제18장

마이크로서비스 아키텍처

마이크로서비스는 최근 몇 년 동안 상당한 인기를 얻은 매우 인기 있는 아키텍처 스타일입니다. 이 장에서는 위상적, 철학적 측면 모두에서 이 아키텍처를 차별화하는 중요한 특징들을 개괄적으로 살펴봅니다.

대부분의 아키텍처 스타일은 특정 패턴이 반복적으로 나타나는 것을 발견한 건축가들이 만들어낸 이름에서 유래합니다. 차세대 트렌드를 결정하는 비밀스러운 건축가 집단은 존재하지 않습니다. 소프트웨어 개발 생태계가 변화함에 따라 건축가들은 여러 가지 결정을 내리고, 이러한 변화에 대처하고 이를 통해 이익을 얻는 가장 일반적인 방법들 중에서 가장 효과적인 것으로 인정받는 것들이 아키텍처 스타일로 자리잡고, 다른 사람들이 이를 모방하게 됩니다.

마이크로서비스는 이러한 점에서 차별화됩니다. 비교적 초기에 이름이 붙여졌기 때문입니다. 마틴 파울러와 제임스 루이스는 2014년 유명한 블로그 게시물 에서 이 새로운 아키텍처 스타일의 특징을 정의하고 설명하며 대중화했습니다. 이 게시물은 마이크로서비스 아키텍처의 정의를 확립하고, 마이크로서비스에 관심 있는 아키텍트들이 그 기본 철학을 이해하는 데 도움을 주었습니다.

마이크로서비스는 소프트웨어 프로젝트를 위한 논리적 설계 프로세스인 도메인 주도 설계(DDD)의 아이디어에서 큰 영감을 받았습니다. 특히 DDD의 한 개념인 경계 컨텍스트(Bounded Context)는 마이크로서비스에 결정적인 영향을 미쳤습니다. 경계 컨텍스트는 결합도를 낮추는 스타일을 나타내며(7 장 97 페이지 의 "도메인 주도 설계의 경계 컨텍스트" 에서 이미 논의됨), 이 때문에 마이크로서비스를 "공유하지 않는(share nothing)" 아키텍처라고 부르기도 합니다.

개발자가 도메인을 정의할 때, 해당 도메인에는 코드 및 데이터베이스 스키마와 같은 산출물에 명시된 여러 엔티티와 동작이 포함됩니다. 예를 들어, 애플리케이션은 카탈로그 항목, 고객, 결제와 같은 개념을 포함하는 CatalogCheckout 이라는 도메인을 가질 수 있습니다 . 기존의 모놀리식 아키텍처에서는 개발자들이 이러한 개념들을 공유하며 재사용 가능한 클래스를 구축하고 서로 연결됩니다.

데이터베이스. 경계 컨텍스트 내에서는 코드와 데이터 스키마 같은 내부 구성 요소들을 결합하여 작업을 생성할 수 있지만, 다른 경계 컨텍스트의 데이터베이스나 클래스 정의와 같이 경계 컨텍스트 외부의 어떤 것과도 결합되지 않습니다. 따라서 각 컨텍스트는 다른 구성 요소를 수용하지 않고 필요한 것만 정의할 수 있으므로 경계 컨텍스트 간의 재사용이 제한됩니다.

재사용은 일반적으로 유익하지만, 소프트웨어 아키텍처의 제1법칙을 기억하세요.
모든 것에는 장단점이 있다. 재사용의 단점은 일반적으로 상속이나 구성을 통해 시스템의 결합도를 높여야 한다는 점이다.

만약 아키텍트의 목표가 마이크로서비스의 주요 목표인 고도로 분리된 시스템이라면, 그들은 재사용보다는 복제를 선호할 것이며, 서비스와 그에 상응하는 데이터를 포함하도록 경계 컨텍스트라는 논리적 개념을 물리적으로 모델링할 것입니다.

위상수학

그림 18-1 은 마이크로서비스의 기본 토폴로지를 보여줍니다 . 단일 목적 아키텍처의 특성상, 이 아키텍처 스타일의 서비스는 오케스트레이션 기반 SOA (17장), 이벤트 기반 아키텍처 (15장), 서비스 기반 아키텍처 (14장)와 같은 다른 분산 아키텍처보다 훨씬 규모가 작습니다. 아키텍트는 각 서비스가 데이터베이스 및 기타 종속 구성 요소를 포함하여 독립적으로 작동하는 데 필요한 모든 부분을 포함할 것으로 예상합니다.

마이크로서비스는 분산 아키텍처 스타일로, 각 서비스는 가상 머신이나 컨테이너와 같은 자체 프로세스에서 실행됩니다. 이처럼 서비스를 분리함으로써 애플리케이션 호스팅을 위한 멀티테넌트 인프라가 많이 사용되는 아키텍처에서 흔히 발생하는 문제를 간단하게 해결할 수 있습니다. 예를 들어, 시스템이 애플리케이션 서버를 사용하여 여러 실행 중인 애플리케이션을 관리하는 경우, 애플리케이션 서버는 네트워크 대역폭, 메모리, 디스크 공간 등을 재사용할 수 있습니다. 그러나 지원되는 모든 애플리케이션의 규모가 계속 커지면 결국 공유 인프라의 특정 리소스가 부족해질 수 있습니다.

또 다른 문제는 공유 애플리케이션 간의 부적절한 격리와 관련이 있습니다. 각 서비스를 별도의 프로세스로 분리하면 공유로 인해 발생하는 모든 문제를 해결할 수 있습니다.
자유롭게 이용 가능한 오픈 소스 운영 체제와 자동화된 머신 프로비저닝 기술이 발전하기 전에는 각 도메인이 자체 인프라를 구축하는 것이 비현실적이었습니다. 하지만 이제 클라우드 리소스와 컨테이너 기술(351페이지의 "클라우드 고려 사항" 참조) 덕분에 팀은 도메인 수준과 운영 수준 모두에서 극도의 분리 효과를 누릴 수 있습니다.

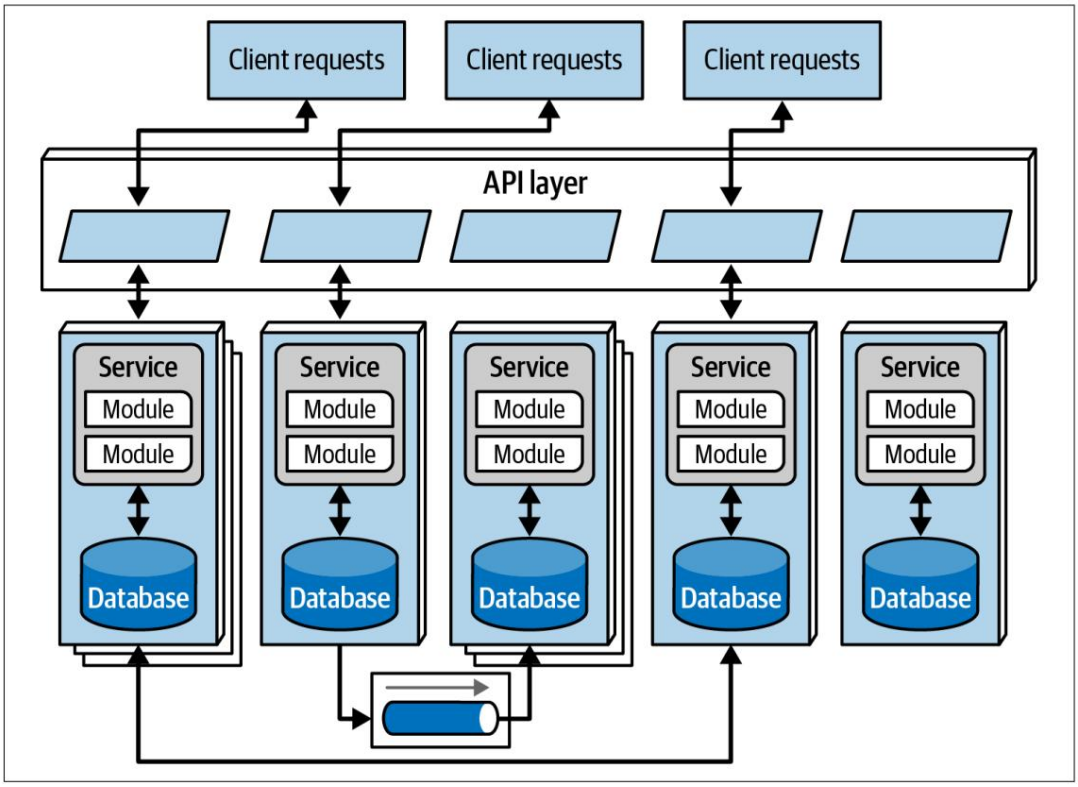


그림 18-1. 마이크로서비스 아키텍처 스타일의 토폴로지

마이크로서비스의 분산 특성은 종종 성능 저하라는 단점을 초래합니다. 네트워크 호출은 메서드 호출보다 훨씬 오래 걸리고, 모든 엔드포인트에서 수행되는 보안 검증은 추가적인 처리 시간을 발생시키므로, 아키텍트는 세분화 정도가 성능에 미치는 영향을 신중하게 고려해야 합니다.

마이크로서비스는 분산 아키텍처이기 때문에, 경험이 풍부한 아키텍트들은 서비스 경계를 넘나드는 트랜잭션 사용을 권장하지 않습니다. 이 아키텍처에서 성공의 핵심은 서비스의 세분성을 결정하는 것입니다.

스타일 사양

다음 섹션에서는 모든 내용을 다루지는 않지만, 마이크로서비스 토폴로지의 중요한 측면과 특히 고유한 요소들을 설명합니다.
마이크로서비스.

경계 컨텍스트

마이크로서비스의 핵심 철학이라고 언급했던 경계 컨텍스트(Bounded Context) 개념을 좀 더 자세히 살펴보겠습니다. 각 서비스는 특정 기능, 하위 도메인 또는 워크플로를 모델링합니다. 따라서 각 경계 컨텍스트에는 필요한 모든 것이 포함됩니다.

해당 기능 또는 하위 도메인 내에서 작동하도록 설계되었으며, 여기에는 논리적 구성 요소 및 클래스, 데이터베이스 스키마, 그리고 서비스가 기능을 수행하는 데 필요한 해당 데이터베이스로 구성된 서비스가 포함됩니다. 각 서비스는 특정 하위 도메인 또는 기능을 나타내도록 설계되었습니다.

이러한 철학은 마이크로서비스 내에서 아키텍트가 내리는 많은 결정의 원동력이 됩니다.
예를 들어, 모놀리식 아키텍처에서는 개발자들이 Address 와 같은 공통 클래스를 애플리케이션의 여러 부분에서 공유하는 것이 일반적입니다. 하지만 마이크로서비스 아키텍처는 결합도를 낮추는 것을 목표로 하기 때문에, 이러한 아키텍처 스타일을 따르는 아키텍트는 결합을 피하기보다는 코드 복제를 통해 모든 코드를 해당 함수 또는 하위 도메인의 경계 컨텍스트 내에 유지하는 방식을 사용합니다.

마이크로서비스는 도메인 분할 아키텍처의 개념을 극단적으로 구현한 것으로, 여러 면에서 도메인 주도 설계의 논리적 개념을 물리적으로 구현한 것이라고 할 수 있습니다.

세분성

마이크로서비스를 설계하는 아키텍트는 종종 적절한 세분화 수준을 찾는 데 어려움을 겪고, '마이크로'라는 용어를 문자 그대로 받아들여 서비스를 너무 작게 만드는 결과를 낳습니다.

그러면 유용한 작업을 수행하기 위해 서비스 간에 다시 통신 연결을 구축해야 하는데, 이는 애초의 목적을 무효화하고 결국 거대한 분산형 진흙 덩어리로 이어집니다.

> 마이크로서비스라는 용어는 명칭일 뿐, 설명이 아닙니다.
> —마틴 파울러

다시 말해, 이 용어를 만든 사람들은 이러한 새로운 스타일을 부를 이름이 필요했고, 당시(2007년경) 지배적인 아키텍처 스타일이었던 서비스 지향 아키텍처(SOA)와 대비시키기 위해 마이크로서비스라는 용어를 선택했습니다. SOA는 "거대 서비스"라고 부를 수도 있었을 것입니다. 그러나 많은 개발자들이 마이크로서비스라는 용어를 설명이 아닌 명령처럼 받아들여 지나치게 세분화된 서비스를 만들어냅니다.

마이크로서비스에서 서비스 경계의 목적은 도메인 또는 워크플로를 정의하는 것입니다. 일부 애플리케이션에서는 비즈니스 프로세스 간의 결합도가 다른 프로세스보다 높기 때문에 시스템의 일부 영역에서 이러한 자연스러운 경계가 넓어질 수 있습니다.
다음은 건축가가 적절한 경계를 찾는 데 도움이 될 수 있는 몇 가지 지침입니다.

