제19장

적절한 것을 선택하세요 건축 양식

상황에 따라 다릅니다! 수많은 선택지(그리고 거의 매일 새로운 선택지가 등장하고 있습니다) 중에서 어떤 아키텍처 스타일을 사용해야 하는지 단정적으로 말씀드리기는 어렵습니다. 조직 내부 환경과 개발하려는 소프트웨어의 특성 등 여러 요인에 따라 최적의 아키텍처 스타일을 결정해야 하기 때문입니다. 아키텍처 스타일 선택은 아키텍처 특성, 도메인 고려 사항, 전략적 목표 등 다양한 요소를 분석하고 장단점을 따져보는 전체 과정을 거쳐야 합니다. 바로 이러한 이유로 2 장 에서 말씀드렸듯이 , 상황에 따라 다릅니다.

결정의 맥락이 어떠하든, 이 장에서는 적절한 건축 양식을 선택하는 데 도움이 되는 몇 가지 일반적인 조언을 제공합니다.

건축계의 변화하는 "패션"

소프트웨어 업계의 아키텍처 스타일 선호도는 여러 요인에 의해 시간이 지남에 따라 변화합니다. 이러한 요인에는 다음이 포함됩니다.

과거에 대한 관찰을 통해 새로운 건

축 양식은 일반적으로 과거 경험, 특히 문제점에 대한 관찰에서 비롯됩니다. 건축가들이 시스템을 다루면서 얻은 경험(이러한 경험은 종종 건축가가 되기로 결심하게 된 계기가 되기도 합니다)은 미래 시스템에 대한 우리의 생각에 영향을 미칩니다. 새로운 건축 디자인은 종종 과거 건축 양식에서 발견된 특정 결함을 개선한 결과물을 반영합니다.

예를 들어, 코드 재사용을 중심으로 하는 아키텍처를 구축한 후, 아키텍트들은 부정적인 절충점을 깨닫고 코드 재사용의 의미를 진지하게 재고하게 되었습니다.

소프트웨어 개발 생태계는 끊임
없이 변화합니다. 이러한 지속적인 변화는 너무나 예측 불가능해서 다음에 어떤 변화가 올지조차 예상하기 어렵습니다. 불과 몇 년 전만 해도 쿠버네티스가 무엇인지 아는 사람은 거의 없었지만, 지금은 많은 개발자들의 일상에서 빼놓을 수 없는 요소가 되었습니다. 그리고 몇 년 후에는 쿠버네티스가 아직 개발되지 않은 다른 도구로 대체될지도 모릅니다.

새로운 기능 아키텍트
는 새로운 도구뿐만 아니라 새로운 패러다임에도 항상 주의를 기울여야 합니다. 새로운 기능이 등장하면 단순히 기존 도구를 다른 도구로 교체하는 것이 아니라 완전히 새로운 패러다임으로 전환해야 할 수도 있습니다. 예를 들어, Docker와 같은 컨테이너 기술의 등장으로 소프트웨어 개발 업계에 일어난 지각변동을 예상한 사람은 거의 없었습니다. 이는 진화적인 단계였지만, 아키텍트, 도구, 엔지니어링 관행 등 여러 분야에 엄청난 영향을 미쳤습니다. 생태계의 끊임없는 변화는 새로운 도구와 기능을 정기적으로 제공합니다. 기존 기술의 새로운 변형처럼 보이는 것조차도 미묘한 차이를 통해 판도를 바꿀 수 있습니다. 새로운 기능이 개발 업계 전체를 뒤흔들 필요도 없습니다. 아키텍트의 목표와 정확히 일치하는 작은 변화 하나만으로도 모든 것이 바뀔 수 있습니다.

가속
생태계는 끊임없이 변화할 뿐만 아니라, 그 변화의 속도는 점점 더 빨라지고 범위도 넓어지고 있습니다. 새로운 도구는 새로운 엔지니어링 방식을 만들어내고, 이는 다시 새로운 설계와 기능으로 이어져 소프트웨어 아키텍트들을 끊임없이 변화하는 상태에 놓이게 합니다. 생성형 AI의 등장과 영향력은 이러한 끊임없는 진화와 그에 따른 예측 불가능성을 보여주는 대표적인 사례입니다.

소프트웨어 개발 대
상 도메인은 기업의 발전이나 다른 회사와의 합병 등으로 인해 끊임없이 변화합니다.

기술 변화에 발맞춰 조직들
은 최소한 일부 기술 변화, 특히 수익에 직접적인 이점을 가져다주는 기술 변화에 보조를 맞추려고 노력합니다.

외부 요인 소프트웨
어 개발과 직접적인 관련은 없어 보이지만, 조직 내 변화를 야기할 수 있는 다양한 외부 요인이 존재합니다. 예를 들어, 아키텍트와 개발자들이 특정 도구에 만족하고 있더라도 라이선스 비용이 지나치게 높아지면 기업은 다른 옵션으로 전환해야 할 수도 있습니다.

건축가는 현재 업계 동향을 이해해야 하며, 소속 조직이 최신 건축 유행을 얼마나 따르는지와 관계없이 어떤 동향을 따르고 언제 예외를 둘지 현명한 결정을 내릴 수 있어야 합니다.

