제17장

오케스트레이션 기반 서비스 지향 아키텍처

건축 양식은 그것이 발전했던 시대의 맥락에서 건축가들에게 의미를 갖지만, 예술 사조와 마찬가지로 후대에는 관련성을 잃어버립니다. 오케스트레이션 기반 서비스 지향 아키텍처(SOA)가 이러한 경향을 잘 보여주는 예입니다. 아키텍처 결정에 영향을 미치는 외부 요인과 논리적이지만 궁극적으로는 재앙적인 조직 철학이 결합되어 SOA는 결국 시대에 뒤떨어지게 되었습니다.

하지만 이는 특정 조직 아이디어가 논리적으로는 타당해 보이지만 개발 과정에서 가장 중요한 부분을 방해할 수 있다는 것을 보여주는 훌륭한 사례입니다.
이는 우리의 첫 번째 법칙을 무시할 때 발생하는 위험성 중 하나를 보여줍니다. 소프트웨어 아키텍처의 모든 것은 장단점이 있다는 것입니다.

위상수학

오케스트레이션 기반 SOA의 토폴로지는 그림 17-1에 나와 있습니다.

이러한 아키텍처 스타일의 모든 예시가 정확히 이러한 계층 구조를 갖는 것은 아니지만, 모두 아키텍처 내에서 서비스 분류 체계를 구축하고 각 계층에 특정하고 명확하게 정의된 책임을 부여한다는 동일한 아이디어를 따릅니다.

서비스 지향 아키텍처는 분산 아키텍처입니다. 그림 17-1 에는 정확한 경계선이 표시되어 있지 않은데 , 이는 조직과 도구에 따라 달라지기 때문입니다. 분류 체계의 일부 요소는 애플리케이션 서버 내부에 존재할 수도 있습니다.

오케스트레이션 기반 SOA는 특정 서비스 분류 체계를 중심으로 하며, 아키텍처 내에서 서로 다른 기술적 책임과 특정 계층에 할당된 역할을 갖습니다.

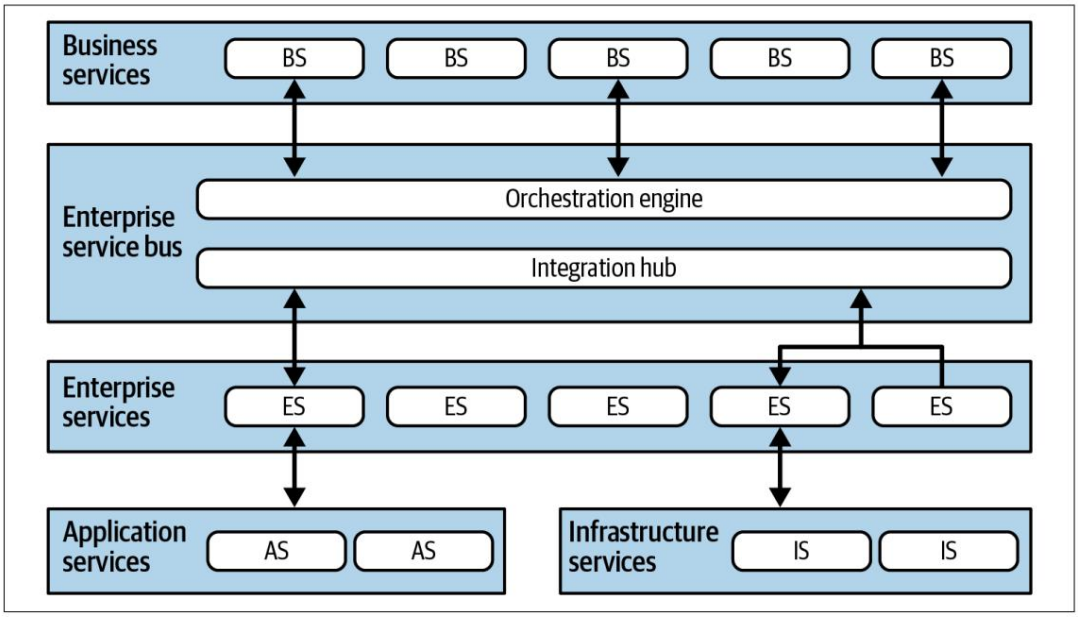

그림 17-1. 오케스트레이션 기반 서비스 지향 아키텍처의 토폴로지

스타일 사양

오케스트레이션 기반 SOA는 오늘날의 아키텍트들에게는 주로 역사적인 관심사입니다. 이러한 아키텍처를 구축하면서 얻은 교훈은 해당 분야의 발전에 중요한 부분을 차지합니다. 하지만 아키텍트들은 여전히 일부 통합 아키텍처 시나리오에서 오케스트레이션 기반 SOA의 일부 요소가 유용하다고 생각합니다.

서비스 지향 아키텍처(SOA)는 1990년대 후반에 등장했는데, 당시에는 모든 종류의 소규모 기업들이 급속도로 성장하여 대기업으로 발돋움하고, 더 작은 기업들과 합병하면서 이러한 성장을 수용하기 위해 더욱 정교한 IT 시스템이 필요했습니다. 그러나 컴퓨팅 자원은 부족하고 귀중했으며 상업적인 목적도 있었습니다. 분산 컴퓨팅이 비로소 가능해지고 필수적인 기술이 되었으며, 많은 기업들이 분산 컴퓨팅의 확장성과 기타 유익한 특성을 필요로 했습니다.

이 시기에 여러 외부 요인으로 인해 아키텍트들은 상당한 시스템 제약 조건을 가진 분산 아키텍처를 설계하게 되었습니다. 오픈 소스 운영 체제가 진지한 작업에 사용할 만큼 안정적이라고 여겨지기 전에는 운영 체제가 비쌌고 기기별로 라이선스를 취득해야 했습니다. 마찬가지로 상용 데이터베이스 서버는 복잡한 라이선스 체계를 가지고 있었는데, 이로 인해 데이터베이스 연결 풀링을 제공하는 애플리케이션 서버 공급업체들이 데이터베이스 공급업체들과 경쟁해야 하는 경우도 있었습니다.

대규모로 사용할 경우 많은 자원이 고가였기 때문에 건축가들은 가능한 한 많은 것을 재사용한다는 공통된 철학을 채택했습니다.

이러한 기술적 우려는 잦은 합병과 성장으로 인한 정보 및 업무 흐름의 중복에 대한 조직적 우려와 결합되었습니다.