목적. 가장
: 명확한 경계는 아키텍처 스타일의 영감의 원천인 문제 영역에 있습니다. 이상적으로 각 마이크로서비스는 기능적으로 응집력이 있어야 하며, 전체 애플리케이션을 위해 하나의 중요한 동작을 수행해야 합니다.

업무
: 경계 컨텍스트는 비즈니스 워크플로이며, 트랜잭션에서 협력해야 하는 엔티티는 종종 좋은 서비스 경계를 제시합니다. 트랜잭션은

분산 아키텍처에서 문제를 일으킬 수 있으므로, 이러한 문제를 피하는 것을 목표로 시스템을 설계하면 더 나은 설계를 얻을 수 있습니다.

안무
(Choreography)는 뛰어난 도메인 격리를 제공하는 서비스 집합이지만, 제대로 작동하려면 광범위한 통신이 필요합니다. 아키텍트는 통신 오버헤드를 줄이기 위해 이러한 서비스들을 더 큰 서비스로 묶는 것을 고려할 수 있습니다.

반복은 훌륭한 서비스 디자인을 보장하는 유일한 방법입니다. 아키텍트는 처음부터 완벽한 세분화 수준, 데이터 종속성 및 커뮤니케이션 스타일을 찾아내는 경우가 드뭅니다. 시스템과 비즈니스 기능에 대해 더 많이 배우면서 디자인을 다듬기 위해 다양한 옵션을 반복적으로 검토합니다.

데이터 격리

경계 컨텍스트 개념에 따라 요구되는 또 다른 사항은 데이터 격리입니다. 많은 아키텍처 스타일에서 영구 저장을 위해 단일 데이터베이스를 사용합니다.
하지만 마이크로서비스는 공유 스키마와 데이터베이스를 통합 지점으로 사용하는 것을 포함하여 모든 종류의 결합을 피하려고 합니다.

서비스 세분성을 고려할 때 데이터 격리 또한 중요한 요소입니다. 엔티티 함정(115 페이지의 "엔티티 함정" 에서 설명)에 주의해야 합니다. 서비스를 데이터베이스의 단일 엔티티처럼 모델링해서는 안 됩니다. 아키텍트는 시스템 내 값을 통합하고 단일 진실 소스를 만들기 위해 관계형 데이터베이스를 사용하는 데 익숙하지만, 아키텍처 전체에 데이터가 분산되어 있는 경우에는 더 이상 이러한 방식이 적합하지 않습니다. 따라서 모든 아키텍트는 이 문제를 어떻게 처리할지 결정해야 합니다. 특정 사실에 대한 진실 소스를 하나의 도메인으로 지정하고 해당 도메인과 연동하여 값을 가져오거나, 데이터베이스 복제 또는 캐싱을 통해 정보를 분산하는 방식이 있습니다.

이처럼 데이터 격리 수준이 높으면 골칫거리가 생기기도 하지만, 기회도 제공됩니다.
이제 단일 데이터베이스를 중심으로 통합할 필요가 없어졌으므로 각 팀은 서비스 예산, 저장 구조 유형, 운영 특성, 프로세스 특성 등을 고려하여 가장 적합한 데이터베이스 기술을 선택할 수 있습니다. 고도로 분리된 시스템의 또 다른 장점은 어떤 팀이든 다른 팀에 영향을 주지 않고 더 적합한 데이터베이스(또는 기타 종속 요소)를 선택하여 방향을 바꿀 수 있다는 것입니다. 다른 팀은 구현 세부 사항에 종속될 수 없기 때문입니다. (데이터 격리 및 데이터베이스 고려 사항에 대한 자세한 내용은 348페이지의 "데이터 토폴로지"에서 다룹니다.)

대부분의 마이

크로서비스 아키텍처는 시스템의 소비자(사용자 인터페이스 또는 다른 시스템에서의 호출)와 마이크로서비스 사이에 API 계층(일반적으로 API 게이트웨이라고 함)을 포함합니다. API 계층은 간단한 리버스 프록시로 구현될 수도 있고, 보안, 네이밍 서비스 등과 같은 공통 관심사를 포함하는 더욱 정교한 게이트웨이로 구현될 수도 있습니다(이러한 내용은 "운영 재사용"에서 자세히 다룹니다).

API 레이어는 다양한 용도로 사용될 수 있지만, 이 아키텍처의 기본 철학에 충실하려면 미들웨어 또는 오케스트레이터로 사용해서는 안 됩니다. 이 아키텍처에서 중요한 모든 비즈니스 로직은 경계 컨텍스트 내에 있어야 하며, 오케스트레이션이나 다른 비즈니스 로직을 미들웨어에 넣는 것은 이 규칙을 위반하는 것입니다.

일반적으로 아키텍트는 기술적으로 분할된 아키텍처에서 미들웨어를 사용하는 반면, 마이크로서비스는 도메인별로 명확하게 분할됩니다.

마이크로서비스 아키텍처에서 API 계층을 사용할 때는 요청 라우팅과 보안, 모니터링, 로깅 등과 같은 횡단 관심사만 포함해야 합니다. 비즈니스 관련 로직을 API 계층에 넣지 않도록 주의해야 합니다.

운영 재사용

마이크로서비스 아키텍처는 결합보다는 중복을 선호하는데, 그렇다면 모니터링, 로깅, 회로 차단기와 같은 운영 관련 기능처럼 결합이 필요한 부분은 어떻게 처리해야 할까요? 전통적인 서비스 지향 아키텍처 철학은 도메인 기능과 운영 기능을 최대한 재사용하는 것이었습니다. 하지만 마이크로서비스 아키텍처에서는 이 두 가지를 분리하려고 합니다.

우려 사항.

팀이 여러 마이크로서비스를 구축하고 나면, 각 마이크로서비스에는 유사성을 통해 이점을 얻을 수 있는 공통 요소가 있다는 것을 깨닫게 됩니다. 예를 들어, 조직에서 각 서비스 팀이 모니터링을 독립적으로 구현하도록 허용하는 경우, 각 팀이 실제로 그렇게 하도록 어떻게 보장할 수 있을까요? 또한 조직은 업그레이드와 같은 문제를 어떻게 처리해야 할까요? 각 팀이 모니터링 도구의 새 버전으로 업그레이드해야 하는 책임은 누구에게 있으며, 업그레이드에는 얼마나 시간이 걸릴까요? 사이드카 패턴은 이러한 문제에 대한 해결책을 제시합니다 (그림 18-2).

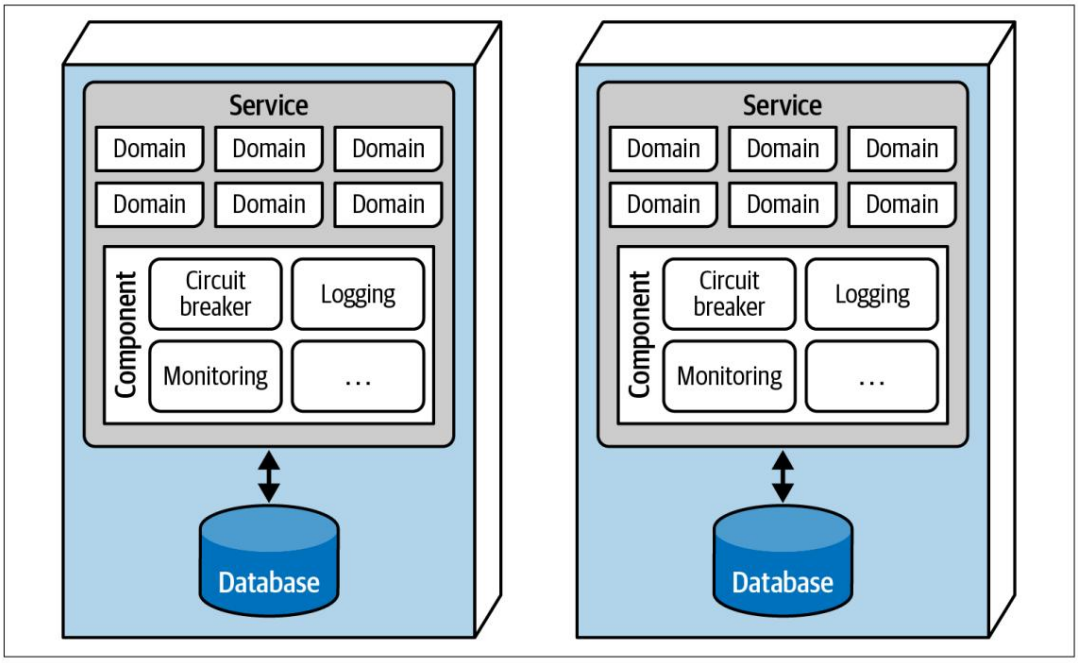


그림 18-2. 마이크로서비스에서의 사이드카 패턴

그림 18-2 에서 공통적인 운영 관련 사항(회로 차단기, 로깅, 모니터링)은 각 서비스 내에 별도의 구성 요소로 나타나며, 각 구성 요소는 개별 팀 또는 공유 인프라 팀에서 소유할 수 있습니다. 사이드카 구성 요소는 결합을 통해 이점을 얻을 수 있는 모든 운영 관련 사항을 처리하므로, 모니터링 도구를 업그레이드해야 할 때 공유 인프라 팀이 사이드카를 업데이트하면 각 마이크로서비스가 새로운 기능을 받게 됩니다 (353페이지의 "팀 토폴로지 고려 사항" 참조).

각 서비스에 공통 사이드카 구성 요소가 포함되면 아키텍트는 팀이 아키텍처 전체에서 이러한 공통 관심사를 통합적으로 제어할 수 있도록 서비스 메시를 구축할 수 있습니다. 그림 18-3 에서 보는 것처럼 사이드카 구성 요소는 연결되어 모든 마이크로서비스에 걸쳐 일관된 운영 인터페이스를 형성합니다.

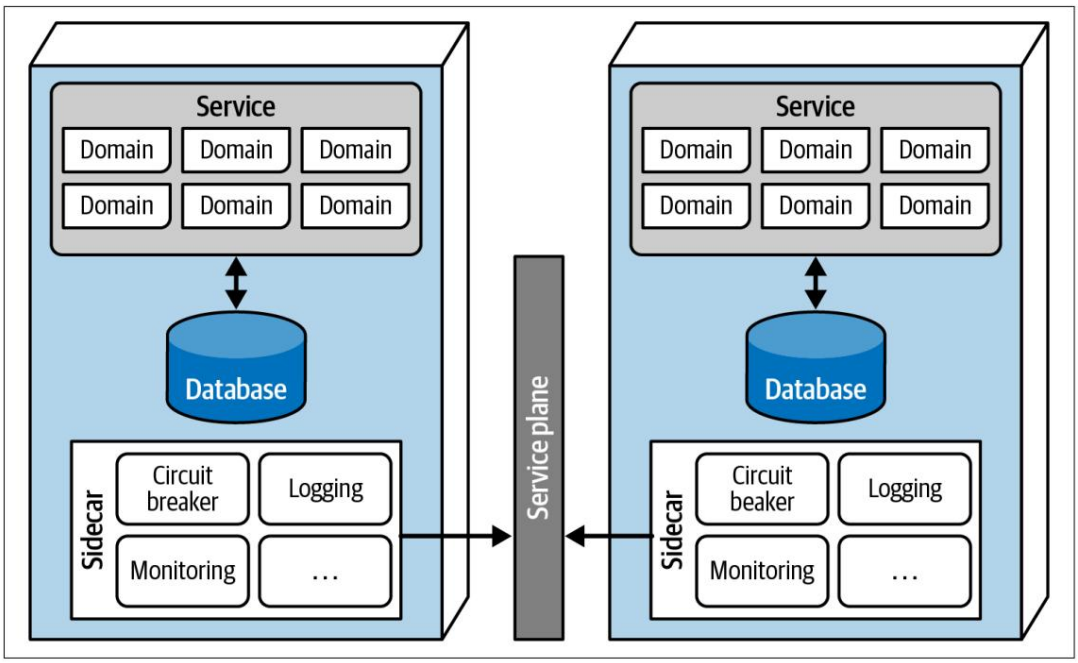


그림 18-3. e 서비스 플레인은 서비스 메시에서 사이드카들을 연결합니다.

그림 18-3 에서 각 사이드카는 서비스 평면에 연결됩니다. 서비스 평면은 각 사이드카를 일관된 인터페이스를 사용하여 연결하고 서비스 메시를 형성하는 통합 소프트웨어(일반적으로 Istio 와 같은 제품 형태)입니다 .

