제14장

서비스 기반 아키텍처 스타일

서비스 기반 아키텍처는 마이크로서비스 아키텍처 스타일의 변형으로, 특히 유연성 덕분에 가장 실용적인 스타일 중 하나로 여겨집니다. 서비스 기반 아키텍처는 분산형 아키텍처이지만, 마이크로서비스나 이벤트 기반 아키텍처와 같은 다른 분산형 아키텍처만큼 복잡하거나 비용이 많이 들지 않아 비즈니스 관련 애플리케이션에 널리 사용됩니다.

위상수학

서비스 기반 아키텍처의 기본 토폴로지는 별도로 배포된 사용자 인터페이스, 별도로 배포된 원격의 세분화되지 않은 서비스, 그리고 선택적으로 단일체 데이터베이스로 구성된 분산형 매크로 계층 구조를 따릅니다.

그림 14-1은 이 스타일의 기본 토폴로지를 보여 주지만 , 별도의 UI와 별도의 데이터베이스를 사용하는 등 다양한 변형이 가능합니다. 이 장에서는 이러한 토폴로지 변형에 대해 자세히 설명합니다.

이러한 아키텍처 스타일에서 서비스는 일반적으로 시스템 내의 특정 도메인 또는 하위 도메인을 나타내므로 도메인 서비스라고 합니다. 도메인 서비스는 일반적으로 세분화되지 않고 시스템 기능의 일부(예: 주문 처리 또는 배송)를 담당합니다. 일반적으로 서로 독립적이며 개별적으로 배포됩니다. 서비스는 일반적으로 모놀리식 애플리케이션과 유사하게 배포되므로 컨테이너화가 필수적이지는 않습니다(물론 Docker 또는 Kubernetes와 같은 컨테이너에 도메인 서비스를 배포하는 것도 하나의 옵션입니다).

단일 모놀리식 데이터베이스를 사용하는 경우, 아키텍트는 변경 관리, 확장성 및 내결함성 문제를 방지하기 위해 도메인 서비스 수를 최소화해야 합니다(12개 이하를 권장합니다).

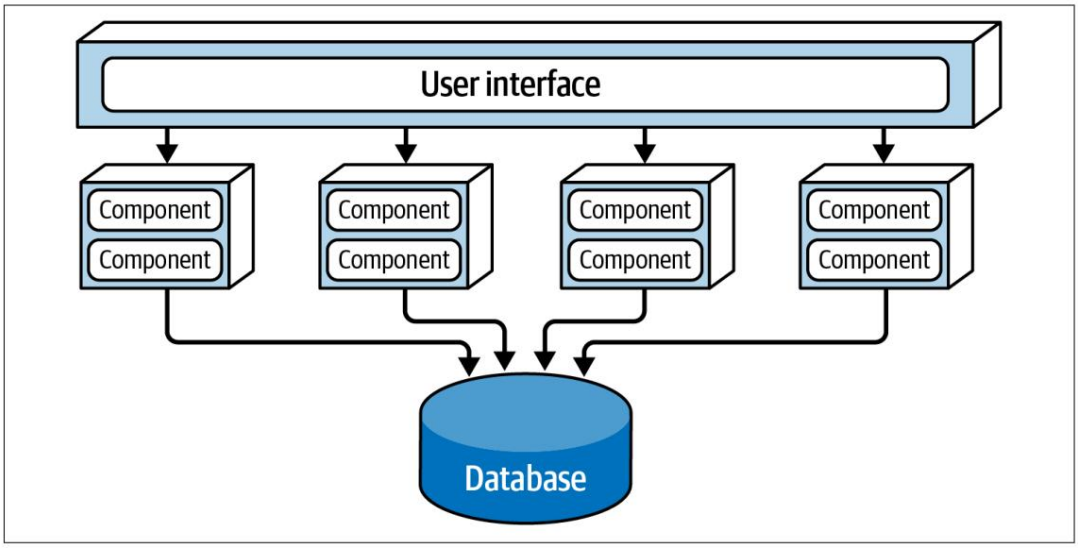


그림 14-1. 서비스 기반 아키텍처 스타일의 기본 토폴로지

일반적으로 각 도메인 서비스는 단일 인스턴스로 배포됩니다. 그러나 시스템의 확장성, 내결함성 및 처리량 요구 사항에 따라 아키텍트는 도메인 서비스의 인스턴스를 여러 개 생성하는 경우가 있습니다. 이 경우 UI가 정상 작동 중이고 사용 가능한 서비스 인스턴스로 연결될 수 있도록 UI와 도메인 서비스 간의 로드 밸런싱 기능이 필요합니다.

서비스는 일반적으로 REST와 같은 원격 액세스 프로토콜을 사용하여 UI에서 원격으로 액세스됩니다. 다른 방법으로는 메시징, 원격 프로시저 호출(RPC), 프록시 또는 게이트웨이가 있는 API 계층, 심지어 SOAP도 있습니다. 그러나 대부분의 경우 UI에는 서비스에 직접 액세스할 수 있도록 서비스 로케이터 패턴 이 내장되어 있으며 , 이 서비스 로케이터 패턴은 API 게이트웨이 또는 프록시 내에 내장될 수도 있습니다.

서비스 기반 아키텍처는 일반적으로 중앙에서 공유되는 단일체 데이터베이스를 사용합니다. 이를 통해 서비스는 기존의 단일 계층형 아키텍처처럼 SQL 쿼리와 조인을 활용할 수 있습니다. 이러한 아키텍처 스타일에서는 일반적으로 도메인 서비스의 수가 적기 때문에 사용 가능한 데이터베이스 연결이 모두 소진되는 문제는 거의 발생하지 않습니다. 하지만 데이터베이스 변경 관리에는 어려움이 있을 수 있습니다. 215페이지의 "데이터 토폴로지"에서는 서비스 기반 아키텍처에서 데이터베이스 변경 사항을 처리하고 관리하는 기법을 설명합니다.

스타일 사양

서비스 기반 아키텍처에서 도메인 서비스는 일반적으로 세분화되지 않은(coarse-grained) 구조를 가지므로, 각 도메인 서비스는 일반적으로 API 파사드 계층, 비즈니스 계층, 영속성 계층으로 구성된 계층형 아키텍처 스타일을 사용하여 설계됩니다. 또 다른 일반적인 설계 방식은 모듈형 모놀리식 아키텍처 스타일(11 장 참조)과 유사하게 각 도메인 서비스를 하위 도메인으로 분할하는 것입니다. 두 가지 접근 방식 모두 그림 14-2 에 나와 있습니다 .

