Machine Translated by Google

제20장

건축 패턴

9 장 에서 우리는 아키텍처 스타일과 아키텍처 패턴을 구분합니다 . 다시 말해, 스타일은 아키텍트가 토폴로지, 물리적 아키텍처, 배포 방식, 통신 방식, 데이터 토폴로지의 차이로 구분하는 명명된 토폴로지입니다. 아키텍처 패턴은 자주 언급되는 책 『디자인 패턴』 에서 영감을 받아 문제에 대한 맥락화된 해결책을 제시합니다.

아키텍처 패턴과 "모범 사례"(21 장 에서 더 자세히 다룹니다)를 구분하는 것이 중요합니다. 어떤 것을 "모범 사례"라고 부르는 것은 특정 상황이 발생할 때마다 건축가가 반드시 그 사례를 활용해야 할 명확한 의무가 있음을 암시합니다. "더 나은 사례"라고 부른다면 최소한 논쟁의 여지가 있겠지만, 우리는 그것을 "모범 사례"라고 부릅니다. 이는 건축가가 고민할 필요 없이 항상 같은 해결책을 따르도록 유도합니다.

패턴과 솔루션을 구분하는 것도 중요합니다. 많은 도구, 프레임워크, 라이브러리 및 기타 소프트웨어 개발 산출물은 구현 방식에 따라 하나 이상의 패턴을 캡슐화하며, 그 충실도나 다른 패턴과의 혼합 정도에 따라 그 패턴이 다르게 나타날 수 있습니다. 따라서 먼저 가장 적합한 패턴을 파악한 다음, 그에 가장 적합한 구현 방식을 선택하는 데 집중해야 합니다.

이 장에서는 이 책 2부에서 소개한 스타일들을 비교하고 맥락을 제공하기 위해 몇 가지 대표적인 패턴을 소개합니다. 첫 번째 예시인 아키텍처 재사용 패턴은 패턴과 구현의 차이점을 명확히 보여줍니다.

Machine Translated by Google

재사용

도메인 결합과 운영 결합 간의 분리는 마이크로서비스와 같은 분산 아키텍처에서 흔히 발생하는 아키텍처적 문제입니다.

도메인과 운영 결합 분리

마이크로서비스 아키텍처 설계 목표 중 하나는 높은 수준의 결합도 감소이며, 이는 흔히 "결합보다는 중복이 낫다"라는 조언으로 나타납니다.

두 서비스가 고객 프로필 정보를 서로 주고받아야 한다고 가정해 보겠습니다. 하지만 도메인 중심의 경계 컨텍스트 아키텍처는 구현 세부 정보를 각 서비스 내부에만 비공개로 유지해야 한다고 규정합니다. 일반적인 해결책은 각 서비스에 ' 프로필'과 같은 엔티티에 대한 자체적인 내부 표현을 제공하고, JSON 형식의 이름-값 쌍과 같이 느슨하게 결합된 방식으로 해당 정보를 전달하는 것입니다.

이를 통해 각 서비스는 통합을 깨뜨리지 않고 기술 스택을 포함한 내부 표현 방식을 자유롭게 변경할 수 있습니다. 개발자들은 일반적으로 코드 중복을 좋아하지 않는데, 이는 동기화 문제, 의미론적 차이 등을 야기할 수 있기 때문입니다. 하지만 코드 중복보다 더 나쁜 것들이 있는데, 마이크로서비스에서는 결합도가 바로 그런 문제입니다.

마이크로서비스를 설계하는 아키텍트는 일반적으로 결합도를 낮추기 위해 구현을 중복해야 하는 경우가 있다는 현실을 받아들입니다. 하지만 높은 결합도가 필요한 기능은 어떻게 해야 할까요? 각 서비스는 모니터링, 로깅, 인증 및 권한 부여, 회로 차단기 등과 같은 공통적인 운영 기능을 갖춰야 합니다. 그러나 각 팀이 이러한 종속성을 관리하도록 맡기면 종종 혼란이 초래됩니다.

예를 들어, 모든 서비스를 보다 쉽게 운영하기 위해 표준 모니터링 솔루션을 하나 선택하려는 회사를 생각해 보겠습니다. 아키텍트는 각 팀이 담당하는 서비스에 대한 모니터링 구현을 책임지도록 결정합니다. 즉, 결제 서비스 팀, 재고 관리 서비스 팀 등이 각각 담당하게 되는 것입니다. 하지만 운영 팀은 각 팀이 실제로 구현을 완료했는지 어떻게 확인할 수 있을까요? 또한, 통합 업그레이드와 같은 문제는 어떻게 처리해야 할까요? 표준화된 모니터링 도구를 조직 전체에 걸쳐 업그레이드해야 할 경우, 각 팀은 어떻게 협력해야 할까요?

Machine Translated by Google

육각형 아키텍처 패턴(그

림 20-1 참조)에서 도메인 로직은 육각형의 중심에 위치하며, 그 주변을 포트와 어댑터가 연결하여 에코시스템의 다른 부분과 연결됩니다(이 패턴은 포트 및 어댑터 패턴이라고도 합니다). 그림 20-1에서 이 패턴을 시각적으로 확인할 수 있습니다 .