그림 18-4 에서 보는 바와 같이 각 서비스는 전체 메시의 노드를 형성합니다 . 서비스 메시는 팀이 모니터링 수준, 로깅 및 기타 공통 운영 문제와 같은 운영 연결을 전역적으로 제어할 수 있는 콘솔 역할을 합니다.

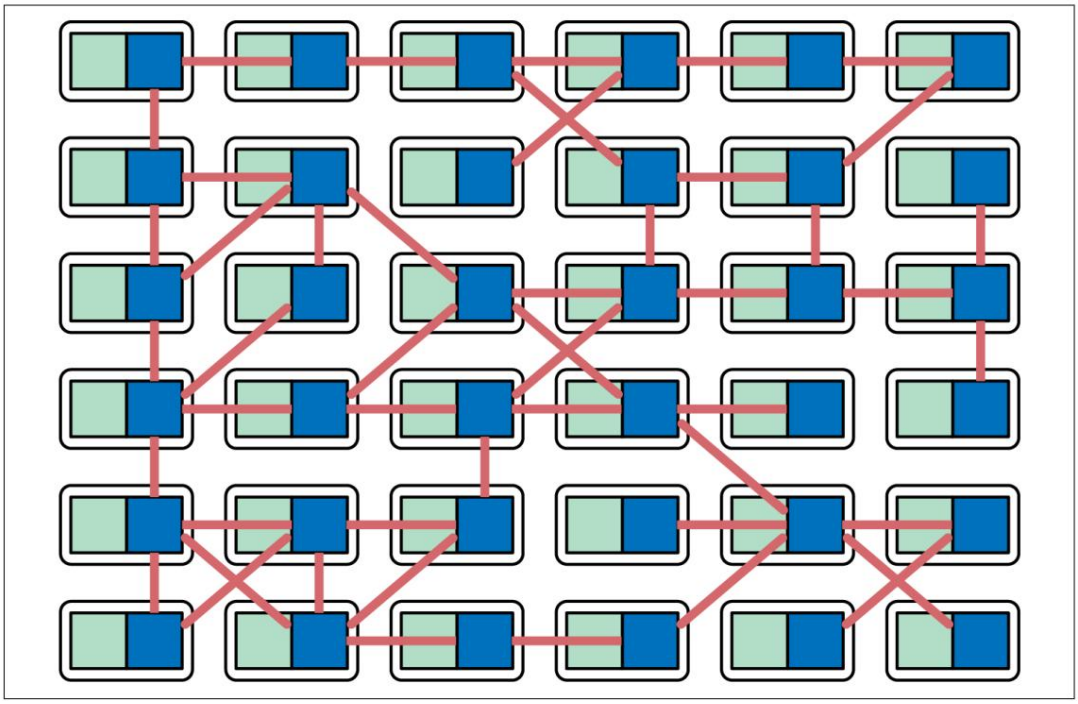


그림 18-4. 마이크로서비스 e서비스메시는 운영 측면에 대한 전체적인 관점을 제공합니다.

아키텍트는 마이크로서비스 아키텍처에 확장성을 구축하는 방법으로 서비스 디스커버리를 활용합니다 . 서비스 디스커버리는 네트워크 내에서 서비스를 자동으로 감지하고 찾는 방법입니다. 요청이 들어오면 단일 서비스를 직접 호출하는 대신 서비스 디스커버리 도구를 거치게 되는데, 이 도구는 요청 수와 빈도를 모니터링하고 확장성 또는 탄력성 문제를 처리하기 위해 새로운 서비스 인스턴스를 생성할 수 있습니다. 아키텍트는 종종 서비스 메시에 서비스 디스커버리를 포함시켜 모든 마이크로서비스에 통합합니다. API 계층은 서비스 디스커버리를 호스팅하는 데 자주 사용되며, 사용자 인터페이스나 다른 호출 시스템이 탄력적이고 일관된 방식으로 서비스를 찾고 생성할 수 있는 단일 위치를 제공합니다.

프런트엔드

마이크로서비스는 결합도 감소를 지향하며, 이상적으로는 사용자 인터페이스뿐만 아니라 백엔드 부분까지 포함해야 합니다. 실제로 마이크로서비스의 초기 구상에는 DDD(도메인 주도 설계)의 경계 컨텍스트 원칙에 따라 UI를 경계 컨텍스트의 일부로 포함시키는 것이었습니다.
하지만 웹 애플리케이션에 필요한 파티셔닝의 현실적인 제약(및 기타 외부 제약 조건) 때문에 이러한 목표를 달성하기는 어렵습니다. 이것이 바로 마이크로서비스 아키텍처에서 흔히 볼 수 있는 두 가지 UI 스타일입니다.

첫 번째 스타일은 그림 18-5 에 나타낸 것처럼 단일 UI를 통해 API 계층을 호출하여 사용자 요청을 처리하는 모놀리식 프런트엔드입니다. 이 프런트엔드는 풍부한 기능을 갖춘 데스크톱, 모바일 또는 웹 애플리케이션일 수 있습니다. 예를 들어, 현재 많은 웹 애플리케이션은 단일 UI를 구축하기 위해 JavaScript 웹 프레임워크를 사용합니다.

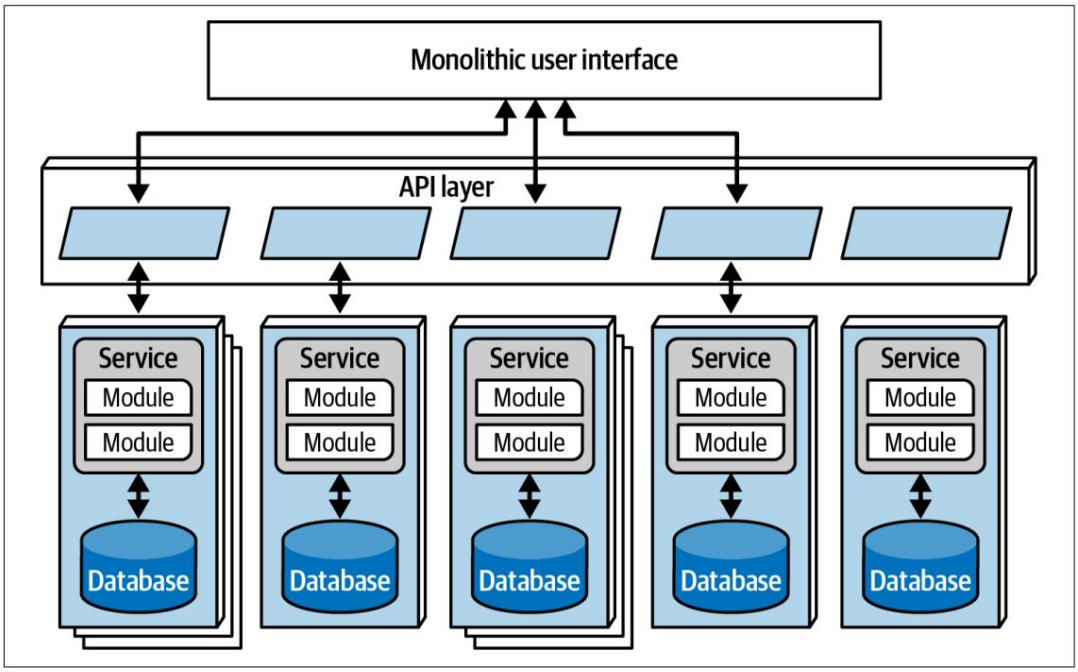


그림 18-5. 단일체 사용자 인터페이스를 갖춘 마이크로서비스 아키텍처

두 번째 UI 옵션은 그림 18-6 에 나와 있는 마이크로 프런트엔드입니다 .

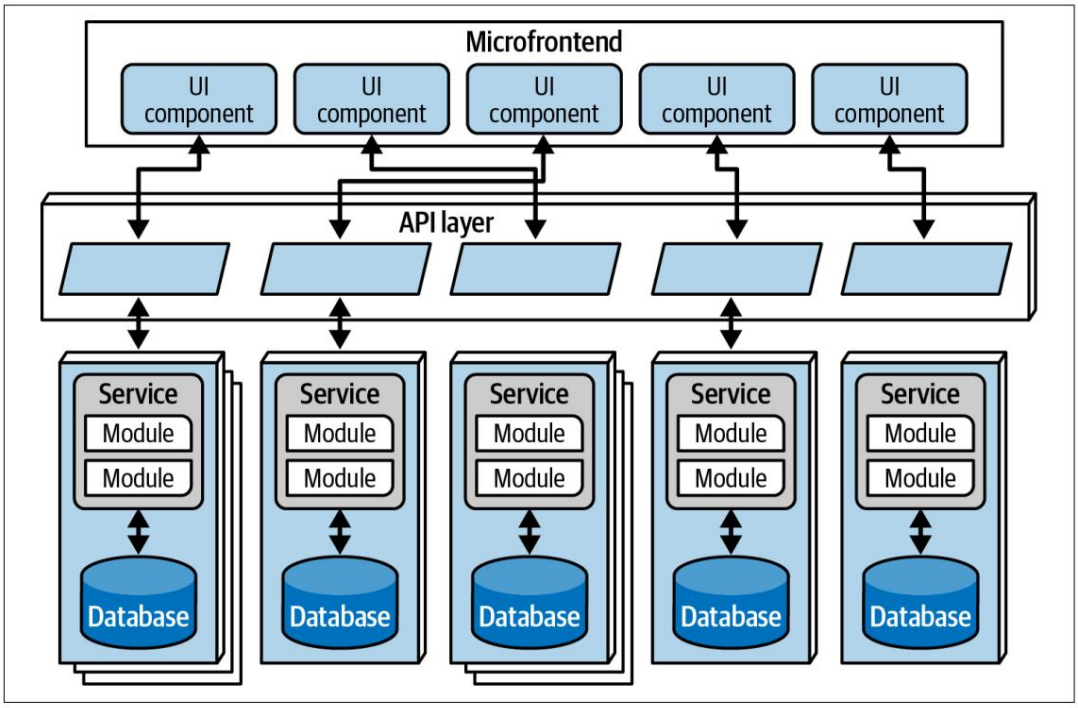


그림 18-6. 마이크로서비스의 마이크로 프런트엔드 패턴

마이크로 프런트엔드 접근 방식은 UI 수준의 컴포넌트를 사용하여 UI와 백엔드 서비스 간에 동기적인 수준의 세분성과 격리를 구현함으로써 UI 컴포넌트와 해당 백엔드 서비스 간의 관계를 형성합니다.

마이크로프런트엔드에 대해 더 자세히 알아보려면 루카 메잘리라의 저서 " 마이크로프런트엔드 구축 (Building Micro-Frontends)" 2판(O'Reilly, 2025)을 적극 추천합니다.

의사소통

마이크로서비스 환경에서 아키텍트와 개발자는 적절한 서비스 세분화 수준을 찾는 데 어려움을 겪는데, 이는 데이터 격리와 통신 모두에 영향을 미칩니다. 올바른 통신 방식을 찾는 것은 팀이 서비스 간의 결합도를 낮추면서도 효과적으로 협업할 수 있도록 도와줍니다.

근본적으로 아키텍트는 동기식 통신과 비동기식 통신 중 어떤 방식을 선택할지 결정해야 합니다. 동기식 통신은 송신자가 수신자의 응답을 기다려야 합니다. 마이크로서비스 아키텍처는 일반적으로 서비스 간 통신 시 프로토콜을 인식하는 이기종 상호 운용성을 활용합니다. 이 복잡한 용어를 구성 요소별로 나누어 그 의미와 중요성을 더 잘 이해해 보겠습니다.

프로토콜 인식

마이크로서비스는 중앙 집중식 통합 허브를 포함하지 않기 때문에 각 서비스는 다른 서비스를 호출하는 방법을 알아야 합니다. 따라서 아키텍트는 일반적으로 특정 서비스들이 서로 호출하는 방식을 표준화합니다. 예를 들어 특정 수준의 REST API, 메시지 큐 등을 사용하는 것입니다. 즉, 서비스는 다른 서비스를 호출할 때 어떤 프로토콜을 사용해야 하는지 알고 있거나 스스로 찾아내야 합니다.

이질적이라는 것은

마이크로서비스가 분산 아키텍처이기 때문에 각 서비스가 서로 다른 기술 스택으로 작성될 수 있다는 것을 의미합니다. 즉, 마이크로서비스는 서로 다른 서비스가 서로 다른 플랫폼을 사용하는 다중 언어 환경을 완벽하게 지원합니다.

상호 운용성은 서비

스들이 서로를 호출하는 것을 의미합니다. 마이크로서비스 아키텍처 설계자들은 트랜잭션 방식의 메서드 호출을 지양하려고 하지만, 서비스들은 일반적으로 네트워크를 통해 다른 서비스를 호출하여 협업하고 정보를 교환합니다.