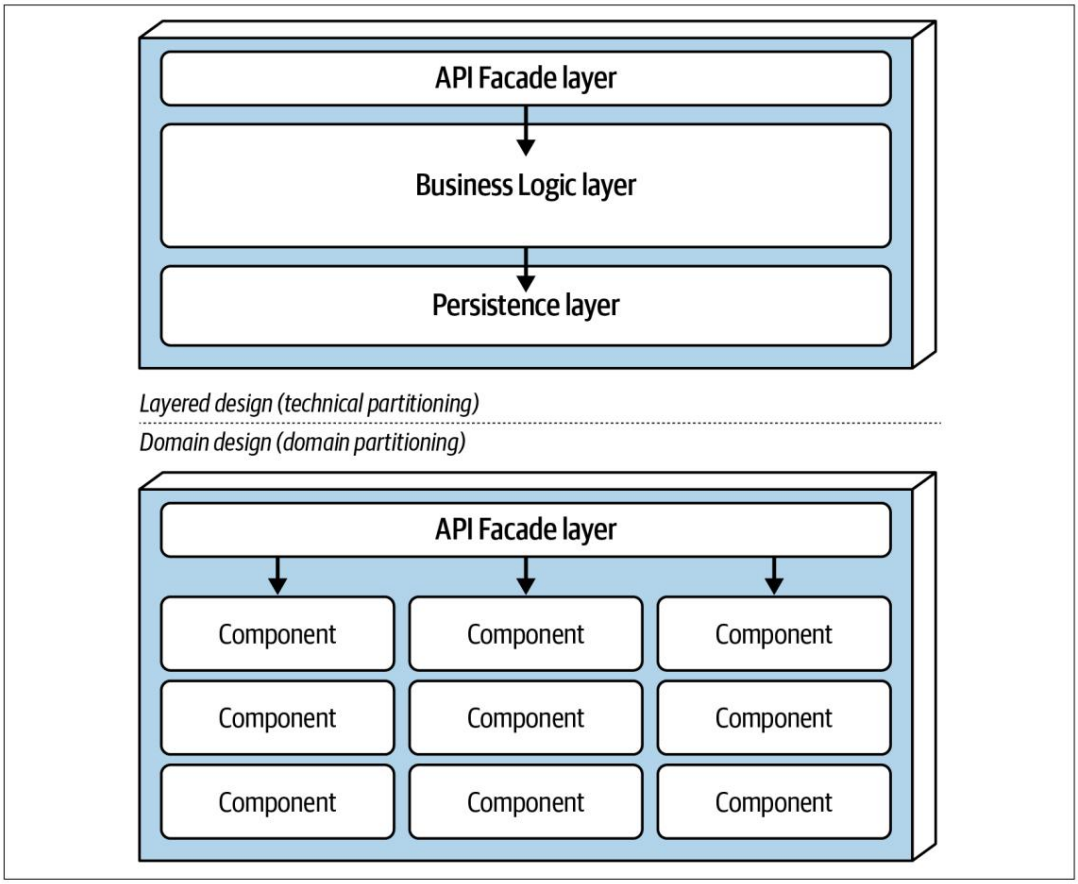


그림 14-2. 도메인 서비스 설계 변형

설계 방식과 관계없이 도메인 서비스는 UI가 상호 작용하여 비즈니스 기능을 실행할 수 있는 일종의 API 액세스 파사드를 포함해야 합니다. 이 API 액세스 파사드는 일반적으로 UI에서 들어오는 비즈니스 요청을 처리하는 역할을 담당합니다.

서비스 기반 아키텍처를 사용하는 전자상거래 사이트의 예를 살펴보겠습니다.
UI에서 비즈니스 요청이 들어옵니다. 고객이 상품을 주문하는 요청입니다. 이 단일 요청은 OrderService 도메인 서비스 내의 API 액세스 인터페이스에서 수신됩니다.
API 접근 인터페이스는 주문 접수, 주문 ID 생성, 결제 처리, 주문 상품 재고 업데이트 등 요청을 이행하는 데 필요한 모든 과정을 내부적으로 조율합니다. 마이크로서비스 아키텍처에서는 이러한 요청을 완료하려면 여러 개의 개별적으로 배포된 원격 서비스를 조율해야 할 가능성이 높습니다. 서비스 기반 아키텍처의 내부 클래스 수준 조율과 마이크로서비스의 외부 서비스 조율 간의 이러한 세분성 차이는 두 아키텍처 방식의 중요한 차이점 중 하나입니다.

도메인 서비스는 세분화되지 않은(coarse-grained) 구조이기 때문에, 표준적인 데이터베이스 커밋 및 롤백을 사용하는 일반적인 ACID(원자성, 일관성, 격리성, 내구성) 데이터베이스 트랜잭션을 통해 단일 도메인 서비스 내에서 데이터베이스 무결성을 보장할 수 있습니다. 이와는 대조적으로, 마이크로서비스와 같은 고도로 분산된 아키텍처의 세분화된 단일 목적 서비스는 BASE 트랜잭션(기본 가용성, 소프트 상태, 최종 일관성)이라는 분산 트랜잭션 기법을 사용합니다. 이러한 세분화된 서비스는 서비스 기반 아키텍처의 ACID 트랜잭션이 제공하는 수준의 데이터베이스 무결성을 지원하지 못합니다.

이 점을 설명하기 위해, 예시로 든 서비스 기반 전자상거래 사이트의 주문 결제 과정을 생각해 보겠습니다. 고객이 주문을 했는데, 결제에 사용한 신용카드가 만료되었다고 가정해 봅시다. 이 결제는 동일한 서비스 내에서 원자적으로 처리되는 트랜잭션이므로, 서비스는 표준 트랜잭션 롤백을 사용하여 데이터베이스에 추가된 모든 정보를 삭제할 수 있습니다. 그런 다음 서비스는 고객에게 결제가 처리될 수 없음을 알립니다.

이제 더 작고 세분화된 서비스들로 구성된 마이크로서비스 아키텍처에서 동일한 프로세스를 생각해 보겠습니다. 먼저, 주문 처리 서비스(OrderPlacement)는 요청을 수락하고, 주문을 생성하고, 주문 ID를 생성한 후, 주문 정보를 주문 테이블에 삽입합니다. 이 작업이 완료되면 주문 처리 서비스는 결제 서비스(PaymentService) 를 원격으로 호출하여 결제를 시도합니다. 신용카드가 만료되어 결제가 불가능한 경우, 주문은 처리될 수 없습니다. 이제 데이터는 일관성이 없는 상태가 됩니다. 주문 정보는 이미 삽입되었지만 승인되지 않은 상태입니다. 이러한 데이터를 일관된 상태로 되돌리려면 주문 처리 서비스에 별도의 작업인 보상 업데이트(9 장 에서 설명)를 적용해야 합니다 .

서비스 설계 및 세분성 도메인 서비스는 세분화가

덜 되어 데이터 무결성과 일관성을 유지할 수 있지만, 큰 단점이 있습니다. 서비스 기반 아키텍처에서 OrderService의 주문 처리 기능을 변경하면 결제 처리 기능을 포함한 서비스의 모든 기능을 테스트하고 재배포해야 합니다. 반면 마이크로서비스 아키텍처에서는 동일한 변경 사항이 더 작고 세분화된 OrderPlacement 서비스에만 영향을 미치므로 PaymentService에 대한 테스트나 배포가 필요하지 않습니다 . 또한 도메인 서비스는 더 많은 기능을 배포하기 때문에 다른 부분(결제 처리 포함)에서 오류가 발생할 위험이 더 큽니다.

마이크로서비스 아키텍처에서는 각 서비스가 단일 책임만을 가지므로, 변경 시 다른 기능이 손상될 가능성이 줄어듭니다.

사용자 인터페이스 옵션 서비스 기

반 아키텍처 스타일은 다양한 UI 변형을 제공하여 매우 유연합니다. 예를 들어, 아키텍트는 그림 14-1 에 나타난 단일 모놀리식 UI를 각 도메인 서비스에 맞춰 여러 개의 개별 UI로 분리할 수도 있습니다.

이렇게 하면 시스템의 전반적인 확장성, 내결함성 및 민첩성이 향상됩니다. 이러한 UI 변형은 그림 14-3에 나와 있습니다.