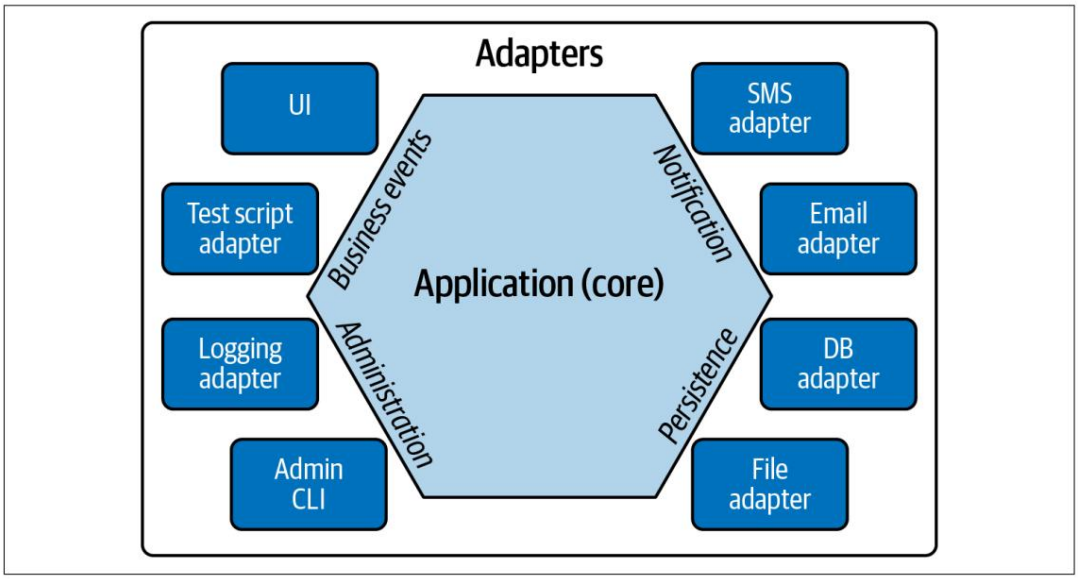

그림 20-1. 육각형 건축 패턴

눈썰미 좋은 독자라면 육각형의 여섯 변 중 네 변만 사용되었다는 것을 알아차렸을 것입니다. 이 패턴의 창시자인 알리스터 콕번은 처음에는 육각형 모양으로 그리고 '육각형 건축 패턴'이라고 이름 붙였습니다. 하지만 '포트 및 어댑터'라는 이름이 훨씬 더 설명적이라는 것을 깨닫고 곧바로 후회했습니다. 이미 늦었기 때문입니다. 많은 건축가들이 '육각형'이라는 단어가 멋있다고 생각해서 그 이름이 굳어졌습니다.

패턴을 구현과 혼동하는 것은 흔한 실수이며, 이것이 좋은 예입니다. 헥사고널 아키텍처는 마이크로서비스를 설명하는 데 사용될 때 잠재적으로 심각한 결함을 가지고 있는데, 이는 해당 패턴의 원래 의도를 이해하는 사람들에게만 해당됩니다. 헥사고널 아키텍처는 현대적인 마이크로서비스보다 앞서 등장했지만, 둘 사이에는 많은 유사점이 있습니다. 하지만 중요한 차이점이 하나 있는데, 바로 데이터 충실도입니다. 헥사고널 아키텍처는 데이터베이스를 마치 플러그인처럼 연결할 수 있는 또 다른 어댑터처럼 취급합니다. 데이터 스키마를 비즈니스 로직으로 포함하지 않은 이유는 (헥사고널 패턴이라는 이름이 만들어졌을 당시 흔했던) 데이터베이스가 완전히 별개의 시스템이라는 오해 때문이었습니다. 에릭 에반스는 그의 저서 "도메인 주도 설계(Domain-Driven Design)"에서 데이터베이스 스키마는 어디에 위치하든 시스템의 비즈니스 로직을 반영하여 변경되어야 한다는 점을 인식함으로써 이러한 오류를 바로잡았습니다.

Machine Translated by Google

이 때문에 육각형 패턴은 건축가들 사이에서 끊임없는 혼란의 원인이 됩니다.
누군가 이 용어를 사용할 때, 운영과 도메인 관련 문제의 분리를 의미하는 것인지, 아니면 데이터를 격리하여 마이크로서비스 설계의 핵심 원칙을 위반하는 문자 그대로의 패턴을 가리키는 것인지 명확히 해야 합니다. 문맥상 오해의 소지가 없다면, "도메인과 운영 관련 문제의 분리"를 줄여서 패턴 이름을 사용하는 것은 괜찮습니다. 하지만 오늘날 아키텍트는 이러한 구현 방식을 더 이상 필요로 하지 않습니다. 이제 육각형 패턴을 구현하는 데 훨씬 더 적합한 메커니즘인 서비스 메시 패턴이 있기 때문입니다.

서비스 메시

334페이지의 "운영 재사용" 에서 우리는 Side!car 및 서비스 메시 패턴을 사용하여 기술적 문제와 도메인 문제를 분리하는 일반적인 아키텍처 접근 방식을 설명했습니다.

사이드카 패턴은 단순히 운영 기능을 도메인에서 분리하는 방법일 뿐만 아니라, 특정 유형의 결합을 해결하기 위한 직교 재사용 패턴입니다(374페이지의 "직교 결합" 참조). 아키텍처 솔루션은 종종 도메인 결합과 운영 결합의 예처럼 여러 유형의 결합을 필요로 합니다.

