

기초

건축 양식과 패턴에 대한 자세한 내용을 알고 싶어하시는 것을 이해하지만, 먼저 이후 장에서 다룰 내용에 대한 적절한 맥락을 설정하기 위해 몇 가지 기본적이고 정의적인 내용을 다뤄야 합니다.

스타일과 패턴

먼저 건축 양식과 건축 패턴을 구분해야 하는데, 이 둘은 쉽게 혼동될 수 있습니다.

건축 양식은 다음과 같은 여러 가지 특징을 포함하여 해당 건축물의 다양한 특성을 설명합니다.

컴포넌트 토폴로지 아

키텍처 스타일은 컴포넌트와 그 종속성이 어떻게 구성되는지를 정의합니다. 예를 들어, 계층형 아키텍처는 컴포넌트 계층을 기술적 기능에 따라 구성하는 반면, 모듈형 모놀리식은 도메인을 중심으로 컴포넌트를 구성합니다(이러한 차이점에 대한 자세한 내용은 [137페이지의 "아키텍처 파티셔닝"](#)을 참조하십시오).

물리적 아키텍처는 종종 디자인에 따라 결정됩니다.

모놀리식 아키텍처가 있는가 하면 분산형 아키텍처도 있습니다. 예를 들어 모듈형 모놀리식 아키텍처는 일반적으로 단일 데이터베이스를 사용하는 모놀리식 아키텍처인 반면, 이벤트 기반 아키텍처는 항상 분산형 아키텍처입니다.

배포 시스템

의 세분성 및 배포 빈도는 종종 아키텍처 스타일과 관련이 있습니다. 일반적으로 팀은 단일 관계형 데이터베이스와 함께 단일 배포 방식으로 모놀리식 아키텍처를 배포합니다. 반대로, 고도로 애자일한 아키텍처는

마이크로서비스와 같은 분산 아키텍처는 자동화된 통합, 자동화된 프로비저닝, 그리고 경우에 따라 자동화된 배포 기능을 제공합니다. 이러한 아키텍처는 일반적으로 훨씬 빠른 주기로 부분적으로 배포됩니다.

통신 방식 아키텍처 스타일은

구성 요소 간의 통신 방식도 결정합니다. 모놀리식 아키텍처는 모놀리식 내부에서 메서드 호출을 할 수 있는 반면, 분산 아키텍처는 REST와 같은 네트워크 프로토콜이나 메시지 큐를 통해 통신합니다.

데이터 토폴로지

는 구성 요소 토폴로지와 마찬가지로 시스템의 아키텍처 스타일에 따라 결정되는 경우가 많습니다. 모놀리식 아키텍처는 일반적으로 단일 데이터베이스를 사용하는 반면, 분산 아키텍처는 아키텍처 스타일의 철학에 따라 데이터를 분리하는 경우가 있습니다.

스타일 이름을 짓는 것은 이러한 복잡한 요소들을 간결하게 설명하는 방법입니다. 각 이름은 잘 알려진 세부 사항들을 담고 있는데, 이것이 바로 디자인 패턴의 목적 중 하나입니다. 그러나 패턴이 맥락에 맞는 해결책을 제시하는 반면, 스타일은 아키텍처에 더 특화되어 위에서 언급한 측면들을 설명합니다. 아키텍처 스타일은 아키텍처의 토폴로지와 가정된 기본 특성(유익한 특성과 해로운 특성 모두 포함)을 설명합니다. **20장**에서 몇 가지 일반적인 현대 아키텍처 패턴을 다룹니다. 아키텍트는 시스템의 공통적인 구성 요소가 되는 몇 가지 기본 스타일을 숙지해야 합니다.

건축 양식은 어디에서 유래했을까요?

일반적인 생각과는 달리, 상아탑에 모여 다음 건축 양식이 무엇인지 결정하는 공식적인 건축 위원회 같은 것은 없습니다. 오히려 새로운 양식은 건축가들이 활동하는 끊임없이 진화하는 생태계 속에서 자연스럽게 나타납니다.

예를 들어, 한 뛰어난 아키텍트가 생태계에 새로 등장한 기능이 오랫동안 골칫거리였던 문제를 해결해 준다는 사실을 발견했다고 가정해 봅시다. 그는 그 기능을 다른 여러 가지 새로운 기능과 기존 기능을 결합하여 활용하기로 결정합니다. 다른 아키텍트들이 이 기발한 해결책을 보고 그대로 따라합니다.

워낙 흔한 현상이 되어서 이름을 붙이면 논의하기가 훨씬 쉬워진다.

마이크로서비스 아키텍처 스타일은 이러한 현상을 잘 보여주는 예입니다. 새로운 DevOps 기능, 신뢰할 수 있는 오픈 소스 운영 체제, 그리고 도메인 중심 설계 철학의 등장으로 아키텍트들은 확장성과 같은 문제를 해결하기 위해 새로운 방식으로 시스템을 구축할 수 있게 되었습니다. 마이크로서비스라는 이름은 당시 일반적이었던 대규모 서비스와 복잡한 오케스트레이션을 특징으로 하는 아키텍처 스타일에 대한 반작용으로 생겨났습니다. 마이크로서비스는 명칭일 뿐, 설명이 아닙니다. 팀에게 가능한 한 가장 작은 서비스를 구축하라는 명령이 아니라, 이러한 아키텍처 스타일을 지칭하는 방식입니다.

기본 패턴

소프트웨어 아키텍처의 역사 전반에 걸쳐 몇 가지 기본적인 패턴이 반복적으로 나타나는데, 이는 일반적으로 코드 구성, 배포 또는 아키텍처의 다른 측면에 대한 유용한 관점을 제공하기 때문입니다. 예를 들어, 기능에 따라 서로 다른 관심사를 분리하는 계층 구조 개념은 소프트웨어 자체만큼이나 오래되었습니다.

하지만 레이어드 스타일(스타일과 패턴 모두)은 다양한 형태로 계속해서 나타나고 있으며, **10장에서 논의할 현대적인 변형도 그중 하나입니다.**

하지만 안타깝게도 아키텍처가 부재할 때 발생하는 또 다른 흔한 안티패턴이 있는데, 바로 '빅 볼 오브 머드(Big Ball of Mud)'입니다.

커다란 진흙 덩어리

건축가들은 뚜렷한 건축 구조가 없는 상태를 '커다란 진흙 덩어리(Big Ball of Mud)'라고 부르는데, 이는 브라이언 푸트와 조셉 요더가 1997년 '프로그램의 패턴 언어(Patterns Languages of Programs)' 컨퍼런스에서 발표한 논문에서 정의한 안티패턴의 이름에서 유래했습니다.

'빅 볼 오브 머드(Big Ball of Mud)'는 구조가 제멋대로이고, 무질서하게 뿔어나가며, 덕트 테이프와 철사로 덕지덕지 붙여 만든 스파게티 코드 정글과 같습니다. 이러한 시스템은 무분별한 성장과 반복적이고 임시방편적인 보수의 흔적을 분명히 보여줍니다. 정보는 시스템의 멀리 떨어진 요소들 사이에서 무분별하게 공유되며, 종종 거의 모든 중요한 정보가 전역적으로 사용되거나 중복되는 지경에 이릅니다.

시스템의 전체적인 구조는 애초에 제대로 정의된 적이 없을지도 모릅니다.

만약 그랬다면, 그 시스템은 알아볼 수 없을 정도로 훼손되었을지도 모릅니다. 건축적 감각이 조금이라도 있는 프로그래머라면 이런 수렁을 피하겠죠. 아키텍처에 관심이 없고, 어쩌면 무너져가는 독의 구멍을 메우는 일상적인 작업에 만족하는 사람들만이 이런 시스템에서 일하는 데 만족할 겁니다.

오늘날 '빅 볼 오브 머드(Big Ball of Mud)'라는 표현은 내부 구조가 제대로 갖춰지지 않고 이벤트 핸들러가 데이터베이스 호출에 직접 연결된 간단한 스크립팅 애플리케이션을 가리키는 말일 수 있습니다. 많은 간단한 애플리케이션들이 이런 식으로 시작하지만, 규모가 커지면서 다루기 어려워집니다.

일반적으로 이러한 유형의 아키텍처는 어떤 경우에도 피해야 합니다. 구조가 부족하여 변경이 점점 더 어려워지기 때문입니다. 또한 이러한 아키텍처는 배포, 테스트 용이성, 확장성 및 성능 측면에서도 문제가 발생합니다. 시스템 규모가 커질수록 아키텍처 부재로 인한 문제점은 더욱 심각해집니다.

불행히도 이러한 안티패턴은 현실에서 상당히 자주 발생합니다. 의도적으로 혼란을 야기하려는 아키텍트는 거의 없지만, 코드 품질 및 구조에 대한 관리가 부족하여 의도치 않게 혼란스러운 프로젝트가 많이 발생합니다. 예를 들어, Neal은 **그림 9-1과 같은 구조를 가진 한 클라이언트 프로젝트를 진행했습니다.**

(명백한 이유로 이름을 밝히지 않는) 의뢰인은 수년에 걸쳐 가능한 한 빨리 자바 기반 웹 애플리케이션을 개발했습니다. **그림 9-1**의 기술 시각화는 해당 애플리케이션의 결합도를 보여줍니다. 원의 둘레에 있는 각 점은 각각의 구성 요소를 나타냅니다.

