
제15장

이벤트 기반 아키텍처 스타일

이벤트 기반 아키텍처(EDA) 스타일은 확장성과 고성능을 갖춘 애플리케이션을 구현하는 데 널리 사용되는 분산형 비동기 아키텍처 스타일입니다.

이벤트 기반 아키텍처는 특히 적응성이 뛰어나 소규모 애플리케이션부터 크고 복잡한 애플리케이션까지 다양한 규모에 적용할 수 있습니다. 이벤트 기반 아키텍처는 비동기적으로 이벤트를 발생시키고 응답하는 분리된 이벤트 처리 구성 요소로 이루어져 있습니다. 독립적인 아키텍처 스타일로 사용하거나 다른 아키텍처 스타일(예: 이벤트 기반 마이크로서비스 아키텍처)에 통합하여 사용할 수 있습니다.

많은 개발자와 소프트웨어 아키텍트는 EDA를 아키텍처 스타일보다는 아키텍처 패턴으로 생각합니다. 하지만 저희는 이에 동의하지 않습니다. 필자들은 EDA만을 사용하여 시스템을 개발한 경험이 있으며, 바로 그 때문에 EDA는 본질적으로 아키텍처 스타일이라고 주장합니다. EDA는 마이크로서비스나 공간 기반 아키텍처와 같은 다른 아키텍처 스타일과 결합하여 하이브리드 아키텍처를 구성할 수도 있지만, 본질적으로는 복잡한 시스템을 설계하는 방식입니다.

대부분의 애플리케이션은 **그림 15-1**에 나타낸 것처럼 요청 기반 모델을 따릅니다. 예를 들어 고객이 지난 6개월간의 주문 내역을 요청하면, 이 요청은 처음에 요청 오케스트레이터에 의해 수신됩니다. 요청 오케스트레이터는 일반적으로 사용자 인터페이스지만, API 계층, 오케스트레이션 서비스, 이벤트 허브, 이벤트 버스 또는 통합 허브를 통해서도 구현될 수 있습니다.

이 시스템의 역할은 요청을 다양한 요청 처리기로 결정론적이고 동기적으로 전달하는 것입니다. 요청 처리기는 데이터베이스에서 고객 정보를 검색하고 업데이트하여 요청을 처리합니다. 주문 내역 정보 검색은 특정 컨텍스트 내에서 시스템에 대한 데이터 기반의 결정론적 요청이지, 시스템이 반응해야 하는 이벤트가 아니므로 요청 기반 모델이라고 할 수 있습니다.

반면 이벤트 기반 모델은 특정 이벤트에 반응하여 조치를 취합니다. 예를 들어 특정 품목에 대한 온라인 경매 입찰을 제출하는 경우를 생각해 보세요. 입찰자는 시스템에 요청을 보내는 것이 아니라, 입찰을 시작하는 것입니다.

현재 호가가 발표된 후 발생하는 이벤트입니다. 시스템은 해당 이벤트에 대응하여 동시에 접수된 다른 입찰가와 비교하고 현재 최고 입찰자를 결정해야 합니다.

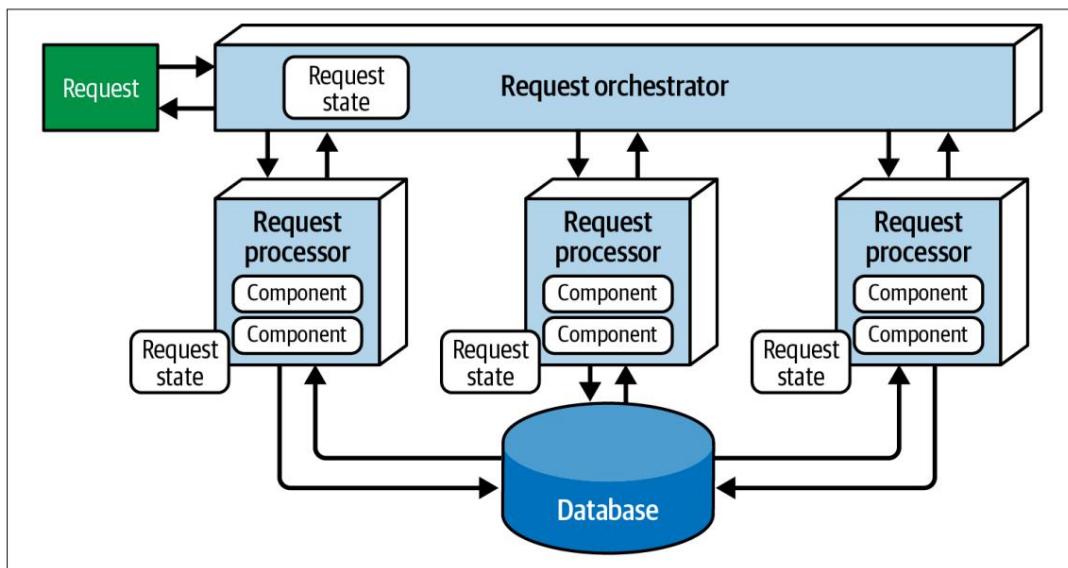


그림 15-1. 요청 기반 모델

위상수학

이벤트 기반 아키텍처는 서비스가 이벤트를 발생시키고 다른 서비스가 해당 이벤트에 응답하는 비동기적 반복-무관 통신 방식을 활용합니다. 이 아키텍처의 네 가지 주요 구성 요소는 시작 이벤트, 이벤트 브로커, 이벤트 처리기(일반적으로 서비스라고 함) 및 파생 이벤트입니다.

시작 이벤트는 전체 이벤트 흐름을 시작하게 하는 이벤트입니다. 이는 온라인 경매에 입찰하는 것과 같은 간단한 이벤트일 수도 있고, 직원이 결혼했을 때 건강 보험 시스템을 업데이트하는 것과 같은 더 복잡한 이벤트일 수도 있습니다. 시작 이벤트는 처리를 위해 이벤트 브로커의 이벤트 채널로 전송됩니다. 단일 이벤트 프로세서는 이벤트 브로커에서 시작 이벤트를 받아 처리를 시작합니다.

이벤트.

최초 이벤트를 수락한 이벤트 프로세서는 해당 이벤트 처리와 관련된 특정 작업(예: 경매 품목 입찰)을 수행한 다음, 파생 이벤트를 이벤트 브로커에 발생시켜 시스템의 나머지 부분에 비동기적으로 자신이 수행한 작업을 알립니다. 다른 이벤트 프로세서는 파생 이벤트에 응답하여 이를 기반으로 특정 처리를 수행한 다음, 새로운 파생 이벤트를 통해 자신이 수행한 작업을 알립니다. 이 과정은 모든 이벤트 프로세서가 유휴 상태가 되고 모든 파생 이벤트가 처리될 때까지 계속됩니다. **그림 15-2**는 이러한 이벤트 처리 흐름을 보여줍니다.

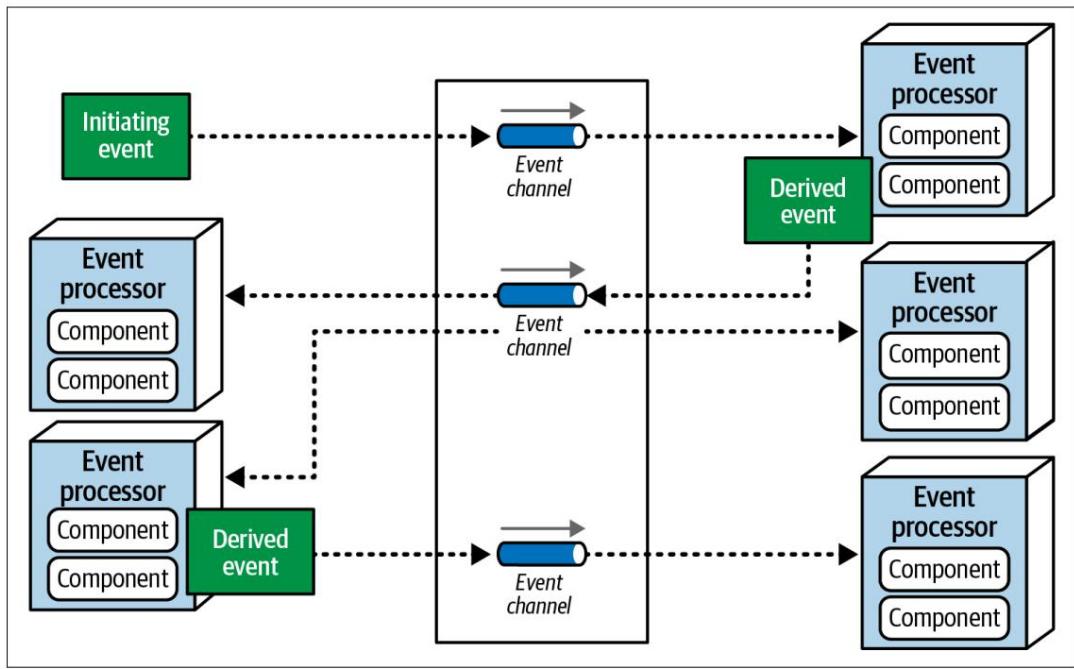


그림 15-2. 이벤트 기반 아키텍처의 기본 토플로지

이벤트 브로커 구성 요소는 일반적으로 연합형(즉, 여러 도메인 기반 클러스터 인스턴스로 구성됨)입니다. 각 연합형 브로커는 해당 도메인의 이벤트 흐름(이벤트 처리를 위한 전체 워크플로) 내에서 사용되는 모든 이벤트 채널(예: 큐 및 토픽)을 포함합니다. 이러한 아키텍처 스타일은 분산되고 비동기적이며, 메시지를 전송한 후 결과를 확인하지 않는 브로드캐스트 방식을 선택하고 있기 때문에 브로커 토플로지는 토픽, 토픽 교환(AMQP(고급 메시지 큐 프로토콜)의 경우) 또는 스트림을 사용하여 발행-구독 메시징 모델을 구현합니다.

EDA 처리의 전반적인 작동 방식을 설명하기 위해 [그림 15-3](#)에 나타낸 일반적인 소매 주문 입력 시스템의 워크플로를 살펴보겠습니다. 이 시스템에서 고객은 상품(예: 이 책)을 주문할 수 있습니다. 이 예에서 주문 접수 이벤트 프로세서는 시작 이벤트(주문하기)를 수신하고, 데이터베이스 테이블에 주문을 삽입한 후 고객에게 주문 ID를 반환합니다. 그런 다음 주문 접수 파생 이벤트를 통해 시스템의 나머지 부분에 주문이 생성되었음을 알립니다. 이 파생 이벤트에는 알림 이벤트 프로세서, 결제 이벤트 프로세서, 재고 이벤트 프로세서 등 세 개의 이벤트 프로세서가 관여하며, 이들은 모두 병렬로 작업을 수행합니다.

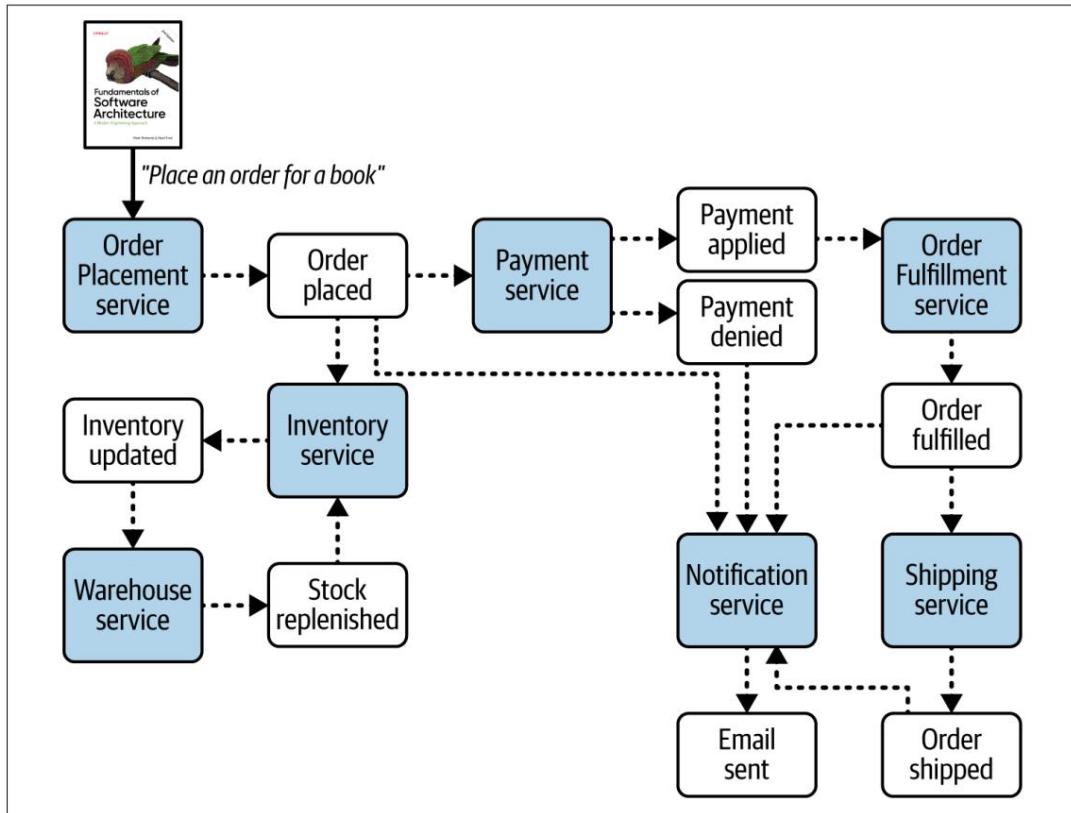


그림 15-3. 이벤트 기반 아키텍처 토플로지의 예

알림 이벤트 처리기는 주문 완료 파생 이벤트를 수신하고 고객에게 주문 세부 정보를 이메일로 전송합니다. 알림 이벤트 처리기가 작업을 수행했으므로 이메일 전송 파생 이벤트를 생성합니다.

하지만 그림 15-3에서 다른 이벤트 프로세서가 해당 파생 이벤트를 수신하지 않는다는 점에 주목 하십시오. 이는 EDA에서 흔히 볼 수 있는 현상이며, 이 방식의 아키텍처 확장성, 즉 향후 이벤트 프로세서가 기존 시스템을 변경하지 않고도 파생 이벤트에 응답할 수 있는 기능을 보여줍니다(235페이지의 "확장 가능한 이벤트 트리거링" 참조).

재고 이벤트 처리기는 주문 완료 파생 이벤트를 수신하고 해당 도서의 재고를 조정합니다. 그런 다음 재고 업데이트 파생 이벤트를 발생시켜 이 작업을 알리고, 이는 창고 이벤트 처리기의 응답을 트리거합니다. 창고 이벤트 처리기는 창고 간의 재고를 관리하고, 재고가 부족해지면 품목을 재주문합니다. 재고가 보충되면 창고 이벤트 처리기는 재고 보충 파생 이벤트를 발생시키고, 재고 이벤트 처리기는 이에 응답하여 현재 재고를 조정합니다.

이 경우 재고 이벤트 처리기는 해당 재고 조정 이벤트를 발생시키지 않습니다. 만약 발생시킨다면, 이는 무한히 반복되는 '독성 이벤트'로 이어질 수 있습니다.



독성 이벤트는 파생 이벤트가 서비스 간에 지속적인 루프 속에서 계속해서 발생하고 응답될 때 발생합니다. 이벤트 기반 아키텍처를 사용할 때 이러한 현상이 빈번하게 발생할 수 있으므로 주의해야 합니다.

결제 이벤트 처리기는 주문 완료 파생 이벤트에 응답하여 고객의 신용카드로 결제 금액을 청구합니다. 그림 15-3은 결제 이벤트 처리기의 동작 결과로 두 가지 파생 이벤트 중 하나가 생성됨을 보여줍니다. 하나는 결제가 적용되었음을 시스템에 알리는 이벤트 (결제 적용됨)이고, 다른 하나는 결제가 거부되었음을 시스템에 알리는 이벤트 (결제 거부됨)입니다. 알림 이벤트 처리기는 결제 거부 파생 이벤트에 관심을 갖는데, 이 경우 고객에게 신용카드 정보를 업데이트하거나 다른 결제 방법을 선택해야 한다는 내용의 이메일을 보내야 하기 때문입니다.

주문 처리 이벤트 프로세서는 결제 적용 파생 이벤트를 수신하고, 작업자에게 품목 위치와 필요한 상자 크기를 알려주는 등 주문 편집 및 포장 과정에서 다양한 자동화 기능을 수행합니다. 이 과정이 완료되면 주문 처리 완료 파생 이벤트를 발생시켜 시스템의 다른 부분에 주문 처리가 완료되었음을 알립니다. 알림 및 배송 이벤트 프로세서 모두 이 파생 이벤트를 수신합니다. 동시에 알림 이벤트 프로세서는 고객에게 주문이 처리되어 배송 준비가 완료되었음을 알리고, 배송 이벤트 프로세서는 배송 방법을 선택하고 주문을 발송한 후 주문 배송 완료 파생 이벤트를 발생시킵니다. 알림 이벤트 프로세서는 또한 주문 배송 완료 파생 이벤트를 수신하고 고객에게 주문이 배송 중임을 알립니다.

모든 이벤트 처리기는 서로 고도로 분리되어 있고 독립적입니다. 이러한 비동기 처리 워크플로를 이해하는 한 가지 방법은 릴레이 경주를 생각해 보는 것입니다. 릴레이 경주에서 주자들은 바통(나무 막대기)을 들고 정해진 거리(예: 1.5km)를 달린 다음 주자에게 바통을 넘겨주고, 다음 주자도 같은 방식으로 마지막 주자가 결승선을 통과할 때까지 계속됩니다.

달리기 선수가 바통을 넘겨주면 경주는 끝난 것이나 다름없으며, 다른 일로 넘어갈 수 있습니다. 이벤트 프로세서도 마찬가지입니다. 이벤트 프로세서가 이벤트를 넘겨받으면 해당 이벤트 처리는 종료되고, 다른 시작 이벤트나 파생 이벤트에 대응할 수 있게 됩니다. 또한, 각 이벤트 프로세서는 다양한 부하 조건이나 백업 상황에 맞춰 독립적으로 확장할 수 있습니다.

스타일 사양

다음 섹션에서는 EDA에 대해 더 자세히 설명하며, 이 복잡한 건축 양식의 고려 사항, 패턴 및 혼합 형태에 대해서도 다룹니다.

이벤트와 메시지의 차이점

이벤트 기반 아키텍처는 이벤트를 활용하여 정보를 전달하고 처리합니다. 하지만 이벤트는 메시지와 정말로 다른 것일까요? 사실, 상당히 다릅니다.

이벤트는 다른 이벤트 처리기들에게 어떤 일이 이미 발생했음을 알리는 역할을 합니다. 예를 들어 "주문을 완료했습니다."와 같습니다. 반면 메시지는 "이 주문에 대한 결제를 적용해 주세요." 또는 "이 주문에 대한 배송 옵션을 제공해 주세요."와 같은 명령이나 질의에 가깝습니다. 이 차이는 미묘하지 않습니다. 이벤트 처리란 이미 발생한 일에 반응하는 것을 의미하는 반면, 메시지는 수행해야 할 작업을 설명합니다. 예시에서 "주문을 완료했습니다."는 메시지와 달리 어떤 처리가 필요한지 명시하지 않으므로 이벤트임이 분명합니다. 이는 EDA의 분리된 특성을 잘 보여줍니다.

이벤트와 메시지의 두 번째 주요 차이점은 이벤트는 일반적으로 수신자의 응답을 필요로 하지 않는 반면, 메시지는 대개 응답을 필요로 한다는 점입니다. 이는 이벤트 처리기 간의 통신 횟수를 줄여주어, 이벤트 처리기들 간의 결합도를 더욱 낮춥니다.

이벤트와 메시지의 또 다른 주요 차이점은 이벤트는 일반적으로 여러 이벤트 처리기에 브로드캐스트되는 반면, 메시지는 거의 항상 하나의 이벤트 처리기에만 전달된다는 점입니다. 간단한 주문 처리 예시에서, 여러 이벤트 처리기가 '주문 생성' 이벤트에 관심을 갖고 이에 응답하는 반면, '결제 적용' 메시지에는 하나의 이벤트 처리기만이 응답합니다. 이벤트는 일반적으로 발행-구독(일대다) 방식의 통신을 사용하는 반면, 메시지는 일반적으로 지점 간(일대일) 방식의 통신을 사용합니다.

이벤트와 메시지의 마지막 차이점은 통신 채널을 나타내는 물리적 아티팩트입니다. 이벤트는 토픽, 스트림 또는 알림 서비스를 사용하여 여러 이벤트 프로세서가 해당 채널을 구독하고 이벤트를 수신 할 수 있도록 합니다.

메시징은 일반적으로 큐 또는 메시징 서비스를 사용하여 특정 이벤트 처리기만이 해당 메시지를 수신하도록 보장합니다.

이벤트 기반 아키텍처는 이름에서 알 수 있듯이 대부분 이벤트를 사용하지만, 다른 이벤트 프로세서에 데이터를 요청하는 경우처럼 메시지를 사용할 수도 있습니다([268페이지의 "데이터 토플로지"](#) 및 [256페이지의 "요청-응답 처리"](#) 참조). 이 장의 뒷부분에서는 메시지를 사용하여 이벤트 처리 순서를 제어하는 중재형 이벤트 기반 아키텍처([258페이지의 "중재형 이벤트 기반 아키텍처"](#) 참조)를 소개합니다.

다음 보기 중 어떤 것이 이벤트이고 어떤 것이 메시지인지 구분할 수 있습니까?

- "어드벤처러스 에어 6557편, 좌회전하여 230도 방향으로 진행하십시오." • "다른 소식으로, 한랭 전선이 이 지역으로 이동해 왔습니다."
- "자, 얘들아, 워크북 145페이지를 펴 봐." • "안녕하세요, 여러분! 회의에 늦어서 죄송합니다."

이것들을 하나씩 살펴보겠습니다.

"어드벤처러스 에어 6557편, 좌측으로 선회하여 230도 방향으로 진행합니다."

이것은 명령(수행해야 할 일)이기 때문에 메시지이며, 다른 여러 조종사들이 메시지를 들을 수도 있지만, 특정 대상(조종사)에게 직접 전달되는 메시지이기 때문입니다.

"다른 소식으로는, 한랭 전선이 이 지역으로 이동해 왔습니다."

이것은 하나의 사건입니다. 여러 사람에게 방송되고, 이미 발생한 일을 설명하며, 뉴스 진행자는 답변을 기대하지 않습니다.
(하지만 답장이 필요 없는 메시지도 있습니다.)

"자, 얘들아, 문제집 145페이지를 펴 봐."

이건 좀 까다롭습니다. 여러 학생에게 방송되는 메시지이긴 하지만, 사실은 메시지입니다. 이미 발생한 일(이벤트)이 아니라, 무언가를하도록 지시하는 명령이기 때문입니다. 이는 이벤트와 메시지의 중요한 차이점을 보여줍니다. 발행-구독 채널을 통해 명령(예: 워크북 145페이지로 이동)을 방송한다고 해서 그것이 이벤트가 되는 것은 아닙니다.

안녕하세요, 여러분! 회의에 늦어서 죄송합니다.

이건 하나의 사건입니다. 왜냐하면 그 사람이 회의에 늦었기 때문입니다! 게다가 이 일은 여러 사람에게 방송되고 있고, 답변은 기대하지 않습니다.

파생 이벤트

파생 이벤트는 EDA에서 매우 중요하고 필수적인 부분입니다. 이러한 이벤트는 최초 이벤트가 수신된 후 이벤트 프로세서에 의해 생성되고 트리가됩니다. 이벤트 프로세서는 처리 내용에 따라 하나 이상의 파생 이벤트를 트리거 할 수 있습니다.

그림 15-3 에서 결제 이벤트 처리기가 고객의 신용카드로 구매 대금을 청구할 때 발생하는 파생 이벤트를 살펴보겠습니다. 그림 15-4 에서 볼 수 있듯이, 신용카드 청구 과정에는 사기 가능성 확인(사기 탐지 이벤트 처리기에서 처리)과 신용카드 잔액 확인(신용 한도 이벤트 처리기에서 처리)이 포함됩니다. EDA는 단일 이벤트 (신용카드 청구)를 활용하여 이 두 가지 작업을 동시에 수행할 수 있습니다.

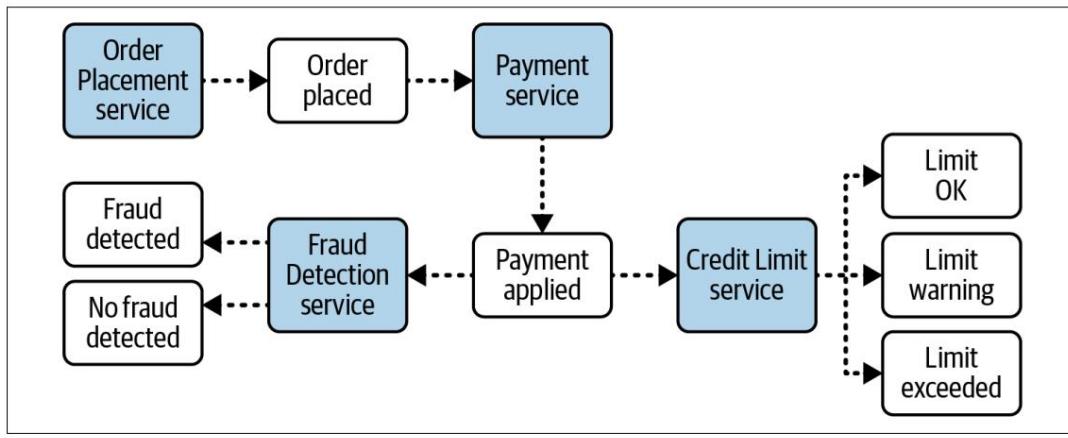


그림 15-4. 파생 이벤트는 시작 이벤트에 대한 응답으로 생성됩니다.