직교 재사용 패턴은 선호하는 계층 구조에 맞지 않는 아키텍처의 특정 측면을 하나 이상의 관심사로 표현하는 방식을 제시합니다. 예를 들어, 마이크로서비스 아키텍처는 도메인을 중심으로 구성되지만, 운영상의 결합으로 인해 이러한 도메인들을 넘어서는 작업이 필요할 수 있습니다. 사이드카를 사용하면 아키텍트는 이러한 관심사를 아키텍처 전체에 걸쳐 일관성을 유지하면서도 여러 도메인을 아우르는 계층으로 분리할 수 있습니다.

> 직교 결합이란 수학에서 두 직선이 직각으로 교차할 때를 말하며, 이는 두 요소가 서로 독립적임을 의미합니다. 소프트웨어 아키텍처에서도 아키텍처의 두 부분은 직교적으로 결합될 수 있습니다. 즉, 완전한 솔루션을 구성하기 위해 서로 다른 목적을 가지면서도 교차해야 하는 경우가 있습니다. 이 장에서 가장 명확한 예는 모니터링과 같은 운영 관련 요소입니다. 이는 필수적이지만 카탈로그 결제와 같은 도메인 동작과는 독립적입니다. 직교 결합을 이해하면 아키텍트는 각 요소 간의 얽힘을 최소화하는 교차점을 찾을 수 있습니다.

Machine Translated by Google

사이드카 패턴은 훌륭한 추상화를 제공하지만, 표 20-1에서 볼 수 있듯이 다른 모든 아키텍처 접근 방식과 마찬가지로 몇 가지 단점이 있습니다.

표 20-1. 사이드카 및 서비스 메시 패턴의 장단점

장점	단점
격리된 결합을 생성하는 일관된 방법입니다.	플랫폼별로 사이드카를 구현해야 합니다.
일관된 인프라 조정이 가능합니다	사이드카 구성 요소는 규모가 커지거나 복잡해질 수 있습니다.
팀별 소유권, 중앙 집중식 소유권 또는 이 둘의 조합, 독립 팀 간의 구현 "차이"	

헥사고널 패턴과 서비스 메시 패턴은 모두 도메인 관련 사항과 운영 관련 사항을 분리하는 재사용 패턴을 구현하는 방법을 보여줍니다. 헥사고널 구현은 범용적인 반면, 서비스 메시는 마이크로서비스 및 기타 분산 아키텍처에 적합합니다. 아키텍트에게 중요한 것은 먼저 패턴, 즉 분리를 파악한 다음 아키텍처에 가장 적합한 구현 방법을 결정하는 것입니다.

의사소통

통신 패턴을 포함한 많은 아키텍처 패턴은 이벤트 기반 아키텍처에서 유래했으며, 15 장 과 18 장 에서 다룬 것처럼 메시지 및/또는 이벤트를 통해 통신하는 모든 분산 아키텍처에 적용됩니다 . 실제로, 해당 장들에서 통신 관련 예시들을 이미 살펴보았지만, 아키텍트들이 무의식적으로 패턴을 구현하는 경우가 많기 때문에 특정 패턴으로 명시적으로 언급하지는 않았습니다. 결국 패턴이란 일반적인 문제에 대한 해결책이기 때문입니다.

오케스트라 편곡 vs. 안무

258페이지의 "매개된 이벤트 기반 아키텍처" 와 341페이지의 "안무 및 오케스트레이션" 에서 이미 살펴본 두 가지 커뮤니케이션 패턴, 즉 그림 20-2에 요약된 안무와 오케스트레이션을 생각해 보십시오.