클래스들이 나열되어 있고, 각 선은 클래스들 간의 연결을 나타내며, 굵은 선일수록 연결이 강함을 의미합니다. 이 코드베이스에서는 클래스를 어떤 식으로든 변경하면 다른 클래스에 미치는 영향을 예측하기 어렵기 때문에 변경 작업이 매우 부담스럽습니다.

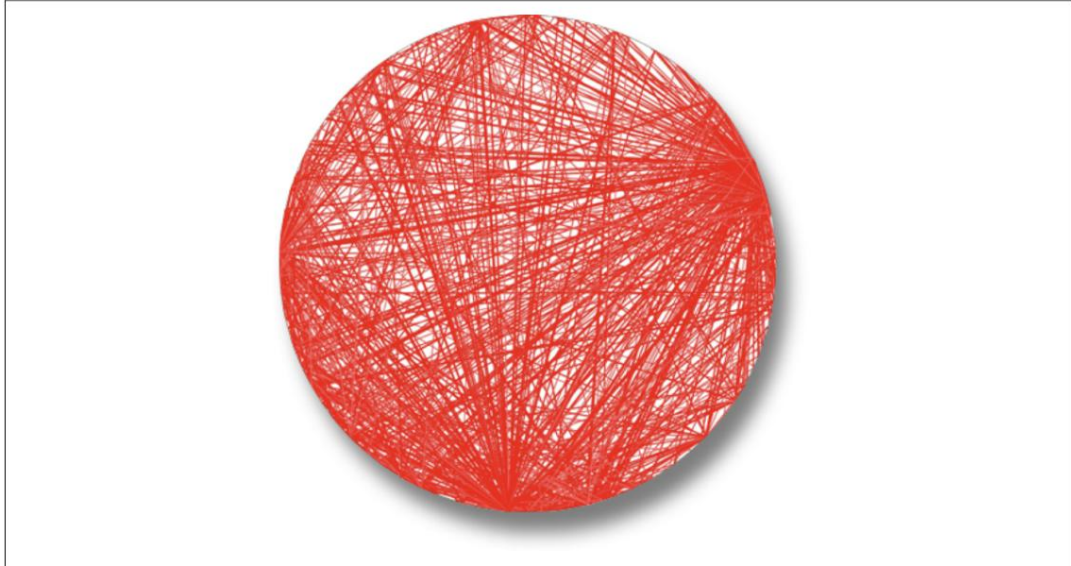


그림 9-1. 실제 코드베이스를 기반으로 시각화한 빅볼 오브 머드 아키텍처. (현재는 사용이 중단된 이클립스 플러그인인 XRay라는 도구를 사용하여 제작되었습니다.)

거대한 진흙 덩어리 형태의 건축물이 가진 문제는 구조가 부족하다는 것만이 아닙니다. 모든 것이 서로 연결되어 있기 때문에 변경 사항은 예측하기 어려운 연쇄적인 부작용을 일으키는 경향이 있습니다. 이 문제는 개발자들이 새로운 기능 개발보다는 버그와 그 부작용을 해결하는 데 모든 시간을 쏟는 심각한 상황에 이를 수 있습니다.

단일 아키텍처 초기에는 컴퓨터

와 그 위에서 실행되는 소프트웨어만 존재했습니다. 둘은 처음에는 하나의 개체로 시작했지만, 발전하고 더욱 정교한 기능에 대한 필요성이 커짐에 따라 분리되었습니다. 예를 들어, 메인프레임 컴퓨터는 단일 시스템으로 시작했지만, 점차 데이터를 별도의 시스템으로 분리했습니다. 마찬가지로, 개인용 컴퓨터(PC)가 처음 상용화되었을 때는 주로 단일 기기에 초점을 맞췄습니다. 네트워크로 연결된 PC가 보편화되면서 클라이언트/서버와 같은 분산 시스템이 등장했습니다.

클라이언트/서버

임베디드 시스템이나 기타 제약이 심한 환경을 제외하면 단일 아키텍처는 거의 존재하지 않습니다. 일반적으로 소프트웨어 시스템은 시간이 지남에 따라 기능을 추가하는 경향이 있으며, 성능 및 확장성과 같은 운영 아키텍처 특성을 유지하기 위해서는 관심을 분리하는 것이 필요해집니다.

시스템의 각 부분을 효율적으로 분리하는 방법에 대한 다양한 아키텍처 스타일이 있습니다. 아키텍처의 기본 스타일 중 하나는 프론트엔드와 백엔드 간의 기술적 기능을 분리하는 것인데, 이를 2계층 또는 클라이언트/서버 아키텍처라고 합니다. 이러한 아키텍처는 시대와 시스템의 컴퓨팅 성능에 따라 다양한 형태로 존재합니다.

데스크톱과 데이터베이스 서버 초기 PC

아키텍처는 개발자들이 윈도우와 같은 사용자 인터페이스에서 풍부한 기능을 갖춘 데스크톱 애플리케이션을 개발하고 데이터를 별도의 데이터베이스 서버에 저장하는 방식을 장려했습니다. 이러한 아키텍처는 표준 네트워크 프로토콜을 통해 연결할 수 있는 독립형 데이터베이스 서버의 등장과 시기를 같이했습니다. 덕분에 프레젠테이션 로직은 데스크톱에 두고, 연산량이 많고 복잡한 작업은 더욱 강력한 데이터베이스 서버에서 처리할 수 있었습니다.

브라우저 및 웹 서버

현대 웹 개발이 등장하면서 웹 브라우저와 웹 서버(그리고 웹 서버는 다시 데이터베이스 서버에 연결됨)로 구성된 아키텍처가 일반화되었습니다. 이러한 책임 분리는 데스크톱 환경과 유사했지만, 브라우저와 같은 클라이언트가 훨씬 더 간소화되어 방화벽 내부와 외부 모두에서 더 폭넓게 배포될 수 있었습니다. 데이터베이스가 웹 서버와 분리되어 있음에도 불구하고, 많은 아키텍트들은 여전히 이를 2계층 아키텍처로 간주합니다. 웹 서버와 데이터베이스 서버는 운영 센터 내의 동일한 유형의 시스템에서 실행되고, 사용자 인터페이스(UI)는 사용자의 브라우저에서 실행되기 때문입니다.

웹 반응성이 향상됨에 따라 브라우저의

JavaScript 구현도 발전하면서 단일 페이지 JavaScript 애플리케이션이 등장했습니다. 이는 기존 데스크톱 버전과 유사하지만, 데스크톱 애플리케이션이 아닌 브라우저에서 JavaScript로 작성된 풍부한 기능을 갖춘 클라이언트를 사용하는 클라이언트/서버 방식의 애플리케이션 제품군입니다.

이 섹션에서 설명하는 것처럼, 애플리케이션의 요구 사항과 플랫폼의 기능에 따라 아키텍처의 여러 부분을 분리하는 계층 구조는 항상 존재합니다.

3단

더욱 세분화된 계층 분리를 제공하는 3계층 아키텍처는 1990년대 후반에 인기를 얻었습니다. 자바와 .NET에서 애플리케이션 서버와 같은 도구가 널리 사용되면서 기업들은 시스템 토폴로지에 더 많은 계층을 구축하기 시작했습니다. 예를 들어, 한 시스템은 강력한 데이터베이스 서버를 사용하는 데이터베이스 계층, 애플리케이션 서버에서 관리하는 애플리케이션 계층, 그리고 생성된 HTML과 (기능이 확장됨에 따라) 점점 더 많이 사용되는 자바스크립트로 코딩된 프론트엔드를 가질 수 있습니다.

3계층 아키텍처는 분산 아키텍처 구축을 용이하게 하기 위해 **CORBA(Common Object Request Broker Architecture)** 및 **DCOM(Distributed Component Object Model)** 과 같은 네트워크 수준 프로토콜과 연동됩니다.

오늘날 개발자들이 TCP/IP와 같은 네트워크 프로토콜의 작동 방식에 대해 걱정하지 않는 것처럼(그냥 작동하기 때문에), 대부분의 아키텍처도 분산 아키텍처에서 이러한 수준의 기본 구조에 대해 걱정할 필요가 없습니다. 그 시대의 도구들이 제공했던 기능들은 오늘날에도 도구(예: 메시지 큐) 또는 아키텍처 패턴(예: 15 장에서 다루는 이벤트 기반 아키텍처)의 형태로 존재합니다.

1990년대 자바 언어가 설계될 당시, 3계층 컴퓨팅이 큰 인기를 끌었습니다. 사람들은 미래에는 모

든 시스템이 3계층 아키텍처를 사용할 것이라고 예상했습니다. 당시 C++와 같은 기존 언어의 공통적인 문제점 중 하나는 시스템 간에 일관된 방식으로 네트워크를 통해 객체를 전송하는 것이 매우 번거롭다는 것이었습니다. 그래서 자바 설계자들은 직렬화라는 메커니즘을 사용하여 이 기능을 핵심 기능에 내장하기로 결정했습니다.