이 단일 동작이 얼마나 많은 파생 이벤트를 생성하는지 주목하십시오. 사기 탐지 이벤트 처리기에서는 이 동작으로 인해 두 가지 파생 이벤트가 발생합니다. 하나는 처리기가 사기를 탐지한 경우이고, 다른 하나는 사기가 탐지되지 않은 경우입니다. 이 두 가지 파생 이벤트는 모두 필요합니다. 왜냐하면 사기 탐지 결과에 따라 다른 이벤트 처리기가 추가 조치를 취할 수 있기 때문입니다.

이제 신용 한도 이벤트 처리기에서 파생된 이벤트를 살펴보겠습니다. 먼저, '한도 충족' 파생 이벤트는 시스템의 나머지 부분에 해당 구매에 대한 위험이 없으며 고객에게 충분한 신용 한도가 있음을 알려줍니다. 실제로 이 특정 이벤트는 이벤트 페이로드에 고객에게 남은 신용 한도 금액을 저장할 수도 있는데, 이는 하위 이벤트 처리기에 유용할 수 있습니다.

둘째, 카드 잔액이 신용 한도에 근접했음을 경고하는 '한도 경고' 파생 이벤트는 다른 하위 이벤트 처리기, 예를 들어 고객에게 신용 한도에 가까워졌음을 알리는 '알림' 이벤트 처리기에서 중요하게 다뤄질 수 있습니다. 마지막으로, '한도 초과' 파생 이벤트는 알림, 구매 거절, 그리고 고객의 신용 한도를 자동으로 연장하여 구매를 허용하는 마케팅 기반 '신용 한도 연장' 이벤트 처리기 등 여러 이벤트 처리기에서 중요하게 다뤄집니다.

이는 이벤트 프로세서에서 둘 이상의 파생 이벤트가 트리거될 수 있음을 보여줍니다. 그러나 처리 장치가 너무 많은 세분화된 이벤트를 내보내는 '모기 떼' 안티패턴에 빠지지 않도록 주의해야 합니다 (246페이지의 "모기 떼 안티패턴" 참조).

확장 가능한 이벤트를 처리할 때, EDA에서는 일반

적으로 각 이벤트 프로세서가 다른 이벤트 프로세서가 해당 작업에 관심이 있는지 여부와 관계없이 자신이 수행한 작업을 시스템의 나머지 부분에 알리는 것이 좋습니다.

이벤트 처리기가 해당 이벤트에 관심을 갖거나 응답하지 않더라도, 해당 이벤트를 확장 가능한 파생 이벤트라고 부릅니다. 이는 해당 이벤트 처리에 추가적인 기능이 필요한 경우를 대비하여 내장된 "후크"를 제공함으로써 아키텍처 확장성을 지원하기 때문입니다. 예를 들어, 복잡한 이벤트 처리 과정(그림 15-5 참조)의 일부로, 알림 이벤트 처리기가 고객에게 특정 조치를 알리는 이메일을 생성하여 전송한다고 가정해 보겠습니다. 그런 다음, 이메일 전송 완료라는 새로운 파생 이벤트를 통해 시스템의 나머지 부분에 이메일 전송 사실을 알립니다.

현재 다른 이벤트 프로세서가 해당 이벤트를 수신하거나 응답하지 않으므로 메시지는 단순히 사라지거나(이벤트 스트리밍의 경우 무시됨) 합니다. 이는 리소스 낭비처럼 보일 수 있지만 그렇지 않습니다. 예를 들어, 회사에서 고객에게 보낸 모든 이메일을 분석하기로 결정했다고 가정해 보겠습니다. 이메일 정보는 이미 '이메일 전송 됨' 파생 이벤트를 통해 얻을 수 있으므로, 팀은 다른 이벤트 프로세서를 변경하지 않고도 최소한의 노력으로 전체 시스템에 새로운 이메일 분석기 이벤트 프로세서를 추가할 수 있습니다.

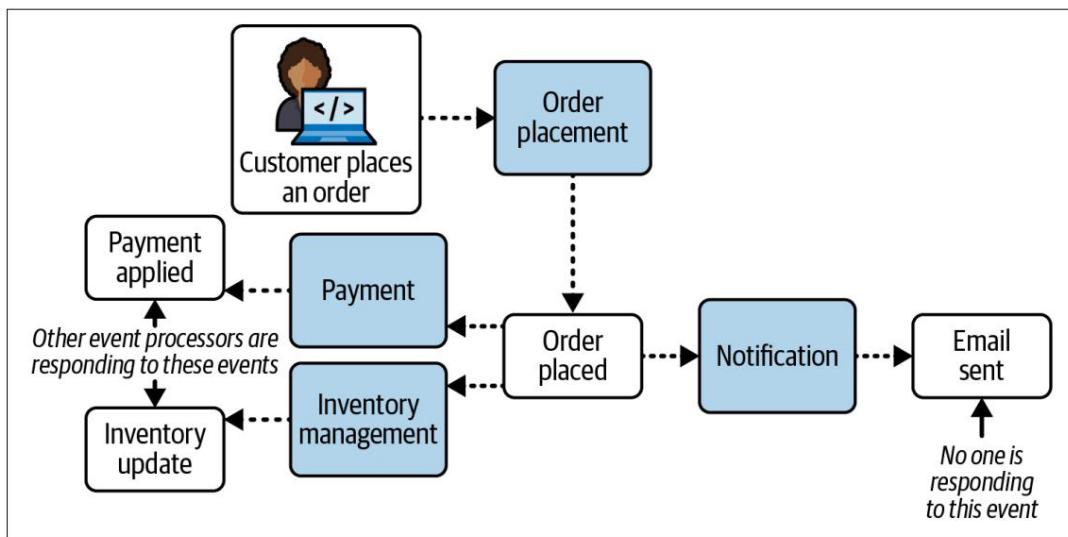


그림 15-5. 알림 이벤트가 전송되었지만 무시되고 사용되지 않았습니다.

비동기 기능 이벤트 기반 아키텍처

타일은 응답이 필요 없는 처리(re-and-forget)와 이벤트 소비자의 응답이 필요한 요청/응답 처리(request-reply processing, 256페이지의 "요청-응답 처리" 참조) 모두에 주로 비동기 통신을 사용한다는 고유한 특징을 가지고 있습니다.

의사소통은 시스템의 전반적인 반응성을 향상시키는 강력한 기술이 될 수 있습니다.

그림 15-6에서 사용자는 웹사이트에 제품 리뷰를 게시하고 있습니다. 이 예시에서 댓글 서비스는 해당 댓글을 검증하고 게시하는 데 3,000밀리초가 걸립니다. 댓글은 여러 구문 분석 엔진을 거쳐야 합니다. 먼저 허용되지 않는 단어가 있는지 확인하고, "생각이 느린 사람"이나 "명확하게 생각할 수 없는 사람"과 같은 욕설이 포함되어 있지 않은지 확인한 후, 마지막으로 댓글이 제품에 대한 내용인지(예를 들어 단순히 정치적인 비난이 아닌지) 문맥을 확인합니다.

그림 15-6의 맨 위 경로는 동기식 RESTful 호출을 사용하여 댓글을 게시합니다.

즉, 서비스가 게시물을 수신하는 데 네트워크 지연 시간 50ms, 댓글을 검증하고 게시하는 데 3,000ms, 그리고 사용자에게 댓글이 게시되었다는 알림을 보내는 데 50ms가 소요됩니다. 따라서 사용자 입장에서 댓글을 게시하는 데 걸리는 총 시간은 3,100ms입니다. 이제 비동기 메시징을 사용하는 아래쪽 경로를 살펴보겠습니다. 이 경우 사용자의 총 게시 시간은 3,100ms가 아닌 25ms에 불과합니다. 시스템이 댓글을 수신하는 데 25ms, 게시하는 데 3,000ms가 걸리므로 총 3,025ms가 소요되지만, 최종 사용자 입장에서는 시스템이 댓글을 수락했다는 응답을 받기까지 25ms밖에 걸리지 않습니다(실제로 게시되기는 전입니다).

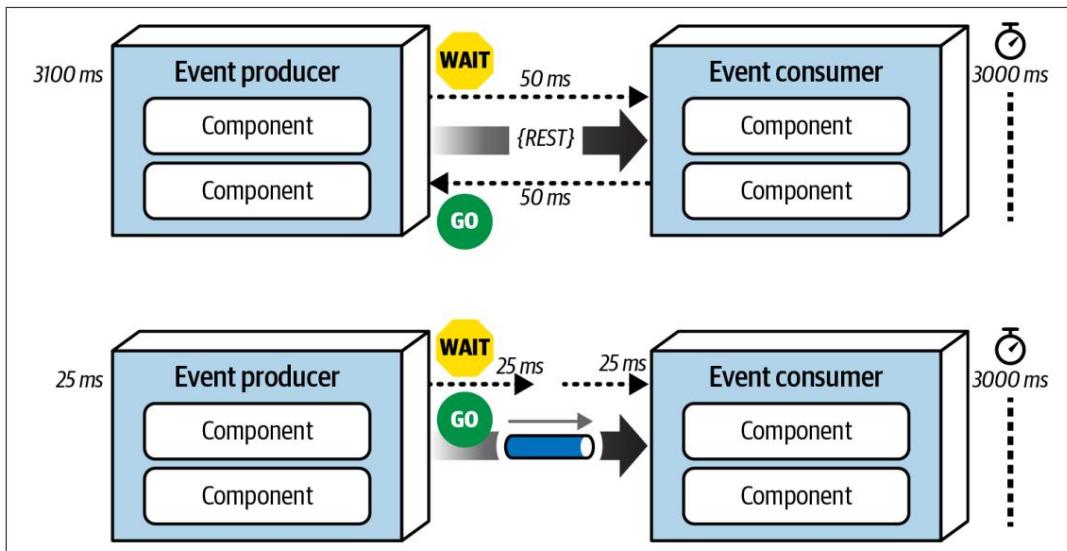


그림 15-6. 동기식 통신과 비동기식 통신 비교

3,100ms와 25ms의 응답 시간 차이는 엄청납니다. 하지만 한 가지 주의할 점이 있습니다. 동기식 상위 경로에서는 사용자가 댓글이 게시되었다는 것을 보장받을 수 있습니다. 반면 비동기식 하위 경로에서는 게시가 완료되었다는 확인만 받을 뿐, 결국 게시될 것이라는 약속만 받게 됩니다.

사용자의 댓글에 욕설이 포함되어 시스템에서 거부되면 어떻게 될까요? 최종 사용자에게 알림을 보낼 방법이 없는데, 혹시 있을까요? 사용자가 등록해야 하는 경우라면요?

웹사이트를 통해 댓글을 게시할 때, 시스템은 해당 댓글에 문제가 있음을 알리고 수정 방법을 제안하는 메시지를 사용자에게 보낼 수 있습니다.

이 예시는 응답성(사용자에게 정보를 제공하는 데 걸리는 시간)과 성능(댓글을 데이터베이스에 삽입하는 데 걸리는 시간)의 차이를 보여줍니다. 사용자가 (확인이나 감사 메시지 외에) 다른 정보를 필요로 하지 않는다면, 왜 기다리게 해야 할까요?

응답성은 사용자가 작업을 수락했으며 곧 처리될 것임을 알리는 데 중점을 두는 반면, 성능은 전체 프로세스의 속도를 높이는 데 중점을 둡니다. 비동기 처리 방식에서 아키텍트는 댓글 서비스가 댓글을 처리하는 방식(응답성)을 최적화하기 위한 어떤 조치도 취하지 않았습니다. 만약 아키텍트가 댓글 서비스를 최적화하는 데 시간을 투자했다면, 예를 들어 텍스트 및 문법 분석 엔진을 병렬로 실행하고, 캐싱 및 기타 유사한 기술을 활용하면서도 동기 통신을 유지하는 방식으로 최적화했다면, 전반적인 성능 향상에 기여할 수 있었을 것입니다.

방금은 간단한 예시였습니다. 좀 더 복잡한 예시는 어떨까요? 이번에는 사용자가 비동기적으로 주식을 매수하는 온라인 주식 거래를 살펴보겠습니다. 만약 사용자에게 오류를 알릴 방법이 없다면 어떻게 될까요?

비동기 통신은 응답성을 크게 향상시키지만, 오류 처리는 큰 문제입니다. 오류 조건을 처리하는 어려움은 이러한 아키텍처 스타일의 복잡성을 가중시킵니다. [249페이지의 "오류 처리"에서는](#) 오류 처리 문제를 해결하기 위한 반응형 아키텍처 패턴인 워크플로우 이벤트 패턴을 설명합니다.

비동기 통신은 응답성이 뛰어날 뿐만 아니라, 동적 분리를 효과적으로 수행하고 두 아키텍처 양자가 동기 통신을 통해 통신할 때 발생하는 안티패턴인 동적 양자 얹힘(Dynamic Quantum Entanglement)을 방지합니다. ([7장](#)에서 아키텍처 양자란 시스템의 나머지 부분과 독립적으로 배포될 수 있고 동기적 동적 결합을 통해 연결되는 시스템의 일부이며, 아키텍처 특성은 양자 수준에서 존재한다는 것을 배웠습니다.) 이 두 아키텍처 양자는 서로 의존하게 되므로 본질적으로 얹히게 됩니다. 이러한 의존성으로 인해 두 양자는 하나의 아키텍처 양자로 통합됩니다. 비동기 통신은 이러한 동적 의존성을 제거함으로써 아키텍처 양자 간의 얹힘을 해소하는 데 도움을 줄 수 있습니다.

이 중요한 점을 설명하기 위해 그림 15-7의 두 시스템을 살펴보겠습니다. 이 예에서 포트폴리오 관리 시스템은 주식을 매수하는 거래 주문을 생성합니다.

이 시스템은 거래 주문을 거래 주문 시스템으로 동기적으로 전송하고, 거래 주문 시스템은 규정 준수 검사를 수행하여 거래 주문을 생성합니다. 두 시스템 간의 통신이 동기적이기 때문에 포트폴리오 관리 시스템은 거래 주문 시스템으로부터 거래 확인 번호를 받을 때까지 반드시 대기해야 합니다. 이 두 시스템은 서로 얹혀 하나의 아키텍처 양자를 형성하게 됩니다.

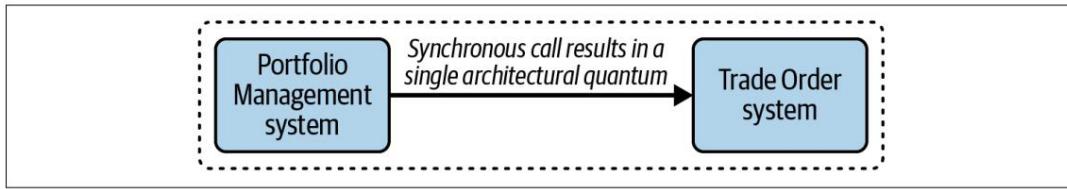


그림 15-7. 이러한 시스템은 동기식으로 인해 단일 아키텍처 양자를 형성합니다.
동적 결합

이러한 상호 연관성의 중요성은 아키텍처적 특성이 이제 두 시스템 사이에 존재한다는 점입니다. 거래 주문 시스템을 사용할 수 없거나 응답하지 않으면 포트폴리오 관리 시스템은 거래 주문을 제출할 수 없습니다.

이는 응답성을 저하시킵니다. 거래 주문 시스템이 느리면 포트폴리오 관리 시스템도 느려집니다. 확장성 또한 문제가 되는데, 포트폴리오 관리 시스템에 확장이 필요하면 거래 주문 시스템도 함께 확장해야 하기 때문입니다. 만약 거래 주문 시스템에 확장이 필요하지 않거나 확장할 수 없다면, 포트폴리오 관리 시스템도 필요한 만큼 확장할 수 없습니다.

건축가는 [그림 15-8](#)에 나타낸 것처럼 두 시스템 간의 동기 호출을 비동기 호출로 대체함으로써 이러한 아키텍처 양자를 분리할 수 있습니다.

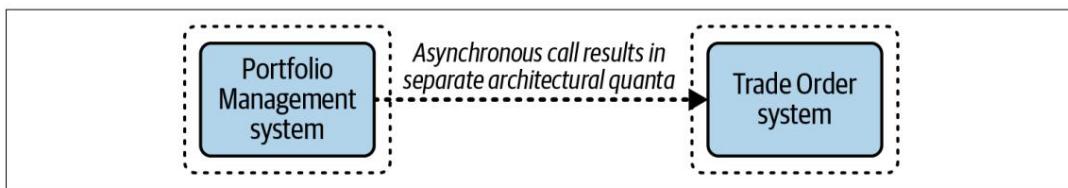


그림 15-8. 이러한 시스템들은 비동기적인 특성으로 인해 서로 다른 아키텍처적 범주를 형성합니다.
동적 결합

포트폴리오 관리 시스템은 비동기 통신을 사용하여 큐 또는 다른 비동기 방식을 통해 거래 주문을 전송할 수 있으므로 거래 주문 시스템이 거래 주문을 생성할 때까지 기다릴 필요가 없습니다. 거래 주문 시스템이 규정 준수 검사를 수행하고 거래 주문을 생성하면 별도의 비동기 채널을 통해 포트폴리오 관리 시스템으로 확인 번호를 전송할 수 있습니다. 두 시스템 간의 동적 결합에서 이러한 의존성을 제거함으로써 두 시스템을 완전히 독립적인 아키텍처 영역으로 구성할 수 있습니다. 거래 주문 시스템을 사용할 수 없거나 응답하지 않는 경우에도 포트폴리오 관리 시스템은 거래 주문을 발행할 수 있으며, 주문이 생성되고 확인 번호가 전송될 것이라는 것을 알고 있습니다.

방송 기능

EDA의 또 다른 고유한 특징은 다른 처리 장치(있는 경우)가 해당 이벤트를 수신하는지 또는 응답으로 어떤 처리를 수행할지 알지 못한 채 이벤트를 브로드캐스트할 수 있다는 것입니다. [그림 15-9](#)에서 볼 수 있듯이, 이는 이벤트 프로세서들을 서로 동적으로 분리합니다.

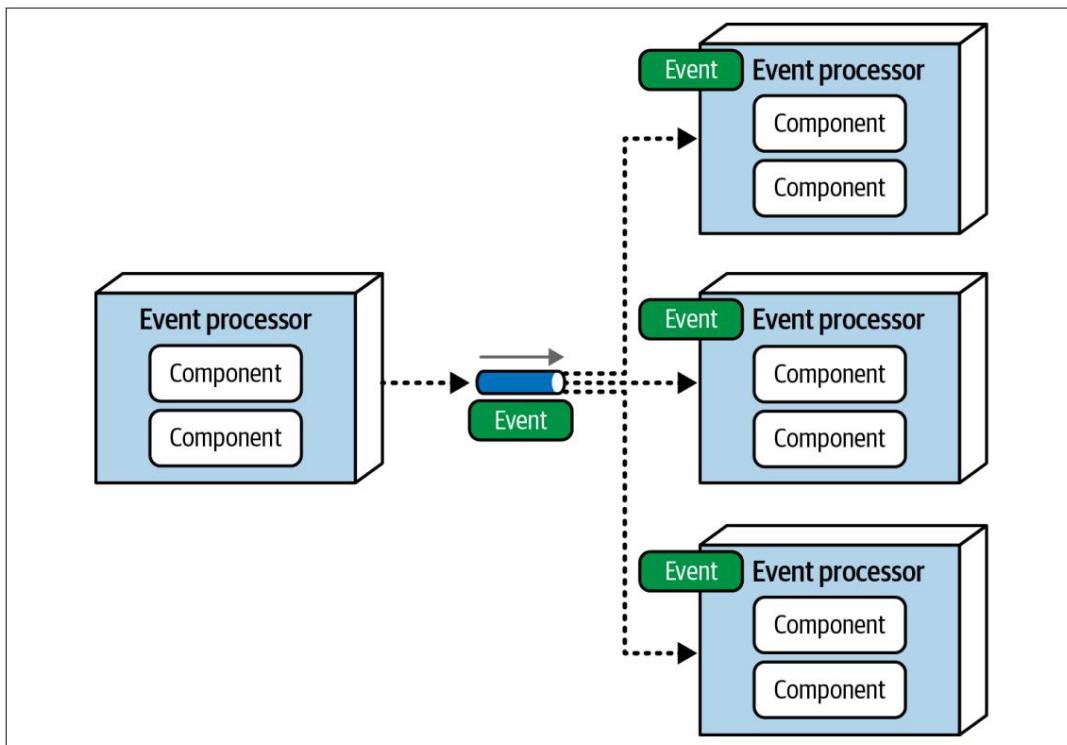


그림 15-9. 다른 이벤트 처리기로 이벤트 브로드캐스팅

브로드캐스트 기능은 최종 일관성 및 복합 이벤트 처리(CEP)를 포함한 많은 패턴에서 필수적인 부분입니다. 예를 들어, 주식 시장에서 거래되는 금융 상품의 가격은 빈번하게 변동합니다. 새로운 티커 가격(특정 주식의 현재 가격)이 발표될 때마다 여러 이벤트 프로세서가 새로운 가격에 반응할 수 있습니다(예: 거래 분석, 주식 매매). 그러나 최신 가격을 발표하는 이벤트 프로세서는 해당 정보가 어떻게 사용될지에 대한 정보 없이 단순히 가격만 브로드캐스트합니다. 이는 하나의 이벤트 프로세서가 다른 이벤트 프로세서의 동작에 대한 정보(또는 의존성)를 갖지 않는다는 점에서 의미론적 분리라고 합니다.

이벤트 페이로드 이

벤트에 포함된 정보를 페이로드라고 합니다. 페이로드는 매우 다양할 수 있습니다. 이벤트 페이로드는 단순한 키-값 쌍일 수도 있고, 하위 프로세스 처리에 필요한 모든 정보를 포함할 수도 있습니다. 기본적으로 데이터 기반 페이로드와 키 기반 페이로드 두 가지 유형이 있습니다. 아키텍트는 시스템에서 발생하는 각 이벤트 유형에 어떤 옵션이 가장 적합한지 신중하게 분석해야 합니다. 이 섹션에서는 두 가지 페이로드 유형과 각각의 장단점을 설명합니다.

데이터 기반 이벤트 페이로드

란 처리에 필요한 모든 정보를 전송하는 이벤트 페이로드입니다. 그림 15-10 의 예에서 고객이 주문을 하면, 먼저 주문 접수 이벤트 프로세서가 전체 주문 정보를 데이터베이스(기록 시스템)에 삽입합니다. 그런 다음 주문 세부 정보(이 경우 45개의 속성, 총 500KB의 메모리)를 포함하는 order_placed 라는 이벤트를 브로드캐스트합니다. 결제 이벤트 프로세서는 이벤트 페이로드에서 주문 ID, 고객 정보, 총 주문 금액을 가져와 결제를 처리합니다. 동시에 재고 관리 이벤트 프로세서는 이벤트 페이로드에서 품목 ID와 수량을 사용하여 구매한 품목의 현재 재고를 조정합니다.

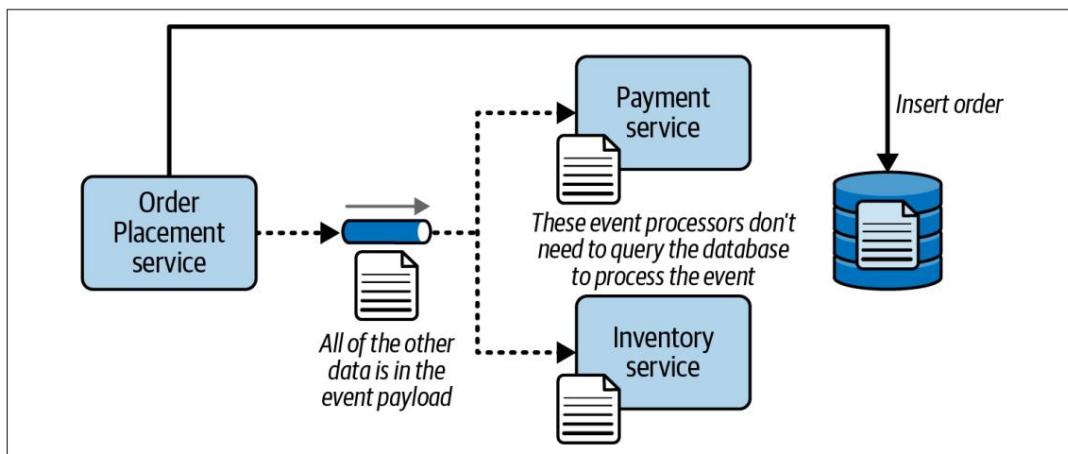


그림 15-10. 데이터 기반 이벤트 페이로드에는 처리에 필요한 모든 데이터가 포함되어 있습니다.

결제 및 재고 관리 이벤트 프로세서는 주문 정보를 가져오기 위해 데이터베이스를 쿼리할 필요가 없었습니다. 데이터가 이미 이벤트 페이로드에 포함되어 있었기 때문입니다. 이는 데이터 기반 이벤트 페이로드를 사용하는 가장 큰 장점 중 하나입니다. 이벤트 프로세서가 데이터베이스를 쿼리하는 횟수가 적을수록 성능, 응답성 및 확장성이 향상됩니다. 또한, EDA의 고도로 동적이고 분산된 특성을 고려할 때, 주문 접수 이벤트 프로세서는 다른 어떤 이벤트 프로세서가 해당 이벤트에 응답하는지 또는 어떤 데이터가 필요한지 알지 못할 수도 있습니다.

