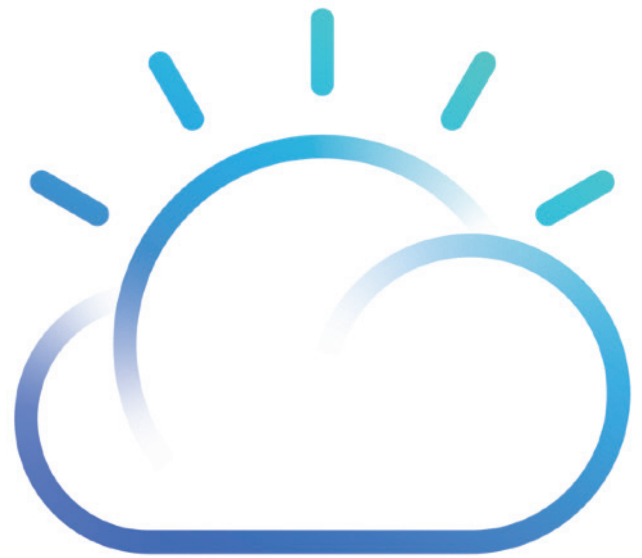
IBM **Cloud**

IBM

마이크로서비스 PoV 안내서

마이크로서비스 이해하기



Roland Barcia – IBM Distinguished 엔지니어, CTO Microservices, NYC IBM Cloud Garage
Kyle Brown – IBM Distinguished 엔지니어, CTO Cloud Architecture
Richard Osowski – IBM 수석 기술 직원, Microservices Adoption

목차

개요 3
마이크로서비스를 사용하여 비즈니스 문제 해결 10
모놀리식 애플리케이션 마이그레이션 10
마이크로서비스 아키텍처 12
마이크로서비스 기능 12
DevOps 13
마이크로서비스 컴퓨팅 옵션 13
인프라 서비스 14
애플리케이션 아키텍처 15
마이크로서비스 스택 17
애플리케이션 17
마이크로서비스 패브릭 17
DevOps 18
컨테이너 관리 18
마이크로서비스 기반 아키텍처의 탄력성 19
마이크로서비스에 대한 고가용성 및 장애 복구 패턴 19
액티브/패시브 21
액티브/대기 21
액티브/액티브 21
마이크로서비스 프로젝트 구현 22
기존 애플리케이션 진화 22
비즈니스 요구 사항 이해 및 정의 22
문화 및 기술 세트 이해 24
기술 이해 26
마이크로서비스 작업 규모 27
마이크로서비스 패턴 28
마이크로서비스에 대한 개발 패턴 28
마이크로서비스에 대한 운영 패턴 29
스트랭글러 패턴에 집중 30
스트랭글러 애플리케이션 패턴을 적용하지 않는 방법 32
스트랭글러 애플리케이션 패턴을 적용하는 최선의 방법 32
내부부터 시작: 32
참고 문헌 34

개요

마이크로서비스 애플리케이션 아키텍처 스타일에서, 애플리케이션은 마이크로서비스라는 네트워크로 연결된 여러 가지 개별 구성 요소로 *구성됩니다*. 마이크로서비스 아키텍처 스타일은 SOA(Services Oriented Architecture) 아키텍처 스타일이 진화한 것입니다 SOA 서비스를 사용하여 구축된 애플리케이션은 기술적 통합 문제에 집중하는 경향이 있으며, 구현된 서비스 수준은 종종 세밀한 (fine-grained)기술 API(Application Programming Interface)입니다. 이와는 대조적으로, 마이크로서비스 접근 방식은 거대한 (large-grained) 비즈니스 API를 통해 명확한 비즈니스 기능을 구현합니다.

두 가지 접근 방식의 가장 큰 차이점은 두 가지 방식이 배포되는 방식입니다. 수년 동안 애플리케이션은 모놀리식 방식으로 패키지되었습니다. 개발자 팀은 비즈니스 요구에 필요한 모든 것을 수행하는 하나의 큰 애플리케이션을 만듭니다. 일단 구축되면 애플리케이션 서버 팜 전체에 애플리케이션이 여러 번 배포됩니다. 개발자는 마이크로서비스 아키텍처 스타일에서 각각 애플리케이션의 일부를 구현하는 여러 개의 소형 애플리케이션을 독립적으로 구축하고 이를 하나로 묶습니다.

간단히 말해, 마이크로서비스 아키텍처는 대규모 사일로 애플리케이션을 훨씬 관리하기 쉽고 완벽하게 분리된 조각으로 분해하는 것입니다.

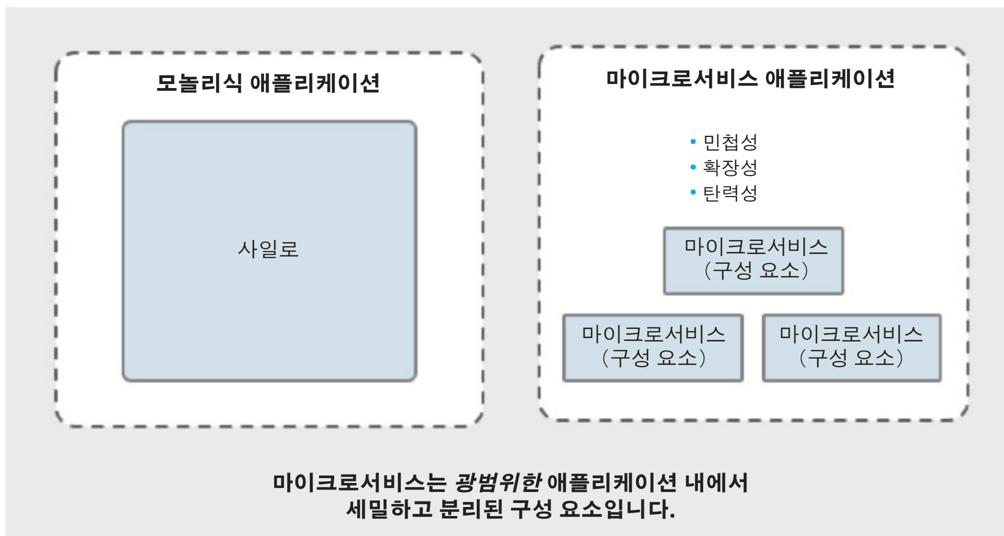


그림 1: 모놀리식 애플리케이션과 마이크로서비스 비교

마이크로서비스 아키텍처를 사용하여 구축한 애플리케이션의 구현을 유도하는 다음과 같은 5가지 간단한 규칙이 있습니다.

1. **거대한 모노리스를 여러 개의 작은 서비스로 분할** - 단일 네트워크 액세스 가능 서비스는 마이크로서비스 애플리케이션을 위해 배치 가능한 가장 작은 단위입니다. 각 서비스는 자체 프로세스를 실행합니다. 이 규칙은 컨테이너당 하나의 서비스(one service per container)라 합니다. 컨테이너는 Docker 컨테이너 또는 Cloud Foundry 런타임과 같이 간단한 다른 배포 메커니즘을 나타냅니다.

2. **단일 함수에 대한 서비스 최적화** - 기존 모놀리식 SOA 접근 방식에서 단일 애플리케이션 런타임은 여러 가지 비즈니스 기능을 수행합니다. 마이크로서비스 접근 방식에서, 서비스당 하나의 비즈니스 기능만 있습니다. 이를 통해 각 서비스를 작고 간단하게 만들어 작성하고 유지 관리할 수 있습니다. 이를 SRP(Single Responsibility Principle)라 합니다.

3. **REST API 및 메시지 브로커를 통한 커뮤니케이션** - SOA 접근 방식의 단점 중 하나는 SOA 서비스를 구현하기 위한 수많은 표준과 옵션이 있다는 것입니다. 마이크로서비스 접근 방식은 서비스가 구현할 수 있는 네트워크 연결 유형을 엄격히 제한하여 최대의 간결성을 달성합니다. 마찬가지로, 마이크로서비스는 데이터베이스를 통한 암시적 커뮤니케이션에 의해 야기된 밀집 결합을 피하는 경향이 있습니다. 서비스 간의 모든 커뮤니케이션은 서비스 API를 통해 이루어지거나 적어도 Claim Check Pattern[Hohpe and Woolf]과 같은 명백한 커뮤니케이션 패턴을 따라야 합니다.

4. **서비스별 CI/CD 적용** - 여러 서비스로 구성된 대형 애플리케이션에서 서로 다른 서비스가 상이한 속도로 발전합니다. 각 서비스에는 고유하고 지속적인 통합/전달 파이프라인이 있어 자연스러운 속도로 수익을 창출할 수 있습니다. 이는 시스템의 모든 부분이 시스템에서 가장 느리게 움직이는 부분의 속도로 강제 해제되는 모놀리식 접근 방식으로는 불가능합니다.

5. **서비스별 고가용성(HA)/클러스터링 결정 적용** - 대규모 시스템을 구축할 때 클러스터링은 모든 경우에 적용(one-size-fits-all)할 수 있는 단일 방식이 아닙니다. 동일한 수준의 모노리스에 있는 모든 서비스를 확장하는 모놀리식 접근 방식은 일부 서버를 과도하게 사용하고 다른 서비스를 너무 적게 사용하거나 심지어 스레드 풀 같은 사용 가능한 모든 공유 리소스를 독점하는 다른 요소에 의해 서비스가 고갈될 수 있습니다. 실제로 대형 시스템에서는 모든 서비스를 확장할 필요가 없습니다. 여러 가지 서비스를 최소한의 서버에 배치하여 리소스를 절약할 수 있습니다. 하지만 다른 시스템은 상당히 많은 서비스를 확장하도록 요구합니다.

다섯 가지 규칙과 이러한 규칙에 대한 이점이 결합되어 마이크로서비스 아키텍처가 널리 보급되게 되었습니다.

개요

마이크로서비스 아키텍처는 대규모 상업용 애플리케이션 개발을 위한 사실상의 표준이 되었습니다. Microservices a definition of this new architectural term라는 책에서, Martin Fowler는 마이크로서비스를 다음과 같이 정의합니다.

"간단히 말해, 마이크로서비스 아키텍처 스타일은 단일 애플리케이션을 작은 서비스 모음으로 개발하고 각각의 애플리케이션을 자체 프로세스에서 실행하며 종종 HTTP 리소스 API라는 경량 메커니즘과 커뮤니케이션하는 접근 방식입니다. 이러한 서비스는 비즈니스 기능을 기반으로 구축되며 완전 자동화된 배포 시스템을 통해 독립적으로 배포할 수 있습니다. 서로 다른 프로그래밍 언어로 작성되고 상이한 데이터 스토리지 기술을 사용할 수 있는 이러한 서비스에 대한 최소한의 중앙 집중식 관리가 이루어지고 있습니다."

