블릿, 모바일 등 다양한 기기에서 동일한 UI 경험을 제공하도록 반응형 디자인을 적용합니다.
  - **접근성(Accessibility)**: UI 요소가 스크린 리더와 호환되며, 키보드 및 보조 장치로 쉽게 접근할 수 있도록 설계합니다.

4. 사용자 피드백 및 개선

a) 사용자 피드백 루프

- **피드백 수집 기능 제공**: 세션 만료 또는 인증 불일치가 발생한 후 사용자가 문제를 보고하거나, UI에 대한 피드백을 제공할 수 있는 기능을 제공합니다.
  UI 요소: 피드백 버튼 또는 폼
  내용: "이 문제가 반복되나요? 피드백을 남겨주세요."

b) 지속적인 UI 개선

- **사용자 피드백 기반 개선**: 수집된 피드백을 분석하여 UI/UX를 지속적으로 개선합니다.

결론

다수의 사용자가 다수의 인벤토리 예약을 하기 위한 데이터 모델의 문제점을 확인하고, 해결을 위한 방안을 제시하시오. 예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

1. 문제 정의
다수의 사용자가 동일한 인벤토리(예: 항공편 좌석, 호텔 객실, 상품 재고 등)를 동시에 예약하려고 할 때, 데이터 모델과 예약 처리 프로세스에서 동시성 문제가 발생할 수 있습니다.
이로 인해 중복 예약, 데이터 불일치, 예약 실패 등이 발생할 수 있습니다.

2. 문제점 분석
동시성 이슈는 주로 다음과 같은 문제점에서 발생합니다

- **레이스 컨디션(Race Condition)**:
여러 사용자가 동시에 같은 인벤토리 아이템을 예약하려고 할 때, 각 프로세스가 동시에 데이터베이스에 접근하여 데이터 일관성을 깨뜨리는 문제가 발생할 수 있습니다.

- **잠금 문제(Locking Issues)**:
데이터베이스에서 동시성 제어를 위해 레코드를 잠그는 방식이 비효율적이거나, 잠금 해제가 제대로 이루어지지 않아 데드락(Deadlock) 또는 다른 사용자 요청이 차단되는 문제가 발생할 수 있습니다.

- **비효율적인 데이터 모델**
인벤토리의 상태를 제대로 반영하지 못하는 데이터 모델로 인해 동시성 제어가 어려워질 수 있습니다. 특히 비동기 방식으로 구현된 경우, 동시성 제어가 더욱 어렵습니다.

3. 해결 방안 제시
동시성 문제를 해결하기 위한 방안을 제시하며, 제약 조건(비동기 처리)을 충족하도록 설계합니다.

■ 방안 1: Optimistic Locking (낙관적 잠금)

1. 버전 관리 추가
- 예약 테이블에 version 필드를 추가하여 각 예약 요청 시점의 상태를 추적합니다.
- 업데이트 시 해당 레코드의 version 값이 현재와 일치하는 경우에만 업데이트를 허용하며, 불일치 시 재시도를 요청하거나 오류를 반환합니다.

2. 비동기 처리
- 예약 요청은 비동기적으로 처리되며, 업데이트 요청 시 Optimistic Locking을 통해 동시성 문제를 해결합니다.
- 클라이언트는 실패한 경우 이를 인지하고, 다시 시도할 수 있도록 설계합니다.

■ 방안 2: Pessimistic Locking (비관적 잠금)

1. 레코드 잠금:
- 사용자가 특정 인벤토리를 예약하려고 할 때 해당 인벤토리 레코드를 읽기 전에 잠급니다.
이는 다른 사용자가 동시에 해당 레코드를 읽거나 수정하지 못하도록 합니다.
- Pessimistic Locking은 일반적으로 더 높은 동시성 제어를 제공하지만, 성능에 부정적인 영향을 미칠 수 있습니다.

2. 비동기 큐 사용:
- 예약 요청을 비동기 큐(예: RabbitMQ, Kafka)에 넣고, 예약 처리를 하나씩 큐에서 꺼내어 처리합니다.
이렇게 하면 비관적 잠금으로 인한 성능 저하를 완화할 수 있습니다.

다수의 사용자가 다수의 인벤토리 예약을 하기 위한 데이터 모델의 문제점을 확인하고, 해결을 위한 방안을 제시하시오. 예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

■ 방안 3: 데이터 모델 개선 및 분산 시스템 도입

1. 재고 감소 시나리오
- 재고를 관리하는 인벤토리 테이블과 예약 테이블을 분리하고, 인벤토리 테이블에서 available_quantity를 관리합니다.
- 예약 요청이 들어올 때마다 비동기적으로 재고를 감소시키는 요청을 보내고, 성공 시 예약을 완료합니다.

2. CQRS 패턴
- Command Query Responsibility Segregation(CQRS) 패턴을 도입하여, 쓰기 작업(예약 및 재고 감소)과 읽기 작업(인벤토리 상태 조회)을 분리합니다.
- 예약은 비동기적으로 처리되며, 쓰기 작업이 완료되면 이벤트를 통해 읽기 모델을 업데이트합니다.

3. 분산 트랜잭션 관리
- 분산 시스템(예: Redis, Cassandra)에서 트랜잭션을 관리하여, 각 노드에서의 동시성 문제를 해결합니다.
  특히, Redis의 WATCH 및 MULTI/EXEC 명령어를 사용해 낙관적 잠금을 구현할 수 있습니다.

■ 방안 4: 이벤트 소싱(Event Sourcing)

1. 이벤트 기반 예약 처리
- 예약 요청을 이벤트로 처리하여, 예약이 완료될 때까지의 모든 상태 변화를 이벤트로 기록합니다.
- 이벤트 소싱을 사용하면 모든 이벤트를 재생하여 시스템의 현재 상태를 복원할 수 있으며, 동시성 문제 발생 시 롤백 또는 재 처리할 수 있습니다.

2. Saga 패턴
- 분산 트랜잭션을 관리하기 위해 Saga 패턴을 도입하여, 각 트랜잭션 단계를 독립적으로 관리하고, 실패 시 롤백 작업을 정의합니다.

4. 프로세스 설계

예약 요청 프로세스(Optimistic Locking 사용 예시)
Step 1: 예약 요청 수신
→ 사용자가 예약 요청을 서버로 보냅니다.
Step 2: 데이터베이스에서 레코드 조회
→ 예약하려는 인벤토리의 현재 상태 및 version을 조회합니다.
Step 3: 예약 처리
→ 비동기적으로 예약을 처리하며, version 필드가 현재 상태와 일치하는지 확인합니다.
일치하지 않으면 실패 메시지를 반환하고, 클라이언트는 다시 시도할 수 있습니다.
Step 4: 성공 시 예약 확정
→ version 필드를 증가시키고, 인벤토리 상태를 업데이트한 후, 예약을 확정합니다.
Step 5: 클라이언트에 응답 반환
→ 예약 성공 또는 실패 여부를 클라이언트에 반환합니다.

데이터 모델 설계:

sql — Copy code

```sql
-- 인벤토리 테이블
CREATE TABLE Inventory (
    id SERIAL PRIMARY KEY,
    item_name VARCHAR(255),
    available_quantity INT,
    version INT DEFAULT 0
);

-- 예약 테이블
CREATE TABLE Reservation (
    id SERIAL PRIMARY KEY,
    user_id INT,
    inventory_id INT,
    quantity INT,
    status VARCHAR(50),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_inventory FOREIGN KEY (inventory_id) REFERENCES Inventory(id)
);
```

다수의 사용자가 하나의 인벤토리 예약을 하기 위한 데이터 모델의 문제점을 확인하고, 해결을 위한 방안을 제시하시오. 예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

■ 문제 정의 및 분석

다수의 사용자가 동시에 하나의 인벤토리를 예약하려고 할 때, 데이터 모델에서 발생할 수 있는 주요 문제는 다음과 같습니다

1. 레이스 컨디션 (Race Condition)
여러 사용자가 동시에 동일한 인벤토리 아이템을 예약하려고 할 때, 데이터베이스에서 동일한 자원을 동시에 수정하려는 시도로 인해 데이터 일관성이 깨질 수 있습니다.
예를 들어, 동일한 좌석이나 상품이 두 명 이상의 사용자에게 동시에 예약되는 경우가 발생할 수 있습니다.

2. 데이터 무결성 손상
동시에 여러 요청이 처리될 때, 인벤토리의 가용 수량이 정확하게 관리되지 않을 수 있습니다. 결과적으로, 실제로는 가용하지 않은 자원을 예약하는 상황이 발생할 수 있습니다.

3. 비효율적인 비동기 처리
비동기 방식으로 예약 프로세스를 처리할 때, 데이터 무결성을 유지하면서 효율적으로 인벤토리를 관리하는 것이 어려울 수 있습니다.

■ 해결 방안

이를 해결하기 위해 데이터 모델과 비동기 예약 처리 프로세스에 다음과 같은 방안을 제시합니다:

1. 데이터 모델 개선

a) 예약 상태 및 잠금 필드 추가

- 잠금 상태 필드 (lock_status) 추가
  . 인벤토리 테이블에 lock_status 필드를 추가하여, 해당 인벤토리가 현재 예약 처리 중인지 여부를 나타냅니다.
  . 이 필드는 예약 처리 중인 인벤토리를 잠그는 역할을 하며, 다른 사용자가 해당 인벤토리를 동시에 예약하지 못하도록 합니다.
- 예약 상태 필드 (reservation_status) 추가
  . 예약 테이블에 reservation_status 필드를 추가하여, 예약 요청이 성공적으로 완료되었는지 여부를 기록합니다.
  . 가능한 상태값: PENDING, CONFIRMED, FAILED

```sql
-- 인벤토리 테이블
CREATE TABLE Inventory (
    id SERIAL PRIMARY KEY,
    item_name VARCHAR(255),
    available_quantity INT,
    lock_status BOOLEAN DEFAULT FALSE, -- 잠금 상태 필드
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 예약 테이블
CREATE TABLE Reservation (
    id SERIAL PRIMARY KEY,
    user_id INT,
    inventory_id INT,
    quantity INT,
    reservation_status VARCHAR(50) DEFAULT 'PENDING', -- 예약 상태 필드
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_inventory FOREIGN KEY (inventory_id) REFERENCES Inventory(id)
);
```

다수의 사용자가 하나의 인벤토리 예약을 하기 위한 데이터 모델의 문제점을 확인하고, 해결을 위한 방안을 제시하시오. 예약 처리 프로세스는 Async 한 방식으로 구현 되어야 한다

2. 비동기 예약 처리 프로세스

a) 낙관적 잠금 (Optimistic Locking) 사용

- 버전 관리 필드 추가:
  . version 필드를 인벤토리 테이블에 추가하여, 각 예약 요청이 처리될 때 이 필드를 사용해 낙관적 잠금을 구현합니다.
  . 각 트랜잭션이 완료될 때마다 version 필드를 증가시키고, 예약 요청 시 현재 version 값을 검증하여 데이터 무결성을 유지합니다.

b) 비동기 메시지 큐 활용

- 메시지 큐 도입:
  . 예약 요청을 처리하기 위해 비동기 메시지 큐(예: RabbitMQ, Apache Kafka)를 사용합니다.
  모든 예약 요청은 메시지 큐에 쌓이고, 워커 프로세스가 큐에서 메시지를 하나씩 처리합니다.
  . 워커는 인벤토리 잠금 상태를 확인하고, 잠겨있지 않으면 인벤토리를 잠근 후 예약을 처리합니다.

c) 트랜잭션 처리 및 재시도 로직

- 트랜잭션 관리:
  . 인벤토리 예약과 관련된 모든 작업은 트랜잭션으로 처리되어야 합니다. 예를 들어, 인벤토리 잠금, 예약 상태 업데이트, 인벤토리 수량 감소 등이 하나의 트랜잭션으로 묶여야 합니다.
  . 트랜잭션이 성공하면 reservation_status를 CONFIRMED로 업데이트하고, 실패 시 FAILED로 표시합니다.
- 재시도 로직:
  . 만약 예약이 실패하거나 트랜잭션 충돌이 발생할 경우, 일정 횟수만큼 자동으로 재시도하는 로직을 구현합니다.
  . 재시도 횟수를 초과하면, 사용자에게 예약 실패를 알리고, 다른 인벤토리를 선택하도록 안내할 수 있습니다.

3. 예약 처리 흐름 예시

Step 1: 예약 요청 수신
→ 사용자가 예약 요청을 보냅니다.
Step 2: 예약 요청 큐에 추가
→ 예약 요청은 메시지 큐에 추가됩니다.
Step 3: 예약 처리 시작
→ 워커 프로세스가 큐에서 메시지를 읽어 예약 처리를 시작합니다.
Step 4: 인벤토리 잠금 시도
→ 워커는 인벤토리의 lock_status 필드를 확인하여 잠금을 시도합니다.
이미 잠금이 되어 있으면 재시도를 위해 예약 요청을 다시 큐에 추가합니다.
Step 5: 예약 처리
→ 잠금에 성공하면, 인벤토리의 available_quantity를 감소시키고, 예약 상태를 PENDING에서 CONFIRMED로 업데이트합니다.
version 필드를 사용해 낙관적 잠금을 검증합니다.
Step 6: 잠금 해제
→ 예약 처리가 완료되면 인벤토리의 lock_status를 FALSE로 변경하여 잠금을 해제합니다.
Step 7: 응답 반환
→ 예약이 성공적으로 완료되면 사용자에게 성공 메시지를 반환합니다. 실패한 경우 재시도를 안내합니다.

선착순 10,000명에게만 음악 스트리밍 상품 구매시 50% 할인을 적용하려고 합니다. 모든 사용자가 동시에 상품을 구매하려고 할 때 발생할 수 있는 동시성 문제를 해결하는 시스템을 구성

■ **아키텍처 개요**

1. 웹 서버: 사용자 요청 처리
2. 애플리케이션 서버: 비즈니스 로직 처리
3. 메시지 큐: 구매 요청 관리
4. 데이터베이스: 사용자 정보 및 구매 정보 저장
5. 캐시 서버: 할인 카운터 관리

■ **상세 구현**

a. 할인 카운터 관리
   1. Redis를 사용하여 원자적 카운터 구현
   2. INCR 명령어로 카운터 증가 및 현재 값 확인

b. 구매 프로세스:
   1. 사용자가 구매 요청
   2. 웹 서버에서 요청 접수 및 유효성 검사
   3. Redis의 카운터 확인 (INCR 사용)
   4. 카운터가 10,000 이하면 구매 요청을 메시지 큐에 전송
   5. 메시지 큐에서 순차적으로 구매 처리

c. 메시지 큐 처리:
   1. Apache Kafka 또는 RabbitMQ 사용
   2. 구매 요청을 순차적으로 처리하여 동시성 문제 해결

d. 데이터베이스 처리:
   1. 트랜잭션을 사용하여 구매 정보 저장
   2. 낙관적 락킹으로 동시 수정 방지

■ **추가 고려사항:**

a. 부하 분산: 로드 밸런서를 사용하여 다수의 웹 서버에 요청 분산

b. 응답 시간 개선: 비동기 처리로 사용자에게 빠른 응답 제공
   구매 결과는 웹소켓 또는 폴링으로 전달

c. 장애 대비 : 메시지 큐의 내구성 설정으로 데이터 유실 방지
   주요 컴포넌트의 다중화로 가용성 확보

d. 모니터링 및 로깅: 실시간 모니터링 시스템 구축
   상세 로깅으로 문제 발생 시 빠른 대응

코드 예시 (의사 코드):

Python

```
import redis
import kafka

redis_client = redis.Redis(host='localhost', port=6379)
kafka_producer = kafka.KafkaProducer(bootstrap_servers=['localhost:9092'])

def process_purchase(user_id):
    # 1. 카운터 증가 및 확인
    current_count = redis_client.incr('discount_counter')

    if current_count <= 10000:
        # 2. 할인 적용 대상
        kafka_producer.send('purchase_requests', f"{user_id},discount")
        return "구매 요청이 접수되었습니다. 할인이 적용됩니다."
    else:
        # 3. 할인 미적용
        kafka_producer.send('purchase_requests', f"{user_id},no_discount")
        return "구매 요청이 접수되었습니다. 할인이 적용되지 않습니다."

# Kafka 컨슈머에서 구매 요청 처리
def process_kafka_messages():
    consumer = kafka.KafkaConsumer('purchase_requests',
                                   bootstrap_servers=['localhost:9092'])
    for message in consumer:
        user_id, discount_status = message.value.decode().split(',')
        process_purchase_in_db(user_id, discount_status == 'discount')

def process_purchase_in_db(user_id, apply_discount):
    # 데이터베이스에 구매 정보 저장 (트랜잭션 사용)
    # 할인 적용 여부에 따라 처리
    pass
```

이 시스템은 Redis의 원자적 연산과 Kafka의 메시지 큐를 활용하여 동시성 문제를 효과적으로 해결합니다. 또한, 비동기 처리를 통해 대규모 동시 요청을 원활히 처리할 수 있습니다..

선착순 10,000명에게만 음악 스트리밍 상품 구매시 50% 할인을 적용하려고 합니다. 모든 사용자가 동시에 상품을 구매하려고 할 때 발생할 수 있는 동시성 문제를 해결하는 시스템을 구성

1. 동시성이 발생하는 이유

레이스 컨디션 (Race Condition)

여러 사용자가 동시에 동일한 인벤토리 아이템을 예약하려고 할 때, 데이터베이스에서 동일한 자원을 동시에 수정하려는 시도로 인해 데이터 일관성이 깨질 수 있습니다.
예를 들어, 동일한 좌석이나 상품이 두 명 이상의 사용자에게 동시에 예약되는 경우가 발생할 수 있습니다.

2. 해결 - Redis 싱글스레드

멀티스레드인 Java와는 다르게 Redis는 기본적으로 싱글 스레드 모델을 사용합니다.
그래서 저는 해결방법을 고민하다가 쿠폰 갯수를 증가시키는 작업을 Redis를 사용하여 싱글 스레드는
모든 명령어가 순차적으로 실행되도록 보장하기 때문에,
복잡한 동시성 제어 로직 없이도 데이터의 일관성을 유지하기 위해 Redis를 사용하기로 결정했습니다.

의존성 추가

implementation 'org.springframework.boot:spring-boot-starter-data-redis'
먼저 redis를 사용하기 위해 , 의존성을 추가해줍니다.

3. 문제 발생 - RDB 성능 저하

저는 db를 AWS RDS(mysql)를 사용중이였습니다.
Redis를 통해 동시성 제어에 관한 이슈는 해결했지만, AWS를 모니터링 해보니 요청시점에
RDB의 cpu 사용률이 너무 높아져 있었습니다..

4. 해결 - Kafka

선착순 쿠폰 발급 요청에 동시에 여러 요청이 오면 rdb에 부담이 가므로 다른 서비스 지연 및 성능에
대한 저하가 올수도있음 이를 해결하기위해 kafka를 통해 해결하기 했습니다

1. 처리량 조절
   - 처리량 조절: Kafka 는 토픽에 있는 데이터를 순차적으로 가져와서 처리하게 됩니다.
   Consumer 가 1개가 있고 토픽에 데이터가 100개가 있다고 가정할때 Consumer 에서는 1번 데이터를
   가져와서 처리가 완료되면 2번 데이터를 가져와서 처리합니다.

   10:00시에 100명의 유저가 1번씩 요청을 보내게 됐을 때 API 에서 직접 처리를 한다면
   데이터베이스에 100번의 요청이 한번에 몰리게 될 것입니다.

   하지만 이 요청을 카프카 프로듀서를 이용하여 토픽에 전송하고 Consumer 를 이용하여 데이터를
   처리한다면 데이터베이스에 요청하는 양을 조절할 수 있게 되므로 DB의 부하를 줄일 수 있습니다.

2. 부하 분산
   요청의 버퍼링: 카프카는 높은 처리량을 지원하는 분산 스트리밍 플랫폼으로, 대량의 요청을 효율적으로
   버퍼링할 수 있습니다. 이를 통해 순간적인 트래픽 급증 시에도 요청을 안정적으로 관리할 수 있습니다.
   스케일 아웃: 카프카 클러스터는 수평적으로 확장 가능합니다.
   따라서 요청의 양이 증가함에 따라 더 많은 컨슈머를 동적으로 추가하여 처리량을 증가시킬 수 있습니다.