결정 기준

건축가는 건축 양식을 선택할 때 도메인 설계 구조에 영향을 미치는 다양한 요소를 모두 고려해야 합니다. 근본적으로 건축가는 두 가지를 설계합니다. 하나는 명시된 도메인이고, 다른 하나는 시스템을 성공적으로 구현하는 데 필요한 구조적 요소(건축적 특성을 통해 제공됨)입니다.

건축 양식을 선택할 때는 다음 요소들에 대해 충분한 지식을 갖춘 후에 접근하십시오.

비즈니스 도메

인의 중요한 측면, 특히 운영 아키텍처 특성에 영향을 미치는 측면을 최대한 많이 이해해야 합니다. 아키텍트는 해당 분야의 전문가일 필요는 없지만, 설계 대상 도메인의 주요 측면에 대한 전반적인 이해는 최소한 갖추어야 합니다. 비즈니스 분석가와 같은 다른 전문가들이 도메인 지식의 부족한 부분을 보완해 줄 수 있습니다.

구조적 결정에 영향을 미치는 건축적 특징

건축적 특성을 분석하여 해당 영역 및 기타 외부 요인을 지원하는 데 필요한 건축적 특성을 파악하고 명확히 설명해야 합니다. 이는 스타일 선택의 핵심 활동 중 하나입니다.

일반적인 아키텍처 스타일은 거의 모든 문제 영역에 적용할 수 있습니다. '일반적'이라는 말 자체가 범용적이라는 의미를 내포하고 있기 때문입니다.
예외는 고도의 확장성을 요구하는 경매 사이트와 같이 특별한 운영 아키텍처 특성이 필요한 도메인입니다. 그러나 대부분의 경우 아키텍처 스타일 간의 실제 차이점은 도메인이 아니라 각 스타일이 다양한 아키텍처 특성을 얼마나 잘 지원하는지에 달려 있습니다.

이 책 2부에서 각 건축 양식을 비교하기 위해 사용한 별자리 도표가 영역이 아닌 건축적 특징에 초점을 맞추고 있다는 것을 눈치채셨을 겁니다. 이는 건축 양식을 선택할 때 건축적 특징을 이해하는 것이 얼마나 중요한지를 보여줍니다.

데이터 아키텍처 설계자

와 데이터 개발자는 데이터베이스, 스키마 및 기타 데이터 관련 문제에 대해 협업해야 합니다. 데이터 아키텍처는 그 자체로 전문 분야이며, 이 책에서는 스타일 관련 고려 사항을 제외하고는 자세히 다루지 않습니다.

하지만 특정 데이터 설계가 아키텍처 설계에 미칠 수 있는 영향을 이해해야 합니다. 특히 새 시스템이 기존 데이터 아키텍처 또는 이미 사용 중인 아키텍처와 상호 작용해야 하는 경우 더욱 그렇습니다.

클라우드 배포는 컴

퓨팅과 데이터가 존재하는 위치를 근본적으로 변화시켜 온 오랜 역사의 중요한 흐름 중 하나입니다. 온프레미스 환경에서 실행되는 애플리케이션을 설계할 때와 클라우드 환경에서 실행되는 애플리케이션을 설계할 때 고려해야 할 사항은 상당히 다릅니다.

클라우드용 애플리케이션을 설계하는 데 참여하고 있습니다. 애플리케이션이 저장해야 하는 데이터 양과 데이터 이동량(이로 인해 상당한 비용이 발생할 수 있음)을 비롯한 여러 가지 사항을 파악하는 것이 중요합니다.

클라우드는 정교한 기능들이 시간이 지남에 따라 어떻게 보편화되는지를 보여주는 훌륭한 사례입니다. 10년 전만 해도 고도의 탄력성과 확장성을 갖춘 온프레미스 시스템을 구축하려면 전문적인 기술이 필요했고, 마치 마법처럼 여겨졌습니다. 하지만 이제는 아키텍트들이 클라우드 공급업체의 설정 매개변수만 변경해도 동일한 결과를 얻을 수 있습니다.

조직적 요인 다양한 외부 요
: 인이 설계에 영향을 미칩니다. 예를 들어, 특정 클라우드 공급업체의 비용 때문에 기업이 이상적인 설계를 채택하지 못할 수도 있습니다. 마찬가지로, 기업이 인수합병을 계획하고 있다는 사실을 알게 되면 아키텍트는 개방형 솔루션과 통합 아키텍처를 선호하게 될 수 있습니다.

프로세스, 팀 및 운영 관련 사항에 대한 지식은 아키텍트의 설계에 영향
: 을 미칩니다. 소프트웨어 개발 프로세스, 아키텍트와 운영 부서 간의 상호 작용(또는 부족), QA 프로세스 등 여러 가지 구체적인 프로젝트 요소가 이에 해당합니다. 예를 들어, 조직이 애자일 엔지니어링 방식에 대한 성숙도가 부족하다면, 애자일 방식을 기반으로 하는 아키텍처 스타일(예: 마이크로서비스)은 어려움을 초래할 수 있습니다.

도메인/아키텍처 동형성 아키텍처 동형성은
: 아키텍처의 일반적인 "형태"를 나타내는 전문 용어입니다. 다시 말해, 전체 토폴로지 내에서 구성 요소들이 서로 의존하는 방식을 의미합니다. 동형성이라는 단어는 "집합과 요소 간의 관계를 보존하는 사상"을 뜻하며, 그리스어 isos(동등한)와 morph(형태 또는 모양)에서 유래했습니다.