마이크로서비스와 SOA 및 API와 같은 패러다임 간의 주요 차이점 중 하나는 배포에 중점을 두고 실행 중인 구성 요소에 중점을 둔다는 데 있습니다. 마이크로서비스는 아래 그림과 같이 인터페이스라기 보다 배포된 구성 요소의 세분성에 중점을 둡니다.

"마이크로서비스" 라는 용어로 인해 발생하는 일반적인 오해

~~마이크로서비스는 더 세분화된(파인 그레인드) 웹 서비스 API입니다.~~

"마이크로(Micro)" 는 노출된 인터페이스의 세분성이 아니라 ***구성 요소의***세분성을 의미합니다.

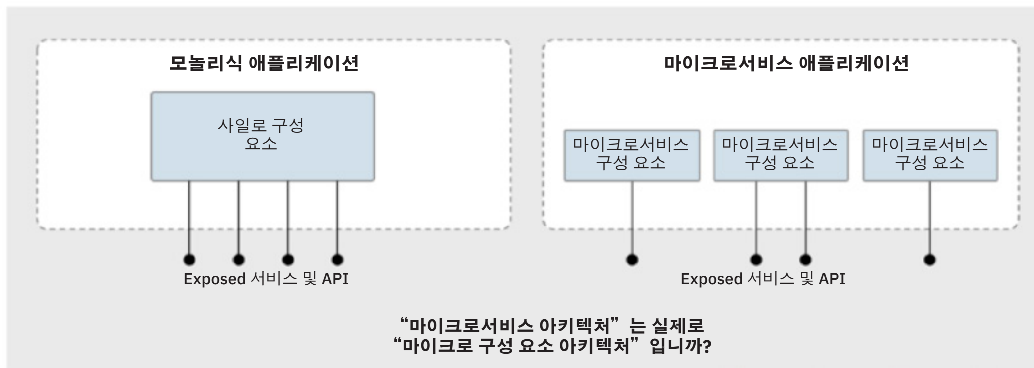


그림 2: API와 마이크로서비스 간의 차이

운영을 중심으로 하는 여러 조직들은 유즈 케이스와 기능이 하나의 대형 애플리케이션에 배포되는 모놀리식 애플리케이션을 구축합니다. 운영과 관련하여 다소 편한 부분이 있지만, 마이크로서비스에 기반하여 이러한 애플리케이션을 변경하기 위해서는 많은 노력이 필요합니다. 다음과 같은 세 가지 요소가 마이크로서비스 개발을 유도합니다.

1. **개발 팀을 구성하는 방법:** 마이크로서비스는 하나의 애플리케이션을 잘 정의된 인터페이스가 있는 단일 기능 모듈로 분해하는 데 중점을 두는 엔지니어링 접근 방식을 사용하여 개발하는 것이 가장 좋으며 이는 서비스의 전체 수명 주기를 담당하는 소규모 팀이 독립적으로 배포하고 운영합니다. 마이크로서비스는 사람들 간의 의사 소통 및 조정을 최소화하는 동시에 변화의 범위와 위험을 줄임으로써 전달 속도를 가속화합니다.

2. **앱을 구축하는 방법:** 마이크로서비스 기반 애플리케이션은 구축 방법과 실행 환경에 대해 몇 가지 가정을 내립니다. 환경은 종종 클라우드 기본 애플리케이션 또는 *12가지 요소 애플리케이션이라 합니다.* 마이크로서비스 기반 아키텍처는 이점을 활용하고 탄력적인 확장, 변경 불가능한 배치, 일회용 인스턴스 및 예측하기 어려운 인프라와 같은 개념을 비롯해 표준화된 클라우드 환경의 문제를 해결합니다.

3. **앱 제공 및 실행 방법:** 애플리케이션을 실행하는 표준 방법으로 컨테이너를 사용하면 마이크로서비스 기반 애플리케이션의 패키징과 실행이 이루어집니다. 컨테이너는 전혀 새로운 기술이 아닙니다. Linux 컨테이너는 하나의 컨트롤 Linux 호스트에서 여러 개의 분리된 Linux 시스템(또는 컨테이너)을 실행할 수 있는 운영 체제 수준의 기능입니다. Linux 컨테이너는 전체적인 VM을 대신하는 경량의 대체 솔루션입니다. 컨테이너가 전혀 새로운 개념은 아니지만 Docker와 같은 프레임워크는 컨테이너를 실행하기 위한 이미지 생성 방법을 고안하여 컨테이너가 널리 사용되도록 했습니다. 컨테이너는 애플리케이션과 모든 하위 애플리케이션을 패키지로 묶어 환경 간에 이동할 수 있고 변경없이 실행 가능한 기본적인 방식을 제공합니다. Cloud Foundry와 같은 다른 프레임워크는 컨테이너를 사용하여 애플리케이션을 실행하지만 가상화를 추상화합니다.

아래 그림에는 모놀리식 애플리케이션 아키텍처가 마이크로서비스 기반 애플리케이션 아키텍처로 어떻게 발전하는지 제시되어 있습니다.

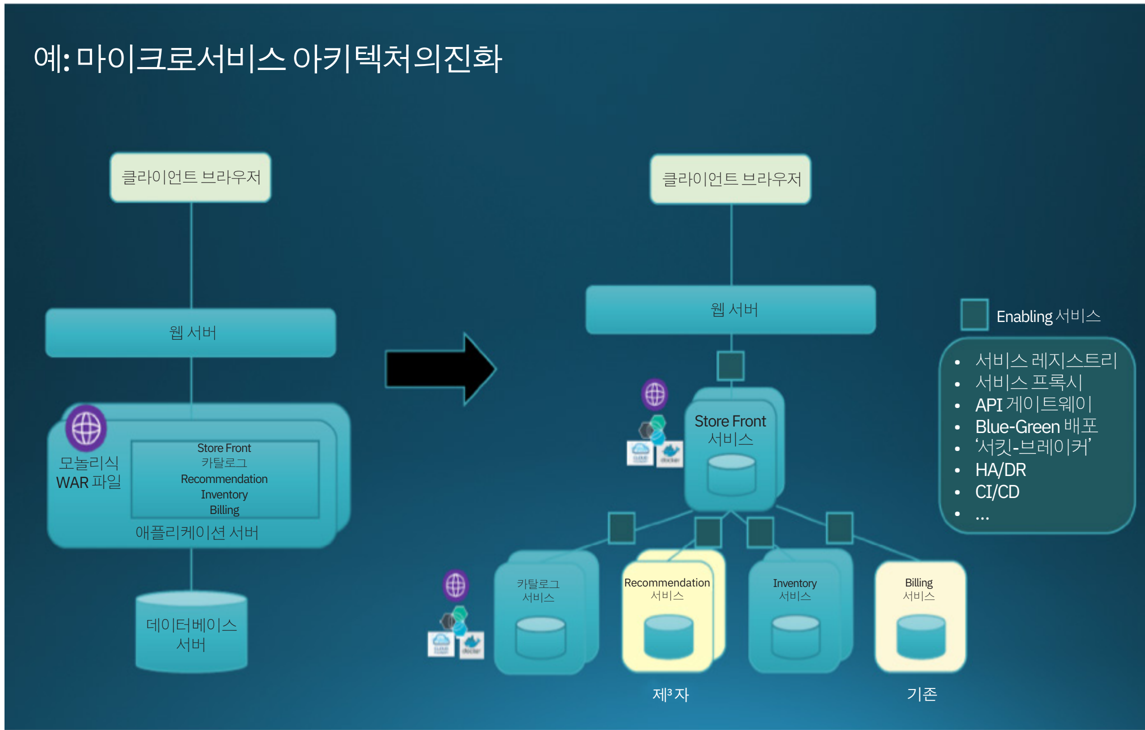


그림 3: 모놀리식 애플리케이션과 마이크로서비스 간의 차이점

그림 3에는 소매 웹 사이트의 모든 구성 요소를 호스팅하는 단일 엔터프라이즈 아카이브가 제시되어 있습니다. 애플리케이션 아카이브에는 카탈로그 및 인벤토리뿐만 아니라 웹 사이트 로직, 비즈니스 로직 및 각 구성 요소에 대한 지속성 로직와 같은 모든 비즈니스 기능이 포함되어 있습니다. 또한 애플리케이션은 종종 단단히 결합된 데이터 모델을 포함하고 다른 애플리케이션과 공유되는 단일 데이터베이스를 공유합니다.

이미지의 오른쪽에서 비즈니스 기능이 현재 자체 데이터를 통해 별도의 애플리케이션에 배포되었음을 알 수 있습니다. 이 새로운 스타일의 아키텍처는 마이크로서비스 커뮤니케이션, 데이터 소유권, 데이터 동기화 및 탄력성과 관련된 문제를 야기합니다.

James Lewis 및 Martin Flower가 작성한 마이크로서비스 아키텍처의 특징 장에는 마이크로서비스 기반 애플리케이션의 주요 측면이 논의되어 있습니다.

위 참고 문헌 예제를 사용하여 작성한 주요 측면에 대한 요약서에는 다음과 같은 내용이 포함되어 있습니다.

서비스를 통한 컴포넌트화: 이러한 측면은 본 백서의 처음 부분에 설명되어 있습니다.

비즈니스 기능을 중심으로 구성: 우리는 앞서 팀 개발 개념에 대해 논의했고, 비즈니스 기능을 중심으로 팀을 조직하기 위해서는 개발 팀에 대한 우리의 생각을 바꿀 필요가 있습니다.

다음과 같이 팀을 역할별로 구성하는 방법을 고려하십시오.

