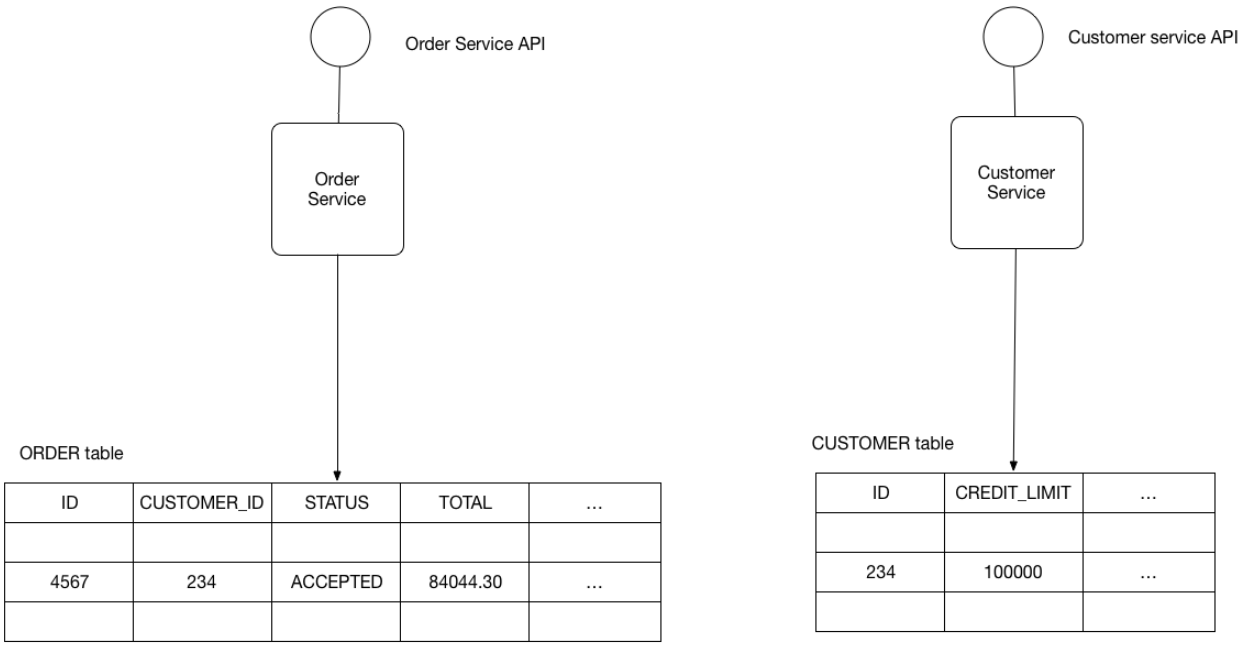
Pattern: Database per service

version 2019.30

상황

온라인 쇼핑몰을 마이크로서비스 아키텍처를 적용하여 개발한다고 생각해봅시다. 대부분의 서비스는 특정 데이터베이스에 데이터를 유지해야합니다. 예를 들어, `Order Service`는 주문에 대한 정보를 저장하고 `Customer Service`는 고객에 대한 정보를 저장합니다.



문제

마이크로서비스 아키텍처에서는 어떤 데이터베이스 아키텍처를 사용할 것인가?