강제된 이질성

마이크로서비스 스타일의 선구자로 잘 알려진 한 아키텍트가 모바일 기기용 개인 정보 관리 소프트웨어를 개발하는 스타트업의 수석 아키텍트로 재직 중이었습니다. 모바일 시장은 변화 속도가 매우 빠르기 때문에, 아키텍트는 개발팀들이 독립적으로 개발하는 데 방해가 될 수 있는 결합 지점을 의도치 않게 만들지 않도록 하고 싶었습니다. 팀들의 기술 수준이 매우 다양했기 때문에, 아키텍트는 새로운 규칙을 도입했습니다. 바로 각 개발팀이 서로 다른 기술 스택을 사용해야 한다는 것이었습니다. 한 팀이 Java를 사용하고 다른 팀이 .NET을 사용한다면, 두 팀이 의도치 않게 클래스를 공유할 가능성이 전혀 없게 된 것입니다!

이러한 접근 방식은 단일 기술 스택으로 표준화할 것을 고집하는 대부분의 기업 거버넌스 정책과는 정반대입니다. 마이크로서비스 환경에서 목표는 가능한 한 가장 복잡한 생태계를 구축하는 것이 아니라, 문제의 범위에 맞는 적절한 규모의 기술을 선택하는 것입니다. 모든 서비스에 강력한 관계형 데이터베이스가 필요한 것은 아니며, 소규모 팀에 그러한 데이터베이스를 강요하는 것은 오히려 속도를 늦추고 이점을 가져다주지 못할 가능성이 큽니다. 이 개념은 마이크로서비스의 고도로 분산된 특성을 활용합니다.

비동기 통신을 위해 아키텍트는 15장 에서 설명한 이벤트 기반 아키텍처와 유사하게 이벤트와 메시지를 사용하는 경우가 많습니다 .

안무 및 오케스트레이션 안무는 EDA와
동일한 통신 방식을 사용합니다. 안무형 아키텍처는 중앙 조정자가 없어 경계 컨텍스트 철학을 존중하고 서비스 간에 분리된 이벤트를 자연스럽게 구현할 수 있습니다.

안무(choreography)에서는 각 서비스가 중앙 중재자 없이 필요에 따라 다른 서비스를 호출합니다. 예를 들어 그림 18-7에 표시된 시나리오를 생각해 보겠습니다. 사용자가 다른 사용자의 위시리스트에 대한 세부 정보를 요청합니다. CustomerWishList 서비스는 필요한 모든 정보를 가지고 있지 않으므로, CustomerDemographics 서비스 를 호출하여 누락된 정보를 가져온 다음 결과를 사용자에게 반환합니다.

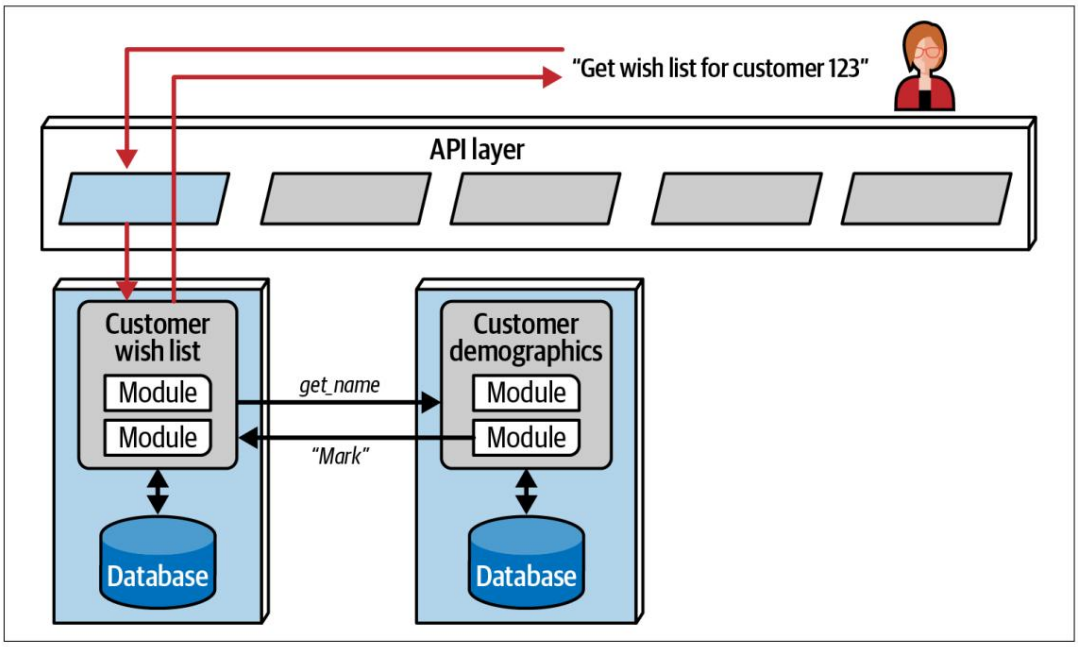


그림 18-7. 마이크로서비스에서 조정 관리를 위한 안무 활용

마이크로서비스 아키텍처는 다른 서비스 지향 아키텍처처럼 전역적인 중재자를 포함하지 않기 때문에, 아키텍트가 여러 서비스 간의 조정을 해야 하는 경우 자체적인 지역화된 중재자(일반적으로 오케스트레이션 서비스라고 함)를 만들 수 있습니다.

그림 18-8 에서 개발자는 호출을 조정하는 것이 유일한 책임인 서비스를 생성합니다. 예를 들어, 사용자는 ReportCustomerInformation 미들웨어를 호출하고, 이 미들웨어는 요청된 정보를 얻기 위해 필요한 다른 모든 서비스를 호출합니다.

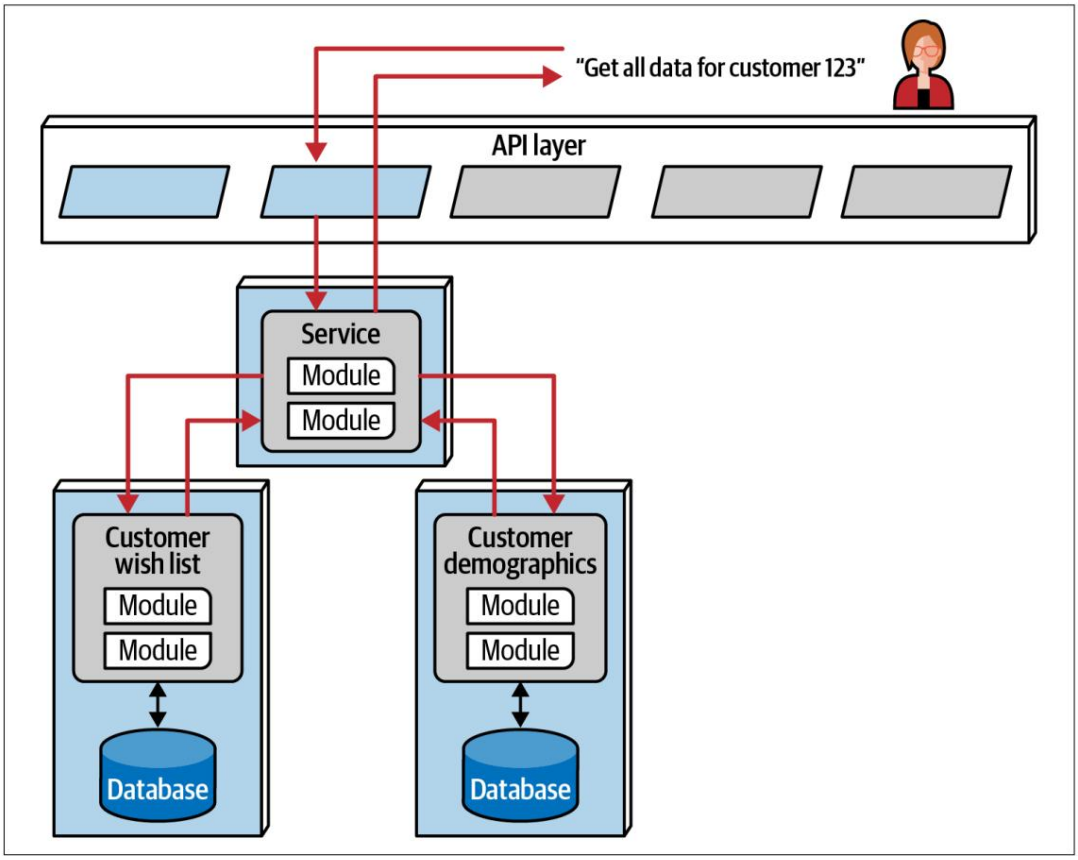


그림 18-8. 마이크로서비스에서 오케스트레이션 사용

소프트웨어 아키텍처의 제1법칙은 이 두 가지 해결책 모두 완벽하지 않으며 각각 장단점이 있음을 시사합니다. 코레오그래피는 마이크로서비스의 고도로 분리된 철학을 유지하여 그 장점을 극대화합니다. 하지만 오류 처리 및 조정과 같은 일반적인 문제를 더 복잡하게 만들기도 합니다.

좀 더 복잡한 워크플로우를 예로 들어 보겠습니다. 그림 18-9 에서 처음 호출되는 서비스는 다양한 다른 서비스들을 조율해야 하며, 기본적으로 다른 도메인 책임 외에도 중재자 역할을 수행해야 합니다. 이를 프런트 컨트롤러 패턴이라고 하는데, 명목상 안무를 담당하는 서비스가 특정 문제에 대한 복잡한 중재자 역할을 하게 되는 패턴입니다. 이 패턴의 단점은 서비스가 여러 역할을 맡게 되면서 복잡성이 증가한다는 것입니다.

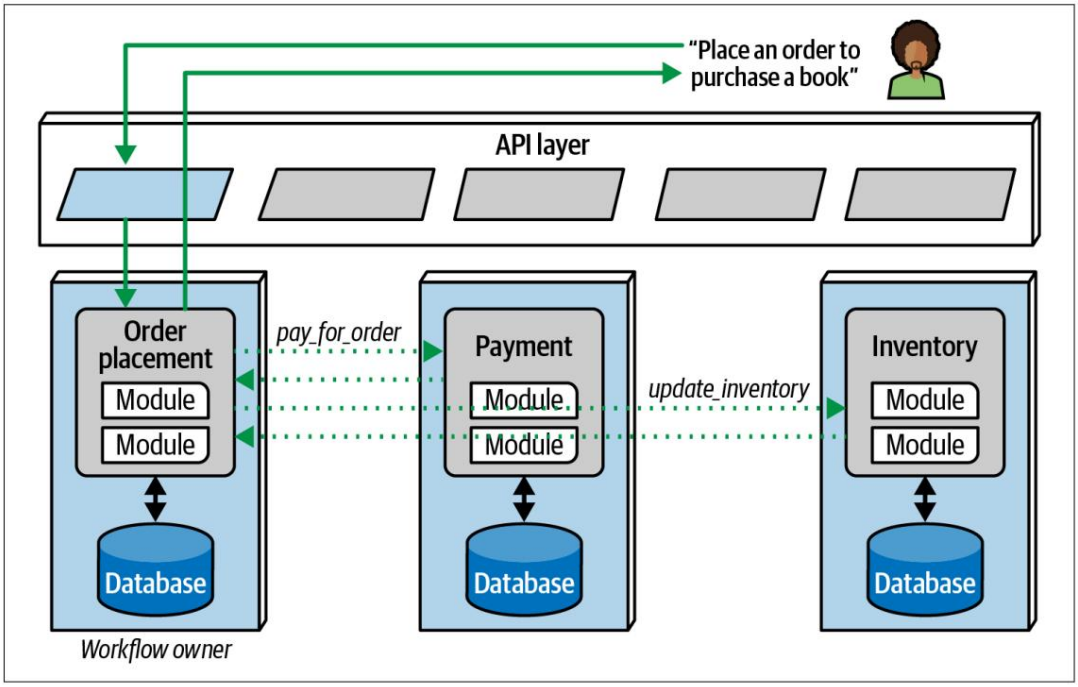


그림 18-9. 복잡한 비즈니스 프로세스에 안무 활용

또는 아키텍트는 그림 18-10 에 나타낸 것처럼 복잡한 비즈니스 프로세스를 위해 오케스트레이션을 사용할 수도 있습니다 . 아키텍트는 비즈니스 워크플로를 조정하는 미들웨어 서비스를 구축하여 서비스 간의 결합을 생성하지만, 동시에 조정 작업을 단일 서비스에 집중시켜 다른 서비스에 미치는 영향을 최소화할 수 있습니다. 도메인 워크플로는 본질적으로 결합되어 있는 경우가 많으므로, 아키텍트의 역할은 도메인과 아키텍처의 목표를 모두 가장 잘 지원하는 방식으로 이러한 결합을 표현하는 방법을 찾는 것입니다.