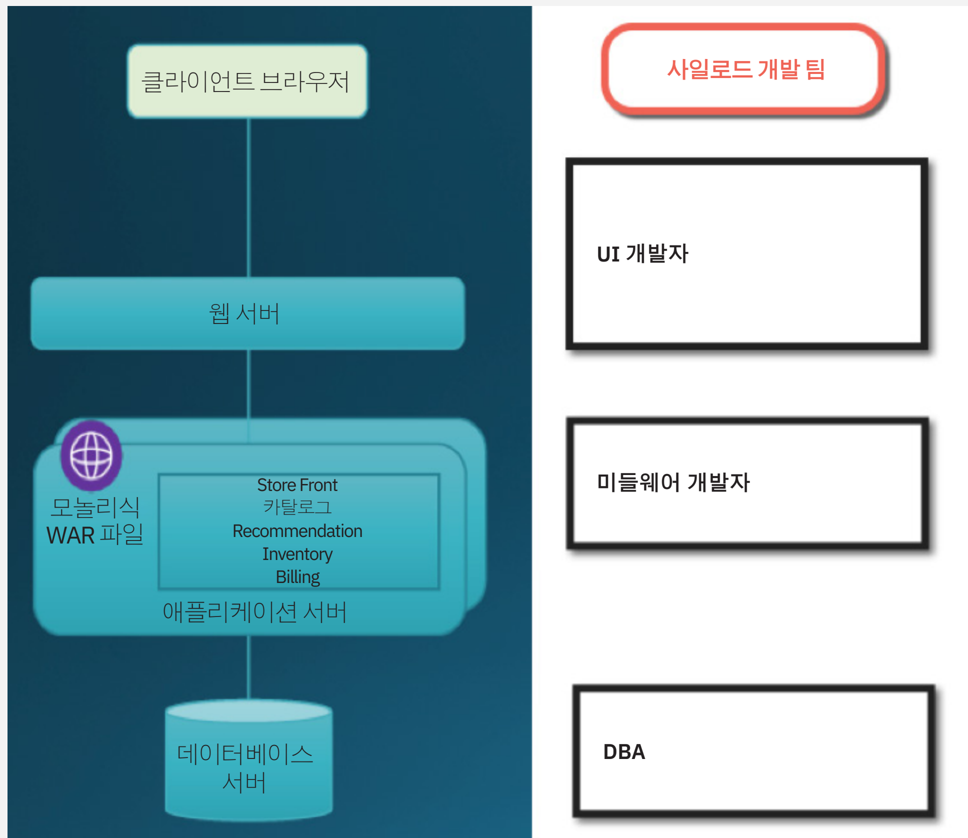


그림 4: 사일로드 개발 팀

이를 그림 5과 같이 비즈니스 기능을 중심으로 구성된 마이크로서비스 기반 팀 접근 방식과 비교해보십시오.

그림 5: 비즈니스 기능에 기반하여 구성된 개발 팀

마이크로서비스를 사용하여 비즈니스 문제 해결

대부분의 회사는 마이크로서비스 기반 접근 방식을 사용하여 새롭고 고유한 클라우드 애플리케이션을 구축합니다. 하지만 많은 기업들이 기존의 단일 애플리케이션을 보다 빠르게 이동할 수 있는 기능을 재설계해야 한다고 느끼고 있습니다.

모놀리식 애플리케이션 마이그레이션

그림 6에는 모놀리식 아키텍처에서 마이크로서비스 기반 아키텍처로 마이그레이션하는 예가 제시되어 있습니다. 이 사례에서, 소매 업체는 보다 빠르게 이동하고 고객에 대해 더 많은 것을 알며 결제와 같은 최신 기능을 도입하기 위해 마이크로서비스로 전환하기를 원합니다.

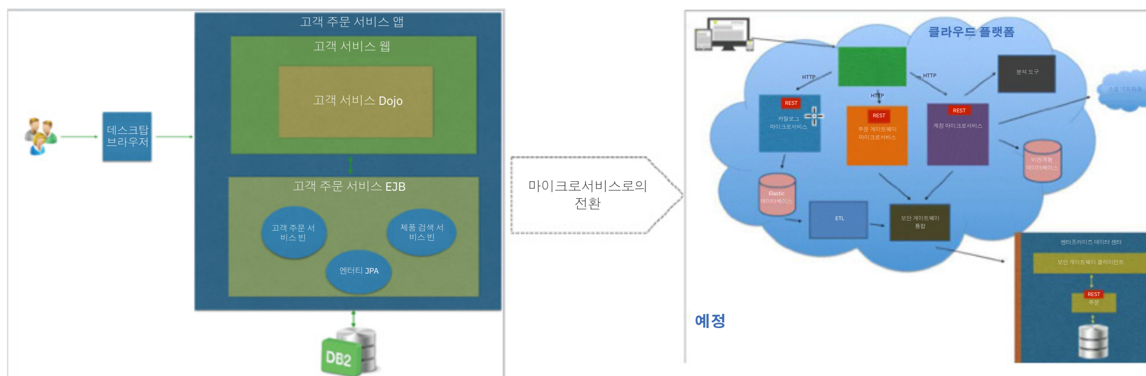


그림 6. 마이크로서비스 기반 아키텍처로 마이그레이션

위 시나리오에서, 소매 업체는 다음 단계에 따라 모놀리식에서 마이크로서비스 아키텍처로 전환했습니다.

1. **카탈로그 처리 방법 결정.** 소매 업체가 해결해야 할 첫 번째 비즈니스 문제는 제품 카탈로그를 어떻게 관리하느냐에 있었습니다. 고객은 제품 데이터를 찾을 수 없었으며 소매 업체는 다른 사이트에 이러한 정보를 노출할 수 없었습니다. 팀은 다음 단계에 따라 비즈니스 카탈로그를 위한 단일 마이크로서비스를 구축하기로 결정했습니다.

   - 일래스틱 서치 (Elastic Search)에 데이터를 가져와 데이터를 검색하고 새로운 데이터 패턴을 식별할 수 있는 새로운 방법을 제시했습니다.
   - 기존 웹 사이트를 새로운 검색 창에 연결했습니다.
   - 소매 업체는 이 파일럿을 통해 하나의 기반으로 새로운 CI/CD 모델을 도입했습니다.

팀은 경영진에게 마이크로서비스 모델로 전환하면 비즈니스를 확장할 수 있다고 확신시켜 주었습니다.

2. **고객에 대한 더 많은 정보 수집** - 팀은 고객에 대해 더 많은 것을 알기 위해 계정 마이크로서비스를 만들었습니다.

    - 우선, 팀 구성원들은 소비자 중심에서 인벤토리 중심 조직으로 비즈니스를 전환하는 방법을 알아 냈습니다. 팀 구성원들은 새로운 고객 모델을 고안하고 구조화되지 않은 데이터 모델을 제공한 Mongo DB 또는 Cloudant 같은 NOSQL 데이터베이스를 사용했습니다. 팀 구성원들은 시간이 지남에 따라 분석, 마케팅 및 코그니티브 데이터를 기반으로 고객 데이터가 풍부해질 것이라는 사실을 알게 되었습니다.
    - 기존 고객 데이터를 사용하여 주문을 추적했습니다.

3. **새로운 사용자 환경 조성** - 팀은 모바일 및 웹 플랫폼을 위한 새로운 프런트 엔드와 새로운 네이티브 모바일 앱을 만들었습니다. 최신 사용자 인터페이스와 카탈로그를 통해 최종 사용자를 위한 새로운 환경을 만들었습니다. 새로운 카탈로그는 기존 주문 로직과 통합되었으며 핵심 비즈니스를 구성하고 너무 복잡해 아직까지 해체되지 않았습니다.

4. **주문 마이크로서비스** - 팀은 모바일용 새 주문 API를 만들고 기존 트랜잭션과 통합하는 데 중점을 두었습니다. 팀은 기존 SOR(System of Record)을 호출하는 어댑터 마이크로서비스를 만들기로 결정했습니다. 팀은 이 어댑터 계층에서 새로운 최신 결제 시스템과 통합할 수 있는 기회를 얻었습니다.

5. **비즈니스 확장** - 소매 업체는 새로운 마이크로서비스 모델을 발전시켜 새로운 옥션 기능을 추가함으로써 비즈니스 모델을 확장시켰습니다.

마이크로서비스 아키텍처

이제 마이크로서비스 기반 아키텍처의 세부 사항에 대해 자세히 알아보도록 하겠습니다.

마이크로서비스 기능

애플리케이션 구성 요소는 클라우드에서 마이크로서비스로 실행되며 마이크로서비스 패브릭을 통해 서로 커뮤니케이션합니다. 서로 다른 유형의 애플리케이션은 각기 다른 패턴을 사용합니다. 나중에 웹/모바일 예제를 보여드리겠습니다. 그림 7에는 마이크로서비스 기반 아키텍처에 필요한기능이 제시되어 있습니다.

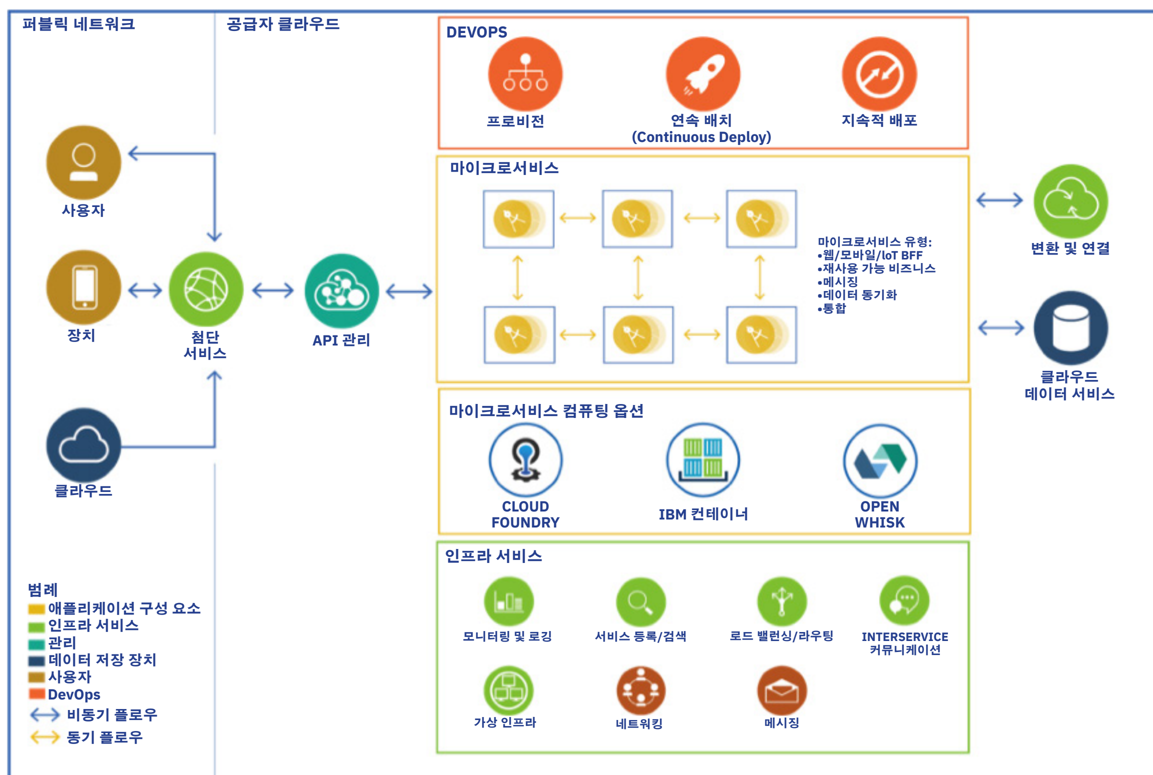


그림 7. 마이크로서비스 기반 아키텍처의 기능

다이어그램에서 왼쪽에서 오른쪽으로 읽은 후 다음 단계로 마이크로서비스 기능을 검토해 보십시오. 여러 마이크로서비스가 API를 통해 노출되며 이러한 마이크로서비스 중 일부가 *API 게이트웨이를 통해 사용됩니다.*

API 게이트웨이는 엔드포인트의 프록시 만큼 간편할 수 있습니다. 일부 API 게이트웨이는 훨씬 정교하며 처음 접촉한 순간 보안, 모니터링 및 API 버저닝 기능이 포함되어 있습니다. 또한 전체 API 관리 시스템은 제3자가 사용할 수 있는 개발자 포털이 포함되어 있습니다.