Forces

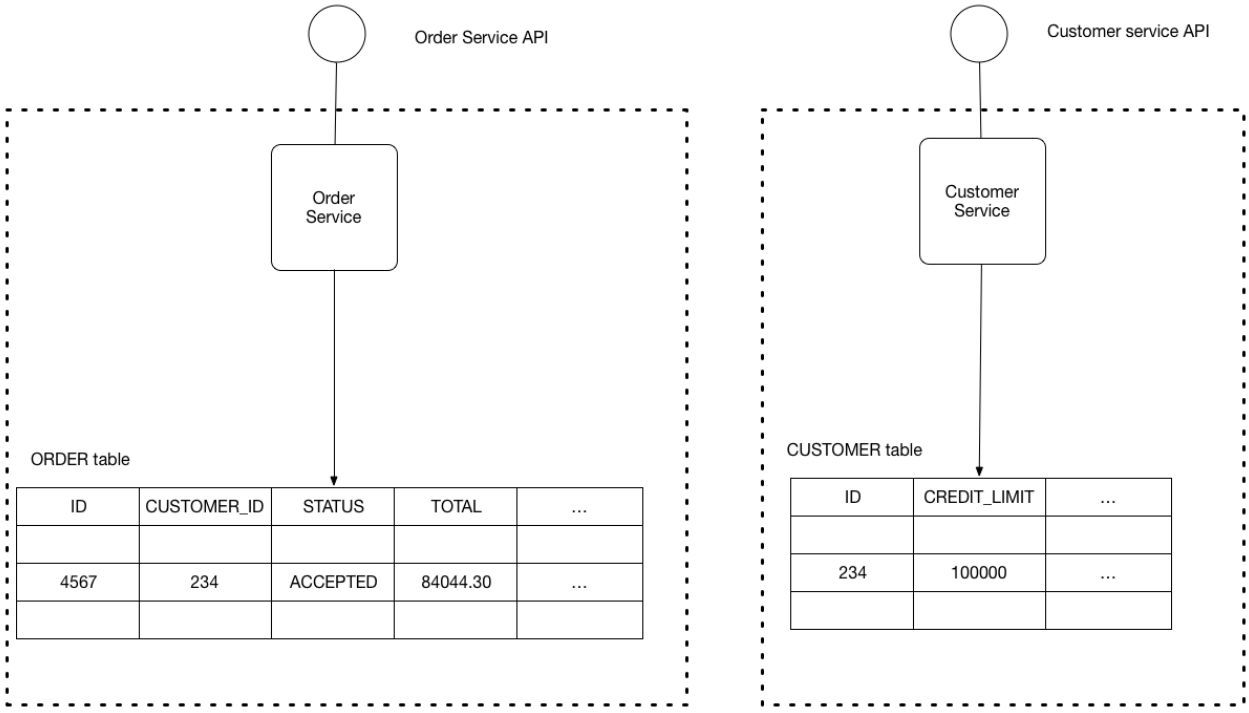
- 서비스는 느슨하게 결합되어 독립적으로 개발, 배포 및 확장될 수 있어야 합니다.
- 일부 비즈니스 트랜잭션은 여러 서비스에 걸쳐있는 invariant를 강제해야 합니다. 예를 들어, `Place Order` 유스 케이스는 새로운 주문이 고객의 신용 한도를 초과하지 않는지 확인해야 합니다. 다른 비즈니스 트랜잭션은 여러 서비스가 소유한 데이터를 업데이트해야 합니다.
- 일부 비즈니스 트랜잭션은 여러 서비스가 소유하는 데이터를 조회해야합니다. 예를 들어, `View Available Credit`의 유스 케이스는 `creditLimit`을 확인하기 위해 Customer를 조회 해야하며 총 주문 금액을 계산하기 위해 Order를 조회해야합니다.

- 일부 쿼리는 여러 서비스가 소유하고 있는 데이터를 조인해야합니다. 예를 들어, 몇몇 지역의 고객과 그들의 최근 주문내역을 확인하기 위해서는 Customer와 Order를 조인해야합니다.
- 데이터베이스는 때때로 확장을 위하여 복제되고 공유되어야합니다. Scale Cube를 참고하세요.
- 서비스들은 서로 다른 데이터 저장 요구 사항을 가집니다. 몇몇 서비스에서는 관계형 데이터베이스가 최고의 선택입니다. 다른 서비스들에서는 MongoDB와 같은 NoSQL 데이터베이스가 최선이 될 수 도 있습니다. 이는 복잡하고 구조화 되지 않은 데이터 또는 Neo4J를 저장하기에 적합합니다. 또한 그래프 데이터를 조회하고 저장하는데에 효율적입니다.

해결책

각 마이크로서비스의 영구 데이터를 해당 서비스에 대해 프라이빗으로 유지하고 API를 통해서만 액세스할 수 있게 하십시오. 각 서비스의 트랜잭션은 소유하고 있는 데이터베이스에만 관여됩니다.

아래의 다이어그램은 이 패턴의 구조를 보여줍니다.



서비스의 데이터베이스는 사실상 해당 서비스 구현의 일부입니다. 데이터베이스는 다른 서비스에서 직접 액세스할 수 없습니다.

서비스의 영구 데이터를 프라이빗으로 유지하는 몇 가지 방법이 있습니다. 각 서비스마다 데이터베이스 서버를 프로비저닝 할 필요가 없습니다. 예를 들어, 관계형 데이터베이스를 사용하고 있다면 다음과 같은 옵션을 선택할 수 있습니다.