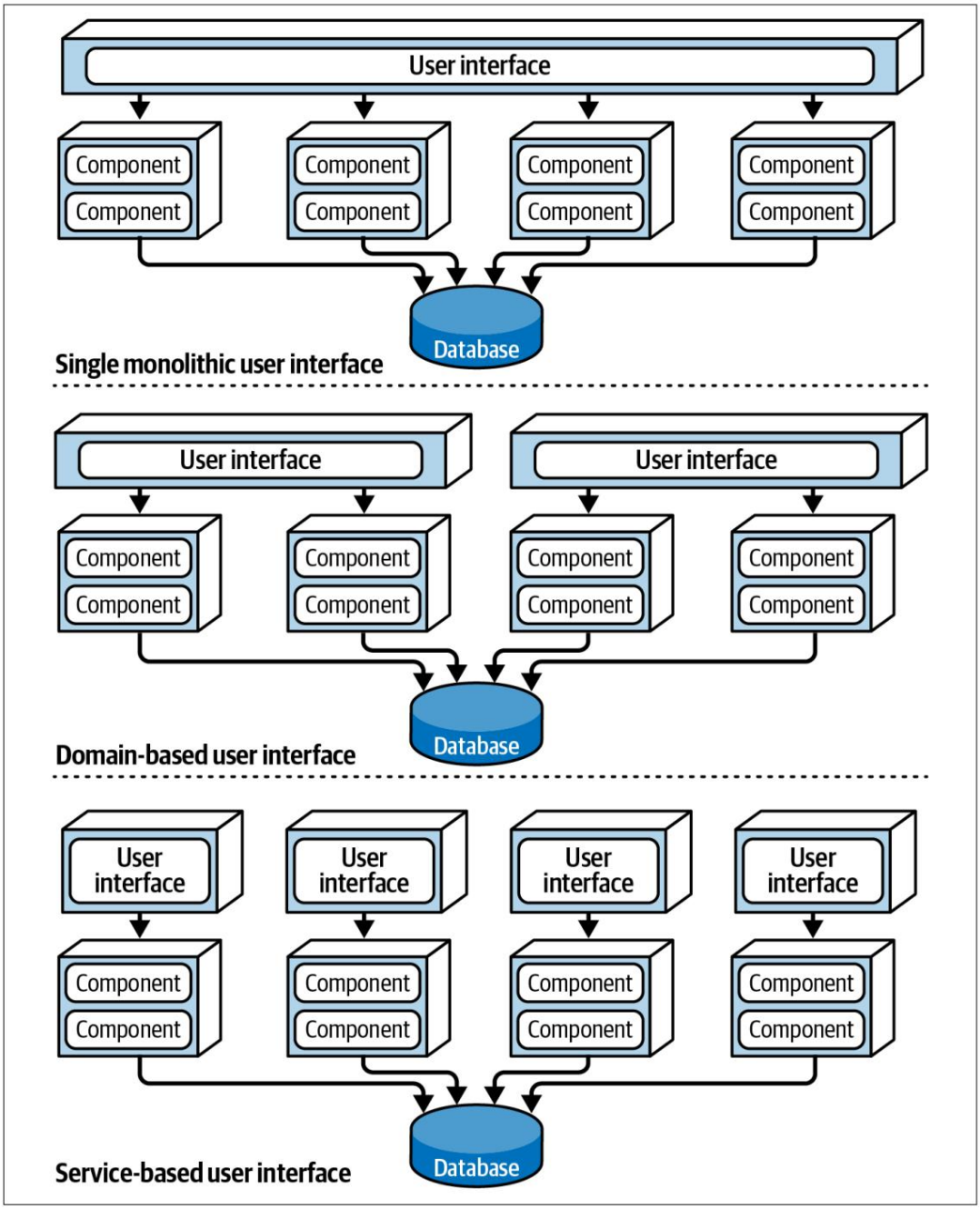


그림 14-3. 서비스 기반 아키텍처에서 세 가지 사용자 인터페이스 변형

예를 들어, 일반적인 주문 시스템은 고객이 주문을 할 수 있는 사용자 인터페이스(UI)와, 주문 포장 담당자가 포장해야 할 품목을 확인할 수 있는 내부 UI, 그리고 고객 지원을 위한 별도의 내부 UI를 가질 수 있습니다.

API 게이트웨이 옵션 이 아키

텍처 스타일의 유연성 덕분에 그림 14-4에서처럼 UI와 서비스 사이에 리버스 프록시 또는 API 게이트웨이로 구성된 API 계층을 추가할 수 있습니다. 이는 도메인 서비스 기능을 외부 시스템에 노출하고, 공통적인 공통 관심사(예: 메트릭, 보안, 감사 요구 사항 및 서비스 검색)를 통합하여 API 게이트웨이 방식으로 이동시키고, 여러 인스턴스를 가진 도메인 서비스의 로드 밸런싱을 수행하는 데 유용합니다.

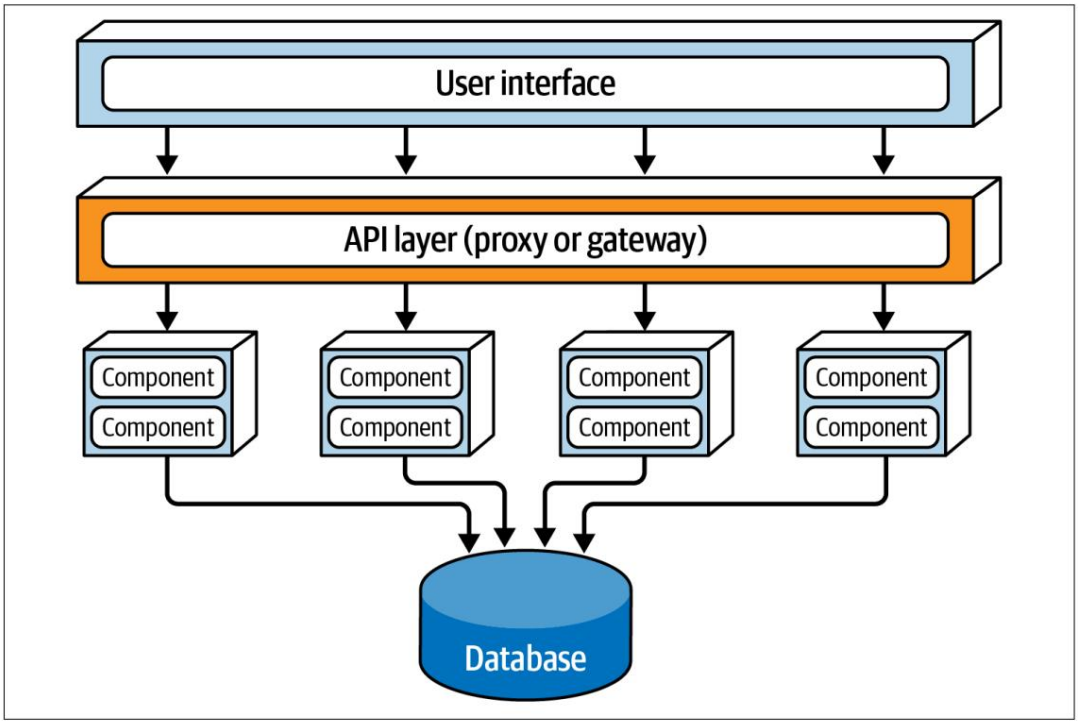


그림 14-4. UI와 도메인 서비스 사이에 API 계층 추가

데이터 토폴로지

서비스 기반 아키텍처는 아키텍트에게 다양한 데이터베이스 토폴로지 선택권을 제공하여 유연성을 더욱 높여줍니다. 이러한 아키텍처 방식은 단일 데이터베이스를 효과적으로 지원할 수 있는 분산 아키텍처라는 점에서 독특합니다. 하지만 단일 데이터베이스를 여러 개의 개별 데이터베이스로 분할할 수도 있으며, 나아가 각 도메인 서비스에 맞는 도메인 범위 데이터베이스를 생성하는 것(마이크로서비스와 유사)까지 가능합니다.

여러 개의 개별 데이터베이스를 사용하는 경우, 아키텍트는 다른 도메인 서비스가 각 데이터베이스의 데이터를 필요로 하지 않도록 해야 합니다. 그렇지 않으면 도메인 서비스 간의 서비스 간 통신이 발생할 수 있습니다. 이러한 아키텍처 방식에서는 다른 도메인 서비스를 호출하는 것보다 데이터를 공유하는 것이 일반적으로 더 바람직합니다. 이러한 데이터베이스 변형은 그림 14-5에 나와 있습니다.