개선된 코드

```
@Repository
public class CouponCountRepository {
    private final RedisTemplate<String, String> redisTemplate;

    public CouponCountRepository(RedisTemplate<String, String> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    public Long increment() {
        return redisTemplate
                .opsForValue()
                .increment("coupon_count");
    }

}
```

`RedisTemplate을` 사용하여 쿠폰 갯수를 1씩 증가시켜줬습니다.

```
public void applyCoupon(CouponRequestDto couponRequestDto) {
        Long count = couponCountRepository.increment();
        //오늘 날짜 기준으로 100개보다 많으면 return
        if (count > 100) {
            return;
        }
        Coupon coupon=Coupon.builder()
                        .memberNo(couponRequestDto.getMemberNo())
                        .tradeBoardNo(couponRequestDto.getTradeBoardNo())
                        .discountPrice(couponRequestDto.getDiscountPrice())
                        .regDate(LocalDateTime.now())
                        .build();

        couponRepository.save(coupon);
    }
```

원격서비스가 자주 타임아웃이 발생하여 오류가 발생합니다. 서비스의 정상적인 응답을 보장하기 위하여 방안을 제안하세요.

1. 서킷 브레이커 패턴(Circuit Breaker Pattern) 도입

a) 서킷 브레이커의 작동 방식

- 서킷 브레이커는 원격 서비스 호출 시 일정 시간 동안 오류가 계속 발생하면 더 이상 호출을 시도하지 않고, 대신 오류를 즉시 반환하는 패턴입니다.
  이는 시스템이 지속적으로 실패하는 서비스에 의존하여 리소스를 낭비하지 않도록 합니다.
- 서킷 브레이커는 다음과 같은 세 가지 상태로 작동합니다:
  . **Closed**: 정상적으로 서비스 호출을 허용하는 상태입니다. 일정 수의 호출이 실패하면 Open 상태로 전환됩니다.
  . **Open**: 서비스 호출을 차단하고, 실패를 즉시 반환하는 상태입니다. 일정 시간이 지나면 Half-Open 상태로 전환됩니다.
  . **Half-Open**: 서비스가 복구되었는지 테스트하기 위해 일부 요청을 허용하는 상태입니다. 테스트가 성공하면 Closed 상태로 전환되고, 실패하면 다시 Open 상태로 전환됩니다.

b) 서킷 브레이커 구현 방법

. **Hystrix, Resilience4j**와 같은 라이브러리를 사용하여 서킷 브레이커 패턴을 쉽게 구현할 수 있습니다.
. 각 서비스 호출에 대해 서킷 브레이커를 적용하여, 일정 시간 동안 지속적으로 타임아웃이 발생하는 경우 서비스 호출을 중단하고, 대신 폴백 메커니즘을 사용합니다.

2. 타임아웃 및 재시도 메커니즘 설정

a) 적절한 타임아웃 설정

- 원격 서비스에 대한 호출 시 타임아웃 시간을 적절하게 설정합니다. 타임아웃이 너무 길면 사용자가 응답을 기다리면서 시스템 자원이 낭비될 수 있고,
  너무 짧으면 일시적인 네트워크 지연에도 불필요한 오류가 발생할 수 있습니다.

b) 재시도 메커니즘 도입

- 타임아웃이 발생했을 때, 즉시 실패로 처리하지 않고 일정 횟수만큼 재시도를 시도합니다.
- Exponential Back off를 사용하여 재시도 간격을 점진적으로 늘려 네트워크의 혼잡도를 줄이고, 서비스가 복구될 시간을 제공할 수 있습니다.

3. 폴백(Fallback) 메커니즘

a) 대체 응답 제공

. 원격 서비스가 타임아웃으로 인해 실패할 경우, 기본적으로 폴백 메커니즘을 사용하여 대체 응답을 제공할 수 있습니다.
. 폴백은 캐시된 데이터나 기본값을 반환하거나, 간단한 에러 메시지를 반환할 수 있습니다. 이를 통해 사용자에게 최소한의 응답을 보장할 수 있습니다.

4. 캐싱 전략 도입

a) 결과 캐싱

. 원격 서비스의 응답 데이터를 캐싱하여 동일한 요청에 대해 원격 서비스를 반복 호출하지 않도록 합니다.
. 캐시 만료 시간을 설정하여, 일정 시간 동안은 캐시된 응답을 사용하고, 만료되면 다시 원격 서비스를 호출합니다. 이를 통해 원격 서비스에 대한 의존성을 줄일 수 있습니다.

5. 부하 분산 및 스케일링

a) 로드 밸런싱 - 원격 서비스 앞 단에 로드 밸런서를 배치하여 트래픽을 여러 인스턴스에 분산시킵니다. 이를 통해 특정 인스턴스에 과부하가 걸리는 것을 방지할 수 있습니다.

b) 자동 스케일링(Auto-Scaling) - 원격 서비스의 인스턴스를 자동으로 확장하거나 축소하여, 트래픽 증가 시에도 서비스의 가용성을 유지합니다.
AWS의 Auto Scaling, Kubernetes의 Horizontal Pod Autoscaling 등이 이를 지원합니다.

6. 모니터링 및 알림 시스템 강화

a) 실시간 모니터링 : 원격 서비스의 성능을 실시간으로 모니터링 하여, 타임아웃 발생 빈도와 응답 시간을 추적합니다.
Prometheus, Grafana 같은 도구를 사용하여 지표를 시각화하고, 문제가 발생할 때 신속히 감지할 수 있도록 합니다.

b) 알림 시스템 : 타임아웃이나 서비스 장애 발생 시 자동으로 알림을 보내는 시스템을 도입합니다.
PagerDuty, Opsgenie 등의 도구를 통해 운영팀에 즉시 통보하여 빠르게 대응할 수 있도록 합니다.

7. 서비스 수준 협약(SLA) 재검토

a) SLA 조정

- 원격 서비스 제공자와의 SLA를 재검토하고, 타임아웃 문제를 해결하기 위해 서비스 제공자의 응답 시간을 개선할 수 있는 방안을 협의합니다.
- 원격 서비스가 타임아웃 문제를 지속적으로 발생시킬 경우, 서비스 제공자 변경을 고려할 수 있습니다.

결론

이러한 방안을 통해 원격 서비스에서 발생하는 타임아웃 문제를 효과적으로 관리할 수 있습니다. 서킷 브레이커, 재시도 메커니즘, 폴백 전략을 조합하면 시스템의 복원력을 높이고, 사용자 경험을 개선할 수 있습니다. 또한, 캐싱, 로드 밸런싱, 모니터링 시스템을 강화하여 서비스의 안정성과 신뢰성을 확보할 수 있습니다.

원격서비스가 자주 타임아웃이 발생하여 오류가 발생합니다. 서비스의 정상적인 응답을 보장하기 위하여 방안을 제안하세요.

1. 타임아웃 설정 조정
- 클라이언트와 서버 양쪽의 타임아웃 설정을 검토하고 필요에 따라 증가시킵니다.
- 네트워크 지연이나 서비스 처리 시간을 고려하여 적절한 타임아웃 값을 설정합니다.

2. 로드 밸런싱 구현
- 여러 서버에 트래픽을 분산시켜 단일 서버의 부하를 줄입니다.
- 서버 간 요청을 균등하게 분배하여 응답 시간을 개선합니다.

3. 서비스 최적화
- 데이터베이스 쿼리 최적화
- 코드 리팩토링을 통한 성능 개선
- 캐싱 메커니즘 도입으로 반복적인 요청 처리 속도 향상

4. 서버 리소스 증설
- CPU, 메모리, 네트워크 대역폭 등 필요한 리소스를 증설합니다.
- 클라우드 환경에서는 오토스케일링을 구현하여 트래픽에 따라 자동으로 리소스를 조절합니다.

5. 비동기 처리 도입
- 시간이 오래 걸리는 작업은 비동기로 처리하고 결과를 나중에 전달합니다.
- 메시지 큐 시스템을 활용하여 작업을 분산 처리합니다.

6. 서킷 브레이커 패턴 적용
- 서비스 장애 시 빠르게 실패 처리하여 연쇄적인 타임아웃을 방지합니다.
- 일정 시간 후 자동으로 서비스 복구를 시도합니다.

7. 재시도 메커니즘 구현
- 일시적인 네트워크 문제로 인한 타임아웃 발생 시 자동으로 재시도합니다.
- 지수 백오프 알고리즘을 적용하여 효율적인 재시도 간격을 설정합니다.

8. 모니터링 및 로깅 강화:
- 실시간 모니터링 시스템을 구축하여 문제를 신속히 감지합니다.
- 상세한 로그를 남겨 문제 원인 분석과 해결을 용이하게 합니다.

9. 네트워크 최적화:
- CDN(Content Delivery Network) 사용으로 지리적 거리에 따른 지연 감소
- 네트워크 장비 및 설정 최적화

10. 서비스 분리 및 마이크로서비스 아키텍처 고려
- 큰 서비스를 작은 단위로 분리하여 각 기능별로 독립적인 확장 가능
- 특정 기능의 장애가 전체 시스템에 미치는 영향 최소화

전반적인 시스템 처리 성능이 지연되어, 스트리밍 상품 외 배송 상품 주문처리 시 주문 완료 시간이 지연되고 있습니다. 이를 개선하기 위한 비동기 처리 방안을 구성하세요

1. 문제 분석

배송 상품 주문 처리에서 성능 지연이 발생하는 주요 원인은 다음과 같습니다:

- **동기식 처리** : 주문 처리의 각 단계가 순차적으로 수행되어, 한 단계가 완료되기 전까지 다음 단계로 진행할 수 없는 구조로 인해 지연이 발생할 수 있습니다.
- **의존성 작업** : 결제, 재고 확인, 배송 준비 등 여러 작업이 하나의 트랜잭션 내에서 동기적으로 처리되기 때문에 병목이 발생할 수 있습니다.
- **자원 경쟁** : 동일한 자원을 두고 여러 프로세스가 경쟁하여 처리 속도가 느려질 수 있습니다.

2. 비동기 처리 방안 구성

a) 비동기 메시지 큐 도입

메시지 큐 사용 (예: RabbitMQ, Apache Kafka):

- 각 주문 처리 단계(결제 확인, 재고 확인, 배송 준비, 알림 전송 등)를 독립적인 서비스로 분리하고, 이들 간의 통신을 메시지 큐를 통해 비동기적으로 처리합니다.
- 주문 생성 시, 주문 요청을 메시지 큐에 넣고, 각 서비스가 큐에서 메시지를 가져와 비동기적으로 작업을 수행합니다.
- 메시지 큐를 사용하면 각 단계가 병렬로 처리될 수 있으며, 특정 단계에서 지연이 발생해도 전체 주문 처리 속도에 큰 영향을 미치지 않습니다.

메시지 큐 처리 흐름:

. 주문 생성: 사용자가 주문을 제출하면 주문 요청이 메시지 큐에 추가됩니다.
. 결제 처리: 결제 서비스가 큐에서 주문 메시지를 가져와 비동기적으로 결제를 처리하고, 성공 또는 실패 결과를 메시지 큐에 다시 보냅니다.
. 재고 확인: 재고 서비스가 결제 성공 메시지를 받아 비동기적으로 재고를 확인합니다. 재고가 충분하지 않은 경우, 주문 취소 메시지를 큐에 추가합니다.
. 배송 준비: 재고가 충분하면 배송 준비를 시작하고, 배송 정보를 큐에 추가하여 사용자에게 알림을 보냅니다.

b) 이벤트 기반 아키텍처

이벤트 소싱(Event Sourcing):

. 주문 처리의 각 단계에서 발생하는 이벤트를 캡처하여, 이벤트 기반으로 시스템을 구성합니다.
. 예를 들어, 결제 완료, 재고 확인 완료, 배송 준비 완료와 같은 이벤트를 트리거로 다음 단계를 비동기적으로 처리합니다.

CQRS 패턴 적용:

. 명령(Command)과 조회(Query) 작업을 분리하여, 주문 처리를 위한 명령은 비동기적으로 처리하고, 조회는 독립적으로 수행합니다.
. CQRS 패턴을 사용하면 명령 처리(주문, 결제, 재고 확인 등)를 비동기적으로 처리하여 시스템의 처리 속도를 개선할 수 있습니다.

c) 작업 풀 및 백그라운드 작업 처리

백그라운드 작업 처리 (Worker Pools):

. 주문 처리 중 시간이 많이 소요되는 작업(예: 외부 시스템과의 통신, 대량 데이터 처리 등)을 백그라운드 작업으로 분리하여 비동기적으로 처리합니다.
. Worker 풀을 사용하여 이러한 작업을 병렬로 처리하고, 시스템의 다른 부분에 영향을 주지 않도록 합니다.

백그라운드 작업 처리 흐름:

. 주문 생성 시, 외부 시스템과 통신하거나 대량 데이터를 처리하는 작업을 백그라운드 작업 큐에 넣습니다.
. 백그라운드 워커가 큐에서 작업을 가져와 처리하고, 완료 후 결과를 기록하거나, 알림 메시지를 큐에 추가합니다.
. 주문 처리는 주문이 생성된 시점에서 완료된 것으로 간주하고, 이후의 작업은 비동기적으로 완료됩니다.

전반적인 시스템 처리 성능이 지연되어, 스트리밍 상품 외 배송 상품 주문처리 시 주문 완료 시간이 지연되고 있습니다. 이를 개선하기 위한 비동기 처리 방안을 구성하세요

d) 캐싱 및 로드 밸런싱

캐싱:

. 자주 참조되는 데이터(예: 제품 정보, 배송 옵션, 재고 상태 등)를 캐시하여 반복적인 데이터베이스 접근을 줄이고 응답 시간을 개선합니다.

. Redis, Memcached와 같은 인메모리 캐시를 사용하여 빠른 조회 성능을 제공합니다.

로드 밸런싱:

. 주문 처리 요청을 여러 인스턴스로 분산하여 시스템의 부하를 균등하게 분산시킵니다.

. 로드 밸런서를 사용하여 각 서비스의 인스턴스에 부하가 집중되지 않도록 합니다.

3. 모니터링 및 성능 최적화

모니터링 시스템 도입:

. Prometheus, Grafana 등을 사용하여 주문 처리 시스템의 성능 지표(처리 시간, 큐 길이, 오류율 등)를 실시간으로 모니터링합니다.

. 비동기 처리 중 발생하는 오류를 추적하고, 병목 현상이 발생하는 부분을 식별하여 최적화할 수 있습니다.

자동 확장(Auto-Scaling):

. 주문 처리량이 급증할 경우, 시스템이 자동으로 확장되도록 설정합니다.
Kubernetes의 Horizontal Pod Autoscaling(HPA) 등을 활용하여 워커 인스턴스를 자동으로 증가시킬 수 있습니다.

결론

이러한 비동기 처리 방안을 통해 주문 처리 시스템의 성능을 크게 개선할 수 있습니다.
메시지 큐, 이벤트 기반 아키텍처, 백그라운드 작업 처리, 캐싱 및 로드 밸런싱을 통해 각 주문 처리 단계를 독립적으로 최적화하고, ₩
동시 처리 능력을 강화할 수 있습니다. 이를 통해 주문 완료 시간을 단축하고, 시스템의 안정성과 확장성을 높일 수 있습니다.

예약 위주의 상품 관련 데이터 모델에 실시간 판매 가능한 상품 유형을 추가 하기 위한 방안을 제안하고 설계 하시오

실시간 판매 가능한 상품 유형 추가를 위한 데이터 모델 설계

예약 위주의 상품 관련 데이터 모델에 실시간 판매 가능한 상품 유형을 추가하려면 기존 예약 시스템과 실시간 판매 시스템이 모두 효율적으로 관리될 수 있는 구조를 설계해야 합니다.
이 설계는 예약 상품과 실시간 판매 상품의 특성을 모두 고려하며, 데이터베이스의 확장성과 성능을 유지할 수 있도록 해야 합니다.

1. 개념 설계

a) 상품 유형의 구분

. 예약 상품 : 사전 예약이 필요한 상품으로, 사용자는 특정 시간이나 날짜에 사용할 수 있도록 예약을 진행합니다. 예: 호텔 예약, 항공권 예약.

. 실시간 판매 상품 : 실시간으로 구매할 수 있는 상품으로, 사용자는 즉시 구매하여 사용할 수 있습니다. 예: 스트리밍 서비스, 디지털 다운로드, 실시간 재고 기반의 물리적 상품.

b) 공통 상품 속성

- **Product** : 공통 속성을 정의하는 기본 테이블로, 모든 상품은 이 테이블에서 관리됩니다.
  . product_id: 고유 상품 ID
  . product_name: 상품명
  . product_type: 상품 유형(예약 상품 또는 실시간 판매 상품)
  . price: 상품 가격
  . available_quantity: 가용 수량 (실시간 판매 상품에만 적용)

2. 데이터 모델 설계

a) 기본 테이블

```
CREATE TABLE Product (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    product_type VARCHAR(50) NOT NULL CHECK (product_type IN ('RESERVABLE', 'REALTIME')),
    price DECIMAL(10, 2) NOT NULL,
    available_quantity INT DEFAULT NULL, -- 실시간 판매 상품에 적용
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

product_type 필드를 사용해 상품이 예약 가능 상품인지, 실시간 판매 가능한 상품인지를 구분합니다.
available_quantity 필드는 실시간 판매 상품에 대해서만 관리되며, 예약 상품의 경우 NULL 값이 설정됩니다.

예약 위주의 상품 관련 데이터 모델에 실시간 판매 가능한 상품 유형을 추가 하기 위한 방안을 제안하고 설계 하시오

b) 예약 상품 관련 테이블

```
CREATE TABLE Reservation (
    reservation_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    user_id INT NOT NULL,
    reservation_date DATE NOT NULL,
    reservation_status VARCHAR(50) DEFAULT 'PENDING',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id)
);
```

Reservation 테이블은 예약 상품에 대한 예약 정보를 관리합니다.
각 예약은 특정 날짜에 대해 예약 상태를 포함합니다. 예약 상태는 'PENDING', 'CONFIRMED', 'CANCELED' 등이 될 수 있습니다.

c) 실시간 판매 상품 관련 테이블

```
CREATE TABLE RealTimeSale (
    sale_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    user_id INT NOT NULL,
    quantity INT NOT NULL,
    sale_status VARCHAR(50) DEFAULT 'COMPLETED',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id)
);
```

RealTimeSale 테이블은 실시간 판매 상품의 거래 내역을 관리합니다.
quantity 필드는 실시간 판매 상품이 판매된 수량을 기록하며, sale_status는 판매 상태(예: 'COMPLETED', 'CANCELED')를 관리합니다.

3. 비즈니스 로직 및 처리 흐름

a) 실시간 판매 처리

- 재고 확인: 사용자가 실시간 판매 상품을 구매하려고 할 때, available_quantity를 확인합니다.
- 재고 감소: 구매가 성공적으로 완료되면 available_quantity를 해당 수량만큼 감소시킵니다.
- 판매 기록: 판매가 완료되면 RealTimeSale 테이블에 거래 내역을 기록합니다.

b) 예약 처리

- 예약 가능성 확인: 사용자가 예약 상품을 예약하려고 할 때, 해당 날짜에 예약 가능한지 확인합니다.
- 예약 기록: 예약이 가능하면 Reservation 테이블에 예약 내역을 기록합니다.
- 예약 상태 업데이트: 예약이 완료되거나 취소될 때 reservation_status를 업데이트합니다.

예약 위주의 상품 관련 데이터 모델에 실시간 판매 가능한 상품 유형을 추가 하기 위한 방안을 제안하고 설계 하시오

c) 혼합 상품 유형 처리

만약 한 상품이 예약 상품이자 실시간 판매 가능한 상품인 경우, 두 유형의 로직을 모두 적용할 수 있습니다.
예를 들어, 실시간으로 재고가 업데이트되며, 특정 시간대에 예약이 필요한 상품에 대해서는 Reservation과 RealTimeSale 테이블을 동시에 활용할 수 있습니다.

4. 확장성 및 성능 고려사항

a) 파티셔닝

실시간 판매 상품의 거래 데이터가 많아질 경우, RealTimeSale 테이블을 시간대별로 파티셔닝하여 성능을 최적화할 수 있습니다.