조직들은 핵심 비즈니스 엔티티 간의 다양성과 불일치로 어려움을 겪었습니다. 이러한 상황 때문에 SOA의 목표가 매력적으로 다가왔습니다. 결과적으로 아키텍트들은 모든 형태의 재사용을 이 아키텍처의 핵심 철학으로 받아들였는데, 그 부작용은 320페이지의 "재사용…과 결합" 에서 다룹니다 . 이 아키텍처 스타일은 또한 아키텍트들이 기술적 분할이라는 개념을 얼마나 극단적으로 밀어붙일 수 있는지를 보여주는 좋은 예입니다. 기술적 분할은 좋은 동기에서 시작될 수 있지만, 극단으로 치달을 경우 부정적인 결과를 초래할 수 있습니다.

서비스 이름이 왜 이렇게 많나요?

아키텍트들은 소프트웨어 아키텍처에서 '서비스'라고 불리는 수많은 개념들 때문에 혼란스러워하는 경우가 많습니다. 이 책에서도 우리는 세 가지 다른 '서비스' 스타일을 다룹니다. 이 장에서는 SOA를, 18장에서는 마이크로서비스를, 그리고 14장 에서는 서비스 기반 아키텍처를 살펴봅니다 .
문제의 일부는 아키텍처를 설명하는 언어의 유연성 부족에 있습니다. 또 다른 원인은 소프트웨어 개발 생태계의 끊임없는 진화에 있습니다. '서비스'는 말 그대로 서비스를 제공하는 무언가를 나타내는 좋은 일반적인 이름이기 때문에 아키텍트들은 이 이름을 재사용하는 경향이 있습니다. 하지만 아키텍처 스타일이 진화함에 따라 '서비스'라는 용어의 의미도 바뀝니다. 예를 들어, 오케스트레이션 기반 SOA의 엔티티 서비스는 마이크로서비스 아키텍처(서비스 기반 아키텍처와는 또 다른 개념)의 서비스와 거의 모든 면에서 다릅니다. 번거롭긴 하지만, 아키텍트들은 '서비스'라는 단어가 이름에 나타날 때마다 문맥을 분석해야 하는 경우가 많습니다. 이 용어 자체가 의미 확산에 시달리고 있는 것입니다.

분류 체계 이 아

키텍처의 핵심 철학은 특정 유형의 추상화와 엔터프라이즈 수준의 재사용입니다. 많은 대기업들은 소프트웨어를 끊임없이 재작성해야 하는 문제에 시달렸습니다. 그들은 명확하게 정의된 계층과 그에 상응하는 책임을 가진 엄격한 서비스 분류 체계를 구축함으로써 이 문제를 점진적으로 해결하는 전략을 세웠습니다. 분류 체계의 각 계층은 궁극적인 추상화와 재사용이라는 두 가지 목표를 지원합니다.

비즈니스 서비스

비즈니스 서비스는 SOA의 최상위에 위치하며 비즈니스 프로세스의 진입점을 제공합니다. 예를 들어, ExecuteTrade나 PlaceOrder 와 같은 서비스는 이러한 서비스에 적합한 동작 범위를 나타냅니다. 당시 흔히 사용되던 판단 기준 중 하나는 아키텍트가 각 서비스에 대해 "우리는 이런 사업을 하고 있는가?"라는 질문에 "예"라고 답할 수 있는지 여부였습니다. 만약 그렇다면, 해당 서비스는 적절한 세분성 수준에 있는 것입니다.

하지만 개발자가 ExecuteTrade 와 같은 비즈니스 프로세스를 수행하기 위해 CreateCustomer 와 같은 메서드가 필요할 수 있지만 , CreateCustomer는 비즈니스 서비스에 적합한 추상화 수준이 아닙니다. 회사는 고객을 생성하는 사업을 하는 것이 아니라, 거래를 실행하기 위해 고객을 생성해야 하는 것입니다.

이러한 서비스 정의에는 코드가 전혀 포함되지 않고 입력, 출력 및 경우에 따라 스키마 정보만 포함됩니다. 비즈니스 사용자 및/또는 분석가가 이러한 서비스 시그니처를 정의하기 때문에 비즈니스 서비스라고 부릅니다.

엔터프라이즈 서비스

는 세분화된 공유 구현체를 포함합니다. 일반적으로 개발자 팀은 고객 생성(CreateCustomer) 또는 견적 계산(CalculateQuote) 과 같은 특정 비즈니스 도메인과 고객, 주문, 품목 과 같은 트랜잭션 엔티티를 중심으로 원자적 동작을 구축 합니다. 이러한 엔터프라이즈 서비스는 오케스트레이션 엔진을 통해 서로 연결되는 비즈니스 서비스를 구성하는 빌딩 블록입니다.

비즈니스 서비스와 엔터프라이즈 서비스 간의 추상화 수준 차이를 주목할 필요가 있습니다. 비즈니스 서비스는 비교적 거친 수준인 반면, 엔터프라이즈 서비스는 세밀한 수준으로 설계되어 다양한 유형의 추상화, 워크플로 및 엔티티를 포괄합니다.
기업 서비스를 설계할 때 아키텍트의 목표는 독립적으로 작동하는 비즈니스 기능의 완벽하게 캡슐화된 구성 요소를 만들어, 이러한 구성 요소를 자유롭게 조합하여 더욱 복잡한 비즈니스 워크플로우를 구축할 수 있도록 하는 것입니다.

훌륭한 목표이긴 하지만, 아키텍트들은 이러한 모든 요소들 사이에서 이상적인 추상화 지점을 찾는 것이 매우 어렵고, 수많은 상충되는 절충점 때문에 사실상 불가능하다는 것을 알게 됩니다. 궁극적으로, 다른 기술적으로 분할된 아키텍처와 마찬가지로, 이 아키텍처는 재사용이라는 필수 요건에서 비롯된 엄격한 책임 분리를 시도합니다. 개발자가 적절한 수준의 세분화된 엔터프라이즈 서비스를 구축할 수 있다면, 비즈니스 부서는 해당 비즈니스 워크플로 부분을 다시 작성할 필요가 없다는 것이 핵심 아이디어입니다. 이론적으로는, 비즈니스 부서는 점진적으로 재사용 가능한 엔터프라이즈 서비스 형태의 재사용 가능한 자산들을 축적해 나갈 수 있을 것입니다.

불행히도, 역동적인 현실과 소프트웨어 개발 생태계의 진화적 영향으로 인해 이러한 시도는 성공적이지 못합니다. 비즈니스 구성 요소는 건축 자재처럼 수십 년 동안 사용할 수 있는 솔루션이 아닙니다. 시장, 기술 변화, 엔지니어링 관행 및 기타 여러 요인들이 소프트웨어 세계에 안정성을 부여하려는 시도를 어렵게 만듭니다.