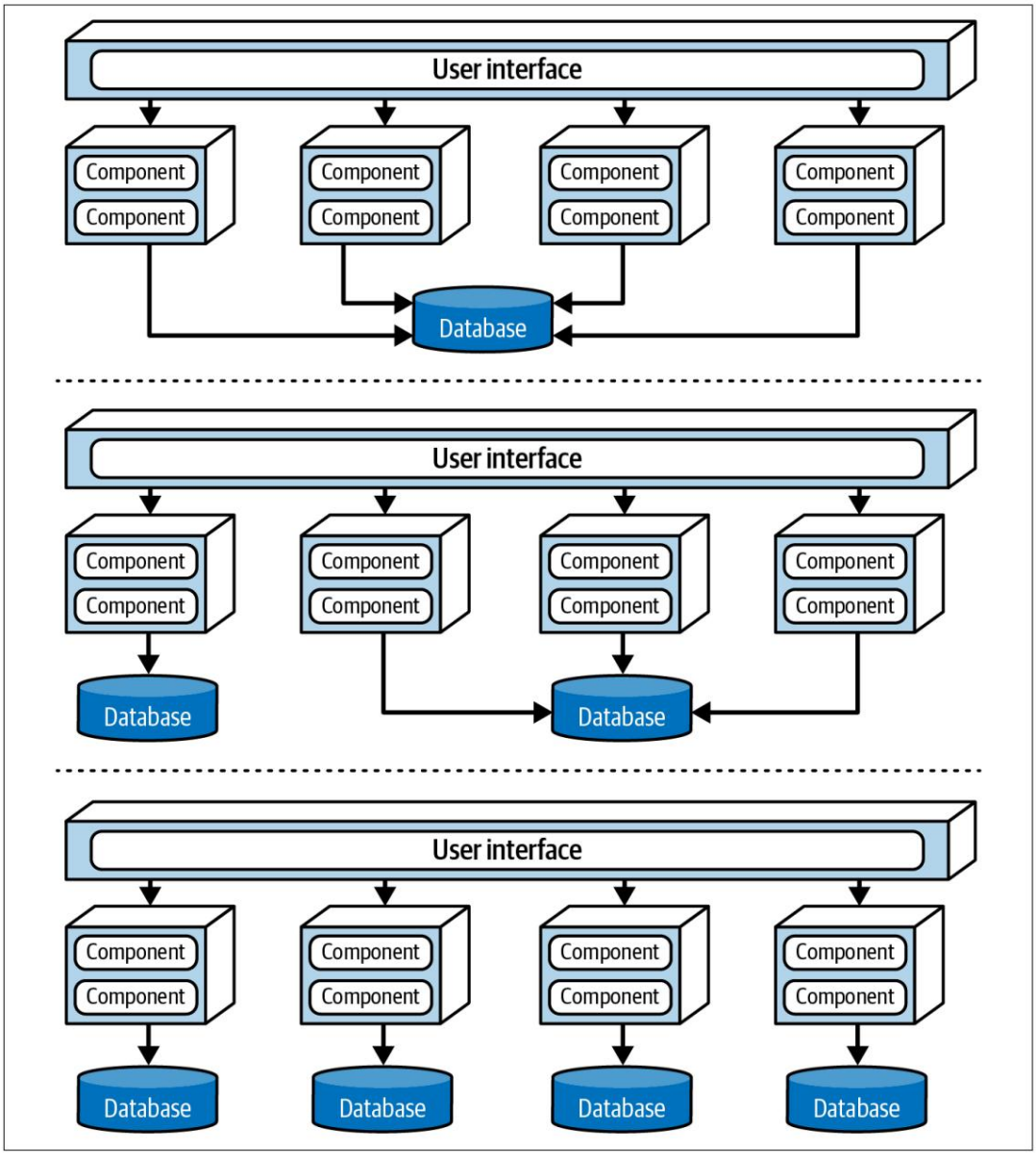


그림 14-5. 서비스 기반 아키텍처 데이터베이스 토폴로지

이러한 아키텍처 스타일은 모놀리식 데이터베이스를 지원하지만, 데이터베이스 테이블의 스키마 변경이 제대로 이루어지지 않으면 모든 도메인 서비스에 영향을 미칠 수 있습니다. 따라서 데이터베이스 변경은 상당한 비용과 위험을 수반하는 작업이며, 많은 노력과 조정, 그리고 전반적인 안정성이 요구됩니다.

서비스 기반 아키텍처에서 데이터베이스 테이블 스키마(엔티티 객체라고 함)를 나타내는 공유 클래스 파일은 일반적으로 모든 도메인 서비스에서 사용하는 JAR 또는 DLL 파일과 같은 사용자 지정 공유 라이브러리에 있습니다. 이러한 공유 라이브러리에는 SQL 코드도 포함될 수 있습니다. 모든 엔티티 객체를 위한 단일 공유 라이브러리를 생성하는 것은 다음과 같습니다.

서비스 기반 아키텍처를 구현하는 가장 비효율적인 방법입니다. 데이터베이스 테이블 구조를 변경하면 해당 엔티티 객체의 라이브러리도 변경해야 하므로, 실제로 변경된 테이블에 접근하는지 여부와 관계없이 모든 서비스를 변경하고 재배포해야 합니다. 공유 라이브러리 버전 관리를 통해 이 문제를 해결할 수 있지만, 상세한 수동 분석 없이는 테이블 변경이 어떤 서비스에 영향을 미칠지 파악하기 어렵습니다. 서비스 기반 아키텍처에서 안티패턴으로 간주되는 이 시나리오는 그림 14-6에 나타나 있습니다.

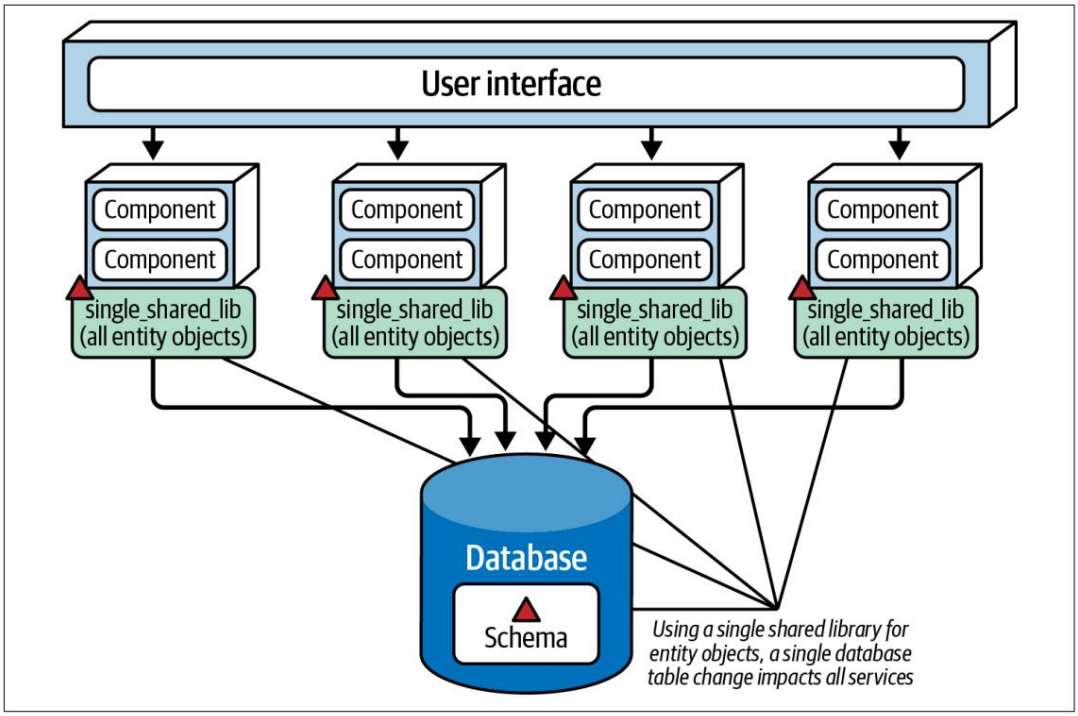


그림 14-6. 데이터베이스 엔티티 객체에 단일 공유 라이브러리를 사용하는 것은 변경 사항이 발생할 때 모든 서비스에 영향을 미치므로 서비스 기반 아키텍처에서 안티패턴으로 간주됩니다.

데이터베이스 변경의 영향과 위험을 완화하는 한 가지 방법은 데이터베이스를 논리적으로 분할하고 해당 논리적 분할을 별도의 공유 라이브러리를 통해 표현하는 것입니다.
그림 14-7 에서 데이터베이스는 공통, 고객, 송장 발행, 주문 및 추적의 다섯 가지 도메인으로 논리적으로 분할되어 있음을 알 수 있습니다. 각 도메인 서비스는 데이터베이스의 논리적 분할과 일치하는 다섯 개의 공유 라이브러리를 사용합니다.
이 기법을 사용하면 아키텍트가 특정 논리 도메인(이 경우 송장 발행) 내의 테이블을 변경할 때마다 해당 엔티티 객체(및 경우에 따라 SQL)가 포함된 공유 라이브러리도 자동으로 수정됩니다. 영향을 받는 서비스는 해당 공유 라이브러리를 사용하는 서비스뿐입니다. 다른 서비스는 이 변경 사항의 영향을 받지 않으므로 재테스트 및 재배포할 필요가 없습니다.