```
CREATE TABLE Orders (
    order_id BIGINT PRIMARY KEY,
    customer_id BIGINT,
    product_id BIGINT,
    order_date TIMESTAMP,
    status VARCHAR(50),
    total_amount DECIMAL(10, 2)
) PARTITION BY RANGE (order_date);
```

b) 캐싱

자주 조회되는 상품 정보(예: 상품명, 가격)는 캐시를 활용하여 데이터베이스 부하를 줄이고 응답 시간을 개선할 수 있습니다.
ex) 읽기 캐시: Redis와 같은 인메모리 캐시를 사용하여 자주 요청되는 제품 정보나 주문 정보를 캐싱.

c) 인덱스 최적화

product_id, user_id에 인덱스를 추가하여 주문과 예약에 대한 조회 성능을 최적화할 수 있습니다.

5. 사용자 인터페이스 및 사용자 경험(UX) 고려

a) 상품 유형에 따른 UI 차별화

예약 상품과 실시간 판매 상품은 UI에서 명확히 구분되어야 합니다.
예를 들어, 예약 상품의 경우 예약 가능 날짜 선택 기능을 제공하고, 실시간 판매 상품의 경우 즉시 구매 버튼과 남은 재고 표시를 제공합니다.

b) 알림 및 확인 프로세스

실시간 판매 상품 구매 후 즉시 확인 알림을 제공하고, 예약 상품의 경우 예약 완료 또는 대기 상태 알림을 제공합니다.

결론

이 데이터 모델 설계는 예약 위주의 상품 데이터 모델에 실시간 판매 가능한 상품 유형을 추가하여 확장성을 유지하면서도 효율적인 데이터 관리를 가능하게 합니다.
실시간 판매와 예약 처리를 분리하여 각각의 특성에 맞는 최적화된 처리를 보장하고, 확장성과 성능을 유지할 수 있도록 설계되었습니다.
이를 통해 ShopWorld는 다양한 상품 유형을 효율적으로 관리하고, 사용자에게 원활한 구매 경험을 제공할 수 있습니다.

자사 직접 판매 상품뿐만 아닌 입점 업체에 대한 상품 주문 관련 서비스 모델을 설계 하시오

입점 업체의 상품 주문을 관리하기 위해, 자사 직접 판매 상품과 입점 업체의 상품을 통합 관리할 수 있는 서비스 모델을 설계해야 합니다.
이 모델은 각 주문이 어느 판매자(자사 또는 입점 업체)에서 이루어졌는지 구분하고, 주문 처리와 배송, 정산 등이 원활하게 이루어질 수 있도록 지원해야 합니다.

1. 기본 개념 설계

a) 판매자 구분

- 자사 직접 판매 상품: 자사가 직접 보유하고 판매하는 상품.
- 입점 업체 상품: 입점한 외부 업체가 보유하고 판매하는 상품.

b) 주문 처리 흐름 구분

- 자사 상품 주문: 자사 창고에서 직접 재고를 관리하고, 주문 처리와 배송을 담당합니다.
- 입점 업체 상품 주문: 입점 업체가 재고를 관리하며, 주문을 수신하고 자체적으로 배송을 처리합니다. 자사는 이 과정에서 중개자 역할을 수행합니다.

2. 데이터 모델 설계

a) 판매자 테이블

```
CREATE TABLE Seller (
    seller_id SERIAL PRIMARY KEY,
    seller_name VARCHAR(255) NOT NULL,
    seller_type VARCHAR(50) NOT NULL CHECK (seller_type IN ('INTERNAL', 'EXTERNAL')),  -- INTERNAL: 자사, EXTERNAL: 입점 업체
    contact_info JSONB,  -- 판매자 연락처 정보
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Seller 테이블은 판매자 정보를 관리하며, 자사와 입점 업체를 구분합니다.
seller_type 필드를 통해 판매자 유형을 정의하고, contact_info 필드를 사용하여 연락처나 주소 등 추가 정보를 JSON 형식으로 저장합니다.

b) 상품 테이블

```
CREATE TABLE Product (
    product_id SERIAL PRIMARY KEY,
    seller_id INT NOT NULL,
    product_name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    stock_quantity INT DEFAULT 0,  -- 재고 수량 (입점 업체도 자사와 동일하게 재고를 관리할 수 있음)
    product_type VARCHAR(50) NOT NULL CHECK (product_type IN ('RESERVABLE', 'REALTIME')),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_seller FOREIGN KEY (seller_id) REFERENCES Seller(seller_id)
);
```

Product 테이블은 상품 정보를 관리하며, 각 상품이 어느 판매자에 속하는지 seller_id로 연결됩니다.
stock_quantity 필드를 통해 자사 및 입점 업체의 재고를 관리합니다.

자사 직접 판매 상품뿐만 아닌 입점 업체에 대한 상품 주문 관련 서비스 모델을 설계 하시오

c) 주문 테이블

```
CREATE TABLE Order (
    order_id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10, 2) NOT NULL,
    order_status VARCHAR(50) DEFAULT 'PENDING',  -- 주문 상태: PENDING, CONFIRMED, SHIPPED, COMPLETED 등
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Order 테이블은 전체 주문 정보를 관리하며, 주문의 총액과 상태를 포함합니다.

d) 주문 항목 테이블

```
CREATE TABLE OrderItem (
    order_item_id SERIAL PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    seller_id INT NOT NULL,  -- 각 주문 항목의 판매자
    quantity INT NOT NULL,
    unit_price DECIMAL(10, 2) NOT NULL,
    total_price DECIMAL(10, 2) NOT NULL,
    shipment_status VARCHAR(50) DEFAULT 'PENDING',  -- 배송 상태: PENDING, SHIPPED, DELIVERED 등
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_order FOREIGN KEY (order_id) REFERENCES Order(order_id),
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id),
    CONSTRAINT fk_seller FOREIGN KEY (seller_id) REFERENCES Seller(seller_id)
);
```

OrderItem 테이블은 각 주문 내역의 항목을 관리하며, 각 항목이 어느 판매자(자사 또는 입점 업체)에서 판매된 것인지 seller_id로 구분합니다.
배송 상태(shipment_status)를 관리하여 입점 업체의 주문이 처리되고 배송되는 과정을 추적합니다.

3. 비즈니스 로직 및 서비스 흐름

a) 주문 생성 및 처리

1. 주문 생성: 사용자가 장바구니에서 상품을 선택하여 주문을 생성합니다. 주문은 자사 상품과 입점 업체 상품을 혼합하여 구성될 수 있습니다.
   주문 생성 시, 각 주문 항목에 대해 seller_id를 할당하여 어느 판매자의 상품인지를 명확히 합니다.
2. 주문 처리: 주문이 생성되면, OrderItem 테이블에서 각 항목이 해당 판매자에게 할당됩니다.
   자사 상품은 자사 창고에서 처리되고, 입점 업체 상품은 해당 업체로 주문 요청이 전달됩니다.
3. 재고 관리: 자사 상품의 경우, 주문이 완료되면 Product 테이블의 stock_quantity가 즉시 감소합니다.
   입점 업체의 경우, 업체가 재고를 관리하며, 자사 시스템에서 재고를 추적할 수 있도록 재고 정보 동기화가 필요할 수 있습니다.
4. 배송 처리: 자사 상품은 자사 물류 시스템에서 처리되며, 입점 업체 상품은 해당 업체가 배송을 책임집니다.
   OrderItem 테이블의 shipment_status를 통해 각 주문 항목의 배송 상태를 관리합니다.

자사 직접 판매 상품뿐만 아닌 입점 업체애 대한 상품 주문 관련 서비스 모델을 설계 하시오

b) 결제 및 정산

1. 결제 처리: 사용자가 주문을 완료할 때 전체 금액을 결제합니다. 자사와 입점 업체의 상품이 혼합된 경우, 각 판매자에 대한 정산 비율을 정의할 수 있습니다.
2. 정산 프로세스: 주문이 완료되면 자사는 입점 업체와의 계약에 따라 정산을 진행합니다. 정산 내역은 입점 업체별로 관리되고, 정산 주기에 따라 입점 업체에 금액을 지급합니다.

4. 확장성과 성능 고려사항

a) 파티셔닝

주문 데이터가 많이 쌓일 경우, Order와 OrderItem 테이블을 시간대별로 파티셔닝하여 성능을 최적화합니다.

b) 인덱스 최적화

seller_id, product_id, order_id에 인덱스를 추가하여 주문 처리 및 조회 성능을 최적화합니다.

c) 캐싱

자주 조회되는 입점 업체의 상품 정보와 재고 상태를 캐시하여 응답 시간을 줄이고, 데이터베이스 부하를 줄입니다.

5. 사용자 인터페이스 및 사용자 경험(UX) 고려

a) 판매자 정보 제공

사용자가 주문하는 상품이 자사 상품인지 입점 업체 상품인지 명확히 구분할 수 있도록 UI에 판매자 정보를 제공합니다.
각 판매자별로 배송 정책이나 환불 정책이 다를 수 있으므로, 주문 전 이에 대한 안내를 제공합니다.

b) 통합된 주문 관리

사용자는 자사 상품과 입점 업체 상품을 동일한 주문 관리 인터페이스에서 확인하고 추적할 수 있습니다.
배송 상태와 정산 상태를 한눈에 볼 수 있도록 주문 내역을 통합 관리합니다.

결론

이 서비스 모델 설계는 자사와 입점 업체의 상품을 통합 관리하면서도, 각 판매자별 특성을 반영하여 효율적으로 운영할 수 있도록 합니다.
자사와 입점 업체 간의 주문 처리, 재고 관리, 배송, 정산이 원활하게 이루어지도록 설계되어 있으며, 확장성과 성능도 고려하여 시스템의 안정성과 사용자 경험을 향상시킬 수 있습니다.

상품 출시에 대한 라이프 사이클을 관리 하는 모델을 제시하고 설계 사유를 제시하시오

상품 출시 라이프 사이클 관리 모델 설계
상품 출시 라이프 사이클을 관리하기 위한 데이터 모델은 상품이 초기 기획 단계부터 출시, 판매, 그리고 종료까지의 모든 단계를 관리할 수 있도록 설계되어야 합니다.
이 모델은 상품의 각 단계에서 필요한 정보를 기록하고, 상태를 추적하며, 관련된 작업을 자동화하거나 지원하는 데 초점을 맞춥니다.

1. 라이프 사이클 단계 정의
상품의 라이프 사이클은 일반적으로 다음과 같은 단계로 구성됩니다:

- **기획 (Planning) :** 상품의 아이디어가 생성되고, 초기 기획이 이루어지는 단계입니다.
- **개발 (Development) :** 상품이 실제로 개발되고, 생산 준비가 진행되는 단계입니다.
- **검토 및 승인 (Review & Approval) :** 개발된 상품이 내부 검토를 거치고, 출시를 위한 최종 승인이 이루어지는 단계입니다.
- **출시 준비 (Launch Preparation) :** 출시일을 준비하고, 마케팅 계획과 출시 관련 작업이 수행되는 단계입니다.
- **출시 (Launch) :** 상품이 공식적으로 시장에 출시되는 단계입니다.
- **판매 및 모니터링 (Sales & Monitoring) :** 상품이 판매되며, 시장 반응을 모니터링하고, 성과를 분석하는 단계입니다.
- **단종 (End of Life) :** 상품의 판매가 종료되고, 재고 관리 및 후속 작업이 이루어지는 단계입니다.

2. 데이터 모델 설계
a) Product 테이블

```
CREATE TABLE Product (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    product_description TEXT,
    product_category VARCHAR(255),
    price DECIMAL(10, 2),
    lifecycle_status VARCHAR(50) NOT NULL CHECK (lifecycle_status IN ('PLANNING', 'DEVELOPMENT', 'REVIEW', 'PREPARATION', 'LAUNCH', 'SALES', 'END_OF_LIFE')),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

설계 사유: Product 테이블은 상품에 대한 기본 정보를 저장합니다. lifecycle_status 필드를 통해 상품이 현재 라이프 사이클의 어느 단계에 있는지를 추적할 수 있습니다.
이 필드는 각 단계에서 상태를 관리하기 위해 사용됩니다.

b) LifecycleEvent 테이블

```
CREATE TABLE LifecycleEvent (
    event_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    event_type VARCHAR(50) NOT NULL CHECK (event_type IN ('CREATED', 'UPDATED', 'APPROVED', 'REJECTED', 'LAUNCHED', 'RETIRED')),
    event_description TEXT,
    event_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id)
);
```

설계 사유: LifecycleEvent 테이블은 각 상품의 라이프 사이클에서 발생하는 주요 이벤트를 기록합니다. 이는 상품의 상태 변화를 추적하고,
각 상태에서 발생한 이벤트를 기록하기 위함입니다.
Event_type 필드는 각 이벤트 유형을 나타내며, 이 데이터를 통해 상품의 진행 상황을 명확하게 파악할 수 있습니다.

상품 출시에 대한 라이프 사이클을 관리 하는 모델을 제시하고 설계 사유를 제시하시오

c) Task 테이블

```
CREATE TABLE Task (
    task_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    task_name VARCHAR(255) NOT NULL,
    task_status VARCHAR(50) NOT NULL CHECK (task_status IN ('PENDING', 'IN_PROGRESS', 'COMPLETED', 'CANCELED')),
    assigned_to VARCHAR(255),
    due_date DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id)
);
```

설계 사유: Task 테이블은 상품 출시와 관련된 모든 작업을 관리하기 위해 설계되었습니다.
각 작업은 특정 상품에 연결되며, 작업의 상태(task_status)와 책임자(assigned_to), 마감일(due_date)을 추적할 수 있습니다.
이 테이블은 프로젝트 관리 기능을 제공하여, 상품 출시가 계획대로 진행되도록 지원합니다.

3. 상품 라이프 사이클 관리 흐름

1. 기획 단계 (Planning)
- **상품 아이디어 생성 :** 새로운 상품이 기획되면 Product 테이블에 초기 기록이 생성되고, lifecycle_status가 PLANNING으로 설정됩니다.
- **관련 작업 생성 :** 기획 단계에서 필요한 작업(예: 시장 조사, 초기 디자인)이 Task 테이블에 생성됩니다.

2. 개발 단계 (Development)
- **개발 시작 :** 상품이 개발 단계로 이동하면, lifecycle_status가 DEVELOPMENT로 업데이트됩니다.
  개발에 필요한 작업들이 Task 테이블에 추가됩니다(예: 프로토타입 제작, 기술 사양 정의).

3. 검토 및 승인 (Review & Approval)
- **내부 검토 및 승인 요청 :** 상품이 개발되면, 내부 검토를 위해 lifecycle_status가 REVIEW로 변경됩니다.
  승인이 완료되면 LifecycleEvent 테이블에 APPROVED 이벤트가 기록됩니다. 반대로 거절되면 REJECTED 이벤트가 기록되며, 수정 작업이 요구될 수 있습니다.

4. 출시 준비 (Launch Preparation)
- **마케팅 및 출시 준비 :** 상품이 승인되면, 출시 준비 작업이 시작되며 lifecycle_status가 PREPARATION으로 변경됩니다.
  마케팅 계획, 광고 제작, 물류 준비 등의 작업이 Task 테이블에 추가됩니다.

5. 출시 (Launch)
- **상품 출시 :** 상품이 공식적으로 출시되면, lifecycle_status가 LAUNCH로 변경됩니다.
  이와 동시에 LifecycleEvent 테이블에 LAUNCHED 이벤트가 기록됩니다.

상품 출시에 대한 라이프 사이클을 관리 하는 모델을 제시하고 설계 사유를 제시하시오

6. 판매 및 모니터링 (Sales & Monitoring)
- **판매 관리 :** 상품이 판매되는 동안 lifecycle_status는 SALES로 유지되며, 판매 성과와 시장 반응을 모니터링합니다.
  필요한 경우 가격 조정, 프로모션 등의 작업이 수행됩니다.

7. 단종 (End of Life)
- **상품 단종 :** 상품이 더 이상 판매되지 않을 때, lifecycle_status가 END_OF_LIFE로 변경됩니다.
  LifecycleEvent 테이블에 RETIRED 이벤트가 기록되며, 남은 재고 처리, 고객 통지 등의 작업이 수행됩니다.

4. 설계 사유 요약

유연한 상태 관리 :

Product 테이블의 lifecycle_status 필드를 통해 상품이 어떤 라이프 사이클 단계에 있는지를 명확하게 관리할 수 있습니다.
이는 각 단계에서 다른 프로세스를 트리거할 수 있게 하여 효율적인 라이프 사이클 관리가 가능합니다.

이벤트 추적:

LifecycleEvent 테이블을 통해 상품의 상태 변화와 중요한 이벤트를 추적할 수 있습니다.
이를 통해 상품의 개발 및 출시 과정에서 발생한 중요한 사항을 기록하고, 분석할 수 있습니다.

작업 관리:

Task 테이블을 통해 각 단계에서 필요한 작업을 관리함으로써, 상품 출시가 체계적으로 이루어질 수 있도록 지원합니다.
이는 프로젝트 관리 측면에서 중요한 기능을 제공합니다.

확장성: 이 모델은 다양한 상품 유형과 복잡한 라이프 사이클 관리에 쉽게 확장될 수 있도록 설계되었습니다.

결론

이 상품 출시 라이프 사이클 관리 모델은 상품이 처음 기획된 시점부터 판매 종료까지의 모든 과정을 체계적으로 관리할 수 있도록 설계되었습니다.
각 단계에서 필요한 작업과 이벤트를 명확하게 관리함으로써, 출시 과정에서의 문제를 최소화하고, 효율적인 운영을 지원할 수 있습니다.
이 모델을 통해 기업은 상품의 성공적인 출시와 지속적인 관리에 필요한 모든 정보를 체계적으로 관리할 수 있습니다.

특정 상품에 대한 옵션별 관리 방안에 대한 모델을 제시하시오

상품 옵션 관리는 특정 상품이 여러 가지 옵션(예: 색상, 사이즈, 스타일 등)을 가질 수 있는 경우에 필수적인 기능입니다.
이러한 옵션들은 각각의 재고, 가격, 가용성 등을 별도로 관리할 수 있어야 합니다.

1. 데이터 모델 개요

a) 상품과 옵션 간의 관계

- **상품(Product) :** 기본 상품 정보를 관리하는 테이블입니다.
- **옵션 그룹(OptionGroup) :** 각 상품에 대해 어떤 종류의 옵션들이 존재하는지를 관리합니다. 예를 들어, 색상, 사이즈 등이 옵션 그룹에 해당합니다.
- **옵션(Option):** 각 옵션 그룹에 속하는 구체적인 옵션 값들을 관리합니다. 예를 들어, 색상 그룹에는 빨강, 파랑, 초록 등이 속합니다.
- **상품 옵션(ProductOption):** 특정 상품과 옵션의 조합을 관리하며, 각 조합에 대해 별도의 재고, 가격 등을 관리할 수 있습니다.

2. 데이터 모델 설계

a) Product 테이블

```
CREATE TABLE Product (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    product_description TEXT,
    base_price DECIMAL(10, 2) NOT NULL,  -- 기본 가격
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

설계 사유: Product 테이블은 기본 상품 정보를 관리합니다. base_price는 옵션을 포함하지 않은 기본 가격을 나타냅니다.

b) OptionGroup 테이블

```
CREATE TABLE OptionGroup (
    option_group_id SERIAL PRIMARY KEY,
    group_name VARCHAR(255) NOT NULL,  -- 예: 색상, 사이즈 등
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

설계 사유: OptionGroup 테이블은 각 상품에 적용할 수 있는 옵션 그룹을 정의합니다. 예를 들어, 색상, 사이즈, 재질 등이 옵션 그룹에 해당합니다.

c) Option 테이블

```
CREATE TABLE Option (
    option_id SERIAL PRIMARY KEY,
    option_group_id INT NOT NULL,
    option_value VARCHAR(255) NOT NULL,  -- 예: 빨강, 파랑, 초록
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_option_group FOREIGN KEY (option_group_id) REFERENCES OptionGroup(option_group_id)
);
```

설계 사유: Option 테이블은 각 옵션 그룹에 속하는 구체적인 옵션 값을 관리합니다. 예를 들어, '색상'이라는 옵션 그룹에 '빨강', '파랑', '초록' 등이 해당됩니다.

특정 상품에 대한 옵션별 관리 방안에 대한 모델을 제시하시오

d) ProductOption 테이블

```
CREATE TABLE ProductOption (
    product_option_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    option_id INT NOT NULL,
    additional_price DECIMAL(10, 2) DEFAULT 0,  -- 옵션에 따른 추가 가격
    stock_quantity INT DEFAULT 0,  -- 옵션별 재고 수량
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id),
    CONSTRAINT fk_option FOREIGN KEY (option_id) REFERENCES Option(option_id)
);
```

설계 사유: ProductOption 테이블은 특정 상품과 옵션의 조합을 관리하며, 이 조합에 대해 별도의 가격(additional_price), 재고(stock_quantity)를 설정할 수 있습니다.
이를 통해 각 상품 옵션별로 상세한 관리가 가능해집니다.

e) ProductOptionCombination 테이블 (선택 사항)

```
CREATE TABLE ProductOptionCombination (
    combination_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    option_combination JSONB NOT NULL,  -- 여러 옵션의 조합을 JSON 형식으로 저장
    total_stock_quantity INT DEFAULT 0,  -- 조합에 따른 총 재고 수량
    total_price DECIMAL(10, 2) NOT NULL,  -- 조합에 따른 총 가격
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product FOREIGN KEY (product_id) REFERENCES Product(product_id)
);
```

설계 사유: ProductOptionCombination 테이블은 복수의 옵션이 결합된 경우(예: 색상과 사이즈) 이 조합에 대한 총 재고 수량과 총 가격을 관리합니다.
여러 옵션의 조합을 JSON 형식으로 저장함으로써 유연하게 여러 옵션을 조합할 수 있습니다.