모든 Java 객체는 직렬화를 지원해야 하는 인터페이스를 구현합니다. 설계자들은 3계층 아키텍처가 영원히 사용될 것이라고 생각했기 때문에 이를 언어 자체에 포함시키면 큰 편의성을 제공할 것이라고 판단했습니다. 물론, 그러한 아키텍처 스타일은 사라졌지만, 그 흔적은 오늘날까지 Java에 남아 있습니다. 사실상 직렬화를 사용하는 사람은 거의 없지만, 새로운 Java 기능은 하위 호환성을 위해 직렬화를 지원해야 하므로 언어 설계자들에게 큰 골칫거리입니다.

소프트웨어 개발을 비롯한 다른 엔지니어링 분야에서 설계 결정의 장기적인 영향을 이해하는 것은 항상 어려운 과제였습니다. 단순한 설계를 선호하라는 끊임없는 조언은 여러모로 미래를 대비하는 전략이라고 할 수 있습니다.

아키텍처 분할

소프트웨어 아키텍처의 첫 번째 법칙은 소프트웨어의 모든 것은 절충의 산물이며, 이는 아키텍트가 아키텍처에서 구성 요소를 분할하는 방식에도 적용된다는 것입니다.

컴포넌트는 일반적인 포함 메커니즘을 나타내므로 아키텍트는 원하는 방식으로 컴포넌트를 분할할 수 있습니다. 몇 가지 일반적인 스타일이 있으며, 각각 장단점이 다릅니다. 특히 최상위 수준 분할은 컴포넌트 배열에 큰 영향을 미칩니다.

그림 9-2에 나타난 두 가지 아키텍처 스타일 중 하나는 많은 사람들에게 익숙한 계층형 모놀리스(10장에서 자세히 설명)입니다. 다른 하나는 모듈형 모놀리스라고 불리는 아키텍처 스타일로, **사이먼 브라운**이 대중화한 것으로, 데이터베이스와 연결된 단일 배포 단위로 구성되며 기술적 기능이 아닌 도메인을 중심으로 분할됩니다(11장에서 자세히 설명). 이 두 스타일은 최상위 계층 분할의 서로 다른 방식을 나타냅니다. 두 가지 방식 모두에서 각 최상위 계층 또는 구성 요소에는 다른 구성 요소가 내장되어 있을 가능성이 높습니다. 최상위 계층 분할은 아키텍처 스타일과 코드 분할 방식을 정의하기 때문에 아키텍트에게 특히 중요합니다.

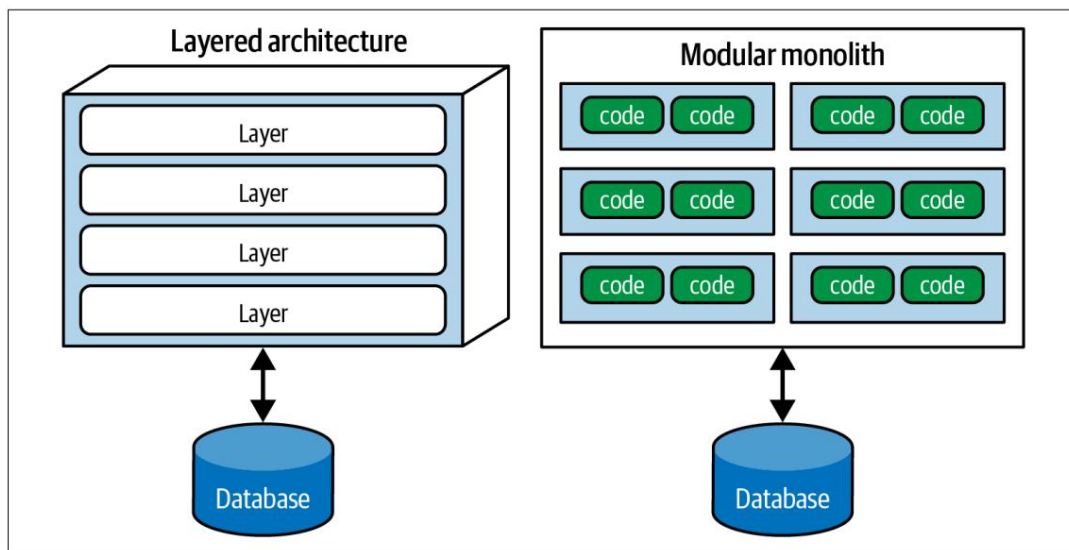


그림 9-2. 최상위 파티셔닝의 두 가지 유형: 기술적 파티셔닝(예: 계층형 아키텍처)과 도메인 파티셔닝(예: 모듈형 모놀리스)

계층형 모놀리스 스타일처럼 기술적 역량을 기준으로 아키텍처를 구성하는 것은 기술적 최상위 수준의 파티셔닝을 나타냅니다.

그림 9-3에서 볼 수 있듯이, 이러한 구성의 일반적인 예는 시스템 설계자가 시스템 기능을 프레젠테이션, 비즈니스 규칙, 서비스, 데이터 영속성 등과 같은 기술적 역량으로 분할한 것입니다. 이러한 방식으로 코드베이스를 구성하는 것은 분명히 합리적입니다. 모든 데이터 영속성 관련 코드가 하나의 계층에 존재하므로 개발자가 쉽게 접근할 수 있습니다.

영속성 관련 코드를 찾기 위해서입니다. 계층형 아키텍처의 기본 개념은 수십 년 전부터 존재했지만, 모델-뷰-컨트롤러(MVC) 디자인 패턴(에릭 프리먼과 엘리자베스 롱슨의 저서 『헤드 퍼스트 디자인 패턴』 (오라일리, 2020)에 소개된 기본 패턴 중 하나)은 이러한 아키텍처 패턴과 잘 맞아떨어져 개발자들이 쉽게 이해할 수 있습니다. 따라서 많은 조직에서 기본 아키텍처로 사용되고 있습니다.

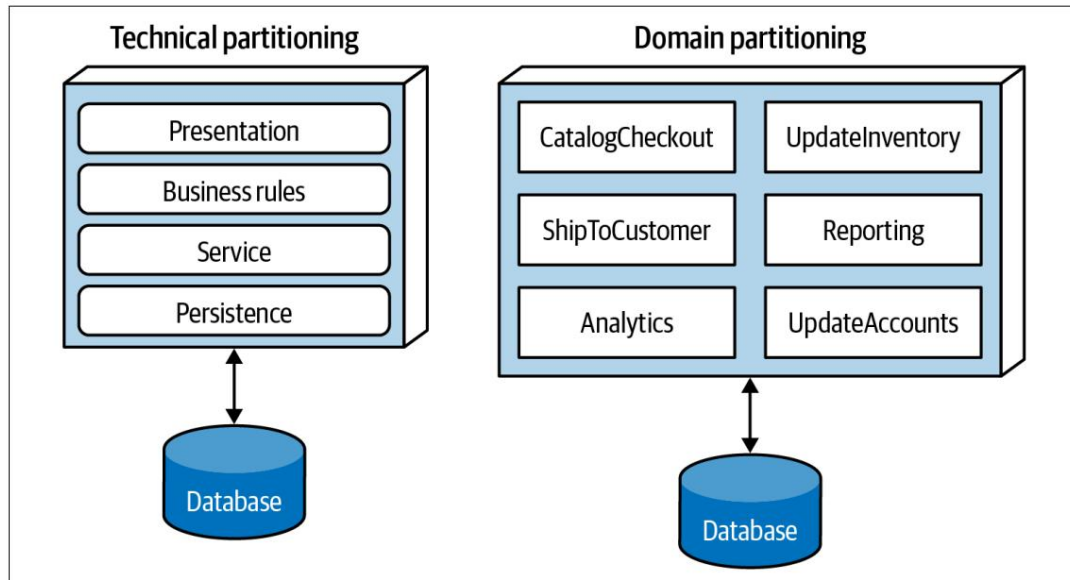


그림 9-3. 아키텍처에서 최상위 파티셔닝의 두 가지 유형

계층형 아키텍처가 널리 사용되면서 나타나는 흥미로운 부작용 중 하나는 기업들이 물리적인 사무실의 좌석 배치를 프로젝트 역할별로 나누는 방식과 관련이 있습니다. 콘웨이의 법칙에 따라 계층형 아키텍처를 사용할 경우, 백엔드 개발자들은 한 부서에, 데이터베이스 관리자(DBA)들은 다른 부서에, 프레젠테이션 팀은 또 다른 부서에 배치하는 것이 타당해 보입니다.

그림 9-3에 나타난 또 다른 아키텍처 변형은 도메인 파티셔닝입니다. 이는 복잡한 소프트웨어 시스템을 기술적 기능이 아닌 도메인별로 구성 요소를 분해하는 모델링 기법입니다. 도메인 파티셔닝은 에릭 에반스의 저서 "도메인 주도 설계(Domain-Driven Design, DDD)"에서 영감을 받았습니다. DDD에서 아키텍트는 서로 독립적이고 결합도가 낮은 도메인 또는 워크플로우를 식별합니다. 마이크로서비스 아키텍처 스타일은 이러한 철학을 기반으로 합니다.