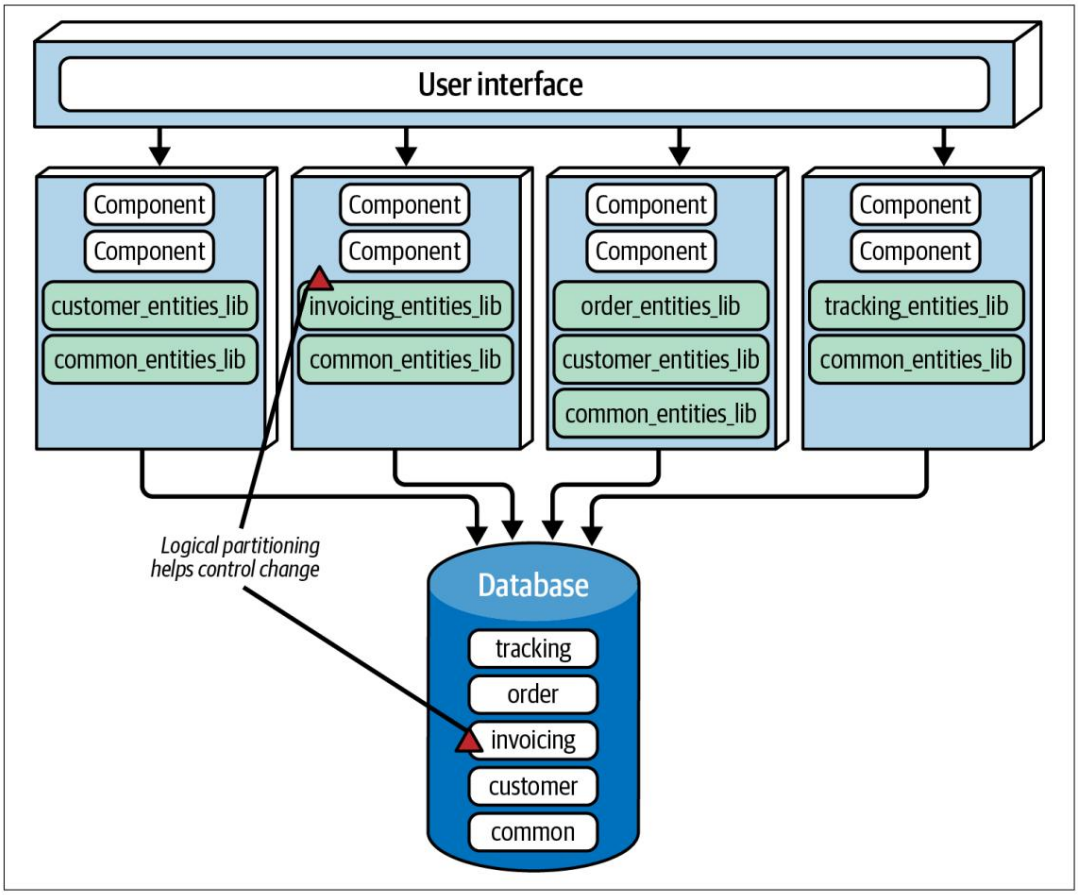


그림 14-7. 데이터베이스 엔티티 객체에 여러 공유 라이브러리 사용

그림 14-7 에서 데이터베이스는 모든 서비스에서 사용되는 공통 도메인과 해당 common_entities_lib 공유 라이브러리를 포함합니다. 이는 비교적 공통적인 경우입니다. 이러한 테이블은 모든 서비스에서 공통으로 사용되므로, 변경하려면 공유 데이터베이스에 접근하는 모든 서비스를 조율해야 합니다. 이러한 테이블(및 해당 엔티티 객체와 도메인 서비스) 변경으로 인해 발생할 수 있는 파급 효과를 완화하는 한 가지 방법은 버전 관리 시스템(사용 가능한 경우)에서 공통 엔티티 객체를 잠그고 데이터베이스 팀만 변경할 수 있도록 하는 것입니다.

이는 변경 사항을 제어하는 데 도움이 되며 모든 서비스에서 사용하는 공통 테이블을 변경하는 것의 중요성을 강조합니다.

서비스 기반 아키텍처에서 데이터베이스 변경 사항을 더 효과적으로 제어하려면, 명확하게 정의된 데이터 도메인을 유지하면서 데이터베이스의 논리적 파티셔닝을 최대한 세분화해야 합니다.

클라우드 고려 사항

분산 아키텍처인 서비스 기반 스타일은 클라우드 환경에서 매우 효과적입니다. 도메인 서비스는 일반적으로 세분화되지 않은 경우가 많지만, 범위가 넓기 때문에 서버리스 함수보다는 컨테이너화된 서비스로 구현되는 경우가 많으며, 클라우드 파일 저장소, 데이터베이스, 메시징 서비스를 쉽게 활용할 수 있습니다.

일반적인 위험

마이크로서비스에서는 서비스 간 통신이 일반적이지만, 서비스 기반 아키텍처 스타일에서는 아키텍트들이 이를 피하려고 합니다. 이상적으로는 도메인들이 최대한 독립적이어야 하며, 결합은 데이터베이스 수준에서만 이루어져야 합니다. 도메인 서비스 간 과도한 통신은 아키텍트가 도메인을 제대로 분할하지 않았거나, 해결하려는 문제에 적합한 아키텍처 스타일이 아니라는 것을 나타내는 좋은 지표입니다.

또 다른 일반적인 위험은 도메인 서비스를 너무 많이 생성하는 것입니다. 실질적인 상한선은 약 12개입니다. 그 이상이 되면 테스트, 배포, 모니터링, 데이터베이스 연결 및 변경에 문제가 발생할 가능성이 높습니다.

통치

이 책 전반에 걸쳐 논의된 순환 복잡도, 확장성, 응답성 등과 같은 일반적인 구조적 및 운영적 아키텍처 거버넌스 기술 외에도, 아키텍트는 서비스 기반 아키텍처의 구조적 무결성을 보장하기 위해 특정 거버넌스 테스트를 적용할 수 있습니다.

도메인 서비스는 최대한 독립적이어야 하므로, 이러한 방식의 거버넌스에서 가장 먼저 해야 할 일은 변경 사항이 여러 도메인 서비스에 걸쳐 영향을 미치지 않도록 하는 것입니다.
만약 그렇다면, 이는 도메인 경계가 적절하게 정의되지 않았거나 서비스 기반 아키텍처 스타일이 해당 문제에 가장 적합하지 않다는 것을 나타내는 좋은 지표입니다.

서비스 간 통신이 불가능한 경우, 아키텍트는 도메인 서비스 간 통신량을 제어할 수도 있습니다. 물론 한 도메인 서비스가 다른 도메인 서비스와 통신해야 하는 상황이나 워크플로우도 있습니다. 예를 들어, 주문 처리 도메인은 고객에게 주문 상태 정보를 이메일로 보내기 위해 고객 알림 도메인 과 통신해야 할 수 있습니다 . 하지만 대부분의 경우, 도메인 서비스는 서로 상당히 독립적이어야 하며, 오케스트레이션은 UI 또는 API 게이트웨이 수준에서 이루어져야 합니다.

팀 토폴로지 고려 사항

서비스 기반 아키텍처는 도메인별로 분할되어 있기 때문에, 팀 구성 또한 도메인 영역별로 정렬될 때(예: 전문성을 갖춘 교차 기능 팀) 가장 효과적입니다.
도메인 기반 요구사항이 발생하면, 해당 도메인에 집중하는 다기능 팀이 다른 팀이나 서비스에 영향을 주지 않고 특정 도메인 서비스 내에서 해당 기능을 공동으로 개발할 수 있습니다. 반대로, UI 팀, 백엔드 팀, 데이터베이스 팀 등과 같이 기술적으로 분리된 팀은 도메인 분할로 인해 이러한 아키텍처 스타일과 잘 맞지 않습니다. 기술 중심 팀에 도메인 기반 요구사항을 할당하려면 팀 간의 원활한 의사소통과 협업이 필요한데, 이는 대부분의 조직에서 어려운 과제입니다.

다음은 "팀 토폴로지 및 구성" 에 설명된 특정 팀 토폴로지에 맞춰 서비스 기반 아키텍처를 설계할 때 건축가가 고려해야 할 몇 가지 사항입니다.
151페이지의 "건축" 항목:

스트림 중심 팀은 도메인
: 경계가 적절하게 정렬되어 있다면, 특히 특정 도메인에 집중된 스트림을 운영하는 경우 이 아키텍처 스타일에 잘 맞습니다. 그러나 스트림이 도메인 서비스로 정의된 경계를 넘어서면 서비스 기반 아키텍처는 더욱 어려워집니다. 이 경우 아키텍트는 도메인 서비스의 경계와 세분성을 분석하여 스트림에 맞춰 재정렬하거나 다른 아키텍처 스타일을 선택해야 합니다.