특정 상품에 대한 옵션별 관리 방안에 대한 모델을 제시하시오

3. 상품 옵션 관리 흐름

1. 옵션 그룹 및 옵션 정의

- 상품을 생성할 때, 해당 상품에 적용할 옵션 그룹(예: 색상, 사이즈)을 정의합니다.
- 각 옵션 그룹에 대해 가능한 옵션 값을 설정합니다.

2. 상품 옵션 설정

- 상품과 각 옵션을 조합하여 ProductOption 테이블에 기록합니다. 여기에는 각 옵션별 추가 가격과 재고 수량을 설정합니다.

3. 옵션 조합 관리

- 여러 옵션이 결합된 상품 조합이 필요할 경우, ProductOptionCombination 테이블을 사용하여 특정 조합에 대한 재고와 가격을 관리합니다.

4. 주문 시 옵션 선택

- 고객이 상품을 주문할 때, 가능한 옵션(예: 색상: 빨강, 사이즈: L)을 선택할 수 있습니다.
- 선택된 옵션에 따라 최종 가격이 계산되고, 재고가 차감됩니다.

4. 설계 사유 요약

- **유연성:** 이 모델은 다양한 옵션 그룹과 옵션 값을 유연하게 관리할 수 있도록 설계되었습니다.
  ProductOptionCombination을 통해 복잡한 옵션 조합도 효율적으로 관리할 수 있습니다.

- **재고 및 가격 관리:** 옵션별로 재고와 가격을 독립적으로 관리함으로써, 각 옵션의 가용성을 정확하게 파악하고 가격 전략을 세울 수 있습니다.

- **확장성:** JSONB 형식을 사용한 옵션 조합 저장을 통해 옵션의 수와 유형이 증가하더라도 모델을 확장하여 관리할 수 있습니다.

결론

이 데이터 모델은 다양한 상품 옵션을 체계적으로 관리할 수 있도록 설계되었습니다.
상품 옵션별로 재고와 가격을 독립적으로 관리할 수 있어, 고객에게 더 나은 구매 경험을 제공할 수 있으며, 동시에 재고 관리의 효율성을 높일 수 있습니다.
또한, 옵션 조합을 유연하게 처리함으로써 복잡한 옵션 관리 요구사항도 충족할 수 있습니다.

상품 주문 절차에 대한 프로세스를 설계 하시오

상품 주문 절차는 고객이 상품을 선택하고 구매를 완료하는 과정에서 여러 단계로 이루어집니다.
이 프로세스는 주문의 생성부터 결제, 재고 확인, 배송, 그리고 최종적인 주문 완료까지 포함됩니다.

1. 상품 주문 절차 개요

1. 상품 탐색 및 선택 : 고객이 상품을 탐색하고 장바구니에 추가합니다.
2. 주문 생성 : 장바구니에서 선택한 상품을 주문으로 생성합니다.
3. 결제 처리 : 결제 수단을 선택하고 결제를 완료합니다.
4. 주문 확인 및 재고 확인 : 주문이 확인되고, 각 상품의 재고를 확인합니다.
5. 주문 처리 및 준비 : 재고가 확인되면, 주문을 처리하고 배송 준비를 시작합니다.
6. 배송 및 추적 : 주문이 발송되고, 고객이 배송 상태를 추적할 수 있습니다.
7. 주문 완료 : 고객이 상품을 수령하면 주문이 완료됩니다.
8 . 반품 및 환불 (필요시) : 고객이 반품 또는 환불을 요청할 수 있습니다.

2. 상세 프로세스 설계

1. 상품 탐색 및 선택
   - 상품 검색 및 조회 고객이 카테고리별 또는 검색 기능을 사용하여 상품을 탐색합니다.
   - 각 상품의 상세 정보 페이지에서 제품 설명, 가격, 재고 상태 등을 확인합니다.
   - 장바구니에 상품 추가

고객이 구매하려는 상품을 장바구니에 추가합니다.
장바구니에 추가된 상품은 구매 확정 전까지 수정(수량 변경, 제거)할 수 있습니다.
2. 주문 생성
주문서 작성:

장바구니에서 구매를 원하는 상품을 선택하고, 주문서 작성 페이지로 이동합니다.
고객의 배송 정보와 연락처 정보를 입력합니다.
주문 요약:

주문할 상품 목록, 수량, 가격을 최종 확인합니다.
적용 가능한 할인이나 프로모션 코드가 있는 경우 이를 입력할 수 있습니다.
3. 결제 처리
결제 방법 선택:

신용카드, 페이팔, 계좌이체, 모바일 결제 등 다양한 결제 방법 중 하나를 선택합니다.
결제 정보 입력 및 확인:

선택한 결제 방법에 따라 필요한 결제 정보를 입력합니다.
결제 정보가 올바른지 확인하고, '결제 완료' 버튼을 클릭합니다.
결제 승인 및 확인:

결제 시스템과 연동하여 결제가 승인됩니다.
결제가 완료되면 고객에게 결제 확인 메시지가 표시됩니다.
결제가 완료된 후, 주문 정보가 데이터베이스에 저장됩니다.
주문 ID가 생성되고, 고객에게 주문 확인 이메일이 전송됩니다.
재고 확인:

각 상품의 재고를 확인합니다.
만약 재고가 부족한 경우, 고객에게 재고 부족 메시지를 보내고 대체 상품 제안 또는 주문 취소 여부를 결정합니다.
5. 주문 처리 및 준비
주문 준비 시작:

재고가 확인되면, 주문 처리 시스템에서 주문을 준비합니다.
포장 및 출고 작업을 진행합니다.
입점 업체의 경우:

입점 업체가 관련 주문을 수신하고, 자체적으로 주문을 준비하여 배송을 시작합니다.
6. 배송 및 추적
배송 정보 업데이트:

주문이 발송되면, 배송 정보가 업데이트됩니다.
고객에게 배송 시작 알림과 함께 추적 번호가 제공됩니다.
배송 상태 추적:

고객은 웹사이트 또는 모바일 앱을 통해 실시간으로 배송 상태를 추적할 수 있습니다.
7. 주문 완료
상품 수령 확인:

고객이 상품을 수령하면, 주문이 완료된 것으로 간주됩니다.
고객이 수령 확인을 하고, 주문 상태가 '완료됨'으로 업데이트됩니다.
리뷰 및 피드백 요청:

고객에게 구매한 상품에 대한 리뷰나 피드백을 남길 수 있도록 요청합니다.
8. 반품 및 환불 (필요시)
반품 요청:

고객이 상품에 문제가 있거나 마음에 들지 않을 경우, 반품 요청을 할 수 있습니다.
반품 정책에 따라 고객에게 반품 절차를 안내합니다.
반품 처리:

반품된 상품이 도착하면 상태를 확인하고, 환불 절차를 시작합니다.
환불이 완료되면 고객에게 확인 메시지를 전송합니다.
3. 프로세스 다이어그램
아래는 각 단계의 흐름을 시각적으로 표현한 프로세스 다이어그램입니다:

MSA 환경에서 Two-Phase Commit 처리 방안을 설계하시오

Microservices Architecture (MSA) 환경에서 데이터 일관성을 유지하면서 분산 트랜잭션을 관리하는 것은 매우 중요한 과제입니다.
Two-Phase Commit (2PC)는 분산 트랜잭션의 일관성을 보장하기 위한 표준 프로토콜입니다.
하지만, MSA 환경에서는 여러 마이크로서비스 간의 상호 작용이 많고, 독립적인 서비스들이 느슨하게 결합되어 있기 때문에,
전통적인 2PC 방식이 성능과 확장성 측면에서 도전 과제를 안고 있습니다.

1. Two-Phase Commit의 기본 개념

2PC는 두 단계로 이루어진 분산 트랜잭션 프로토콜입니다:

Prepare Phase:
- 트랜잭션 관리자가 각 참여자에게 트랜잭션을 준비(Prepare)하라고 요청합니다.
- 각 참여자는 로컬 트랜잭션을 준비하고, 성공 여부를 트랜잭션 관리자에게 응답합니다.
- 모든 참여자가 준비 상태임을 확인하면, 트랜잭션 관리자는 커밋을 진행합니다.

Commit Phase:
- 트랜잭션 관리자가 각 참여자에게 트랜잭션을 커밋하라고 요청합니다.
- 각 참여자는 로컬 트랜잭션을 커밋하고, 성공 여부를 트랜잭션 관리자에게 응답합니다.
- 모든 참여자가 커밋을 완료하면, 트랜잭션은 성공적으로 종료됩니다. 한 참여자라도 실패하면, 모든 참여자에게 롤백을 요청합니다.

2. MSA 환경에서의 2PC 적용 시 고려사항

MSA 환경에서는 다음과 같은 고려사항이 있습니다:

네트워크 지연 및 실패:
- MSA에서는 서비스 간 네트워크 지연이나 장애가 발생할 가능성이 높기 때문에, 2PC의 동기적 특성이 성능에 영향을 미칠 수 있습니다.

서비스 독립성:
- 각 마이크로서비스는 독립적으로 동작해야 하기 때문에, 2PC는 서비스 간 강한 결합을 유발할 수 있습니다.

확장성:
- 2PC는 참여자가 많아질수록 성능이 저하될 수 있으며, 확장성이 제한될 수 있습니다.

MSA 환경에서 Two-Phase Commit 처리 방안을 설계하시오

3. 2PC 처리 방안 설계

MSA 환경에서 2PC를 성공적으로 구현하기 위해 다음과 같은 방안을 제안합니다:

a) 트랜잭션 관리자를 통한 2PC 구현

트랜잭션 관리 서비스 (Transaction Coordinator):

- MSA 환경에서 2PC를 처리하기 위해 중앙 트랜잭션 관리 서비스(Tx Coordinator)를 도입합니다.
- 이 서비스는 모든 참여 서비스에게 Prepare 및 Commit/Abort 요청을 전달하고, 각 서비스로부터 응답을 수집하여 트랜잭션을 제어합니다.

트랜잭션 단계

1. **트랜잭션 시작**: 클라이언트가 트랜잭션을 시작하면, 트랜잭션 관리 서비스가 트랜잭션 ID를 생성하고, 관련 서비스를 호출합니다.
2. **Prepare 단계:**
   - 트랜잭션 관리 서비스는 각 참여 서비스에게 Prepare 요청을 전송합니다.
   - 각 서비스는 로컬 작업을 수행한 후, 트랜잭션을 로그에 기록합니다(로컬 트랜잭션의 일관성 보장).
   - 모든 참여 서비스가 준비 완료를 통보하면, 트랜잭션 관리 서비스는 Commit 단계로 이동합니다.
3. **Commit 단계:**
   - 트랜잭션 관리 서비스는 모든 참여 서비스에게 Commit 요청을 전송합니다.
   - 각 서비스는 로컬 트랜잭션을 커밋하고, 결과를 관리 서비스에 통보합니다.
   - 모든 참여 서비스가 성공적으로 커밋을 완료하면, 트랜잭션은 성공적으로 종료됩니다.
4. **Rollback 처리:**
   - 만약 어느 한 참여 서비스라도 Prepare 단계에서 실패하거나, Commit 단계에서 문제가 발생하면, 트랜잭션 관리 서비스는 모든 참여 서비스에 Rollback을 요청합니다.
   - 각 서비스는 Rollback을 수행하고, 트랜잭션이 일관되지 않은 상태로 남지 않도록 보장합니다.

b) Saga 패턴 도입 (대안)

MSA 환경에서는 전통적인 2PC의 성능 이슈를 해결하기 위해 Saga 패턴을 고려할 수 있습니다.
Saga 패턴은 장기 실행 트랜잭션을 다루는 데 적합하며, 각 작업이 독립적으로 커밋되고, 전체 프로세스가 실패할 경우 보상 작업을 통해 롤백합니다.

Saga 단계

1. **각 서비스의 로컬 트랜잭션 실행:** 각 서비스는 독립적인 로컬 트랜잭션을 실행하고, 성공적으로 커밋됩니다.
2. **보상 트랜잭션 정의:** 각 서비스는 실패 시 호출될 보상 트랜잭션을 정의하여, 기존 작업을 취소하거나 수정할 수 있습니다.
3. **트랜잭션 실패 처리:** 전체 프로세스 중 어느 한 단계에서 실패가 발생하면, 이전에 실행된 모든 서비스는 보상 트랜잭션을 실행하여 상태를 원래대로 되돌립니다.
4. **최종 일관성 보장:** 모든 단계가 성공적으로 완료되면 트랜잭션이 성공적으로 완료됩니다. Saga 패턴은 최종 일관성을 보장하며, 2PC와 달리 동기적인 트랜잭션 관리가 필요하지 않습니다.

c) Timeout 및 Retry 메커니즘

Timeout 설정: 각 트랜잭션 단계에서 Timeout을 설정하여, 너무 오래 기다리지 않도록 합니다. Timeout이 발생하면 해당 트랜잭션을 실패로 간주하고 Rollback 또는 보상 작업을 수행합니다.
Retry 전략: 일시적인 네트워크 오류나 서비스 중단으로 인한 트랜잭션 실패 시, 일정 횟수만큼 재시도를 시도하여 트랜잭션의 성공률을 높입니다.

4. 예외 처리 및 모니터링

예외 처리: 각 단계에서 발생할 수 있는 예외 상황에 대해 적절한 예외 처리 로직을 구현합니다. 예외 상황 발생 시 트랜잭션 관리 서비스는 전체 트랜잭션을 실패로 처리하고 Rollback을 수행합니다.
모니터링 및 알림: 분산 트랜잭션의 상태를 실시간으로 모니터링하고, 실패나 지연 발생 시 운영팀에 알림을 보내 빠르게 대응할 수 있도록 합니다.

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

분산 트랜잭션이 필요한 이유
- 현재 재직중인 회사에서의 시스템은 수십여개의 서버들이 연결된 MSA 형태로 구성되어 있습니다.
- 시스템 별로 각기 DB를 분리하여 독립적으로 관리하고 트랜잭션의 가장 중요한 성질 중 하나는 '원자성' 입니다.
  단일 DB를 구성할 때와 다르게 DB를 분산하여 운영하게 될 경우 원자성을 만족시키기 어려울 수 있습니다.

A와 B의 데이터베이스가 분산되어있는경우
다음과 같은 이유로 분산트랜잭션에 대한 관리가 필요합니다.

1. 네트워크 지연 및 실패 이슈
분산 시스템에서는 여러 노드가 네트워크를 통해 통신합니다. 네트워크 지연이나 실패로 인해 특정 노드의 응답을 받지 못하거나 지연될 수 있습니다.
이로 인해 트랜잭션의 일부분만 커밋되고 일부분은 롤백되는 상황이 발생할 수 있습니다.

2. 데드락
여러 노드가 서로의 자원 또는 데이터에 동시에 접근하려 할 때, 상호간의 대기 상태에 빠져서 진행을 할 수 없게 되는 현상입니다.
분산 트랜잭션에서는 데드락을 해결하기 위한 중앙화된 관리 메커니즘이 없어 복잡한 해결 전략이 필요합니다.

3. 데이터 일관성 유지의 어려움
분산 시스템에서 데이터의 복제본이 여러 노드에 분산 저장될 수 있습니다. 따라서 한 노드에서의 데이터 변경이 모든 노드에 즉시 반영되지 않으면 일관성 문제가 발생할 수 있습니다.

```
@Service
@RequiredArgsConstructor
public class OrderService {

    private final InventoryService inventoryService;
    private final PaymentService paymentService;

    public void orderProcess(Order order) {
        inventoryService.decreaseStock(order.getItemId(), order.getQuantity());
        paymentService.charge(order.getUserId(), order.getTotalPrice());
    }
}
```

주문 과정의 일부를 코드로 나타내 보았습니다. 재고 서비스와 결제 서비스. 각 서비스는 각각의 데이터베이스를 가지고 있습니다.
주문이 들어올 때,결제는 성공했지만 재고가 없다면? 이 경우, 두 서비스의 트랜잭션은 롤백되어야 합니다.

위 코드에서 decreaseStock과 charge 메서드가 각기 다른 데이터베이스에 연결된다면, 한 메서드는 성공하고 다른 메서드가 실패할 위험이 있습니다.

이러한 상황에서의 일관성을 보장하기 위해 우리는 분산트랜잭션을 관리할 프로세스의 필요성이 있습니다.

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

그럼 스프링에서는?
Spring Boot 대표적으로 2-Phase-Commit(2PC) 또는 SAGA 패턴을 사용하여 분산 트랜잭션을 관리합니다.

Spring Boot에서 2PC를 구현하는 한 가지 방법은 **XA(eXtended Architecture)** 프로토콜을 사용하는 것입니다.
Spring Boot는 여러 서비스에서 트랜잭션을 관리하는 데 사용할 수 있는 **Atomikos 및 Bitronix 트랜잭션 관리자**를 통해 XA 트랜잭션을 지원합니다.

XA는 분산 컴퓨팅 환경에서 여러 리소스 (예: 데이터베이스, 메시징 시스템) 간의 트랜잭션을 조율하기 위한 표준 인터페이스입니다.
이 인터페이스는 2-Phase-Commit (2PC, Two-Phase Commit) 프로토콜을 사용하여 트랜잭션을 완료합니다.

2-Phase-Commit(2PC)
XA 프로토콜은 2PC 알고리즘을 통해 작동됩니다.
2PC은 두 단계로 구분되어 작동됩니다.

- **Prepare Phase (준비 단계):** 트랜잭션 매니저 (TM)는 모든 리소스 매니저 (RM)에게 트랜잭션 커밋 준비를 알립니다. RM들은 이 요청을 받고 필요한 모든 작업을 준비하며 준비가 완료되면 응답합니다.

- **Commit/Rollback Phase (커밋/롤백 단계):** 모든 RM이 준비되면 TM은 트랜잭션을 커밋합니다. 만약 어떤 RM이 준비되지 않았다면, TM은 트랜잭션을 롤백합니다.

유저가 계좌 A에서 계좌 B로 자금을 이체하는 경우를 예시로 살펴보겠습니다.

```
public interface BankService {
    void prepareTransfer(int amount);
    void commit();
    void rollback();
}

@Component
public class AccountAService implements BankService {

    // 계좌 A에서 금액을 차감하기 전의 준비 작업(계좌호출 차감 금액 확인등..)
    @Override
    public void prepareTransfer(int amount) {
        if(balance >= amount) {
            balance -= amount;
            return true;
        } else {
            return false; // 잔액 부족
        }
    }
}
```

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

```
@Component
public class AccountBService implements BankService {

                    // 계좌 B로 입금하기 전의 준비 작업(입금 계좌 확인 및 금액 확인 등..)
    @Override
    public void prepareTransfer(int amount) {
        balance += amount;
        return true;
    }
}

@Service
@RequiredArgsConstructor
public class TransferService {

    private final AccountAService accountA;
    private final AccountBService accountB;

    public void transfer(int amount) {
        boolean isPrepared = accountA.prepareTransfer(amount) && accountB.prepareTransfer(amount);

        if (isPrepared) {
            accountA.commit();
            accountB.commit();
        } else {
            accountA.rollback();
            accountB.rollback();
        }
    }
}
```

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

단계별 설명 :

1. Prepare Phase (준비 단계)
- 유저가 TransferService를 통해 금액 이체를 요청합니다.
- Coordinator(TransferService)는 계좌 A (AccountAService)와 계좌 B (AccountBService)에게 prepare 상태인지 확인을 요청합니다 (= prepareTransfer 메서드 호출)
- 각 계좌 서비스는 자신의 상태를 확인하여 prepare 상태이면 준비 완료 응답을 반환하고, 그렇지 않으면 준비 실패 응답을 반환합니다.

2. Commit/Rollback Phase (커밋/롤백 단계)
- Coordinator는 모든 계좌 서비스로부터 응답을 받습니다.
  만약 모든 서비스가 prepared 응답을 반환하면, Coordinator는 각 계좌 서비스에 커밋을 요청합니다. (= commit 메서드 호출)
- commit 호출에 따라 계좌 A에서는 출금이 이루어지고, 계좌 B에서는 입금이 이루어집니다.
  그러나 하나 이상의 서비스에서 prepared 실패 응답을 받으면, Coordinator는 롤백을 요청합니다.(=rollback 메서드 호출)
- rollback 호출에 따라 이전 상태로 복구됩니다.

3. 원자성 보장:
2PC의 핵심은 여러 데이터베이스나 서비스에 걸쳐 있는 트랜잭션도 하나의 트랜잭션처럼 다룰 수 있게 해준다는 것입니다.
즉, 모든 작업이 성공적으로 수행되거나 아무 작업도 수행되지 않은 것처럼 보장됩니다.
코디네이터의 중요성: 코디네이터는 분산 트랜잭션을 관리하고 조율하는 중요한 역할을 합니다.
모든 작업의 상태를 모니터링하고, 최종 커밋 또는 롤백 결정을 내립니다.