처리를 위해서입니다. 이벤트 페이로드에 모든 정보를 전송하면 각 응답 이벤트 처리기가 처리를 수행하는 데 필요한 정보를 확보할 수 있습니다. 특히 엄격한 경계 컨텍스트, 도메인 기반 또는 서비스별 데이터베이스 방식의 데이터 토플로지에서는 이벤트 처리기가 주문 정보가 포함된 데이터베이스에 접근하지 못할 수도 있습니다([268페이지의 "데이터 토플로지" 참조](#)).

이러한 장점들은 더욱 반응성이 뛰어나고 확장 가능하며 유연한 시스템을 구축하는 방법을 보여주지만, 데이터 기반 페이로드에는 몇 가지 단점도 있습니다. 첫째, 여러 시스템에 데이터가 저장될 경우 데이터 일관성과 무결성을 유지하기가 더 어렵습니다. 모든 주문 정보가 데이터베이스와 시스템에서 발생하는 이벤트에 모두 저장되어 있기 때문에, 특히 처리 중에 주문 정보가 업데이트될 경우 정보가 쉽게 동기화되지 않을 수 있습니다.

예를 들어, 고객이 100개의 상품을 주문했지만 실제로는 1개만 주문하려고 했고, 주문 제출 직후에 실수를 깨달았다고 가정해 보겠습니다. 또는 고객이 주문 직후에 잘못된 배송 주소를 입력했다는 사실을 알게 된 경우도 있습니다(저자들도 이런 경험을 여러 번 했습니다). 이러한 상황에서 고객은 즉시 주문 정보를 정확한 정보로 수정합니다. 유일한 기록 시스템인 데이터베이스에는 수정된 값이 저장되지만, 이전 값을 포함하는 일부 이벤트는 즉시 처리되지 않을 수 있습니다. 즉, 아직 처리 중인 이전의 잘못된 값이 새로운 올바른 값 대신 사용될 수 있다는 의미입니다. 이 시나리오를 더욱 복잡하게 만드는 것은 EDA에서 이벤트 발생 시점을 제어하기가 매우 어렵다는 점입니다. 따라서 새로운 값이 이전 값보다 먼저 처리될 가능성이 있습니다. 이는 다른 이벤트 프로세서가 이전의 잘못된 값을 사용하는 경우, 해당 값이 새로운 올바른 값 위에 겹쳐질 수 있음을 의미합니다.

데이터 기반 이벤트 페이로드의 두 번째 주요 단점은 계약 관리 및 버전 관리에 관한 것입니다. 이 시스템에서 주문은 45개의 속성을 가지고 있다는 것을 알고 있습니다. 이 모든 정보가 이벤트 페이로드에 포함되어 있기 때문에 이벤트에는 일종의 계약, 즉 전송되는 데이터를 구조화하는 방법이 필요합니다. 이제 아키텍트는 수많은 결정에 직면하게 됩니다. 페이로드 유형은 JSON 객체로 할 것인가? XML 객체로 할 것인가? 계약은 엄격해야 할까, 아니면 유연해야 할까? (엄격한 계약은 JSON 스키마, GraphQL 사양 또는 클래스 정의와 같은 스키마 또는 객체 정의를 사용하는 반면, 유연한 계약은 간단한 JSON 이름-값 쌍을 사용할 수 있습니다.)

이러한 결정 각각에는 많은 절충점이 따르며, 각각은 이벤트 처리기 간에 긴밀한 정적 결합을 형성합니다.

그리고 버전 관리가 있습니다. 엄격한 계약의 경우, 아키텍트나 개발자는 이벤트 헤더에 [벤더 MIME 유형](#)을 사용하여 버전 번호를 지정할 수 있습니다. 이는 시스템의 민첩성을 높이고 하위 호환성을 유지하는 데 도움이 됩니다(다른 이벤트 프로세서와의 호환성 문제 방지). 그러나 모든 이벤트 프로세서는 동일한 버전 관리 로직을 활용해야 하므로 강력한 거버넌스가 필요합니다. 이벤트 프로세서가 엄격한 계약의 페이로드에 응답할 때 계약 버전을 무시하는 경우, 해당 버전을 변경하면 문제가 발생할 수 있습니다.

해당 스키마는 이벤트 프로세서의 오류를 유발할 가능성이 높습니다. 또한 EDA와 같이 고도로 분리된 비동기 아키텍처에서는 버전 정보 교환 및 사용 중단 전략을 구현하기가 매우 어렵습니다. 이러한 모든 요소로 인해 데이터 기반 이벤트 페이로드는 다소 취약해집니다.

데이터 기반 이벤트 페이로드는 스템프 결합(stamp coupling) 문제에 직면할 수 있습니다. 스템프 결합은 여러 모듈(이 경우 이벤트 프로세서)이 공통 데이터 구조를 공유하지만, 그 구조의 일부(그리고 많은 경우 서로 다른 부분)만 사용하는 정적 결합의 한 형태입니다. 이러한 상황이 발생하면 공통 데이터 구조를 변경해야 할 경우, 해당 데이터와 관련이 없는 이벤트 프로세서까지 포함하여 다른 이벤트 프로세서도 변경해야 할 수 있습니다.

그림 15-11은 스템프 결합 방식과 아키텍처에 미치는 부정적인 영향을 보여줍니다. 이 예에서 주문 접수 이벤트 처리기는 주문에 대한 모든 정보를 담고 있는 45개의 속성으로 구성된 `order_placed` 이벤트를 500KB 크기로 전송합니다. 재고 이벤트 처리기는 `order_created` 이벤트에 응답 하지만, `item_id` 와 `quantity`라는 두 가지 속성만 필요로 하며, 총 크기는 30바이트에 불과합니다. 이 예에서 페이로드를 변경하는 경우, 예를 들어 주소 줄 속성을 제거하면 해당 필드에 관심이 없더라도 재고 이벤트 처리기에 영향을 미치게 됩니다.

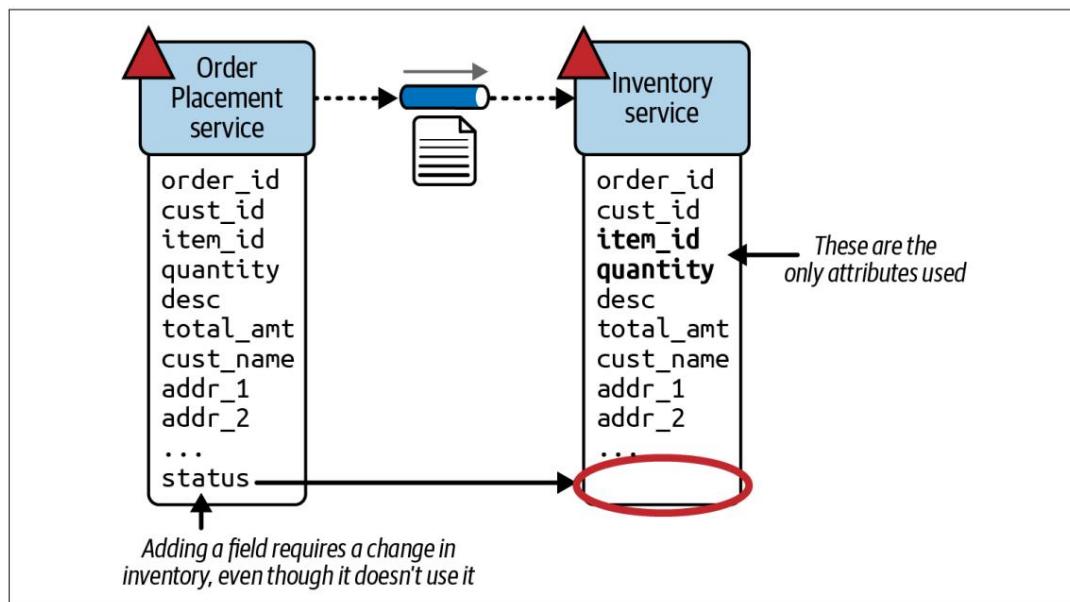


그림 15-11. 다른 서비스가 전송된 데이터의 일부만 필요로 하는 스템프 결합의 예

이 예시에서 엄격한 계약을 사용하는 계약 버전 관리는 인벤토리 이벤트 프로세서가 제대로 작동하지 않을 위험을 완화하는 데 도움이 되지만, 계약 버전이 더 이상 사용되지 않거나 계약 변경으로 인해 호환성이 깨지는 시점이 오면 개발자는 다시 테스트하고 재배포해야 합니다.

스탬프 결합에서 흔히 간과되는 문제점 중 하나는 대역폭 활용입니다.

분산 컴퓨팅 의 세 번째 오류는 "대역폭은 무한하다"는 것입니다. 물론 그렇지 않습니다. 실제로 대부분의 클라우드 기반 환경에서 대역폭은 비용의 핵심 요소입니다. 그림 15-11의 예시로 돌아가서, 데이터 기반 결제 시스템을 생각해 보면, 고객이 초당 500건의 주문을 할 경우, 500KB 크기의 이벤트 하나를 재고 이벤트 처리기로 전송하는 데 초당 25만 KB의 대역폭이 소모됩니다.

하지만 시스템이 실제로 필요한 30바이트의 데이터만 전송하는 경우, 이벤트는 초당 15KB의 대역폭만 사용하게 됩니다. 이는 데이터 기반 이벤트 페이로드를 사용할 때 반드시 조사해 볼 만한 엄청난 차이입니다.

아키텍트들이 스탬프 결합을 제한하는 이유 중 하나는 각 메시지 소비자가 처리에 필요한 데이터만 포함하는 자체 계약을 갖는 **소비자 중심 계약**을 활용하기 위해서입니다. 그러나 EDA의 브로드캐스트 기능과 시스템이 어떤 이벤트 프로세서가 이벤트에 응답할지 항상 알 수 없다는 점 때문에 이벤트 중심 아키텍처에서 이벤트를 사용하는 소비자 중심 계약을 활용하기는 어렵습니다. 이러한 이유(그리고 데이터 기반 이벤트 페이로드의 다른 단점을 해결하기 위해)로 많은 아키텍트들이 키 기반 이벤트 페이로드를 사용합니다.

키 기반 이벤트 페이로드란 이

벤트의 컨텍스트를 식별하는 키(예: 주문 ID 또는 고객 ID)만 포함하는 이벤트 페이로드입니다. 키 기반 이벤트 페이로드를 사용하는 경우, 이벤트에 응답하는 이벤트 프로세서는 이벤트 처리에 필요한 정보를 검색하기 위해 데이터베이스를 쿼리해야 합니다.

고객이 주문을 하면 주문 처리 이벤트 프로세서가 해당 주문을 데이터베이스에 삽입하고 `order_placed`라는 키 기반 이벤트를 발생시킵니다. 이 이벤트에는 주문 ID가 간단한 JSON 형식으로 포함된 단일 키 값이 있습니다.

```
{ "order_id": "123" }
```

키 기반 이벤트 페이로드의 주요 단점 중 하나는 이벤트에 응답하는 각 이벤트 프로세서가 주문 처리에 필요한 정보를 얻기 위해 데이터베이스를 쿼리해야 한다는 것입니다. 예를 들어, 결제 이벤트 프로세서가 이벤트에 응답할 때 결제 처리에 필요한 주문 정보를 얻기 위해 데이터베이스를 쿼리해야 합니다. 재고 이벤트 프로세서도 동시에 이벤트에 응답하므로 품목 ID와 수량을 검색하기 위해 데이터베이스를 쿼리해야 합니다. 이는 응답성, 성능 및 확장성을 저해할 수 있으며, 특히 이벤트 기반 아키텍처와 같이 고도로 병렬적이고 비동기적인 아키텍처에서 데이터베이스에 과부하를 초래할 수 있습니다. (이러한 위험을 완화하는 방법은 268페이지의 "데이터 토플로지"를 참조하십시오.) 또한 필요한 데이터에 쉽게 접근할 수 없는 경우(예: 다른 이벤트 프로세서의 경계 컨텍스트 내에 있는 경우) 키 기반 이벤트 페이로드는 문제가 될 수 있습니다. 그림 15-12는 이러한 문제를 보여줍니다.

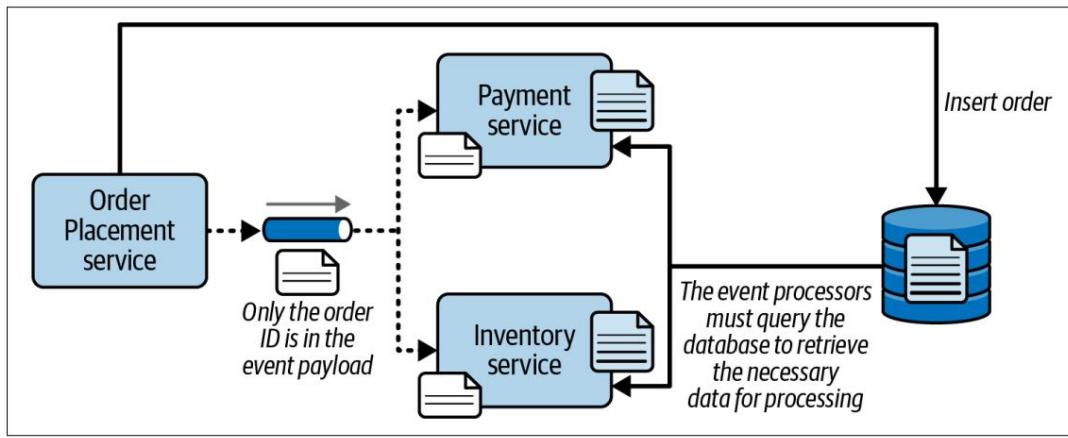


그림 15-12. 키 기반 이벤트 페이로드의 경우 컨텍스트 키만 포함됩니다.

이벤트

하지만 키 기반 이벤트 페이로드를 사용하면 성능 및 확장성 문제를 상쇄할 수 있는 여러 장점이 있습니다. 첫 번째 주요 장점은 단일 기록 시스템 덕분에 전반적인 데이터 일관성과 데이터 무결성이 향상된다는 것입니다. 이벤트 관련 데이터가 데이터베이스 한 곳에만 저장되므로 키 기반 이벤트 페이로드는 데이터 기반 이벤트 페이로드보다 이벤트 처리 중 데이터 변경 사항을 훨씬 쉽게 처리할 수 있습니다.

두 번째 주요 장점은 키 기반 이벤트 페이로드의 계약이 매우 단순하고 변경되는 경우가 드물기 때문에 아키텍트가 일반적으로 스키마가 없는 느슨한 JSON 또는 XML을 통해 구현한다는 점입니다. 따라서 키 기반 이벤트 페이로드는 데이터 기반 이벤트 페이로드에서 흔히 발생하는 계약 변경 관리, 버전 관리, 커뮤니케이션 및 사용 중단 전략과 관련된 문제를 겪지 않습니다.

키 기반 이벤트 페이로드의 또 다른 장점은 데이터 기반 이벤트 페이로드처럼 스탬프 결합 및 대역 폭 문제를 일으키지 않는다는 것입니다. 이벤트와 관련된 불투명한 데이터가 없기 때문에 계약은 간단하고 크기가 작으며 최소한의 대역폭만 사용합니다. 따라서 네트워크 및 메시지 브로커 관점에서 데이터 기반 이벤트 페이로드보다 더 빠른 성능을 보이는 경향이 있습니다.

데이터 기반 이벤트

페이로드와 키 기반 데이터 페이로드 중 하나를 선택하려면 신중한 장단점 분석이 필요합니다. 모든 것을 다 사용하거나 아무것도 사용하지 않는 식의 선택이 아니라, 각 이벤트 유형에 따라 다른 페이로드 유형을 사용할 수 있다는 점을 기억하십시오. 표 15-1은 데이터 기반 및 키 기반 이벤트 페이로드와 관련된 장단점을 요약합니다.

표 15-1. 데이터 기반 이벤트 페이로드와 키 기반 이벤트 페이로드 비교

기준	데이터 기반 페이로드	키 기반 페이로드
성능 및 확장성 우수	나쁜	
계약 관리	나쁜	좋은
스탬프 커플링	나쁜	좋은
대역폭 활용	나쁜	좋은
데이터베이스 접근 제한됨 양호		나쁜
전반적인 시스템 취약성	나쁜	좋은

이 두 옵션 간의 전반적인 절충점은 결국 확장성으로 귀결된다는 점에 주목하십시오.

성능과 계약 관리 및 대역폭 활용도를 비교합니다. 어떤 것이 맞는지 물어보세요.

각 특정 이벤트에 따라 하나가 더 중요합니다. 일부 이벤트 처리에는 두 가지 이상의 요소가 필요합니다.

극도로 높은 규모와 성능을 요구하는 경우, 데이터 기반 이벤트 페이로드가 사용됩니다.

더 나은 선택이 될 것입니다. 일부 이벤트 처리 데이터는 빈번하게 변경될 수 있습니다.

이 경우 키 기반 이벤트 페이로드가 더 적합할 수 있습니다.

소프트웨어 아키텍처의 대부분과 마찬가지로 아키텍트의 선택은 스펙트럼상에 놓여 있습니다.

단순한 이진수가 아닙니다. 그렇기 때문에 어떤 상황이 발생하는지 주의 깊게 살펴야 합니다.

빈혈성 사건으로 알려져 있습니다.

빈혈 사건

빈약한 이벤트는 페이로드에 충분한 정보가 포함되어 있지 않은 파생 이벤트입니다.

이벤트 처리기가 결정을 내리는 데 도움이 되는 정보가 부족하고, 필요한 기능이 부족합니다.

후속 처리를 위한 컨텍스트입니다.

그림 15-13은 빈혈로 인해 발생하는 사건을 보여줍니다. 이 예에서 고객은 다음과 같은 증상을 보입니다.

사용자가 사용자 프로필의 일부 정보를 업데이트했습니다. 해당 정보가 업데이트되면...

데이터베이스에서 고객 프로필 이벤트 프로세서는 `profile_updated` 이벤트를 발생시킵니다.

키 기반 이벤트 페이로드를 사용하여 고객 ID만 전달하는 이벤트를 처리합니다.

키 기반 데이터.

이 이벤트에 대응하는 세 가지 서비스는 고객 ID와 다음 정보만 수신합니다.

고객 프로필이 변경되었다는 맥락입니다. 첫 번째 서비스 (서비스 1)는 다음과 같습니다.

프로필에서 어떤 데이터가 변경되었는지 전혀 모르겠습니다. 이름, 주소, 기타 중요한 정보일 수도 있습니다.

정보를 찾고 계신가요? 안타깝게도 데이터베이스를 조회해도 이 질문에 대한 답을 얻을 수 없습니다.

서비스 1은 어떻게 대응해야 할지, 어떤 조치를 취해야 할지 전혀 모릅니다. 서비스 2가 대응합니다.

`profile_updated` 이벤트에 대한 것이지만, 키만으로는 필요한지 여부를 알 수 없습니다.

추가 처리를 수행하기 위해서입니다. 마지막으로 서비스 3은 동일한 내용에 응답합니다.

이벤트는 발생하지만 이전 값이 무엇이었는지 알 수 없으므로 해당 작업을 수행할 수 없습니다.

처리 과정입니다. 이 세 가지 이벤트 프로세서 모두 어떤 방식으로든 응답해야 합니다.

고객이 프로필을 업데이트하려고 하지만 정보 부족으로 업데이트할 수 없습니다. 이러한 경우는 다음과 같습니다.

빈약한 이벤트: 이벤트를 추가로 분석하는 데 필요한 정보가 포함되지 않은 이벤트.

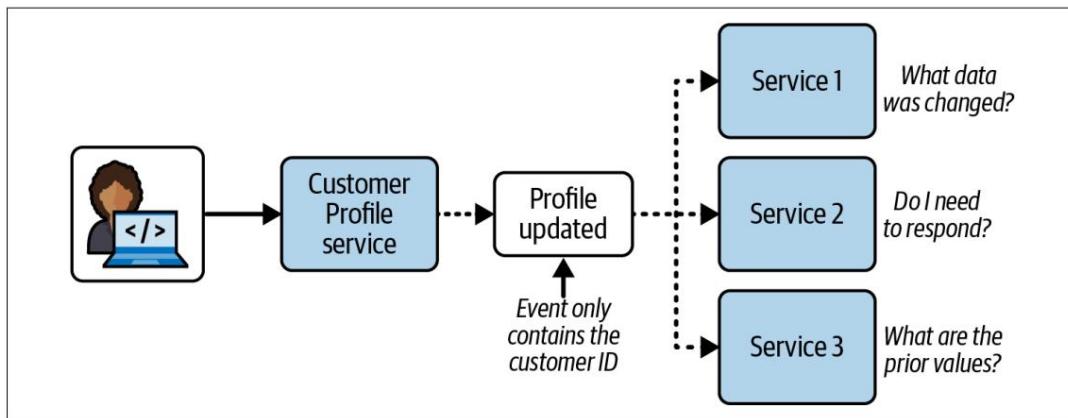


그림 15-13. 빈약한 사건은 사건을 처리하는 데 필요한 맥락이 부족합니다.

이와 같은 부실한 상황을 방지하려면 대부분의 데이터베이스에 이전 값이 반영되지 않는 경우가 많으므로 업데이트된 고객 정보와 이전 값을 모두 포함해야 합니다.

이는 이벤트 페이로드 세분성의 스펙트럼을 보여주는 예시입니다. 스펙트럼의 맨 왼쪽에는 키 기반 이벤트 페이로드가 있는데, 이벤트에는 키만 포함됩니다. 이는 주문 생성이나 삭제 시에는 유용하지만, 고객이 주문을 업데이트할 때는 적합하지 않습니다. 스펙트럼의 맨 오른쪽에는 데이터 기반 이벤트 페이로드가 있는데, 필요한 데이터든 아니든 모든 데이터가 포함됩니다. 바로 이 지점에서 스탬프 결합 문제가 발생합니다. 고객 프로필 업데이트 시나리오는 이러한 양극단 사이 어딘가에 위치하는데, 적절한 수준의 정보를 제공하여 빈약한 파생 이벤트 문제를 피할 수 있기 때문입니다.

모기떼 앤티패턴

빈혈 현상과 관련된 역패턴으로 '모기 떼'라는 것이 있습니다. 모기는 머리 주위를 윙윙거리며 성가시게 날아다니는 아주 작은 곤충으로, 화창한 날에도 실내로 들어가고 싶어 만들 정도입니다.

빈약한 이벤트는 이벤트 페이로드의 세분성에 관한 것이지만, 모기떼 앤티패턴은 트리거된 이벤트 자체의 세분성과 이벤트 프로세서에서 트리거되는 파생 이벤트의 수에 관한 것입니다.

아키텍트가 단일 이벤트 프로세서에서 너무 많은 파생 이벤트를 발생시키면, '파리 떼' 앤티패턴에 빠질 위험이 있습니다.

그림 15-14에 나와 있는 신용카드 결제 예시를 생각해 보겠습니다. 고객이 주문을 하고 신용카드로 결제가 이루어지는 경우입니다. 신용카드 결제가 완료되면 결제 이벤트 처리기가 결제 적용 이벤트를 발생시키고, (다행히) 사기 탐지 이벤트 처리기가 이 이벤트를 수신합니다. 이 이벤트 처리기는

각 청구 건을 분석하여 합법적인지 사기인지 판단합니다. 결과에 관계없이 사기 탐지 이벤트 처리기는 사기 검사 결과를 이벤트 페이로드에 포함하는 `fraud_checked` 파생 이벤트를 발생시킵니다.

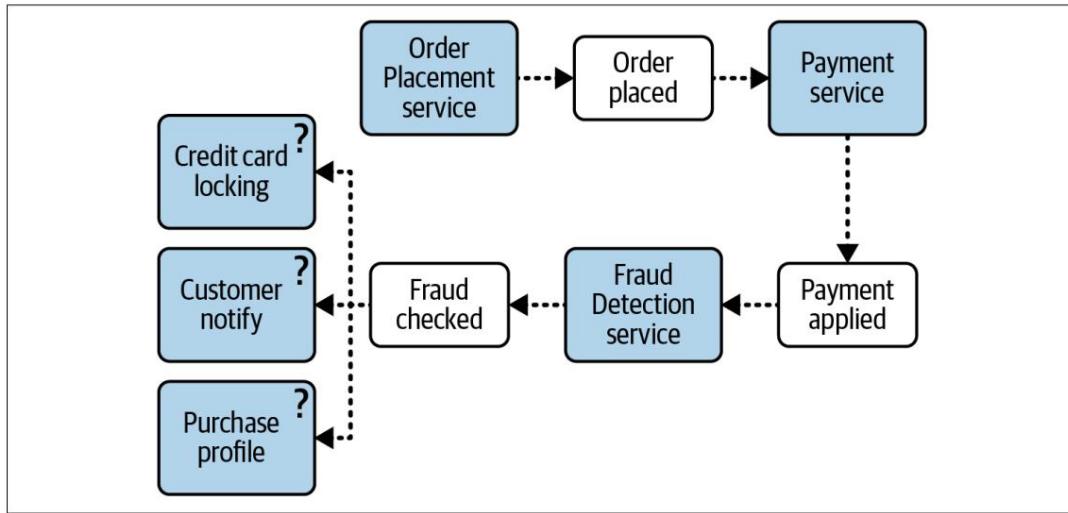


그림 15-14. 지나치게 거친 해상도로 표현된 이벤트의 예

세 명의 이벤트 처리 담당자가 신용카드 사기 검사 결과에 관심을 갖고 있습니다.

- 사기 행위가 감지되면 신용 카드 잠금 이벤트 처리기가 해당 고객을 잠금니다!
추가 요금 청구를 방지하기 위해 고객의 신용카드 정보를 수집합니다.
- 고객 알림 이벤트 처리기는 고객에게 사기 가능성을 알립니다. • 사기가 감지되지 않으면 구매 프로필 이벤트 처리기가 정보를 업데이트합니다.
알고리즘.