건축가는 건축물의 적합성을 고려할 때 건축물의 일반적인 형태를 생각합니다. 예를 들어, 그림 19-1에 나타난 계층형 건축 양식과 모듈형 단일체 건축 양식의 두 가지 건축 동형 다이어그램의 차이점을 생각해 보세요. 각 건축 양식의 내부 형태가 명확하게 드러납니다. 즉, 계층별 분리와 영역별 분리입니다.

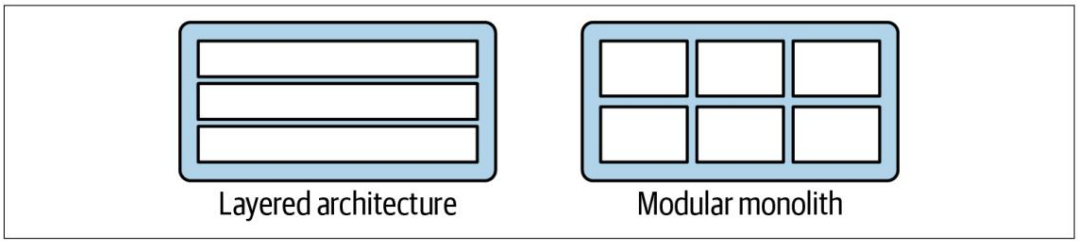

그림 19-1. 적층형 및 모듈형 모놀리스의 동형 표현 비교

단일체형과 분산형의 차이는 그림 19-2에 나타낸 동형 도면에서도 명확하게 드러납니다. 여기서는 핵심 구성 요소의 분산을 통해 아키텍처의 거시적 구조를 명확히 보여줍니다.

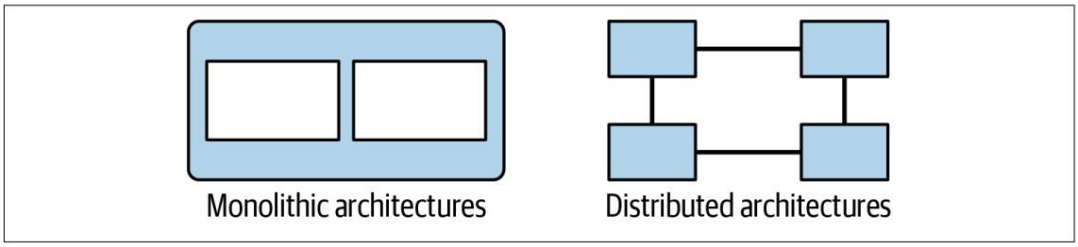

그림 19-2. 단일형 아키텍처와 분산형 아키텍처 스타일의 동형 표현 비교

일부 문제 영역은 아키텍처의 토폴로지와 잘 맞아떨어집니다. 예를 들어, 마이크로커널 아키텍처 스타일은 사용자 정의 기능이 필요한 시스템에 매우 적합합니다. 설계자는 플러그인 형태로 사용자 정의 기능을 설계할 수 있기 때문입니다. 또 다른 예로, 수많은 개별 연산이 필요한 유전체 분석 시스템은 다수의 개별 프로세서를 제공하는 우주 기반 아키텍처에 적합할 수 있습니다.

마찬가지로, 일부 문제 영역은 특정 아키텍처 스타일에 특히 적합하지 않을 수 있습니다. 확장성이 뛰어난 시스템은 대규모 모놀리식 설계에서 어려움을 겪는데, 이는 고도로 결합된 코드베이스가 많은 수의 동시 사용자를 지원하기 어렵기 때문입니다. 의미론적 결합도가 매우 높은 문제 영역은 고도로 분리된 분산 아키텍처와 잘 맞지 않습니다. 예를 들어, 각 페이지가 이전 페이지의 컨텍스트에 기반하는 여러 페이지로 구성된 보험 애플리케이션은 고도로 결합된 문제입니다. 이러한 문제는 마이크로서비스와 같은 분리된 아키텍처로 모델링하기 어렵습니다. 서비스 기반 아키텍처처럼 의도적으로 결합된 아키텍처가 이러한 문제에 더 적합합니다.

이 모든 것을 고려하여 건축 양식을 선택할 때 몇 가지 결정을 내려야 합니다.

단일형 방식 vs. 분산형 방식?

단일 아키텍처 특성 세트로 설계가 충분할까요, 아니면 시스템의 각 부분에 서로 다른 아키텍처 특성이 필요할까요? 단일 특성 세트라면 모놀리식 아키텍처가 적합할 수 있습니다(물론 다른 요소들을 고려했을 때 분산 아키텍처가 더 적합할 수도 있습니다). 각 부분에 서로 다른 아키텍처 특성 세트가 필요하다면 분산 아키텍처가 필요할 것입니다. 이러한 결정을 내리는 데에는 아키텍처 양자 (7장 참조) 개념이 유용합니다.

데이터는 어디에 저장되어야 할까요?