애플리케이션 서비스

는 아키텍처 내의 모든 서비스가 엔터프라이즈 서비스처럼 동일한 수준의 세분성이나 재사용성을 필요로 하는 것은 아닙니다. 애플리케이션 서비스는 일회성으로 구현되는 단일 서비스입니다.
예를 들어, 어떤 애플리케이션에 위치 정보가 필요하지만, 조직에서 그 정보를 재사용 가능한 서비스로 만드는 데 시간과 노력을 들이고 싶지 않을 수 있습니다. 이때 일반적으로 단일 애플리케이션 팀이 소유하는 애플리케이션 서비스가 이러한 문제를 해결해 줍니다.

인프라 서비스

인프라 서비스는 모니터링, 로깅, 인증, 권한 부여 등과 같은 운영 관련 사항을 지원합니다. 이러한 서비스는 일반적으로 구체적인 구현체로 구성되며, 운영팀과 긴밀히 협력하는 공유 인프라팀이 관리합니다. 아키텍트는 이러한 아키텍처를 구축할 때 기술적 분할을 핵심 철학으로 삼기 때문에, 인프라 서비스를 분리하여 구축하는 것이 타당합니다.

오케스트레이션 엔진 및 메시지 버스 오케

스트레이션 엔진은 이 분산 아키텍처의 핵심을 이루며, 트랜잭션 조정 및 메시지 변환과 같은 기능을 포함한 오케스트레이션을 사용하여 비즈니스 서비스 구현을 통합합니다. 오케스트레이션 엔진은 비즈니스 서비스와 엔터프라이즈 서비스 간의 관계, 매핑 방식, 트랜잭션 경계를 정의합니다. 또한 통합 허브 역할을 하여 아키텍트가 사용자 정의 코드를 패키지 및 레거시 소프트웨어 시스템과 통합할 수 있도록 합니다. 이러한 기능 조합은 엔터프라이즈 서비스 버스(ESB)와 같은 도구의 현대적인 활용 사례를 보여줍니다. 대부분의 아키텍트는 전체 아키텍처를 ESB 중심으로 구축하는 것을 바람직하지 않다고 생각하지만, ESB는 통합 작업이 많은 환경에서 매우 유용합니다. 아키텍트가 통합 허브와 오케스트레이션 엔진을 결합해야 하는 경우, 이미 이러한 기능을 포함하는 도구를 사용하는 것이 더 효율적일 수 있습니다. (이는 아키텍트로서 개발해야 할 또 다른 중요한 역량, 즉 과장된 홍보와 허황된 광고를 구분하여 도구의 진정한 용도를 파악하는 능력을 보여줍니다.)

메시지 버스가 아키텍처의 핵심을 이루기 때문에 콘웨이의 법칙(139페이지의 "콘웨이의 법칙" 참조)은 이 엔진을 담당하는 통합 아키텍트 팀이 조직 내에서 정치적인 세력이 되고 결국 관료주의적 병목 현상을 초래하는 경향이 있음을 정확하게 예측합니다.

이처럼 중앙 집중식으로 분류된 접근 방식은 매력적으로 들릴 수 있지만, 실제로는 대부분 실패로 끝났습니다. 트랜잭션 동작을 오케스트레이션 도구에 맡기는 것은 좋아 보이지만, 아키텍트들은 적절한 세분화 수준을 찾는 데 어려움을 겪습니다. 분산 트랜잭션으로 래핑된 몇 가지 서비스를 구축할 수는 있지만, 아키텍처는 점점 더 복잡해집니다. 개발자들은 엔티티들이 수많은 워크플로우에 관여하게 되면서 서비스 간의 적절한 트랜잭션 경계를 어디에 두어야 할지 결정해야 합니다. 경영진은 기업들이 트랜잭션 빌딩 블록을 엔터프라이즈 서비스로 성공적으로 구축할 수 있을 것이라고 예측하고 기대했지만, 실제로는 어려운 것으로 드러났습니다.

메시지 흐름 모

든 요청은 이 아키텍처의 로직이 있는 오케스트레이션 엔진을 통과합니다. 따라서 그림 17-2에서 보는 것처럼 내부 호출의 경우에도 메시지 흐름은 엔진을 통과합니다.