안타깝게도, `fraud_checked` 파생 이벤트가 하나만 발생해도 모든 이벤트 처리기가 해당 이벤트에 응답하고, 페이로드를 분석하여 결과를 확인한 후 조치를 취할지 여부를 결정해야 합니다. 이 파생 이벤트는 너무 세분화되어 있어 모든 이벤트 처리기가 추가적인 처리를 수행해야 합니다. 즉, 단일 파생 이벤트의 페이로드를 분석하여 조치를 취할지 여부를 결정해야 하는 것입니다. 만약 사기가 감지되지 않았다면, 구매 프로필 이벤트 처리기만 조치를 취하면 되므로 대역폭과 처리 능력이 낭비됩니다.

그림 15-15에서처럼 두 개의 별도 파생 이벤트 (`fraud_detected` 및 `no_fraud_detected`)를 발생시키는 것이 훨씬 더 효율적인 접근 방식입니다. 여기서 사기 탐지 이벤트 프로세서가 발생시키는 파생 이벤트는 이벤트 페이로드 외부에 컨텍스트를 제공하므로 각 이벤트 프로세서는 이벤트의 내부 페이로드를 분석하지 않고도 응답 여부를 결정할 수 있습니다.

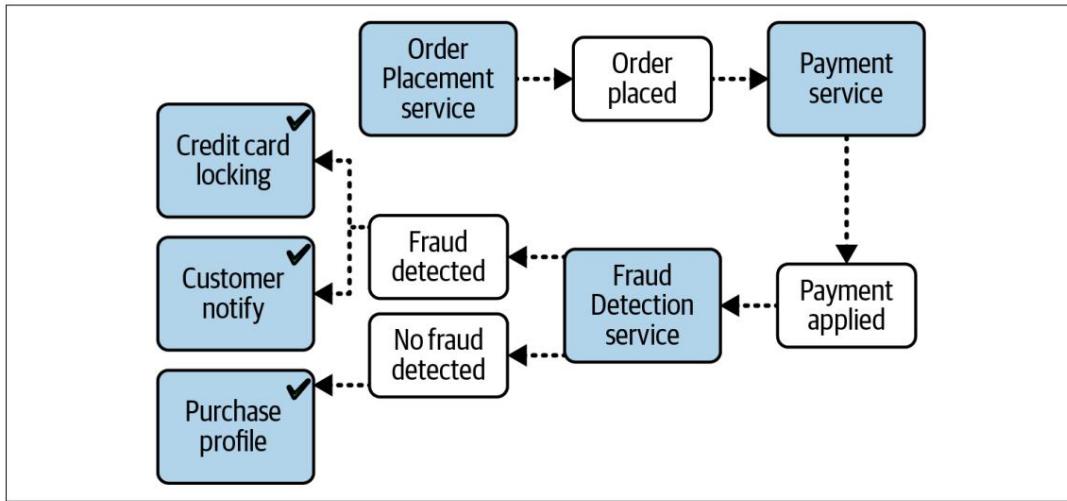


그림 15-15. 여러 이벤트를 동시에 발생시키면 더욱 효율적인 처리와 의사 결정이 가능해진다.

이 예시에서는 각 결과에 대해 여러 개의 파생 이벤트를 발생시킴으로써 이벤트 흐름을 개선하고, 변경 빈도를 줄이며, 처리 효율을 높일 수 있습니다. 그러나 너무 많은 파생 이벤트를 발생시키면 '파리 때' 안티패턴이 발생할 수 있습니다.

그림 15-16에 나타난 시나리오는 이러한 안티패턴이 어떻게 발생할 수 있는지 보여줍니다. 고객이 최근 이사를 하여 웹사이트에서 사용자 프로필을 수정해야 합니다. 신용카드 청구지 주소, 배송지 주소(주문 상품이 배송될 곳), 그리고 전화번호를 기준 유선전화에서 휴대전화로 변경해야 합니다. 고객이 프로필 변경을 위해 제출 버튼을 누르면, 고객 프로필 이벤트 프로세서가 업데이트 요청을 수신하고 데이터베이스를 업데이트한 후, 추가 처리에 필요한 정보를 포함하는 각 업데이트에 대한 별도의 이벤트를 발생시킵니다.

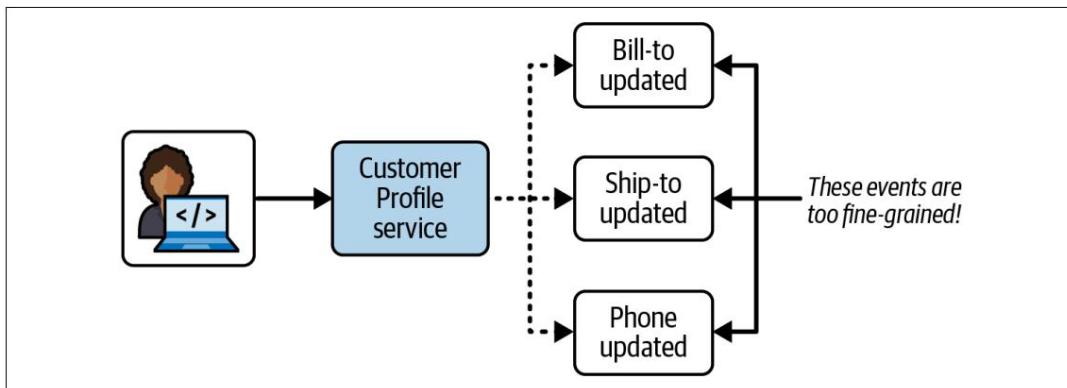


그림 15-16. 너무 많은 ne-grained 파생 이벤트를 발생시키는 것을 모기떼 역패턴(Swarm of Gnats antipattern)이라고 합니다.

너무 세분화된 이벤트를 많이 발생시키면 시스템이 과부하되어 동일한 내용(예: 고객이 사용자 프로필을 업데이트함)과 관련된 파생 이벤트가 과도하게 생성될 수 있습니다. 이러한 안티패턴은 다른 이벤트 처리기에 서도 수많은 작은 파생 이벤트를 발생시켜 결국 시스템의 전체적인 이벤트 흐름을 파악하기 어렵게 만듭니다.

이러한 안티패턴을 피하기 위해 아키텍트는 각 개별 프로필 업데이트를 모든 업데이트된 필드의 업데이트 전후 데이터를 포함하는 단일 profile_updated 파생 이벤트로 묶어 전체 작업을 처리할 수 있습니다. 이보다 효율적인 접근 방식은 [그림 15-17에 나와 있습니다](#).

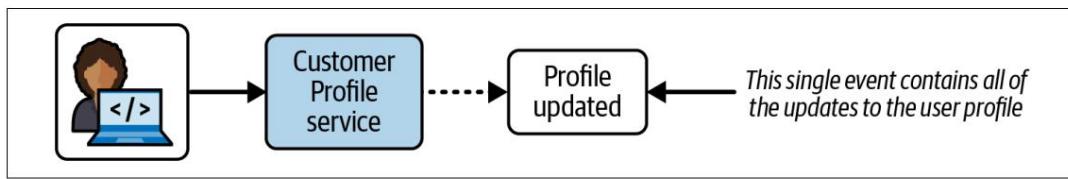


그림 15-17. 개별 상태 변화를 단일 파생 이벤트로 결합하면 모기떼(Swarm of Gnats) 안티패턴을 피할 수 있습니다.

파생 이벤트에 적합한 세분성 수준을 결정하는 것은 상당히 어려울 수 있습니다.

이벤트 흐름을 단순화하고 '파리 떼' 안티패턴을 피하려면 처리 결과 또는 상태 변화에 집중하는 것이 좋습니다.

오류 처리

반응형 아키텍처의 워크플로우 이벤트 패턴은 비동기 워크플로우에서 오류를 처리하는 한 가지 방법입니다. 이 패턴은 시스템이 응답성에 영향을 주지 않고 비동기 오류를 처리할 수 있도록 함으로써 복원력과 응답성 모두를 향상시킵니다.

워크플로우 이벤트 패턴은 [그림 15-18](#)에서 보는 것처럼 워크플로우 딜리게이트를 사용하여 위임, 격리 및 복구 기능을 활용합니다. 이 패턴에서 이벤트 프로세서는 메시지 채널을 통해 이벤트 컨슈머에게 데이터를 비동기적으로 전달합니다. 이벤트 컨슈머가 데이터를 처리하는 동안 오류가 발생하면 즉시 해당 오류를 워크플로우 프로세서 서비스에 위임하고 이벤트 큐의 다음 메시지로 넘어갑니다. 이렇게 하면 다음 메시지가 즉시 처리되므로 전반적인 응답성이 동일하게 유지됩니다. 이벤트 컨슈머가 오류를 파악하는 데 시간을 소비하게 되면 큐의 다음 메시지를 처리하지 못하게 되어 다음 메시지뿐만 아니라 처리 큐에서 대기 중인 다른 모든 메시지의 처리가 지연됩니다.

워크플로 프로세서 서비스가 오류를 수신하면 메시지의 문제점을 파악하려고 시도합니다. 고정적이고 결정론적인 오류일 수도 있고, 머신러닝이나 AI 알고리즘을 사용하여 데이터의 이상 징후를 분석할 수도 있습니다. 어떤 경우든 워크플로 프로세서는 프로그램적으로(즉, 자체적으로) 오류를 처리합니다.

사람의 개입으로 원본 데이터가 손상되어 수정이 시도된 후, 원래 큐로 다시 전송됩니다. 이벤트 컨슈머는 이 업데이트된 메시지를 새로운 메시지로 인식하고 다시 처리하려고 시도하며, 이번에는 성공할 가능성이 더 높습니다.

물론 워크플로 처리기가 메시지의 오류를 항상 파악할 수 있는 것은 아닙니다. 이러한 경우, 워크플로 처리기는 메시지를 다른 큐로 보내고, 해당 큐는 숙련된 담당자의 데스크톱 대시보드로 전송됩니다. 담당자는 메시지를 검토하고 수동으로 수정한 후 (일반적으로 회신 메시지 헤더 변수를 통해) 원래 큐로 다시 전송합니다.

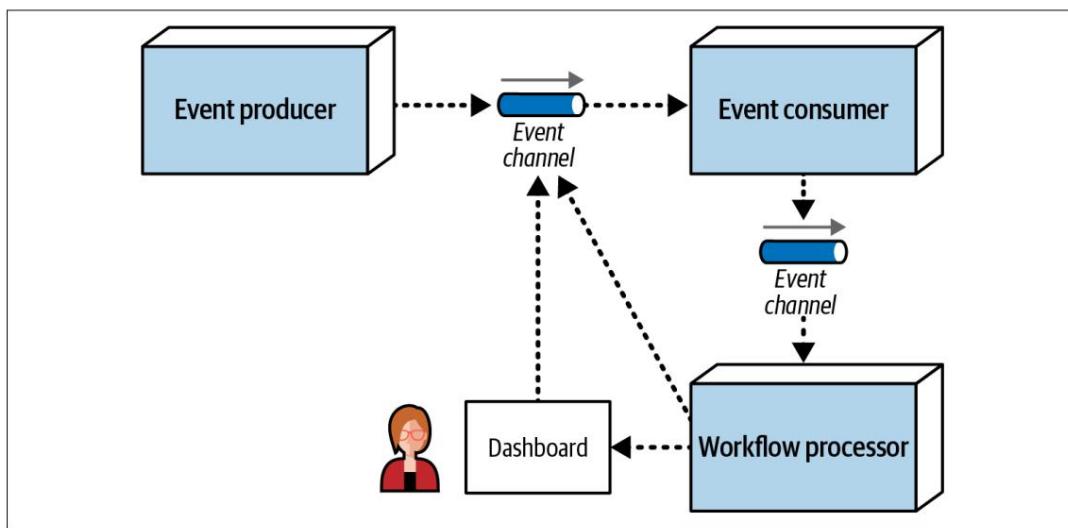


그림 15-18. 반응형 아키텍처의 워크플로우 이벤트 패턴

한 지역의 거래 자문가가 다른 지역의 대형 거래 회사를 대신하여 거래 주문(어떤 주식을 몇 주 매수할지에 대한 지시)을 접수한다고 가정해 보겠습니다. 자문가는 이러한 거래 주문들을 보통バス켓이라고 부르는 형태로 묶어 비동기적으로 전국에 있는 브로커에게 전송하고, 브로커는 해당 주식을 매수합니다. 예시를 단순화하기 위해 거래 지시에 대한 계약이 다음 조건을 준수해야 한다고 가정해 보겠습니다.

ACCOUNT(문자열), SIDE(문자열), SYMBOL(문자열), SHARES(long)

대형 거래 회사가 거래 자문사로부터 다음과 같은 애플(AAPL) 거래 주문을 받았다고 가정해 보겠습니다.

```

12654A87FR4,구매,AAPL,1254
87R54E3068U,구매,AAPL,3122
6R4NB7609JJ,구매,AAPL,5433
2WE35HF6DHF,매수,AAPL,8756주
764980974R2,구매,AAPL,1211
1533G658HD8,구매,AAPL,2654
  
```

네 번째 거래 지시문 (2WE35HF6DHF,BUY,AAPL,8756 SHARES) 에는 거래 주식 수 뒤에 SHARES라는 단어가 있습니다 . 주 거래 회사가 오류 처리 기능 없이 이러한 비동기 거래 주문을 처리할 경우, 거래 배치 서비스에서 다음과 같은 오류가 발생합니다 .

```
"main" 스레드에서 예외 발생 java.lang.NumberFormatException:  
    입력 문자열: "8756 SHARES"에서
```

```
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
호출, java.lang.Long.parseLong(Long.java:589) 호출, java.lang.Long.<init>(Long.java:965) 호출, trading.TradePlacement.execute(Tr
```

이 예외가 발생하는 이유는 비동기 요청이기 때문에 동기적으로 응답하여 오류를 수정할 사용자가 없기 때문입니다. 따라서 거래 배치 서비스는 오류 상황을 기록하는 것 외에는 아무것도 할 수 없습니다.

워크플로우 이벤트 패턴을 적용하면 이 오류를 프로그래밍 방식으로 수정할 수 있습니다. 기본 회사는 거래 자문 서비스나 해당 서비스가 전송하는 거래 주문 데이터에 대한 제어 권한이 있으므로 오류를 직접 수정해야 합니다([그림 15-19 참조](#)). 동일한 오류 (2WE35HF6DHF,BUY,AAPL,8756 SHARES) 가 발생하면 거래 배치 서비스는 예외에 대한 오류 정보를 함께 전달하여 비동기 메시지를 통해 거래 배치 오류 서비스로 오류를 즉시 전달하여 처리합니다.

```
거래 내역: 12654A87FR4, 매수, AAPL, 1254  
거래 내역: 87R54E3068U, 매수, AAPL, 3122  
거래 내역: 6R4NB7609JJ, 매수, AAPL, 5433  
거래 주문 오류: "2WE35HF6DHF,BUY,AAPL,8756 SHARES"  
거래 오류 처리기로 전송 <-- 오류 수정 작업을 위임하고 다음 단계로 넘어갑니다.  
거래 내역: 764980974R2, 매수, AAPL, 1211  
...
```

거래 배치 오류 서비스는 워크플로우 대리자 역할을 하며 오류를 수신하고 예외를 검사합니다. 문제가 '주식 수' 필드의 'SHARES'라는 단어에 있음을 확인한 거래 배치 오류 서비스는 'SHARES'라는 단어를 제거하고 거래를 재처리하기 위해 다시 제출합니다.

```
거래 주문 수신 오류: 2WE35HF6DHF, 매수, AAPL, 8756 주  
거래 확정: 2WE35HF6DHF, 매수, AAPL, 8756  
재처리를 위해 거래 내역을 다시 제출합니다.
```

TradePlacement 서비스에서 이제 고정 금리 거래를 성공적으로 처리할 수 있습니다.

...
체결된 거래: 1533G658HD8,BUY,AAPL,2654 체결된 거래
래: 2WE35HF6DHF,BUY,AAPL,8756 <-- 이것은 원래 오류로 발생한 거래입니다

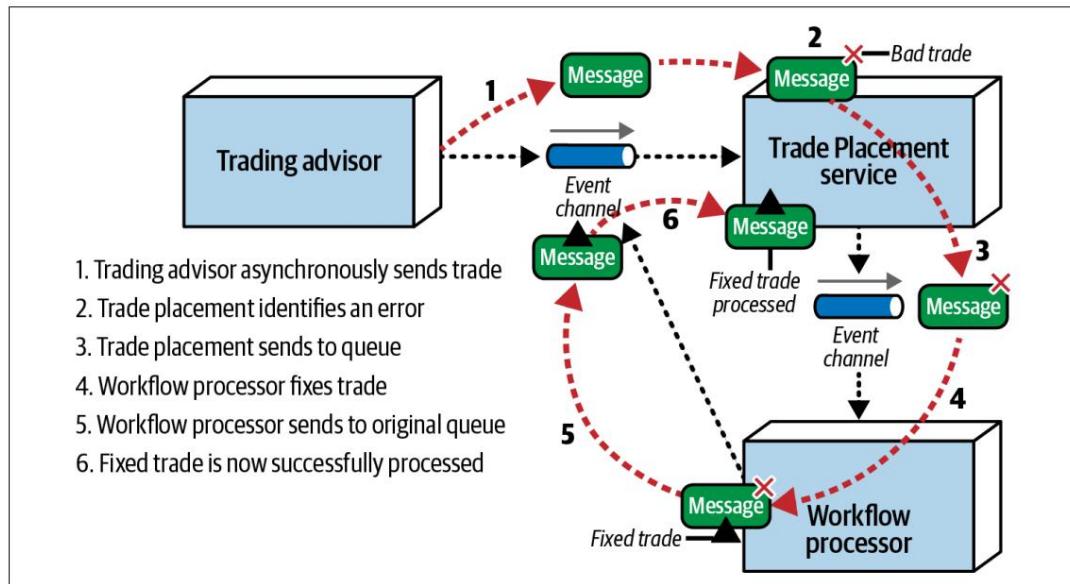


그림 15-19. 워크플로우 이벤트 패턴을 이용한 오류 처리

워크플로우 이벤트 패턴을 사용할 때 발생하는 한 가지 결과는 워크플로우 프로세서로 전송된 후 다시 제출된 메시지가 순서대로 처리되지 않는다는 것입니다. 거래 예시에서 메시지 순서는 매우 중요합니다. 특정 계좌 내의 모든 거래는 순서대로 처리되어야 하기 때문입니다(예: 동일한 증권 계좌 내에서 IBM 매도 거래는 AAPL 매수 거래 보다 먼저 처리되어야 함). 특정 컨텍스트(이 경우 증권 계좌 번호) 내에서 메시지 순서를 유지하는 것은 복잡하지만 불가능한 것은 아닙니다. 이 문제를 해결하는 한 가지 방법은 거래 배치 서비스에서 오류가 발생한 거래의 증권 계좌 번호를 큐에 저장하는 것입니다.

동일한 증권 계좌 번호로 이루어진 모든 거래는 나중에 처리될 때까지 임시 대기열에 저장됩니다(선입선출, 즉 FIFO 순서). 오류가 있는 거래가 수정되고 처리되면, 거래 배치 서비스는 해당 계좌의 나머지 거래를 대기열에서 꺼내 순서대로 처리합니다.

데이터 손실 방지 비동기 통신을 다루

는 설계자는 항상 데이터 손실, 즉 이벤트나 메시지가 누락되거나 최종 목적지에 도달하지 못하는 상황을 우려합니다. 다행히 데이터 손실을 방지할 수 있는 기본적인 기술들이 이미 마련되어 있습니다.

아키텍트는 다양한 방식으로 이벤트 채널을 구현할 수 있습니다. 대부분의 이벤트 기반 아키텍처는 이벤트 발생 및 응답을 위해 AMQP(Advanced Message Queuing Protocol)를 사용합니다. AMQP 브로커의 예로는 Amazon SNS(Simple Notification Service), RabbitMQ, Solace, Azure Event Hubs 등이 있습니다. AMQP를 사용하면 이벤트가 익스체인지에 게시됩니다. 익스체인저는 이벤트를 소비하는 이벤트 프로세서가 설정한 바인딩 규칙을 사용하여 해당 이벤트를 구독하는 각 이벤트 프로세서의 큐로 이벤트를 전달합니다. AMQP 브로커는 데이터 손실을 방지하기 위해 이벤트 포워딩 패턴이라는 기술을 활용할 수도 있으며, 이 기술은 이 섹션에서 설명합니다.

또 다른 이벤트 채널 구현 방식으로는 Jakarta Messaging API(이전에는 Java Message Service, JMS)가 있습니다. 이는 큐에서 사용하는 2단계 전달 프로세스 대신 토픽을 사용합니다. 그럼에도 불구하고 Jakarta Messaging은 이벤트에 응답하는 이벤트 프로세서가 영구 구독자로 구성된 경우 데이터 손실을 방지하기 위해 이벤트 전달 패턴을 활용할 수 있습니다. 영구 구독자란 이벤트를 수신하는 것이 보장되는 구독자를 말합니다. 이벤트 프로세서가 다운되거나 다른 이유로 사용할 수 없는 경우, JMS 토픽은 구독한 이벤트 프로세서가 다시 사용 가능해질 때까지 이벤트를 저장합니다.

또 다른 이벤트 채널 구현 방법으로는 Kafka를 이벤트 브로커(큐와 토픽을 포함하는 소프트웨어 제품)로 사용하는 이벤트 스트리밍이 있습니다. 이벤트 스트리밍에서 데이터 손실을 방지하는 기술은 이 섹션에서 설명하는 이벤트 포워딩 패턴에 사용되는 기술과는 매우 다릅니다. 이러한 스트리밍 이벤트 브로커를 사용할 때 데이터 손실을 방지하는 방법에 대한 자세한 내용은 [Kafka 웹사이트](#)를 참조하십시오.

이벤트 프로세서 A가 메시지 브로커에 비동기적으로 이벤트를 게시하고, 이 이벤트가 최종적으로 AMQP 큐 또는 JMS 토픽으로 전송되는 일반적인 시나리오를 생각해 보겠습니다.

이벤트 프로세서 B는 이벤트에 응답하여 페이로드의 데이터를 데이터베이스에 삽입합니다. 그림 15-20에서 보는 바와 같이, 이 시나리오에서 데이터 손실이 발생할 수 있는 세 가지 방법이 있습니다.

1. 이벤트 프로세서 A가 이벤트를 발행하는 동안 이벤트 브로커로부터 확인 응답을 받기 전에 충돌이 발생합니다. 또는 이벤트 브로커가 이벤트 프로세서 A에 확인 응답을 보냈지만, 다른 이벤트 프로세서가 이벤트를 수락하기 전에 충돌이 발생합니다.
2. 이벤트 프로세서 B는 큐에서 이벤트를 수락하지만 이벤트를 처리하기 전에 충돌합니다.
3. 이벤트 프로세서 B는 데이터 오류로 인해 메시지를 데이터베이스에 저장할 수 없습니다.

이러한 데이터 손실 영역 각각은 이벤트 전달 패턴을 통해 완화할 수 있습니다.

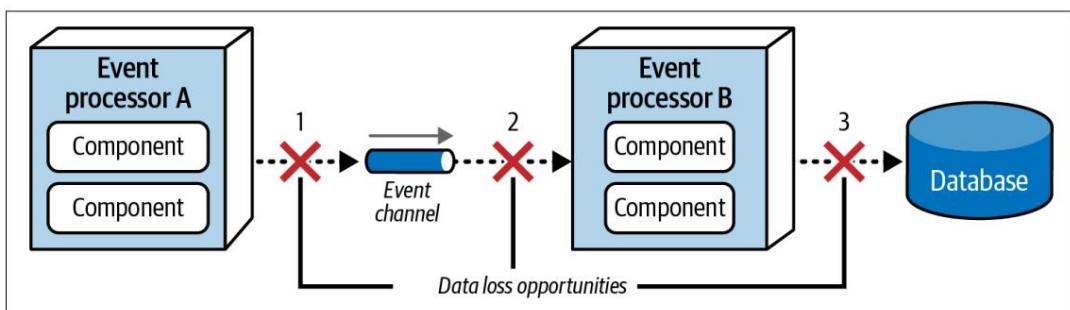


그림 15-20. 이벤트 기반 아키텍처에서 데이터 손실이 발생할 수 있는 위치

첫 번째 문제는 이벤트가 큐에 도달하지 못하거나 이벤트가 읽히기 전에 브로커에 오류가 발생하는 경우입니다. 이를 해결하려면 동기 전송(synchro! nous send)과 함께 영구 메시지 큐를 사용해야 합니다. 영구 메시지 큐는 전달 보장을 지원합니다. 이벤트 브로커가 이벤트를 수신하면 빠른 검색을 위해 메모리에 이벤트를 저장할 뿐만 아니라 파일 시스템이나 데이터베이스와 같은 물리적 데이터 저장소에도 이벤트를 저장합니다. 이벤트 브로커가 다운되더라도 이벤트는 디스크에 물리적으로 저장되어 있으므로 브로커가 다시 작동할 때에도 처리할 수 있습니다. 동기 전송은 이벤트 프로세서에서 블로킹 대기를 수행하여 브로커가 이벤트를 데이터베이스에 저장했음을 확인할 때까지 이벤트 트리거를 중지합니다.

이 두 가지 기본 기술은 이벤트 생성자와 큐 사이의 데이터 손실을 방지합니다. 이벤트가 이벤트 생성자에 남아 있거나 큐 내에 영구적으로 저장되기 때문입니다.

클라이언트 승인 모드라는 기본적인 메시징 기법은 두 번째 문제, 즉 이벤트 처리기 B 가 큐에서 사용 가능한 다음 이벤트를 꺼내지만 처리하기 전에 충돌하는 문제를 해결할 수 있습니다. 기본적으로 이벤트가 큐에서 읽히면 해당 큐에서 즉시 제거됩니다(이를 자동 승인 모드라고 합니다). 클라이언트 승인 모드는 이벤트를 큐에 유지하고 클라이언트 ID를 첨부하여 다른 소비자가 해당 이벤트를 읽거나 처리할 수 없도록 합니다. 이 모드를 사용하면 이벤트 처리기 B가 충돌하더라도 이벤트가 큐에 보존되어 메시지 손실을 방지할 수 있습니다.