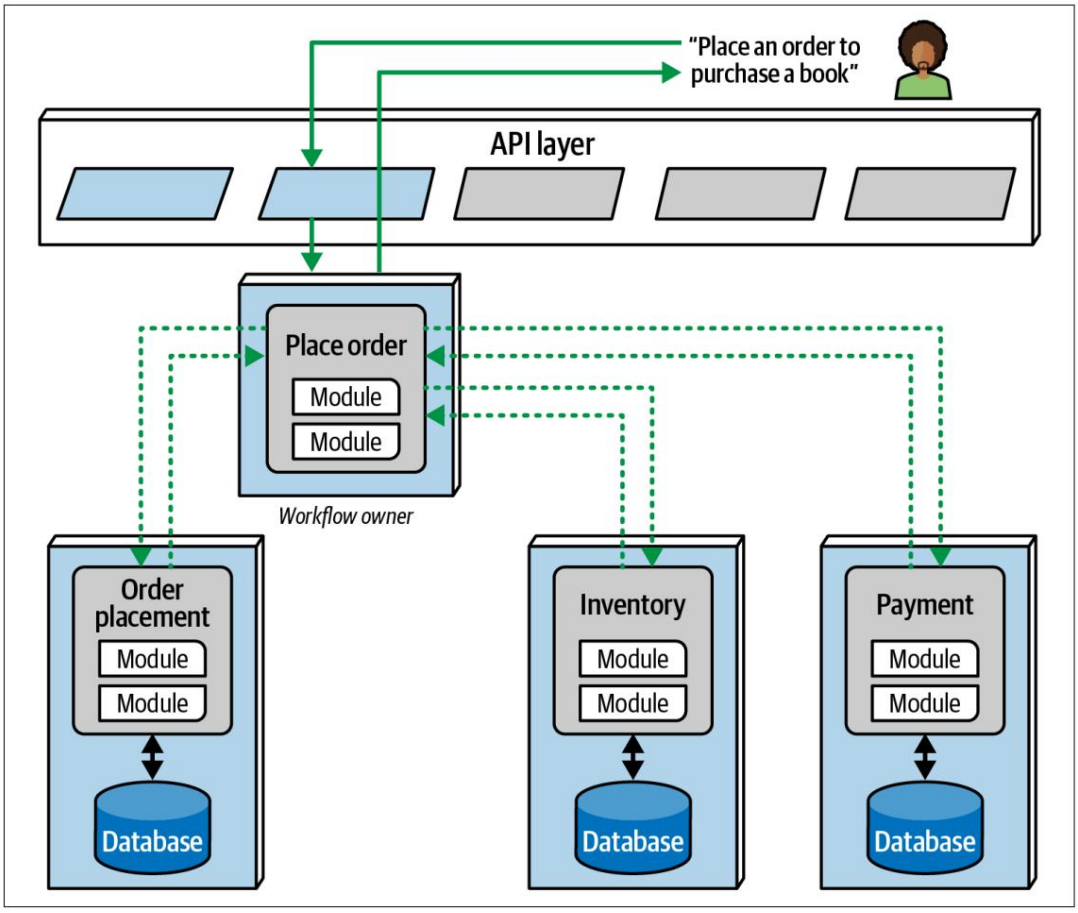


그림 18-10. 복잡한 비즈니스 프로세스에 오케스트레이션을 사용하는 방법

스에서 극단적인 결합도 감소를 추구하지만, 서비스 간 트랜잭션을 어떻게 조정할 것인지에 대한 문제에 종종 직면합니다. 마이크로서비스는 아키텍처에서와 마찬가지로 데이터베이스에서도 동일한 수준의 결합도 감소를 요구하기 때문에, 모놀리식 애플리케이션에서는 당연했던 원자성이 분산 애플리케이션에서는 문제가 됩니다.

서비스 경계를 넘나드는 트랜잭션 구축은 마이크로서비스의 핵심 원칙인 결합 해제를 위반할 뿐만 아니라, 최악의 동적 연결성인 값의 연결성(Connascence of Values, 49 페이지의 "연결성" 참조)을 초래합니다. 서비스 간 트랜잭션을 구현하려는 아키텍트에게 드릴 수 있는 최고의 조언은 "절대 하지 마세요!"입니다. 대신 서비스 세분화를 개선하세요. 마이크로서비스 아키텍처를 트랜잭션으로 연결해야 한다는 사실은 설계가 지나치게 세분화되었다는 신호입니다.

여러 마이크로서비스에 걸쳐 있는 트랜잭션은 피하고, 대신 서비스 세분화를 개선하세요!

"상황에 따라 다르다"는 원칙에 따라 예외는 항상 존재합니다. 예를 들어, 서로 다른 두 서비스가 완전히 다른 아키텍처 특성을 필요로 하고, 명확한 서비스 경계를 설정해야 하지만, 트랜잭션 조정은 여전히 필요한 상황이 발생할 수 있습니다. 이러한 경우, 아키텍트는 장단점을 신중하게 고려한 후 특정 트랜잭션 패턴을 활용하여 트랜잭션을 오케스트레이션할 수 있습니다.

마이크로서비스에서 분산 트랜잭션은 일반적으로 사가 패턴(Saga pattern)이라는 것을 사용하여 처리됩니다. 문학에서 사가는 일련의 긴 사건들을 묘사하여 영웅적인 결말에 이르는 서사시를 의미하며, 이러한 의미에서 이 트랜잭션 패턴의 이름이 유래되었습니다.

그림 18-11 에서 서비스는 여러 서비스 호출 간의 중개자 역할을 하여 트랜잭션을 조정합니다. 중개자는 트랜잭션의 각 부분을 호출하고 성공 또는 실패 여부를 기록하며 결과를 조정합니다. 모든 것이 계획대로 진행되면 서비스와 해당 서비스에 포함된 데이터베이스의 모든 값이 동기적으로 업데이트됩니다. 오류가 발생하여 트랜잭션의 한 부분이 실패하면 중개자는 트랜잭션의 어떤 부분도 성공하지 못하도록 해야 합니다. 그림 18-12 에 나타난 상황을 생각해 보십시오.

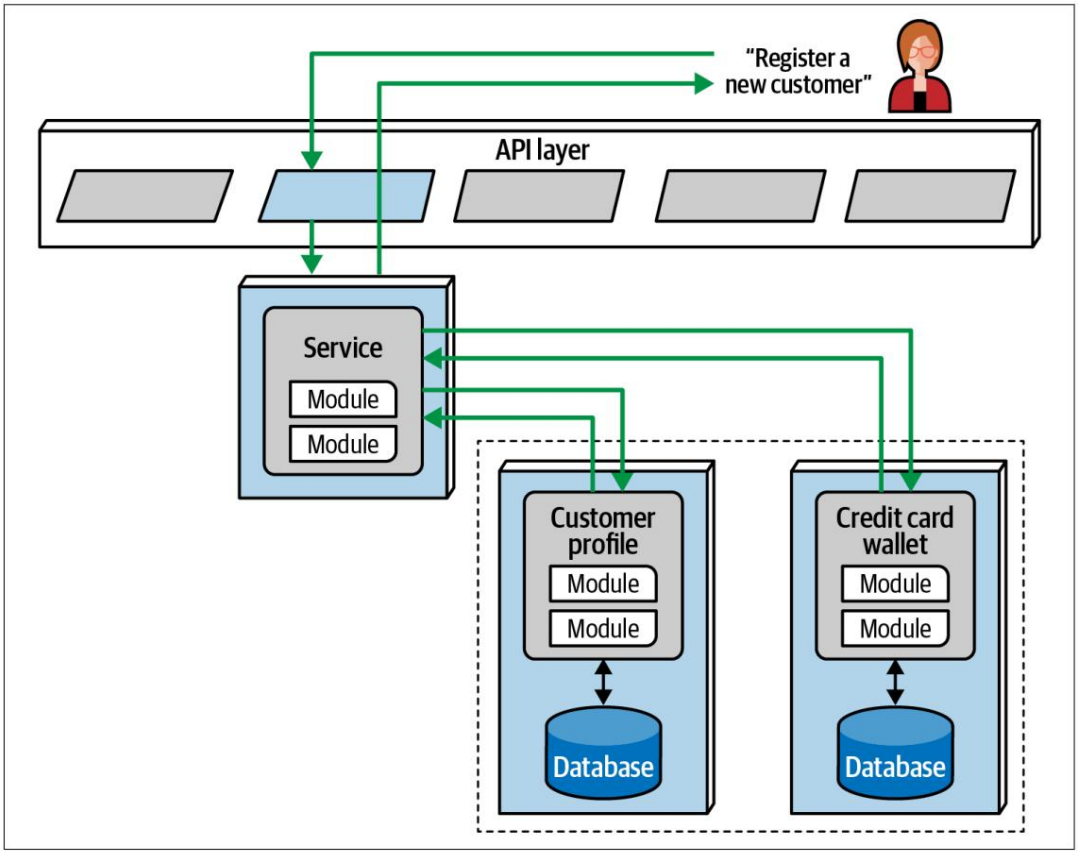


그림 18-11. 마이크로서비스 아키텍처에서의 사가 패턴

트랜잭션의 첫 번째 부분이 성공했지만 두 번째 부분이 실패한 경우, 중개자는 트랜잭션에 참여한 다른 모든 서비스(성공한 서비스)에 이전 요청을 취소하도록 요청해야 합니다. 이러한 트랜잭션 조정 방식을 보상 트랜잭션 프레임워크라고 합니다. 개발자는 일반적으로 중개자가 전체 성공을 나타낼 때까지 각 요청을 대기 상태로 유지함으로써 이 패턴을 구현합니다. 그러나 특히 대기 중인 트랜잭션 상태에 따라 달라지는 새로운 요청이 발생할 경우 비동기 요청을 처리하는 것은 복잡해질 수 있습니다. 사용되는 프로토콜과 관계없이 보상 트랜잭션은 네트워크 수준에서 많은 조정 트래픽을 발생시킵니다.

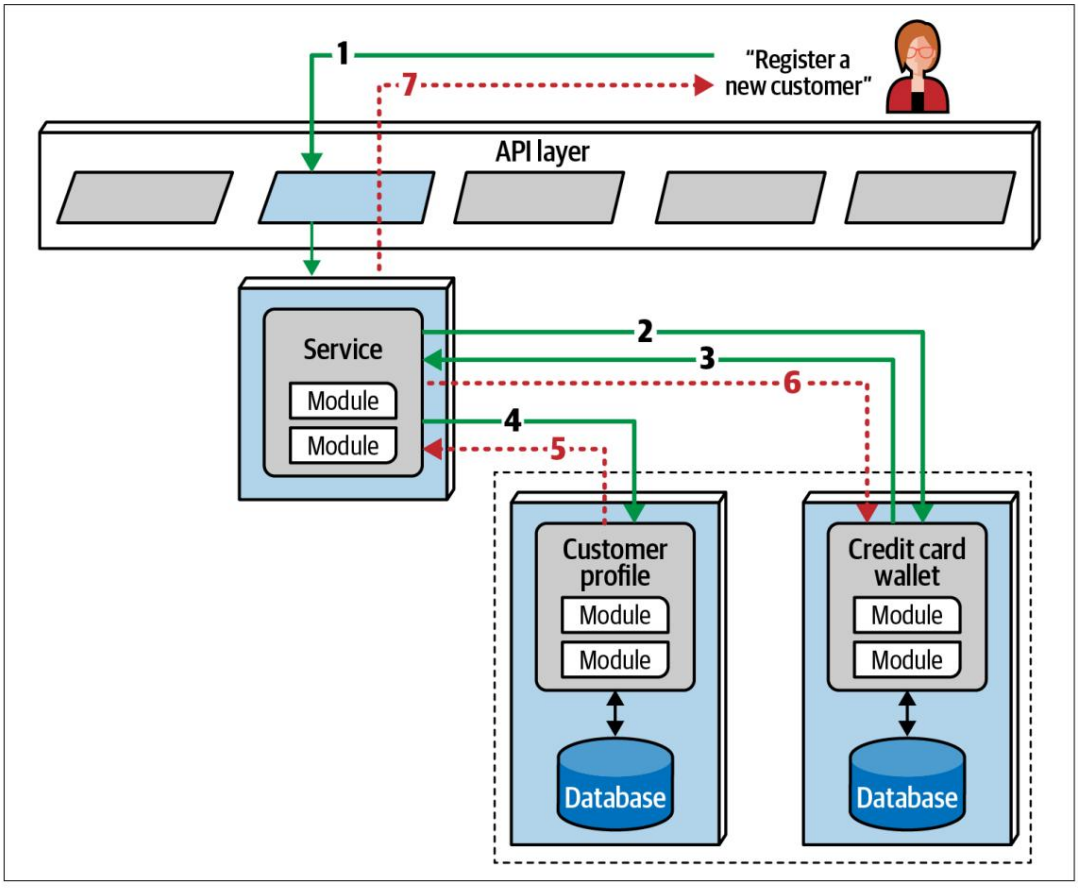


그림 18-12. 오류 조건에 대한 트랜잭션을 보상하는 사가 패턴

때로는 몇몇 트랜잭션이 여러 서비스를 거쳐야 할 필요가 있지만, 만약 그것이 아키텍처의 주된 특징이라면 마이크로서비스는 적합한 선택이 아닐 가능성이 높습니다!