Machine Translated by Google

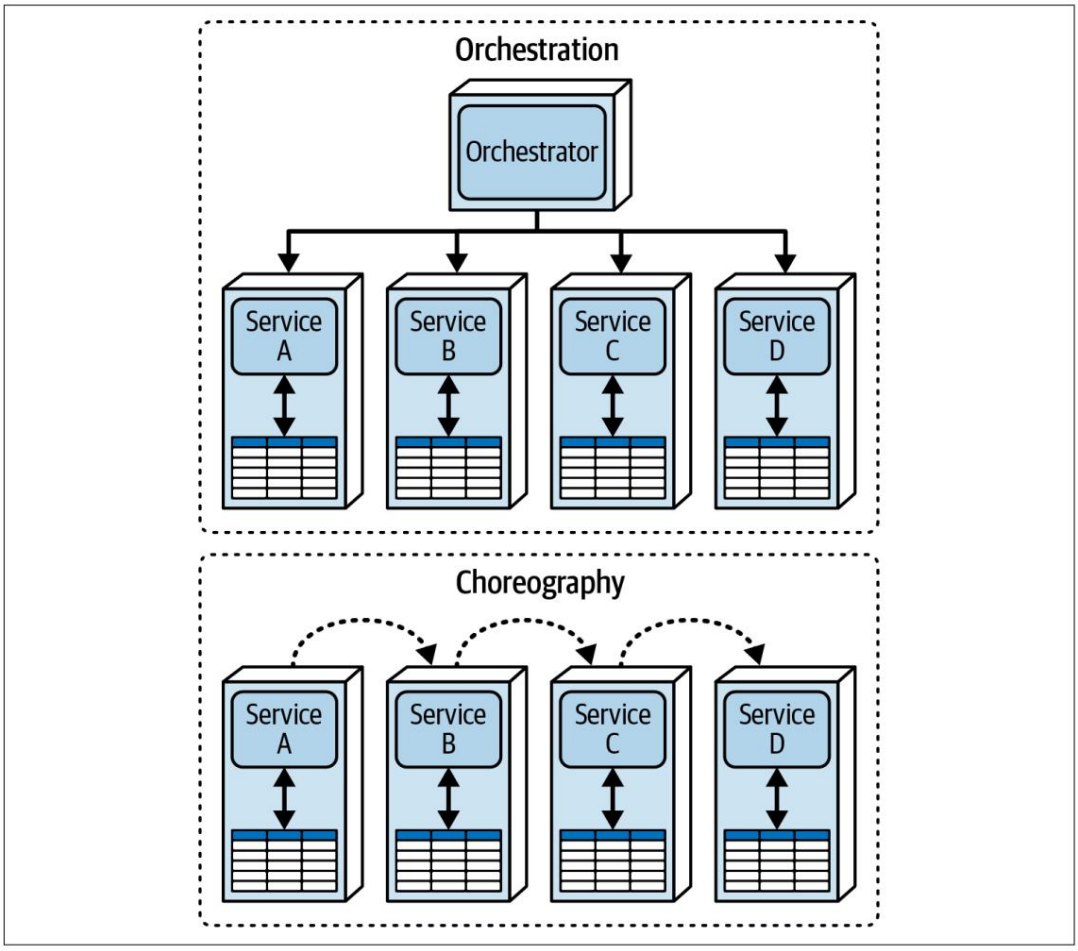

그림 20-2. 마이크로서비스 아키텍처에서의 오케스트레이션 및 안무

그림 20-2 에 나타난 각 동형 워크플로우에서 워크플로우를 구성하기 위해서는 네 개의 도메인 서비스(서비스 A~D)가 협력해야 합니다. 오케스트레이션의 경우, 워크플로우의 코디네이터 역할을 하는 별도의 서비스인 오케스트레이터도 있습니다.

우리는 이러한 소통을 오케스트레이션과 중재라는 두 가지 관점에서 설명했지만, 기본적인 패턴은 동일합니다. 아키텍트는 구현 내부에 숨겨진 패턴을 인식할 수 있으면 그로 인한 장단점을 더욱 명확하게 파악할 수 있기 때문에 이점을 얻습니다.

중재와 오케스트레이션의 장단점을 설명할 때, 많은 부분이 동일하게 다뤄졌다는 점에 주목하세요. 여기서는 장점을 요약해 보겠습니다.

중앙 집중식 워크플로우를 활용

하면 복잡성이 증가함에 따라 아키텍트는 상태, 동작 및 경계 조건에 대한 통합 구성 요소를 활용할 수 있습니다.

Machine Translated by Google

오류 처리 오류 처리
: 는 많은 도메인 워크플로의 주요 부분이며, 워크플로의 상태 소유자가 있으면 이러한 작업을 효율적으로 수행할 수 있습니다.

복구 가능성 측면
: 에서, 오케스트레이터는 워크플로의 상태를 모니터링하므로 하나 이상의 도메인 서비스에 단기적인 장애가 발생할 경우 아키텍트는 재시도 로직을 추가할 수 있습니다.

상태 관리 오케스트레이
: 터를 사용하면 워크플로의 상태를 쿼리할 수 있으므로 다른 워크플로 및 기타 임시 상태를 위한 공간을 제공합니다.

오케스트레이션의 일반적인 단점은 다음과 같습니다.

응답성 모든 통신
: 은 오케스트레이터를 거쳐야 하므로 처리량 병목 현상이 발생하여 응답성을 저해할 수 있습니다.

내결함성
: 오케스트레이션은 도메인 서비스의 복구 가능성을 향상시키지만, 워크플로에 잠재적인 단일 장애 지점을 만들 수 있습니다. 이는 이중화를 통해 해결할 수 있지만, 그만큼 복잡성도 증가합니다.

확장성 측면
: 에서 볼 때, 이러한 의사소통 방식은 안무만큼 확장성이 좋지 않습니다. 왜냐하면 오케스트레이터가 더 많은 조정 지점을 추가하게 되어 잠재적인 병렬 처리가 줄어들기 때문입니다.

서비스 결합도: 중앙 오
: 케스트레이터가 있으면 해당 오케스트레이터와 도메인 구성 요소 간의 결합도가 높아지는데, 이는 때때로 필요하지만 마이크로서비스 아키텍처에서는 바람직하지 않습니다.

마찬가지로, 우리는 마이크로서비스 아키텍처와 이벤트 기반 아키텍처 모두에서 안무(choreography)에 대해 논의했습니다. 안무 기반 워크플로의 장단점은 다음과 같습니다.

반응성: 이러한 의
: 사소통 방식은 병목 현상이 적어 병렬 처리가 가능한 기회가 더 많습니다.

확장성 측면
: 에서, 오케스트레이터와 같은 조정 지점이 없기 때문에 보다 독립적인 확장이 가능합니다.

Machine Translated by Google

내결함성

단일 오케스트레이터가 없기 때문에 아키텍트는 여러 인스턴스를 사용하여 내결함성을 강화할 수 있습니다. 물론 여러 개의 오케스트레이터를 생성할 수도 있지만, 모든 통신이 오케스트레이터를 통해 이루어져야 하므로 여러 개의 오케스트레이터를 사용할 경우 워크플로의 전반적인 내결함성 수준에 더 민감해집니다.