세 번째 문제는 이벤트 처리기 B가 데이터 오류로 인해 이벤트를 데이터베이스에 저장할 수 없는 경우인데, 이는 ACID 트랜잭션을 통한 데이터베이스 커밋으로 해결할 수 있습니다. 이벤트 처리기가 데이터베이스 커밋을 실행하면 데이터가 데이터베이스에 저장되는 것이 보장됩니다. 마지막 참여자 지원(LPS)은 모든 처리가 완료되었고 이벤트가 저장되었음을 확인하는 응답을 통해 저장된 큐에서 이벤트를 제거합니다. 이는 이벤트가 이벤트 처리기 A에서 데이터베이스로 전송되는 동안 손실되지 않았음을 보장합니다. 이러한 기법들은 [그림 15-21](#)에 나타나 있습니다.

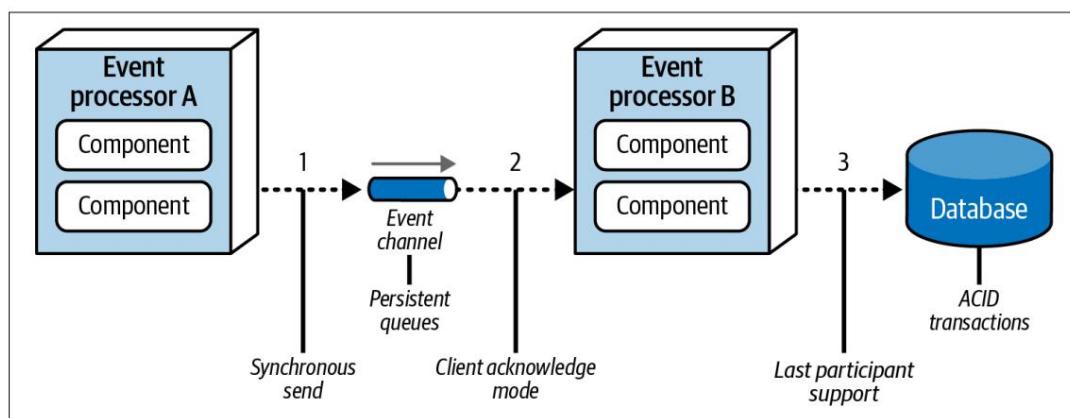


그림 15-21. 이벤트 기반 아키텍처에서 데이터 손실 방지

요청-응답 처리 지금까지 이 장에서

는 이벤트 소비자로부터 즉각적인 응답이 필요하지 않은 비동기 요청을 다루었습니다. 하지만 다른 이벤트 처리기로부터 정보를 즉시 받아야 하는 경우, 예를 들어 이벤트를 발생시키기 전에 확인 ID나 승인 응답을 기다려야 하는 경우는 어떻게 처리해야 할까요? 이러한 시나리오에서는 요청을 완료하기 위해 동기 통신이 필요합니다.

EDA에서 동기 통신은 일반적으로 요청-응답 메시징(때때로 의사 동기 통신이라고도 함)을 통해 이루어집니다. 요청-응답 메시징 내의 각 이벤트 채널은 요청 큐와 응답 큐라는 두 개의 큐로 구성됩니다. 정보를 요청하는 메시지 생산자는 비동기적으로 데이터를 요청 큐로 전송한 후 제어권을 메시지 생산자에게 반환합니다. 메시지 생산자는 추가 처리를 수행하고 응답 큐에서 응답을 기다립니다. 메시지 소비자는 메시지를 수신하고 처리한 후 응답을 응답 큐로 전송합니다. 이벤트 생산자는 응답 데이터가 포함된 메시지를 수신합니다. 이러한 기본 흐름은 [그림 15-22에 나와 있습니다](#).

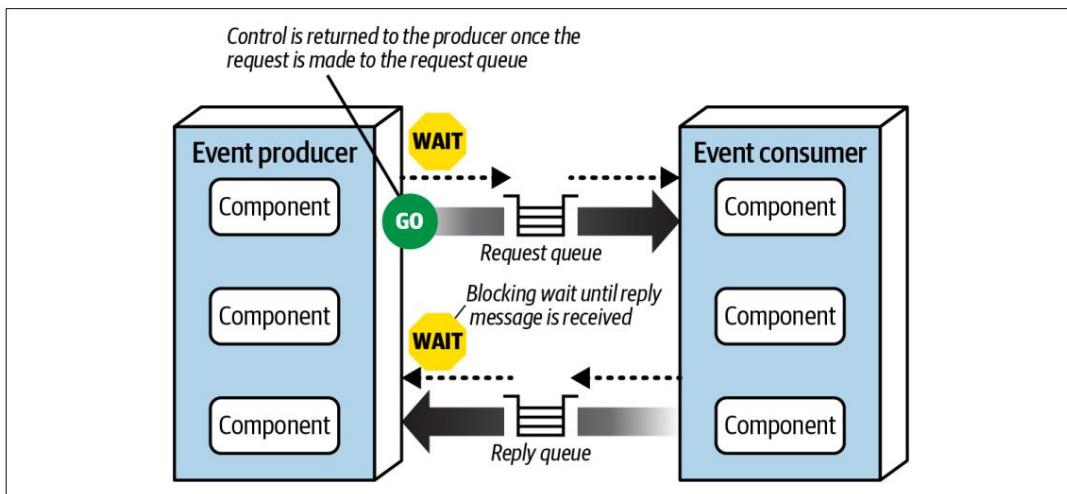


그림 15-22. 요청-응답 메시지 처리

요청-응답 메시징을 구현하는 주요 방법은 두 가지입니다. 첫 번째(그리고 가장 일반적인) 방법은 응답 메시지의 메시지 헤더에 상관 ID(CID) 필드를 추가하는 것입니다. 이 필드는 일반적으로 원래 요청 메시지의 메시지 ID(그림 15-23에서는 단순히 ID라고 함)로 설정됩니다. 작동 방식은 다음과 같습니다.

1. 이벤트 생성자는 요청 큐에 메시지를 보내고 고유 메시지 ID(ID 124)를 기록합니다. 이 경우 CID는 null입니다.
2. 이벤트 생성자는 메시지 헤더의 CID가 원래 메시지 ID(124)와 같은 메시지 필터(메시지 선택기라고도 함)를 사용하여 응답 큐에서 차단 대기를 수행합니다. 응답 큐에는 ID 855(CID 120)와 ID 856(CID 122)의 두 메시지가 있습니다. 이 두 메시지는 모두 이벤트 소비자가 찾고 있는 CID(124)와 일치하지 않으므로 선택되지 않습니다.
3. 이벤트 소비자는 메시지(ID 124)를 수신하고 요청을 처리합니다.
4. 이벤트 소비자는 응답을 포함하는 회신 메시지를 생성하고 설정합니다. 메시지 헤더의 CID를 원래 메시지 ID(124)로 변환합니다.
5. 이벤트 소비자는 새 메시지 ID(857)를 응답 큐로 보냅니다.
6. 이벤트 생성자는 CID(124)가 일치하므로 메시지를 수신합니다. 2단계에서 선택한 메시지입니다.

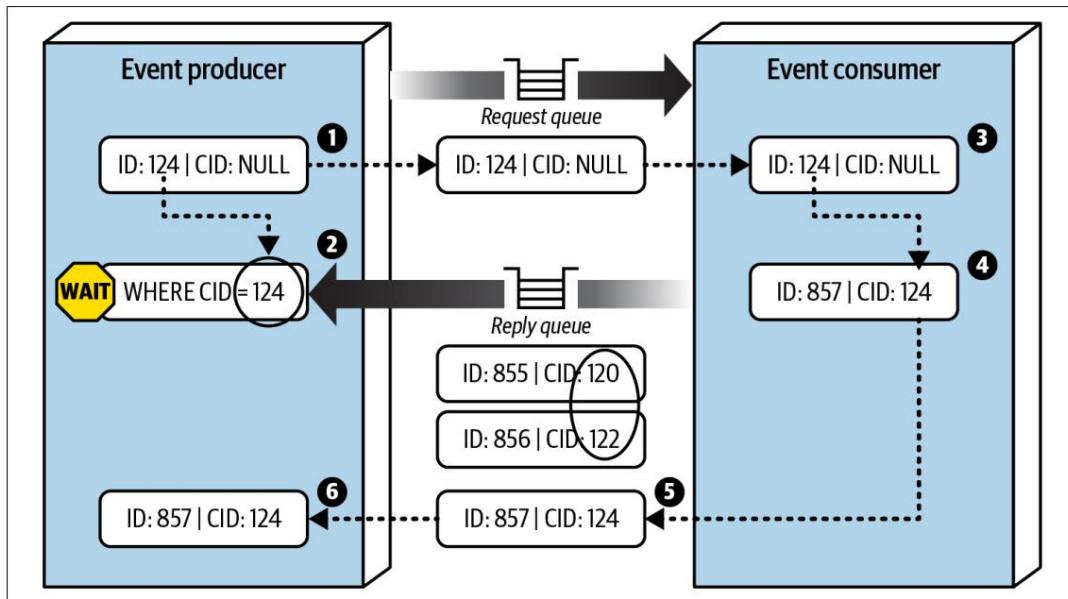


그림 15-23. 상관 ID를 사용한 요청-응답 메시지 처리

요청-응답 메시징을 구현하는 또 다른 방법은 응답 큐로 임시 큐를 사용하는 것입니다. 임시 큐는 특정 요청 전용으로, 요청이 발생할 때 생성되고 요청이 종료될 때 삭제됩니다. 이 기법은 그림과 같이 구현됩니다.

그림 15-24에서처럼 임시 큐는 특정 요청에 대해서만 이벤트 생성자가 알고 있는 전용 큐이기 때문에 상관 관계 ID가 필요하지 않습니다. 임시 큐 기법은 다음과 같이 작동합니다.

1. 이벤트 프로듀서는 임시 큐를 생성하거나(메시지 브로커에 따라 자동으로 생성됨) 응답 헤더(reply-to 헤더)에 임시 큐의 이름(또는 메시지 헤더에 합의된 다른 사용자 지정 속성)을 전달하여 요청 큐로 메시지를 보냅니다.
2. 이벤트 생성자는 임시 응답 큐에서 블로킹 대기를 수행합니다. 이 큐로 전송되는 모든 메시지는 원래 메시지를 보낸 이벤트 생성자에게만 속하므로 메시지 선택기가 필요하지 않습니다.
3. 이벤트 소비자는 메시지를 수신하고 요청을 처리한 후, reply-to 헤더에 지정된 응답 큐로 응답 메시지를 보냅니다.
4. 이벤트 처리기는 메시지를 수신하고 임시 큐를 삭제합니다.

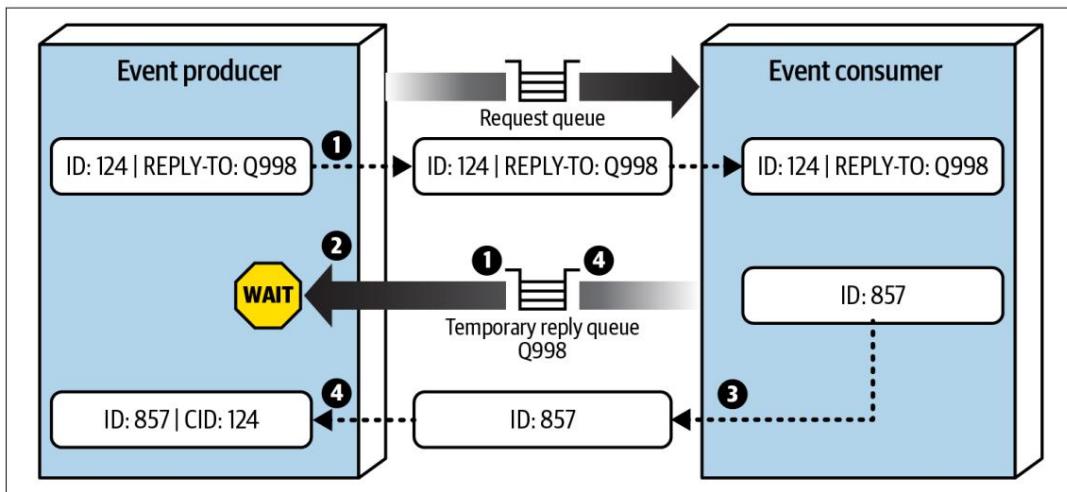


그림 15-24. 임시 큐를 사용한 요청-응답 메시지 처리

임시 큐 방식은 훨씬 간단하지만, 메시지 브로커는 각 요청마다 임시 큐를 생성한 후 즉시 삭제해야 합니다. 이로 인해 브로커 속도가 상당히 저하될 수 있으며, 특히 메시지 양이 많거나 동시 접속자가 많은 경우 전반적인 성능과 응답성에 영향을 미칠 수 있습니다. 따라서 일반적으로 상관 ID 방식을 사용하는 것을 권장합니다.

매개된 이벤트 기반 아키텍처

이 장에서는 지금까지 이벤트 프로세서가 브로드캐스트 기능을 통해 이벤트를 트리거하고 여러 이벤트 프로세서가 해당 이벤트에 응답하는, 안무형 EDA에 초점을 맞추었습니다. 하지만 아키텍트가 더 많은 제어 권한을 원하는 경우도 있을 수 있습니다.

이벤트 처리 과정에서 아키텍트는 미들웨어 토플로지라고 알려진 오케스트레이션된 형태의 EDA를 사용할 수 있습니다.

미들웨어 토플로지는 이 장에서 지금까지 설명한 표준 choreo! 그래프 기반 EDA 토플로지의 몇 가지 단점을 해결합니다. 이 토플로지는 여러 이벤트 프로세서 간의 조정이 필요한 시작 이벤트의 워크플로를 관리하고 제어하는 이벤트 미들웨어를 중심으로 구성됩니다. 미들웨어 토플로지를 구성하는 아키텍처 요소는 시작 이벤트, 이벤트 큐, 이벤트 미들웨어, 이벤트 채널 및 이벤트 프로세서입니다.

중요한 점은 매개된 토플로지는 일반적으로 이벤트보다는 메시지를 사용한다는 것입니다([232페이지](#)의 "이벤트 대 메시지" 참조). 이러한 메시지는 일반적으로 발생한 이벤트(예: order_shipped)가 아니라 명령(예: ship_order)입니다.

안무형 토플로지와 마찬가지로, 시작 이벤트가 전체 프로세스를 시작합니다. 그러나 중재자형 토플로지 ([그림 15-25](#))에서는 이벤트 중재자가 시작 이벤트를 수신합니다. 중재자는 이벤트 처리와 관련된 단계만 알고 있으므로, 해당 파생 메시지를 생성하여 전용 메시지 채널(일반적으로 큐)로 지점 간 방식으로 전송합니다. 그러면 이벤트 처리기가 전용 이벤트 채널을 수신하고 메시지를 처리한 후 (일반적으로) 작업이 완료되면 중재자에게 응답합니다. 중재자형 토플로지 내의 이벤트 처리기는 추가적인 파생 메시지를 통해 시스템의 다른 부분에 자신이 수행한 작업을 알리지 않습니다.

대부분의 미들웨어 토플로지 구현에서는 여러 개의 미들웨어가 사용되며, 각 미들웨어는 특정 도메인 또는 이벤트 그룹과 연결됩니다. 이는 미들웨어 토플로지에서 문제가 될 수 있는 단일 장애 지점을 방지하고 전반적인 처리량과 성능을 향상시킵니다. 예를 들어, 고객 미들웨어는 신규 고객 등록 및 프로필 업데이트와 같은 모든 고객 관련 이벤트를 처리하고, 주문 미들웨어는 장바구니에 상품 추가 및 결제와 같은 주문 관련 활동을 처리할 수 있습니다.

아키텍트가 이벤트 미디에이터를 구현하는 방식은 일반적으로 미디에이터가 처리하는 메시지의 특성과 복잡성에 따라 달라집니다. 예를 들어, 간단한 오류 처리 및 오케스트레이션이 필요한 이벤트의 경우 [Apache Camel](#), [Mule ESB](#) 또는 [Spring Integration](#)과 같은 미디에이터로 충분합니다. 이러한 유형의 미디에이터 내에서 메시지 흐름과 메시지 경로는 일반적으로 이벤트 처리 워크플로를 제어하기 위해 Java 또는 C#과 같은 프로그래밍 코드로 사용자 정의됩니다.

하지만 이벤트 워크플로에 조건부 처리가 많고 복잡한 오류 처리 지시문을 포함하는 여러 동적 경로가 필요한 경우 [Apache ODE](#) 또는 [Oracle BPEL Process Manager](#)와 같은 미들웨어가 더 적합한 선택일 수 있습니다. 이러한 미들웨어는 이벤트 처리 단계를 설명하는 XML과 유사한 구조 [인 BPEL](#)(Business Process Execution Language)을 기반으로 합니다. BPEL 아티팩트에는 오류 처리에 사용되는 구조화된 요소도 포함되어 있습니다.

리디렉션, 멀티캐스팅 등. BPEL은 강력하지만 배우기에는 상대적으로 복잡한 언어이므로 아키텍트는 일반적으로 BPEL 엔진 제품군의 GUI 도구를 사용하여 미들웨어를 생성합니다.

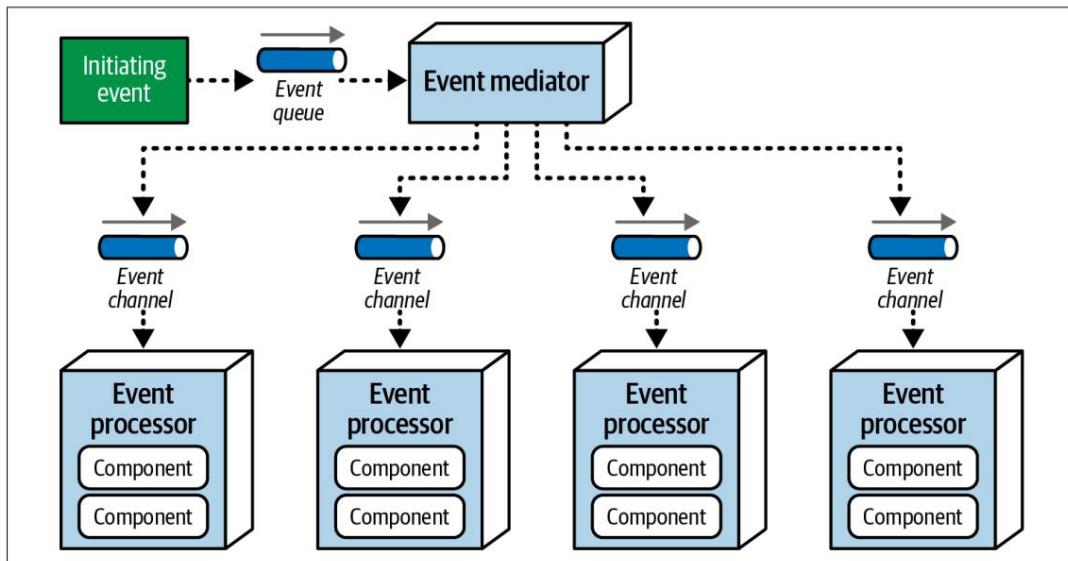


그림 15-25. 매개체 토플로지

BPEL은 복잡하고 동적인 워크플로우에 적합하지만, 이벤트 처리 과정 전반에 걸쳐 사람의 개입이 필요 한 장기 실행 트랜잭션이 포함된 이벤트 워크플로우에는 적합하지 않습니다. 예를 들어, `place_trade` 개시 이벤트를 통해 거래가 체결된다고 가정해 보겠습니다. 이벤트 미들웨어는 이 이벤트를 수락하지만, 처리 과정에서 거래량이 특정 주식 수를 초과하여 수동 승인이 필요하다는 것을 알게 됩니다. 이 경우 이벤트 미들웨어는 이벤트 처리를 중단하고, 선임 트레이더에게 알림을 보내 수동 승인을 요청한 후 승인을 기다려야 합니다. 이러한 경우에는 이벤트 미들웨어를 사용하는 것보다 **jBPM** 과 같은 비즈니스 프로세스 관리(BPM) 엔진을 사용하는 것이 더 적합합니다.

어떤 이벤트 미디에이터를 구현할지 선택하기 전에, 해당 미디에이터가 처리할 이벤트 유형을 파악하는 것 이 중요합니다. 복잡하고 장시간 소요되는, 사람의 상호작용이 포함된 이벤트의 경우 Apache Camel 은 사용 및 유지 관리가 매우 어려울 수 있습니다. 마찬가지로, 간단한 이벤트 흐름에 BPM 엔진을 사용 하는 것은 Apache Camel이 며칠 만에 처리할 수 있는 작업을 몇 달씩 허비하는 결과를 초래할 수 있습니다.

물론 모든 이벤트가 하나의 복잡성 범주에 깔끔하게 들어맞는 경우는 드뭅니다. 따라서 이벤트를 단순, 어려움, 복잡으로 분류하고, 모든 이벤트를 Apache Camel이나 Mule과 같은 단순 미들웨어를 통해 처리하는 것을 권장합니다. 단순 미들웨어는 이벤트 분류에 따라 이벤트를 직접 처리하거나 더 복잡한 이벤트 미들웨어로 전달할 수 있습니다. [그림 15-26](#) 에 나타낸 이 미들웨어 위임 모델은 모든 유형의 이벤트 가 가장 효율적으로 처리할 수 있는 유형의 미들웨어에 의해 처리되도록 보장합니다.

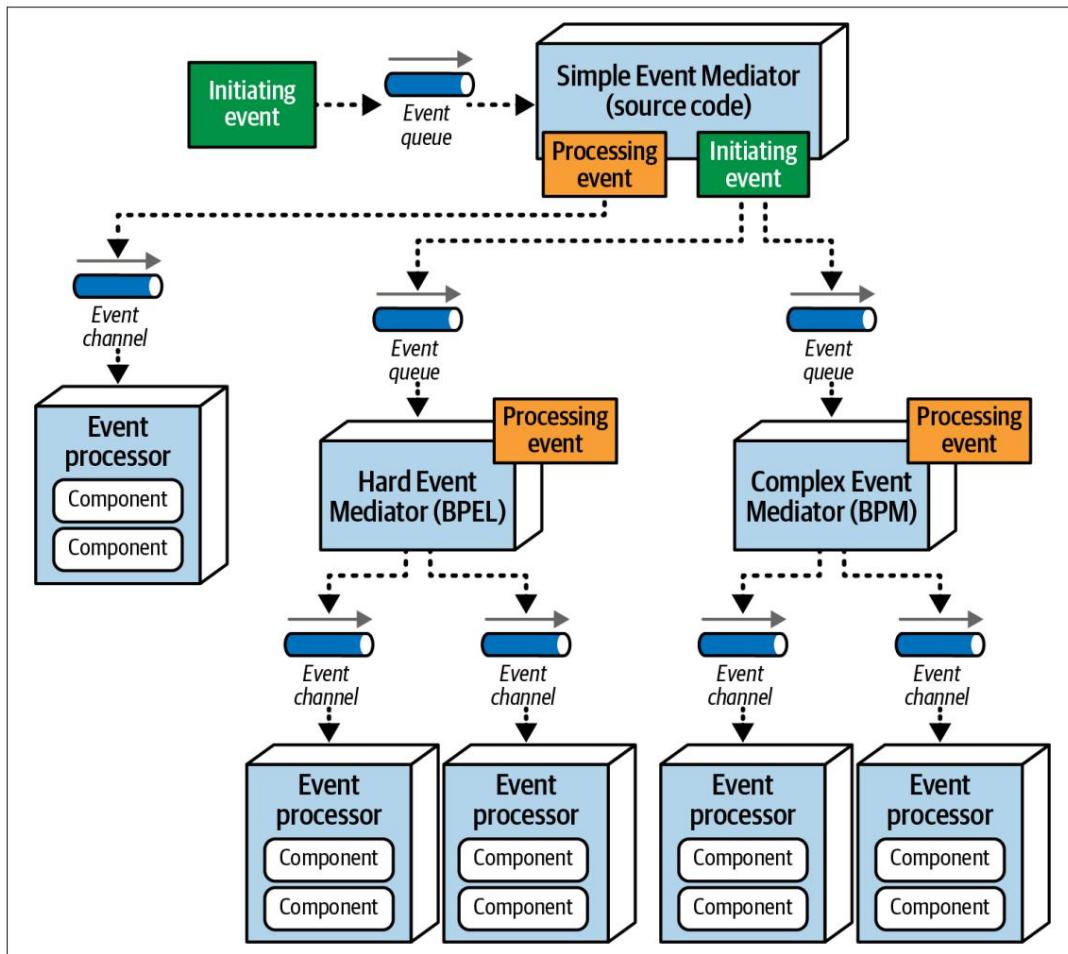


그림 15-26. 적절한 유형의 이벤트 중재자에게 이벤트 위임

그림 15-26에서 볼 수 있듯이, 단순 이벤트 미디에이터는 이벤트 워크플로가 단순 미디에이터가 전적으로 처리할 수 있을 만큼 간단한 경우 파생 메시지를 생성하여 전송합니다. 그러나 시작 이벤트가 복잡하거나 어려운 것으로 분류되면 단순 이벤트 미디에이터는 원래 시작 이벤트를 해당 미디에이터(BPEL 또는 BPM)로 전달합니다. 단순 이벤트 미디에이터는 원래 이벤트를 가로챈 후에도 해당 이벤트가 완료되는 시점을 파악해야 할 수 있으며, 전체 워크플로(클라이언트 알림 포함)를 다른 미디에이터에 위임할 수도 있습니다.