단일 구조 아키텍처의 경우, 설계자는 일반적으로 단일 관계형 데이터베이스 또는 몇 개의 관계형 데이터베이스를 사용할 것이라고 가정합니다. 분산 아키텍처에서는,

데이터를 영구 저장할 서비스를 결정해야 하는데, 이는 데이터가 아키텍처를 통해 어떻게 흐를지, 그리고 워크플로우를 어떻게 구축할지까지 고려해야 함을 의미합니다. 아키텍처를 설계할 때는 구조와 동작 모두를 고려해야 하며, 더 나은 조합을 찾기 위해 설계를 반복적으로 수정하는 것을 두려워하지 마세요.

서비스들은 동기적으로 통신해야 할까요, 아니면 비동기적으로 통신해야 할까요?

데이터가 저장될 위치를 결정했다면, 다음으로 고려해야 할 사항은 서비스 간 통신 방식입니다. 동기식 통신을 사용할지 비동기식 통신을 사용할지 결정해야 합니다.

동기 통신은 종종 더 편리하지만, 확장성, 신뢰성 및 기타 바람직한 특성을 희생해야 할 수도 있습니다. 비동기 통신은 성능과 확장성 측면에서 고유한 이점을 제공할 수 있지만, 데이터 동기화, 교착 상태, 경쟁 조건, 디버깅 등과 관련된 많은 문제점도 야기할 수 있습니다. (이러한 문제들은 15장 에서 자세히 다룹니다.)

동기식 통신은 설계, 구현 및 디버깅 측면에서 어려움이 적기 때문에 가능한 한 동기식 통신을 기본으로 사용하고, 필요한 경우에만 비동기식 통신을 사용하는 것이 좋습니다.

기본적으로 동기 통신을 사용하고, 필요에 따라 비동기 통신을 사용합니다.
필요한.

이 설계 프로세스의 결과물은 선택된 아키텍처 스타일(및 모든 혼합 스타일)을 포함하는 아키텍처 토폴로지, 가장 많은 노력이 필요한 설계 부분에 대한 아키텍처 결정 기록(ADR), 그리고 중요한 원칙과 운영 아키텍처 특성을 보호하기 위한 아키텍처 적합성 함수입니다.

모놀리스 사례 연구: 실리콘 샌드위치

5 장의 실리콘 샌드위치 아키텍처 카타에서 아키텍처 특성 분석을 통해 단일 퀀텀 칩이면 시스템 구현에 충분하다는 결론을 내렸습니다. 예산이 넉넉하지 않은 간단한 애플리케이션을 개발하는 상황이었기 때문에 모놀리식 아키텍처의 단순함이 매력적이었습니다.

하지만 저희는 실리콘 샌드위치를 위해 두 가지 다른 구성 요소 설계 방식을 만들었습니다. 하나는 도메인 분할 방식이고, 다른 하나는 기술적 분할 방식입니다. 복습이 필요하시면 5장 을 다시 참조하시기 바랍니다 . 이 장에서는 이러한 간단한 솔루션을 다시 살펴보고 각 옵션에 대한 설계를 만들고 장단점을 논의합니다. 먼저 모놀리식 아키텍처부터 시작하겠습니다.

모듈형 모놀리스

모듈형 모놀리스는 단일 데이터베이스를 사용하여 도메인 중심 구성 요소를 구축하고 단일 양자로 배포합니다. 실리콘 샌드위치용 모듈형 모놀리스 설계는 그림 19-3에 나와 있습니다.

이 시스템은 단일 관계형 데이터베이스를 사용하는 모놀리식 아키텍처이며, 전체 비용을 절감하기 위해 단일 웹 기반 UI(모바일 기기에 대한 세심한 설계 고려)로 구현되었습니다. 식별된 각 도메인은 구성 요소로 나타납니다. 시간과 자원이 충분하다면 테이블 및 기타 데이터베이스 자산을 도메인 구성 요소와 동일한 방식으로 분리하는 것을 고려해 보세요. 이렇게 하면 향후 요구 사항에 따라 분산 아키텍처로 마이그레이션하는 것이 훨씬 쉬워집니다.