2PC를 사용하였을 경우의 문제점

트랜잭션의 책임이 Coordinator Node에 있으며 이 부분이 단일 실패지점(SPOF)가 될 수 있습니다.
전체 트랜잭션이 완료될 때까지 서비스에서 사용하는 리소스가 잠겨 있어 서비스가 완료될 때까지 대기하여야 합니다.
때문에 지연 시간이 늘어나고 리소스가 차단되어 확장이 어려워질 수 있습니다.
NoSQL은 2PC-분산 트랜잭션을 지원하지 않습니다.

SAGA 패턴

SAGA 패턴은 MSA환경에서 일관성을 지키기 여렵다는 것을 기반으로, **약간의 일관성을 포기하고 Eventual Consistency(최종 일관성)**을 보장하여 효율성을 높이기 위한 패턴입니다.

2PC에서는 트랜잭션을 하나의 트랜잭션으로 묶어서 처리를 하지만, **SAGA 패턴은 긴 트랜잭션을 여러 개의 짧은 로컬 트랜잭션으로 분리하는 접근 방식입니다.**
각 트랜잭션은 다른 트랜잭션의 완료를 기다리지 않고 독립적으로 실행됩니다. 따라서 트랜잭션의 원자성을 지켜줄 방법이 필요합니다.
만약 중간에 문제가 발생하면, 보상(Compenstation) 트랜잭션이 실행되어 이전 트랜잭션을 롤백하는 것과 같은 효과를 가져옵니다.

각 로컬 트랜잭션은 자신의 트랜잭션을 끝내고 다음 트랜잭션을 호출하는 메시지, 이벤트를 생성하게 됩니다.

그럼 보상 트랜잭션이 뭔데?

보상 트랜잭션은 분산된 트랜잭션 중 일부가 실패할 경우, 그 실패 전에 성공적으로 완료된 트랜잭션을 보상 즉, 되돌리는 역할을 하는 트랜잭션입니다.
SAGA 패턴의 트랜잭션은 분산된 여러 독립적인 트랜잭션이기 때문에, 어떤 서비스의 트랜잭션이 실패하면 단일 트랜잭션 처럼 롤백 메커니즘을 사용할 수 없습니다.
대신 보상 트랜잭션을 사용하여 이전에 성공한 트랜잭션의 효과를 취소합니다.

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

보상트랜잭션이 실패할 경우에는?
보상트랜잭션도 하나의 트랜잭션이기 때문에, 다양한 요인들로 인해 실패할 수 있습니다.
이에 대한 대비도 필요합니다!

사가 패턴은 이벤트기반으로 작동합니다.
- 보상 트랜잭션을 카프카 같은 데이터 스트리밍 서비스 같은 곳에서 처리하게 하고 멱등키와 함께 재시도 프로세스를 추가합니다.
  이후 N번이상 실패 할경우에는 어쩔수 없지만... 개발자가 수동으로 오류를 해결할 수 있게 알람을 주어야 합니다.

멱등키 활용 로직 예시

```
public class CompensationTransaction {
    private IdempotencyKey key; // 멱등키를 활용
    private Event event;

    public CompensationTransaction(IdempotencyKey key, Event event) {
        this.key = key;
        this.event = event;
    }

    public void execute() {
        if(!isProcessed(key)) {
            // 보상 로직 수행
            processCompensation(event);
            markAsProcessed(key);
        }
    }
}
```

멱등키의 사용 이유는 링크를 참조해주세요.
(토스페이먼츠에서 너무 정리를 잘해주셔서 꼭 보셨으면 좋겠습니다)
https://velog.io/@tosspayments/%EB%A9%B1%EB%93%B1%EC%84%B1%EC%9D%B4-%EB%AD%94%EA%B0%80%EC%9A%94

SAGA 패턴의 구현방법
- SAGA 패턴을 구현하는 방법은 두가지가 있습니다.
  Choreography SAGA(코레오크레피 사가), Orchestration SAGA(오케스트레이션 사가)

1. Choreography SAGA
Choreography 방식은 각 서비스끼리 이벤트를 주고 받는 방식입니다.
각 서비스가 다른 서비스의 로컬 트랜잭션을 이벤트 트리거하는 방식으로 이루어 집니다.

이 방식은 중앙집중된 지점이 없이 모든 서비스가 메시지 브로커(RabbitMQ, Kafka)를 통해 이벤트를 Pub/Sub 하는 구조입니다.

중앙 집중형 관리방식이 아니기 때문에 SPOF(단일 실패지점)이 없습니다.
새로운 서비스 추가가 필요할 때 서비스간 연결을 잘 확인해야 합니다.
서비스끼리 이벤트를 주고 받기 때문에 큰 시스템의 경우 구조의 파악이 어려워 질 가능성이 있습니다.
트랜잭션을 시뮬레이션 하기 위해 모든 서비스를 실행해야 하기 때문에 통합테스트와 디버깅이 어려운 점이 있습니다.

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

```
public class MessageBroker {
    public static void publish(String event, double amount) {
        // 이벤트 발행 로직
    }

    public static void subscribe(String event, Service service) {
        // 이벤트 구독 로직
    }
}

public class AAccountService {
    public void deductAmount(double amount) {
        if (canDeduct(amount)) {
            MessageBroker.publish("AmountDeducted", amount);
        } else {
            MessageBroker.publish("TransferFailed", amount);
        }
    }

    @Subscribe("CreditFailed")
    public void revertDeduction(double amount) {
        // 보상 로직
    }

    private boolean canDeduct(double amount) {
        return true;
    }
}

public class BAccountService {
    @Subscribe("AmountDeducted")
    public void creditAmount(double amount) {
        if (canCredit(amount)) {
            MessageBroker.publish("AmountCredited", amount);
        } else {
            MessageBroker.publish("CreditFailed", amount);
        }
    }

    private boolean canCredit(double amount) {
        return true;
    }
}
```

A. AccountService에서 금액을 인출하려고 시도합니다.
인출에 성공하면, "AmountDeducted" 이벤트가 MessageBroker를 통해 발행됩니다.
B. AccountService는 "AmountDeducted" 이벤트를 구독하고 있으므로 이 이벤트를 수신하고 금액을 입금하려고 시도합니다.
만약 BAccountService에서 입금에 실패하면, "CreditFailed" 이벤트가 발행됩니다.
AAccountService는 "CreditFailed" 이벤트를 구독하고 있으므로 이 이벤트를 수신하고 인출된 금액을 되돌립니다.

MSA 환경에서의 분산 트랜잭션 관리: 2PC & SAGA 패턴

2. Orchestration SAGA

오케스트레이션 사가는 중앙 집중형으로 실행 흐름을 관리하게 됩니다.
Ochestrator는 요청을 실행, 각 서비스의 상태를 확인하고, 실패에 대한 보상 트랜잭션을 실행합니다.

```
public class MessageBroker {
    public static void publish(String event, double amount) {
        // Publish event
    }

    public static void subscribe(String event, Service service) {
        // Subscribe
    }
}

public class Orchestrator {
    AAccountService aAccountService;
    BAccountService bAccountService;

    public Orchestrator(AAccountService aService, BAccountService bService) {
        this.aAccountService = aService;
        this.bAccountService = bService;
    }

    public void transferAmount(double amount) {
        if (aAccountService.deductAmount(amount)) {
            if (!bAccountService.creditAmount(amount)) {
                aAccountService.revertDeduction(amount);
            }
        }
    }
}

public class AAccountService {
    public boolean deductAmount(double amount) {
        if (canDeduct(amount)) {
            return true;
        } else {
            return false;
        }
    }

    public void revertDeduction(double amount) {
        // 보상 로직 구현
    }

    private boolean canDeduct(double amount) {
        return true;
    }
}
```

```
public class BAccountService {
    public boolean creditAmount(double amount) {
        if (canCredit(amount)) {
            return true;
        } else {
            return false;
        }
    }

    private boolean canCredit(double amount) {
        return true;
    }
}
```

Ochestration은 트랜잭션 처리를 위한 Manager 인스턴스가 별도로 존재합니다.
Ochestrator가 중앙 집중형 컨트롤러 역할 → 각 서비스에서 실행할 트랜잭션을 관리를 하게됩니다.
Ochestrator는 요청을 실행, 각 서비스의 상태를 확인하고, 실패에 대한 보상 트랜잭션을 실행합니다.
많은 서비스가 있는 복잡한 워크플로우에 적합합니다.
- A서비스는 B서비스의 트랜잭션의 결과를 알필요가 없습니다.

흐름을 파악하는데도 좋습니다.
Ochestrator가 전체 워크플로우를 관리하기 때문에 SPOF(단일 실패지점)가 될 가능성이 있습니다.
이후에는 직접 코드를 짜보면서 Ochestration을 공부해보도록 하겠습니다.

고객의 비밀번호를 안전하게 저장하기 위해 어떤 방법을 사용할 수 있는가?

고객의 비밀번호를 안전하게 저장하기 위해서는 암호학적 방법을 사용하여 비밀번호를 해싱하고, 이를 보호하기 위한 추가적인 보안 조치를 취해야 합니다.

1. 암호화 해싱 함수 사용
비밀번호는 평문으로 저장해서는 안 되며, 안전한 해싱 알고리즘을 사용하여 해시 값을 저장해야 합니다.
해싱 함수는 비밀번호를 고정된 크기의 고유한 데이터로 변환하는 함수로, 역으로 원래 비밀번호를 계산할 수 없도록 설계되었습니다.

1. PBKDF2 (Password-Based Key Derivation Function 2):
반복적으로 해싱을 수행하여 계산을 복잡하게 만들어 공격이 어렵도록 합니다. 많은 보안 라이브러리에서 지원하며, 해시 계산에 시간이 걸리도록 설정할 수 있습니다.

2. bcrypt:
비밀번호 해싱을 위한 알고리즘으로, salt와 함께 해시를 계산하며, 반복 횟수를 조정하여 해싱의 난이도를 높일 수 있습니다. 공격자가 모든 가능한 비밀번호 조합을 시도하는 시간(Brute-force 공격)을 늘릴 수 있습니다.

3. scrypt:
bcrypt와 유사하지만, 메모리 사용량도 조절할 수 있어 더욱 강력한 보안을 제공합니다. 특히, 병렬화된 공격(Parallel Attacks)에 더 강합니다.

2. Salt 추가
Salt란?:
해싱하기 전에 비밀번호에 추가되는 임의의 데이터입니다. salt는 고유한 값이어야 하며, 비밀번호 해시와 함께 저장됩니다. 같은 비밀번호라도 사용자가 다르다면 서로 다른 해시 값이 생성되도록 합니다.

1. Salt의 중요성:
rainbow table 공격(미리 계산된 해시값을 사용하여 빠르게 비밀번호를 역추적하는 방법)을 방어할 수 있습니다. 각 사용자의 비밀번호에 고유한 salt를 사용하면, 공격자가 rainbow table을 사용해도 공격을 수행하기 어려워집니다.

3. Pepper 사용
Pepper란?:
비밀번호에 추가로 더하는 비밀 키(pepper)로, salt와는 달리 시스템 전체적으로 공통된 값으로 유지됩니다. pepper는 코드나 환경 변수에 저장하며, 해시와 함께 저장하지 않습니다.

Pepper의 사용법:
비밀번호에 pepper를 추가한 후 해싱을 수행합니다. 이는 추가적인 보안을 제공하며, 데이터베이스가 유출되더라도 공격자가 해시값만으로는 원래 비밀번호를 알아낼 수 없게 합니다.

4. 비밀번호 해시를 안전하게 저장하기 위한 추가 조치
1. 키 스트레칭 (Key Stretching): 해시 계산을 여러 번 반복하여(예: bcrypt의 반복 횟수를 높임) 계산을 더 복잡하게 만듭니다. 이는 공격자가 해시값을 풀기 위해 더 많은 시간과 자원을 소모하게 만듭니다.

2. 정기적인 해시 알고리즘 업데이트: 시간이 지나면서 더 강력한 해시 알고리즘이 등장할 수 있습니다. 시스템의 해시 알고리즘이 약화된 것으로 판명되면, 더 강력한 알고리즘으로 마이그레이션해야 합니다.

3. 보안적으로 강력한 환경 변수 관리: Pepper와 같은 비밀 데이터를 환경 변수로 관리하고, 안전한 저장소(예: AWS Secrets Manager, HashiCorp Vault)에서 관리합니다. 데이터베이스 유출만으로는 Pepper 값을 알아낼 수 없도록 합니다.

5. 비밀번호 재사용 방지 및 강도 요구
1. 비밀번호 강도 정책: 사용자에게 충분히 강력한 비밀번호(예: 최소 길이, 대문자, 소문자, 숫자, 특수문자 포함)를 사용하도록 요구합니다.

2. 비밀번호 재사용 방지: 이전에 사용했던 비밀번호를 다시 사용할 수 없도록 하여, 유출된 비밀번호를 재사용하는 것을 방지합니다.

6. 2단계 인증(2FA) 도입
1. 추가 보안 계층: 비밀번호 이외의 인증 수단(예: SMS 코드, OTP, 인증 앱)을 추가하여 보안을 강화합니다. 비밀번호가 유출되더라도 추가 인증 없이 접근할 수 없도록 합니다.

결론
비밀번호를 안전하게 저장하기 위해서는 안전한 해싱 알고리즘(PBKDF2, bcrypt, scrypt)을 사용하고, salt와 pepper를 활용하여 보안을 강화해야 합니다.
이를 통해 비밀번호의 무작위 대입 공격을 방지하고, 데이터베이스가 유출되더라도 사용자의 비밀번호를 안전하게 보호할 수 있습니다.
추가적으로, 비밀번호 강도 요구사항 및 2단계 인증을 도입하여 전반적인 계정 보안을 강화할 수 있습니다.

고객이 안전하게 로그인할 수 있도록 로그인 절차를 설계하고, 이에 필요한 보안 대책을 설명하세요.힌트: JWT(JSON Web Token)와 같은 토큰 기반 인증을 사용하세요. TLS/SSL을 통해 데이터 전송 시 암호화하는 것도 중요한 보안 대책입니다.

고객이 안전하게 로그인할 수 있도록 설계된 로그인 절차는 강력한 보안 대책을 포함해야 하며, 고객의 인증 정보가 안전하게 보호되도록 해야 합니다.

1. 로그인 절차 설계

a) 로그인 요청

1. TLS/SSL을 통한 보안 통신: 모든 로그인 요청은 HTTPS를 통해 전송되며, TLS/SSL로 암호화되어야 합니다. 이는 클라이언트와 서버 간의 통신이 안전하게 이루어지도록 보장합니다.

2. 사용자 자격 증명 입력: 사용자는 로그인 폼에 자신의 자격 증명(이메일/사용자 이름 및 비밀번호)을 입력합니다.

3. 서버로 로그인 요청 전송: 사용자가 자격 증명을 제출하면, 클라이언트는 이를 암호화된 통신을 통해 서버에 전송합니다.

b) 서버 측 인증 처리

1. 자격 증명 검증: 서버는 데이터베이스에서 사용자의 이메일/사용자 이름을 검색하고, 해당 사용자의 해시된 비밀번호와 입력된 비밀번호를 비교합니다.
해시된 비밀번호는 안전한 해시 알고리즘(예: bcrypt, PBKDF2, scrypt)을 사용하여 저장되며, 입력된 비밀번호는 동일한 해시 알고리즘을 통해 해시된 후 비교됩니다.

2. 인증 성공 시 JWT 생성: 인증에 성공하면, 서버는 사용자 정보를 바탕으로 JWT를 생성합니다.
JWT는 사용자 식별자(user_id), 발급 시간(iat), 만료 시간(exp) 등의 정보를 포함할 수 있습니다.

3. JWT 서명: JWT는 서버의 비밀 키를 사용하여 서명됩니다. 서명된 JWT는 클라이언트가 이후 요청에서 자신의 신원을 증명하는 데 사용됩니다.

4. 클라이언트로 JWT 전송: 서버는 서명된 JWT를 클라이언트에 반환합니다. 이 토큰은 클라이언트의 로컬 스토리지 또는 쿠키에 저장됩니다.

c) 인증된 요청 처리

1. JWT 포함 요청: 클라이언트는 이후 모든 인증된 요청(예: 사용자 정보 조회, 주문 내역 확인 등)에 JWT를 포함시켜 서버에 전송합니다.
JWT는 일반적으로 HTTP 헤더의 Authorization 필드에 Bearer 스키마로 포함됩니다.

2. JWT 검증: 서버는 요청을 수신하면, JWT를 검증합니다.
토큰이 변조되지 않았는지 확인하기 위해 서명을 검증하고, 토큰의 만료 시간(exp)과 유효성을 확인합니다.

3. 요청 처리 및 응답: JWT가 유효하면, 서버는 요청된 작업을 수행하고, 결과를 클라이언트에 응답합니다.
만약 JWT가 유효하지 않거나 만료된 경우, 클라이언트에게 인증 실패 메시지를 반환하고 재로그인을 요청할 수 있습니다.

2. 보안 대책 설명

a) TLS/SSL을 통한 데이터 전송 암호화

1. 목적: TLS/SSL을 사용하여 클라이언트와 서버 간의 모든 통신을 암호화함으로써, 데이터 전송 중에 발생할 수 있는 도청 및 중간자 공격(Man-in-the-Middle Attack)을 방지합니다.

2. 구현 방법: HTTPS 프로토콜을 통해 모든 요청과 응답을 암호화하며, 클라이언트와 서버 간의 세션을 안전하게 유지합니다.

b) 비밀번호 안전 관리

1. 안전한 해시 알고리즘 사용: 비밀번호는 bcrypt, PBKDF2, scrypt와 같은 강력한 해시 알고리즘을 사용하여 해시 처리 후 데이터베이스에 저장됩니다.
해시 처리 시 각 비밀번호에 고유의 salt를 추가하여 동일한 비밀번호라도 다른 해시 값을 가지도록 합니다.

2. 비밀번호 강도 요구사항:
사용자가 안전한 비밀번호를 설정하도록 비밀번호 강도 요구사항(최소 길이, 대문자/소문자/숫자/특수문자 포함)을 적용합니다.

c) JWT 보안

1. 서명된 JWT: JWT는 서버의 비밀 키로 서명되어, 클라이언트가 발급받은 토큰을 수정할 수 없도록 보장합니다. 서명 검증을 통해 서버는 토큰의 진위 여부를 확인할 수 있습니다.
2. JWT 만료 시간 설정: JWT에는 만료 시간(exp)을 설정하여, 일정 기간 이후에는 토큰이 무효화되도록 합니다. 만료된 토큰으로는 서버에 접근할 수 없으며, 재로그인이 필요합니다.
3. JWT 저장 위치: 클라이언트 측에서 JWT를 안전하게 저장해야 합니다. 로컬 스토리지는 XSS 공격에 취약할 수 있으므로, 보안이 중요한 애플리케이션에서는 HttpOnly 및 Secure 속성이 설정된 쿠키에 저장하는 것이 더 안전할 수 있습니다.

고객이 안전하게 로그인할 수 있도록 로그인 절차를 설계하고, 이에 필요한 보안 대책을 설명하세요.힌트: JWT(JSON Web Token)와 같은 토큰 기반 인증을 사용하세요. TLS/SSL을 통해 데이터 전송 시 암호화하는 것도 중요한 보안 대책입니다.

d) 재로그인 및 토큰 갱신

1. Refresh Token 사용: 짧은 수명의 Access Token과 더불어, 길게 유효한 Refresh Token을 사용하여 새로운 Access Token을 발급받을 수 있습니다.
Refresh Token은 Access Token과 달리 서버에서만 저장하며, 이로 인해 유효 기간이 길고, 이를 통해 새로운 Access Token을 안전하게 발급받을 수 있습니다.

e) 2단계 인증(2FA) 도입

1. 추가적인 보안 계층: 비밀번호 외에도 SMS 또는 인증 앱을 사용한 2단계 인증을 통해 보안을 강화합니다. 이로 인해 비밀번호가 유출되더라도 추가적인 인증 절차가 필요하여 계정을 보호할 수 있습니다.

f) 로그인 시도 제한

- Brute-Force 공격 방지:
-특정 IP에서의 로그인 시도를 제한하거나, 일정 횟수 이상 실패 시 계정을 잠금 처리하여 Brute-Force 공격을 방지합니다.

g) 로그인 알림

사용자 알림: 새로운 기기나 위치에서 로그인 시도 시 사용자에게 알림을 보내어, 허가되지 않은 로그인 시도를 신속히 감지할 수 있도록 합니다.

결론

이 로그인 절차 설계는 TLS/SSL을 통한 안전한 통신과 JWT를 기반으로 한 토큰 인증을 중심으로 하며, 비밀번호 관리, 토큰 만료 관리, 2단계 인증, 로그인 시도 제한 등의 추가적인 보안 대책을 통해 고객의 계정을 안전하게 보호할 수 있도록 설계되었습니다. 이를 통해 데이터 전송의 기밀성, 무결성, 인증, 그리고 사용자의 안전한 로그인 경험을 보장할 수 있습니다.