- Private-tables-per-service – 각 서비스는 해당 서비스만 액세스할 수 있는 테이블의 집합을 가집니다.
- Schema-per-service – 각 서비스는 서비스 별로 프라이빗한 데이터베이스 스키마를 가집니다.
- Database-server-per-service – 각 서비스 별로 데이터베이스 서버를 가집니다.

private-tables-per-service와 schema-per-service는 낮은 오버헤드를 가집니다. 서비스 하나 당 하나의 스키마를 가지는 것은 소유권을 명확하게 보여줍니다. 높은 처리량을 요구하는 서비스의 경우 데이터베이스 서버를 소유해야할지도 모릅니다.

모듈성을 강제하는 장벽을 만드는 것은 좋은 아이디어입니다. 예를 들어, 각 서비스 별로 서로 다른 데이터베이스 유저 아이디를 할당거나 grant와 같은 액세스 제어 메커니즘을 사용할 수 있습니다. 캡슐화를 강제하는 장벽이 없다면, 개발자는 서비스의 API를 우회하여 직접 데이터에 액세스하려는 유혹에 빠지게됩니다.

예제

FTGO application은 이러한 접근법을 사용하는 애플리케이션의 예제입니다. 각 서비스는 공유된 MySQL 서버에서 논리적으로 소유하고 있는 데이터베이스에 대한 접근 권한을 가진 인증 자격을 가집니다. 자세한 정보는 blog post를 참고하십시오.

예상되는 상황

서비스 별로 데이터베이스를 사용하면 다음과 같은 이점이 있습니다.

- 서비스가 느슨하게 결합되도록 합니다. 한 서비스의 데이터베이스를 변경해도 다른 서비스에는 영향을 미치지 않습니다.
- 각 서비스는 요구사항에 적합한 데이터베이스의 종류를 선택할 수 있습니다. 예를 들어, 텍스트 검색을하는 서비스는 ElasticSearch를 사용할 수 있습니다. 소셜 그래프를 조작하는 서비스는 Neo4j를 사용할 수 있습니다.

서비스 별로 데이터베이스를 사용하면 다음과 같은 결점이 있습니다.

- 여러 서비스에 걸쳐있는 비즈니스 트랜잭션을 구현하는 것은 쉽지 않습니다. Distributed transactions are best avoided because of the CAP theorem. 그러나 많은 모던 데이터베이스(NoSQL)을 이를 지원하지 않습니다.
- 여러 데이터베이스에있는 데이터를 조인하는 쿼리를 구현하는 것은 쉽지 않습니다.
- 여러 SQL 및 NoSQL 데이터베이스를 관리하는 것은 복잡합니다.

다음과 같이 여러 서비스에 걸쳐 있는 트랜잭션과 쿼리를 구현하는 다양한 패턴/솔루션이 있습니다.

- Implementing transactions that span services - Saga pattern을 사용하십시오.
- Implementing queries that span services:
    - API Composition - 데이터베이스가 아닌 애플리케이션에서 조인을 수행합니다. 예를 들어, 하나의 서비스(또는 API 게이트웨이)는 고객의 최근 주문 내역을 얻기 위해 먼저 customer 서비스로 부터 고객에 대한 정보를 검색하고 order 서비스에 쿼리 함으로써 고객과 주문에 대한 정보를 검색할 수 있습니다.
    - Command Query Responsibility Segregation (CQRS) - 여러 서비스의 데이터를 포함하는 하나 또는 그 이상의 materialized views를 유지하십시오. 하나의 뷰는 서비스에 의해 관리되며 서비스는 각 서비스에서 데이터를 수정함에 따라 발행되는 이벤트를 구독하므로써 뷰를 관리합니다. 예를 들어, 온라인 쇼핑몰은 특정 지역에 거주하면서 최근에 주문한 고객을 검색하는 쿼리를 고객과 주문을 조인하는 뷰를 유지함으로써 구현할 수 있습니다. 서비스가 고객과 주문에 관련된 이벤트를 구독하고 이를 통해 뷰를 수정합니다.

관련 패턴

- Microservice architecture pattern는 이 패턴을 필요하게 만듭니다.

- Saga pattern은 일관된 트랜잭션을 구현하는데 유용한 방법입니다.
- API Composition와 Command Query Responsibility Segregation (CQRS)패턴은 쿼리를 구현하는데 유용한 방법입니다.
- Shared Database anti-pattern은 마이크로서비스가 데이터베이스를 공유함으로써 생기는 문제를 설명합니다.

> **참고 문헌**
>
> 본 문서는 Microservices.io에서 작성한 A pattern language for microservices를 번역한 문서입니다.
>
> **Reference**
>
> This document is a translation of A pattern language for microservices written by Microservices.io.