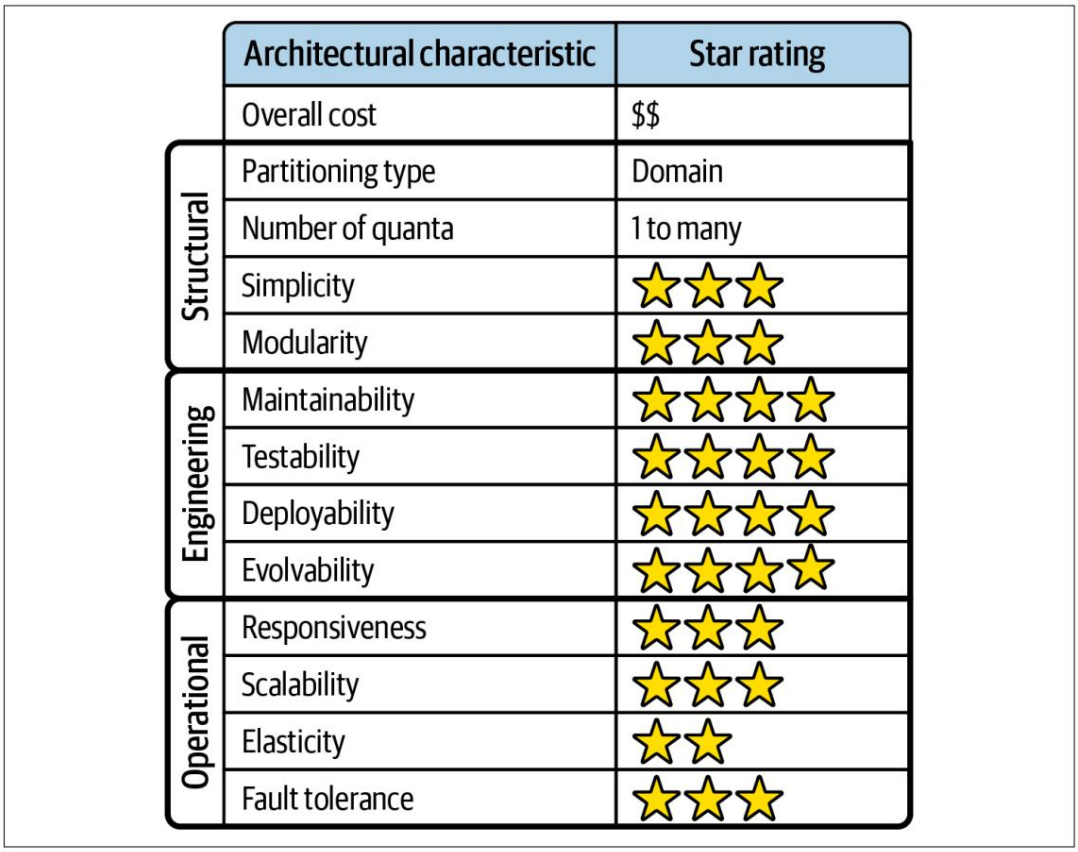
서비스 기반 아키텍
: 처는 도메인 서비스의 세분화되지 않은 특성 때문에 다른 분산 아키텍처에 비해 팀 지원 토폴로지와 결합할 때 효율성이 떨어집니다. 그러나 아키텍트는 각 도메인 서비스 내에서 적절한 구성 요소를 신중하게 식별하고 생성함으로써 이러한 스타일의 모듈성을 향상시킬 수 있습니다. 그러면 전문가와 여러 분야에 걸친 팀 구성원들이 이러한 구성 요소를 기반으로 제안을 하고 실험을 수행할 수 있습니다.

복잡한 하위 시스템 팀
: 복잡한 하위 시스템 팀은 이러한 아키텍처 스타일의 도메인 수준 및 하위 도메인 수준의 모듈성을 활용하여 다른 팀 구성원(및 서비스)과 독립적으로 복잡한 도메인 또는 하위 도메인 처리에 집중할 수 있습니다.

플랫폼 팀은 서비스 기
: 반 아키텍처의 높은 모듈성 덕분에 공통 도구, 서비스, API 및 작업을 활용하여 플랫폼 팀 토폴로지의 이점을 극대화할 수 있습니다.

스타일 특징

그림 14-8 의 특성 평가표에서 별 1개는 해당 아키텍처 특성이 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 아키텍처 특성이 아키텍처 스타일에서 가장 강력한 특징 중 하나임을 의미합니다. 평가표에 제시된 각 특성에 대한 정의는 4 장 에서 확인할 수 있습니다 .



	Architectural characteristic	Star rating
	Overall cost	$$
Structural	Partitioning type	Domain
Structural	Number of quanta	1 to many
Structural	Simplicity	★★★
Structural	Modularity	★★★
Engineering	Maintainability	★★★★
Engineering	Testability	★★★★
Engineering	Deployability	★★★★
Engineering	Evolvability	★★★★
Operational	Responsiveness	★★★
Operational	Scalability	★★★
Operational	Elasticity	★★
Operational	Fault tolerance	★★★

그림 14-8. 서비스 기반 아키텍처 특성 평가

서비스 기반 아키텍처는 도메인 분할 아키텍처로, 기술적 고려 사항(예: 프레젠테이션 로직 또는 영속성 로직)보다는 도메인에 따라 구조가 결정됩니다.

7장 에서 소개하고 13장 에서 다시 살펴본 전자제품 재활용 애플리케이션인 'Going Green'을 예로 들어 보겠습니다. 이 장에서는 'Going Green'이 서비스 기반 아키텍처를 사용한다고 가정합니다. 각 서비스는 별도로 배포되는 소프트웨어 단위로, 특정 영역(예: 품목 평가)에 한정됩니다. 이 영역 내에서 변경 사항이 발생하면 해당 서비스와 그에 상응하는 사용자 인터페이스 및 데이터베이스에만 영향을 미칩니다. 특정 평가 변경 사항을 지원하기 위해 다른 부분은 수정할 필요가 없습니다.

분산 아키텍처에서 퀀타의 수는 하나 이상일 수 있습니다. 예를 들어, Going Green의 모든 서비스가 동일한 데이터베이스나 UI를 공유하는 경우 전체 시스템은 단일 퀀타가 됩니다. 그러나 211페이지의 "스타일 세부 사항" 에서 논의된 것처럼 UI와 데이터베이스 모두 연합(분리)될 수 있으며, 이로 인해 전체 시스템 내에 여러 개의 퀀타가 존재할 수 있습니다. 그림 14-9 에서 Going Green 시스템은 두 개의 퀀타를 포함합니다. 하나는 고객 대면 애플리케이션 부분으로, 별도의 고객 UI, 데이터베이스 및 서비스 (견적 및 품목 상태)를 포함합니다. 다른 하나는 전자 기기의 수거, 평가 및 재활용과 같은 내부 운영을 담당합니다. 내부 운영 퀀타는 별도로 배포된 서비스와 두 개의 개별 UI를 포함하지만, 모두 동일한 데이터베이스를 공유하므로 애플리케이션의 내부 운영 부분은 단일 퀀타가 됩니다.