서비스 분리: 오케스트
레이터가 없으므로 결합도가 낮아집니다.

안무를 활용한 소통 방식의 단점은 다음과 같습니다.

분산 워크플로우에서 워크플로

우 소유자가 없으면 오류 및 기타 경계 조건을 관리하기가 더 어려워집니다.

국가 관리 중앙 집중식 국

가 관리 주체가 없다는 것은 지속적인 국가 관리를 저해합니다.

오류 처리 오케스트레

이터가 없으면 도메인 서비스가 워크플로에 대한 더 많은 지식을 갖춰야 하므로 오류 처리가 더 어려워집니다.

복구 가능성: 오케스

트레이터가 없으면 재시도 및 기타 복구 노력을 수행하기가 더 어려워집니다.

이 두 가지 패턴은 스타일과 패턴의 차이점을 명확하게 보여줍니다. 모든 분산 아키텍처는 이러한 통신 패턴을 사용할 수 있으며, 아키텍트는 각 패턴의 장단점을 평가하는 방법을 이해해야 합니다. 소프트웨어 아키텍처의 두 번째 법칙을 기억하세요. 장단점 분석은 한 번만 하고 끝낼 수 없습니다.

또한 그것들은 공통적인 패턴이 도처에 존재한다는 것을 보여주며, 이것이 바로 그것들이 존재하는 이유입니다.
흔한.

CQRS

분산 아키텍처(그리고 일부 모놀리식 아키텍처)에서 흔히 볼 수 있는 또 다른 일반적인 통신 패턴은 CQRS(Command-Query-Responsibility-Segregation)입니다. 이 간단한 통신 및 데이터 패턴은 일반적으로 데이터베이스와의 모놀리식 통신을 두 부분으로 분리합니다. 그림 20-3을 참조하십시오.

Machine Translated by Google

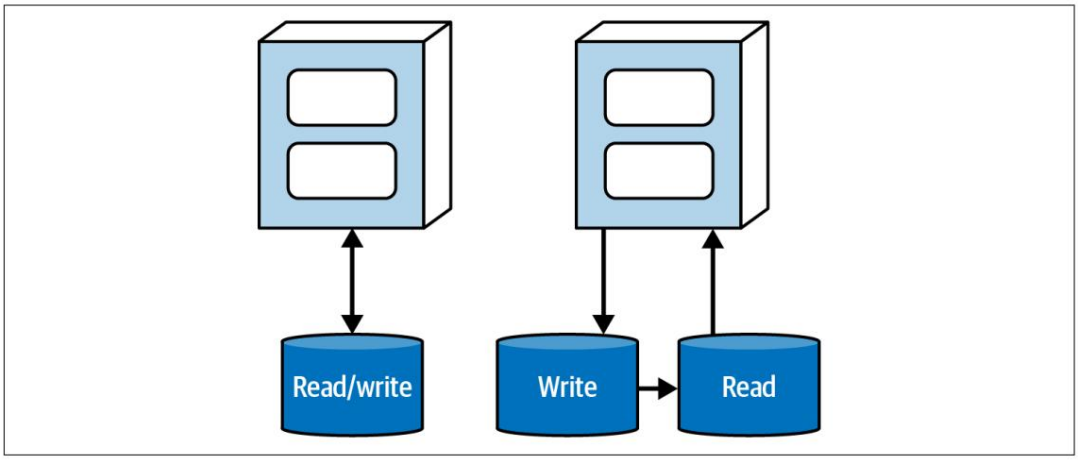

그림 20-3. 클라이언트/서버 방식과 CQRS 방식 비교

그림 20-3 의 왼쪽 구조는 일반적인 클라이언트/서버 데이터 상호 작용을 보여줍니다. 이 구조 에서 애플리케이션은 데이터베이스를 애플리케이션 인프라의 일부로 활용하여 데이터베이스에 쿼리를 보내고 트랜잭션 쓰기 작업을 수행합니다. 이는 일반적인 패턴이지만, 일부 시스템에서는 읽기 및 쓰기 작업량이 현저히 다르거나 보안 및 기타 이유로 읽기와 쓰기 작업을 분리해야 하는 경우가 있습니다. 이러한 시스템의 경우, 그림 20-3 의 오른쪽에 나타낸 CQRS(클라이언트 쿼리 응답 시스템)가 이 문제를 해결합니다.

CQRS는 쓰기 작업을 하나의 데이터 저장소(일반적으로 데이터베이스, 때로는 영구 메시지 큐와 같은 다른 인프라 구조)에 격리하고, 이 저장소는 데이터를 다른 데이터베이스와 동기화(일반적으로 비동기적으로)하여 읽기 요청을 처리합니다.

읽기와 쓰기를 분리함으로써 아키텍트는 데이터에 따라 서로 다른 아키텍처 특성을 적용할 수 있습니다. 또한 필요한 경우 각 데이터베이스에 대해 서로 다른 데이터 모델을 사용할 수 있습니다.

CQRS는 다양한 유형의 데이터 기능, 보안 문제 또는 물리적 분리를 통해 이점을 얻을 수 있는 기타 요소에 따라 서로 다른 아키텍처 특성을 지원하는 데이터 통신 패턴의 좋은 예입니다.