미들웨어 토플로지가 어떻게 작동하는지 살펴보기 위해, 앞서 오케스트레이션 토플로지 섹션에서 설명했던 소매 주문 입력 시스템을 미들웨어 토플로지를 사용하는 경우로 다시 살펴보겠습니다. 미들웨어는 이 특정 이벤트를 처리하는 데 필요한 단계를 알고 있습니다. 미들웨어 구성 요소의 내부 이벤트 흐름은 [그림 15-27](#)에 나와 있습니다.

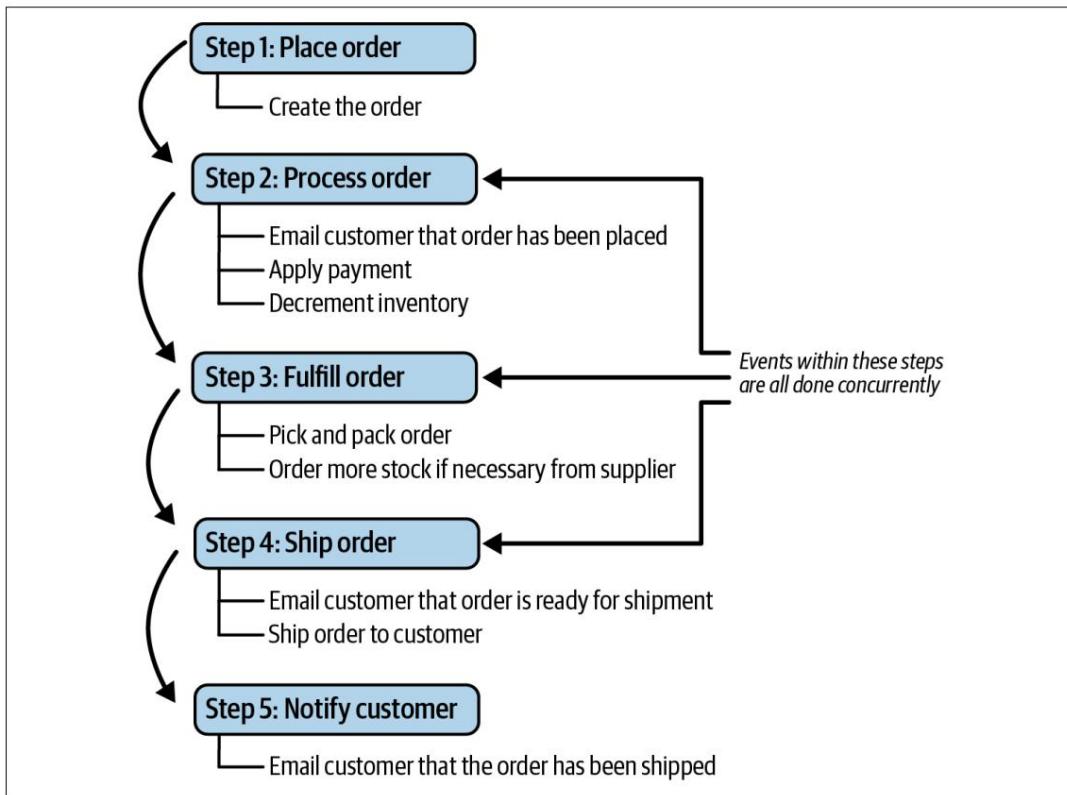


그림 15-27. 중재자가 주문을 넣는 절차

이전 예시와 마찬가지로, 동일한 시작 이벤트 (주문 접수) 가 전용 큐를 통해 이벤트 중개자에게 전송되어 처리됩니다. 고객 중개자는 이 시작 이벤트를 수신하고 그림 15-27 의 흐름에 따라 파생 메시지를 생성하기 시작합니다 . 2단계, 3단계, 4단계에 나타난 이벤트는 모두 단계 간에 동시에 순차적으로 처리됩니다. 즉, 3단계(주문 처리)가 완료되고 확인되어야만 4단계(주문 배송)에서 고객에게 주문 배송 준비 완료 알림이 전송될 수 있습니다.

고객 중개자는 시작 이벤트를 수신하면 주문 생성 파생 메시지를 생성하여 주문 배치 대기열로 보냅니다 (그림 15-28 참조).

주문 접수 이벤트 처리기는 메시지를 수락하고 주문을 검증 및 생성한 후, 미들웨어에 승인 메시지와 주문 ID를 반환합니다. 이 시점에서 미들웨어는 주문이 접수되었음을 고객에게 알리기 위해 주문 ID를 보낼 수도 있고, 주문 접수와 관련된 특정 비즈니스 규칙에 따라 모든 단계가 완료될 때까지 기다릴 수도 있습니다.

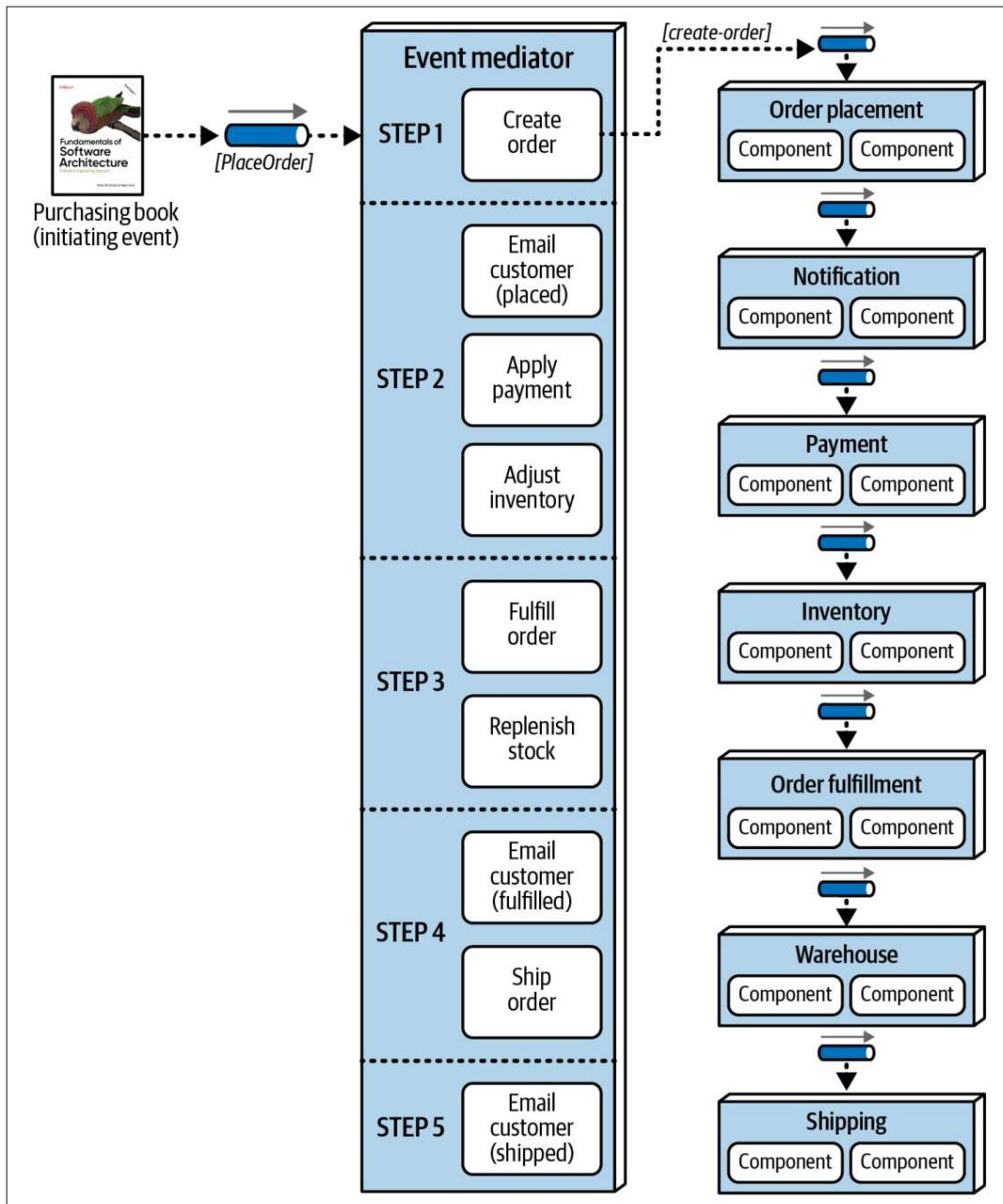


그림 15-28. 종재자 예시의 1단계

1단계가 완료되었으므로, 미들웨어는 2단계로 이동하여([그림 15-29 참조](#)) 고객에게 이메일 전송, 결제 적용, 재고 조정이라는 세 가지 파생 메시지를 동시에 생성합니다 . 이 세 메시지를 각각의 큐에 전송합니다. 세 개의 이벤트 프로세서는 모두 이 메시지를 수신하고 각자의 작업을 수행한 후, 처리가 완료되었음을 미들웨어에 알립니다. 미들웨어는 3단계로 넘어가기 전에 세 개의 병렬 프로세스 모두로부터 확인 응답을 받을 때까지 기다려야 합니다.

병렬 이벤트 처리기 중 하나에서 오류가 발생하면 미들웨어가 수정 조치를 취할 수 있습니다(이 섹션 뒷부분에서 자세히 설명합니다).

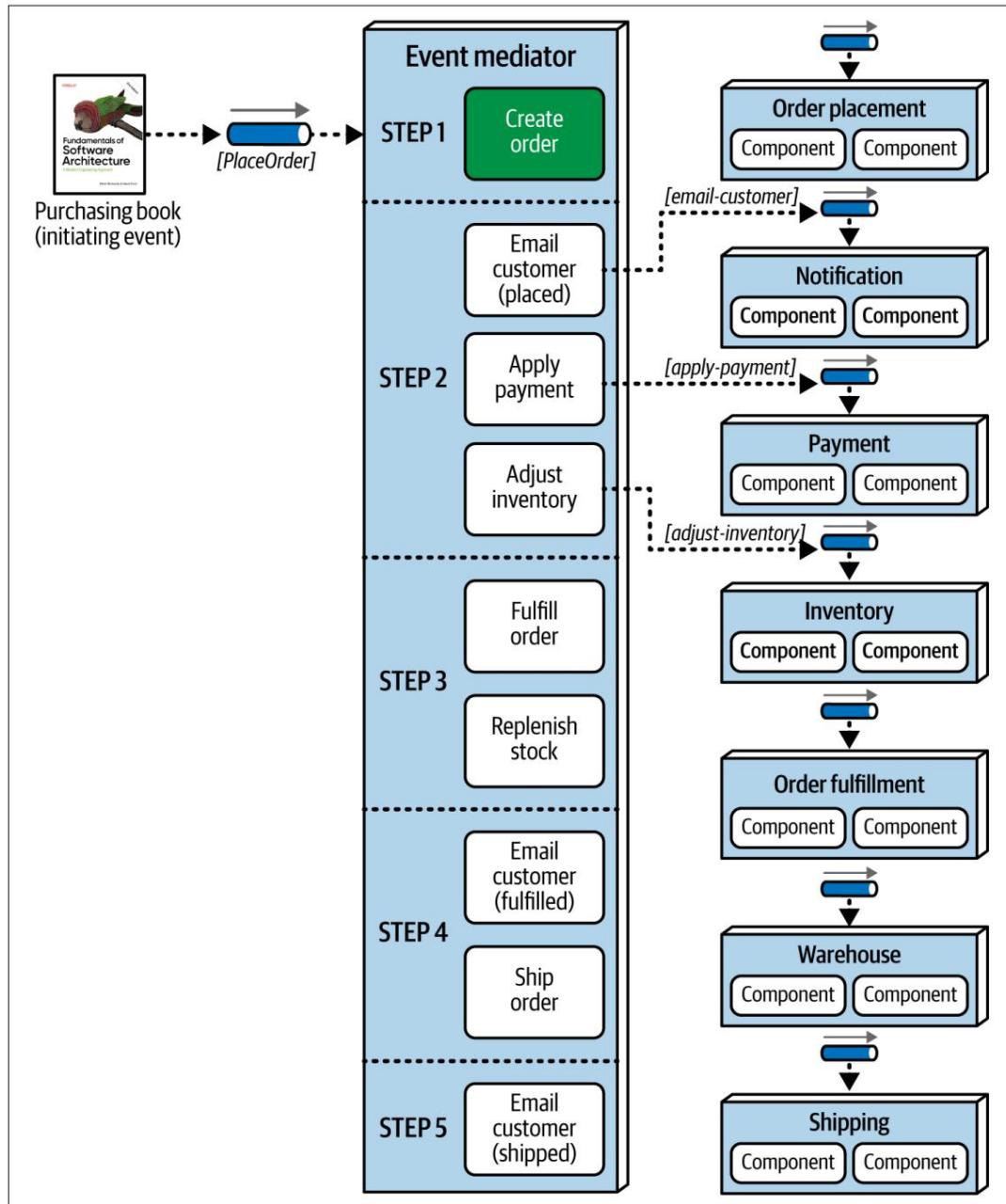


그림 15-29. 중재자 예시의 2단계

중재자가 2단계의 모든 이벤트 처리기로부터 성공적인 확인을 받으면 3단계로 이동하여 주문을 처리할 수 있습니다([그림 15-30 참조](#)). 이 두 메시지 (주문 처리 및 재고 주문)는 동시에 발생할 수 있습니다. 주문 처리 및 창고 이벤트 처리기는 메시지를 수락하고 작업을 수행한 후 중재자에게 확인을 반환합니다.

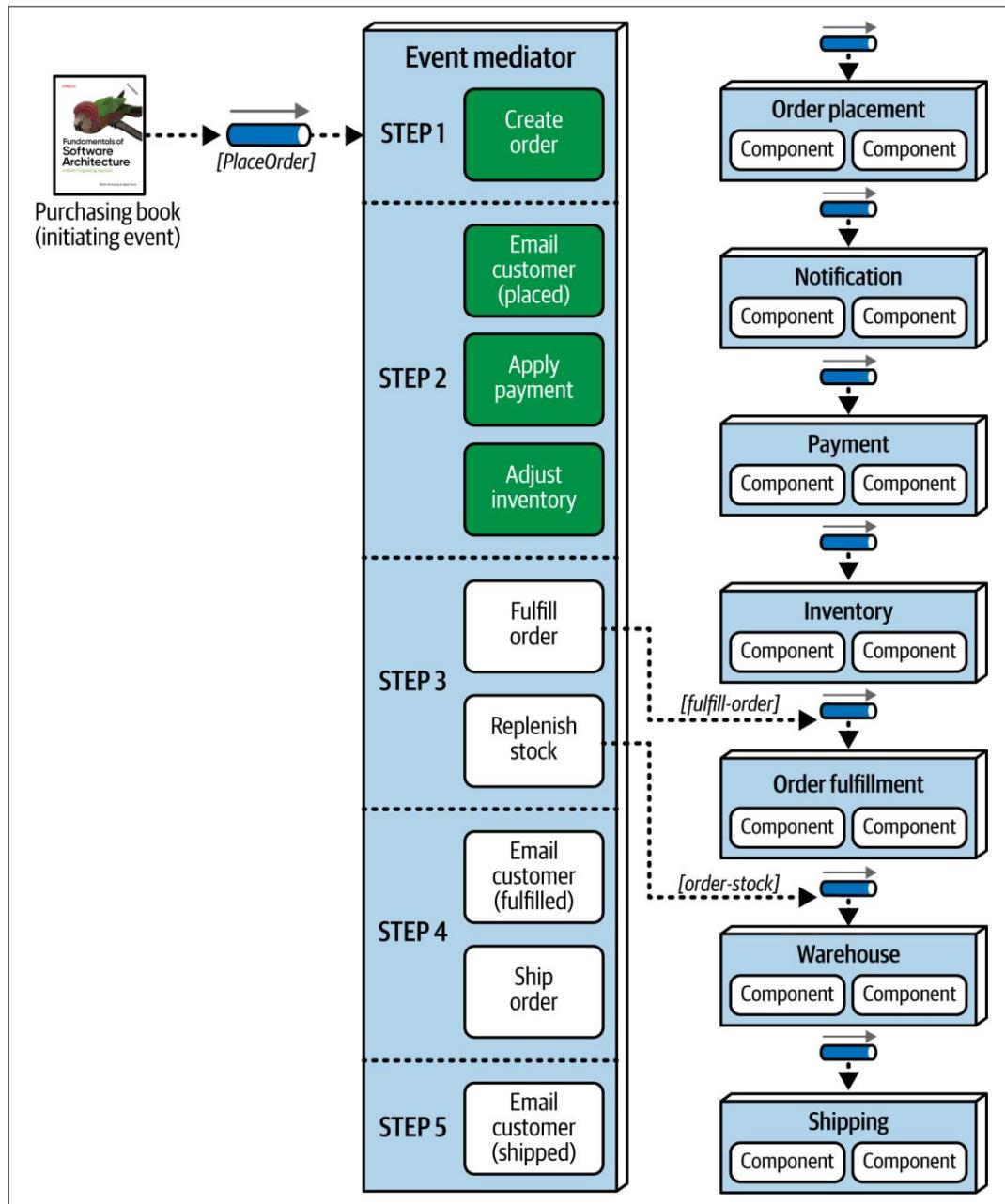


그림 15-30. 중재자 예시의 3단계

중재자는 4단계(그림 15-31 참조)로 이동하여 주문을 배송합니다. 이 단계에서는 두 가지 파생 메시지가 생성됩니다. 하나는 주문 배송 메시지이고, 다른 하나는 고객에게 보낼 이메일 메시지로, 고객에게 주문 배송 준비가 완료되었음을 알리는 구체적인 정보를 제공합니다.

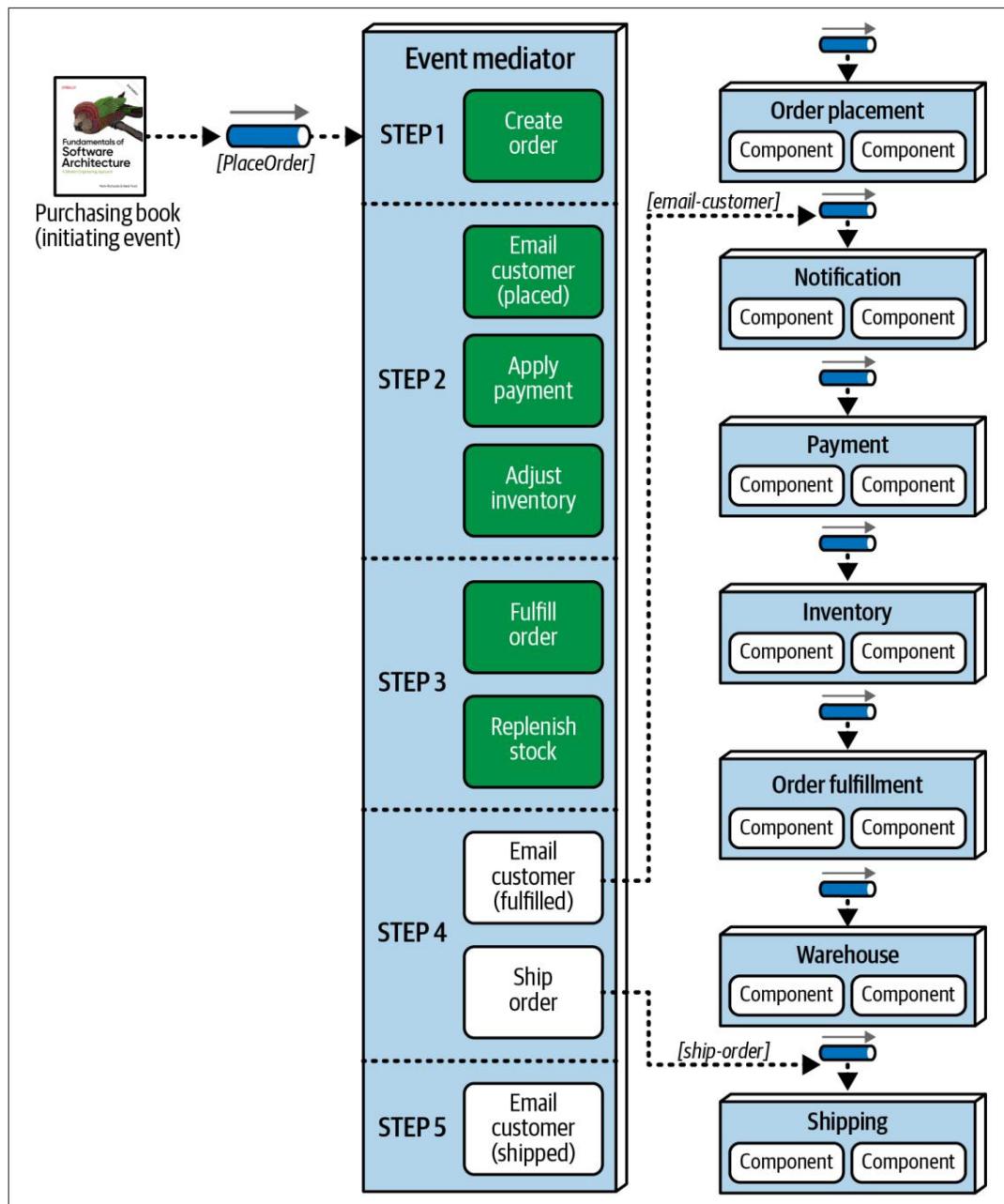


그림 15-31. 중재자 예시의 4단계

마지막으로, 중재자는 5단계([그림 15-32 참조](#))로 이동하여 주문이 배송되었음을 고객에게 알리는 상황별 이메일 메시지를 생성합니다. 이로써 워크플로가 종료됩니다. 중재자는 시작 이벤트 흐름이 완료되었음을 표시하고 시작 이벤트와 관련된 모든 상태를 제거합니다.

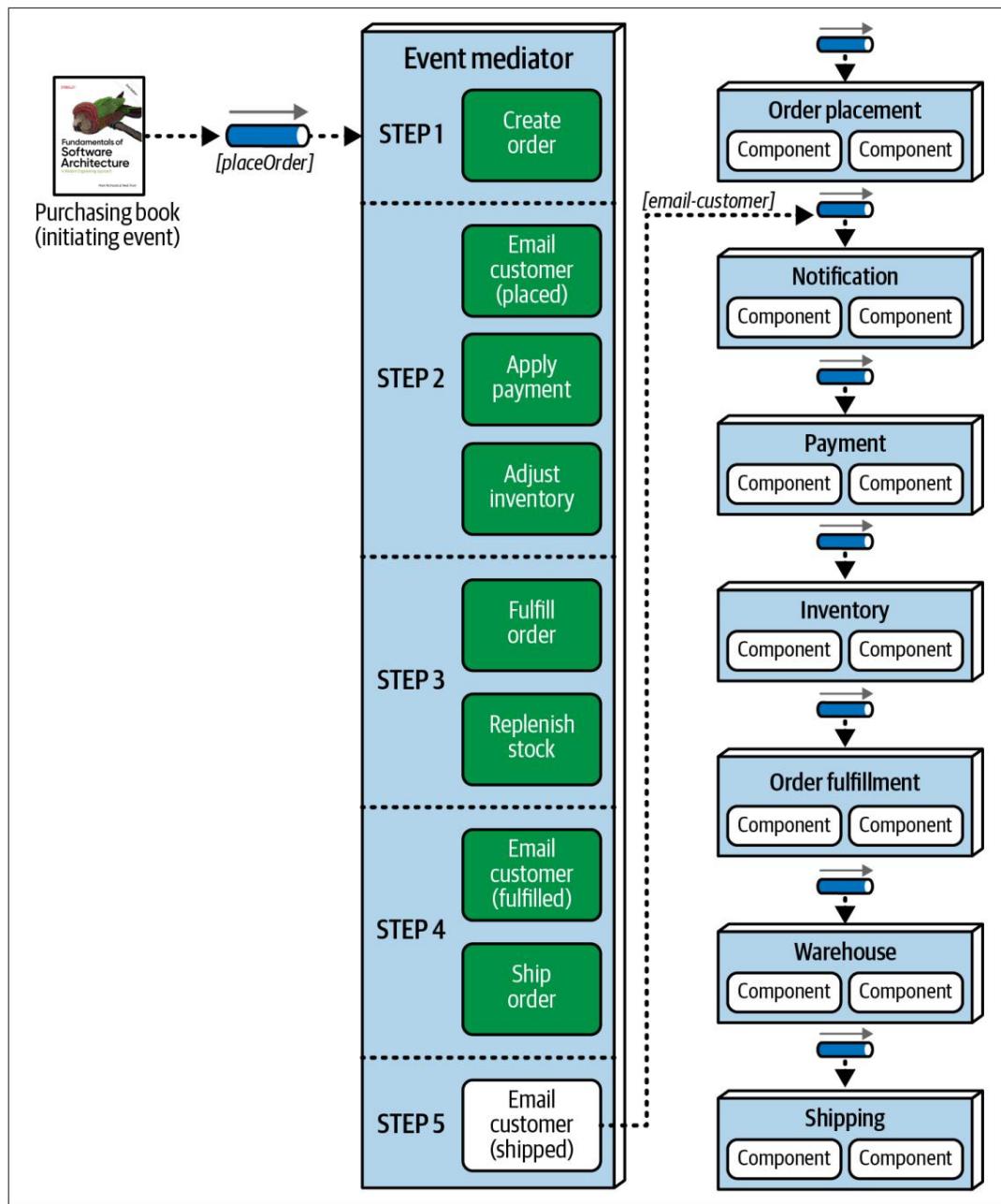


그림 15-32. 중재자 예시의 5단계

이 토플로지에서는 안무형 토플로지와 달리 미들웨어 구성 요소가 워크플로에 대한 정보와 제어 권한을 갖습니다. 미들웨어는 이벤트 상태를 유지하고 오류 처리, 복구 및 재시작 기능을 관리할 수 있습니다. 예를 들어, 앞의 예시에서 신용카드가 만료되어 결제가 적용되지 않았다고 가정해 보겠습니다. 미들웨어는 이 오류 조건을 수신하면 결제가 적용될 때까지 주문이 처리될 수 없다는 것을 알고(3단계) 워크플로를 중지하고 요청 상태를 자체 영구 데이터 저장소에 기록합니다. 결제가 완료되면 워크플로는 중단된 지점(이 경우 3단계 시작 부분)부터 다시 시작할 수 있습니다.