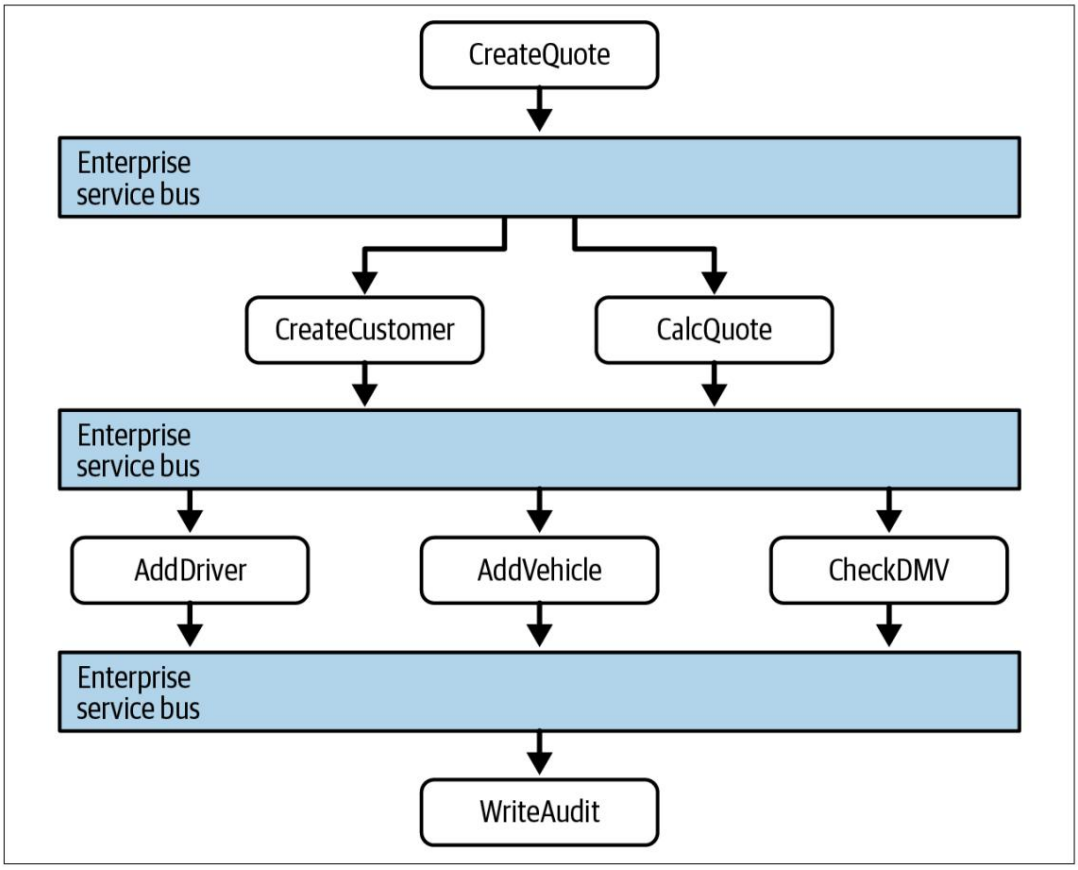

그림 17-2. 서비스 지향 아키텍처를 사용한 메시지 흐름

그림 17-2 에서 CreateQuote 비즈니스 레벨 서비스는 워크플로를 정의하는 서비스 버스를 호출합니다. 워크플로는 CreateCustomer 및 CalculateQuote 서비스에 대한 호출로 구성되며 , 각 서비스 는 애플리케이션 서비스도 호출합니다. 서비스 버스는 이 아키텍처 내의 모든 호출에 대한 중개자 역할을 하며, 통합 허브이자 오케스트레이션 엔진으로 기능합니다.

재사용…그리고 결합

이 아키텍처를 처음 활용한 설계자들의 주요 목표 중 하나는 서비스 수준에서의 재사용, 즉 시간이 지남에 따라 점진적으로 재사용할 수 있는 비즈니스 동작을 구축하는 능력이었습니다. 그들은 가능한 한 적극적으로 재사용 기회를 찾아내라는 지시를 받았습니다.

예를 들어, 그림 17-3 에 나타난 상황을 생각해 보십시오 . 한 건축가는 보험 회사의 6개 부서 각각에 다음과 같은 개념이 포함되어 있음을 깨달았습니다.
고객.

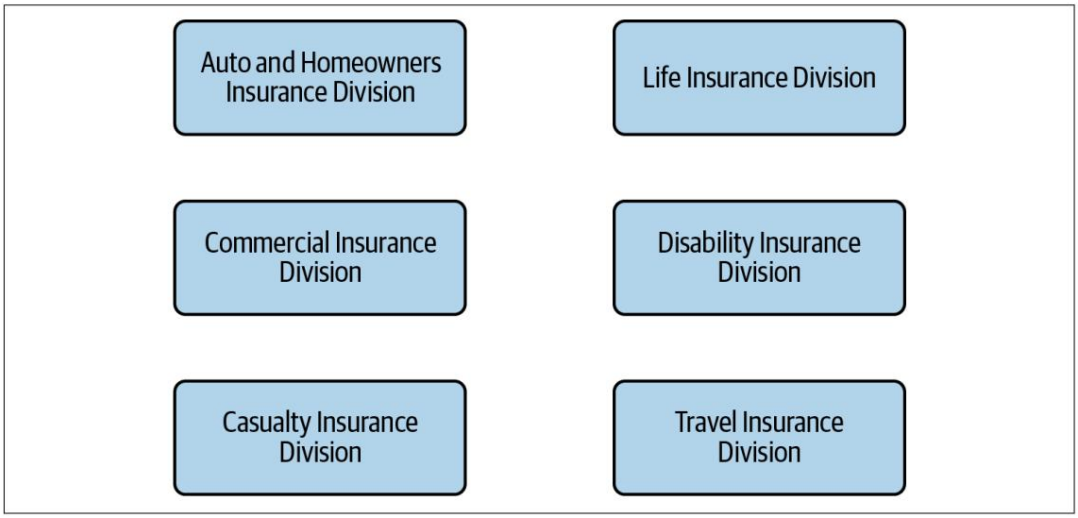

그림 17-3. 서비스 지향 아키텍처에서 재사용 기회 모색

따라서 적절한 SOA 전략은 고객 관련 부분을 재사용 가능한 서비스로 추출한 다음, 기존 서비스들이 표준 고객 서비스를 참조하도록 하는 것입니다. 그림 17-4에서 볼 수 있듯이, 아키텍트는 모든 고객 동작을 단일 고객 서비스로 분리하여 재사용이라는 목표를 달성했습니다.

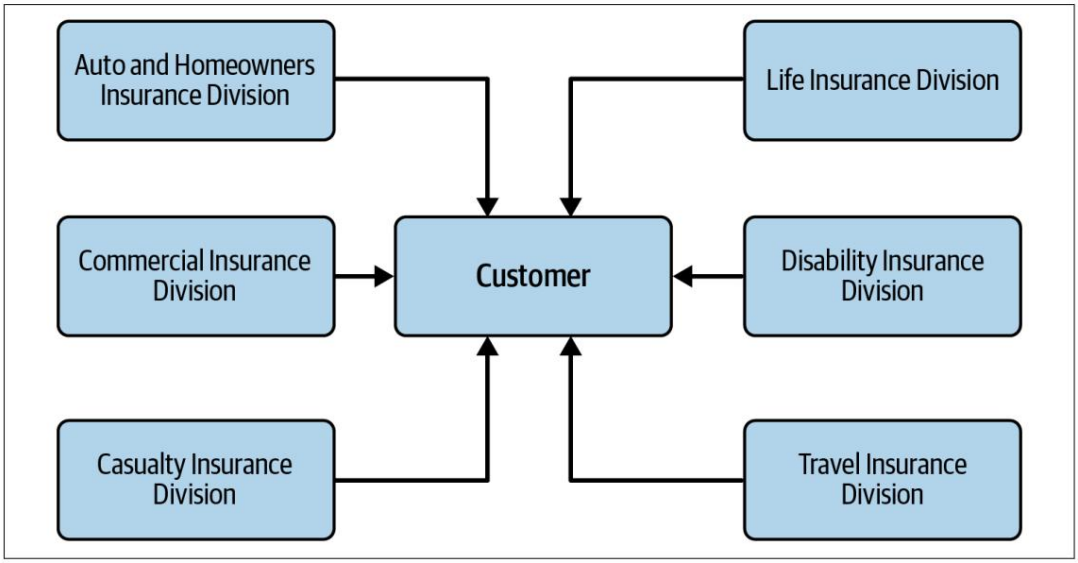

그림 17-4. 서비스 지향 아키텍처에서 정형 표현 구축

건축가들은 이러한 설계의 부정적인 측면을 서서히 깨달았습니다. 첫째, 팀이 재사용을 중심으로 시스템을 구축하면 구성 요소 간의 결합도가 크게 높아집니다. 결국 재사용은 결합을 통해 구현되기 때문입니다. 예를 들어 그림 17-4 에서 고객 서비스 에 대한 변경 사항은 다른 모든 서비스에 파급 효과를 미칩니다. 이로 인해 점진적인 변경조차 위험해집니다. 각 변경 사항은 잠재적으로 엄청난 파급 효과를 가져오기 때문입니다. 이는 결국 조정된 배포, 전체적인 테스트, 그리고 엔지니어링 효율성을 저해하는 여러 요소를 필요로 합니다.