하부 구조

소프트웨어 아키텍처에는 팀이 일반적인 문제에 대한 유용한 상황별 해결책을 찾은 모든 곳에 패턴이 존재하며, 이러한 패턴은 종종 생태계의 다른 부분과 교차합니다(자세한 내용은 26장을 참조하십시오).

아키텍트는 구성 요소, 데이터 요소, API…그리고 인프라 간의 결합도를 중요하게 생각하며, 브로커-도메인 패턴이 그 대표적인 예입니다.

Machine Translated by Google

브로커-도메인 패턴

이 섹션에서는 그림 20-4 에 나와 있는 것처럼 이벤트 기반 아키텍처로 구현된 동일한 주문 처리 워크플로를 살펴보겠습니다 .

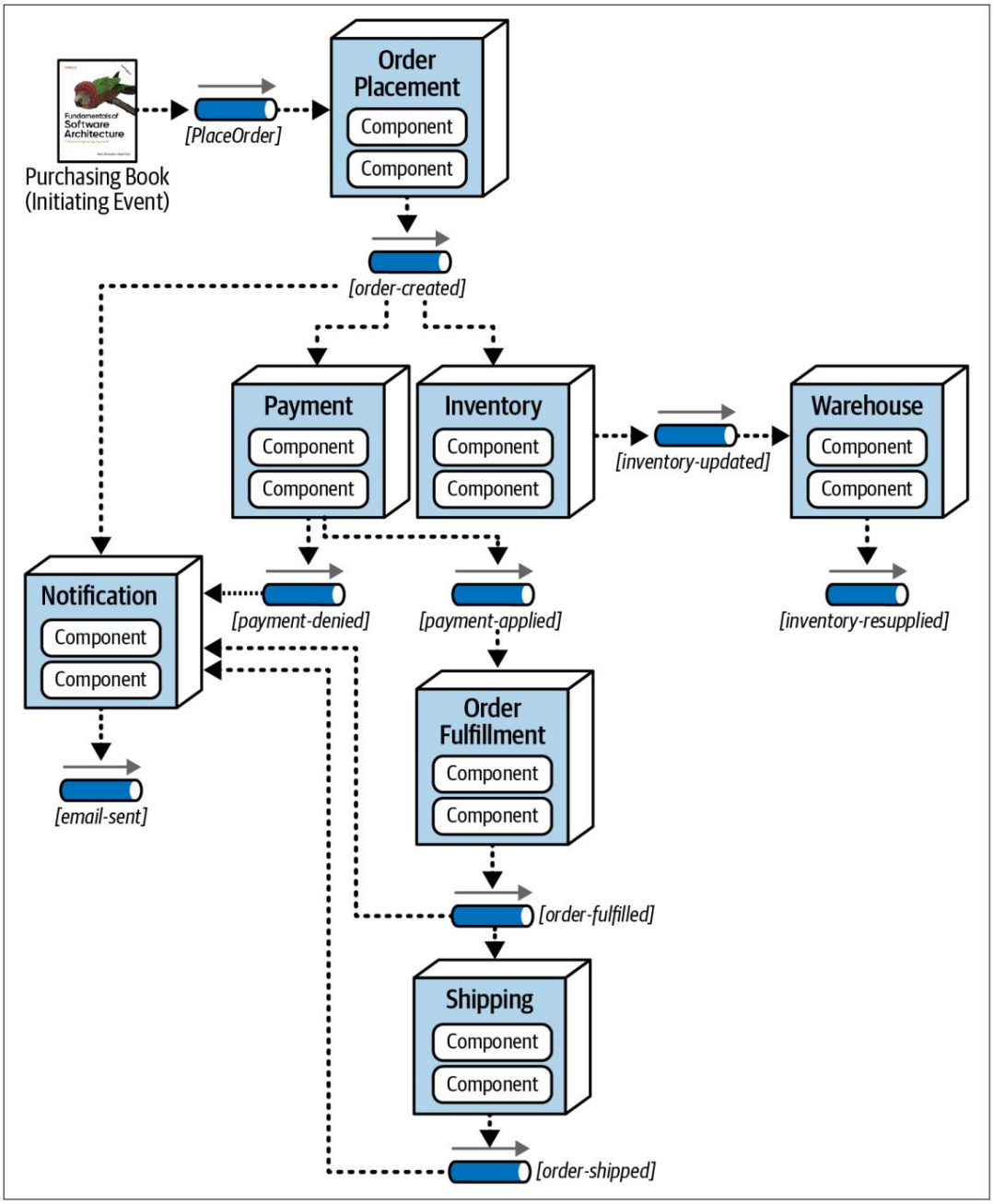

그림 20-4. EDA에서 구현된 주문 배치 워크플로

Machine Translated by Google

보시다시피, EDA는 서비스 간 통신에 이벤트를 사용하므로 이벤트 핸들러는 워크플로우를 구축하기 위해 적절한 서비스에 구독해야 합니다. 이벤트 핸들러는 아키텍처 인프라의 일부인 브로커에 의해 구현됩니다. EDA에서 토픽 또는 큐는 일반적으로 발신자가 소유합니다. 예를 들어, 결제 서비스는 토픽을 구독하기 위해 해당 토픽의 주소를 알아야 합니다.

그림 20-5 에서 OrderPlacement는 다른 프로세서들이 구독할 브로커를 "소유"하고 있습니다. 즉, 이 서비스를 지원하는 데 필요한 인프라에는 브로커가 포함됩니다. 시스템이 모든 통신에 하나의 브로커만 사용하는 경우, 모든 서비스는 인프라의 단일 부분에 의존하게 됩니다.