모듈형 모놀리식 아키텍처는 기술적 기능보다는 도메인이나 워크플로를 중심으로 분할됩니다. 구성 요소들이 서로 중첩되는 경우가 많기 때문에 그림 9-3에 나타난 도메인 분할 아키텍처(예: CatalogCheckout)의 각 구성 요소는 영구 저장 라이브러리를 사용하고 비즈니스 규칙을 위한 별도의 계층을 가질 수 있지만, 최상위 수준의 분할은 여전히 도메인을 중심으로 이루어집니다.

콘웨이의 법칙

1960년대 후반, **멜빈 콘웨이**는 콘웨이의 법칙으로 알려지게 된 관찰 결과를 발표했습니다.

시스템을 설계하는 조직은 해당 조직의 의사소통 구조를 그대로 복제한 설계를 만들어낼 수밖에 없다는 제약을 받습니다.

쉽게 말해, 이 법칙은 한 집단이 기술적 산물을 설계할 때, 그 설계의 구조는 그 집단 구성원들이 소통하는 방식을 반영한다는 것을 시사합니다.

조직의 모든 계층에 있는 사람들은 이 법칙이 작용하는 것을 목격하고 때로는 이를 바탕으로 결정을 내립니다. 예를 들어, 조직에서 직원들을 기술적 역량에 따라 나누는 것은 흔한 일이지만, 이는 공통의 관심사를 인위적으로 분리하는 것으로 협업을 방해할 수 있습니다.

Thoughtworks의 Jonny Leroy가 제시한 관련 개념으로는 '**역 콘웨이 기법(Inverse Conway Maneuver)**'이 있는데, 이는 원하는 아키텍처를 구축하기 위해 팀과 조직의 구조를 함께 발전시켜 나가는 것을 제안합니다. 이 개념은 이후 '팀 토폴로지'라는 용어로 널리 알려지게 되었습니다.

기업들은 팀 구성 방식이 소프트웨어 아키텍처를 포함한 비즈니스의 여러 중요한 측면에 상당한 영향을 미칠 수 있다는 사실을 깨닫기 시작했습니다.

이어지는 스타일 중심의 장에서는 각 아키텍처 스타일이 이러한 팀 유형에 미치는 영향에 대해 논의합니다.

아키텍처 패턴 간의 근본적인 차이점 중 하나는 각 패턴이 지원하는 최상위 파티셔닝 유형입니다. 이어지는 스타일별 장에서 각 패턴에 대한 이러한 차이점을 자세히 다룹니다. 최상위 파티셔닝은 아키텍트가 구성 요소를 기술적인 관점에서 처음 식별하지 않으면 도메인별로 식별할지를 결정하는 데에도 큰 영향을 미칩니다.

기술적 파티셔닝을 사용하는 아키텍트는 시스템 구성 요소를 기술적 기능별로 구성합니다. 예를 들어, 표현 방식, 비즈니스 규칙, 영속성 저장 방식 등으로 구분합니다. 계층형 아키텍처의 주요 원칙 중 하나는 기술적 관심사를 분리하는 것으로, 이를 통해 구성 요소 간의 결합도를 효과적으로 낮출 수 있습니다. 예를 들어, 서비스 계층이 하위의 영속성 계층과 상위의 비즈니스 규칙 계층에만 연결되어 있다면, 영속성 계층의 변경은 해당 계층에만 영향을 미칠 가능성이 높습니다. 이러한 결합도 감소는 종속 구성 요소에 연쇄적인 부작용이 발생할 가능성을 줄여줍니다.

기술적 분할을 사용하여 시스템을 구성하는 것은 분명히 논리적이지만, 소프트웨어 아키텍처의 모든 것과 마찬가지로 몇 가지 절충점이 있습니다. 기술적 분할을 통해 개발자는 기능별로 정리된 코드베이스의 특정 범주를 빠르게 찾을 수 있지만, 대부분의 실제 소프트웨어 시스템은 기술적 기능을 넘나드는 워크플로를 필요로 합니다.

그림 9-4에 나타난 기술적으로 분할된 아키텍처에서 카탈로그 체크아웃의 일반적인 비즈니스 워크플로를 생각해 보겠습니다. 기술적으로 분할된 아키텍처에서 카탈로그 체크아웃을 처리하는 코드는 모든 계층에 나타납니다. 즉, 도메인이 기술 계층 전체에 걸쳐 분산되어 있습니다.

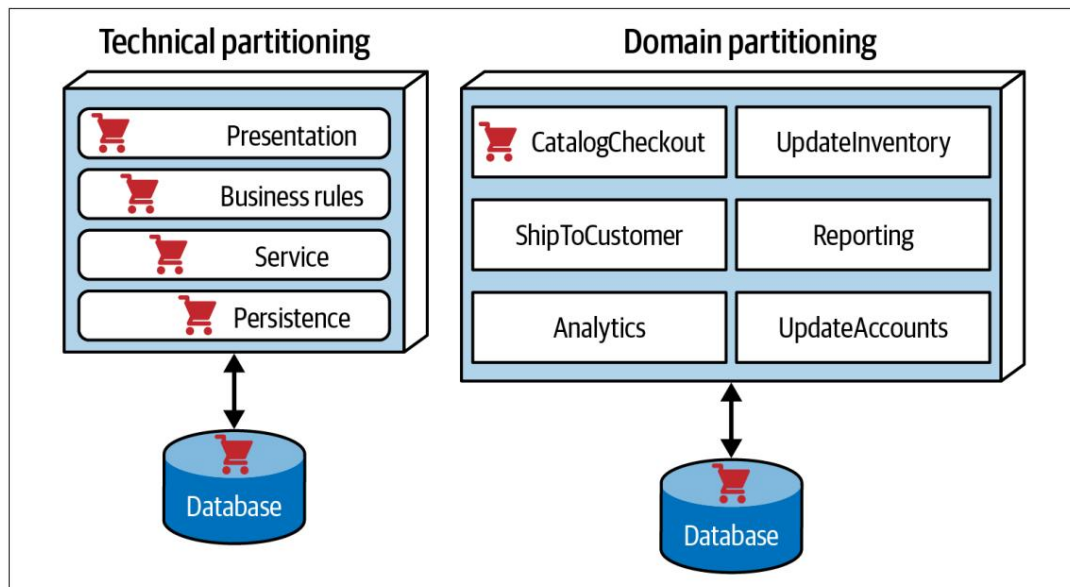


그림 9-4. 기술적으로 분할된 아키텍처와 도메인별로 분할된 아키텍처에서 도메인/워크플로우가 나타나는 위치

그림 9-4에 나타난 도메인 분할 아키텍처와 비교해 보면, 이 아키텍처에서는 설계자들이 워크플로와 도메인을 중심으로 최상위 구성 요소를 구축했습니다. 각 구성 요소에는 레이어를 포함한 하위 구성 요소가 있을 수 있지만, 최상위 분할은 도메인에 초점을 맞추어 프로젝트에서 가장 자주 발생하는 변경 사항을 더 잘 반영합니다.

이 두 가지 스타일 중 어느 하나가 다른 하나보다 더 옳다고 할 수는 없습니다. (소프트웨어 아키텍처의 제1법칙을 참조하십시오.) 하지만 지난 몇 년 동안 모놀리식 아키텍처와 분산형(예: 마이크로서비스) 아키텍처 모두에서 도메인 분할을 선호하는 업계의 뚜렷한 추세를 관찰해 왔습니다. 앞서 언급했듯이, 이는 아키텍트가 가장 먼저 내려야 할 결정 중 하나입니다.

Kata: 실리콘 샌드위치 - 파티셔닝

71페이지에 있는 예시 카타 중 하나인 "카타: 실리콘 샌드위치"의 경우를 생각해 보십시오.

먼저 실리콘 샌드위치의 두 가지 가능성 중 첫 번째인 도메인 분할(그림 9-5 참조)을 살펴보겠습니다.