행동을 한 곳으로 통합하는 것의 또 다른 부정적인 부작용은 그림 17-4 의 자동차 보험과 장애 보험 사례를 생각해 보면 알 수 있습니다 . 단일 고객 서비스를 지원하기 위해 각 부서는 조직이 고객에 대해 알고 있는 모든 세부 정보를 포함해야 합니다. 자동차 보험에는 운전면허증이 필요한데, 이는 차량이 아닌 개인의 재산입니다. 따라서 고객 서비스는 장애 보험 부서에서는 전혀 중요하지 않은 운전면허증 관련 세부 정보까지 포함해야 합니다. 그럼에도 불구하고 장애 보험 팀은 단일 고객 정의로 인한 추가적인 복잡성을 처리해야 합니다. DDD(도메인 주도 설계)에서 전체적인 재사용을 지양하는 이유는 이러한 유형의 아키텍처에서 얻은 경험에서 비롯된 것입니다.

오케스트레이션 기반 SOA에 대한 가장 뼈아픈 사실은 기술적 분할에 지나치게 집중된 아키텍처를 구축하는 것이 비현실적이라는 점일 것입니다. 분리와 재사용이라는 철학적 관점에서는 타당해 보이지만, 실제로는 악몽과도 같습니다.

예를 들어, 개발자들은 흔히 "CatalogCheckout에 새 주소 줄 추가 "와 같은 작업을 수행합니다. CatalogCheckout 과 같은 도메인 개념은 이 아키텍처 전체에 너무 얇게 흩어져 있어서 사실상 먼지가 되어버렸습니다. SOA에서는 이러한 작업이 여러 계층에 걸쳐 수십 개의 서비스와 단일 데이터베이스 스키마 변경을 포함할 수 있습니다. 더욱이, 현재 엔터프라이즈 서비스가 적절한 트랜잭션 세분성으로 정의되어 있지 않으면 개발자는 설계를 변경하거나 트랜잭션 동작을 변경하기 위해 거의 동일한 새 서비스를 구축해야 합니다. 이렇게 되면 재사용성은 사실상 불가능해집니다.

데이터 토폴로지

이 책에서 다루는 다른 많은 아키텍처 스타일과는 달리, 오케스트레이션 기반 SOA의 데이터 토폴로지는 그 역사적 기원을 고려할 때 그다지 흥미롭지 않습니다.
여러 부분으로 구성된 분산 아키텍처임에도 불구하고, 일반적으로 하나(또는 몇 개)의 관계형 데이터베이스를 사용하는데, 이는 1990년대 후반의 모든 분산 아키텍처에서 흔히 사용되던 방식입니다. 트랜잭션 처리조차도 일반적으로 데이터베이스가 아닌 이 아키텍처에 맡겨졌습니다. 메시지 버스는 토폴로지 내 각 엔티티에 대한 선언적 트랜잭션 상호 작용을 포함하는 경우가 많았으며, 이를 통해 개발자, 아키텍트 또는 기타 사용자는 데이터베이스나 엔티티의 상황적 재사용 여부와 관계없이 트랜잭션 동작을 결정할 수 있었습니다.

이 시대의 건축가들은 데이터를 마치 낯선 나라처럼 여겼습니다. SOA와 이벤트 기반 아키텍처 모두에서 데이터는 필수적인 요소이지만, 당시에는 데이터를 문제 영역의 일부라기보다는 단순히 통합 지점으로 취급했습니다.

정말요? 선언적 트랜잭션이요?!?

네, 정말입니다. 오케스트레이션 기반 SOA가 전성기를 누리던 시절, 많은 애플리케이션 서버의 "특징" 중 하나는 구성 관리자가 원하는 트랜잭션 컨텍스트에 따라 개별 엔티티의 트랜잭션 범위를 변경할 수 있도록 하는 것이었습니다. (당시 유행했던 장황하지만 파싱하기 쉬운 구성 형식을 선호했던 만큼, 이러한 설정은 XML로 선언되었습니다.) 엔티티(JavaBean의 특수 유형인 EntityBeans 라고 함) 선언의 일부는 워크플로 참여 시 트랜잭션 범위를 결정하며, 아키텍트는 워크플로가 트랜잭션인지 아닌지를 직접 선언합니다. 그러면 애플리케이션 서버는 데이터베이스와 상호 작용하여 엔티티 및/또는 워크플로의 원하는 동작에 맞는 데이터베이스 트랜잭션을 생성하고 관리합니다.

이러한 방식은 두 가지 이유로 대체로 실패했습니다. 첫째, 개발자가 런타임 시 트랜잭션 동작 방식을 알 수 없으면 엔티티와 종속성에 상당한 복잡성이 추가됩니다. 이로 인해 개발자는 트랜잭션 범위만 다른 거의 동일한 버전의 엔티티를 만들어야 합니다. 둘째, 벤더들이 메시지 버스에 아무리 정교한 기능을 추가하더라도, 수많은 오류 모드가 시스템이 트랜잭션을 깔끔하게 관리하지 못하게 하는 예외적인 상황이 끊임없이 발생하여 사람이 풀어야 할 복잡하고 일관성 없는 문제를 야기합니다. 트랜잭션과 같은 시스템의 복잡하고 다면적인 기능은 깔끔하게 추상화할 수 없습니다. 추상화 과정에서 너무 많은 취약점이 발생하면 신뢰성을 확보할 수 없습니다.

클라우드 고려 사항

오케스트레이션 기반 SOA는 클라우드보다 수십 년 앞서 등장했기 때문에 (원래 형태의) 이 아키텍처를 클라우드에 구축하는 것은 고려 대상이 아닙니다.

하지만 오늘날 이러한 방식의 사용은 워크플로우에 통합되고 참여해야 하는 클라우드 및 온프레미스 서비스에 적합한 통합 아키텍처라는 장점을 제공합니다.
이 아키텍처는 주로 통합 아키텍처이기 때문에 클라우드 기반 서비스 및 시설과 잘 어울립니다.

일반적인 위험

지난 세기 말과 이번 세기 초에 이러한 아키텍처의 가장 큰 위험 요소는 주로 비용, 구현 기간, 그리고 (놀랍게도) 시스템 유지 관리 및 업데이트의 어려움에 관한 것이었습니다.