마이크로서비스에서 트랜잭션을 관리하는 것은 복잡하며, 더 깊이 파고들 수도 있습니다. 저희는 다양한 시나리오를 해결하기 위해 8가지의 서로 다른 트랜잭션 사가 패턴을 개발했으며, 이는 Pramod Sadalage, Zhamak Dehghani와 공저한 책 『Soware Architecture: e Hard Parts』 (O'Reilly, 2021)의 12장에서 확인할 수 있습니다.

데이터 토폴로지

이 장에서 논의했듯이 데이터는 마이크로서비스 아키텍처에서 매우 중요한 역할을 합니다. 사실 마이크로서비스는 아키텍트가 데이터를 분리해야 하는 유일한 아키텍처 스타일입니다. 다른 분산 아키텍처에서 모놀리식 데이터베이스를 사용하는 것이 가능하지만(항상 효과적인 것은 아니지만), 마이크로서비스에서는 선택 사항이 아닙니다. 215페이지의 "데이터 토폴로지" 와 268페이지의 "데이터 토폴로지" 에서 논의했듯이 도메인 데이터베이스도 마찬가지입니다 . 이는 마이크로서비스의 세분화된 특성, 경계 컨텍스트, 그리고 대부분의 마이크로서비스 생태계에서 발견되는 수많은 서비스 때문입니다.

마이크로서비스 환경에서 단일형 데이터베이스가 적합하지 않은 이유를 설명하기 위해 60개의 서비스가 동일한 데이터베이스를 공유하는 시나리오를 생각해 보겠습니다. 가장 먼저 발생하는 문제는 아키텍처 내 변경 사항을 관리하는 것입니다. 그림 18-13 에서 볼 수 있듯이, 데이터베이스 구조를 변경(예: 열 이름 변경 또는 테이블 삭제)하면 해당 데이터를 사용하는 60개의 서비스 모두에 상응하는 변경이 필요합니다. 데이터베이스 변경 사항을 배포하는 동시에 개별적으로 배포된 60개의 서비스에 대한 유지 관리, 테스트 및 릴리스를 조정해야 한다고 상상해 보세요!

이 임무는 말 그대로 벅찬 일이 될 것이며, 재앙으로 끝날 가능성이 매우 높습니다.

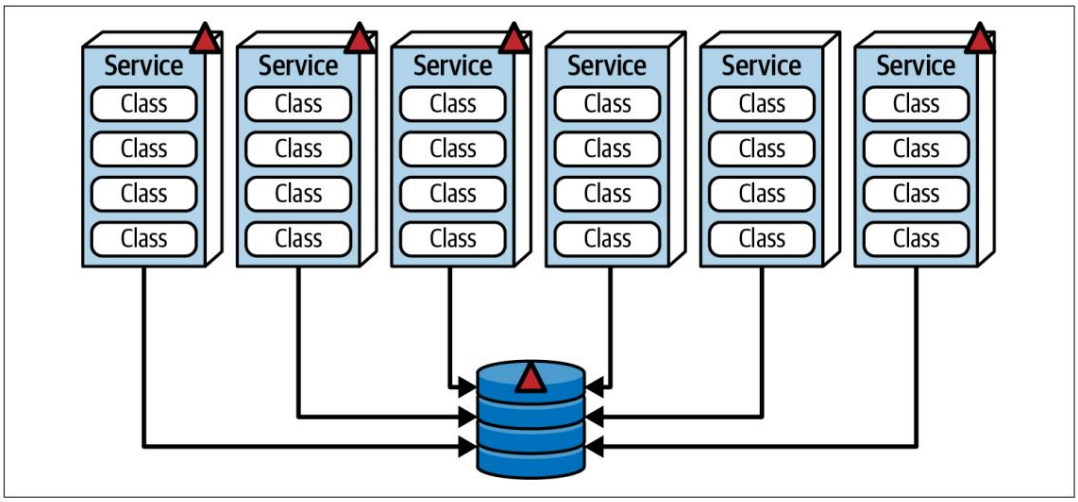


그림 18-13. 단일 데이터베이스를 이용한 변경 관리는 매우 어려운 과제이다.

단일 데이터 토폴로지와 마이크로서비스를 결합할 때 가장 큰 문제는 물리적 경계 컨텍스트라는 개념 자체가 무너진다는 점일 것입니다. 앞서 언급했듯이,

경계 컨텍스트는 특정 비즈니스 기능 또는 하위 도메인을 실행하는 데 필요한 모든 기능(데이터베이스 및 해당 데이터 구조 포함)을 포괄합니다. 모든 서비스가 동일한 데이터베이스와 데이터 구조를 공유하는 경우 경계 컨텍스트는 더 이상 필요하지 않습니다.

단일형 데이터의 또 다른 문제는 확장성과 탄력성입니다. 많은 운영 도구와 제품이 동시 부하를 자동으로 모니터링하고 증가에 맞춰 서비스 인스턴스 수를 조정하지만, 그에 맞춰 확장되는 데이터베이스는 많지 않습니다. 이러한 불균형은 전반적인 응답성 저하와 요청 시간 초과를 초래할 수 있습니다. 일반적으로 각 서비스 인스턴스에 위치하는 데이터베이스 연결 관리 또한 문제입니다. 서비스 및 서비스 인스턴스 수가 증가함에 따라 서비스에서 사용 가능한 데이터베이스 연결이 빠르게 부족해져 연결 대기 및 요청 시간 초과가 발생할 수 있습니다.

마지막으로, 데이터베이스가 충돌, 계획된 유지 보수 또는 백업으로 인해 사용할 수 없게 되면 전체 마이크로서비스 생태계가 다운됩니다. 이는 모놀리식 데이터베이스를 사용하는 모든 아키텍처 방식과 마찬가지입니다. 극단적인 경우는 아니지만, 도메인 기반 데이터베이스 토폴로지도 확장성, 데이터베이스 연결 풀 관리, 가용성 및 내결함성 측면에서 동일한 문제점을 겪을 수 있습니다.

이러한 이유로 마이크로서비스의 표준 데이터베이스 토폴로지는 서비스별 데이터베이스(Database-per-Service) 패턴입니다. 이 데이터베이스 토폴로지에서는 각 마이크로서비스가 자체 데이터를 소유하며, 이 데이터는 그림 18-14에서 보는 것처럼 별도의 데이터베이스 또는 스키마 내의 테이블 형태로 저장됩니다.

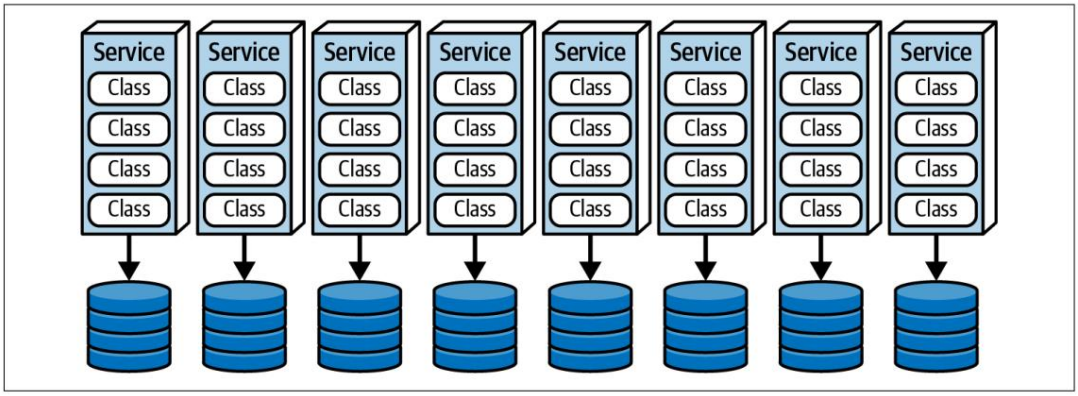


그림 18-14. e 서비스별 데이터베이스 패턴은 마이크로서비스에 사용되는 일반적인 데이터베이스 토폴로지입니다.

이 토폴로지는 경계 컨텍스트를 유지하여 아키텍트가 변경 사항을 더 쉽게 제어할 수 있도록 합니다. 데이터가 필요한 다른 서비스는 일종의 계약을 통해 소유 서비스에 데이터를 요청해야 하므로 데이터의 내부 구조와 분리됩니다. 데이터베이스 구조를 변경해도 경계 컨텍스트 내의 소유 서비스에만 영향을 미칩니다. 따라서 아키텍트는 데이터베이스 유형을 자유롭게 변경할 수 있습니다.

(관계형 데이터베이스에서 문서 데이터베이스로의 변환과 같이) 다른 데이터베이스에 영향을 주지 않고
서비스.

서비스별 데이터베이스 토폴로지는 확장성, 탄력성, 가용성 및 내결함성 측면에서 탁월한 성능을 제공하는데, 이는 단일체 또는 도메인 기반 데이터베이스 토폴로지에서 부족한 부분입니다. 또한, 경계 컨텍스트 내에서 데이터베이스 연결을 관리하는 것이 단일체 또는 도메인 기반 데이터베이스보다 훨씬 쉽습니다.

마이크로서비스 환경에서 서비스별 데이터베이스(Database-per-Service) 토폴로지가 널리 사용되지만, 몇 가지 단점이 있습니다. 예를 들어, 두 개 이상의 서비스가 동일한 데이터베이스 테이블에 쓰기 작업을 수행하는 경우는 어떻게 될까요? 또는 성능상의 이유로 경계 컨텍스트 외부의 서비스가 데이터베이스를 직접 쿼리해야 하는 경우는 어떻게 될까요? 이러한 경우(꽤 흔한 경우입니다) 그림 18-15에서처럼 여러 서비스가 하나의 데이터베이스를 공유할 수 있습니다. 하지만 하나의 데이터베이스(또는 스키마)를 공유하는 서비스는 5~6개를 넘지 않도록 하는 것이 좋습니다. 그 이상이 되면 변경 관리, 확장성, 탄력성, 가용성, 내결함성 등에서 동일한 문제가 발생하기 시작합니다.

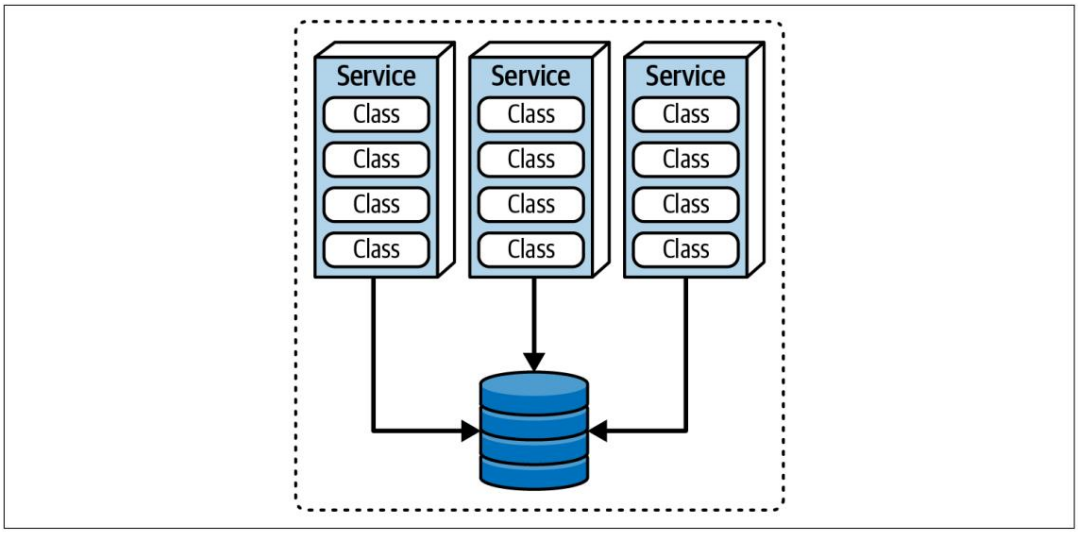


그림 18-15. 몇몇 마이크로서비스 간에 데이터를 공유하는 것이 가능합니다.