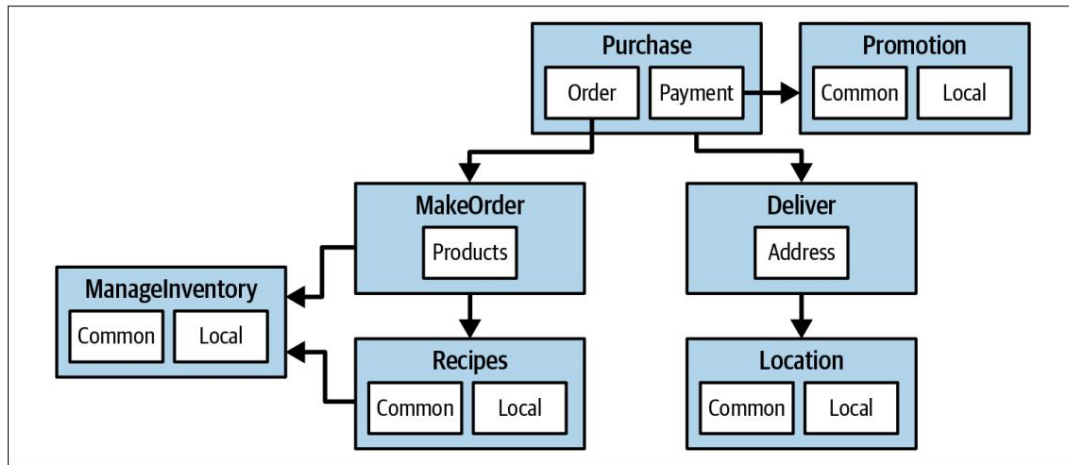


그림 9-5. 실리콘 샌드위치를 위한 도메인 분할 설계

그림 9-5에서 아키텍트는 도메인(워크플로우)을 중심으로 설계하여 구매, 프로모션, 주문 제작, 재고 관리, 레시피, 배송 및 위치에 대한 개별 구성 요소를 만들었습니다. 이러한 구성 요소 내에는 일반적인 유형과 지역별 변형 유형의 사용자 정의를 처리하는 하위 구성 요소가 포함되어 있습니다.

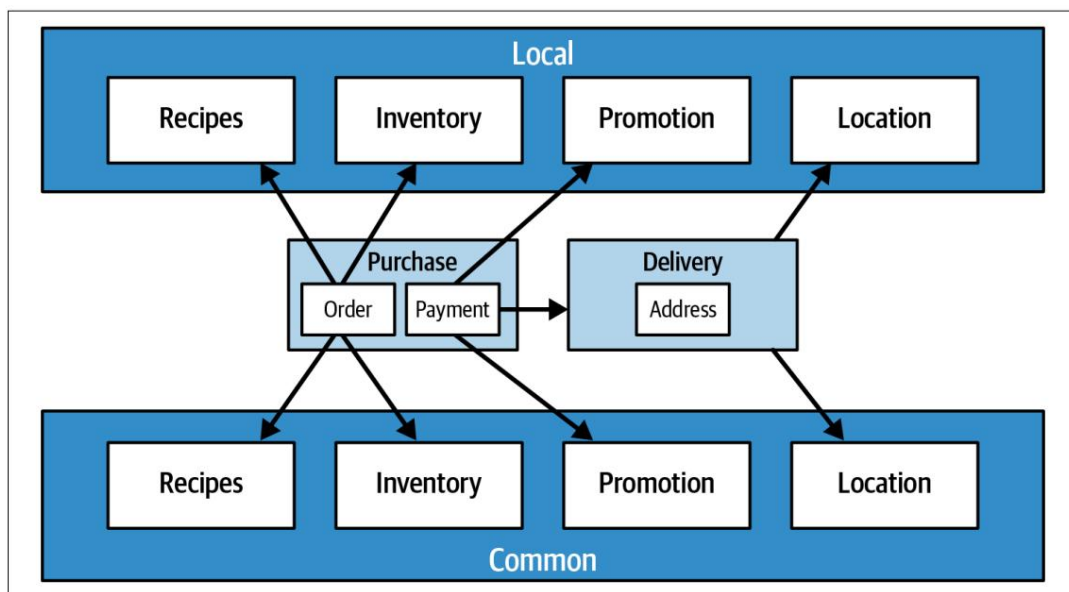


그림 9-6. 실리콘 샌드위치를 위한 기술적으로 분할된 설계

대안적인 설계에서는 그림 9-6에 나타난 것처럼 공통 부분과 로컬 부분을 각각 별도의 파티션으로 분리합니다. 공통 부분과 로컬 부분은 최상위 구성 요소를 나타내고, 구매 부분과 배송 부분은 워크플로를 처리하는 역할을 합니다.

어느 방식이 더 좋을까요? 상황에 따라 다릅니다! 각 파티셔닝 방식은 서로 다른 장점과 단점을 가지고 있습니다.

도메인 분할 아키텍처는 위

크플로 및/또는 도메인별로 최상위 구성 요소를 분리합니다.

장점 • 구현

방식이 아닌 비즈니스 운영 방식에 더 가깝게 설계되었습니다!

사고 과정의 세부 사항

- 도메인을 중심으로 여러 분야의 전문가로 구성된 팀을 더 쉽게 구성할 수 있습니다.
- 모듈형 모놀리식 및 마이크로서비스 아키텍처와 더욱 밀접하게 연관됩니다.
스타일
- 메시지 흐름이 문제 영역과 일치함 • 데이터 및 구성 요소를 분산 아키텍처로 쉽게 마이그레이션 가능

단점 • 사용자

지정 코드가 여러 곳에 나타납니다

기술적 분할 아키텍처는 개별 워크

플로가 아닌 기술적 기능을 기준으로 최상위 구성 요소를 분리합니다. 이는 모델-뷰-컨트롤러(MVC) 분리 또는 기타 임의적인 기술적 분할에서 영감을 받은 계층 구조로 나타낼 수 있습니다. **그림 9-6**에 나타난 아키텍처는 사용자 정의 기능을 기준으로 구성 요소를 분리합니다.

장점 • 사용

자 정의 코드를 명확하게 분리합니다. • 계층형 아키텍처 패턴에 더욱 부합합니다.

단점 • 전역적 결

합도가 높아짐. 공통 또는 로컬 구성 요소의 변경

이는 다른 모든 구성 요소에 영향을 미칠 가능성이 높습니다.

- 개발자는 공통 영역과 개별 영역 모두에 도메인 개념을 복제해야 할 수도 있습니다.
로컬 레이아웃.
- 일반적으로 데이터 수준에서 결합도가 높습니다. 이러한 시스템에서는 애플리케이션 아키텍트와 데이터 아키텍트가 협력하여 사용자 정의 및 도메인을 포함한 단일 데이터베이스를 구축할 가능성이 높습니다. 이는 나중에 아키텍트가 이 아키텍처를 분산 시스템으로 마이그레이션하려는 경우 데이터 관계를 풀어내는 데 어려움을 초래할 수 있습니다. 아키텍처 스타일을 선택하는 데 영향을 미치는 다른 많은 요소들이 있으며, 이는 **2부에서 다룹니다.**

단일형 아키텍처와 분산형 아키텍처 비교

1부에서 배웠듯이 아키텍처 스타일은 크게 두 가지 유형으로 분류할 수 있습니다. 하나는 모든 코드가 하나의 배포 단위로 존재하는 모놀리식 아키텍처이고, 다른 하나는 원격 액세스 프로토콜을 통해 연결된 여러 배포 단위로 구성된 분산 아키텍처입니다. 완벽한 분류 체계는 없지만, 분산 아키텍처는 모놀리식 아키텍처 스타일에는 없는 공통적인 문제점과 어려움을 가지고 있기 때문에 이러한 분류 체계는 다양한 아키텍처 스타일을 구분하는 데 유용한 기준이 됩니다.

이 책의 2부에서는 다음과 같은 건축 양식을 자세히 설명합니다.

모놀리식 아키텍

처 • 계층형 아키텍처 (10장) • 파이프라인 아키텍처
(12장) • 마이크로커널 아키텍처 (13장)

분산됨

• 서비스 기반 아키텍처 (14장) • 이벤트 기반 아키텍처
(15장) • 공간 기반 아키텍처 (16장) • 서비스 지향 아
키텍처 (17장) • 마이크로서비스 아키텍처 (18장)

분산 아키텍처는 단일 아키텍처에 비해 성능, 확장성 및 가용성 측면에서 훨씬 강력하지만, 상당한 단점도 가지고 있습니다. 모든 분산 아키텍처가 직면한 첫 번째 문제점은 1994년 Sun Microsystems의 L. Peter Deutsch와 그의 동료들이 처음으로 제시한 "**분산 컴퓨팅의 오류**"에 설명되어 있습니다. 오류란 누군가가 사실이라고 믿거나 가정하는 거짓된 것을 말합니다. 분산 컴퓨팅의 여덟 가지 오류는 오늘날의 분산 아키텍처에도 모두 적용됩니다. 다음 섹션에서는 각 오류에 대해 설명합니다.

오류 1: 네트워크는 신뢰할 수 있다

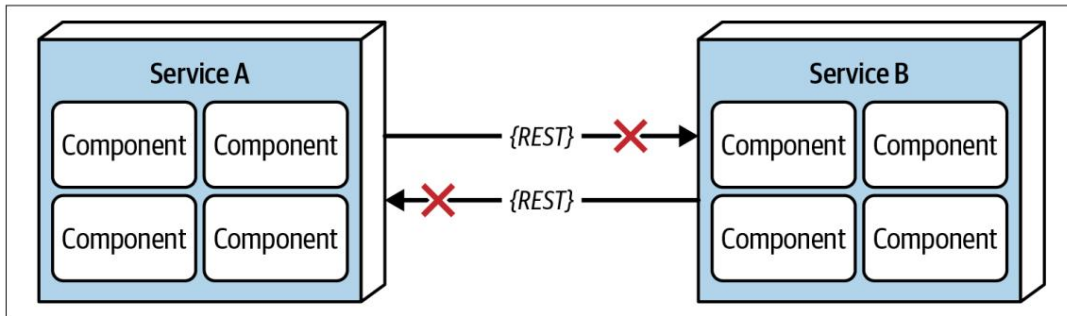


그림 9-7. e 네트워크는 신뢰할 수 없습니다.