이러한 프로젝트 중 상당수는 매우 비용이 많이 드는 다년간의 사업이었으며, 중요한 결정이 수반되었습니다!

이러한 프로젝트들은 회사 내 고위층에서 결정되었습니다. 기업들은 이러한 프로젝트들을 "실패"라고 부르기보다는, DDD의 개념에 더 부합하는, 더 나은 경계를 가진 통합 아키텍처로 전환하는 경우가 대부분이었습니다.

아키텍트가 최신 시스템에서 ESB(Enterprise Service Building)를 통합 기능으로 사용할 때 가장 큰 위험은 ESB가 점차 전체 아키텍처를 캡슐화하도록 허용하는 것입니다. 이를 '의도치 않은 SOA 안티패턴'이라고 하는데, 아키텍트가 자신도 모르게 점진적으로 오케스트레이션 중심의 SOA를 구축해 나가는 현상을 말합니다. '의도치 않은 SOA'를 피하려면 아키텍트는 오케스트레이션에 대한 적절한 캡슐화 경계를 설정하고 트랜잭션 경계와 같은 문제에 세심한 주의를 기울여야 합니다.

통치

이러한 아키텍처가 인기를 끌던 시절에는 현대적인 통합 검사 방식은 흔하지 않았습니다.
팀들은 공식적인 품질 보증 수준의 테스트 외에는 SOA를 거의 테스트하지 않았기 때문에, 도구 및 프레임워크 개발자들은 개별 구성 요소에 대한 테스트를 용이하게 하는 데에는 거의 신경을 쓰지 않았습니다. 메시지 버스와 그 관련 구성 요소들의 거대한 구조를 위한 모의 객체와 스텁을 생성하는 몇몇 테스트 프레임워크가 등장했지만, 이러한 프레임워크들은 항상 다루기 어렵고 일관성이 부족했습니다.

거버넌스 역시 같은 한계에 직면했습니다. 아키텍처 거버넌스를 자동화한다는 개념은 테스트 자동화보다 훨씬 더 생소했습니다. 당시 "거버넌스"는 무거운 프레임워크, 회의, 코드 리뷰 등 모두 수동으로 이루어지는 작업들을 의미했습니다.

하지만 아키텍트들은 여전히 ESB가 제공하는 특정 기능 조합이 필요한 조직 내에서 전략적으로 ESB를 활용합니다. 특히 많은 기업은 최신 시스템과 상호 작용해야 하는 레거시 시스템을 보유하고 있으며, 이러한 시스템은 종종 결과를 결합하고 동작을 집계해야 하는데, 이는 ESB의 핵심 기능을 모두 설명합니다.

이러한 시나리오에서 적합도 함수는 데이터 또는 경계 컨텍스트가 생태계의 다른 부분으로 "누출"되는 것을 방지하는 데 중요한 역할을 할 수 있습니다.

예를 들어, ESB(Enterprise Service Bus)를 사용하여 ERP(Enterprise Resource Planning) 패키지, 온라인 판매 도구, 그리고 최신 마이크로서비스 기반 회계 서비스 간의 조정을 수행하는 시스템을 생각해 보겠습니다. 이 시나리오에서 시스템은 ERP 및 판매 시스템에서만 데이터를 읽고 회계 마이크로서비스에만 데이터를 써야 합니다. 아키텍트는 먼저 모든 통신이 일관되게 로그에 기록되도록 하는 적합성 함수를 구축한 다음, 다음과 같은 적합성 함수(의사 코드로 제시됨)를 작성할 수 있습니다.

```
지난 24 시간 동안 의 ERP 로그를 ERP-logs 에 읽어들이고, 지난 24 시간 동안
의 판매 로그를 Sales-logs 에 읽어들입니다 . ERP-logs 의 각 항목 에 대해
'operation' 이 'update' 이고
        'target'이 'accounting'이 아닌 경우,
            적합성 함수 위반을 높이세요
                    "통합 지점 간 통신 오류"
```

```
            만약 종료한다면
        판매 기록 의 각 항목
            'operation' 이 'update' 이고 'target'이 'accounting' 이 아닌 경우
                적합성 함수 위반을 높이세요
                        "통합 지점 간 통신 오류"
            만약 종료한다면
```

이 적합성 검사 기능은 두 통합 지점의 로그 항목을 읽어 회계 시스템이 아닌 다른 대상을 위한 업데이트 작업이 발생하지 않도록 합니다 .

이러한 적합성 함수를 사용하면 아키텍트는 ESB와 같은 도구를 전략적으로 활용하는 동시에 팀이 일반적으로 오용하는 부분을 방지하는 안전장치를 구축할 수 있습니다.

팀 토폴로지 고려 사항

건축가들이 오케스트레이션 기반 SOA를 설계할 때 데이터 토폴로지를 고려하지 않았던 것처럼, 팀 토폴로지 역시 이 아키텍처 스타일이 유행하던 당시에는 생소한 주제였습니다.

사실, 이러한 스타일의 엄격한 분류 체계는 아키텍트들이 팀 토폴로지 원칙을 개발하게 된 계기가 된 의사소통의 안티패턴 역할을 했습니다. 이 아키텍처의 목표는 책임의 극단적 분리와 그에 따른 팀 구성원의 분리입니다. 이를 채택한 기업들 중에서도 비즈니스 서비스를 개발하는 사람과 엔터프라이즈 서비스를 개발하는 사람이 대화하는 경우는 매우 드물었습니다.

그들은 계약서나 인터페이스 같은 기술적 산출물을 통해 소통할 것으로 예상되었습니다. 이러한 방식의 추상화 수준은 여러 통합 계층을 만들어내고, 각 계층은 서로 다른 팀에서 구현하며, 기업 수준의 티켓팅 도구를 사용하여 소통합니다. 개발자들이 이러한 방식으로 기능을 구축하는 데 시간이 많이 소요되는 이유를 쉽게 알 수 있습니다.

스타일 특징

현재 우리가 아키텍처 스타일을 평가하는 데 사용하는 많은 기준들은 오케스트레이션 기반 SOA가 인기를 끌던 시절에는 우선순위가 아니었습니다. 애자일 소프트웨어 개발 운동이 막 시작되었고, 이러한 아키텍처를 사용할 가능성이 높은 대규모 조직에는 아직 널리 보급되지 않았기 때문입니다.