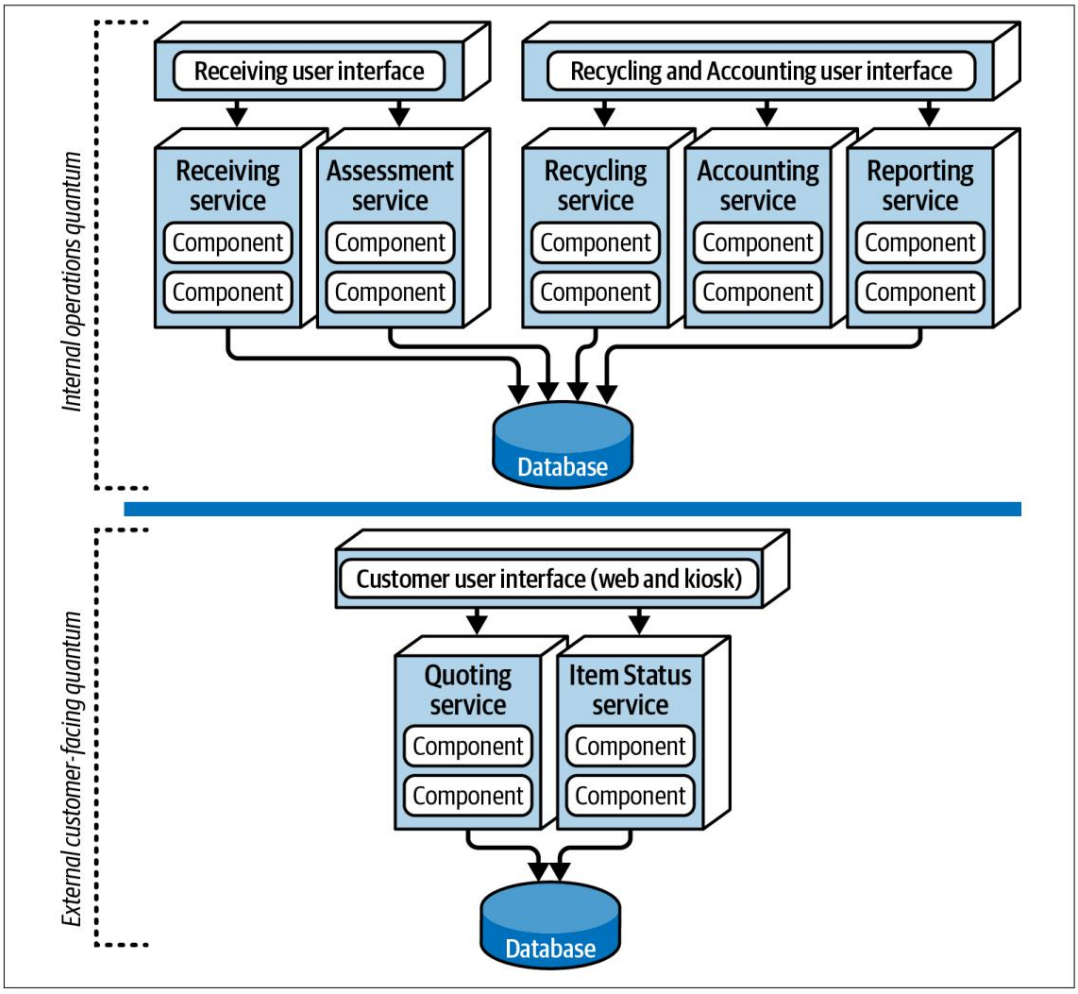


그림 14-9. 서비스 기반 아키텍처에서 분리된 양자

서비스 기반 아키텍처에 만점(5점)을 주지는 않았지만, 여러 중요한 영역에서 높은 점수(4점)를 받았습니다. 이 아키텍처 스타일을 사용하여 애플리케이션을 도메인별로 분리된 서비스로 배포하면 변경 속도가 빨라지고(애자일), 도메인 범위 지정에 기반한 모듈화로 테스트 범위가 넓어지며(테스트 용이성), 단일체 아키텍처보다 위험 부담이 적으면서 더 자주 배포할 수 있습니다(배포 용이성). 이러한 세 가지 특징은 제품 출시 시간을 단축하여 기업이 새로운 기능을 제공하고 버그를 비교적 빠르게 수정할 수 있도록 해줍니다.

서비스 기반 아키텍처는 내결함성과 전반적인 애플리케이션 가용성 측면에서도 높은 평가를 받습니다. 도메인 서비스는 일반적으로 세분화되지 않은 경우가 많지만, 이러한 아키텍처 스타일에서 서비스는 대개 자체적으로 완결성을 가지며 코드와 데이터베이스 공유 덕분에 서비스 간 통신이 거의 필요하지 않다는 점에서 높은 별 4개 등급을 받았습니다. 따라서 하나의 도메인 서비스(예: Going Green의 수신 서비스)에 장애가 발생하더라도 나머지 6개 서비스에는 영향을 미치지 않습니다.

서비스의 세분화되지 않은 특성으로 인해 확장성은 별 3개에 그쳤으며, 탄력성 또한 별 2개에 머물렀습니다. 이러한 아키텍처 스타일에서도 프로그래밍 방식의 확장성과 탄력성은 분명히 가능하지만, 마이크로서비스와 같이 세분화된 서비스를 사용하는 아키텍처 스타일보다 더 많은 기능을 복제하기 때문에 비용 효율성과 시스템 자원 효율성이 떨어집니다.

일반적으로 서비스 기반 아키텍처는 처리량 향상이나 장애 조치가 필요한 경우가 아니면 각 서비스의 인스턴스를 하나만 사용합니다. 그림 14-9 에 나타난 Going Green이 좋은 예입니다. Going Green에서는 견적 및 품목 상태 서비스 만 고객 문의량이 많아도 확장이 필요합니다. 다른 운영 서비스는 단일 인스턴스만 필요하므로 단일 인메모리 캐싱이나 데이터베이스 연결 풀링과 같은 기능을 더 쉽게 지원할 수 있습니다.

단순성과 전반적인 비용 효율성은 서비스 기반 아키텍처를 마이크로서비스, 이벤트 기반 아키텍처, 심지어 공간 기반 아키텍처와 같은 더 비싸고 복잡한 분산 아키텍처와 차별화하는 두 가지 주요 요인입니다. 이러한 특징 덕분에 서비스 기반 아키텍처는 구현하기 가장 쉽고 비용 효율적인 분산 아키텍처 중 하나입니다. 이는 매력적인 장점이지만, 언제나 그렇듯이 장단점이 존재합니다. 비용과 복잡성이 높을수록 확장성, 탄력성, 내결함성과 같은 네 가지 핵심 특성이 더욱 뛰어나게 됩니다.

유연성과 더불어 3성급 및 4성급 아키텍처 특성이 다양하게 나타나는 서비스 기반 아키텍처는 가장 실용적인 스타일 중 하나입니다.
훨씬 강력한 분산 아키텍처 스타일도 많지만, 많은 기업들은 그러한 강력한 성능에 드는 비용이 너무 높다고 생각합니다. 또 어떤 기업들은 애초에 그렇게 강력한 성능이 필요하지 않다고 생각하기도 합니다. 마치 페라리를 사놓고 출퇴근길 교통 체증에 시달리며 운전하는 것과 같습니다. 멋있어 보이긴 하지만, 성능과 속도, 민첩성을 낭비하는 셈이죠!

서비스 기반 아키텍처는 도메인 주도 설계와도 자연스럽게 어울립니다. 서비스는 세분화되지 않고 도메인 범위로 구성되기 때문에 각 도메인은 해당 도메인을 포괄하는 별도로 배포된 서비스에 잘 들어맞습니다. 이러한 기능을 단일 소프트웨어 단위로 분리하면 해당 도메인에 변경 사항을 적용하기가 더 쉬워집니다.

분산 아키텍처에서는 데이터베이스 트랜잭션을 유지 관리하고 조정하는 것이 항상 문제입니다. 이러한 아키텍처는 일반적으로 전통적인 ACID 트랜잭션(데이터베이스 업데이트가 단일 작업 단위로 조정되고 함께 수행됨)보다는 최종 일관성(독립적인 데이터베이스 업데이트가 결국 서로 동기화됨)에 의존하기 때문입니다. 서비스 기반 아키텍처는 도메인 서비스가 세분화되지 않은(coarse-grained) 구조이기 때문에 다른 어떤 분산 아키텍처 스타일보다 ACID 트랜잭션을 더 효과적으로 활용합니다. 즉, 트랜잭션 범위가 특정 도메인 서비스로 설정되어 대부분의 모놀리식 애플리케이션에서 볼 수 있는 전통적인 커밋-롤백 트랜잭션 기능을 사용할 수 있습니다.

마지막으로, 서비스 기반 아키텍처는 세분화 및 서비스 조정의 복잡성에 얽매이지 않고 높은 수준의 모듈성을 달성하고자 하는 아키텍트에게 좋은 선택입니다(18 장 341페이지 의 "안무 및 오케스트레이션" 참조).

예시 및 사용 사례

서비스 기반 아키텍처 스타일의 유연성과 강력함을 설명하기 위해, 아이폰이나 갤럭시 휴대폰과 같은 오래된 전자기기를 재활용하는 시스템인 '고잉 그린(Going Green)' 사례를 다시 살펴보겠습니다.

Going Green의 처리 과정은 다음과 같습니다.

1. 고객은 Going Green(웹사이트 또는 키오스크를 통해)에 비용이 얼마나 드는지 문의합니다.
   중고 전자 기기 매입 비용 (견적 포함).
2. 견적에 만족하시면 고객은 기기를 재활용 업체로 보내주시면 됩니다!
   회사(수신).
3. Going Green은 기기의 상태를 평가합니다(평가).
4. 기기가 양호한 작동 상태인 경우, Going Green은 고객에게 해당 기기에 대한 대금을 지급합니다(회계 처리).
   이 과정에서 고객은 언제든지 웹사이트에 접속하여 기기의 상태를 확인할 수 있습니다(기기 상태).
5. 평가 결과에 따라 Going Green은 해당 기기를 안전하게 폐기하고 부품을 재활용하거나, Facebook Marketplace 또는 eBay와 같은 제3자 판매 플랫폼에서 재판매합니다(재활용).
6. Going Green은 재활용 활동에 대한 재무 및 운영 보고서를 정기적으로 작성합니다(보고).