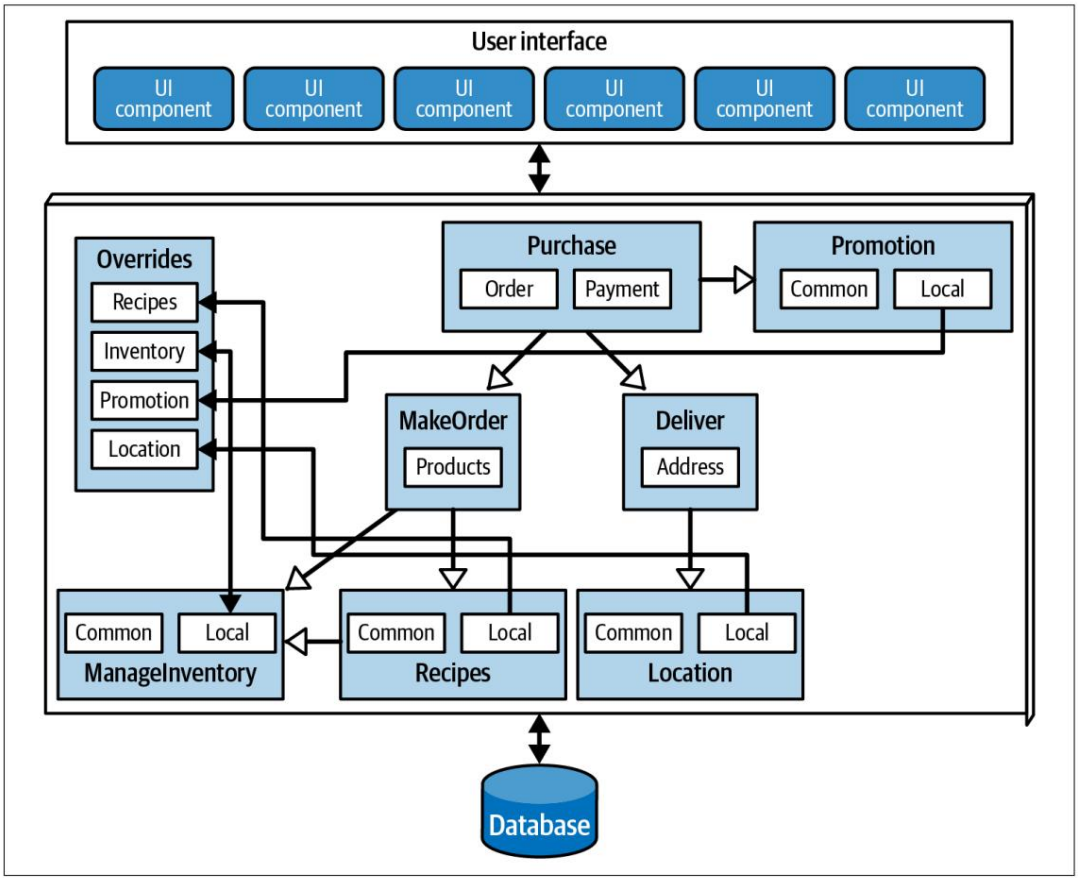

그림 19-3. 실리콘 샌드위치의 모듈형 모놀리스 구현

아키텍처 스타일 자체가 본질적으로 사용자 정의를 지원하지 않기 때문에 해당 기능은 도메인 설계에 포함되어야 합니다. 이 경우 아키텍트는 개발자가 개별 사용자 정의를 업로드할 수 있는 오버라이드 엔드포인트를 설계합니다. 따라서 모든 도메인 구성 요소가 이 오버라이드를 참조하도록 해야 합니다.

각 사용자 정의 가능한 특성에 대한 구성 요소입니다. (이 부분을 확인하는 것은 아키텍처 적합성 함수에 딱 맞는 작업일 것입니다.)

마이크로커널

실리콘 샌드위치에서 확인한 아키텍처 특징 중 하나는 사용자 정의 가능성입니다. 그림 19-4 는 도메인/아키텍처 동형성을 사용하여 마이크로커널 아키텍처를 통해 이를 구현하는 방법을 보여줍니다.