그림 18-15 에서 서비스와 데이터베이스를 둘러싼 상자는 경계 컨텍스트를 나타냅니다. 서비스들이 데이터베이스를 공유한다고 해서 경계 컨텍스트가 없는 것은 아닙니다. 단지 아키텍트가 더 넓은 범위의 경계 컨텍스트를 설정했다는 의미입니다. 예를 들어, 결제 처리를 신용카드, 상품권, 페이팔, 포인트 적립 등 다양한 결제 유형으로 분리해야 하는 타당한 이유가 있을 수 있지만, 이러한 개별 서비스들은 여전히 동일한 데이터를 업데이트하고 접근해야 합니다.

마찬가지로, 아키텍트는 단일 배송 서비스를 각 배송 방식별로 별도로 배포되는 서비스로 분할할 수 있지만, 이러한 서비스들은 모두 동일한 데이터에 대한 업데이트와 접근 권한을 필요로 합니다.

더 넓은 경계 컨텍스트 내에서 마이크로서비스 간에 데이터를 공유할 때 가장 큰 단점은 데이터베이스 변경을 제어하는 것입니다. 데이터베이스 스키마가 변경될 경우, 아키텍트는 여러 서비스의 변경 및 배포를 조율해야 하므로 데이터베이스 변경이 더 위험해지고 민첩성이 떨어집니다. 또한 비즈니스 문제와 상황에 따라 확장성, 탄력성 및 내결함성에 부정적인 영향을 미칠 수도 있습니다.

클라우드 고려 사항

물론 마이크로서비스는 온프레미스 시스템에도 배포할 수 있으며, 특히 Kubernetes 및 Cloud Foundry 와 같은 서비스 오케스트레이션 플랫폼의 보급이 활발해짐에 따라 이러한 아키텍처 스타일은 클라우드 기반 배포에 매우 적합합니다. 따라서 마이크로서비스는 때때로 "클라우드 네이티브" 아키텍처라고 불리기도 합니다.

클라우드 환경(예: AWS)에서 볼 수 있는 서비스 기반 접근 방식과 결합된 가상 머신, 컨테이너 및 데이터베이스의 온디맨드 프로비저닝은 마이크로서비스 아키텍처 스타일에 잘 어울립니다.

주의 깊은 독자라면 왜 서버리스를 아키텍처 스타일로 포함시키지 않았는지 궁금해하실 수도 있습니다 . 서버리스는 요청 시 함수가 실행되고 필요한 시스템 리소스가 온디맨드 방식으로 할당되는 클라우드 컴퓨팅 모델을 의미합니다. AWS Lambda, Google Cloud Functions, Azure Cloud Functions 등 이 서버리스 함수의 예입니다 . 하지만 저희는 서버리스가 아키텍처 스타일이 아니라 마이크로서비스 아키텍처 스타일의 배포 모델이라고 생각합니다.

이 장의 앞부분에서 설명했듯이 마이크로서비스는 하나의 목적을 가진, 독립적으로 배포되는 소프트웨어 단위로, 한 가지 기능을 매우 효율적으로 수행합니다(그래서 '마이크로'라는 접두사가 붙습니다). 따라서 마이크로서비스는 다른 아키텍처 스타일의 서비스보다 상대적으로 세분화된 구조를 갖는 경향이 있습니다. 이는 본질적으로 서버리스 기능을 설명하는 것이기도 하며, 바로 이러한 이유로 서버리스를 마이크로서비스의 일부로 간주합니다.

하지만 클라우드 환경에서 마이크로서비스를 반드시 서버리스 함수로 배포할 필요는 없습니다. 컨테이너화된 서비스로도 충분히 쉽게 배포할 수 있습니다. 대부분의 클라우드 공급업체는 쿠버네티스(또는 쿠버네티스 플랫폼의 변형)를 채택하고 있어 개발자는 서버리스 서비스처럼 컨테이너화된 마이크로서비스를 간편하게 배포할 수 있습니다.

일반적인 위험

마이크로서비스에서 가장 큰 위험 중 하나는 서비스를 너무 작게 만드는 것입니다. 2016년에 저자 중 한 명(마크)은 해변의 모래알갱이처럼 서비스를 너무 세분화하는 안티패턴을 '모래알갱이(Grains of Sand)'라고 명명했습니다. 앞서 언급했듯이 마이크로서비스에서 '마이크로'는 서비스의 크기가 아니라 서비스의 기능을 의미합니다. 서비스의 세분화는 마이크로서비스에서 매우 중요한 측면이어서 저희 책 ' 소프트웨어 아키텍처: 하드 파츠(Soware Architecture: Hard Parts)'에서 이 주제를 7장에서 자세히 다루고 있습니다.

마이크로서비스에서 흔히 발생하는 또 다른 위험은 서비스 간 과도한 통신입니다. 마이크로서비스의 세분화된 특성과 엄격한 경계 컨텍스트로 인해 마이크로서비스 생태계 내의 서비스들은 필연적으로 서로 통신해야 합니다. 이러한 통신은 워크플로 처리(예: 서버리스 마이크로서비스의 경우 안무 또는 AWS 스텝 함수) 또는 다른 서비스의 데이터 접근(각 서비스와 데이터 내의 경계 컨텍스트로 인해) 때문일 수 있습니다. 이유가 무엇이든, 과도한 동적 결합과 서비스 간 통신을 피하도록 주의해야 합니다. 이 문제는 종종 서비스를 지나치게 세분화한 결과이며, 서비스를 통합하여 더 큰 규모의 마이크로서비스로 만들면 해결할 수 있습니다.

마이크로서비스의 또 다른 위험은 데이터 공유를 지나치게 하는 것입니다. 마이크로서비스에서 데이터 공유가 가능하고 때로는 필요하다는 점을 언급했지만, 과도한 데이터 공유는 시스템의 변경 관리, 확장성, 내결함성, 그리고 전반적인 민첩성에 위험을 초래합니다. 이러한 요소들은 마이크로서비스가 강점을 보이는 부분입니다(354페이지의 "스타일 특징" 참조). 데이터 공유가 필요한 시점과 서비스 통합을 통해 데이터 공유 문제를 해결해야 하는 시점을 명확히 구분해야 합니다.

마이크로서비스 아키텍처 스타일에서 흔히 간과되는 마지막 위험은 코드 재사용과 기능 공유입니다. 코드 재사용은 소프트웨어 개발에 필수적인 부분입니다. 그러나 코드와 기능의 재사용은 마이크로서비스의 원칙에 정면으로 위배됩니다. 이 때문에 이 장 앞부분에서 "아무것도 공유하지 않는" 아키텍처라는 용어가 사용된 것입니다. 아키텍트가 사용자 정의 라이브러리(JAR 파일이나 DLL 등)를 통해 서비스 간에 공통 기능을 공유하는 경우, 경계 컨텍스트의 일부가 무너지게 됩니다.

다시 말해, 재사용되는 코드는 여러 경계 컨텍스트에 분산되어 있으므로 특정 함수 또는 하위 도메인의 모든 기능이 해당 경계 컨텍스트 내에 포함되어 있지 않습니다. 따라서 공유 코드에 대한 변경 사항은 다른 경계 컨텍스트의 서비스를 손상시킬 수 있습니다. 버전 관리가 이러한 문제를 해결하는 데 도움이 되지만, 코드 공유는 마이크로서비스 생태계에 상당한 복잡성을 더합니다.

통치

마이크로서비스에서 사용되는 많은 거버넌스 규칙과 기법은 이전 섹션에서 설명한 일반적인 위험을 해결합니다. 마이크로서비스 아키텍처에서의 거버넌스는 주로 구조적 붕괴를 방지하는 데 중점을 둡니다.

무엇보다도 아키텍트는 서비스 간의 정적 및 동적 결합 정도를 모니터링하고 제어하기 위해 거버넌스를 적용해야 합니다. 7장 에서 살펴본 바와 같이 정적 결합은 마이크로서비스가 공통의 사용자 정의 라이브러리 또는 타사 라이브러리를 공유할 때 발생하며, 서비스 간 통신이 필요할 때 계약 형태로도 발생합니다. 계약은 특히 중요합니다. 아키텍트가 비동기 통신 프로토콜을 사용하여 서비스를 동적으로 분리할 수 있지만, 통신 유형과 관계없이 서비스 간에 사용되는 계약으로 인해 여전히 정적으로 결합될 수 있기 때문입니다.

소프트웨어 구성 요소 목록, 배포 스크립트 및 종속성 관리 도구는 아키텍트가 서비스 간에 공유되는 아티팩트의 수를 더 잘 이해하고 관리하는 데 도움이 될 수 있습니다. 정적 결합이 어느 정도까지 허용 가능한 수준인지 정확히 규정할 수는 없지만, 서비스 간의 결합을 최소화하도록 노력하는 것이 좋습니다.

동적 결합을 제어하는 것은 정적 결합을 제어하는 것보다 훨씬 더 어렵습니다.
적절한 메트릭을 수집하려면 창의성과 일관성이 필요합니다. 일반적인 거버넌스 기법 중 하나는 로그를 사용하여 다른 서비스 호출을 식별하는 것입니다. 서비스가 내부 또는 타사 서비스를 호출할 때, 해당 서비스는 호출된 서비스, 사용된 프로토콜 등의 정보와 함께 상호 작용을 로그에 기록합니다. 아키텍트는 적합도 함수를 통해 이 정보를 분석하여 마이크로서비스 생태계 전반의 동적 결합 수준을 더 잘 이해할 수 있습니다. 그러나 이 접근 방식은 각 서비스가 일관된 방식으로 로깅을 통해 이 정보를 노출하도록 철저한 거버넌스가 필요합니다. 모든 서비스가 컴파일 시점에 바인딩하고 일관된 로깅을 보장하는 데 사용할 수 있는 일관된 API를 제공하는 한 가지 방법은 사용자 지정 라이브러리(예: JAR 파일 또는 DLL)를 사용하는 것입니다.

마이크로서비스에서 동적 결합 지표를 수집하는 또 다른 방법은 레지스트리 항목을 활용하는 것입니다. 서비스의 첫 번째 인스턴스가 시작될 때마다 해당 서비스는 특정 계약(예: JSON)을 통해 서비스 간 호출을 Apache ZooKeeper와 같은 사용자 지정 구성 서비스 또는 구성 서버에 등록합니다. 그러면 아키텍트는 구성 서버에 쿼리를 보내 마이크로서비스 생태계 전체의 서비스 간 호출 맵을 얻을 수 있으며, 이를 통해 서비스 간 통신량을 관리하고 제어할 수 있습니다.

팀 토폴로지 고려 사항

마이크로서비스 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 구성 및 토폴로지" 에 설명된 특정 팀 구성에 맞춰 마이크로서비스 아키텍처를 구축하려는 아키텍트를 위한 몇 가지 고려 사항입니다.
151페이지의 "건축" 항목:

스트림 정렬 팀
: 도메인 경계가 제대로 정렬되어 있다면, 특히 특정 분야에 집중된 스트림으로 구성된 팀은 이러한 아키텍처 스타일에서 효과적으로 협업할 수 있습니다.

하지만 특정 하위 도메인이나 도메인 외부에 있는 여러 경계 컨텍스트와 서비스에 걸쳐 작업 스트림이 있는 팀의 경우 마이크로서비스 아키텍처는 더욱 어려워집니다. 이러한 경우 마이크로서비스의 경계 컨텍스트와 세분성을 분석하고 작업 스트림에 맞춰 재조정하거나 다른 아키텍처 스타일을 선택하는 것이 좋습니다.

지원팀은 마이크로
서비스 아키텍처에서 공유 서비스를 활용하여 전문적인 문제나 공통적인 문제를 해결할 수 있을 때 가장 효과적입니다. 마이크로서비스는 모듈화 수준이 높기 때문에 지원팀은 스트림 팀과 독립적으로 작업하여 방해 없이 추가적인 전문 기능과 공유 기능을 제공할 수 있습니다. 또한 플랫폼 팀과 협력하여 서비스 메시를 구성하는 사이드카 컴포넌트를 생성하는 데 도움을 줄 수도 있습니다(334페이지의 "운영 재사용" 참조).

복잡한 하위 시스템 팀
복잡한 하위 시스템 팀은 이러한 아키텍처 스타일의 서비스 수준 모듈성을 활용하여 복잡한 도메인 또는 하위 도메인 처리에 집중하고 다른 팀 구성원(및 서비스)으로부터 독립적으로 운영할 수 있습니다.

