Micro Service

비지니스 민첩성

- 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Amazon 과 Netflix, Uber를 비롯해 성공한 Unicorn 기업들의 공통점 특징은 이미 익숙한 비즈니스에 새로운 비즈니스 개념과 기술을 융합해 자신만의 특화된 서비스를 제공한다는 것
  - ✓ 자신만의 특화된 서비스를 제공하려는 시도를 누구보다 빨리 실행했고 사용자 피드백을 반영해 끊임없이 서비스를 개선
  - ✓ Unicorn 기업들의 가장 큰 장점은 비즈니스 민첩성(Agility)이며 이것이 기업 성공의 가장 큰 요인
  - ✓ 인터넷의 발전과 모바일 환경의 대중화로 비즈니스 민첩성에 대한 중요성은 항상 강조돼 왔고 비즈니스 민첩성을 지원하기 위한 시스템 측면의 많은 투자와 시도가 이뤄져 왔으나 성공 사례가 많지 않았지만 이처럼 지지부진했던 노력이 Cloud 환경의 등장과 이를 잘 활용한 Amazon, Netflix 같은 기업의 사례에서 증명돼 실질적인 비즈니스 성과로 나타나고 있음

비지니스 민첩성

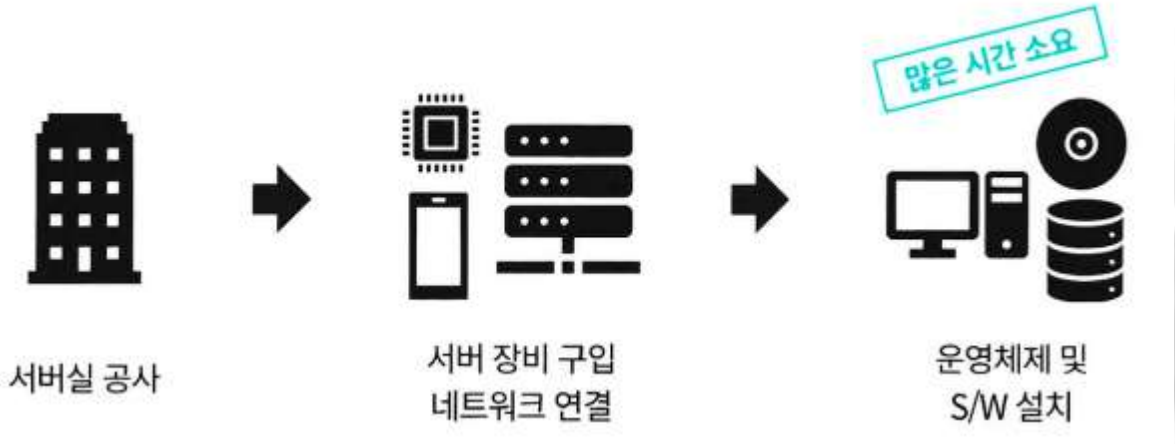
- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ 아마존의 배포 속도
    - 11.6초 - 2011년 Velocity라는 해외 콘퍼런스 에서 언급된 숫자로 11.6초는 100미터 달리기 기록으로 치기에도 매우 빠른 속도인데 이 수치는 아마존의 비즈니스 서비스가 배포되는 주기로 11.6초마다 아마존 쇼핑몰의 소스 코드가 변경되어 배포된다는 의미
    - 2019년에 국내 AWS 컨퍼런스에서 발표된 내용 - 아마존 서비스의 배포 주기가 초당 1.5번으로 향상