DevOps

DevOps를 사용해 강력한 자동화 전략을 개발하여 프로비저닝, 연속 배치 및 지속적인 배포를 포괄하는 성공적인 마이크로서비스 아키텍처를 만들어야 합니다. 전략에는 다음이 포함되어야 합니다.

- **프로비저닝** - 성공적인 마이크로서비스 아키텍처는 애플리케이션 환경을 위한 자동 프로비저닝이 필요합니다. PaaS(Platform as a Service) 계층은 일반적으로 IBM® 클라우드® 의 CaaS(Container as a Service)와 같은 관리형 서비스를 통해 이러한 서비스를제공합니다. Kubernetes 또는 DDC(Docker Datacenter)와 같은 Docker 오케스트레이션 시스템을 사용하여 IaaS(Infrastructure as a Service) 계층에서 직접 컨테이너 인프라를 실행하는 경우, VM 환경에서 자동 프로비저닝을 사용하여 해당 환경을 프로비저닝하고 및 업데이트하는 방법을 자동화해야 합니다.

- **연속 배치** - 개발 환경에서 Docker 이미지 또는 마이크로서비스 애플리케이션을 위한 자동화된 구축 및 배포 프로세스가 필요합니다. 여러 가지클라우드전략을 가지고 있는 경우, 구축 및 배포 자동화가 자동 확장 또는 클라우드 정책과 같은 작업의 차이를 추상화해야 합니다.

- **연속 배포** - DevOps 환경은 테스트 중심 개발 및 자동화된 테스트의 강력한 문화를 지원합니다. 여기에는 장치 테스트, 자동화된 기능 및 성능 테스트 환경 유효성 검사가 포함됩니다.

마이크로서비스 컴퓨팅 옵션

마이크로서비스를 실행하기 위한 다양한 컴퓨팅 옵션이 있습니다.

- **Docker 컨테이너** - 이 컨테이너는 클라우드 및 사내 환경에최고의 이식성을 제공합니다. Docker 컨테이너를 사용할 때 강력한 DevOps가 필요합니다.

- **Cloud Foundry** - 이 개방형 소스 PaaS는 마이크로서비스가 실행되고 서로 커뮤니케이션하는 방법 및 컨테이너와 같은 가상화를 위해 추상화를 제공합니다. Cloud Foundry는 어느 정도 수준의 이식성을 제공하지만 클라우드 환경에서 더 많은 스택을 실행해야 합니다.

- **기능** - 추상화와 사용 용이성 수준이 가장 높은 이벤트 기반 컴퓨팅 옵션. 개발자는 간단한 이벤트 처리 도구(Handler)를 배포하여 클라우드 구성 요소가 중앙 메시지 허브로 보내는 이벤트에 응답합니다. 모든 가상화가 추상화됩니다.

- **VM 또는 베어 메탈** - 마이크로서비스 기반 애플리케이션을 만들어 VM(Virtual Machine)에서 실행할 수 있습니다. 이를 위해서는 더 많은 프로비저닝과 DevOps가 성공을 거두어야 합니다. VM 및 베어 메탈은 사용 용이성에 가장 큰 유연성을 부여합니다.

그림 8에 이러한 옵션이 요약되어 있습니다.

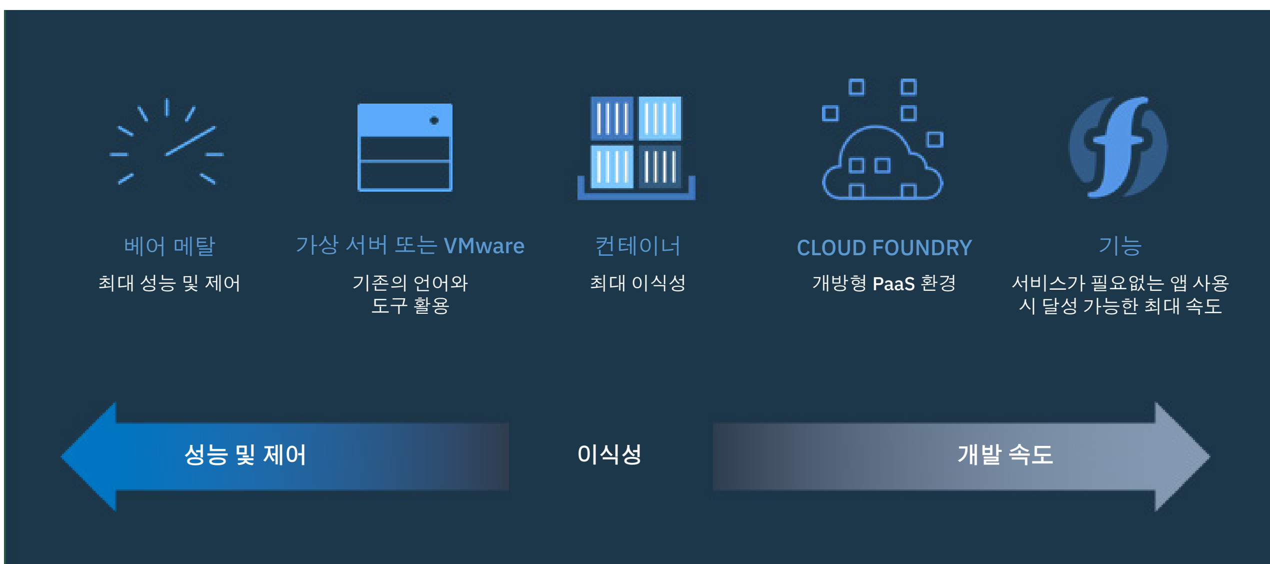


그림 8. 마이크로서비스 컴퓨팅 옵션

인프라 서비스

마이크로서비스를 둘러싼 클라우드 환경은 네트워킹, 메시징, 마이크로서비스 통신, 로깅 및 모니터링, 가상화, 서비스 검색 및 프록시, 복원 기능 등에 필요한 패브릭 및 서비스를 제공합니다.

애플리케이션 아키텍처

다음은 마이크로서비스 애플리케이션 아키텍처의 예입니다.

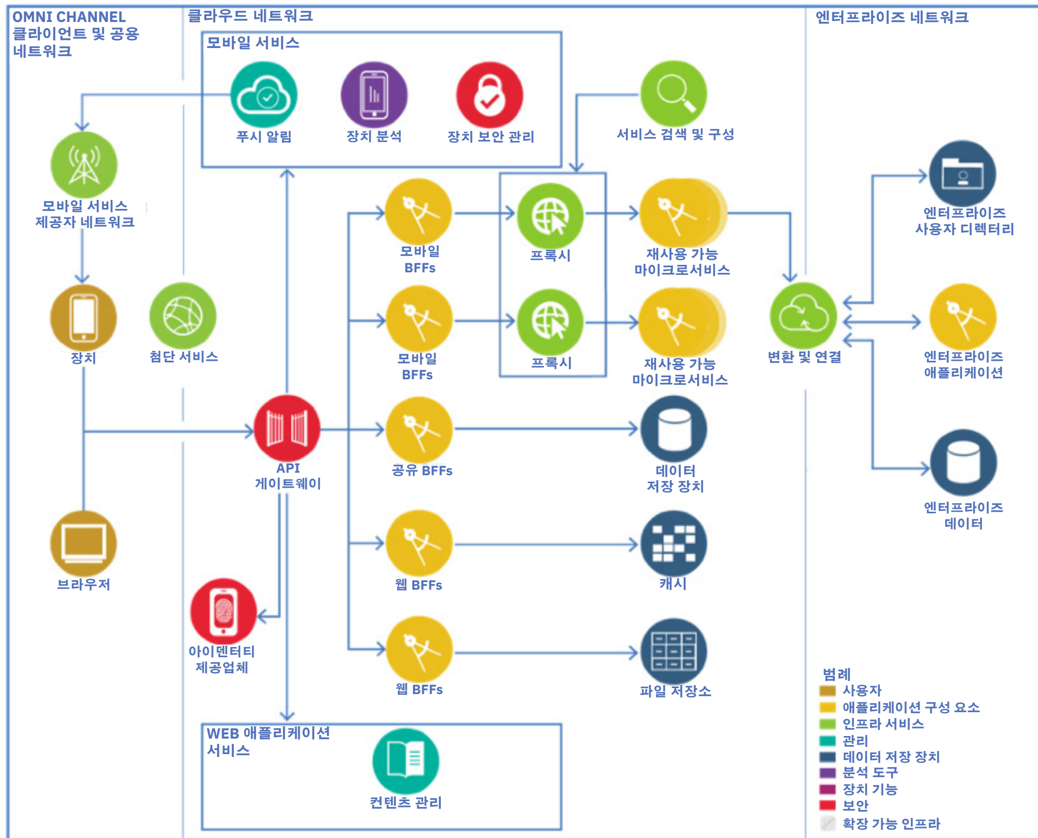


그림 9. 마이크로서비스 애플리케이션 아키텍처

이 아키텍처의 구성 요소를 요약하는 방법:

- 이 예제의 OmniChannel 애플리케이션에는 기본 iOS 애플리케이션 및 Angular기반 웹 애플리케이션이 포함됩니다. 이 다이어그램에는 이 애플리케이션의 장치와 브라우저가 묘사되어 있습니다.
- 모바일 애플리케이션은 IBM Mobile Analytics for Cloud 서비스를 사용하여 운영 및 비즈니스에 대한 장치 분석 결과를 수집합니다.

- 두 클라이언트 애플리케이션 모두 API 게이트웨이를 통해 API를 호출합니다. API 게이트웨이, IBM API Connect가 OAuth Provider를 제공하여 API 보안을 구현합니다.