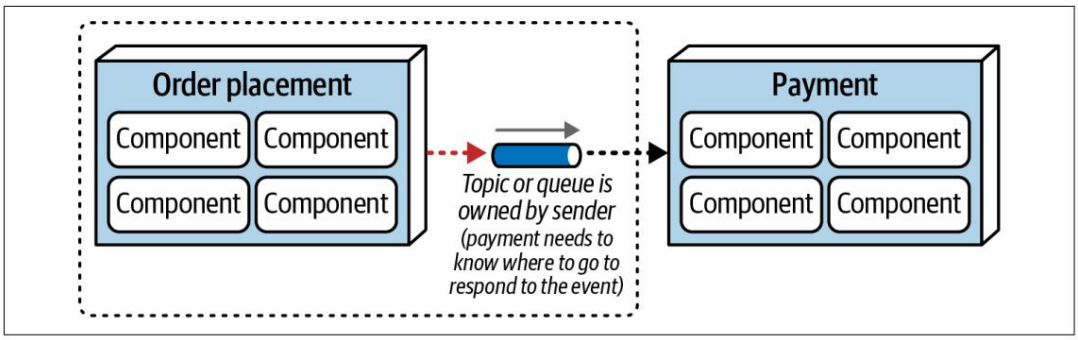

그림 20-5. 이벤트 기반 안무에서 토픽 또는 큐는 일반적으로 발신자가 소유합니다.

Machine Translated by Google

그림 20-6 에 나타난 워크플로우의 인프라는 단일 브로커를 포함하고 있어, 각 이벤트 프로세서는 워크플로우 협력자를 구독하기 위해 어디로 가야 하는지 "알고" 있습니다. 또한 단일 브로커를 통해 로깅, 모니터링 및 기타 거버넌스를 위한 단일 위치를 확보할 수 있습니다.

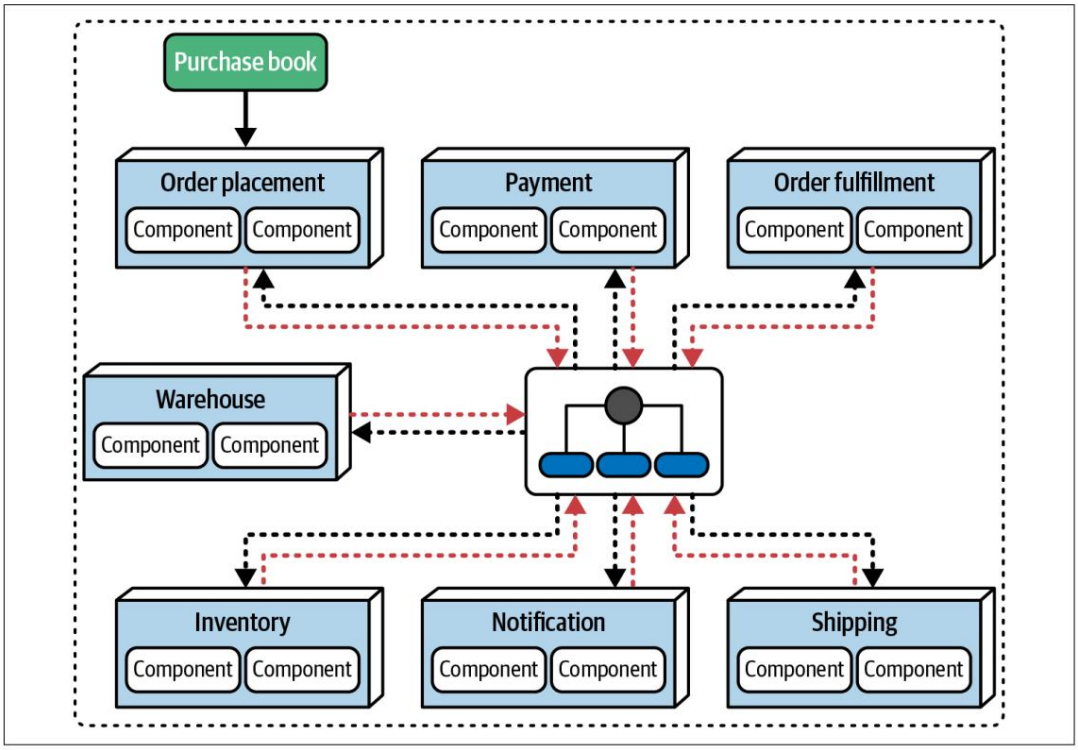

그림 20-6. 전체 워크플로우에 하나의 브로커 사용

하지만 분산 아키텍처의 목표 중 하나는 내결함성을 향상시키는 것입니다.
그림 20-6 에서 단일 브로커가 다운되면 어떻게 될까요? 전체 워크플로가 중단됩니다. 또 다른 잠재적인 문제는 확장성입니다. 모든 메시지가 단일 브로커를 거쳐야 하는 경우 메시지 양이 증가함에 따라 브로커에 과부하가 걸릴 위험이 있습니다.

Machine Translated by Google

또 다른 접근 방식은 도메인 브로커 패턴으로, 도메인의 세분성과 유사한 방식으로 인프라를 처리합니다. 그림 20-7을 참조하십시오.