그림 17-5 의 특징 평가표에서 별 1개는 특정 건축 특징이 건축물에서 제대로 구현되지 않았음을 의미하고, 별 5개는 해당 건축 특징이 해당 스타일의 가장 강력한 특징 중 하나임을 의미합니다. 평가표에 제시된 특징에 대한 정의는 4 장 에서 확인할 수 있습니다 .

SOA는 아마도 지금까지 시도된 범용 아키텍처 중 가장 기술적으로 세분화된 아키텍처일 것입니다! 실제로 이러한 구조의 단점에 대한 반발로 마이크로서비스와 같은 더 현대적인 아키텍처가 등장하게 되었습니다. SOA는 단일 양자 컴퓨팅을 가지고 있으며, 심지어

분산 아키텍처이긴 하지만, 두 가지 이유가 있습니다. 첫째, 일반적으로 단일 데이터베이스 또는 몇 개의 데이터베이스만 사용하기 때문에 여러 다양한 관심사 간에 결합 지점이 생깁니다.
둘째, 그리고 더 중요하게는, 오케스트레이션 엔진은 거대한 결합점 역할을 합니다. 아키텍처의 어떤 부분도 모든 동작을 조율하는 미들웨어와 다른 특성을 가질 수 없습니다. 따라서 이 아키텍처는 모놀리식 아키텍처와 분산 아키텍처 모두의 단점을 해결할 수 있습니다.

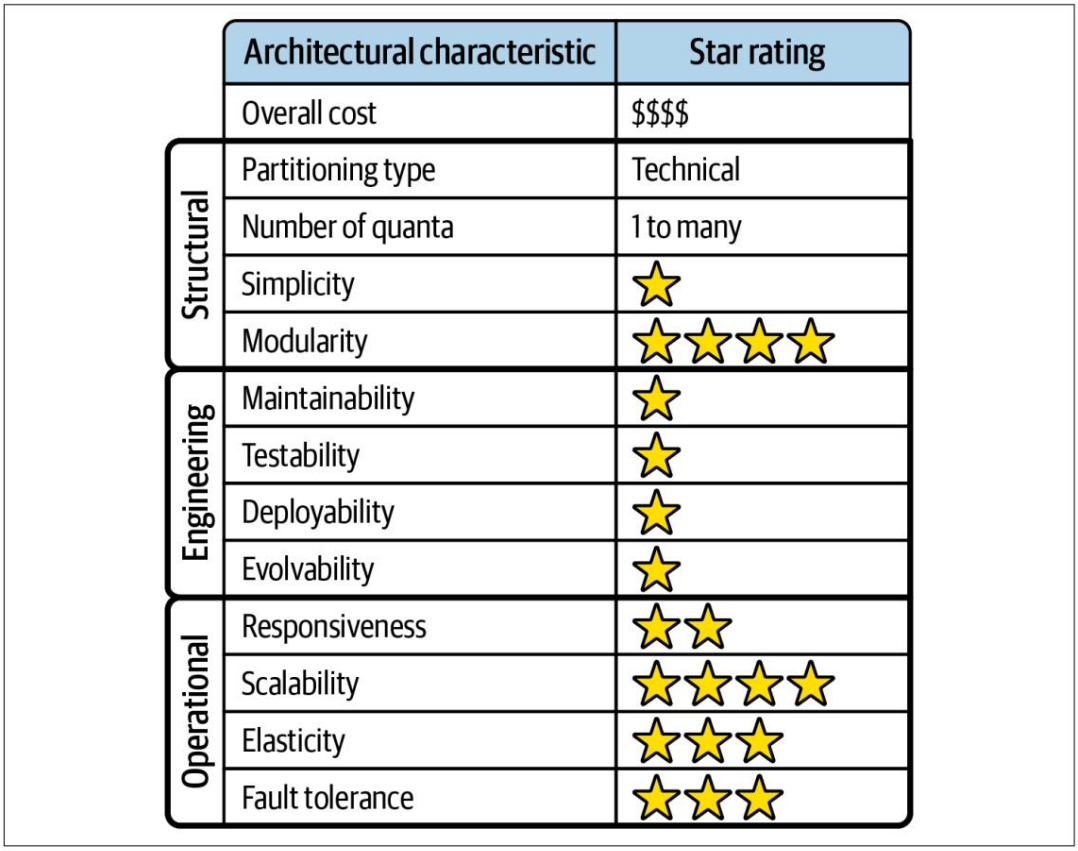

그림 17-5. 서비스 지향 아키텍처 특성 평가

배포 용이성 및 테스트 용이성과 같은 최신 엔지니어링 목표는 이 아키텍처에서 처참한 결과를 보여줍니다. 이는 해당 목표에 대한 지원이 미흡할 뿐만 아니라, 개발 당시 이러한 목표가 중요하게 여겨지지 않았기 때문입니다.

이 아키텍처는 구현상의 어려움에도 불구하고 탄력성과 확장성 같은 몇 가지 목표를 지원합니다. 툴 벤더들은 애플리케이션 서버 간 세션 복제 및 기타 기술을 구축하여 이러한 시스템의 확장성을 확보하기 위해 막대한 노력을 기울였습니다. 그러나 분산 아키텍처의 특성상 각 비즈니스 요청이 아키텍처 곳곳에 분산되어 처리되므로 성능은 항상 아쉬운 부분입니다.

이러한 모든 요인 때문에 단순성과 비용은 대부분의 아키텍트가 선호하는 관계와는 정반대입니다. 오케스트레이션 기반 SOA는 아키텍트들에게 기술적 파티셔닝의 실질적인 한계와 분산 트랜잭션이 실제 환경에서 얼마나 어려울 수 있는지를 알려주었기 때문에 중요한 이정표였습니다.

예시 및 사용 사례

이러한 아키텍처의 대표적인 사례는 1990년대 후반과 2000년대 초반에 많은 대기업에서 찾아볼 수 있었습니다. 하지만 점차 마이크로서비스와 같은 더욱 민첩하고 도메인 기반의 분산 아키텍처로 대체되었습니다. 대기업조차도 변화는 불가피하며 소프트웨어는 정적인 것이 아니라 시장 상황과 새로운 기능에 맞춰 변화해야 한다는 사실을 깨달았습니다.