그림 14-10은 서비스 기반 아키텍처를 사용하여 구현된 이 시스템을 보여줍니다. 앞서 언급한 각 도메인 영역은 별도로 배포되는 독립적인 도메인 서비스로 구현됩니다. 확장이 필요한 서비스(따라서 여러 서비스 인스턴스가 필요한 서비스)는 높은 처리량이 필요한 서비스입니다(이 경우 고객에게 제공되는 견적 및 품목 상태 서비스). 다른 서비스는 확장이 필요하지 않으므로 단일 서비스 인스턴스만 필요합니다.

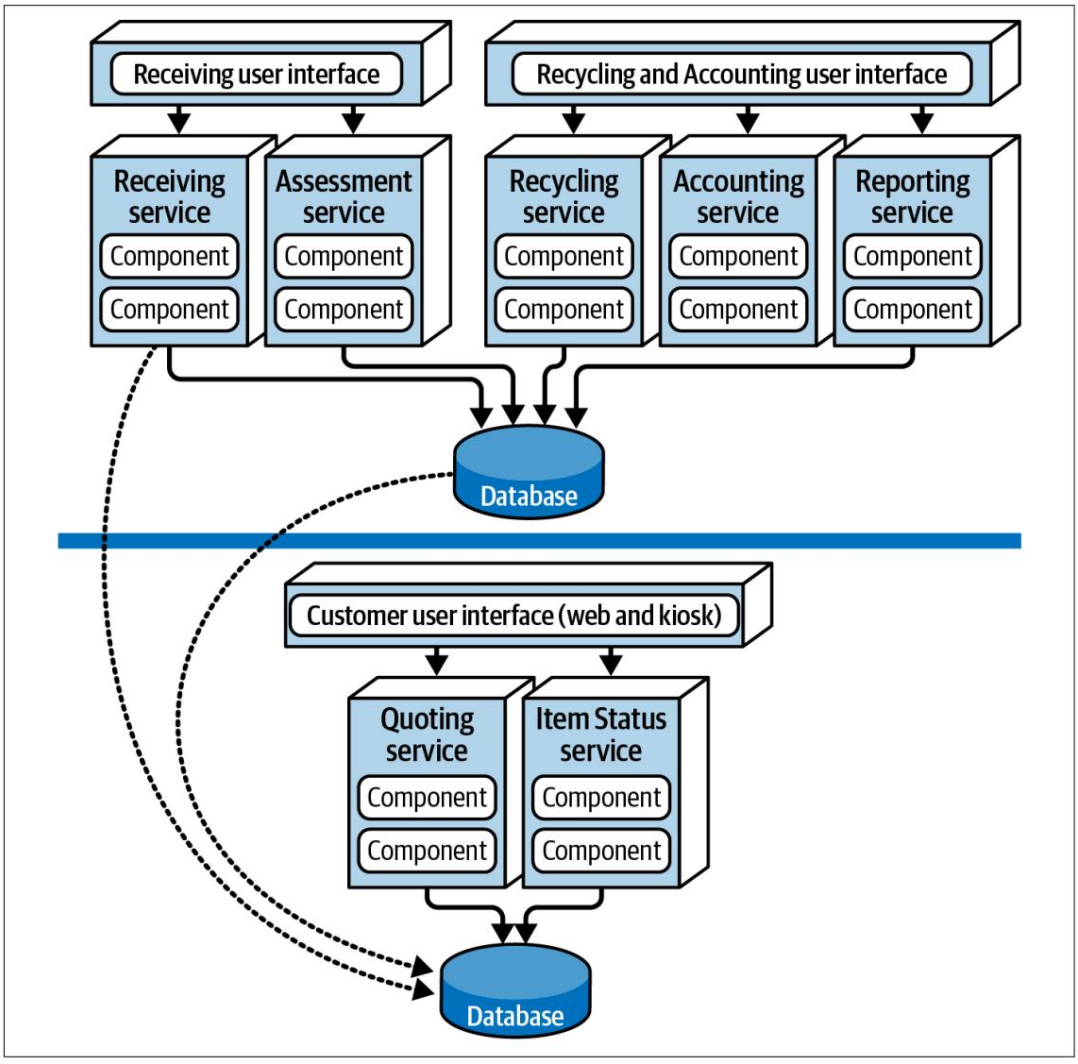


그림 14-10. 서비스 기반 아키텍처를 사용하는 Going Green의 전자제품 재활용 애플리케이션

이 예시에서는 UI 애플리케이션이 고객 응대, 수령, 재활용 및 회계 도메인으로 분리되어 있습니다. 이러한 분리는 UI 수준에서 뛰어난 내결함성, 확장성, 그리고 적절한 보안(외부 고객이 내부 기능에 접근할 수 있는 네트워크 경로가 없기 때문)을 제공합니다. 또한, 다음 사항에도 유의하십시오.

두 개의 물리적 데이터베이스가 별도로 존재합니다. 하나는 외부 고객 대면 운영용이고, 다른 하나는 내부 운영용입니다. 이러한 구성을 통해 내부 데이터와 운영은 외부 운영과 분리된 네트워크 영역(가로선으로 표시됨)에 상주할 수 있습니다. 또한, 접근 제한 및 데이터 보호 측면에서 훨씬 뛰어난 보안을 제공하며, 별도의 아키텍처 영역을 구성합니다. 방화벽을 통한 단방향 접근을 통해 내부 서비스는 고객 대면 정보에 접근하고 업데이트할 수 있지만, 그 반대는 불가능합니다. 또는 데이터베이스에 따라 내부 테이블 미러링과 외부 테이블 동기화를 사용하여 두 데이터베이스 간의 데이터를 동기화할 수도 있습니다.

또한, 평가 서비스는 새로운 제품이 입고되거나 출시됨에 따라 지속적으로 변경됩니다. 서비스 기반 아키텍처를 사용하면 이러한 빈번한 변경 사항이 단일 도메인 서비스로 격리되어 민첩성, 테스트 용이성 및 배포 용이성을 제공합니다.

서비스 기반 아키텍처는 매우 유연한 아키텍처 스타일로, 다른 분산 아키텍처에 비해 비교적 저렴한 비용으로 도메인 분할, 확장성, 민첩성, 내결함성, 가용성 및 응답성을 제공합니다.

이러한 요인들 때문에 인기 있는 선택지가 되었습니다.

서비스 기반 아키텍처는 조직이 다른 분산 아키텍처 스타일로 마이그레이션하든, 처음부터 새로운 분산 시스템을 구축하든 관계없이 다른 분산 아키텍처로 마이그레이션하기 위한 좋은 "디딤돌" 역할을 합니다. 이것이 바로 우리가 말하고자 하는 핵심입니다.

> 애플리케이션의 모든 부분이 마이크로서비스일 필요는 없습니다.
> —마크 리처즈

목표 아키텍처 스타일로 마이그레이션하기 전에 서비스 기반 아키텍처를 "디딤돌"로 활용하면 팀은 도메인을 분석하고 아키텍처의 어떤 부분을 마이크로서비스로 구현해야 할지 결정할 수 있습니다. 예를 들어, '친환경 캠페인' 사례에서 재활용 및 회계 서비스는 더 이상 세분화할 필요 없이 도메인 서비스로 유지하는 것이 적절합니다. 하지만 평가 서비스는 자주 변경되고 높은 수준의 민첩성이 요구되므로 각 전자 기기 유형별로 별도의 서비스로 분리해야 합니다. 만약 '친환경 캠페인' 팀이 이 단계를 건너뛰고 바로 마이크로서비스 아키텍처로 전환했다면, 필요하지 않은 기능까지 포함하여 모든 기능이 마이크로서비스로 구현되었을 가능성이 높습니다.

이것들은 서비스 기반 아키텍처가 아키텍트들 사이에서 선호되는 여러 이유 중 일부에 불과합니다. 하지만 이는 다양한 분산 아키텍처 스타일 중 하나일 뿐이며, 특정 비즈니스 문제에 가장 적합한 아키텍처를 결정하기 위해서는 모든 스타일을 이해하는 것이 중요합니다. 따라서 다른 분산 아키텍처들을 살펴보겠습니다.