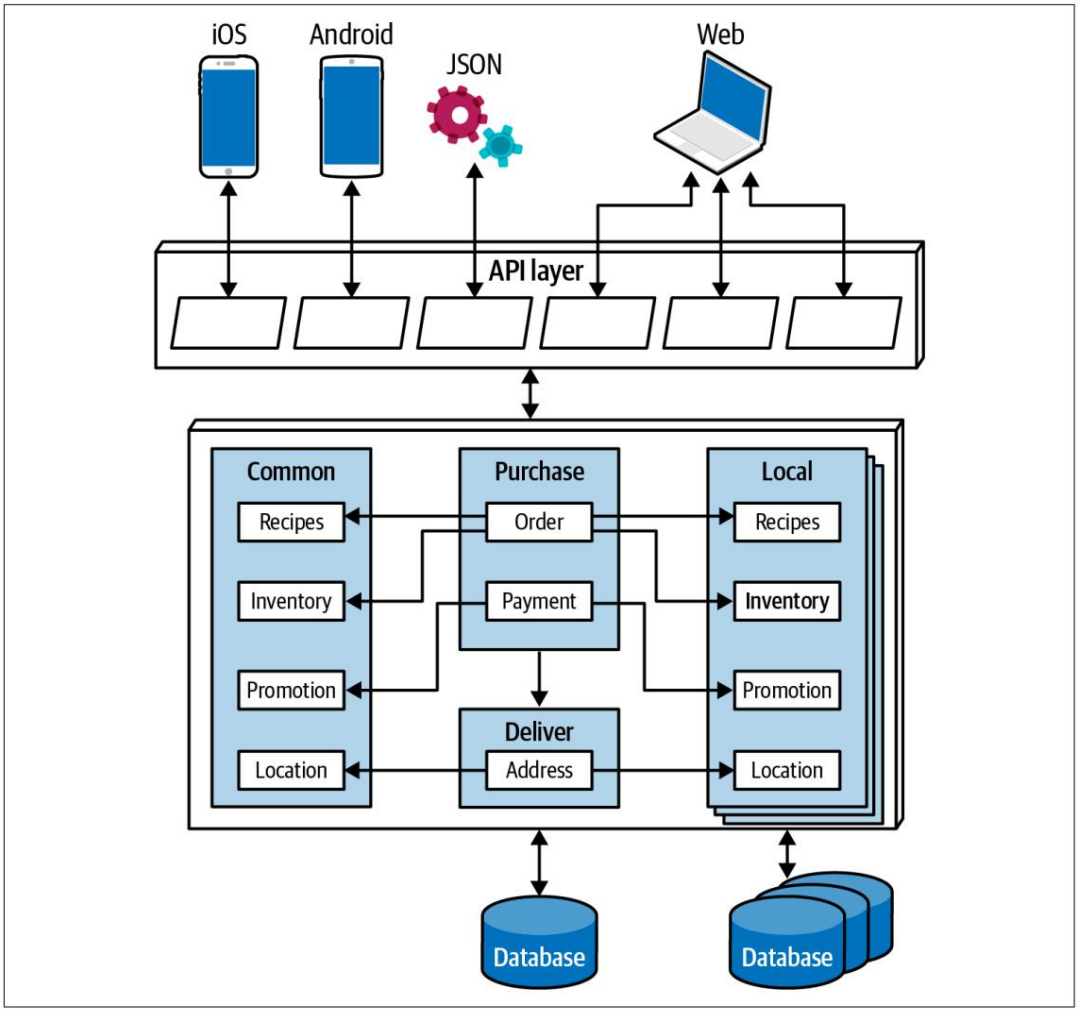

그림 19-4. 실리콘 샌드위치의 마이크로커널 구현

그림 19-4 에서 핵심 시스템은 도메인 구성 요소와 단일 관계형 데이터베이스로 구성됩니다. 모듈형 모놀리식 설계와 마찬가지로 도메인과 데이터 설계 간의 세심한 동기화는 향후 핵심 시스템을 분산 아키텍처로 마이그레이션하는 데 도움이 될 것입니다. 각 사용자 정의 기능은 플러그인으로 제공되며, 공통 기능은 단일 플러그인 세트(해당 데이터베이스 포함)와 일련의 로컬 플러그인으로 구성됩니다.

각 플러그인은 고유한 데이터를 가지고 있습니다. 플러그인들은 서로 연결될 필요가 없기 때문에 각각 데이터를 유지하고 독립적으로 작동할 수 있습니다.

이 설계의 또 다른 독특한 요소는 백엔드를 프런트엔드로 사용하는(BFF) 패턴을 활용한다는 점입니다. 이를 통해 API 계층은 핵심 아키텍처 외에도 얇은 마이크로커널 어댑터 역할을 합니다. API 계층은 백엔드에서 일반 정보를 제공하고, BFF 어댑터는 이 정보를 프런트엔드 장치에 적합한 형식으로 변환합니다. 예를 들어, iOS용 BFF는 일반적인 백엔드 출력을 받아 데이터 형식, 페이지네이션, 지연 시간 및 기타 요소를 iOS 네이티브 애플리케이션이 기대하는 형식에 맞게 조정합니다. 각 BFF 어댑터를 구축함으로써 최대한 풍부한 사용자 인터페이스를 구현할 수 있으며, 향후 다른 장치를 지원하도록 아키텍처를 확장할 수 있습니다. 이는 마이크로커널 방식의 장점 중 하나입니다.

이 두 가지 실리콘 샌드위치 아키텍처 모두에서 통신은 동기식으로 이루어질 수 있습니다. 왜냐하면 이 아키텍처는 극한의 성능이나 탄력성을 요구하지 않으며, 모든 작업에 오랜 시간이 걸리지 않기 때문입니다.

분산형 사례 연구: 가고, 가고, 사라지다

8장의 Going, Going, Gone(GGG) 카타는 몇 가지 흥미로운 아키텍처 과제를 제시합니다. 125페이지의 "사례 연구: Going, Going, Gone - 구성 요소 발견"에서 살펴본 구성 요소 분석을 통해, 이 아키텍처의 각 부분은 서로 다른 특성을 필요로 한다는 것을 알 수 있습니다. 예를 들어, 경매 진행자와 입찰자 같은 역할에 따라 가용성과 확장성에 대한 요구 사항이 달라집니다.

GGG 시스템 요구사항에는 확장성, 탄력성, 성능 및 기타 까다로운 운영 아키텍처 특성에 대한 야심찬 기대치가 명시적으로 기재되어 있습니다. 아키텍처 패턴은 세부적인 수준에서 높은 수준의 맞춤 설정이 가능해야 합니다. 후보 분산 아키텍처 중 저수준 이벤트 기반 아키텍처와 마이크로서비스가 요구되는 아키텍처 특성의 대부분을 가장 잘 충족합니다. 이 두 가지 중 마이크로서비스는 운영 아키텍처 특성 간의 다양성을 지원하는 데 더 적합합니다. (순수 이벤트 기반 아키텍처는 일반적으로 아키텍처 특성이 아니라 오케스트레이션된 통신을 사용하는지 또는 안무된 통신을 사용하는지에 따라 구성 요소를 구분합니다.)

명시된 성능 목표를 마이크로서비스 환경에서 달성하는 것은 어려운 과제일 수 있지만, 아키텍처의 약점을 해결하는 가장 좋은 방법은 이를 고려하여 설계하는 것입니다. 예를 들어, 마이크로서비스는 본질적으로 높은 확장성을 제공하지만, 과도한 오케스트레이션이나 지나친 데이터 분리 등으로 인해 특정 성능 문제가 발생하는 경우가 많습니다.