플랫폼 팀은 마이
크로서비스에서 발견되는 높은 수준의 모듈성을 활용하여 스트림 중심 팀이 공통 도구, 서비스, API 및 작업을 이용함으로써 플랫폼 팀 토폴로지의 이점을 최대한 활용할 수 있도록 지원합니다. 많은 경우, 플랫폼 팀(때로는 지원 팀과 협력하여)은 사이드카(334페이지의 "운영 재사용" 참조) 와 서비스 메시에서 발견되는 공통 운영 기능을 생성하고 유지 관리하는 데 집중하여 스트림 중심 팀이 이러한 운영 업무에서 벗어날 수 있도록 합니다.

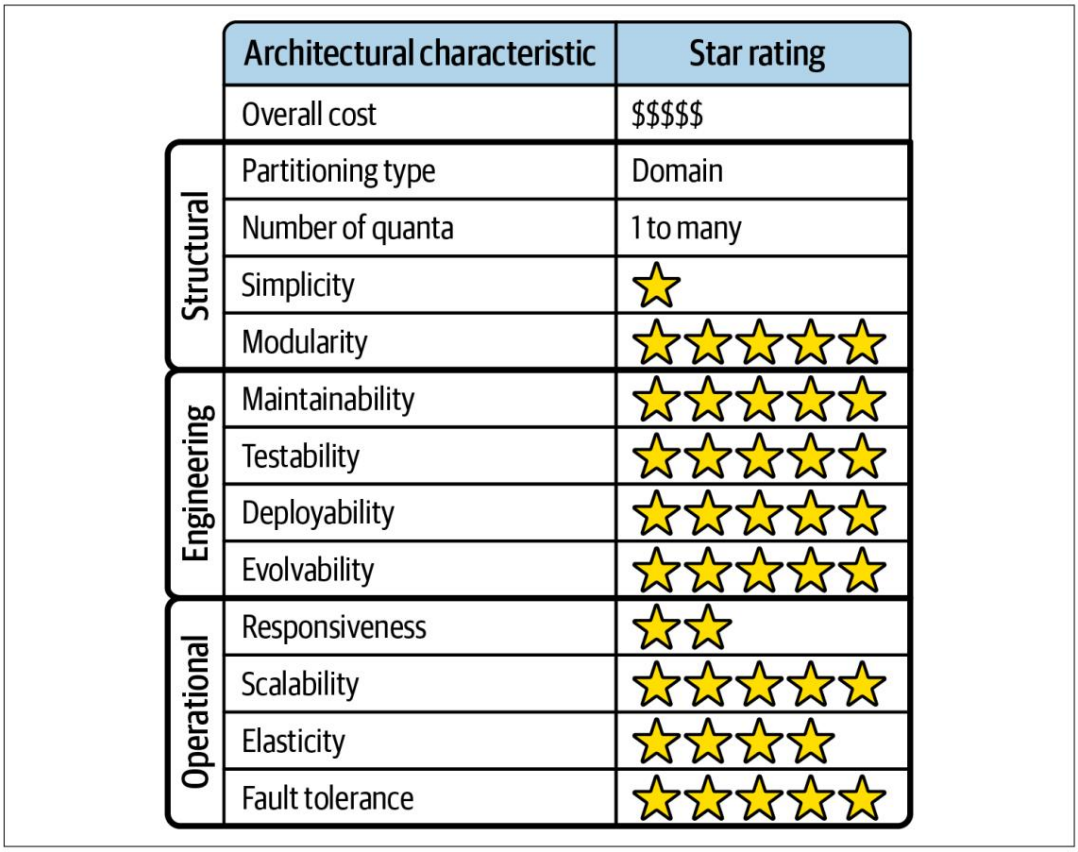
우려 사항.

스타일 특징

마이크로서비스 아키텍처 스타일은 그림 18-16 에 나타낸 표준 평가 척도에서 여러 극단적인 값을 보여줍니다. 별 1개는 특정 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 .

마이크로서비스는 자동화된 배포 및 테스트 용이성과 같은 최신 엔지니어링 관행을 매우 효과적으로 지원합니다. 마이크로서비스는 운영 자동화를 향한 끊임없는 진전을 보여주는 DevOps 혁명 없이는 존재할 수 없었습니다.

우려 사항.



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

그림 18-16. 마이크로서비스 특성 평가

이러한 아키텍처 스타일에서 서비스의 독립적이고 단일 목적적인 특성, 즉 세밀한 구성은 일반적으로 높은 내결함성을 제공하며, 따라서 이 특성에 대해 높은 평가를 받습니다.

이 아키텍처의 또 다른 장점은 확장성, 유연성 및 진화 가능성입니다.
지금까지 개발된 가장 확장성이 뛰어난 시스템 중 상당수는 마이크로서비스를 성공적으로 활용해 왔습니다. 마찬가지로, 이 방식은 자동화와 운영과의 지능적인 통합에 크게 의존하기 때문에 아키텍트는 확장성을 위한 기반을 마련할 수 있습니다. 또한, 점진적인 수준에서 높은 수준의 결합도 유지를 지향하는 이 아키텍처는 아키텍처 수준에서도 진화적 변화를 추구하는 현대 비즈니스 관행을 지원합니다.

현대 기업의 변화 속도는 빠르지만, 소프트웨어 개발은 그 속도를 따라잡기 위해 고군분투하고 있습니다.
매우 작고 고도로 분리된 배포 단위로 구성된 아키텍처는 더 빠른 변화 속도를 지원할 수 있습니다.

마이크로서비스에서는 성능이 흔히 문제가 됩니다. 분산 아키텍처는 작업을 완료하기 위해 많은 네트워크 호출을 해야 하므로 성능 오버헤드가 높습니다.
또한 각 엔드포인트에 대한 신원 및 접근 권한을 확인하기 위해 보안 검사를 수행해야 하므로 지연 시간이 더 늘어납니다. 마이크로서비스는 데이터 지연 문제도 안고 있는데, 요청을 처리하기 위해 여러 서비스의 조정이 필요한 경우 데이터베이스 호출이 여러 번 발생하기 때문입니다.

이러한 이유로 마이크로서비스 환경에서는 성능 향상을 위한 다양한 아키텍처 패턴이 존재하며, 여기에는 네트워크 호출 과부하를 방지하기 위한 지능형 데이터 캐싱 및 복제 등이 포함됩니다. 마이크로서비스가 오케스트레이션보다는 코레오그래피를 사용하는 또 다른 이유는 성능 향상입니다. 결합도가 낮을수록 통신 속도가 빨라지고 병목 현상이 줄어들기 때문입니다.

마이크로서비스는 각 서비스 경계가 도메인에 대응하는, 도메인 분할 아키텍처의 전형적인 예입니다. 경계 컨텍스트 덕분에 마이크로서비스는 현대 아키텍처 중에서도 양자 역학적 척도가 가장 명확하게 드러나는 아키텍처라고 할 수 있습니다. 극단적인 분리라는 핵심 철학은 어려움을 야기하기도 하지만, 제대로 구현했을 때는 엄청난 이점을 가져다줍니다. 모든 아키텍처와 마찬가지로, 아키텍트는 규칙을 이해하고 현명하게 규칙을 깨뜨릴 수 있어야 합니다.

예시 및 사용 사례

기능 및 데이터 모듈화 수준이 높은 시스템은 마이크로서비스 아키텍처에 매우 적합합니다. 이 아키텍처의 강점을 잘 보여주는 사례로는 환자의 심박수, 혈압, 산소 포화도 등 생체 신호를 모니터링하는 의료 시스템을 들 수 있습니다. 시스템이 모니터링하는 각 생체 신호는 독립적인 기능으로 자체 데이터를 관리하며, 이는 마이크로서비스 아키텍처와 경계 컨텍스트 개념에 잘 부합합니다.

이 환자 모니터링 시스템은 환자 모니터링 장치에서 입력값을 읽고, 생체 징후를 기록하고, 이러한 생체 징후에서 문제나 이상 징후를 분석하고, 문제가 발견되면 의료 전문가에게 알림을 보냅니다. 각 생체 징후는 다른 서비스 및 데이터와 대부분 독립적인 별도의 마이크로서비스로 표현될 수 있습니다.

하지만 이러한 독립성의 예외는 특정 생체 신호(예: 심박수)가 경고 또는 문제를 분석하기 위해 다른 생체 신호(예: 수면 모니터링 서비스)로부터 추가 정보를 필요로 하는 경우입니다.

그림 18-17은 마이크로서비스 아키텍처를 사용하여 이 시스템을 설계하는 방법을 보여줍니다. 각 생체 신호가 별도의 마이크로서비스로 구현되고, 각 마이크로서비스가 생체 신호 측정값과 과거 데이터를 위한 자체 데이터 저장소를 유지하는 것을 확인할 수 있습니다.

모든 생체 신호 서비스에 공통으로 적용되는 알림 기능은 'Alert Staff'라는 공유 서비스로 구현됩니다. 이 서비스는 특정 서비스에서 측정값에 이상이 감지될 경우 간호사나 의사에게 알림을 보냅니다. 각 서비스는 최신 생체 신호 측정값을 'Display Vital Signs' 라는 공유 서비스 로 구현된 환자 병실 모니터로 비동기적으로 전송합니다 .

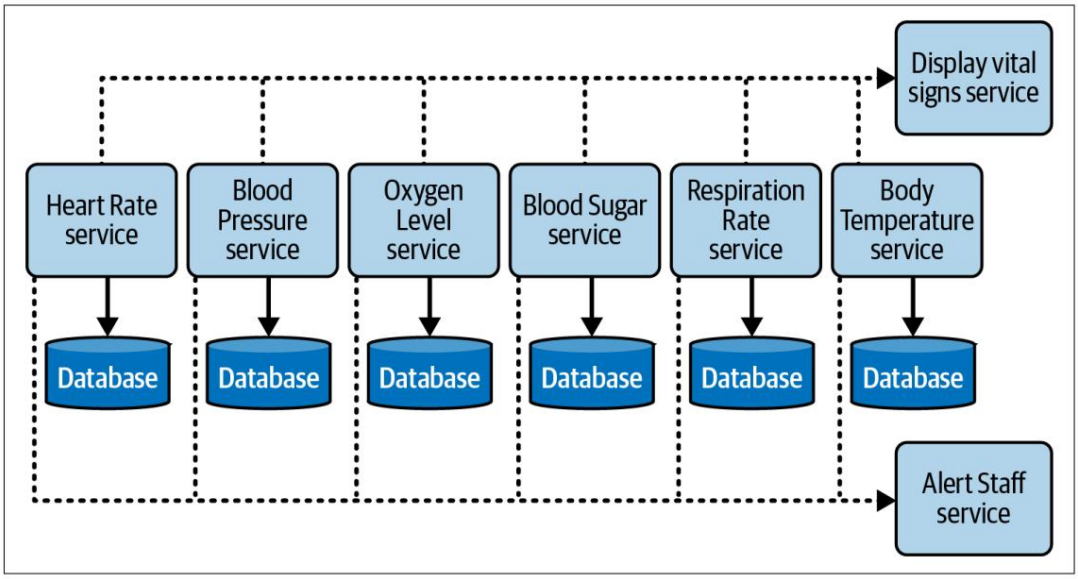


그림 18-17. 마이크로서비스 아키텍처를 사용하여 구현된 환자 의료 모니터링 시스템

이 예시는 마이크로서비스 아키텍처의 강점과 장점을 명확하게 보여줍니다. 예를 들어, 내결함성을 살펴보면 생체 신호 서비스 중 하나에 문제가 발생하거나 응답하지 않더라도 다른 모든 생체 신호 모니터링 서비스는 정상적으로 작동합니다. 이는 특히 의료 모니터링 분야에서 매우 중요합니다. 또 다른 강점은 테스트 용이성입니다. 개발자가 혈압 모니터링 서비스에 대한 유지보수를 수행하는 경우, 테스트 범위가 작아서 해당 생체 신호 서비스에 대한 완벽한 테스트를 수행하고 다른 생체 신호 서비스에 영향을 미치지 않도록 할 수 있습니다. 마지막으로, 마이크로서비스의 또 다른 강점인 진화 용이성은 아키텍트가 다른 서비스에 영향을 주지 않고도 생체 신호 모니터링 서비스를 쉽게 추가할 수 있다는 것을 보여줍니다.

이 장에서는 마이크로서비스 아키텍처의 중요한 측면들을 많이 살펴보았으며, 더 자세히 알아볼 수 있는 훌륭한 자료들이 많이 있습니다. 마이크로서비스에 대한 심층적인 학습을 위해서는 다음 자료들을 추천합니다.

- Sam Newman 저 , Building Microservices, 2nd Edition (O'Reilly, 2021)
- Luca Mezzalira 저, Building Micro-Frontends, 2nd Edition (O'Reilly, 2025)
- Mark Richards 저, Microservices vs. Service-Oriented Architecture (O'Reilly, 2016)
- Mark Richards 저, Microservices AntiPatterns and Pitfalls (O'Reilly, 2016)