개발자와 설계자 모두 네트워크가 안정적이라고 생각하지만, 실제로는 그렇지 않습니다. 시간이 지남에 따라 네트워크의 신뢰성은 향상되었지만, 여전히 전반적으로 네트워크는 불안정한 상태입니다. 이는 모든 분산 아키텍처 스타일에 중요한 문제인데, 분산 아키텍처는 서비스 간 통신을 위해 네트워크에 의존하기 때문입니다. 그림 9-7에서 볼 수 있듯이, 서비스 B는 완전히 정상 작동 중일 수 있지만, 네트워크 문제로 인해 서비스 A가 서비스 B에 접근할 수 없습니다. 더 심각한 경우, 서비스 A가 서비스 B에 데이터 처리를 요청했지만 네트워크 문제로 인해 응답을 받지 못할 수도 있습니다. 이것이 바로 서비스 간에 타임아웃이나 회로 차단기와 같은 장치가 존재하는 이유입니다. 시스템이 네트워크에 더 많이 의존할수록(특히 마이크로서비스 아키텍처의 경우) 불안정해질 가능성이 더 커집니다.

두 번째 오류: 지연 시간은 0이다

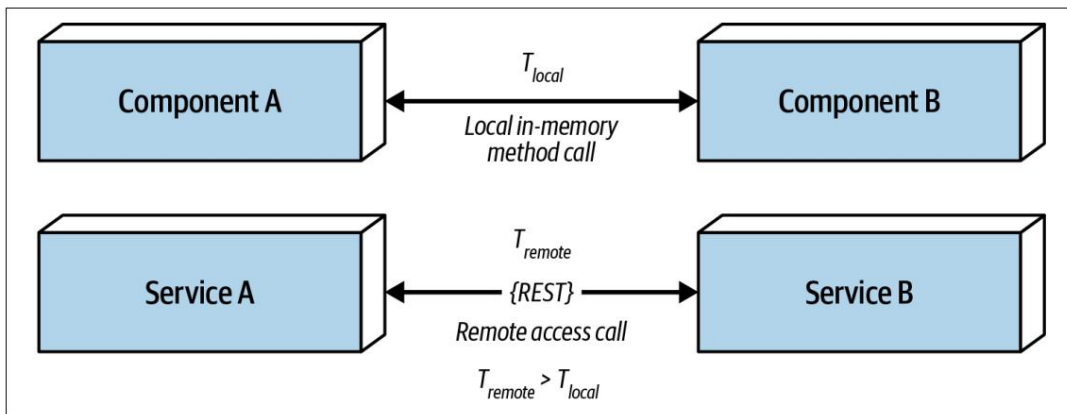


그림 9-8. 지연 시간은 0이 아닙니다.

그림 9-8에서 볼 수 있듯이, 메서드 또는 함수 호출을 통해 다른 구성 요소에 로컬 호출을 할 경우 해당 구성 요소에 접근하는 데 걸리는 시간 (t_{local})은 나노초 또는 마이크로초 단위로 측정됩니다. 그러나 동일한 호출이 원격 액세스 프로토콜(예: REST, 메시징 또는 RPC)을 통해 이루어질 경우, 시간 (t_{remote})은 다른 단위로 측정됩니다.

지연 시간은 밀리초 단위이므로 항상 t_{local} 보다 큼니다. 분산 아키텍처에서 지연 시간은 0이 될 수 없지만, 대부분의 설계자는 빠른 네트워크를 사용하고 있다고 주장하며 이러한 오류를 무시합니다. 스스로에게 질문해 보세요. 실제 운영 환경에서 RESTful 호출의 평균 왕복 지연 시간이 얼마인지 알고 있습니까? 60밀리초입니까? 아니면 500밀리초입니까?

분산 아키텍처, 특히 마이크로서비스를 사용할 때는 아키텍트는 반드시 평균 지연 시간을 알아야 합니다. 서비스의 세분화된 특성과 서비스 간 통신량 때문에 분산 아키텍처의 실현 가능성을 판단하는 유일한 방법이기 때문입니다.

예를 들어, 요청당 평균 지연 시간이 100ms라고 가정해 보겠습니다. 특정 비즈니스 기능을 수행하기 위해 서비스 호출을 연결하면 요청에 1,000ms가 추가됩니다! 평균 지연 시간을 아는 것도 중요하지만, 95번째 백분위수부터 99번째 백분위수까지의 지연 시간을 아는 것은 훨씬 더 중요합니다. 시스템의 평균 지연 시간이 60ms(양호한 수준)일지라도 95번째 백분위수는 400ms일 수 있습니다! 분산 아키텍처에서 성능을 저하시키는 것은 대개 이러한 "롱테일" 지연 시간입니다. 대부분의 경우 네트워크 관리자가 지연 시간 값을 제공할 수 있습니다(148페이지의 "오해 #6: 관리자는 한 명뿐이다" 참조).

세 번째 오류: 대역폭이 대세다!

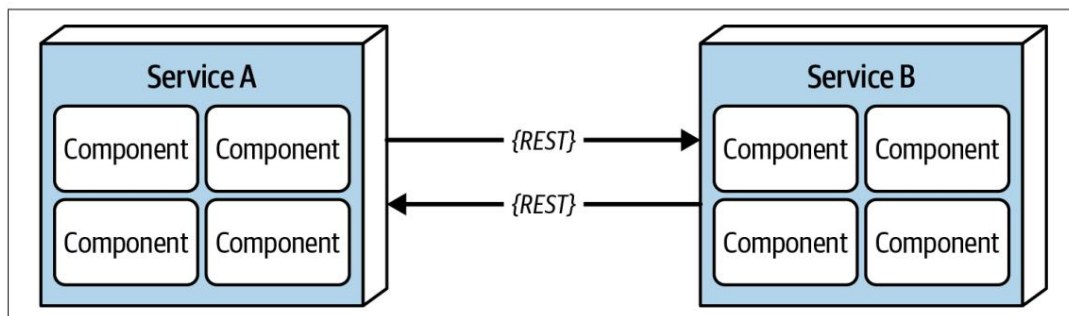


그림 9-9. 대역폭은 무한하지 않다

모놀리식 아키텍처에서는 일반적으로 대역폭이 문제가 되지 않습니다. 비즈니스 요청이 모놀리식 시스템에 들어가면 처리하는 데 필요한 대역폭이 거의 없거나 전혀 없기 때문입니다. 그러나 **그림 9-9에서 볼 수 있듯이**, 마이크로서비스와 같은 분산 아키텍처에서 시스템이 더 작은 배포 단위(서비스)로 분할되면 이러한 서비스 간의 통신에 상당한 대역폭이 사용됩니다. 이는 네트워크 속도를 저하시켜 지연 시간(오류 #2)과 신뢰성(오류 #1)에 영향을 미칩니다.

이 오류의 중요성을 설명하기 위해 그림 9-9에 나타난 두 서비스를 생각해 보겠습니다. 서비스 A는 웹사이트의 위시리스트 항목을 관리하고, 서비스 B는 고객 프로필을 관리한다고 가정해 봅시다. 위시리스트 요청이 서비스 A로 들어올 때마다, 서비스 A는 위시리스트 응답 계약에 필요한 고객 이름을 얻어야 합니다. 이름을 얻기 위해 서비스 B에 서비스 간 호출을 해야 합니다. 서비스 B는 총 500KB에 해당하는 45개의 속성을 서비스 A로 반환하는데, 서비스 A는 이름(200바이트)만 필요로 합니다. 언뜻 보기에는 큰 차이가 없어 보일 수 있지만, 위시리스트 항목 요청은 초당 약 2,000번 발생합니다. 즉, 서비스 A는 초당 2,000번 서비스 B를 호출하는 것입니다. 요청당 500KB의 데이터를 사용한다고 가정하면, 각 서비스 간 호출은 1Gbps의 대역폭을 소모합니다.

스탬프 결합이라고 불리는 이러한 형태의 결합은 분산 아키텍처에서 상당한 대역폭을 소모합니다. 만약 서비스 B가 서비스 A에 필요한 200바이트의 데이터만 전달한다면, 총 400Kbps의 대역폭만 사용하게 됩니다.

스탬프 결합 문제는 다음과 같은 방법으로 해결할 수 있습니다.

- 비공개 RESTful API 엔드포인트 생성 • 계약에서 필드 선택기 사용 • GraphQL을 사용하여 계약 분리 • 가치 중심 계약과 소비자 중심 계약의 결합 • 내부 메시징 엔드포인트 사용

어떤 기술을 사용하든 분산 아키텍처에서 이러한 오류를 해결하는 가장 좋은 방법은 서비스나 시스템이 필요한 데이터만 전송하도록 하는 것입니다.