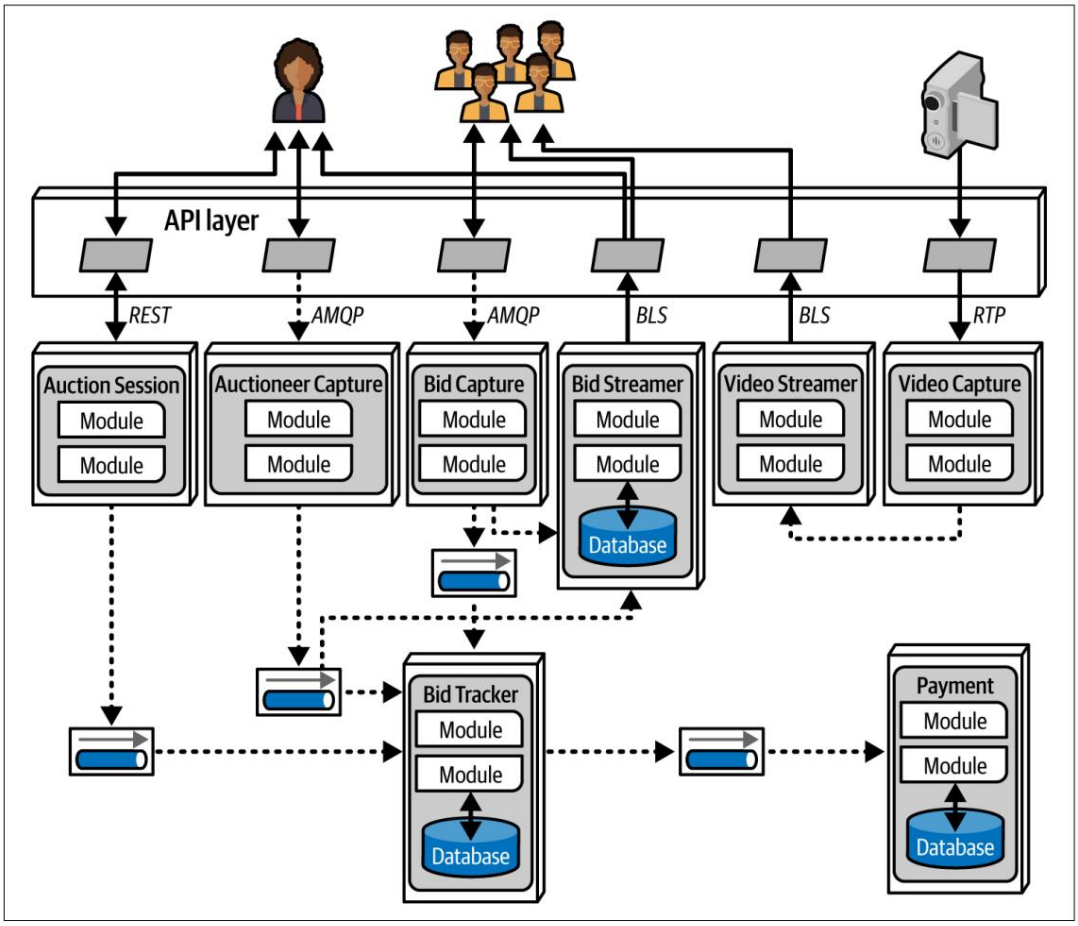

그림 19-5. Going, Going, Gone의 마이크로서비스 구현

그림 19-5 에서 식별된 각 구성 요소는 아키텍처에서 서비스가 되며, 구성 요소와 서비스 세분성이 일치합니다. GGG는 세 가지의 서로 다른 사용자 인터페이스를 가지고 있습니다.

입찰자: 온

라인 경매에 참여한 수많은 입찰자들.

경매 진행자는

경매당 한 명입니다.

스트리머는 비

디오 스트리밍 및 입찰 정보를 입찰자에게 제공하는 서비스입니다. 이는 읽기 전용 스트림이므로 업데이트가 필요한 경우 불가능한 최적화가 가능합니다.

GGG 아키텍처 설계에는 다음과 같은 서비스가 포함됩니다.

입찰 접수 기능은

온라인 입찰자의 응모를 접수하여 비동기적으로 입찰 추적 시스템 으로 전송합니다 .
이 서비스는 온라인 입찰을 위한 통로 역할을 하므로 지속적인 관리가 필요하지 않습니다.

비드 스트리머

입찰 정보를 고성능 읽기 전용 방식으로 온라인 참가자들에게 스트리밍합니다.
개울.

입찰 추적기

경매 진행자 입력(Auctioneer Capture) 과 입찰 입력(Bid Capture) 모두에서 입찰 정보를 추적하여 두 가지 정보 흐름을 통합하고 가능한 한 실시간에 가깝게 입찰 순서를 정렬합니다.
이 서비스에 대한 두 개의 인바운드 연결은 모두 비동기식이므로 개발자는 메시지 큐를 버퍼로 사용하여 매우 다양한 메시지 흐름을 처리할 수 있습니다.
요금.

경매인 캡처는 경매인을 대신하

여 입찰을 캡처합니다. 125페이지의 "사례 연구: Going, Going, Gone—구성 요소 발견" 분석에서 입찰 캡처 와 경매인 캡처 의 아키텍처 특성이 상당히 다르다는 것을 알게 되어 이들을 분리하기로 했습니다.

경매 세션은 개별 경매의 워

크플로를 관리합니다.

결제는 경매

세션이 완료된 후 결제 정보를 처리하는 제3자 결제 제공업체를 통해 이루어집니다 .