미디에이터 토플로지는 안무형 토플로지와 관련된 문제점을 해결하지만, 자체적인 단점도 있습니다. 우선, 복잡한 이벤트 흐름 내에서 발생하는 동적 처리를 선언적으로 모델링하는 것이 매우 어렵습니다. 따라서 미디에이터 토플로지 내의 많은 워크플로는 일반적인 처리만 수행하면서, 재고 부족이나 기타 비정형 오류와 같은 복잡한 이벤트 처리의 동적 특성을 처리하기 위해 미디에이터 토플로지와 안무형 토플로지를 결합한 하이브리드 모델을 사용합니다. 또한, 이벤트 프로세서는 안무형 토플로지와 마찬가지로 쉽게 확장할 수 있지만, 미디에이터 역시 확장해야 하므로 전체 이벤트 처리 흐름에서 병목 현상이 발생할 수 있습니다. 미디에이터 토플로지에서 이벤트 프로세서는 안무형 토플로지에서만큼 고도로 분리되어 있지 않습니다. 마지막으로, 미디에이터가 이벤트 처리를 제어하기 때문에 이 토플로지의 성능은 안무형 토플로지보다 떨어집니다.

안무형 토플로지와 중재형 토플로지 간의 절충점은 본질적으로 워크플로 제어 및 오류 처리 기능과 고성능 및 확장성 간의 균형에 달려 있습니다. 중재형 토플로지에서도 성능과 확장성은 여전히 우수하지만, 안무형 토플로지만 큼 높지는 않습니다.

데이터 토플로지

이벤트와 이벤트 처리에 대한 이야기가 많다 보니 EDA의 데이터 측면을 간과하기 쉽습니다. 데이터베이스 토플로지는 이 아키텍처 스타일의 독특하고 흥미로운 측면입니다. 다양한 옵션을 제공하며, 각 옵션은 상당한 장단점을 가지고 있어 전체 아키텍처에 큰 영향을 미칠 수 있습니다. EDA 내의 다양한 데이터베이스 토플로지를 설명하기 위해 [그림 15-3](#)에서 보여준 예시를 단순화한 버전을 사용하겠습니다 ([그림 15-33 참조](#)).

고객이 주문을 하면 주문 접수 이벤트 처리기가 주문을 생성하고 주문 완료 이벤트를 발생시킵니다. 이 이벤트에 대해 결제 및 재고 이벤트 처리기가 응답합니다. 결제가 완료되면 주문 처리 이벤트 처리기가 포장 담당자가 주문을 준비하도록 지원하고 주문 완료 이벤트를 발생시킵니다. 배송 이벤트 처리기는 주문 완료 이벤트에 응답하여 고객에게 주문을 배송함으로써 처리 과정을 완료하고 고객의 요청을 이행합니다.

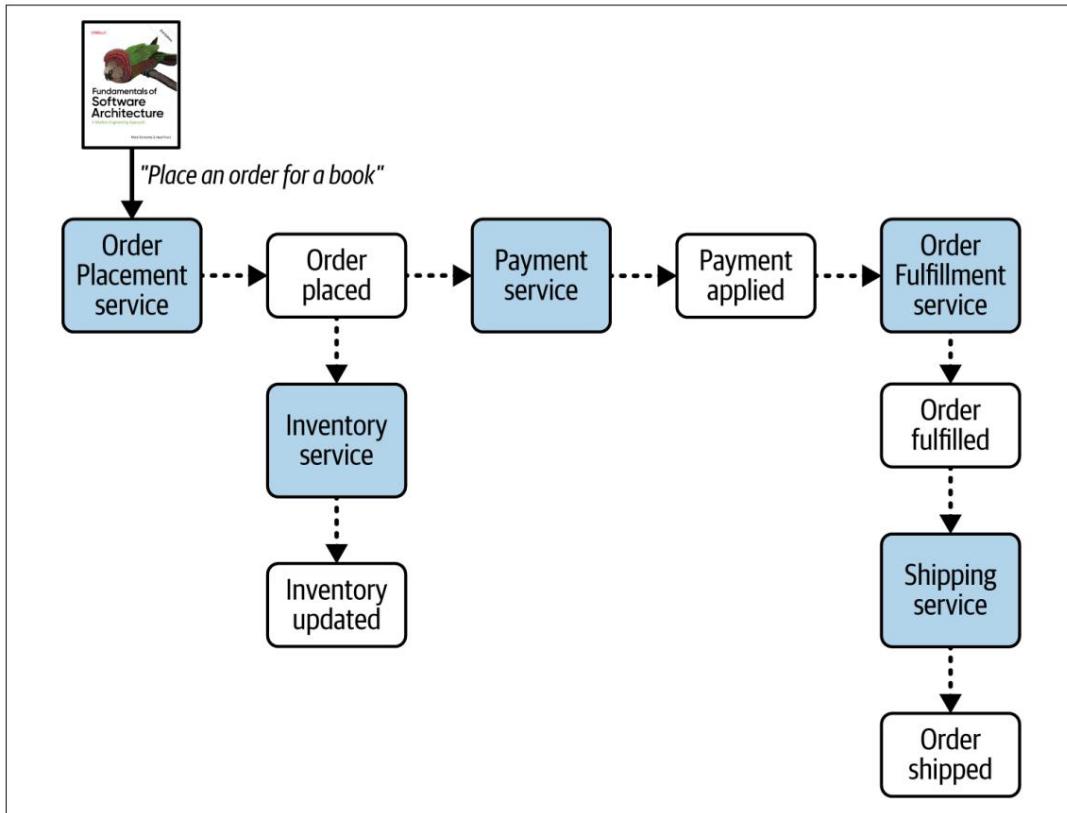


그림 15-33. EDA를 활용한 주문 입력 시스템의 간소화된 예시

EDA의 한 가지 복잡한 점은 주문 처리 이벤트 프로세서가 두 가지 정보를 알아야 한다는 것입니다. 하나는 현재 재고 수량이고, 다른 하나는 고객 위치에 따라 이용 가능한 배송 옵션입니다. 이 정보를 얻는 방식은 아키텍처에서 사용하는 데이터베이스 토플로지 유형에 따라 달라집니다. 각 데이터베이스 토플로지 옵션의 주요 장단점을 살펴보겠습니다.

단일체 데이터베이스 토플로지 EDA에서

가장 먼저, 그리고 아마도 가장 흔하게 사용되는 데이터베이스 토플로지는 단일체 데이터베이스 토플로지입니다. 이 토플로지에서는 모든 데이터가 중앙 데이터베이스를 통해 모든 이벤트 프로세서에서 사용 가능합니다.

단일체 데이터베이스 토플로지의 주요 이점은 모든 이벤트 프로세서가 다른 이벤트 프로세서와 동기적으로 통신할 필요 없이 데이터베이스에서 필요한 데이터를 직접 조회할 수 있다는 것입니다. 이벤트 기반 아키텍처는 비동기 통신을 통해 통신하는 고도로 분리된 이벤트 프로세서에 의존하기 때문에 이는 매우 중요한 장점입니다. 그림 15-34에서 주문 접수 이벤트 프로세서는 중앙 단일체 데이터베이스를 간단히 조회하여 현재 재고 수량과 고객의 배송 옵션을 가져올 수 있습니다.

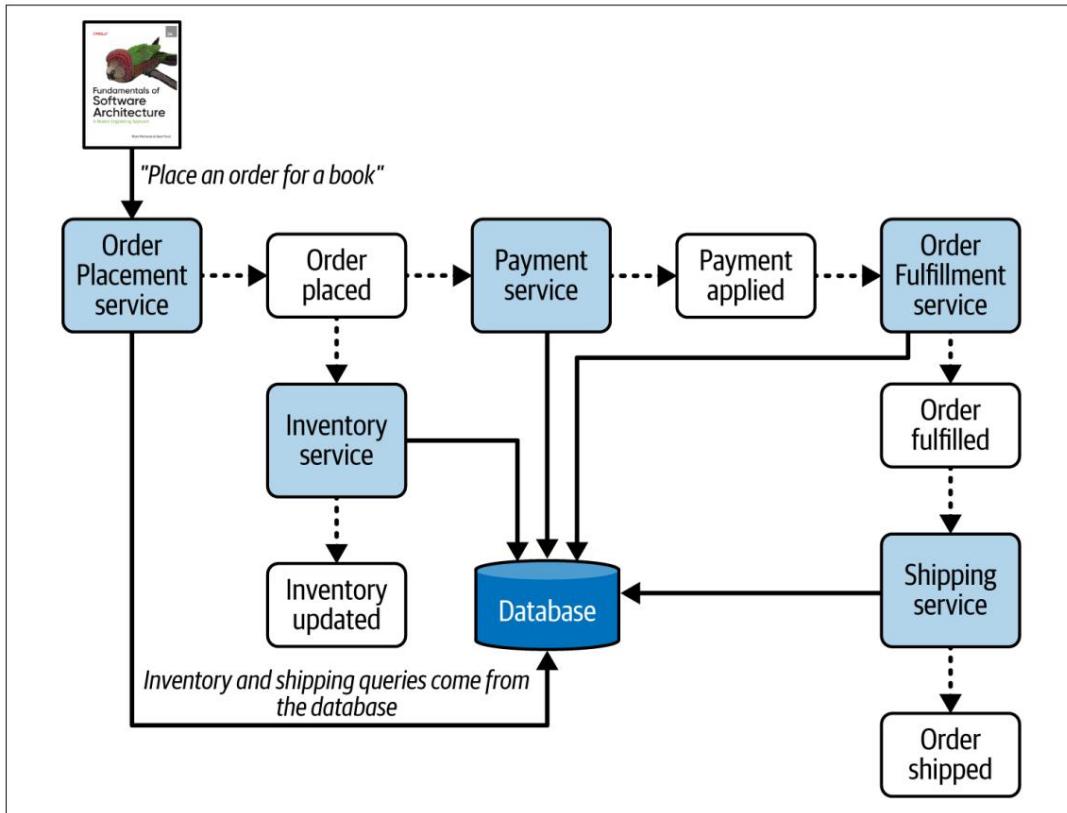


그림 15-34. 단일체 데이터베이스 토플로지에서는 데이터가 데이터베이스에서 직접 사용 가능합니다.

단일체 데이터베이스 토플로지는 이벤트 프로세서 간의 결합도를 낮추고 통신을 제한하는 장점이 있지만, 몇 가지 심각한 단점을 가지고 있습니다. 그중 첫 번째는 내결함성 문제입니다. 중앙 단일체 데이터베이스가 다운되거나 유지보수를 위해 가동이 중단되면 모든 이벤트 프로세서를 사용할 수 없게 됩니다.

두 번째 문제는 확장성입니다. 이벤트 기반 아키텍처는 비동기 통신을 사용하기 때문에 각 이벤트 프로세서는 다른 프로세서와 독립적으로 확장할 수 있습니다. 이벤트 채널은 기본적으로 백프레셔 지점 역할을 하여 개별 이벤트 프로세서가 다른 이벤트 프로세서의 확장 여부와 관계없이 필요에 따라 확장할 수 있도록 합니다. 그러나 모든 이벤트 프로세서가 동일한 데이터베이스에 동시에 쿼리를 실행하고 쓰기를 수행하는 경우 데이터베이스도 이러한 요구 사항을 충족할 수 있도록 확장되어야 합니다. 많은 데이터베이스는 높은 동시성 부하에서 이를 달성하지 못합니다.

세 번째 문제는 변경 관리입니다. 데이터베이스 구조가 변경될 때(예: 열이나 속성 삭제) 여러 이벤트 프로세서가 영향을 받으며, 단일 데이터베이스 변경에 대해서도 서로 협력해야 합니다.

마지막으로, 단일체 데이터베이스 토플로지는 공유되는 단일체 데이터베이스로 인해 필연적으로 단일 아키텍처 양자를 생성합니다.

도메인 데이터베이스 토플로지 EDA 내에

서 가능한 또 다른 데이터베이스 토플로지는 이벤트 프로세서를 다양한 도메인으로 그룹화하고 각 도메인이 자체 데이터베이스를 소유하는 도메인 데이터베이스 토플로지입니다 ([그림 15-35](#)).

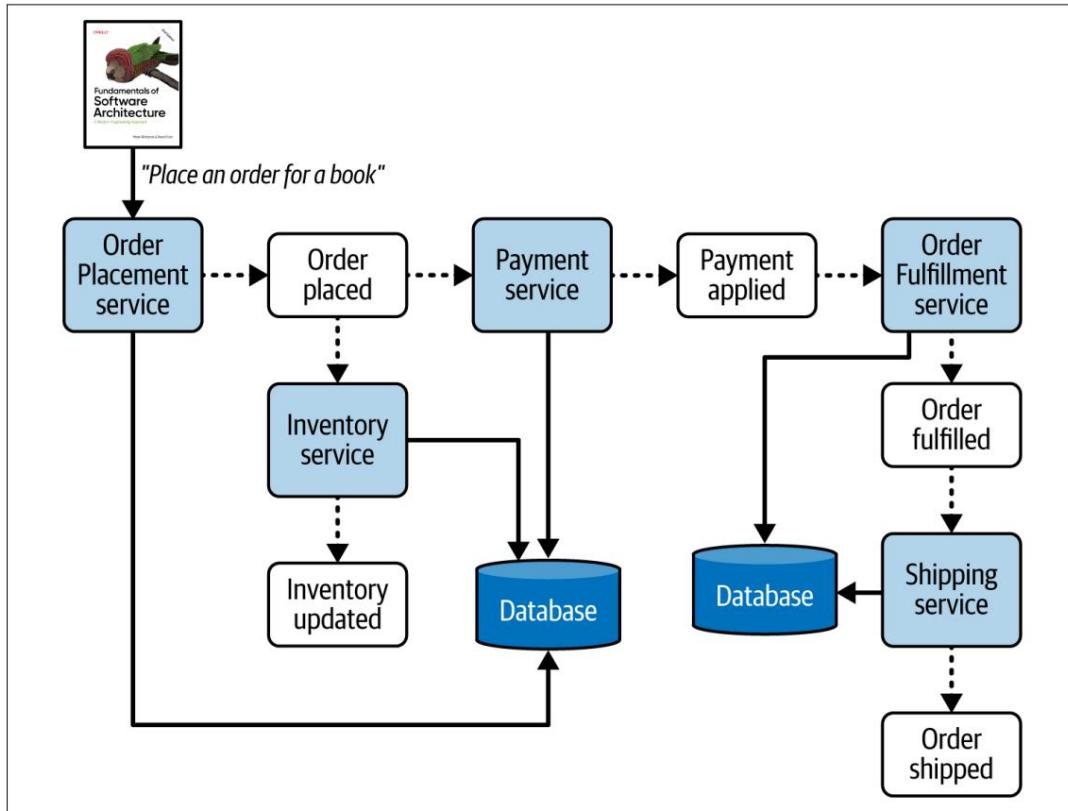


그림 15-35. 도메인 데이터베이스 토플로지는 각 도메인에 대해 별도의 데이터베이스를 사용합니다.

도메인 분할로 인해 단일체 데이터베이스 토플로지에 비해 도메인 데이터베이스 토플로지가 갖는 주요 이점은 내결함성, 확장성 및 변경 제어의 용이성입니다. [그림 15-35](#) 의 예에서 주문 처리 도메인 데이터베이스(주문 이행 및 주문 배송 이벤트를 포함하는 이벤트 채널은 백프레셔 지점 역할을 하여 주문 처리 데이터베이스를 사용할 수 있게 되더라도 주문 접수 도메인은 여전히 정상적으로 작동하며 주문을 계속 접수할 수 있습니다. 결제 적용 파생 이벤트를 포함하는 이벤트 채널은 백프레셔 지점 역할을 하여 주문 처리 데이터베이스를 사용할 수 있게 될 때까지 이벤트를 대기열에 저장합니다. 확장성과 변경 제어 측면에서도 마찬가지입니다. 각 도메인 데이터베이스는 해당 도메인에 특화된 이벤트 프로세서에 기반한 확장성만 고려하면 되며, 데이터베이스 구조가 변경되더라도 도메인 범위 이벤트 프로세서만 변경하면 됩니다.

하지만 주문 접수 이벤트 처리기가 필요로 하는 두 가지 정보, 즉 현재 보유 중인 도서 수량과 배송 옵션을 고려해 보세요.

도메인 데이터베이스 토플로지를 사용하면 주문 접수 이벤트 처리기는 단일 데이터베이스 토플로지에서와 마찬가지로 도메인 데이터베이스를 쿼리하여 도서 재고를 가져올 수 있습니다. 그러나 배송 옵션을 가져오기 위해 주문 배송 이벤트 처리기를 동기적으로 호출해야 하므로 이러한 서비스는 동기적으로 연결됩니다([그림 15-36 참조](#)).

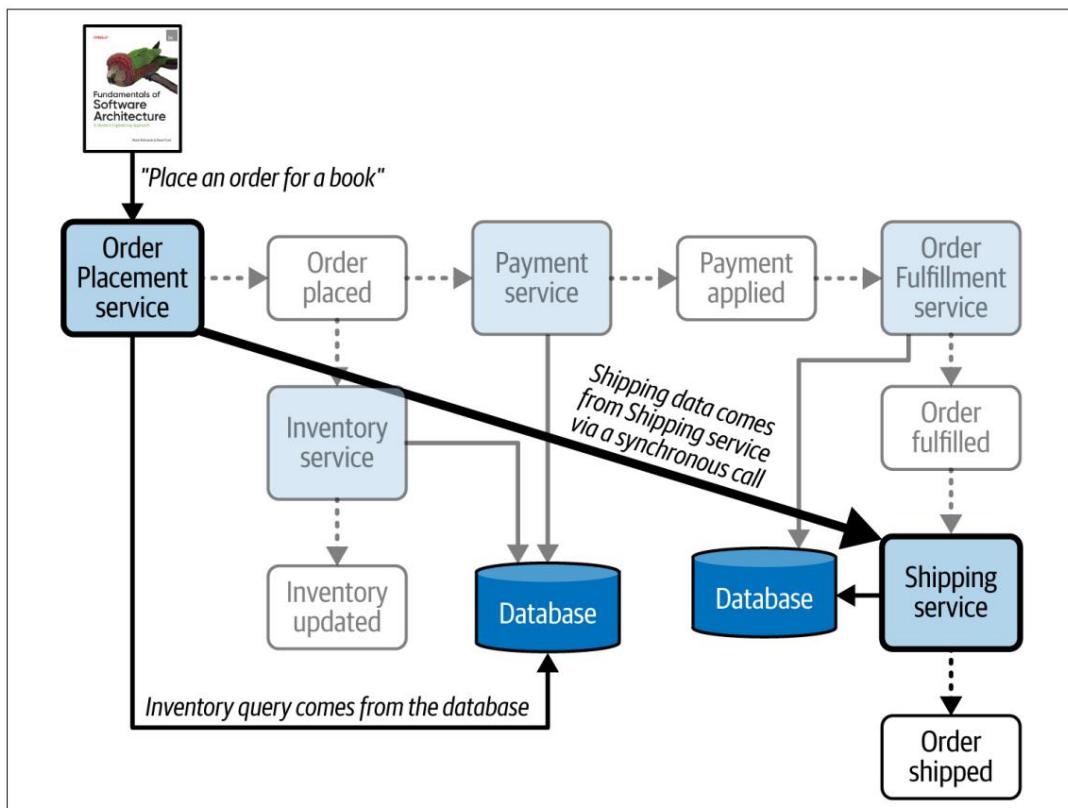


그림 15-36. 주문 접수 이벤트 처리기에 필요한 데이터는 다른 도메인의 이벤트 처리기와 동기식 통신을 필요로 할 수 있습니다.

EDA와 같이 고도로 동적이고 분리된 아키텍처에서는 동기적 결합을 최대한 피해야 합니다. 동기 호출은 내결함성과 확장성을 저해하여 이러한 토플로지의 장점을 상쇄할 수 있습니다. 도메인들이 서로 상당히 독립적으로 유지되도록 항상 확인하고, 서비스 간 동기 호출을 가능한 한 최소화해야 합니다. 이벤트 프로세서 간에 동기 통신이 지나치게 많이 필요한 경우, 도메인 경계를 재평가하거나, 도메인을 단일 도메인으로 통합하거나, 모놀리식 데이터베이스 토플로지로 전환하는 것을 고려해야 합니다.

전용 데이터 토플로지 EDA에서 사용

할 수 있는 또 다른 옵션은 전용 데이터베이스 토플로지입니다. 마이크로서비스 환경에서는 이를 서비스별 데이터베이스 패턴이라고도 합니다. 이 데이터베이스 토플로지에서는 각 이벤트 프로세서가 마이크로서비스와 유사하게 엄격하게 구성된 경계 컨텍스트 내에서 자체 전용 데이터베이스를 소유합니다(18 장 348 페이지의 "데이터 토플로지" 참조). 이 토플로지는 그림 15-37에 나와 있습니다 .

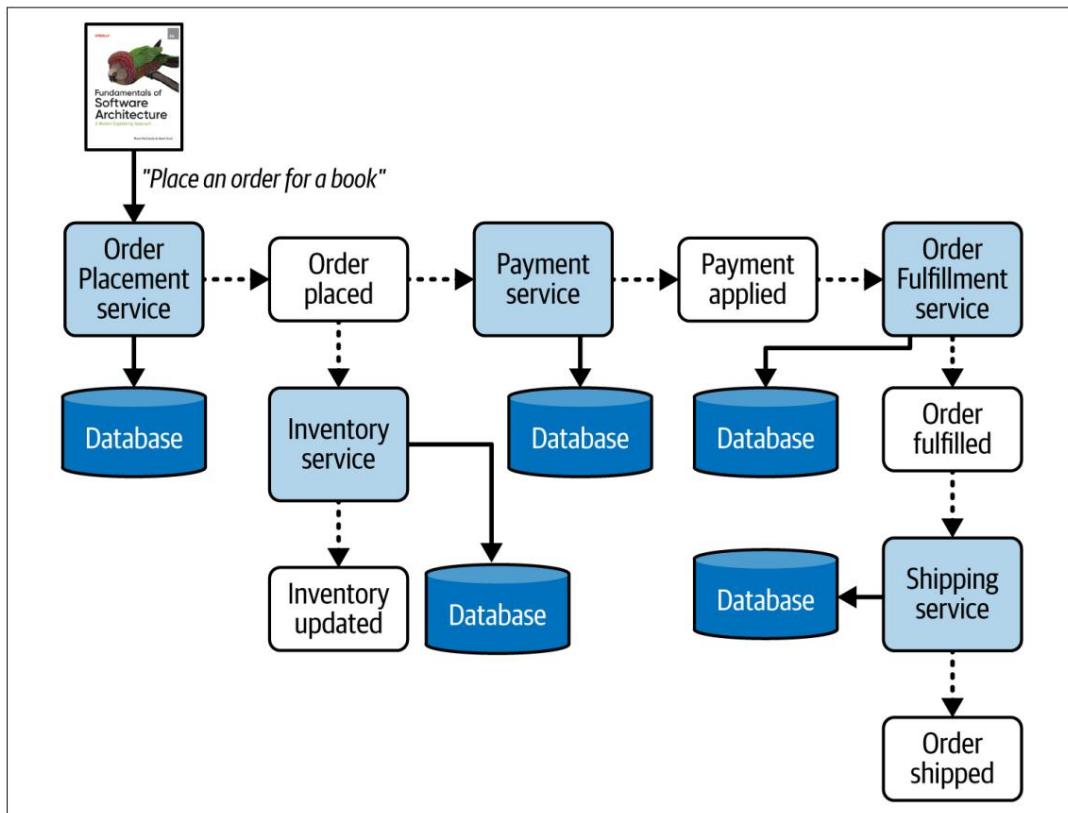


그림 15-37. EDA의 전용 데이터베이스 토플로지는 각 이벤트 프로세서에 대해 별도의 데이터베이스를 사용합니다.

당연히 전용 데이터베이스 토플로지는 사용 가능한 모든 토플로지 중에서 내결함성, 확장성 및 변경 제어 수준이 가장 높습니다. 이벤트 프로세서나 데이터베이스에 장애가 발생하더라도 해당 이벤트 프로세서에만 영향을 미치고 다른 모든 이벤트 프로세서는 정상적으로 작동합니다. 데이터베이스는 경계 컨텍스트 내의 단일 이벤트 프로세서에 따라서만 확장하면 되므로 이 토플로지는 이 섹션에서 논의된 세 가지 토플로지 중 가장 확장성이 뛰어납니다. 또한 데이터베이스 구조 변경은 해당 데이터베이스에 연결된 이벤트 프로세서에만 영향을 미칩니다.

단점으로는 데이터베이스 기술 스택에 따라 매우 비용이 많이 드는 옵션이 될 수 있다는 점입니다. 하지만 가장 큰 단점은 이벤트 프로세서 간의 동기식 동적 결합일 것입니다. 이러한 단점을 설명하기 위해 다음을 살펴보겠습니다.

주문 접수 이벤트 처리기가 처리에 필요한 두 가지 정보, 즉 도서 재고와 배송 옵션을 다시 살펴보겠습니다. 이 토플로지에서 주문 접수 처리기는 필요한 정보를 얻기 위해 재고 이벤트 처리기와 주문 배송 이벤트 처리기 모두에 동기 호출을 해야 하므로 아키텍처 전체에 긴밀한 동기 결합 지점이 형성됩니다(그림 15-38 참조). 도메인 데이터베이스 토플로지와 마찬가지로 이 데이터베이스 토플로지 옵션을 선택하기 전에 각 이벤트 처리기의 모든 데이터 관련 요구 사항을 파악해야 합니다.