- API는 BFF(Backend for Frontend)라 부르는 Node.js 마이크로서비스 패턴으로 구현됩니다. 가능한 다른 이름에 대한 예로는 xAPI(Experience API) 또는 Iteration API를 들 수 있습니다. 이 계층에서, 프런트 엔드 개발자는 일반적으로 프런트 엔드를 위해 백 엔드 로직을 작성합니다. Inventory BFF는 Express 프레임워크를 사용하여 구현됩니다. Social Review BFF는 API Connect LoopBack 프레임워크를 사용하여 구현됩니다. 이러한 마이크로서비스는 IBM Cloud에서 Cloud Foundry 애플리케이션으로 실행됩니다.

고객 사례 연구 읽기

기술 세부 정보 다운로드

- Node.JS BFF는 재사용 가능 Java 마이크로서비스의 또 다른 계층을 호출합니다. 실제 프로젝트에서는 다른 팀이 일반적으로 이를 작성합니다. 이러한 재사용 가능 마이크로서비스는 Spring Boot을 사용하여 Java에서 작성됩니다. 이러한 마이크로서비스는 Docker를 사용하여 IBM 컨테이너 내에서 실행됩니다.

- Node BFF와 Java 마이크로서비스는 마이크로서비스 패브릭을 사용하여 서로 커뮤니케이션합니다. 이에 대한 예로는 Istio 및 Netflix OSS를 들 수 있습니다.

- Java 마이크로서비스는 클라우드 데이터베이스 및 통합 계층과 상호 작용할 수 있습니다.

마이크로서비스 스택

마이크로서비스 스택을 지정하는 것이 중요합니다. 다음은 몇 가지 선택 사항이 있는 스택의 예입니다. 애플리케이션, 패브릭 및 DevOps를 비롯한 여러 계층을 지정해야 합니다. IBM이 이러한 마이크로서비스 구현 기술을 어떻게 판단하고 있는지 알아보려면 IBM Microservices Decision Guide를 참조하십시오.

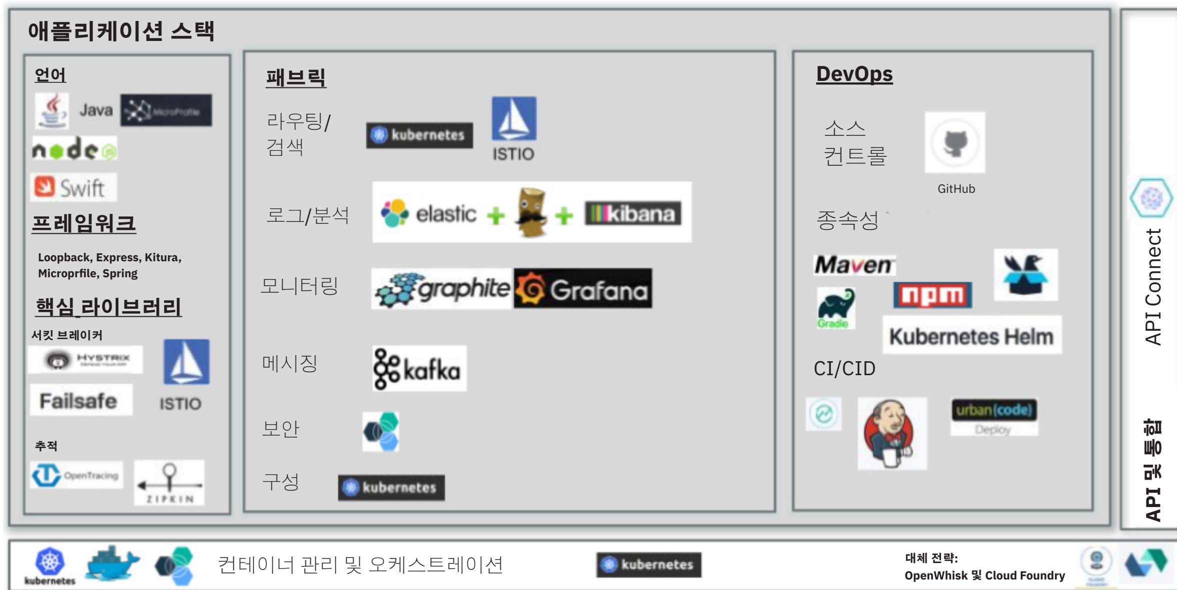


그림 10. 마이크로서비스 스택

애플리케이션

언어와 런타임 스택이 애플리케이션을 작성합니다. 예제에는 최신의 경량 Java 플랫폼, MicroProfile, WebSphere Liberty와 같은 엔터프라이즈 에디션(Java EE) 스택, Spring과 같은 대체 Java 스택 또는 StrongLoop와 같은 Node.js 스택이 포함됩니다.

특정 언어 라이브러리는 회로 차단 및 추적과 같은 마이크로서비스 애플리케이션 요소를 처리해야 합니다.

마이크로서비스 패브릭

마이크로서비스 패브릭은 다음과 같이 일련의 필요한 기능을 제공하는 기능 스택입니다.

- 라우팅 및 검색은 클라우드 인식, 마이크로서비스 간 커뮤니케이션을 위한 핵심 기능입니다. 마이크로서비스 인스턴스는 자동으로 조정되고 동적 클러스터가 포함될 수 있으며, 이름으로 마이크로서비스를 검색한 다음 실행 중인 인스턴스로 전달해야 합니다. Zuul/Eureka와 같은 프레임워크는 이를 Netflix OSS 스택의 일부로 제공합니다. Istio는 polyglot에 기반한 선호되는 옵션이며 서비스 호출 라우팅에 대한 더욱 세분화된(파인 그레인드) 제어 기능과 중요한 플랫폼에 대한 제어 기능을 제공합니다. 아래 그림에는 Istio 아키텍처의 예제가 제시되어 있습니다.

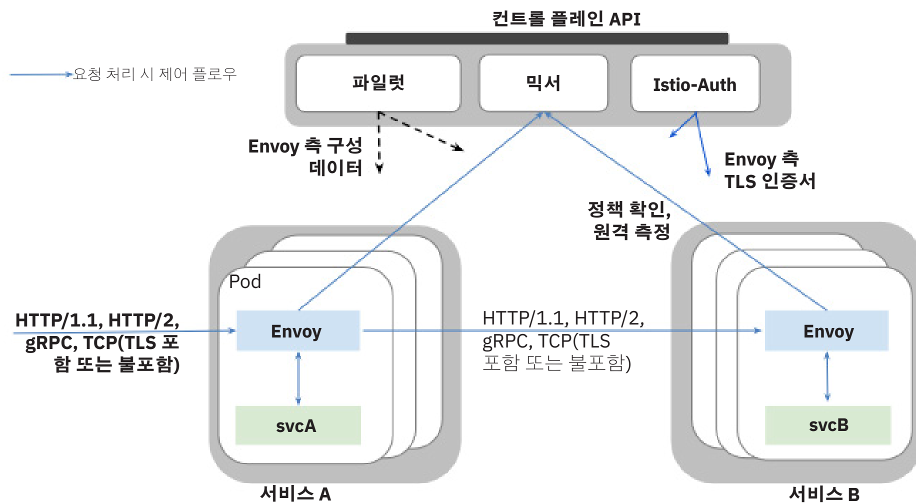


그림 11. Istio 아키텍처

- 로그 및 분석을 위해 스트리밍 로그를 캡처하여 여러 위치로 전달하는 스택이 필요합니다. 이를 위해 사용 가능한 몇 가지 프레임워크가 있습니다.

- 런타임, 애플리케이션 및 서킷 브레이커 정보를 비롯한 여러 데이터에 대한 대시보드 모니터링이 필요합니다. 이러한 도구는 로그 스택을 사용하여 통합된 관점을 제공합니다.

- 메시징 및 보안 계층도 중요합니다.

DevOps

프로비저닝, 오케스트레이션, 구축 및 배포 및 운영에 대한 요구 사항을 완료하려면 강력한 DevOps 도구 세트가 있어야 합니다.

컨테이너 관리

마이크로서비스 스택을 지원하려면 컨테이너 오케스트레이션 및 런타임 환경(예: IBM Cloud Container Service, Kubernetes 또는 Docker Data Center)이 필요합니다.

마이크로서비스 기반 아키텍처의 탄력성

이러한 세밀한 (fine-grained) 컴포넌트와 함께 마이크로서비스 아키텍처의 모든 탄력성 요점을 알아야 합니다. 여기에는 고가용성, 장애조치 (fail-over), DR, 회로 차단 및 격리가 포함됩니다.

그림 12에는 마이크로서비스 기반 아키텍처의 탄력성이 설명되어 있으며 글로벌 로드 밸런서 및 다중 사이트 리던던시의 사용 사례가 제시되어 있습니다.

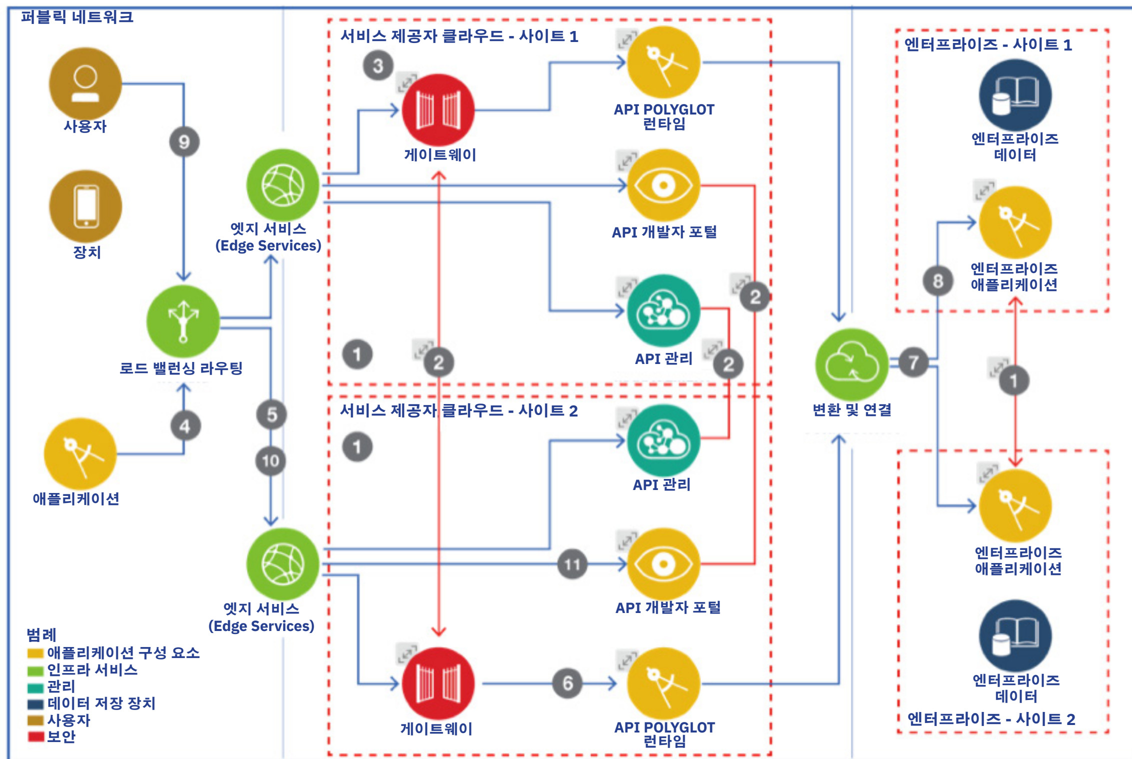


그림 12. 마이크로서비스 아키텍처의 탄력성

마이크로서비스에 대한 고가용성 및 장애 복구 패턴

탄력성(Resiliency) 고려 사항:

- 중복 데이터 다시 종료
- 복제 데이터 다시 종료
- 구역별로 여러 번 배포
- 여러 구역에 배포(데이터 센터 3개)
- 글로벌 로드 밸런싱

탄력성을 다룰 때는 고가용성(HA)과 장애 복구(DR)를 구분하는 것이 중요합니다.