아키텍트들은 대규모 조직 전반에 걸쳐 효과적인 재사용을 달성하기 위해 오케스트레이션 기반 SOA 아키텍처를 구축했지만, 결국 엄격하고 정교한 분류 체계 때문에 일반적인 변경 및 업데이트를 구현하는 데 어려움을 겪는다는 사실을 깨달았습니다. 예를 들어, 일반적인 도메인 변경 사항 중 하나는 단일 엔티티에 대한 세부 정보를 업데이트하는 것입니다.
운이 좋으면 개발자는 엔터프라이즈 서비스 계층의 구성 요소만 변경하면 작업을 완료할 수 있습니다. 그러나 최악의 경우(엔터프라이즈 아키텍트나 비즈니스 이해관계자가 이러한 유형의 변경을 예상하지 못한 경우) 개발자는 아키텍처의 네다섯 개 계층을 업데이트해야 할 수도 있으며, 각 계층에서 매우 밀접하게 연관된 변경 작업을 수행해야 합니다. 이러한 방식으로 일하는 아키텍트는 '변경'이라는 단어를 듣는 것을 두려워하는데, 이는 심층적인 분석이 필요하고 작업 범위가 매우 가변적이기 때문입니다.

324페이지의 "거버넌스" 에서 언급했듯이 , 아키텍트는 특히 통합 아키텍처를 위해 오케스트레이션 기반 SOA(예: ESB)의 구성 요소를 여전히 사용합니다. 예를 들어, ESB는 통신, 프로토콜 및 계약 변환을 용이하게 하는 통합 허브와 아키텍트가 다양한 통합 엔드포인트 간의 워크플로를 구축할 수 있도록 하는 오케스트레이션 엔진을 모두 포함합니다. 오케스트레이션 기반 SOA는 여러 계층의 간접 연결을 포함하므로 아키텍트는 그림 17-6에서 보여주는 것처럼 엔터프라이즈 서비스를 통합 지점, 패키지 소프트웨어 또는 맞춤형 코드 등 다양한 방식으로 구현할 수 있습니다.

클라이언트 요청은 메시지 버스를 사용하여 어떤 엔터프라이즈 서비스를 호출할지, 어떤 순서로 호출할지, 그리고 어떤 정보를 수집할지 결정합니다. 엔터프라이즈 서비스는 차례로 API를 통해 사용자 정의 코드, 기존 시스템 또는 패키지 소프트웨어와 통신합니다.
곧.

오케스트레이션 기반 SOA는 아키텍트가 생태계의 제약 조건 내에서 대규모 통합 문제를 처리하는 방식에 대한 흥미로운 혁신을 보여줍니다.
예를 들어, 이 아키텍처가 인기를 끌던 당시 대부분의 조직에서는 오픈 소스 운영 체제를 사용하지 않았기 때문에 마이크로서비스와 같은 대안 아키텍처는 비용이 너무 많이 들어 실현 불가능했습니다. 아키텍트는 과거의 접근 방식에서 교훈을 얻어야 합니다. 우리는 계속해서 이러한 교훈을 공유할 수 있습니다.

여전히 의미가 있는 부분은 활용하면서, 실패한 부분과 그 이유에서 얻은 교훈을 내면화한다.

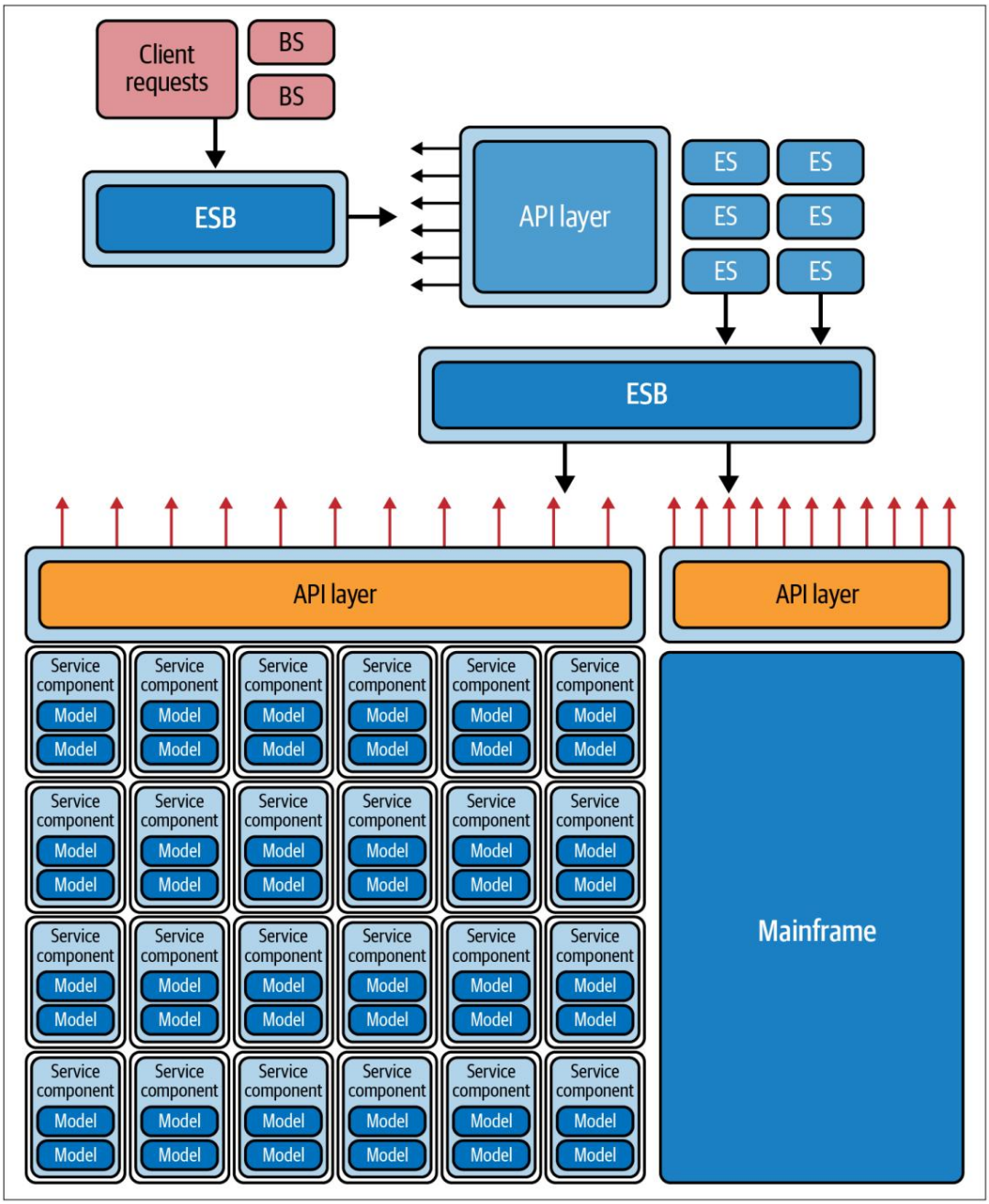

그림 17-6. 이 아키텍처 스타일의 추상화 계층은 구현을 가능하게 합니다. 유연성