SELECT * FROM users WHERE user_id = '<사용자 입력>'; 위와 같은 입력이 취약한 이유를 설명하세요

SQL 인젝션(SQL Injection) 공격에 매우 취약합니다. SQL 인젝션은 공격자가 사용자 입력 필드를 통해 악의적인 SQL 코드를 주입하여 데이터베이스를 비정상적으로 조작하거나, 민감한 데이터를 탈취하는 공격 기법입니다.

1. SQL 인젝션 취약점의 발생 원인
이 쿼리에서는 사용자가 입력한 값이 쿼리에 그대로 포함됩니다. 만약 사용자가 의도적으로 악의적인 SQL 코드를 입력하면, 이 코드가 쿼리의 일부로 실행되어 의도하지 않은 결과를 초래할 수 있습니다. 예를 들어, 다음과 같은 입력이 제공될 수 있습니다:

user_id = '1'; -- '
이 경우, 쿼리는 다음과 같이 변환됩니다:

SELECT * FROM users WHERE user_id = '1'; -- ';
SQL에서 --는 주석을 의미하기 때문에, 그 뒤에 있는 모든 내용이 무시됩니다. 이로 인해 쿼리는 사실상 다음과 같이 실행됩니다:

SELECT * FROM users WHERE user_id = '1';
이 쿼리는 user_id가 1인 사용자의 모든 정보를 반환하게 됩니다.

user_id = '1' OR '1'='1';
이 경우 쿼리는 다음과 같이 변경됩니다:

SELECT * FROM users WHERE user_id = '1' OR '1'='1';
이 쿼리는 user_id가 1이거나 '1'='1'인 모든 레코드를 반환합니다. '1'='1'은 항상 참이기 때문에, 이 쿼리는 users 테이블의 모든 레코드를 반환하게 됩니다.

2. SQL 인젝션으로 인한 보안 위협
- **데이터 유출:** 공격자는 SQL 인젝션을 통해 데이터베이스에서 민감한 데이터를 쉽게 조회할 수 있습니다.
- **데이터 변조:** 공격자는 SQL 인젝션을 사용하여 데이터베이스의 데이터를 삭제, 수정, 삽입할 수 있습니다.
- **권한 상승:** 일부 경우에는 공격자가 데이터베이스의 권한을 상승시켜 시스템 전체를 장악할 수 있습니다.
- **서비스 장애:** 악의적인 입력을 통해 데이터베이스 쿼리를 방해하거나, 무한 루프를 유도하여 서비스 장애를 일으킬 수 있습니다.

3. SQL 인젝션 방지 방법
SQL 인젝션을 방지하기 위해 다음과 같은 보안 대책을 적용해야 합니다:

a) 준비된 쿼리(Prepared Statements) 및 파라미터화된 쿼리 사용
준비된 쿼리와 파라미터화된 쿼리는 사용자 입력이 SQL 쿼리의 구조에 영향을 미치지 않도록 합니다. 이는 쿼리에서 입력된 값을 데이터로 취급하며, 쿼리 자체의 구조를 변경할 수 없게 만듭니다.

```
# Python 예시 (파이썬 데이터베이스 라이브러리 사용)
cursor.execute("SELECT * FROM users WHERE user_id = ?", (user_id,))
```
이 방법을 사용하면, user_id의 값이 쿼리에 문자 그대로 포함되지 않고, SQL 서버에서 적절히 처리됩니다.

b) 입력 검증 및 정규화
사용자 입력을 철저히 검증하고, 예상치 못한 문자나 SQL 명령어가 포함되어 있는지 확인해야 합니다.
- 화이트리스트 사용: 허용된 값만을 받아들이도록 검증합니다.
- 정규 표현식: 사용자 입력을 특정 패턴으로 제한하는 방법을 사용합니다.

c) ORM 사용
ORM(Object-Relational Mapping) 라이브러리는 SQL 쿼리를 자동으로 생성하고, 데이터베이스와의 상호작용을 추상화하여 SQL 인젝션의 위험을 줄입니다.

d) 최소 권한 원칙
데이터베이스 사용자를 최소 권한 원칙에 따라 설정하여, 애플리케이션이 필요한 최소한의 데이터베이스 권한만을 가지도록 합니다. 이를 통해 SQL 인젝션 공격으로 인한 피해를 최소화할 수 있습니다.

결론
SELECT * FROM users WHERE user_id = '<사용자 입력>';는 사용자 입력이 그대로 쿼리에 포함되기 때문에 SQL 인젝션 공격에 매우 취약합니다. 이를 방지하기 위해 준비된 쿼리와 파라미터화된 쿼리, 입력 검증, ORM 사용, 최소 권한 원칙 등의 보안 대책을 적용해야 합니다. 이를 통해 데이터베이스를 SQL 인젝션으로부터 보호하고, 애플리케이션의 보안성을 크게 향상시킬 수 있습니다.

월별 조회 성능 향상과 데이터 삭제를 효율적으로 하기 위한 방안을 제시하세요. **힌트: 파티셔닝(partitioning)을 사용하여 데이터를 월별로 분할하세요.**

call_records 테이블의 월별 조회 성능을 향상시키고, 데이터 삭제를 효율적으로 하기 위해 파티셔닝(partitioning) 기법을 적용할 수 있습니다.
파티셔닝을 통해 테이블을 논리적으로 분할하면, 특정 월에 대한 데이터를 조회하거나 삭제하는 작업을 더 빠르고 효율적으로 수행할 수 있습니다.

1. 파티셔닝 설계

범위 파티셔닝(Range Partitioning)을 사용하여 call_date 컬럼을 기준으로 데이터를 월별로 분할할 수 있습니다.
이는 각 파티션이 특정 월의 데이터를 포함하게 하여, 월별 데이터 조회와 삭제 작업을 최적화할 수 있게 합니다.

```
-- 기존 테이블을 파티셔닝 테이블로 변경
CREATE TABLE call_records (
    id INT,
    user_id INT,
    call_date DATE,
    duration INT,
    destination VARCHAR(50),
    PRIMARY KEY (id, call_date)  -- 파티션 키로 call_date를 포함
)
PARTITION BY RANGE (call_date);

-- 각 월별 파티션을 생성
CREATE TABLE call_records_2023_01 PARTITION OF call_records
FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');

CREATE TABLE call_records_2023_02 PARTITION OF call_records
FOR VALUES FROM ('2023-02-01') TO ('2023-03-01');
-- 이후 다른 월별 파티션을 동일하게 생성
```

1. 수평 파티셔닝 (Horizontal Partitioning): 일명 '샤딩(Sharding)'이라고도 함
   동일한 스키마를 가진 데이터를 여러 테이블/DB로 분할
   예: 고객 데이터를 지역별로 나누기
2. 수직 파티셔닝 (Vertical Partitioning): 테이블의 열을 기준으로 데이터를 분할
   자주 사용하는 컬럼과 그렇지 않은 컬럼을 분리
   예: 고객 기본 정보와 상세 정보를 별도 테이블로 분리
3. 기능적 파티셔닝 (Functional Partitioning): 비즈니스 기능이나 사용 패턴에 따라 데이터 분할
   예: 주문, 재고, 고객 데이터를 별도의 DB로 분리
4. 범위 파티셔닝 (Range Partitioning): 특정 컬럼의 값 범위를 기준으로 데이터 분할
   예: 날짜별로 데이터 분할 (2023년 1월, 2월 등)
5. 리스트 파티셔닝 (List Partitioning): 특정 컬럼의 값 목록을 기준으로 데이터 분할
   예: 지역 코드별로 데이터 분할
6. 해시 파티셔닝 (Hash Partitioning): 해시 함수를 사용하여 데이터를 균등하게 분배
   예: 고객 ID의 해시값을 기준으로 분할
7. 복합 파티셔닝 (Composite Partitioning): 두 가지 이상의 파티셔닝 방법을 조합
   예: 날짜로 범위 파티셔닝 후, 각 파티션 내에서 해시 파티셔닝
8. 라운드 로빈 파티셔닝 (Round-robin Partitioning): 데이터를 순차적으로 각 파티션에 균등하게 분배
   데이터의 균등한 분포가 필요할 때 사용

2. 파티셔닝의 이점

월별 조회 성능 향상: 특정 월의 데이터를 조회할 때 해당 월에 해당하는 파티션만 접근하게 됩니다. 이로 인해 전체 테이블을 검색하는 것보다 훨씬 빠른 쿼리 성능을 얻을 수 있습니다.

데이터 삭제 효율성: 1년이 지난 데이터를 삭제할 때, 개별적으로 행을 삭제하지 않고, 해당 월에 해당하는 파티션을 드롭(DROP)하여 즉시 데이터 삭제를 수행할 수 있습니다. 이 방법은 훨씬 효율적이며, 삭제 성능을 극대화할 수 있습니다.

3. 예시: 월별 데이터 조회

파티셔닝이 적용된 테이블에서 특정 월의 데이터를 조회하는 SQL 예시입니다:

```
-- 2023년 1월의 통화 기록을 조회
SELECT * FROM call_records
WHERE call_date >= '2023-01-01' AND call_date < '2023-02-01';
```

이 쿼리는 해당 월의 파티션에서만 데이터를 검색하기 때문에 성능이 최적화됩니다.

4. 예시: 1년이 지난 데이터 삭제

1년이 지난 데이터를 삭제할 때는, 해당 월의 파티션을 드롭하는 방식으로 간단하게 수행할 수 있습니다:

```
-- 2022년 1월의 통화 기록이 저장된 파티션 삭제
DROP TABLE call_records_2022_01;
```

이 방법은 매우 효율적이며, 대량의 데이터를 일괄 삭제하는 데 걸리는 시간을 크게 줄일 수 있습니다.

5. 자동화된 파티션 관리

매달 새로운 데이터를 삽입하기 전에, 새로운 월에 대한 파티션을 자동으로 생성하도록 스크립트를 작성할 수 있습니다. 또한, 1년이 지난 데이터 파티션을 자동으로 삭제하도록 스케줄링할 수 있습니다.

```
-- 예: 새로운 월별 파티션 생성
CREATE TABLE call_records_2023_03 PARTITION OF call_records
```

월별 조회 성능 향상과 데이터 삭제를 효율적으로 하기 위한 방안을 제시하세요. **힌트: 파티셔닝(partitioning)을 사용하여 데이터를 월별로 분할하세요.**

파티셔닝을 사용하여 call_records 테이블을 월별로 분할하는 방안의 장점은 다음과 같습니다:

1. 쿼리 성능 향상
- **선택적 데이터 접근:** 파티셔닝을 통해 데이터를 월별로 분할하면, 특정 월의 데이터를 조회할 때 해당 월에 해당하는 파티션만 접근합니다.
이는 전체 테이블을 검색하는 것보다 훨씬 적은 데이터를 스캔하기 때문에 쿼리 성능이 크게 향상됩니다.

- **인덱스 크기 감소:** 각 파티션은 독립적인 인덱스를 가질 수 있어, 파티션의 데이터 양이 줄어들수록 인덱스 크기도 줄어듭니다.
작은 인덱스는 더 빠른 검색을 가능하게 하며, 인덱스 유지 비용도 줄어듭니다.

2. 데이터 관리 효율성
- **효율적인 데이터 삭제:** 1년이 지난 데이터를 삭제할 때, 각 월에 해당하는 파티션을 드롭(DROP)하는 방식으로 데이터를 일괄 삭제할 수 있습니다.
이는 대량의 데이터를 개별적으로 삭제하는 것보다 훨씬 빠르고, 성능에 미치는 영향도 적습니다.

- **데이터 정리 및 유지보수 간소화:** 파티션을 사용하면 오래된 데이터를 정리하고, 새로운 데이터를 추가하는 작업이 간편해집니다.
특히 일정 주기로 오래된 데이터를 삭제하거나 아카이빙하는 작업을 손쉽게 관리할 수 있습니다.

3. 테이블 및 인덱스의 성능 최적화
- **작은 파티션 크기 유지:** 각 파티션이 월별 데이터만을 포함하기 때문에, 테이블과 인덱스의 크기가 제한됩니다.
작은 테이블과 인덱스는 메모리 내에서 효율적으로 관리될 수 있으며, 디스크 I/O도 줄어듭니다.

- **병렬 처리 가능성:** 여러 파티션이 존재하는 경우, 데이터베이스 시스템은 병렬 처리를 통해 성능을 더욱 향상시킬 수 있습니다.
예를 들어, 서로 다른 파티션에서 데이터를 동시에 읽거나 쓰는 작업을 병렬로 수행할 수 있습니다.

4. 확장성 및 관리 용이성
- **대규모 데이터 처리:** 통화 기록과 같은 대규모 데이터셋을 효과적으로 관리할 수 있습니다.
파티셔닝은 데이터 양이 증가함에 따라 성능 문제를 최소화하면서 테이블을 확장할 수 있는 방법을 제공합니다.

- **장기적 데이터 관리:** 파티셔닝을 사용하면 시간이 지나면서 데이터가 계속 축적될 경우에도 효율적으로 데이터를 관리할 수 있습니다.
예를 들어, 오래된 데이터를 주기적으로 삭제하거나, 파티션을 통해 쉽게 아카이빙할 수 있습니다.

5. 데이터베이스 유지 관리 비용 절감
- **효율적인 백업 및 복구:** 파티셔닝된 테이블은 각 파티션을 독립적으로 백업하거나 복구할 수 있습니다. 이를 통해 전체 테이블을 백업할 필요 없이,
특정 파티션만 백업 또는 복구하는 작업이 가능해집니다. 이는 데이터베이스 관리자의 작업 부담을 줄여줍니다.

- **성능 튜닝 간소화:** 파티셔닝을 통해 성능 문제를 특정 파티션으로 국한시킬 수 있어, 문제 해결 및 성능 튜닝 작업이 훨씬 쉬워집니다.

6. 데이터 접근 제어 및 보안 강화

- **파티션 단위의 접근 제어:** 파티셔닝을 통해 특정 파티션에 대한 접근 권한을 별도로 설정할 수 있습니다.
예를 들어, 특정 월의 데이터에 대해서만 접근 권한을 부여하거나 제한할 수 있습니다.

- **데이터 격리:** 파티셔닝된 데이터는 논리적으로 분리되어 있으므로, 데이터 손상 또는 부정 접근 시 피해를 최소화할 수 있습니다.
이는 특히 민감한 데이터가 포함된 경우 보안 측면에서 유리합니다.

결론

제시된 파티셔닝 방안은 통화 기록 데이터의 월별 조회 성능을 크게 향상시키고, 1년이 지난 데이터를 효율적으로 삭제할 수 있는 방법을 제공합니다.
이를 통해 대량의 데이터를 관리하는 데 필요한 성능 최적화, 유지 보수 비용 절감, 확장성 등의 이점을 제공하며, 장기적인 데이터 관리 및 성능 튜닝 작업을 간소화할 수 있습니다.

데이터 삭제를 자동화하는 방안을 제시하세요.힌트**: 스케줄러나 이벤트 트리거를 사용하여 일정 기간마다 자동으로 오래된 데이터를 삭제하는 스크립트를 실행할 수 있습니다.**

데이터 삭제를 자동화하는 방안을 설계하기 위해서는 1년이 지난 통화 기록 데이터를 매달 자동으로 삭제하는 메커니즘을 구축할 필요가 있습니다.
이를 위해 데이터베이스 관리 시스템(DBMS)의 스케줄링 기능을 활용하여 자동화할 수 있습니다.

1. 자동 데이터 삭제 개요

매달 말일 또는 새로운 달의 첫날, 1년이 지난 통화 기록 파티션을 자동으로 삭제하는 작업을 스케줄링합니다.
이를 통해 관리자는 매달 반복적으로 데이터를 수동으로 삭제할 필요가 없으며, 데이터베이스는 항상 최신의 데이터를 유지하게 됩니다.

2. 자동화 방법 1: 데이터베이스 스케줄러 사용

대부분의 DBMS(예: PostgreSQL, MySQL, Oracle 등)에는 정기적으로 SQL 스크립트를 실행할 수 있는 스케줄링 기능이 있습니다.

a) PostgreSQL에서의 구현 예시

PostgreSQL에서는 pg_cron 또는 pgAgent와 같은 스케줄러 확장을 사용하여 정기적으로 SQL 작업을 실행할 수 있습니다.

pg_cron 확장 설치 (PostgreSQL 12+)

```
CREATE EXTENSION IF NOT EXISTS pg_cron;
```

데이터 삭제 스크립트 작성

매달 새로운 파티션이 생성되고, 1년이 지난 파티션이 삭제되도록 스크립트를 작성합니다. 예를 들어, 2022년 1월의 파티션을 삭제하는 SQL은 다음과 같습니다:

```
DROP TABLE IF EXISTS call_records_2022_01;
```

스케줄러 작업 생성

매달 말일 23:59에 1년이 지난 파티션을 자동으로 삭제하는 작업을 예약합니다.

```
SELECT cron.schedule('monthly_partition_cleanup', '59 23 L * *', $$ DROP TABLE IF EXISTS call_records_YYYY_MM;$$);
```

여기서 YYYY_MM은 삭제할 연도와 월을 나타내며, 실제 구현에서는 해당 월을 계산하는 로직이 필요합니다. 스크립트에서 동적으로 삭제할 파티션 이름을 계산해야 합니다.

b) MySQL에서의 구현 예시

MySQL에서는 EVENT SCHEDULER를 사용하여 정기적으로 SQL 작업을 실행할 수 있습니다.

EVENT SCHEDULER 활성화

```
SET GLOBAL event_scheduler = ON;
```

데이터 삭제 이벤트 생성

매달 1일 자정에 1년이 지난 파티션을 자동으로 삭제하는 이벤트를 생성합니다.

```
CREATE EVENT IF NOT EXISTS delete_old_partitions
ON SCHEDULE EVERY 1 MONTH
STARTS '2023-02-01 00:00:00'
DO
BEGIN
    SET @partition_name = CONCAT('call_records_', DATE_FORMAT(DATE_SUB(CURDATE(), INTERVAL 1 YEAR), '%Y_%m'));
    SET @drop_query = CONCAT('DROP TABLE IF EXISTS ', @partition_name);
    PREPARE stmt FROM @drop_query;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
END;
```

이 이벤트는 매달 1일에 실행되어 1년 전의 데이터를 자동으로 삭제합니다

데이터 삭제를 자동화하는 방안을 제시하세요.힌트**: 스케줄러나 이벤트 트리거를 사용하여 일정 기간마다 자동으로 오래된 데이터를 삭제하는 스크립트를 실행할 수 있습니다.**

3. 자동화 방법 2: 운영 시스템 스크립트 및 CRON 사용

데이터베이스 스케줄러 외에도 운영체제의 CRON 작업을 활용하여 삭제 작업을 자동화할 수 있습니다. 이는 데이터베이스 스케줄링 기능이 제한적인 경우 유용할 수 있습니다.

a) CRON 작업 생성 (리눅스 환경 예시)

SQL 스크립트 생성

데이터 삭제를 위한 SQL 스크립트를 파일로 작성합니다. 예를 들어, delete_old_partitions.sql 파일을 생성합니다:

```
DROP TABLE IF EXISTS call_records_2022_01;
```

스크립트에서 YYYY_MM 부분은 스크립트를 실행할 때 동적으로 계산될 수 있도록 합니다.

CRON 작업 설정

매달 1일 자정에 이 SQL 스크립트를 실행하는 CRON 작업을 설정합니다:

```
0 0 1 * * psql -U username -d database_name -f /path/to/delete_old_partitions.sql
```

psql 명령어를 사용하여 스크립트를 실행하고, 1년이 지난 데이터를 자동으로 삭제합니다.

4. 자동화 방법 3: 어플리케이션 레벨에서 처리

만약 데이터베이스의 스케줄러 기능이 부족하거나 더 복잡한 로직이 필요한 경우, 어플리케이션 레벨에서 데이터 삭제를 처리할 수 있습니다.

어플리케이션 코드 작성

주기적으로 실행되는 작업을 어플리케이션에 추가하여, 1년이 지난 파티션을 삭제하도록 합니다. 이는 Java, Python, Node.js 등의 언어로 구현할 수 있습니다.

스케줄러 라이브러리 사용

어플리케이션에서 주기적으로 작업을 실행할 수 있는 라이브러리(예: Python의 schedule 라이브러리, Java의 Quartz 스케줄러 등)를 사용하여 정기적인 파티션 삭제 작업을 자동화합니다.