HA는 시스템에서 업데이트 배포, 호스팅 VM 재부팅, 호스팅 OS에 보안 패치 적용 등의 유지 관리 활동이 수행될 때 최종 사용자에게 서비스가 제공되도록 보장합니다.

HA는 일반적으로 정전, 지진, 심각한 하드웨어 고장 또는 전체 사이트 연결 손실로 인한 전체 사이트 손실과 같이 예기치 않게 발생하는 중대한 문제를 의미하지 않습니다. 이러한 사례에서 서비스에 엄격한 SLO(Service Level Objective)가 있는 경우 최소한 두 가지 IBM Cloud 영역에 의존하여 전체 애플리케이션 스택(인프라, 서비스 및 애플리케이션 구성 요소)이 중복되도록 해야 합니다. 이는 일반적으로 장애 복구(DR) 아키텍처로 정의됩니다.

DR 솔루션을 구현하는 여러 가지 옵션이 있습니다. 간소화를 위해 서로 다른 옵션을 세 가지 주요 범주로 그룹화할 수 있습니다.

액티브/패시브, 액티브/대기 및 **액티브/액티브.**

액티브/패시브	액티브/대기	액티브/액티브
이 옵션을 사용하면 전체 애플리케이션 스택을 하나의 위치에서 활성 상태로 유지하고 다른 애플리케이션 스택은 다른 위치에 배포되지만 유휴 상태로 유지되거나 차단됩니다. 기본 사이트를 오랫동안 사용할 수 없는 경우 백업 스택에서 애플리케이션 스택이 활성화됩니다. 일반적으로 기본 사이트에서 입수한 백업 정보를 복원해야 합니다. RTO (복구 목표 시간)가 몇 시간 미만이므로 데이터 손실 문제가 있거나 서비스 가용성이 중요한 경우에는 이 접근 방식을 사용하지 않는 것이 좋습니다.	이 옵션을 사용하면 전체 애플리케이션 스택이 기본 및 백업 위치에서 활성화됩니다. 하지만 기본 사이트만 사용자의 트랜잭션을 처리합니다. 백업 사이트는 데이터베이스 복제 또는 디스크 복제와 같은 데이터 복제를 통해 기본 위치 상태에 대한 복제본을 저장합니다. 기본 사이트를 오랫동안 사용할 수 없는 경우 모든 클라이언트 트랜잭션이 백업 사이트로 전달됩니다. 이 접근 방식은 우수한 RPO (복구 지점 목표) 및 RTO(분)를 제공하지만 이중 배치로 인해 액티브/패시브보다 훨씬 비쌉니다. 대기 자산을 사용하여 확장성과 처리량을 개선할 수 없으므로 리소스가 낭비됩니다.	이 경우 두 위치가 모두 활성화되고 클라이언트 트랜잭션은 라운드 로빈, 로드 밸런싱 및 위치와 같이 미리 정의된 정책에 따라 두 영역에 모두 분산됩니다. 하나의 사이트가 실패하면 다른 사이트가 모든 클라이언트를 처리합니다. 이 구성을 사용하면 제로에 가까운 RPO 및 RTO를 달성할 수 있습니다. 단점은 두 위치 모두 사용할 수 있을 때 기능의 절반으로 사용되더라도 두 영역이 전체 로드를 처리할 수 있는 크기여야 한다는 데 있습니다. 이 경우에서 IBM Cloud 서비스를 위한 오토 스케일링이 BlueCompute 샘플 애플리케이션과 유사하게 요구 사항에 따라 리소스를 할당합니다.

확장성 및 성능 고려 사항

탄력성을 추가한다는 것은 일반적으로 중복 배포를 의미하며 성능 및 확장성을 개선하기 위해 탄력성을 사용할 수도 있습니다. 위 섹션에서 설명한 바와 같이 액티브/액티브 사례의 경우에도 해당합니다. 글로벌 애플리케이션의 경우 Akamai 또는 Dyn의 글로벌 라우팅 솔루션을 사용하여 사용자의 트랜잭션을 가장 가까운 위치로 리디렉션하여 응답 시간과 대기 시간을 개선할 수 있습니다.

마이크로서비스 프로젝트 구현

다음의 논리적 단계는 조직에서 마이크로서비스 기반 프로젝트를 구현하는 방법을 이해하는 것입니다.

처음부터 또는 기존 단일 시스템에서 마이크로서비스 시스템을 구축할 수 있습니다. 대부분의 IBM 고객은 기존 모노리스를 업데이트하기 위해 마이크로서비스 여정을 시작하므로 이 섹션에서는 기존 모놀리식 아키텍처에서 작업할 때 마이크로서비스를 평가하고 구현하는 방법에 중점을 둡니다.

이전의 애플리케이션 없이 마이크로서비스 프로젝트를 구축하는 것이 의미가 있고 중요하지만 마이크로서비스를 구축할 때는 모노리스 우선 접근 방식을 취하십시오. 간단히 말해, 이는 아이디어를 먼저 검증할 수 있는 방법으로 애플리케이션을 구축다는 의미입니다.

그런 다음, 이 백서의 원칙을 적용하여 초기 모노리스를 마이크로서비스 프로젝트로 확장하고 발전시킵니다. 비즈니스에 가치를 제공하지 않는 구조적으로 순수한 마이크로서비스를 만드는 것은 가치가 없습니다.

모노리스 우선 접근 방식을 사용하여 마이크로서비스 애플리케이션을 구현하는 우수한 직접 계정은 The Chronicles의 Game On! 팀이 문서화했습니다.

기존 애플리케이션 진화

마이크로서비스 기반 변환 작업 수행 시 이해해야 할 많은 개념이 있다는 사실을 알게 되었을 것입니다. 이 섹션에서는 비즈니스, 문화 및 기술 세트 그리고 기술과 같은 성공적인 마이크로서비스 프로젝트를 구현하기 위해 이해해야 하는 세 가지 영역에 대해 설명할 것입니다.

비즈니스 요구 사항 이해 및 정의

마이크로서비스로의 전환을 고려하고 있는 이유는 무엇입니까? 비즈니스에 더 빨리 가치를 부여하고 보다 나은 사용자 환경을 제공하기 위해 많은 사업 주체들은 더욱 효율적인 소프트웨어 개발 및 운영 방법이 필요합니다.

기존 애플리케이션 및 인프라에 대한 마이크로서비스 프로젝트의 영향을 이해하기 전에 비즈니스의 어느 부분이 너무 느리게 움직여 만족스러운 결과를 산출하지 못하는지 이해해야 합니다. 여러 가지 사례에서, 조직의 SOE(Systems of Engagement) 때문에 속도가 느려집니다. 이 시스템은 웹, 모바일 및 API를 비롯한 다양한 채널을 통해 사용할 수 있습니다. 속도의 부족은 마이크로서비스 기반 아키텍처로 전환하는 주된 이유입니다.

마이크로서비스 지향 접근 방식을 채택하기 전에 무엇이 시장 속도를 따라가지 못하는지 알아야 합니다. 먼저 가속화하기 위해 애플리케이션의 어느 부분을 개선하고 변경해야 하는지 파악해야 합니다. 거기서부터 기존 모노리스의 어느 부분을 마이크로서비스 진화의 대상으로 지정해야 하는지 정확히 알 수 있습니다.

이 단계에서 사용자 환경 플로우 또는 아키텍처 다이어그램과 같은 디자인 및 아키텍처 산출물을 사용하는 것이 중요합니다. 팀은 그림 13과 같이 모노리스의 섹션을 신속하게 식별하고 우선 순위를 지정할 수 있습니다.

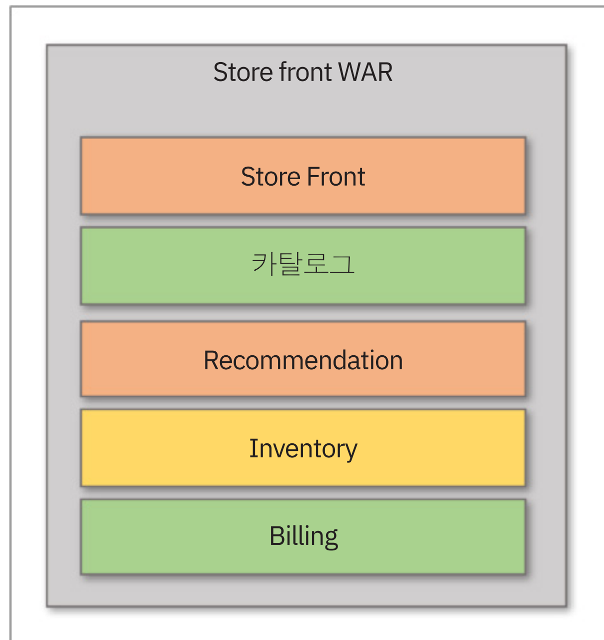


그림 13. 기존 모노리스의 고충, 우선순위 지정을 위해 Red-Yellow-Green 척도 사용

이러한 식별 프로세스는 철저히 수행할 필요는 없으며 사실상 반복적이어야 합니다. 주요 목표는 다음과 같습니다.

- 모노리스가 제공하는 별도의 비즈니스 기능 식별
- 비즈니스 기능을 변경하는 데 필요한 상대 속도와 복잡성 이해
- 특정 비즈니스 기능에 대한 피드백 주기를 단축하고자 하는 사업 주체의 욕구 이해

문화 및 기술 세트 이해

마이크로서비스 기반 아키텍처에만 국한되는 것은 아니지만 디지털 전환 과정에서 조직의 팀, 문화 및 기술 세트를 철저히 이해하는 것이 무엇보다 중요합니다.