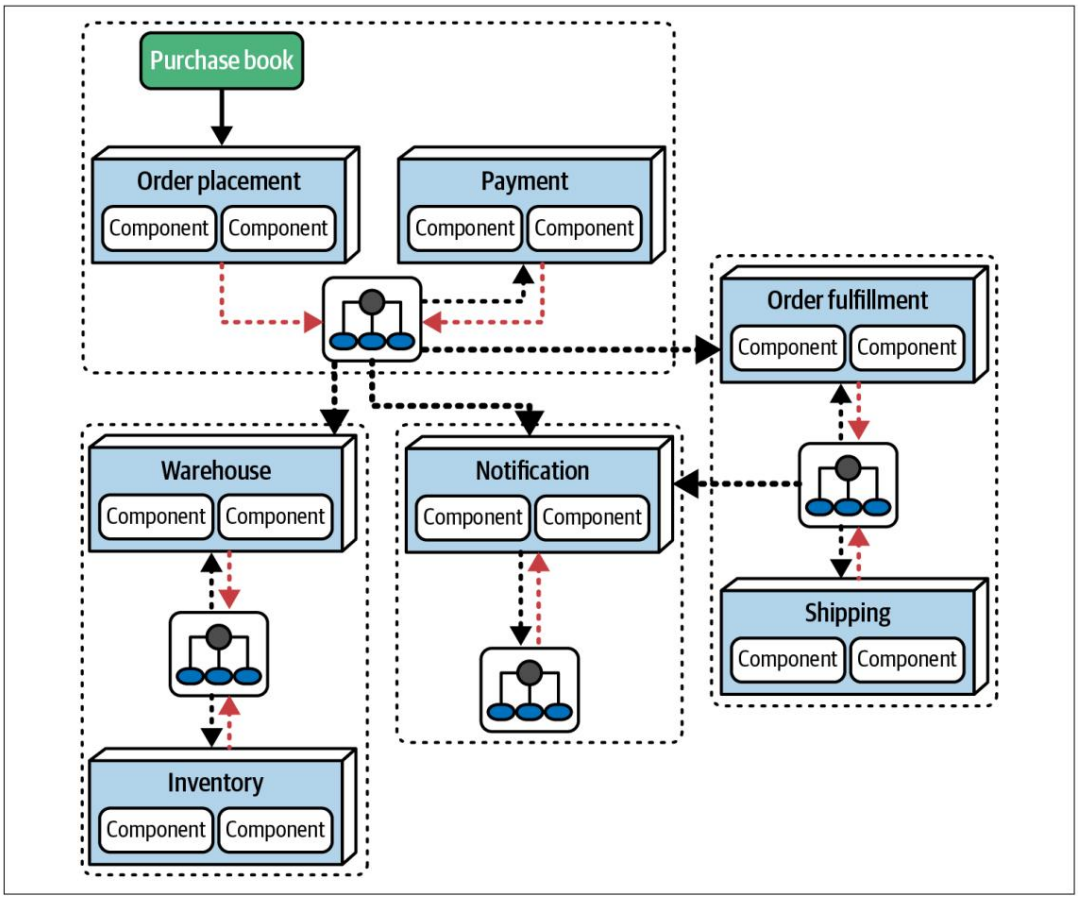

그림 20-7. 도메인 브로커 패턴을 활용하여 인프라에 소유권을 할당하는 방법

그림 20-7 에 나타난 대안적인 아키텍처에서는 각 관련 서비스 그룹이 하나의 브로커를 공유하며, 이는 아키텍처의 전체 도메인 분할을 반영합니다. 이 솔루션은 우수한 서비스 검색 기능을 유지하면서 내결함성, 확장성, 탄력성 및 기타 여러 운영 아키텍처 특성을 향상시킵니다. 그러나 이러한 접근 방식 중 어느 것도 "최적의 방법"은 아닙니다. 단일 브로커 패턴의 장단점은 표 20-2에 나와 있습니다.

표 20-2. 단일 브로커 패턴 거래

장점	단점
중앙 집중식 검색	내결함성
최소한의 인프라 처리량 제한	

Machine Translated by Google

도메인 브로커 패턴에는 표 20-3에 나와 있는 바와 같이 몇 가지 장단점이 있습니다.

표 20-3. 도메인-브로커 패턴 거래-OS

장점	단점
더 나은 격리	대기열/주제 찾기가 더 어려워짐
도메인 경계와 일치합니다. 인프라가 많을수록 비용이 더 많이 듭니다.	
확장성이 더 뛰어남	유지 관리해야 할 움직이는 부품이 더 많아졌습니다.

건축가는 설계 과정에서 탐색과 영역 분리의 필요성 사이에서 균형을 유지해야 합니다.
이러한 인프라 패턴 중 어떤 것이 특정 상황에 가장 적합한가
시스템.

Machine Translated by Google

제3부

기술 및 소프트 스킬

유능한 소프트웨어 아키텍트는 소프트웨어 아키텍처의 기술적 측면뿐만 아니라 아키텍트처럼 사고하고, 개발 팀을 이끌고, 다양한 이해관계자에게 아키텍처를 효과적으로 전달하는 데 필요한 핵심 기술과 소프트 스킬도 갖춰야 합니다. 이 책의 해당 장에서는 유능한 소프트웨어 아키텍트가 되기 위해 필요한 핵심 기술과 소프트 스킬을 다룹니다.

Machine Translated by Google