결론

위에서 설명한 다양한 자동화 방법을 통해, 1년이 지난 통화 기록 데이터를 효율적으로 삭제할 수 있습니다.
데이터베이스의 스케줄러 기능을 사용하거나, 운영체제의 CRON 작업, 또는 어플리케이션 레벨에서의 스케줄링을 통해 자동화된 데이터 관리 프로세스를 구현할 수 있습니다.
이러한 자동화는 데이터베이스의 성능을 유지하고, 오래된 데이터를 체계적으로 정리하는 데 중요한 역할을 합니다.

대규모 고객에게 효율적으로 Push 알림을 전송하면서도 서버 부하를 최소화하고, 메시지가 신속하게 전달되도록 해야 합니다. 알림을 제공하기 위해 웹소켓(WebSocket) 기술 검토

WebSocket은 서버와 클라이언트 간에 실시간 양방향 통신을 가능하게 하는 프로토콜로, Push 알림과 같은 실시간 메시지 전송에 매우 적합합니다.
K 통신사가 수백만 명의 고객에게 효율적으로 Push 알림을 제공하기 위해 WebSocket을 사용한다면, 다음과 같은 기본적인 방법과 단계로 WebSocket 통신을 설정할 수 있습니다.

1. WebSocket 기본 개념

WebSocket은 HTTP와 달리 클라이언트와 서버 간의 지속적인 연결을 유지하여, 양방향으로 데이터를 실시간으로 주고받을 수 있습니다.
한 번 연결이 수립되면, 클라이언트와 서버는 데이터를 서로에게 푸시(Push)할 수 있으며, 이를 통해 지연 없이 실시간 통신이 가능합니다.

2. 서버 측 WebSocket 설정

a) WebSocket 서버 시작

서버에서 WebSocket을 사용하려면 WebSocket 서버를 설정해야 합니다. 예를 들어, Node.js를 사용하여 WebSocket 서버를 설정하는 방법은 다음과 같습니다:

Node.js와 WebSocket 라이브러리 설치

```
npm install ws
```

WebSocket 서버 코드 작성

```
javascript
Copy code
const WebSocket = require('ws');

// WebSocket 서버 생성, 8080 포트에서 대기
const wss = new WebSocket.Server({ port: 8080 });

// 클라이언트 연결 시 이벤트 처리
wss.on('connection', ws => {
    console.log('클라이언트 연결됨');

    // 클라이언트로부터 메시지 수신 시 처리
    ws.on('message', message => {
        console.log(`수신된 메시지: ${message}`);
    });

    // 클라이언트에게 메시지 전송
    ws.send('서버에 연결되었습니다.');
});

console.log('WebSocket 서버가 8080 포트에서 실행 중...');
```

b) 클라이언트 관리

다중 클라이언트 지원:

WebSocket 서버는 다수의 클라이언트를 동시에 처리할 수 있어야 합니다.
ws 객체를 통해 연결된 각 클라이언트에게 메시지를 전송하거나, 특정 클라이언트에게만 메시지를 전송할 수 있습니다.

WebSocket은 웹 브라우저와 서버 간의 실시간,
양방향, 전이중(full-duplex) 통신을 가능하게 하는
프로토콜입니다.

1. 개념
   - HTTP 프로토콜 위에서 작동하는 별도의 프로토콜
   - 단일 TCP 연결을 통해 지속적인 연결 유지
2. 작동 방식:
   - 초기 연결은 HTTP 핸드셰이크로 시작
   - 연결 수립 후 WebSocket 프로토콜로 전환
   - 클라이언트와 서버 간 실시간 데이터 교환
3. 주요 특징:
   - 양방향 통신: 서버와 클라이언트 모두 자유롭게 메시지 전송 가능
   - 실시간 데이터 전송: 지연 시간 최소화
   - 효율적인 리소스 사용: 연결 유지에 적은 오버헤드
   - 크로스 도메인 통신 지원
4. 사용 사례:
   - 실시간 채팅 애플리케이션
   - 라이브 피드 및 알림 시스템
   - 실시간 협업 도구
   - 게임과 같은 실시간 상호작용 애플리케이션
5. 장점:
   - HTTP polling에 비해 네트워크 트래픽 감소
   - 서버 부하 감소
   - 실시간 데이터 업데이트 가능
6. 단점:
   - 모든 브라우저에서 지원되지 않을 수 있음 (오래된 버전)
   - 방화벽이나 프록시 서버에서 차단될 수 있음
7. 구현:
   - 클라이언트 측: JavaScript의 WebSocket API 사용
   - 서버 측: Node.js의 ws, Socket.IO 등의 라이브러리 사용
8. 보안:
   - wss:// (WebSocket Secure) 프로토콜을 통한 암호화 지원
9. HTTP와의 차이:
   - HTTP: 요청-응답 모델, 연결 비지속성
   - WebSocket: 지속적 연결, 양방향 통신
10. 폴백(Fallback) 메커니즘:
   - WebSocket 지원되지 않을 경우 long polling 등의 대체 기술 사용 가능

대규모 고객에게 효율적으로 Push 알림을 전송하면서도 서버 부하를 최소화하고, 메시지가 신속하게 전달되도록 해야 합니다. 알림을 제공하기 위해 웹소켓(WebSocket) 기술 검토

브로드캐스트 메시지:

특정 이벤트 발생 시 모든 클라이언트에게 알림을 전송할 수 있도록 브로드캐스트 기능을 구현합니다.

```
// 모든 클라이언트에게 메시지 브로드캐스트
function broadcast(data) {
    wss.clients.forEach(client => {
        if (client.readyState === WebSocket.OPEN) {
            client.send(data);
        }
    });
}
// 예: 특정 이벤트 발생 시
broadcast('중요 공지: 새로운 프로모션이 시작되었습니다.');
```

3. 클라이언트 측 WebSocket 설정

클라이언트(모바일 앱 또는 웹 애플리케이션)에서 WebSocket 서버에 연결하는 방법은 다음과 같습니다:

WebSocket 객체 생성 및 서버에 연결

```
// WebSocket 객체 생성 및 서버에 연결
const socket = new WebSocket('ws://localhost:8080');

// 서버와의 연결이 성공했을 때
socket.onopen = function(event) {
    console.log('WebSocket 연결 성공');
    // 서버로 메시지 전송
    socket.send('클라이언트에서 메시지를 보냅니다.');
};

// 서버로부터 메시지를 수신했을 때
socket.onmessage = function(event) {
    console.log('서버로부터 수신된 메시지:', event.data);
};

// WebSocket 연결이 닫혔을 때
socket.onclose = function(event) {
    console.log('WebSocket 연결이 닫혔습니다.');
};

// WebSocket 에러가 발생했을 때
socket.onerror = function(error) {
    console.error('WebSocket 오류:', error);
};
```

클라이언트에서 메시지 전송 및 수신

클라이언트는 WebSocket 객체를 통해 서버에 메시지를 전송하거나, 서버로부터 수신된 메시지를 처리할 수 있습니다.

대규모 고객에게 효율적으로 Push 알림을 전송하면서도 서버 부하를 최소화하고, 메시지가 신속하게 전달되도록 해야 합니다. 알림을 제공하기 위해 웹소켓(WebSocket) 기술 검토

4. 실시간 메시지 전송

a) 서버에서 클라이언트로 Push 알림 전송

서버에서 특정 이벤트(예: 새로운 프로모션 시작) 발생 시, 연결된 모든 클라이언트에게 실시간으로 Push 알림을 전송합니다.

```
// 예시: 특정 조건에서 모든 클라이언트에게 Push 알림 전송
function sendPromotionNotification(promotionDetails) {
    const message = `새로운 프로모션: ${promotionDetails.title} - ${promotionDetails.description}`;
    broadcast(message);
}
```

b) 클라이언트에서 서버로 메시지 전송

클라이언트는 필요에 따라 서버로 실시간 메시지를 전송할 수 있습니다. 예를 들어, 사용자가 특정 기능을 사용했을 때 서버에 알릴 수 있습니다.

```
// 클라이언트가 서버로 메시지 전송
socket.send('사용자가 프로모션에 관심을 보였습니다.');
```

5. 부하 관리 및 확장성

WebSocket 서버는 많은 클라이언트를 동시에 처리할 수 있어야 하므로, 다음과 같은 부하 관리 및 확장성 전략을 적용할 수 있습니다:

로드 밸런싱:

여러 WebSocket 서버 인스턴스를 실행하고, 로드 밸런서를 통해 클라이언트 요청을 분산시킵니다.

스케일 아웃:

서버 부하가 증가할 경우, WebSocket 서버 인스턴스를 추가하여 수평 확장을 통해 처리 용량을 늘릴 수 있습니다.

메시지 큐:

메시지 브로드캐스트 시 메시지 큐(예: RabbitMQ, Apache Kafka)를 사용하여 메시지 전달의 신뢰성을 높이고, 서버 부하를 줄일 수 있습니다.

결론

K 통신사는 WebSocket을 사용하여 수백만 명의 고객에게 실시간으로 Push 알림을 제공할 수 있습니다.
WebSocket을 사용한 서버와 클라이언트 간의 양방향 통신은 실시간 메시지 전달에 적합하며, 서버 부하를 최소화하면서도 신속한 알림 전송을 가능하게 합니다.
로드 밸런싱과 메시지 큐와 같은 부하 관리 기술을 통해 확장성을 확보하고, 많은 고객에게 안정적으로 알림 서비스를 제공할 수 있습니다.

웹소켓 연결에서 발생할 수 있는 보안 문제들을 제시하고, 각각의 문제 해결 방안을 제시하세요.

WebSocket 연결에서 발생할 수 있는 보안 문제는 여러 가지가 있으며, 이러한 문제들은 웹 애플리케이션의 보안을 크게 위협할 수 있습니다.

1. 교차 사이트 스크립팅(XSS) 공격

문제

- WebSocket 연결에서 전송되는 데이터가 충분히 검증되지 않으면, 공격자가 악성 스크립트를 주입하여 클라이언트 측에서 실행되도록 할 수 있습니다.
  이로 인해 사용자의 세션이 탈취되거나 악성 코드가 실행될 수 있습니다.

해결 방안:

- 데이터 검증 및 인코딩:
  . 서버에서 클라이언트로 전송되는 모든 데이터는 HTML 인코딩을 수행하여 스크립트가 실행되지 않도록 해야 합니다.
  . 입력된 데이터에 대해 유효성 검사를 철저히 수행하고, 악의적인 입력을 차단합니다.
- 콘텐츠 보안 정책(Content Security Policy, CSP) 설정:
  . CSP를 적용하여 외부 스크립트의 실행을 제한하고, XSS 공격을 방지합니다.

```
// 데이터 인코딩 예시
function sanitizeInput(input) {
    return input.replace(/&/g, '&amp;')
                .replace(/</g, '&lt;')
                .replace(/>/g, '&gt;')
                .replace(/"/g, '&quot;')
                .replace(/'/g, '&#x27;')
                .replace(/₩//g, '&#x2F;');
}

// 사용 예시
const sanitizedMessage = sanitizeInput(userInput);
```

2. 웹소켓 연결의 도청(Interception) 및 중간자 공격(Man-in-the-Middle Attack, MITM)

문제:

- WebSocket 연결이 암호화되지 않은 경우, 공격자는 트래픽을 도청하거나 변조할 수 있습니다. 이를 통해 민감한 정보를 탈취하거나, 데이터 무결성을 훼손할 수 있습니다.

해결 방안:

- SSL/TLS를 통한 WebSocket 연결 암호화:
  . WebSocket 연결을 wss:// 프로토콜(SSL/TLS를 사용한 WebSocket)로 설정하여, 모든 데이터가 암호화된 채로 전송되도록 합니다. 이를 통해 데이터 도청과 중간자 공격을 방지할 수 있습니다.

```
// WebSocket 연결 설정 예시 (SSL/TLS 사용)
const socket = new WebSocket('wss://example.com/socket');
```

- SSL/TLS 인증서 관리:
  . 서버 측에서는 신뢰할 수 있는 CA(Certificate Authority)에서 발급한 SSL/TLS 인증서를 사용하여, 클라이언트가 서버의 신원을 확인할 수 있도록 합니다.

3. 무단 접근 및 세션 하이재킹(Session Hijacking)

문제:

공격자가 WebSocket 연결에 무단으로 접근하거나 세션 정보를 탈취하여 사용자의 세션을 가로채면, 사용자 계정에 무단 접근이 발생할 수 있습니다.

웹소켓 연결에서 발생할 수 있는 보안 문제들을 제시하고, 각각의 문제 해결 방안을 제시하세요.

해결 방안:
- 강력한 인증 및 세션 관리:
  . WebSocket 연결 전에 사용자가 적절한 인증 절차를 거치도록 하며, 인증된 세션에서만 WebSocket을 사용할 수 있게 합니다.
  . JWT(JSON Web Token)와 같은 토큰 기반 인증을 사용하여, 클라이언트의 신원을 확인하고, 토큰을 주기적으로 갱신하여 세션 하이재킹을 방지합니다.

세션 타임아웃 설정:
- 일정 기간 동안 사용되지 않은 WebSocket 세션을 자동으로 종료하여 세션 하이재킹 위험을 줄입니다.

4. DOS(서비스 거부) 공격

문제:
공격자가 대량의 WebSocket 연결을 생성하여 서버 리소스를 소모시킬 경우, 서버가 과부하 상태에 빠져 정상적인 서비스를 제공할 수 없게 됩니다.

해결 방안:
- 연결 제한 및 속도 제한 설정:
  . 동일한 IP 주소에서 오는 WebSocket 연결 수를 제한하거나, 특정 시간 내에 생성할 수 있는 연결의 수를 제한하여 DOS 공격을 방지합니다.
- 캡차(CAPTCHA) 및 인증 절차:
  . WebSocket 연결 전 캡차를 도입하여 자동화된 공격을 방지하고, 사용자가 사람임을 확인하는 절차를 추가할 수 있습니다.

웹 애플리케이션 방화벽(WAF) 사용:
WAF를 사용하여 의심스러운 트래픽을 필터링하고, 비정상적인 WebSocket 연결 시도를 차단할 수 있습니다.

5. 메시지 위조(Forged Message) 및 인젝션 공격

문제:
공격자가 서버나 클라이언트로 위조된 메시지를 보내서 의도하지 않은 동작을 유도할 수 있습니다. 예를 들어, SQL 인젝션이나 명령어 인젝션과 같은 공격이 발생할 수 있습니다.

해결 방안:
- 메시지 검증 및 데이터 인코딩:
  . 서버와 클라이언트가 주고받는 모든 메시지에 대해 철저히 검증하고, 예상치 못한 데이터를 포함한 메시지는 차단합니다.
  . 클라이언트로부터 입력된 데이터는 항상 인코딩 및 필터링하여 SQL 인젝션, 명령어 인젝션 등을 방지합니다.

명확한 프로토콜 정의:
- WebSocket 메시지의 포맷과 내용에 대한 명확한 프로토콜을 정의하고, 클라이언트와 서버가 이를 엄격히 준수하도록 합니다.

결론

WebSocket 연결에서 발생할 수 있는 보안 문제는 다각적으로 존재하며, 이를 방지하기 위해 다양한 보안 대책이 필요합니다.
SSL/TLS를 통한 암호화, 강력한 인증, 메시지 검증 및 인코딩, 연결 제한 및 세션 관리 등과 같은 방법을 통해 WebSocket 연결의 보안을 강화할 수 있습니다.
이러한 대책을 적절히 구현함으로써 K 통신사는 고객에게 안전하고 신뢰할 수 있는 Push 알림 서비스를 제공할 수 있습니다.

이 웹 애플리케이션은 다양한 데이터 조회 및 계산 작업을 수행하며, 많은 수의 고객 요청을 처리해야 합니다. 특히, 특정 고객 정보 조회 및 통계 계산이 빈번히 발생하며, 이로 인해 데이터베이스에 큰 부하가 가해지고 있습니다. 이를 해결하기 위해 메모리 캐시를 활용하여 성능을 최적화하고자 합니다.

캐시의 효율성을 높이기 위한 캐시 갱신 정책과 캐시 만료 정책
통신사 K의 고객 서비스 웹 애플리케이션에서 메모리 캐시를 활용하여 성능을 최적화하기 위해, 적절한 캐시 갱신 정책과 캐시 만료 정책을 설계하는 것이 중요합니다.
이들 정책은 캐시의 효율성을 높이고, 데이터의 최신성을 유지하는 데 중요한 역할을 합니다.

1. 캐시 갱신 정책(Cache Eviction Policy)
캐시 갱신 정책은 캐시에 저장된 데이터를 어떻게 갱신하고, 캐시가 가득 찼을 때 어떤 데이터를 제거할지 결정하는 방법입니다. 다음은 대표적인 캐시 갱신 정책입니다:

a) LRU (Least Recently Used)
설명: LRU는 가장 오랫동안 사용되지 않은 데이터를 먼저 제거하는 정책입니다. 즉, 최근에 자주 사용된 데이터는 캐시에 남아 있게 되고, 오랫동안 사용되지 않은 데이터는 제거됩니다.

적용 시나리오: 고객 정보나 통계 계산 결과와 같은 데이터가 자주 조회되고, 일정 시간 내에 재사용될 가능성이 높은 경우 LRU 정책이 유효합니다.
이를 통해 최신 데이터를 캐시에 유지하며, 자주 조회되는 데이터의 접근 속도를 높일 수 있습니다.

장점:
- 자주 사용되는 데이터를 캐시에 오래 유지할 수 있습니다.
- 캐시 메모리 효율성을 높일 수 있습니다.

단점:
- 자주 접근되지 않더라도, 특정 패턴의 접근이 발생할 경우, 필요한 데이터가 캐시에서 제거될 수 있습니다.

b) LFU (Least Frequently Used)
설명: LFU는 사용 빈도가 가장 낮은 데이터를 먼저 제거하는 정책입니다. 사용 횟수를 기준으로 캐시 데이터를 갱신하므로, 자주 사용되지 않는 데이터는 빠르게 제거됩니다.

적용 시나리오: 특정 고객 정보나 통계 계산 결과가 반복적으로 자주 조회되는 경우, LFU 정책이 효과적일 수 있습니다.
이 정책은 캐시에 자주 조회되는 데이터를 유지하여, 동일한 데이터에 대한 중복 연산을 줄일 수 있습니다.

장점: 자주 조회되는 데이터를 장기적으로 캐시에 유지할 수 있습니다.

단점: 한 번 사용된 데이터가 계속해서 캐시에 유지될 수 있으며, 일시적으로 자주 사용된 데이터가 오래 남아 있을 수 있습니다.

c) FIFO (First In, First Out)
설명: FIFO는 캐시에 가장 먼저 들어온 데이터를 먼저 제거하는 정책입니다. 데이터가 캐시에 들어온 순서대로 제거되며, 오래된 데이터가 삭제됩니다.

적용 시나리오: 모든 데이터가 거의 동일하게 사용되며, 데이터의 사용 패턴이 특정하지 않을 때 사용될 수 있습니다. 그러나 LRU나 LFU에 비해 효율성이 떨어질 수 있습니다.

장점: 구현이 간단하고, 예측 가능한 방식으로 동작합니다.

단점: 데이터의 사용 빈도나 최근 사용 여부를 고려하지 않으므로, 중요한 데이터가 제거될 수 있습니다.

이 웹 애플리케이션은 다양한 데이터 조회 및 계산 작업을 수행하며, 많은 수의 고객 요청을 처리해야 합니다. 특히, 특정 고객 정보 조회 및 통계 계산이 빈번히 발생하며, 이로 인해 데이터베이스에 큰 부하가 가해지고 있습니다. 이를 해결하기 위해 메모리 캐시를 활용하여 성능을 최적화하고자 합니다.

2. 캐시 만료 정책(Cache Expiration Policy)

캐시 만료 정책은 캐시에 저장된 데이터가 일정 시간이 지나면 자동으로 삭제되거나 갱신되도록 하는 정책입니다.
이는 데이터의 최신성을 유지하고, 캐시 내 데이터의 유효성을 보장하는 데 중요한 역할을 합니다.

a) TTL (Time-To-Live)

설명: TTL은 캐시된 데이터가 일정 시간이 지나면 만료되도록 설정하는 정책입니다. TTL을 설정하면, 데이터가 일정 시간이 지나면 자동으로 캐시에서 제거됩니다.

적용 시나리오: 고객 정보나 통계 데이터의 최신성이 중요하지만, 데이터의 생성 비용이 높은 경우 사용됩니다. 고객의 현재 상태 정보나 주기적으로 변하는 통계 데이터는 일정 시간 후 갱신

장점: 데이터의 최신성을 유지할 수 있습니다. 일정 시간이 지나면 캐시가 자동으로 갱신되어, 오래된 데이터가 제거됩니다.

단점: 데이터가 빈번하게 갱신될 경우, TTL 설정에 따라 캐시 히트율이 낮아질 수 있습니다.

b) Absolute Expiration (절대 만료 시간)

설명: 캐시 데이터에 특정 만료 시간을 지정하는 정책입니다. 절대 만료 시간은 데이터가 캐시에 저장된 시점과 관계없이 특정 시점에 데이터가 만료되도록 합니다.