비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ 아마존의 배포 속도
    - 비즈니스는 계속 변경되므로 개선된 시스템도 계속 배포돼야 하는데 기업은 새로운 아이디어를 시스템에 반영해 적시에 오픈하고 그 반응을 살펴보고자 하고 기존 서비스를 보완하는 부가 서비스가 필요한 시점이나 새로운 서비스를 공개했지만 반응이 좋지 않아 이를 개선해야 할 때도 시스템을 빠르게 변경해서 배포해야 하므로 빠른 배포 주기는 비즈니스의 민첩성을 간접적으로 보여주는 지표라 할 수 있음
    - 보통 서비스는 기획, 분석, 설계, 구현 과정을 거쳐 빌드 후 배포되는데 그 과정이 길게는 몇 개월 걸리고 규모가 작은 수정이라 하더라도 몇 주가 걸리기 마련인데 아마존 쇼핑몰은 이 같은 전체 과정이 독립적으로 완료되어 초당 1.5번씩 서비스가 변경, 개선되고 있는데 비유하면 인간의 몸에서 낡은 세포가 죽고 새로운 세포가 생겨나는 것처럼 시스템이 살아 숨 쉬는 것 과 같음
    - 국내 한 쇼핑몰의 시스템 배포 주기가 1주라면 중간에 긴급 배포가 필요하면 주중에 1번 더 배포하는데 그럼 가장 빠른 배포 주기를 3일이라 볼 수 있는데 비즈니스 개선 주기가 3일인 것
    - 초당 1.5번이면 0.66초에 1번인데 아마존의 서비스는 0.66초마다 진화하고 국내 쇼핑몰은 3일마다 진화하는 셈
    - 3일마다 새로운 상품을 전시하고 매장 환경을 개선하는 상점과 0.66초마다 변화하는 상점 과 도 같은데 두 상점이 경쟁하면 어떤 회사가 우위에 있을지 답은 자명

비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라의 등장
    - 전형적인 시스템 인프라 구축 과정을 살펴보면 서버를 도입하고 네트워크를 구축한 뒤 각 서버마다 운영체제를 설치하고 서비스에 필요한 소프트웨어를 설치하는 과정으로 진행되고 전 과정을 완료하기까지 적게는 며칠에서 몇 주, 길게는 몇 달이 걸리기도 함
    - 애플리케이션을 개발하기 위한 인프라 환경을 준비하는 작업은 간단하지 않음
    - 어떤 기업에서는 아예 이 작업만 전담하는 인프라 조직을 두기도 하는데 이러한 활동을 개발 조직이 직접 담당하지 않고 인프라 조직에 요청한다면 의사소통 시간이 더해지므로 더 오랜 시간이 걸리기 마련