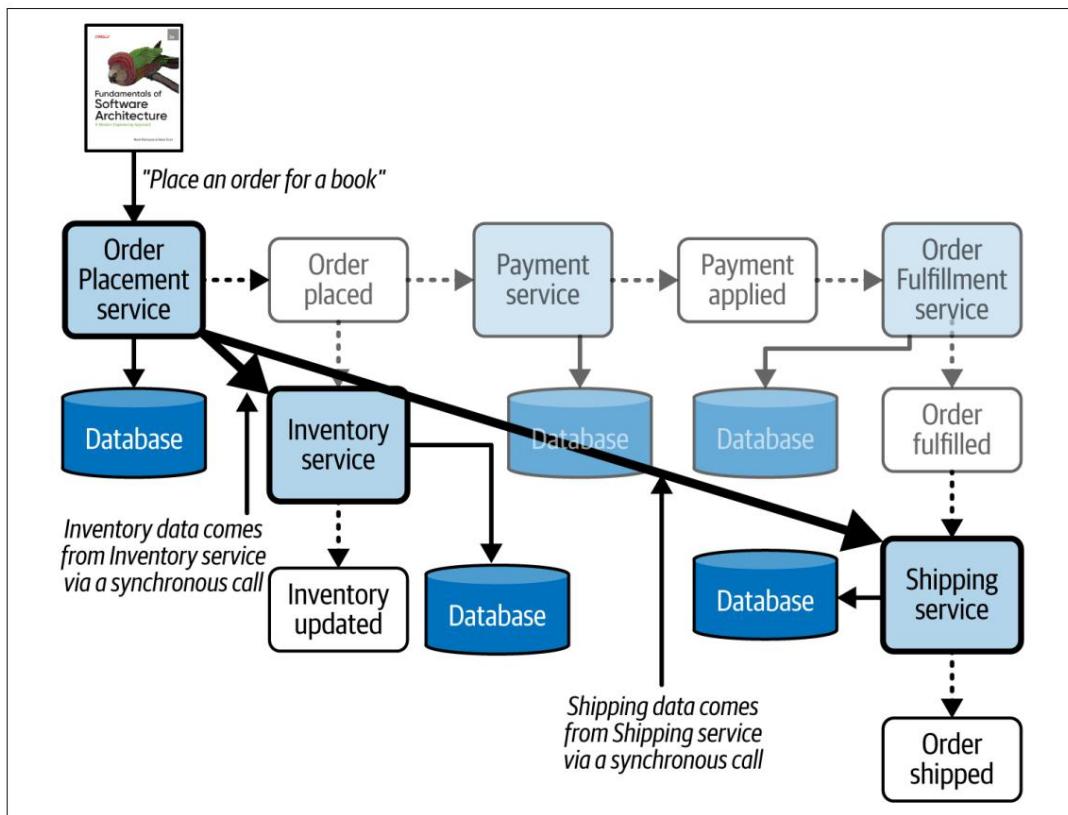


그림 15-38. 주문 접수 이벤트 처리기에 필요한 데이터는 다른 이벤트 처리기와 동기식 통신을 필요로 할 수 있습니다.

전용 데이터베이스 토플로지는 이벤트 프로세서가 대부분 자체적으로 작동하고, 경계 컨텍스트와 해당 데이터베이스 내의 데이터만 필요로 하는 경우에 적합한 선택입니다. 이벤트 프로세서 간의 통신량이 너무 많은 경우(276페이지의 "관리" 참조), 아키텍트는 전반적인 성능과 확장성을 향상시키기 위해 도메인 또는 모듈리식 데이터베이스 토플로지로 전환하는 것을 고려해야 합니다.

하지만 특정 상황에서 데이터베이스 구조가 자주 변경되는 경우, 영향을 받는 이벤트 처리기의 수를 최소화하기 위해 이러한 운영 특성 간의 절충이 필요할 수 있습니다.

클라우드 고려 사항

이벤트 기반 아키텍처(EDA)는 높은 수준의 분리성 덕분에 클라우드 기반 환경 및 구현에 매우 적합합니다. EDA는 클라우드 공급업체가 제공하는 비동기 서비스를 손쉽게 활용할 수 있으며, 클라우드 인프라 및 클라우드 기반 서비스의 탄력적인 특성이 EDA의 형태와 잘 맞아떨어집니다. 요컨대, 클라우드 기반 환경은 EDA에 최적의 선택입니다.

일반적인 위험

EDA와 관련된 주요 위험 중 하나는 비결정적 이벤트 처리로 인해 부작용이 발생할 수 있다는 점입니다. 예를 들어, 이벤트 프로세서가 예기치 않게 파생 이벤트를 트리거하거나, 응답해야 할 때 이벤트에 응답하지 않는 경우가 있습니다. 이벤트 기반 아키텍처에서 이벤트 워크플로는 매우 복잡해질 수 있으며, 이벤트가 트리거될 때 정확히 어떤 일이 발생할지 예측하기 어려운 경우가 많습니다.

또 다른 큰 위험은 이벤트 기반 아키텍처 내에서 정적 결합도가 지나치게 높아 취약해지는 것입니다. EDA는 동적 결합도가 매우 높지만, 이벤트 페이로드 계약(240페이지의 "이벤트 페이로드" 참조) 또한 정적 결합도를 높일 수 있습니다. 아키텍트는 특정 이벤트에 어떤 이벤트 프로세서가 응답하는지 항상 알 수 없기 때문에 계약을 변경하는 것은 매우 어려운 작업입니다. 이벤트 페이로드 계약이 변경되면 다른 여러 이벤트 프로세서에 부정적인 영향을 미쳐 이러한 아키텍처 스타일의 전반적인 취약성을 더욱 악화시킬 수 있습니다. 키 기반 이벤트 페이로드는 이러한 위험을 완화하는 데 도움이 되지만, 확장성 및 성능 문제, 그리고 빈약한 이벤트 발생 가능성(240페이지의 "이벤트 페이로드" 참조)과 같은 위험이 존재합니다.

이벤트 프로세서 간의 과도한 동기 통신에도 주의해야 합니다. 이벤트 기반 아키텍처는 고도로 동적으로 분리된 이벤트 프로세서 덕분에 강력한 성능을 발휘합니다. 하지만 이벤트 프로세서들이 계속해서 동기적으로 통신해야 한다면, 이벤트 기반 아키텍처가 가장 적합한 아키텍처 스타일이 아닐 가능성이 높습니다.

마지막으로, 전반적인 상태 관리는 EDA에서 위험 요소이자 과제입니다. 시작 이벤트가 완전히 처리된 시점을 아는 것은 중요하지만, EDA의 비결정적이고 비동기적인 병렬 이벤트 처리 방식 때문에 이를 파악하기가 매우 어렵습니다. 경우에 따라 아키텍트는 최종 처리 지점을 식별하고 시작 이벤트를 수락한 이벤트 프로세서가 해당 "종료" 이벤트를 구독하도록 할 수 있지만, 대부분의 경우 이를 파악하기 어렵습니다. 결과적으로 시작 이벤트가 완전히 처리된 시점이나 현재 상태조차 파악하기 어렵습니다.

통치

EDA와 관련된 거버넌스의 대부분은 비구조적이며, 전체적인 거버넌스 체계의 일부로서 로그 형태의 관찰 가능성을 필요로 합니다. 인프라 및 환경에 따라 EDA와 관련된 일부 거버넌스 지표는 수동으로 수집해야 할 수도 있습니다.

이러한 스타일의 거버넌스에서 핵심적인 두 가지 영역은 계약 관리를 통한 정적 결합과 동기 호출을 통한 동적 결합입니다. 이 두 가지 측면 모두 EDA에서 구조적 결함으로 간주되므로 주의 깊게 살펴봐야 합니다.

정적 결합 관점에서 아키텍트는 이벤트 페이로드 계약의 변경 속도 및 전체 스템프 결합과 같은 요소에 대한 거버넌스를 설정할 수 있습니다. EDA는 본질적으로 결합도가 낮기 때문에 계약 변경은 매우 위험할 수 있습니다. 특히 스 키마가 연결되지 않은 이벤트 계약을 변경하면 하위 이벤트 처리기가 제대로 작동하지 않을 수 있습니다. 이는 비결 정적인 엔드투엔드 이벤트 흐름을 테스트하기가 매우 어렵기 때문에 EDA에서 특히 위험합니다.

스탬프 결합(240페이지의 "이벤트 페이로드" 참조)은 이벤트에 응답하는 이벤트 프로세서가 이벤트 계약에서 사용하지 않는 필드를 지속적으로 기록하고 관찰함으로써 관리할 수 있습니다. 이러한 사용되지 않는 필드를 관찰하면 계약 크기를 줄이고 대역폭을 감소시키며 스템프 결합과 그에 따른 이벤트 프로세서의 불필요한 변경 사항을 관리하는 데 도움이 됩니다.

동적 결합 관점에서 아키텍트는 자동화된 적합성 함수를 작성하여 로그 및 기타 관찰 가능한 수단(예: 소스 코드 주석 또는 표준 동기식 사용자 지정 식별자 라이브러리 사용)을 통해 이벤트 프로세서 간의 동기식 통신을 관찰하고 추적할 수 있습니다. 이벤트 기반 아키텍처에서 동기식 통신은 특히 도메인 또는 전용 데이터베이스 토플로지를 사용하는 경우, 그 필요성을 확인하기 위해 추적 및 논의되어야 합니다.

팀 토플로지 고려 사항

EDA는 각 도메인을 구성하는 수많은 아티팩트(다수의 이벤트 프로세서, 이벤트 채널, 메시지 브로커, 그리고 데이터베이스 토플로지에 따라 데이터베이스까지)로 인해 기술적으로 분할된 아키텍처로 간주됩니다. 그럼에도 불구하고, 팀이 도메인 영역 내에서 정렬되어 있는 경우(예: 전문성을 갖춘 교차 기능 팀) EDA는 효과적으로 작동할 수 있습니다. 하지만 특정 유형의 팀 토플로지([151페이지의 "팀 토플로지와 아키텍처" 참조](#))에서는 EDA를 구현하는 데 어려움이 있을 수 있습니다.

다음은 이러한 특정 팀 구성과 EDA 간의 정렬과 관련하여 고려해야 할 몇 가지 사항입니다.

스트림 중심 팀은 시스

템 규모에 따라 이벤트 프로세서의 분산된 특성으로 인해 도메인 기반 변경 사항을 구현하는 데 어려움을 겪을 수 있습니다. EDA에서 도메인과 하위 도메인은 일반적으로 여러 이벤트 프로세서와 파생 이벤트로 구현되므로 스트림 중심 팀은 이러한 모든 구성 요소를 이해하는 데 어려움을 느낄 수 있습니다. 예를 들어, 주문 처리 워크플로에 단계를 추가하려면 여러 이벤트 프로세서를 변경해야 할 뿐만 아니라 기존 파생 이벤트가 트리거되는 방식(및 시점)을 재구성해야 할 수 있습니다. 이벤트 기반 아키텍처가 크고 복잡할수록 스트림 중심 팀의 효율성은 떨어집니다.

활성화 팀은 파

생 이벤트와 그 계약에 기반한 이벤트 프로세서 간의 통합이 필요하기 때문에 이벤트 기반 아키텍처에서 제대로 작동하지 않습니다. 스트림 내에서 활성화 팀이 실험과 효율성을 추구하는 과정에서 스트림 중심 팀이 전체 이벤트 흐름을 이해하고 관리하는 데 방해가 될 수 있으며, 일반적으로 스트림 중심 팀과 활성화 팀 간의 과도한 조정이 필요합니다.

복잡한 하위 시스템 팀

복잡한 서브시스템 팀은 EDA의 분리된 비동기적 특성 덕분에 EDA와 협업하기에 적합합니다. 복잡한 처리는 별도의 이벤트 프로세서를 통해 쉽게 분리할 수 있으며, 복잡한 서브시스템 팀은 이러한 처리에 집중하고, 덜 복잡한 처리는 스트림 지향 팀에 맡길 수 있습니다. 이벤트 프로세서는 고도로 동적으로 분리되어 있기 때문에, 스트림 지향 팀은 정적 이벤트 페이로드 계약 및 파생 이벤트에 대해서만 복잡한 서브시스템 팀과 협력하면 됩니다.

EDA에서 플랫폼

팀은 개발자가 공통 도구, 서비스, API 및 작업을 활용하여 플랫폼 팀 토탈로지의 이점을 누릴 수 있도록 지원합니다. 이는 주로 EDA의 기술적 분할 덕분입니다. 특히 팀에서 메시지 브로커, 이벤트 허브, 이벤트 버스 및 기타 이벤트 채널 아티팩트와 같은 EDA의 인프라 관련 부분을 플랫폼 관련 요소로 취급하는 경우 더욱 그렇습니다.

스타일 특징

그림 15-39의 특성 평가표에서 별 1개는 해당 아키텍처 특성이 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 스타일의 가장 강력한 특징 중 하나임을 의미합니다. 평가표에 제시된 각 특성에 대한 정의는 4 장에서 확인할 수 있습니다.

이벤트 기반 아키텍처는 본질적으로 기술적으로 분할된 아키텍처입니다. 즉, 특정 도메인이 여러 이벤트 프로세서에 분산되어 있고 서로 연결되어 있습니다.

브로커, 계약(이벤트 페이로드) 및 토픽을 통해 전달됩니다. 특정 도메인의 변경 사항은 일반적으로 여러 이벤트 프로세서 및 기타 메시징 아티팩트에 영향을 미치므로 EDA는 일반적으로 도메인 분할 방식으로 간주되지 않습니다.

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

그림 15-39. EDA 특성 등급

EDA 내의 퀸텀 수는 각 이벤트 프로세서 내의 데이터베이스 상호 작용 및 시스템이 요청-응답 처리를 사용하는지 여부에 따라 하나에서 여러 개까지 다양할 수 있습니다. EDA에서의 통신은 비동기 호출에 기반하지만, 여러 이벤트 프로세서가 단일 데이터베이스 인스턴스를 공유하는 경우 모두 동일한 아키텍처 퀸텀 내에 포함됩니다. 요청-응답 처리의 경우에도 마찬가지입니다. 이벤트 프로세서 간의 통신은 여전히 비동기적이지만, 이벤트 소비자가 즉시 응답해야 하는 경우 해당 이벤트 프로세서들을 동기적으로 연결하여 단일 퀸텀을 형성합니다.

예를 들어, 한 이벤트 프로세서가 다른 이벤트 프로세서에게 주문을 처리해 달라는 요청을 보낸다 고 가정해 보겠습니다. 첫 번째 이벤트 프로세서는 다른 이벤트 프로세서로부터 주문 ID를 받아야 만 작업을 계속할 수 있습니다. 만약 두 번째 이벤트 프로세서(주문을 처리하고 주문 ID를 생성하는 프로세서)가 다운되면 첫 번째 이벤트 프로세서는 작업을 계속할 수 없습니다. 즉, 이들은 동일한 아키텍처 퀸텀에 속하며 리소스를 공유합니다.

둘 다 비동기 메시지를 송수신하지만, 동일한 아키텍처적 특성을 가지고 있습니다.

이벤트 기반 아키텍처(EDA)는 성능, 확장성, 내결합성 측면에서 매우 높은 평점(별 4~5개)을 받았으며, 이는 EDA의 주요 강점입니다. 높은 성능은 비동기 통신과 고도로 병렬화된 처리를 결합하여 달성됩니다. 높은 확장성은 이벤트 프로세서(경쟁 소비자 또는 소비자 그룹이라고도 함)의 프로그래밍 방식 로드 밸런싱을 통해 구현됩니다. 요청 부하가 증가함에 따라 추가 요청을 처리하기 위해 추가 이벤트 프로세서를 프로그래밍 방식으로 추가할 수 있습니다. 별 5개가 아닌 4개를 준 이유는 데 이터베이스 때문입니다(이러한 특성에 대해 별 5개를 받은 예는 [16장의](#) 공간 기반 아키텍처 부분을 참조하십시오). EDA는 분리된 비동기 이벤트 프로세서를 통해 내결합성을 확보하며, 이는 최종 일관성과 이벤트 워크플로 처리를 제공합니다. 하위 프로세서를 사용할 수 없는 경우, 사용자 인터페이스 또는 요청을 보내는 이벤트 프로세서가 즉각적인 응답을 필요로 하지 않는다면 시스템은 나중에 이벤트를 처리할 수 있습니다.

EDA는 전반적인 단순성과 테스트 용이성 측면에서 상대적으로 낮은 평가를 받는데, 이는 주로 비결정적이고 동적인 이벤트 흐름 때문입니다. 요청 기반 모델에서는 결정적 흐름의 경로와 결과가 일반적으로 알려져 있기 때문에 테스트가 비교적 쉽습니다.

하지만 이벤트 기반 모델에서는 그렇지 않습니다. 때때로 아키텍트는 이벤트 프로세서가 동적 이벤트에 어떻게 반응할지, 또는 어떤 메시지를 생성할지 예측하지 못하는 경우가 있습니다. 이러한 시스템을 비결정적 워크플로라고 합니다. 이러한 시스템의 "이벤트 트리 다이어그램"은 매우 복잡해져 수백, 심지어 수천 개의 시나리오를 생성할 수 있으므로 관리 및 테스트가 매우 어렵습니다.

마지막으로, EDA는 진화 가능성이 매우 높기 때문에 별 5개 만점을 줄 수 있습니다. 기존 또는 새로운 이벤트 프로세서를 통해 새로운 기능을 추가하는 것은 비교적 간단합니다. 트리거된 파생 이벤트를 통해 후크를 제공하면 이벤트와 해당 데이터가 이미 다른 처리에서 사용 가능하므로 새로운 기능을 추가하기 위해 인프라나 기존 이벤트 프로세서를 변경할 필요가 없습니다.

EDA의 단점으로는 시작 이벤트와 관련된 전체 워크플로를 제어하기 어렵다는 점이 있습니다. 이벤트 처리는 조건 변화에 따라 매우 역동적으로 진행되며, 시작 이벤트에 기반한 비즈니스 트랜잭션이 언제 완료되었는지 파악하기 어렵습니다.

EDA에서 오류 처리는 큰 과제입니다. 일반적으로 (중재자 토플로지를 제외하고는) 비즈니스 트랜잭션을 모니터링하거나 제어하는 중개자가 없기 때문에 오류가 발생하면 다른 서비스는 오류를 인지하지 못합니다. 해당 시작 이벤트에 기반한 비즈니스 프로세스는 멈춰버리고 자동 또는 수동 개입 없이는 진행할 수 없게 됩니다. 반면 다른 모든 프로세스는 오류를 무시하고 계속 진행됩니다. 예를 들어, 주문 시스템의 결제 이벤트 처리기가 충돌하여 할당된 작업을 완료하지 못하면 재고 이벤트도 마찬가지로 중단됩니다.

프로세서는 여전히 재고를 조정하고, 이후의 모든 이벤트 프로세서는 모든 것이 정상인 것처럼 반응합니다.

EDA에서 비즈니스 거래를 재개(복구 가능성)하는 것은 매우 어렵습니다.

시작 이벤트 처리 과정에서 다른 작업들이 비동기적으로 수행되었기 때문에, 시작 이벤트를 다시 제출하는 것이 불가능한 경우가 많습니다.

요청 기반 모델과 이벤트 기반 모델 중 선택하기 요청 기반 모델과 이벤트 기반 모델은 모두 소프트웨어 시스템 설계에 적합한 접근 방식입니다. 하지만 성공을 위해서는 올바른 모델을 선택하는 것이 중요합니다. 고객 프로필 데이터 조회와 같이 구조가 잘 짜여 있고 데이터 기반의 요청이 필요한 경우, 워크플로에 대한 확실성과 제어가 우선이라면 요청 기반 모델을 선택하는 것이 좋습니다. 반면, 높은 응답성과 확장성이 요구되고 복잡하고 동적인 사용자 처리가 필요한 유연하고 액션 기반의 이벤트에는 이벤트 기반 모델을 선택하는 것이 좋습니다.

이벤트 기반 모델의 장단점을 이해하는 것은 최적의 모델을 결정하는 데에도 도움이 됩니다. 표 15-2는 이벤트 기반 모델의 장점과 단점을 나열합니다.

EDA.

표 15-2. 이벤트 기반 모델의 장단점

요청 기반 방식 대비 장점		그것들을 고민하세요
동적 사용자 콘텐츠에 대한 더 나은 응답은 최종 일관성만 지원합니다.		
확장성과 유연성 향상	처리 과정에 대한 제어력이 떨어짐	
향상된 민첩성과 변화 관리 능력, 이벤트 진행 결과에 대한 불확실성 감소		
더 나은 적응성과 확장성	테스트 및 디버깅이 어렵습니다.	
향상된 반응성과 성능		
더욱 향상된 실시간 의사 결정		
상황 인식에 대한 더 나은 반응		

예시 및 사용 사례

시스템 내부 또는 외부에서 발생하는 상황에 대응하는 데 초점을 맞춘 모든 비즈니스 문제는 EDA(탐색적 설계)를 적용하기에 좋은 대상입니다. 이 장 전체에서 사용한 주문 입력 시스템 예시는 주문 처리를 병렬적이고 독립적으로 수행할 수 있다는 점에서 EDA의 좋은 활용 사례입니다. 높은 수준의 응답성, 성능, 확장성, 내결함성 및 탄력성이 요구되는 시스템 또한 EDA를 적용하기에 적합합니다.

EDA의 강력함과 효율성을 보여주는 또 다른 좋은 사용 사례는 현재 진행 중인 'Going, Going, Gone' 경매 시스템입니다. 이 시스템에서는 사용자들이 경매에 나온 상품에 입찰하고, 최종 입찰에서 이의가 없으면 해당 상품을 낙찰받게 됩니다.

이러한 시스템에서는 입찰자 수를 알 수 없는 경우가 많으므로 확장성과 유연성이 모두 필요합니다. 특히 경매에 시간 제한이 있고 입찰이 마감되는 경우에는 더욱 그렇습니다.

이러한 시스템은 높은 수준의 응답성 또한 필요로 합니다. 온라인 입찰 시스템과 EDA가 잘 어울리는 가장 큰 이유는 EDA가 입찰을 시스템에 대한 요청이 아니라 발생한 이벤트로 간주한다는 점입니다.

그림 15-40 에서 볼 수 있듯이, 입찰자가 입찰을 제출하면 시스템에서 여러 가지 작업이 수행됩니다. 이러한 작업은 모두 비동기적으로 처리될 수 있으며, 동시에 또는 나중에 백엔드 처리(예: 입찰자 추적 이벤트 프로세서)를 통해 수행될 수 있습니다. 입찰자가 입찰을 제출하면 입찰 캡처 이벤트 프로세서가 해당 시작 이벤트를 수신하고, 이전 입찰가보다 높은지 판단하여 입찰 완료 이벤트를 발생시킵니다. 경매 진행자 이벤트 프로세서는 해당 이벤트에 응답하여 웹사이트에 새로운 입찰 가격을 업데이트합니다. 동시에 입찰 스트리머 이벤트 프로세서도 동일한 이벤트에 응답하여 입찰 정보를 웹사이트 입찰 내역 또는 개별 입찰자에게 스트리밍합니다(UI에 따라 다름). 마지막으로 입찰자 추적 이벤트 프로세서는 입찰자와 입찰 정보를 추적 및 감사 목적으로 저장합니다.

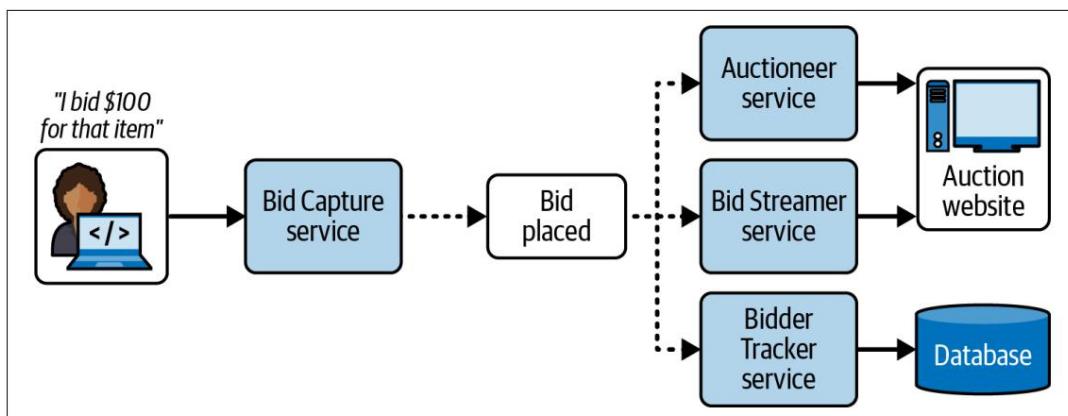


그림 15-40. EDA를 활용한 온라인 입찰 시스템 예시

이 시작 이벤트의 워크플로에는 다른 많은 이벤트 처리기가 관여하고, 그로부터 파생된 여러 이벤트가 발생하지만, 시스템의 이 작은 부분은 이벤트 기반 아키텍처 스타일을 잘 활용하여 응답성, 내결 합성, 확장성 및 유연성을 보여줍니다.

이벤트 기반 아키텍처(EDA)는 매우 복잡한 아키텍처 스타일이지만, 동시에 매우 강력한 스타일이기도 합니다. EDA의 복잡성을 감수할 가치가 있는지 판단하기 위해서는 비즈니스 문제에 필요한 워크플로와 처리 과정을 면밀히 분석해야 합니다. 필요한 처리 과정의 대부분이 요청 기반이라면, [18 장](#)에서 다루는 마이크로서비스 아키텍처 스타일을 고려하는 것이 좋습니다.