적용 시나리오: 특정 이벤트나 일정이 있는 데이터에 사용될 수 있습니다. 예를 들어, 특정 프로모션 정보는 프로모션 종료 시점에 자동으로 캐시에서 제거되도록 설정할 수 있습니다.

장점: 특정 시점에 데이터를 만료시키고, 정확한 시간에 캐시를 갱신할 수 있습니다.

단점: 설정된 시간이 지나면 캐시가 만료되므로, 데이터 접근이 잦은 경우에는 성능이 떨어질 수 있습니다.

c) Sliding Expiration (슬라이딩 만료 시간)

설명: 슬라이딩 만료 시간은 캐시에 접근할 때마다 만료 시간이 연장되는 정책입니다. 즉, 캐시 데이터가 일정 시간 동안 접근되지 않으면 만료되지만, 접근할 때마다 만료 시간이 갱신됩니다.

적용 시나리오: 자주 사용되는 데이터에 적합하며, 사용 빈도에 따라 캐시 데이터를 유지하고자 할 때 유용합니다.

장점: 자주 사용되는 데이터를 캐시에 오래 유지할 수 있습니다.

단점: 데이터가 계속해서 캐시에 남아 있을 수 있어, 중요한 데이터를 위한 캐시 공간이 부족해질 수 있습니다.

3. 캐시 갱신 및 만료 정책 결합

통신사 K의 고객 서비스 웹 애플리케이션에서 효율적인 캐싱을 위해 다음과 같은 결합된 전략을 사용할 수 있습니다:

LRU + TTL: 자주 조회되는 데이터를 캐시에 유지하면서도, 일정 시간 후 자동으로 제거되는 정책을 사용할 수 있습니다. 이 조합은 데이터의 최신성을 유지하면서도 캐시 메모리의 효율성을 높입니다.
LFU + Sliding Expiration: 자주 사용되는 데이터가 캐시에 오래 남아 있도록 하되, 일정 기간 사용되지 않으면 자동으로 제거되도록 설정할 수 있습니다. 이를 통해 중요한 데이터를 캐시에 오랫동안 유지
FIFO + Absolute Expiration: 데이터의 사용 순서를 기준으로 캐시에서 제거하되, 특정 시점에 만료되도록 설정하여, 일정 기간 동안만 유효한 데이터를 관리할 수 있습니다.

결론

캐시의 효율성을 높이기 위해 적절한 캐시 갱신 정책과 만료 정책을 선택하는 것은 매우 중요합니다. LRU와 LFU 같은 갱신 정책을 사용하여 자주 사용되는 데이터를 캐시에 오래 유지하면서도, TTL이나 절대 만료 시간을 통해 데이터의 최신성을 보장할 수 있습니다. 이러한 전략을 통해 통신사 K는 데이터베이스 부하를 줄이고, 웹 애플리케이션의 성능을 최적화할 수 있습니다.

RESTful API를 설계할 때 고려해야 할 주요 요소를 설명하세요.

RESTful API를 설계할 때는 여러 요소를 고려하여 효율적이고 안전한 API를 구축해야 합니다.
이러한 요소는 리소스 식별, HTTP 메서드 사용, 상태 코드 관리, 데이터 형식, 보안, 그리고 성능 최적화 등이 포함됩니다.

1. 리소스 설계 (URI 설계)

리소스 식별: RESTful API에서 리소스는 URI(Uniform Resource Identifier)를 통해 식별됩니다. URI는 자원의 위치를 명확히 표현해야 하며, 직관적이어야 합니다.
예: /customers, /customers/{customer_id}

명사형 URI: URI는 리소스를 나타내는 명사로 구성되어야 하며, HTTP 메서드를 사용하여 동작을 정의해야 합니다.
예: 고객 목록 조회: GET /customers, 고객 생성: POST /customers, 고객 수정: PUT /customers/{customer_id}, 고객 삭제: DELETE /customers/{customer_id}

계층적 구조: URI는 리소스 간의 관계를 나타내기 위해 계층적으로 구성될 수 있습니다.
예: 특정 고객의 주문 조회: GET /customers/{customer_id}/orders

2. HTTP 메서드 사용

GET: 리소스를 조회할 때 사용합니다. 서버에서 데이터를 가져올 때 GET 요청을 사용하며, 서버에 데이터를 변경하지 않습니다.
예: GET /customers (모든 고객 조회), 예: GET /customers/{customer_id} (특정 고객 조회)

POST: 새로운 리소스를 생성할 때 사용합니다. 클라이언트가 서버에 데이터를 전송하여 새로운 리소스를 생성하도록 요청합니다.
예: POST /customers (새로운 고객 생성)

PUT: 기존 리소스를 수정할 때 사용합니다. 전체 리소스를 대체하는 방식으로 동작하며, 자원이 존재하지 않으면 새로 생성할 수도 있습니다.
예: PUT /customers/{customer_id} (특정 고객 정보 수정)

PATCH: 기존 리소스의 일부를 수정할 때 사용합니다. 자원의 일부 속성만 변경할 경우 사용됩니다.
예: PATCH /customers/{customer_id} (특정 고객 정보 부분 수정)

DELETE: 특정 리소스를 삭제할 때 사용합니다.
예: DELETE /customers/{customer_id} (특정 고객 삭제)

3. HTTP 상태 코드

- 200 OK: 요청이 성공적으로 처리된 경우 사용합니다. GET, PUT, DELETE 요청의 성공적인 응답에서 사용됩니다.
- 201 Created: POST 요청으로 새로운 리소스가 성공적으로 생성된 경우 사용합니다. 생성된 리소스의 URI를 Location 헤더에 포함할 수 있습니다.
- 204 No Content: 요청이 성공적으로 처리되었으나, 응답 본문에 반환할 데이터가 없는 경우 사용합니다. DELETE 요청 후에 자주 사용됩니다.
- 400 Bad Request: 클라이언트의 요청이 잘못되었을 때 사용합니다. 요청 데이터의 형식이나 값이 유효하지 않은 경우 반환됩니다.
- 401 Unauthorized: 인증이 필요하지만 제공되지 않았거나, 제공된 인증이 유효하지 않을 때 사용합니다.
- 403 Forbidden: 인증은 되었으나, 요청된 리소스에 대한 권한이 없을 때 사용합니다.
- 404 Not Found: 요청된 리소스를 찾을 수 없을 때 사용합니다.
- 500 Internal Server Error: 서버에서 예기치 않은 오류가 발생했을 때 사용합니다.

4. 데이터 형식

JSON(JavaScript Object Notation): RESTful API에서 가장 널리 사용되는 데이터 형식입니다. 클라이언트와 서버 간에 직관적이고 쉽게 읽을 수 있는 데이터를 주고받기 위해 JSON을 사용합니다.

RESTful API를 설계할 때 고려해야 할 주요 요소를 설명하세요.

5. 보안 고려

HTTPS 사용: 모든 API 요청은 HTTPS(SSL/TLS)를 통해 암호화된 통신 채널에서 이루어져야 합니다.
이를 통해 클라이언트와 서버 간의 데이터 전송 시 기밀성을 유지하고, 중간자 공격(MITM) 등을 방지할 수 있습니다.

- **인증 및 권한 관리:**
  . API Key: 간단한 인증 메커니즘으로, 요청 시 API 키를 함께 전송하여 클라이언트의 신원을 확인합니다.
  . OAuth 2.0: 더 강력한 인증 및 권한 부여 메커니즘으로, 접근 토큰을 사용하여 리소스에 대한 권한을 관리합니다.
  . JWT (JSON Web Token): 클라이언트가 인증되었음을 증명하는 토큰을 사용하여 API 요청 시 신원을 확인합니다.

- **입력 검증:** 모든 입력 데이터는 철저히 검증되어야 하며, SQL 인젝션, XSS 등과 같은 공격을 방지하기 위해 적절한 필터링과 인코딩이 필요합니다.

6. 성능 최적화

- 캐싱:
  . 자주 변경되지 않는 데이터(예: 공통 설정 값이나 코드 데이터)를 캐시하여 성능을 최적화할 수 있습니다.
  . HTTP 캐싱 헤더(Cache-Control, ETag 등)를 사용하여 클라이언트와 서버 간 캐시 전략을 설정할 수 있습니다.

- **페이징(Pagination):**
  . 대량의 데이터를 조회할 때는 한 번에 모든 데이터를 반환하지 않고, 데이터를 페이징하여 클라이언트로 전송합니다. 이를 통해 네트워크 트래픽을 줄이고, 서버 부하를 감소시킬 수 있습니다.

  예: GET /customers?page=1&size=20 (20개의 고객 데이터를 첫 페이지에서 조회)

- **배치 처리:** 여러 개의 요청을 한 번에 처리할 수 있도록 배치 처리 엔드포인트를 제공할 수 있습니다. 이는 네트워크 요청 횟수를 줄이고, 성능을 개선할 수 있습니다.

7. 버전 관리

- **API 버전 관리:** API는 시간이 지남에 따라 변경될 수 있으므로, 버전 관리를 통해 클라이언트가 사용할 API 버전을 명시할 수 있도록 해야 합니다.

- **URI에 버전 포함:** /v1/customers
- **헤더에 버전 포함:** Accept: application/vnd.companyname.v1+json

결론

RESTful API를 설계할 때는 리소스의 명확한 식별, 적절한 HTTP 메서드 사용, 상태 코드 관리, 데이터 형식의 표준화, 보안 고려, 성능 최적화, 그리고 버전 관리 등 다양한 요소를 신중하게 고려해야 합니다. 이러한 요소들을 잘 반영한 API는 사용하기 쉽고, 확장성과 유지보수성이 높으며, 보안과 성능 측면에서도 안정적인 애플리케이션을 구축할 수 있게 합니다.

API의 보안을 강화하기 위한 방법을 설명하고,
예를 들어 설명하세요.힌트**: OAuth2, JWT를 사용한 토큰 기반 인증, HTTPS를 통한 데이터 암호화, 입력 검증 및 데이터 인코딩을 통해 API 보안을 강화할 수 있습니다.**

API 보안을 강화하기 위해 다양한 방법을 사용할 수 있습니다.
여기서는 OAuth2와 JWT를 사용한 토큰 기반 인증, HTTPS를 통한 데이터 암호화, 그리고 입력 검증 및 데이터 인코딩에 대해 설명하겠습니다.

1. OAuth2를 사용한 토큰 기반 인증
OAuth2는 널리 사용되는 인증 및 권한 부여 프레임워크로, 클라이언트가 자원 소유자(사용자)를 대신해 보호된 리소스에 접근할 수 있도록 권한을 위임하는 방법입니다.
OAuth2는 주로 액세스 토큰을 발급하여 API에 대한 접근 권한을 부여합니다.

a) OAuth2의 주요 흐름:
- **클라이언트 인증:** 클라이언트 애플리케이션이 OAuth2 서버에 인증을 요청합니다. 클라이언트는 인증 정보를 서버에 제공하며, 사용자의 승인을 요청합니다.

- **사용자 승인:** 사용자가 클라이언트 애플리케이션에 대한 접근 권한을 승인합니다.

- **액세스 토큰 발급 :** OAuth2 서버는 클라이언트 애플리케이션에게 액세스 토큰을 발급합니다. 이 토큰은 클라이언트가 보호된 리소스에 접근할 때 사용됩니다.

- **API 요청 시 토큰 사용:** 클라이언트는 API 요청 시, HTTP 헤더에 발급받은 액세스 토큰을 포함하여 서버에 요청을 보냅니다. 서버는 이 토큰을 검증하여 클라이언트의 권한을 확인합니다.

- **OAuth2 사용 예시:**
. 액세스 토큰 발급: 클라이언트가 사용자 인증 후, OAuth2 서버로부터 액세스 토큰을 발급받습니다:

```
POST /oauth2/token
Host: authorization-server.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=AUTH_CODE&redirect_uri=REDIRECT_URI&client_id=CLIENT_ID&client_secret=CLIENT_SECRET
보호된 리소스 요청:
```

. 클라이언트가 액세스 토큰을 사용하여 보호된 리소스에 접근합니다:

```
GET /api/customers
Host: api.example.com
Authorization: Bearer ACCESS_TOKEN
```

2. JWT(JSON Web Token)를 사용한 토큰 기반 인증
JWT는 JSON 형식의 웹 토큰으로, 클라이언트가 인증된 상태임을 서버에 증명하기 위해 사용됩니다.
JWT는 자체적으로 서명되어 있어 변조 방지 기능을 제공하며, 클라이언트와 서버 간의 인증 정보를 안전하게 전달하는 데 유용합니다.

a) JWT의 주요 구성 요소:
. 헤더(Header): 토큰의 유형과 해싱 알고리즘 정보를 포함합니다.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

API의 보안을 강화하기 위한 방법을 설명하고,
예를 들어 설명하세요.힌트**: OAuth2, JWT를 사용한 토큰 기반 인증, HTTPS를 통한 데이터 암호화, 입력 검증 및 데이터 인코딩을 통해 API 보안을 강화할 수 있습니다.**

. 페이로드(Payload): 사용자 정보와 클레임(Claims)이 포함됩니다. 클레임은 토큰에 포함된 사용자 정보와 속성을 나타냅니다.

```
{
  "sub": "user_id",
  "name": "John Doe",
  "iat": 1516239022
}
```

. 서명(Signature): 헤더와 페이로드를 조합하고 비밀 키로 서명한 값으로, 토큰의 진위 여부를 검증합니다.

. JWT 사용 예시:

토큰 발급: 클라이언트가 로그인 성공 후, 서버로부터 JWT를 발급받습니다:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyX2lkIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"
}
```

보호된 리소스 요청: 클라이언트가 JWT를 사용하여 보호된 리소스에 접근합니다:

```
GET /api/customers
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyX2lkIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

3. HTTPS를 통한 데이터 암호화

HTTPS는 HTTP 프로토콜을 SSL/TLS 암호화 프로토콜과 결합한 것으로, 클라이언트와 서버 간의 모든 데이터 전송을 암호화하여 보호합니다.
이를 통해 전송 중 데이터가 도청이나 중간자 공격에 의해 노출되는 것을 방지합니다.

a) HTTPS 사용 예시:

- HTTPS 연결: 클라이언트가 HTTPS를 통해 서버에 연결하고 데이터를 주고받습니다:

```
GET https://api.example.com/customers
HTTPS는 클라이언트와 서버 간에 보안 연결을 설정하고, 모든 전송 데이터를 암호화합니다.
```

- SSL/TLS 인증서: 서버는 클라이언트에게 SSL/TLS 인증서를 제공하여, 클라이언트가 서버의 신원을 확인하고 보안 연결을 설정할 수 있습니다.

4. 입력 검증 및 데이터 인코딩

입력 검증과 데이터 인코딩은 클라이언트가 서버에 제공하는 데이터를 검사하고, 그 데이터가 안전하게 처리되도록 인코딩하는 과정을 포함합니다.
이를 통해 SQL 인젝션, XSS(교차 사이트 스크립팅) 등의 공격을 방지할 수 있습니다.

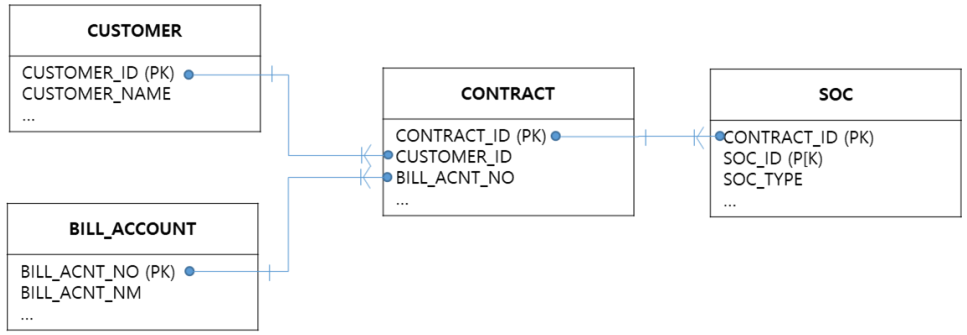
a) 입력 검증 예시:

- SQL 인젝션 방지: 사용자 입력을 SQL 쿼리에 직접 포함시키지 않고, 파라미터화된 쿼리 또는 준비된 문을 사용하여 입력 데이터를 처리합니다.

- XSS 방지: 클라이언트가 서버에 전송하는 데이터(예: HTML 코드)를 인코딩하여 브라우저에서 실행되지 않도록 합니다.

```
// 사용자 입력을 인코딩하여 XSS 방지
function sanitizeInput(input) {
    return input.replace(/&/g, '&amp;')
                .replace(/</g, '&lt;')
                .replace(/>/g, '&gt;')
                .replace(/"/g, '&quot;')
                .replace(/'/g, '&#x27;')
                .replace(/₩//g, '&#x2F;');}
```

서로다른 사용자가 동일 계약의 상품 정보를 동시에 변경하는 것을 방지하도록 방안을 제시하세요




```plaintext
+-----------------+       +--------------------+       +-----------------+
|   Contracts     |       |      Products      |       |     Users       |
+-----------------+       +--------------------+       +-----------------+
| contract_id PK  |<------+ product_id PK      |       | user_id PK      |
| contract_name   |       | contract_id FK     |       | username        |
| ...             |       | product_name       |       | ...             |
+-----------------+       | price              |       +-----------------+
                          | version            |
                          | last_modified      |
                          | locked_by FK       |
                          +--------------------+
                                   |
                                   |
                                   v
                          +---------------------+
                          |  PessimisticLock    |
                          +---------------------+
                          | product_id PK, FK   |
                          | locked_by FK        |
                          | locked_at TIMESTAMP |
                          +---------------------+
```

낙관적 잠금:
Products 테이블의 version 필드를 사용,
데이터 수정 시 다른 사용자가 먼저 수정하지 않았는지 확인합니다.
이 필드를 통해 동일 데이터에 대한 동시 수정 시도 시 충돌을 방지합니다.

비관적 잠금:
사용자가 특정 상품을 수정하려고 할 때 PessimisticLock 테이블에 잠금을 설정합니다.
이 잠금이 유지되는 동안 다른 사용자는 동일 상품을 수정할 수 없습니다.

locked_by 필드:
Products 테이블과 PessimisticLock 테이블에 locked_by 필드를 사용하여 어느 사용자가 현재 잠금을 가지고 있는지를 기록합니다.
이를 통해 동시에 다른 사용자가 수정할 수 없도록 제어합니다.

토큰을 사용하는

인증 : 세션
인가 : 토큰

Session(세션)과 Token(토큰)의 차이는?
https://velog.io/@ddangle/Session%EC%84%B8%EC%85%98%EA%B3%BC-Token%ED%86%A0%ED%81%B0%EC%9D%98-%EC%B0%A8%EC%9D%B4%EB%8A%94

OAuth 2.0과 OIDC(OpenID Connect) 프로토콜
https://velog.io/@gnlee95/oauth2-and-oidc
https://sabarada.tistory.com/248
https://sabarada.tistory.com/264

Spring Security 구조, 흐름 그리고 역할 알아보기
https://velog.io/@hope0206/Spring-Security-%EA%B5%AC%EC%A1%B0-%ED%9D%90%EB%A6%84-%EA%B7%B8%EB%A6%AC%EA%B3%A0-%EC%97%AD%ED%95%A0-%EC%95%8C%EC%95%84%EB%B3%B4%EA%B8%B0

동시성 부하분산
https://velog.io/@mw310/%EC%8B%9D%EA%B5%AC%ED%95%98%EC%9E%90MSA-%EC%84%A0%EC%B0%A9%EC%88%9C-%EC%8B%9C%EC%8A%A4%ED%85%9C-%EC%BF%A0%ED%8F%B0-%EC%84%9C%EB%B9%84%EC%8A%A4-%EA%B5%AC%ED%98%84%ED%95%B4%EB%B3%B4%EC%9E%90

https://velog.io/@akfls221/%EB%8F%99%EC%8B%9C%EC%84%B1%EC%97%90-%EB%8C%80%ED%95%9C-%ED%95%B4%EA%B2%B0%EB%B0%A9%EB%B2%95%EC%9D%84-%EC%95%8C%EC%95%84%EB%B3%B4%EC%9E%90

DB 트랜젝션
https://catsbi.oopy.io/78b397fd-cead-401c-aa71-7bdc30b867d4

마이크로서비스 인증/인가 패턴
https://engineering-skcc.github.io/microservice%20outer%20achitecture/outer-arch-Auth/

MSA 환경에서 장애 전파를 막기 위한 서킷 브레이커 패턴
https://hudi.blog/circuit-breaker-pattern/

MSA 분산 트랜잭션 관리
https://baebalja.tistory.com/622

웹소켓

무료 ERD
https://dbdiagram.io/d