일반적으로 엔지니어링 모노리스에서는 대부분의 조직이 사일로 팀에 포함되며 필요 시 소프트웨어 개발 수명주기에 따라 참여합니다. 이는 종종 잘 정의된 경계를 생겨나게 합니다. 이러한 경계에 따라 역할과 책임이 제한됩니다. 그림 14에는 일반적인 모놀리식 조직 구조가 제시되어 있습니다.

그림 14. 전통적인 IT 조직 구조

이에 비해 마이크로서비스 아키텍처는 팀이 완벽한 소프트웨어 개발 및 운영 수명주기를 책임질 능력이 있는 경우에만 성공할 수 있습니다. 팀이 전체 DevOps 수명주기를 소유하기 위해서는 역할과 책임이 다른 구성원이 필요합니다.

이러한 다기능 팀은 마이크로서비스 기반 아키텍처의 역량을 강화하기 위해 구성되었습니다. 사일로 팀 구성원 대신 모든 역할과 책임이 동일한 팀에 포함됩니다. 디자인에서 개발 및 운영에 이르는 모든 구성원들이 긴밀히 협력하며 종종 함께 배치됩니다. 실제 팀 구조는 본 백서의 범위를 벗어나지만 가상 팀 경계는 다기능적 방식으로 형성되었을 때 비즈니스를 혁신시키는 과정에서 가장 성공적이라 할 수 있습니다.

이를 통해 설계, 개발 및 운영 관계자가 팀에 모두 나타나므로 비즈니스가 설계 환경을 명확하게 파악하고 가능한 딜리버리 일정을 이해하며 운영 비용을 최소화할 수 있습니다. 그림 15에는 최적의 DevOps 팀 구성 방법이 제시되어 있습니다.

그림 15. 마이크로서비스 성공을 위해 최적화된 DevOps 팀

다기능 팀은 팀 전체에서 개인 기술의 빠른 성장을 지원합니다. 팀이 설계에서 운영, 런타임 데이터에 이르기까지 마이크로서비스가 담당하는 모든 것을 소유할 경우 단일 팀 구성원은 하나의 작업을 수행하도록 좌천되지 않습니다. 종종 프런트 엔드 엔지니어는 데이터베이스 관리 기술을 개발하고 운영 중심 팀 구성원은 사용자 인터페이스 프레임워크의 차이점에 대해 더 많이 배웁니다. 이러한 방식의 기술 확장은 마이크로서비스를 통해 전체 IT 조직을 성공적으로 운영할 수 있게 해줍니다. 매우 구체적인 역할을 수행하는 전문가를 찾는 것보다 우수한 팀 구성원으로 구성된 새로운 팀을 구성하는 것이 훨씬 쉽습니다.

비즈니스 문제 및 팀의 문화와 기술 세트 문제를 해결하지 않으면 마이크로서비스 기술을 효과적으로 구현할 수 없으므로 동일한 프로세스 및 구조를 일관되게 배치해야 합니다.

기술 이해

기존 기술 스택에 대한 적절한 분석은 조직마다 매우 다양하지만 간소화된 접근 방식은 마이크로서비스 프로젝트의 초기 및 지속적 성공을 보장하는 데 도움이 됩니다. 소규모로 시작하고 반복적이고 혁신적인 성공을 정의하는 것은 모든 것을 한꺼번에 접하는 방식보다 훨씬 더 달성 가능하고 유익한 접근 방식입니다.

그림 16. 마이크로서비스 진화에 대한 반복적 접근 방식

기술을 이해하는 첫 단계는 기존의 모노리스에 있는 코스 그레인드 (coarse-grained)서비스를 파악하는 데 있습니다. 도메인 중심 디자인 (DDD) 원칙에 맞춰 이를 정의할 수 있습니다. 이러한 원리를 사용하면 결코 구축되지 않은 기존 애플리케이션에 잘 정의된 디자인 원칙을 적용할 수 있습니다. 이러한 코스 그레인드 (coarse-grained)서비스를 파악하면 데이터 구조의 복잡성, 현재 구성 요소와 새로운 코스 그레인드 서비스를 담당하는 팀 간의 결합 수준을 이해하는 데 도움이 됩니다. 성공적인 검토를 통해 해당 서비스 내부 및 서비스 전반에 걸쳐 데이터 경계를 명확하게 이해할 수 있습니다.

코스 그레인드 서비스를 파악한 경우, 피일 그레인드 (fine-grained) 마이크로서비스로 발전시키는 방법에 대한 계획을 수립해야 합니다. 이전 작업에 기반한 이러한 마이크로서비스는 모두 유사한 데이터를 사용하고 자체 데이터를 관리하며 다른 서비스에 읽고 쓰는 데 필요한 데이터가 무엇인지 이해해야 합니다. 여기에서 개별적으로 파인 그레인드 마이크로서비스의 탄력성, 확장성 및 민첩성을 파악하고 구현할 수 있습니다.

앞서 언급한 바와 같이, API와 마이크로서비스는 일대일 비교 대상이 아니지만 더 큰 전체의 두 부분일 뿐입니다. 파인 그레인드 마이크로서비스를 더 잘 이해하면 중요한 인터페이스, 선택 사항인 인터페이스 및 더 이상 필요없는 인터페이스를 비롯해 사용자의 인터페이스를 더 잘 이해할 수 있습니다. 기존 인터페이스 또는 API를 코스그레인드 또는 파인그레인드 마이크로서비스 중 하나에 매핑할 수 없는 경우, 필요하지 않을 가능성이 높습니다.

마이크로서비스 작업 규모

모든 분석 및 계획 작업이 완료되면 일정, 전달 속도 및 예상 결과를 지정해야 합니다. 이는 매우 다양하지만 기존의 디지털 전환 프로젝트에서 완전히 새로운 프로세스가 아닐 수 있습니다.

비즈니스를 이해하고 팀 구조를 이해하며 기술을 이해하기 위한 노력은 입증된 개념 범위, 파일롯 범위 또는 대규모의 발전 범위인지 여부에 관계없이 조직이 해당 모노리스의 마이크로서비스 진화를 모두 이해할 준비를 마치도록 보장합니다.

마이크로서비스 패턴

마이크로서비스에 대한 개발 패턴

개발 과정에서, 패턴은 알려진 솔루션을 일반적인 유형의 요구 사항에 적용하는 데 유용하며 이는 마이크로서비스에도 적용됩니다. Martin Fowler의 마이크로서비스 디자인 원칙 중 하나는 마이크로서비스가 "비즈니스 기능을 중심으로 구성" 된다는 사실입니다.[2] 이러한 원칙은 단지 분배할 수 있기 때문에 무언가를 해야 한다는 것을 의미하지 않는다는 발견과 직접 관련이 있습니다.

이러한 패턴은 일반적으로 마이크로서비스에 사용됩니다.

- 파사드(Façade) **패턴은** 시스템 또는 서브시스템을 위한 특정한 외부 API를 정의합니다. 이 패턴의 하위 텍스트는 이 API가 비즈니스 중심이라는 것입니다.

- **엔터티 및 집계 패턴은** 비즈니스 인터페이스 측면에서 설계하는 데 익숙하지 않은 개발 팀을 위해 마이크로서비스에 직접 매핑되는 특정 비즈니스 개념을 파악하는 데 유용합니다.

- **서비스 패턴은** 단일 엔터티 또는 집계에 해당하지 않는 작업을 마이크로서비스에 필요한 엔터티 기반 접근 방식으로 매핑하는 수단을 제공합니다.

- **어댑터 마이크로서비스 패턴은** 여러 가지 경우에서 개발 팀이 패턴 데이터에 대한 분산 제어 권한을 가지고 있지 않는 기업 세계에서 유용합니다. 어댑터 마이크로서비스에서, 서로 다른 두 개의 API를 채택합니다. 하나의 API는 전통적인 마이크로서비스와 동일한 도메인 중심 기술과 함께 RESTful 또는 경량 메시징 기술을 사용하여 구축된 비즈니스 지향 API입니다. 두 번째 API는 레거시 API 또는 전통적인 WS-* 기반 SOAP 서비스입니다.

- **스트랭글러 애플리케이션 패턴은** 비즈니스 및 애플리케이션이 실제로 녹색 필드 환경에 실제로 존재하지 않는다는 사실을 나타냅니다. 마이크로서비스의 혜택을 누릴 수 있는 프로그램은 리팩터링 가능한 대형 모놀리식 애플리케이션입니다. 이 패턴은 리팩터링을 관리하기 위한 접근 방식을 제공합니다. 스트랭글러 애플리케이션 패턴은 이 백서의 뒷부분에 자세히 설명되어 있습니다.

마이크로서비스에 대한 작업 패턴

마이크로서비스는 단일 서비스를 변경하고 배포하는 속도를 가속화하지만 상응하는 모놀리식 애플리케이션과 비교할 때 일련의 서비스를 관리하고 유지 관리하는 데 더 많은 노력이 필요합니다. 본래부터 기존 애플리케이션 관리를 위해 개발된 마이크로서비스에 대한 이러한 운영 패턴은 DevOps의 운영 측면에 적용됩니다.

- **서비스 레지스트리 패턴을** 사용하면 다운스트림 마이크로서비스의 구현을 변경할 수 있으며 DevOps 파이프라인의 여러 단계에서 다양하게 서비스 위치를 선택할 수 있습니다. 특정 마이크로서비스 엔드 포인트가 사용자 코드로 하드 코딩되지 않도록 방지하여 이를 달성할 수 있습니다. 서비스 레지스트리가 없는 경우, 마이크로서비스 콜 체인을 통해 코드 변경 사항이 전파되기 시작할 때 사용자 애플리케이션이 빠르게 머뭇거립니다.
- **상관 ID 및 로그 수집기 패턴은** 더 나은 분리를 달성하는 동시에 마이크로서비스를 보다 쉽게 디버깅할 수 있습니다. 상관 ID 패턴은 서로 다른 여러 언어로 작성된 다수의 마이크로서비스를 통해 추적 전파를 허용합니다. 로그 수집기 패턴은 다수의 서로 다른 마이크로서비스의 로그 정보를 검색 가능한 단일 저장소로 집계함으로써 상관 ID를 보완합니다. 이러한 패턴을 함께 사용하면 각 호출 스택의 서비스 수 또는 깊이에 관계없이 마이크로서비스의 효율적이고 이해하기 쉬운 디버깅이 가능합니다.
- **서킷 브레이커 패턴은** 이미 발생했음을 알게 될 경우 다운 스트림 오류를 처리하는 데 시간을 낭비하지 않도록 도와줍니다. 이렇게 하려면 다운 스트림 서비스가 오작동할 때 감지하고 이를 호출하려는 시도를 방지하는 업스트림 서비스에 코드의 *서킷 브레이커* 섹션을 위치시켜야 합니다. 이 접근 방식의 이점은 각 호출이 빠르게 실패한다는 데 있습니다. 다운스트림 호출이 실패할 것이라는 사실을 알게될 때 사용자에게 더 나은 전반적인 환경을 제공하고 스레드 및 연결 풀과 같은 리소스를 잘못 관리하는 것을 방지할 수 있습니다.