오류 4: 네트워크는 안전하다

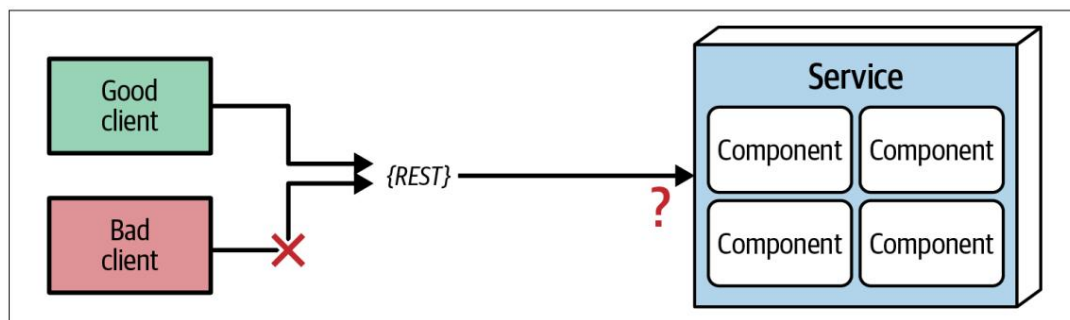


그림 9-10. e 네트워크는 안전하지 않습니다.

대부분의 아키텍트와 개발자는 가상 사설망(VPN), 신뢰 네트워크, 방화벽 사용에 너무 익숙해져서 분산 컴퓨팅의 이러한 함정, 즉 네트워크가 안전하지 않다는 점을 간과하는 경향이 있습니다. 그림 9-10에서 보는 것처럼 각 분산 배포 단위의 모든 엔드포인트는 보안이 강화되어야 합니다.

알 수 없거나 잘못된 요청이 발생할 수 있습니다. 모놀리식 아키텍처에서 분산 아키텍처로 전환할 때 위협 및 공격에 노출되는 면적이 크게 증가하여 보안이 훨씬 더 어려워집니다. 서비스 간 통신을 포함한 모든 엔드포인트를 보호해야 한다는 점 또한 마이크로서비스 및 서비스 기반 아키텍처와 같은 동기식 고도 분산 아키텍처 스타일에서 성능이 저하되는 또 다른 이유입니다.

오류 5: 위상은 절대 변하지 않는다

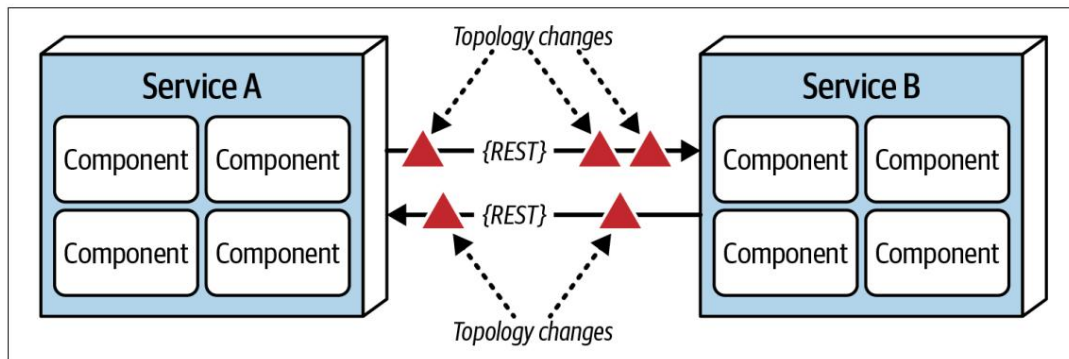


그림 9-11. 네트워크 토폴로지는 항상 변합니다.

그림 9-11에 나타난 오류 5는 라우터, 허브, 스위치, 방화벽, 네트워크 및 어플라이언스를 포함한 전체 네트워크 토폴로지에 관한 것입니다. 설계자들은 토폴로지가 고정되어 절대 변하지 않는다고 가정합니다. 물론 토폴로지는 끊임없이 변합니다. 이 오류가 갖는 의미는 무엇일까요?

월요일 아침 출근했는데 프로덕션 환경에서 서비스 타임아웃이 계속 발생해서 모두가 정신없이 뛰어다니고 있다고 가정해 보세요. 당신은 팀들과 함께 이 문제가 왜 발생하는지 알아내려고 안간힘을 씁니다. 주말 동안 새로운 서비스는 배포된 적이 없습니다. 도대체 무엇이 문제일까요? 몇 시간 후, 당신은 새벽 2시에 진행된 "사소한" 네트워크 업그레이드가 시스템의 모든 지연 시간 가정을 무효화시켜 타임아웃과 회로 차단을 유발했다는 사실을 발견합니다.

아키텍트는 운영 및 네트워크 관리자와 지속적으로 소통하여 변경 사항과 시기를 파악하고, 예상치 못한 상황을 방지하기 위해 필요한 조정을 해야 합니다. 이는 당연하고 쉬워 보이지만, 실제로는 그렇지 않습니다. 사실, 이러한 오류는 곧바로 다음 오류로 이어집니다.

오류 6: 관리자는 단 한 명뿐이다



그림 9-12. 네트워크 관리자는 한 명이 아니라 여러 명 있습니다.

아키텍트들은 흔히 이러한 오류에 빠지곤 합니다. 바로 관리자 한 명과만 협업하고 소통하면 된다고 생각하는 것입니다. 그림 9-12에서 볼 수 있듯이, 일반적인 대기업에는 수십 명의 네트워크 관리자가 있습니다. 그렇다면 아키텍트는 자연 시간이나 토폴로지 변경에 대해 누구와 논의해야 할까요? 이러한 오류는 분산 아키텍처의 복잡성과 모든 것이 제대로 작동하도록 하기 위해 필요한 조정 작업의 양을 보여줍니다. 단일 배포 단위를 가진 모놀리식 애플리케이션은 이 정도의 소통과 협업을 필요로 하지 않습니다.

오류 7: 운송비는 0이다

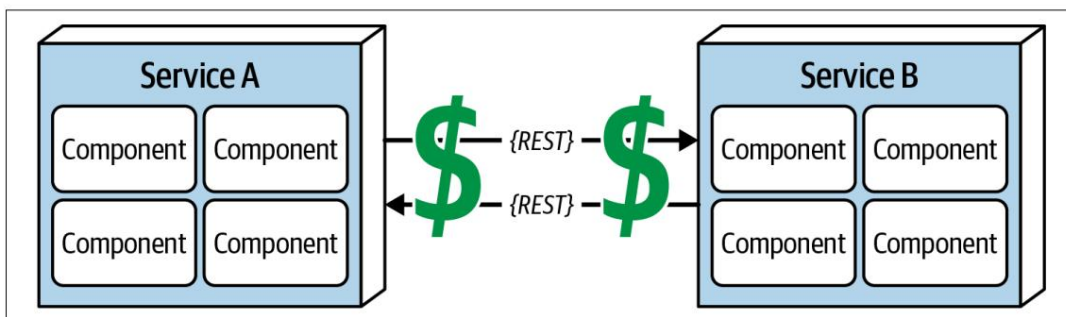


그림 9-13. 원격 접속에는 비용이 발생합니다.

많은 소프트웨어 아키텍트들이 그림 9-13에 나타난 이 오류를 두 번째 오류(자연 시간이 0이라는 가정)와 혼동합니다. 여기서 전송 비용은 자연 시간이 아니라 "간단한 RESTful 호출"을 수행하는 데 드는 실제 금전적 비용을 의미합니다. 아키텍트들은 간단한 RESTful 호출을 수행하거나 모놀리식 애플리케이션을 분산시키는 데 필요한 충분한 인프라가 이미 구축되어 있다고 잘못 생각합니다. 하지만 대개 그렇지 않습니다. 분산 아키텍처는 하드웨어, 서버, 게이트웨이, 방화벽, 새로운 서브넷, 프록시 등에 대한 필요성이 증가하기 때문에 모놀리식 아키텍처보다 훨씬 더 많은 비용이 듭니다.

끝.

분산 아키텍처를 설계하려는 건축가들은 이러한 오류로 인해 예상치 못한 문제에 직면하는 것을 방지하기 위해 용량, 대역폭, 지연 시간 및 보안 영역 측면에서 현재 서버 및 네트워크 토폴로지를 분석할 것을 권장합니다.

오류 8: 네트워크는 동질적이다

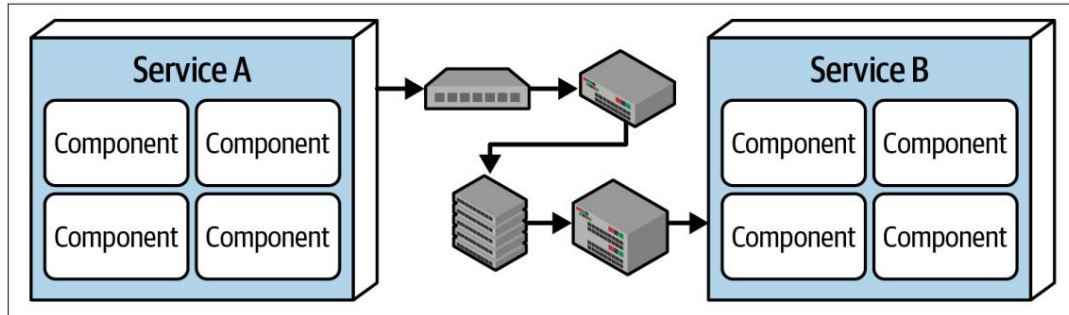


그림 9-14. e 네트워크는 동질적이지 않습니다.