비디오 캡처 기능 은 실

시간 경매의 비디오 스트림을 캡처합니다.

비디오 스트리머는 경매

영상을 온라인 입찰자들에게 스트리밍합니다.

본 아키텍처에서는 동기식 및 비동기식 통신 방식을 모두 신중하게 고려했습니다. 비동기식 통신을 선택한 주된 이유는 서비스별로 운영 아키텍처 특성이 다양하기 때문입니다.

예를 들어, 결제 서비스가 500ms마다 한 번씩만 새로운 결제를 처리할 수 있고, 많은 경매가 동시에 종료되는 경우, 서비스 간 동기식 통신을 사용하면 타임아웃 및 기타 안정성 문제가 발생할 수 있습니다.

메시지 큐는 아키텍처에서 중요하지만 취약한 부분에 안정성을 더해줍니다.

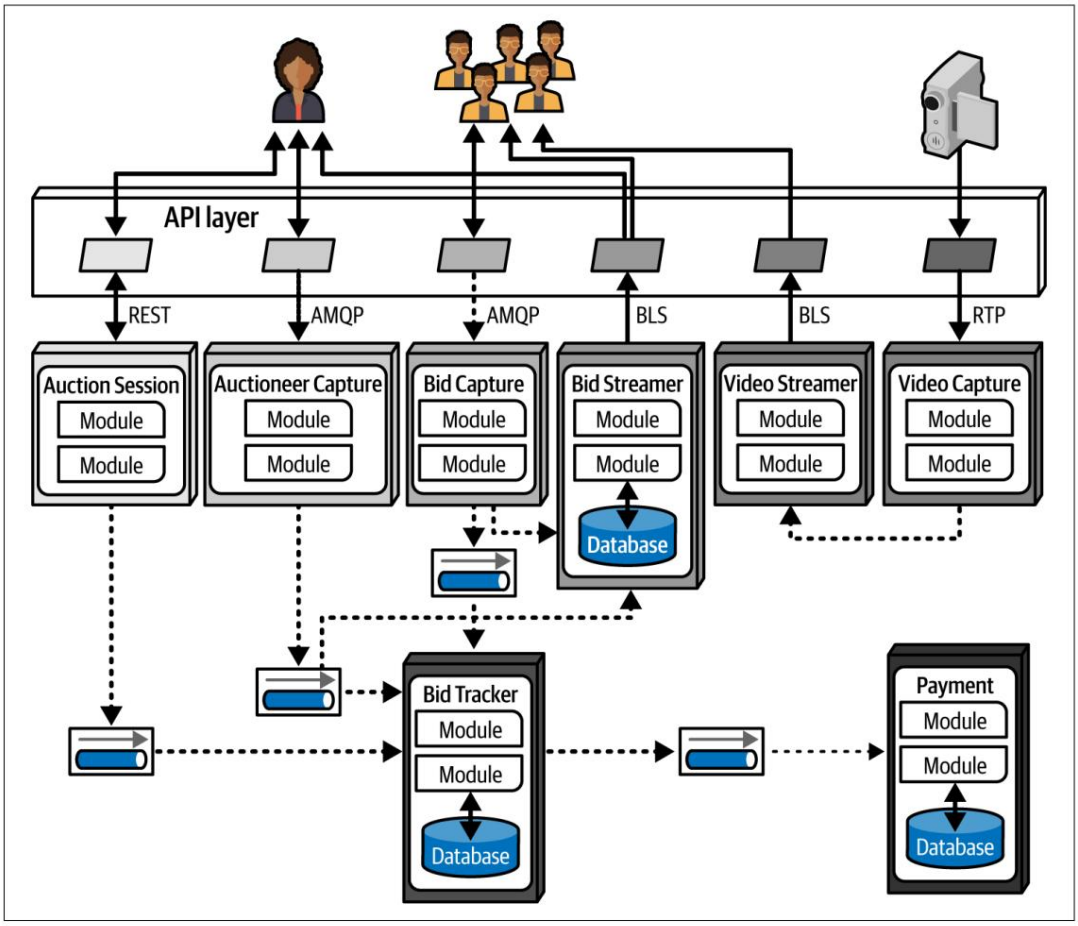

그림 19-6. GGG에 대한 양자 경계

최종적으로 이 설계는 그림 19-6 에서 볼 수 있듯이 결제, 경매 진행자, 입찰자, 입찰자 스트림, 입찰 추적기 등 다섯 가지 양자로 요약됩니다. 다이어그램에서는 컨테이너 스택을 통해 여러 인스턴스를 나타냅니다. 구성 요소 설계 단계에서 양자 분석을 활용함으로써 서비스, 데이터 및 통신 경계를 더 쉽게 파악할 수 있었습니다.

참고로, 이것이 GGG에 맞는 "올바른" 디자인은 아니며, 유일한 디자인도 아닙니다.
이것이 최선의 설계라고 주장하는 것은 아니지만, 여러 가지 장단점을 고려했을 때 가장 나은 선택으로 보입니다. 마이크로서비스를 선택하고 이벤트와 메시지를 지능적으로 활용함으로써, 이 아키텍처는 일반적인 아키텍처 패턴의 장점을 최대한 활용하는 동시에 향후 개발 및 확장을 위한 기반을 구축할 수 있습니다.