스트랭글러 패턴에 집중

파울러의 스트랭글러 패턴은 감싸고 있는 나무를 죽게 만드는 덩굴나무 비유에 근거합니다. 이 개념은 웹 애플리케이션(비즈니스 도메인의 다른 측면에 기능적으로 매핑된 개별 URI(Uniform Resource Identifier)와 별도로 구축)구조를 사용하여 애플리케이션을 다른 기능 도메인으로 분할하고 새로운 마이크로서비스 기반 구현 도메인으로 대체(한 번에 하나의 도메인 대체)하는 데 있습니다. 이 두 가지 측면은 같은 URI 공간에 나란히 상주하는 별도의 애플리케이션을 구성합니다. 시간이 지남에 따라 새롭게 리팩터링된 애플리케이션이 최종적으로 모놀리식 애플리케이션을 차단할 때까지 본래의 애플리케이션을 제거하거나 대체합니다.

스트랭글러 패턴은 변형, 공존 및 제거 단계를 거칩니다.

- **변형** - 예를 들어, IBM Cloud 또는 기존 환경에서 병렬로 새 사이트를 생성하지만 보다 최근의 접근 방식에 기반합니다.
- **공존** - 한동안 기존 사이트를 그대로 남겨둡니다. 새로 구현된 기능을 위해 기존 사이트에서 새 사이트로 점진적으로 리디렉션합니다.
- **제거** - 트래픽이 사이트의 해당 부분에서 리다이렉션될 때 기존 사이트에서 이전 기능을 제거하거나 기존 사이트 유지 관리를 단순히 중지합니다.

이 패턴을 적용하는 이점 중 하나는 이 패턴이 사용자가 하나의 큰 마이그레이션 시 모든 것을 수행하려고 시도했을 때보다 일정을 훨씬 더 단축시킨다는 점입니다. 또한 이 패턴은 마이크로서비스 채택을 위한 점진적 접근 방식을 제공합니다. 사용자 환경에서 이러한 접근 방식이 효과적이지 않을 경우, 방향을 변경하는 간단한 방법이 있습니다.

스트랭글러 애플리케이션 패턴은 언제 작동하며 언제 작동하지 않습니까?

- **웹 또는 API 기반 모노리스 -** 기존 웹 또는 API 기반 모노리스에서 시작하는 것이 이 패턴을 성공적으로 적용하기 위한 첫 번째 요구 사항입니다. 스트랭글러 패턴의 목적은 새 기능과 이전 기능 간에 쉽게 앞뒤로 이동하는 방법을 제공하는 데 있습니다. 애플리케이션이 웹 애플리케이션인 경우, URL 구조는 시스템을 어떻게 구현하고 시스템의 어떤 부분을 구현할지 선택할 수 있는 프레임워크를 제공합니다. 하지만 REST로 변환하는 SOAP API 세트 또는 메시징 시스템에 구현된 큐 세트와 같이 고정된 API 세트에 기반하는 모든 애플리케이션을 통해 이 패턴을 적용할 수 있습니다. 반면에 두꺼운 클라이언트 애플리케이션이나 수많은 기본 모바일 애플리케이션은 애플리케이션을 쉽게 분리할 수 있는 구조가 없으므로 이 접근 방식에 적합하지 않습니다.

- **표준화된 URL 구조(실제 URL 사용) -** 웹 애플리케이션은 모두 웹 구조(예: HTTP 및 HTML)에 의해 부여된 표준에 따라 작동하지만 다양한 애플리케이션 아키텍처를 사용하여 웹 애플리케이션을 구현할 수 있습니다. 이 접근 방식은 애플리케이션을 분리하려는 시도가 복잡해질 수 있는 여지가 많이 있습니다. 예를 들어, 포털 접근 방식과 같이 서버 요청 아래에 중간 계층이 있는 경우, URL을 사용하여 애플리케이션을 분리할 때 문제가 발생할 수 있습니다. 전환 및 라우팅에 대한 결정은 브라우저 수준에서 내려지지 않지만 훨씬 복잡한 애플리케이션에서 더 심층적으로 이루어집니다.

- **메타 UI -** UI가 비즈니스 프로세스에 기반하거나 즉석에서 구축되면 UI 및 비즈니스 로직용 코드를 서로 다른 마이크로서비스로 분리하는 것이 어려워집니다. 이 접근 방식은 여전히 작동하지만 청크 크기(아래 참조)가 더 커지므로 모든 것을 한 번에 구현해야 합니다.

스트랭글러 애플리케이션 패턴을 적용하지 않는 방법

스트랭글러 애플리케이션 패턴은 완전한 대안이 될 수 없습니다. 모든 경우 또는 특히 다음과 같은 경우에서 이 패턴을 적용하고 싶지 않을 것입니다.

- 한 번에 한 페이지씩 적용하지 마십시오. 가장 작은 "조각" (아래의 배포 관리 섹션 참조)은 단일 마이크로서비스입니다. 마이크로서비스는 완전하고 자급 자족할 필요가 있으며 더 중요한 것은 관리하는 데이터를 완전히 소유해야한다는 것입니다. 일관성 문제를 방지하기 위해 한 번에 데이터에 대해 서로 다른 두 가지 데이터 접근 방법을 취하지 않기를 바랄 것입니다.

- 패턴을 동시에 모두 적용하지 마십시오. 이렇게 하면, 스트랭글러 패턴을 실제로 적용하지 않고 Big Bang 접근 방식으로 되돌아갈 수 있습니다.

스트랭글러 애플리케이션 패턴을 적용하는 최선의 방법

따라서 단일 페이지가 너무 작고 전체 애플리케이션이 너무 클 경우, 스트랭글러 애플리케이션을 적용하기에 적합한 세분성 수준은 무엇입니까? 성공하기 위해서는 애플리케이션 리팩터링의 두 가지 측면을 혼합해야 합니다.

- 마이크로서비스 디자인에 백 엔드 리팩터링(**내부**)

- 마이크로서비스를 수용하고 리팩터링을 구동하는 새로운 기능을 변경하기 위해 프런트 엔드 리팩터링(**외부**)

내부부터 시작:

1. 먼저 애플리케이션 디자인에서 제한된 컨텍스트를 확인하여 시작합니다. 제한된 컨텍스트는 Eric Evans의 Domain-Driven Design에 제시되어 있는 또 다른 패턴입니다. 이 책에 따르면, 제한된 컨텍스트는 "모델이 적용되는 컨텍스트를 정의합니다."[3] 제한된 컨텍스트는 더 큰 비즈니스 모델 세트 내의 여러 엔터티의 의미를 제한하는 공유된 개념적 프레임워크입니다. 항공 부문에서 항공편 예약은 제한된 컨텍스트인 반면, 항공사 로열티 프로그램은 하나의 또 다른 제한된 컨텍스트입니다. 항공 업계에서 "항공편" 과 같은 용어를 공유할 수 있지만 이러한 용어의 사용 및 정의 방법은 상당히 많이 다릅니다.

2. 가장 작고 비용이 가장 적게 드는 제한된 컨텍스트를 선택하여 리팩터링합니다. 복잡성이 가장 낮은 것부터 복잡한 것 순으로 다른 제한된 컨텍스트의 순위를 매깁니다. 보다 복잡하고 잠재적으로 값 비싼 리팩터링 작업을 수행하기 전에 리팩터링의 가치를 입증하고 프로세스 채택 시 발생하는 문제를 해결하기 위해 가장 복잡한 제한된 컨텍스트부터 시작합니다.

3. 위에 설명된 바와 같이 *도메인 중심 디자인에서* 엔터티, 집계 및 서비스 패턴을 적용하여 컨텍스트 내에서 마이크로서비스를 개념적으로 계획합니다. 이 시점에서, 단지 마이크로서비스가 존재할 가능성이 있다는 것을 이해하려고 시도하기 때문에 다음 단계에서는 근사치를 사용할 수 있습니다.

그런 다음, 외부로 이동합니다.

1. 기존 UI에서 화면 간의 관계를 분석합니다. 특히, 여러 화면을 서로 밀접하게 연결하는 대규모 플로우를 검색합니다. 항공사 웹 사이트를 구축하는 경우, 하나의 플로우가 항공권을 예약하는 것일 수 있으며 이 웹 사이트는 예약 프로세스를 완료하는 데 필요한 정보를 제공하는 여러 개의 관련 화면으로 구성됩니다. 항공사의 로열티 프로그램에 가입할 때 다른 플로우가 중심이 될 수 있습니다. 플로우 세트를 이해하면 다음 리팩터링 단계로 전환하는 데 도움이 됩니다.

2. 기존 UI 또는 새 UI를 검사할 때 내부에서 식별된 마이크로서비스에 해당하는 측면을 검색합니다. 어떤 플로우가 어떤 마이크로서비스와 일치하는지 확인합니다. 이 단계에서 페이지의 한쪽 면에 있는 플로우 목록과 다른 면에 있는 각 플로우를 구현할 수 있는 마이크로서비스 목록이 출력됩니다.

3. UI 변경 사항에 일관성이 있어야 한다는 가정 하에 청크의 크기를 조정합니다. 하나 이상의 플로우가 고객에게 공개될 수 있는 최소 변경 크기라고 가정하지만, 청크 자체는 하나의 플로우보다 클 수 있습니다. 예를 들어, 고객 로열티가 모두 두 개 또는 세 개의 개별 플로우로 구성될 수 있더라도 하나의 청크로 간주할 수 있습니다.

4. 한 번에 전체 청크를 해제할지 또는 일련의 조각으로 청크를 해제할지 선택합니다.

참고 문헌

- IBM Cloud Garage Method, Microservices Architecture Center
- End-to-End Reference Implementation (GitHub)
- Refactoring to Microservices
- Microservices Patterns IO
- Exploring Microservices: Game-On (GitBook)

[1, 2]"Martin Fowler, *Microservices a definition of this new architectural term,* *https://martinfowler.com/articles/microservices.html*

[3]Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003*

Fin.