대부분의 설계자와 개발자는 그림 9-14 에서처럼 네트워크가 단일 공급업체의 네트워크 하드웨어로만 구성된 동질적인 네트워크라고 가정합니다. 하지만 이는 사실과 거리가 멉니다. 대부분의 기업 인프라는 여러 공급업체의 네트워크 하드웨어를 사용합니다.

그래서 어쩌라는 걸까요? 이 오류의 핵심은 다양한 하드웨어 제조사의 제품들이 서로 원활하게 호환되지 않는다는 점입니다. 주니퍼 네트워크스 하드웨어가 시스코 시스템즈 하드웨어와 완벽하게 통합될까요? 대부분은 작동하고, 네트워킹 표준도 시간이 지남에 따라 발전해 왔기 때문에 이러한 문제는 예전만큼 심각하지 않습니다. 하지만 모든 상황, 부하, 환경이 완벽하게 테스트된 것은 아니므로 네트워크 패킷 손실이 간혹 발생할 수 있습니다. 이는 네트워크 안정성과 지연 시간 및 대역폭에 대한 가정 및 주장에 영향을 미칩니다. 다시 말해, 이 오류는 다른 모든 오류와 연결되어 네트워크를 다룰 때 (분산 아키텍처를 사용할 때는 불가피하게 발생하는) 혼란과 좌절의 악순환을 초래합니다.

다른 오류들

앞서 언급한 여덟 가지 오류는 유명한 관찰 결과이며, 모든 건축가는 도이치의 목록을 통해 배우거나, 혹은 경험을 쌓아가는 과정에서 하나씩 뼈아픈 경험을 통해 배우게 됩니다. 저자들은 또한 그 유명한 목록을 확장하여, 거의 모든 건축가가 공통적으로 겪는 고통스러운 교훈들을 몇 가지 제시하고자 합니다.

오류 #9. 버전 관리는 쉽다. 두 서비

스가 통신해야 할 때, 통신에 필요한 정보를 포함하는 계약을 통해 정보를 주고받습니다. 서비스의 내부 구현은 시간이 지남에 따라 발전하여 서비스가 받아들이고 다른 서비스로 전달하는 필드가 변경되는 경우가 많습니다. 이 문제를 해결하는 한 가지 방법은 버전 관리를 사용하는 것입니다.

계약서, 즉 기존 계약서와 새 계약서에 대해 서로 다른 정보 세트를 포함하는 여러 버전을 만드는 것입니다. 그러나 이看似 간단한 보이는 결정은 여러 가지 절충안을 필요로 합니다.

- 팀에서 개별 서비스 수준으로 버전을 관리해야 할까요, 아니면 전체 시스템으로 관리해야 할까요? • 버전 관리 범위는 어디까지 확장해야 할까요? 아키텍처의 어느 부분에 버전 관리가 필요할까요?

그것을 지지하기 위해서요?

- 팀은 특정 시점에 몇 개의 버전을 지원해야 할까요? (일부 팀은 의도치 않게 서로 다른 목적으로 수십 개의 버전을 지원하는 경우가 있습니다.)

- 팀에서 시스템 수준 또는 서비스 수준에서 이전 버전을 더 이상 사용하지 않도록 해야 할까요? 서비스?

버전 관리는 서비스 간 통신을 발전시키는 데 합리적인 접근 방식이지만, 설계자가 예상해야 할 여러 가지 장단점이 있습니다.

오류 #10. 보상 업데이트는 항상 작동한다. 보상 업데이트는 오케

스트레이터 서비스와 같은 메커니즘을 통해 여러 관련 서비스가 동시에 업데이트되도록 하는 아키텍처 패턴입니다. 만약 업데이트가 제대로 이루어지지 않으면 오케스트레이터가 업데이트를 되돌립니다. 보상 업데이트는 오케스트레이터가 상태를 이전 상태로 되돌리는 되돌리기 작업을 실행하는 것입니다.

이는 대부분의 아키텍트가 아무 생각 없이 항상 작동한다고 가정하는 일반적인 패턴이지만, 실제로는 그렇지 않습니다. 보상 업데이트가 실패하면 어떻게 될까요? 아키텍트는 마이크로서비스와 같은 분산 아키텍처에서 복잡한 상호 작용이 어떻게 작동하는지 보여줄 때, 보상 업데이트가 어떻게 작동하는지도 함께 보여줘야 합니다. 따라서 마이크로서비스에서 트랜잭션 워크플로를 설계하는 아키텍트는 "일반적인" 보상 워크플로를 고려해야 할 뿐만 아니라, 업데이트와 보상 업데이트(또는 그 일부)가 모두 실패할 경우 어떻게 복구할지 또한 고려해야 합니다.

오류 #11. 관찰 가능성은 선택 사항이다(분산 아키텍처의 경우).

분산 아키텍처에서 아키텍트가 우선시하는 일반적인 아키텍처적 특징 중 하나는 관찰 가능성입니다. 즉, 모니터나 로그를 통해 각 서비스가 다른 서비스 및 생태계와 상호 작용하는 방식을 관찰할 수 있는 능력입니다. 로깅은 단일체 아키텍처에서도 유용하지만, 포괄적인 상호 작용 로그 없이는 디버깅하기 어려운 다양한 통신 오류 모드가 존재하는 분산 아키텍처에서는 필수적입니다.

팀 토폴로지 및 아키텍처

아키텍처와 팀들은 이전에 논의했던 아키텍처 분할의 의미를 넘어 아키텍처와 팀 토폴로지의 교차점에 대해 많은 연구를 진행해 왔습니다. Mat! thew Skelton과 Manuel Pais가 저술한 매우 영향력 있는 책인 "**Team Topologies**" (IT Revolution Press, 2019)에서는 소프트웨어 아키텍처와 교차하는 여러 팀 유형을 정의합니다.

스트림 중심 팀은 팀 토폴

로지 용어에서 특정 비즈니스 영역 또는 역량에 맞춰진 작업 흐름을 의미합니다. 스트림 중심 팀은 제품, 서비스 또는 특정 기능 세트와 같은 단일 작업 라인에 집중합니다.

흐름 중심 팀의 목표는 조직에 개별적인 가치를 제공하는 것이기 때문에 가능한 한 빠르게 움직이는 것입니다. 따라서 다른 유형의 팀들은 흐름 중심 팀의 진행을 방해할 수 있는 마찰 요소를 줄이도록 설계되었습니다. 팀들.

지원팀은 특정 역

량의 격차를 해소하고, 필요하지만 시급하지 않은 연구, 학습 및 기타 작업을 수행할 수 있는 공간을 제공합니다. 이들은 전문 분야의 지식과 자원을 제공하여 해당 분야 팀을 지원합니다. 훌륭한 지원팀은 협업적이고 적극적입니다.

복잡한 하위 시스템 팀

많은 시스템에는 고도로 전문화된 시스템이나 부품이 포함되어 있으며, 이러한 시스템이나 부품에는 마찬가지로 전문적인 기술이 필요합니다. 복잡한 하위 시스템 팀의 구성원은 복잡한 하위 시스템이나 영역을 완벽하게 이해하고 있으며, 스트림 중심 팀이 이를 적용하는 데 도움을 줄 수 있습니다. 이들의 목표는 다른 팀의 인지 부하를 줄이는 것입니다.

플랫폼 팀은 솔루션

션을 위한 내부 서비스와 구성 요소를 제공합니다. **에반 보처(Evan Botcher)**의 정의에 따르면, 플랫폼이란...

셀프 서비스 API, 도구, 서비스, 지식 및 자원을 기반으로 강력한 내부 제품을 구성할 수 있습니다. 자율적인 개발팀은 이 플랫폼을 활용하여 조정 작업을 줄이고 더 빠른 속도로 제품 기능을 제공할 수 있습니다.

플랫폼 팀은 다른 팀들을 지원하며, 불필요한 마찰을 제거하는 동시에 품질 및 보안과 같은 문제에 대한 필요한 거버넌스를 제공합니다.

이제 특정 스타일로 넘어가 볼까요!

건축가는 다양한 건축 양식을 이해해야만 철충 분석을 수행할 수 있습니다. 각 양식은 서로 다른 건축적 특성을 지니며, 각각의 양식이 가장 잘 어울리는 최적의 지점이 있습니다. 다양한 양식과 그 기본 철학을 학습함으로써 건축가는 각각의 양식이 언제 가장 효과적인지(또는 최소한 가장 덜 나쁜 경우인지) 더 잘 이해할 수 있습니다.