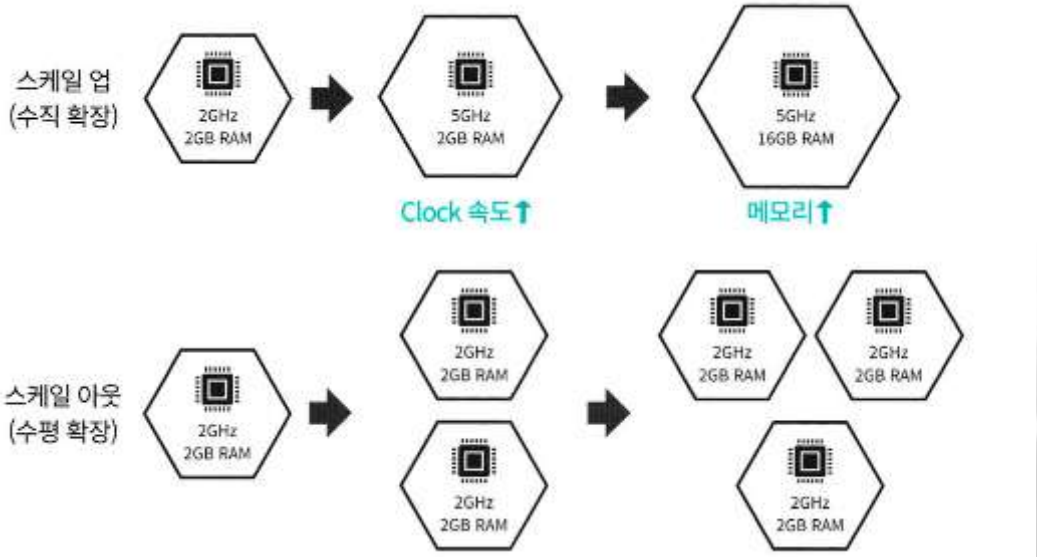


비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라의 등장
    - 새로운 서비스 개발을 위한 프로젝트를 시작한다고 생각해 보면 당연히 서버나 스토리지 같은 하드웨어, 네트워크, 운영체제 등을 위한 초기 투자 비용을 고려해야 하고 그것들을 지속적으로 관리하는 비용 또한 적지 않은데 이러한 환경에서는 좋은 아이디어가 있고 직접 애플리케이션을 개발할 능력이 있더라도 빨리 서비스를 런칭해 볼 수가 없으며 운 좋게 아이디어에 대한 투자를 받고 어찌 어찌 해서 어렵게 개발 환경을 구축한 뒤 서비스를 개발해 런칭했지만 서비스가 실패로 끝났다면 초기 투입된 인프라 비용을 건질 수 없음
    - 최근에는 위와 같은 문제가 Cloud 인프라의 등장으로 해결되었는데 서비스 개발에 필요한 시스템 인프라를 준비하는 데 오랜 시간이 들지 않음
    - 적당한 Cloud 플랫폼 벤더를 선택해 필요한 시점에 몇 번의 클릭으로 손쉽게 개발 시스템 인프라를 마련할 수 있음
    - 아마존은 자사의 풍부한 Cloud 서비스 경험을 아마존 웹서비스(AWS: Amazon Web Service)라는 사업으로 제공하고 마이크로소프트사의 애저(Azure), 구글의 구글 Cloud 플랫폼(Google Cloud Platform)0! 있고 최근에는 오라클까지 Cloud 플랫폼 사업을 벌이고 있음
    - 수많은 스타트업이 생겨나고 참신한 서비스를 빠르게 선보이는 것을 보면 이러한 Cloud 인프라가 얼마나 강력하게 비즈니스 개발의 민첩성을 촉진하는지 알 수 있음

비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라에 어울리는 애플리케이션의 조건
    - ● 스케일 업 과 스케일 아웃
      - ◆ 사용량 증가에 따른 성능 및 가용성을 높이는 방법으로 스케일 업(Scale- up)과 스케일 아웃(Scale-out)이 있음



비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라에 어울리는 애플리케이션의 조건
    - ● 스케일 업 과 스케일 아웃
      - ◆ 쇼핑몰 시스템 운영자는 타임세일 기간에 밀려올 트래픽에 대비하려고 쇼핑몰 시스템의 용량을 증설하려 하는 경우
        - ▪ 첫 번째 단계는 스케일 업 작업으로 전체 시스템의 트래픽 최대치를 계산해서 대용량 처리가 가능하도록 시스템 용량을 증설하는 것인데 그럼에도 예상 트래픽을 초과해 시스템이 다운될 수 있음
        - ▪ 두 번째 단계는 확장 탄력성을 보장하는 스케일 아웃을 설정하는 것으로 이 경우 CPU 나 메모리의 트래픽 양이 한계 수치에 도달하면 시스템의 인스턴스를 설정된 개수로 복제해서 증가시키는데 CPU 사용량이 70% 이상으로 증가하면 1개였던 쇼핑몰 인스턴스가 2개로 늘어나도록 하면 사용량이 2대의 인스턴스로 적절히 분산됨

비지니스 민첩성

- 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라에 어울리는 애플리케이션의 조건
    - 특정 서비스만 탄력성 있게 확장(스케일 아웃)
      - 시스템 운영자 입장에서 생각해 보면 더 좋은 방법이 있는데 1주일간의 세일 기간 중 정작 바쁜 업무는 세일 이벤트를 수행하는 부분인데 나머지 부분은 사용자가 몰리지 않아 더 한가해질 수도 있는데 이 모든 서비스의 시스템 용량을 증설하고 사용량이 몰리면 이 모든 것을 복제할 필요가 있을까? 당연히 낭비
      - 세일 이벤트를 담당하는 조각만 용량이 증설되고 사용량에 따라 복제되어 트래픽에 대비하면 되는데 이를 위해서는 전체가 한 덩어리로 묶여 있는 애플리케이션을 레고 블록처럼 분리하면 됨
      - 레고 블록과 같다면 세일 기간에 타임세일 이벤트를 수행하는 레고 모듈만 열심히 일할 수 있도록 용량을 증설하고 트래픽에 반응해 복제되게 설정하면 됨
      - 시스템을 작은 단위의 독립적인 서비스 연계로 구성

비지니스 민첩성

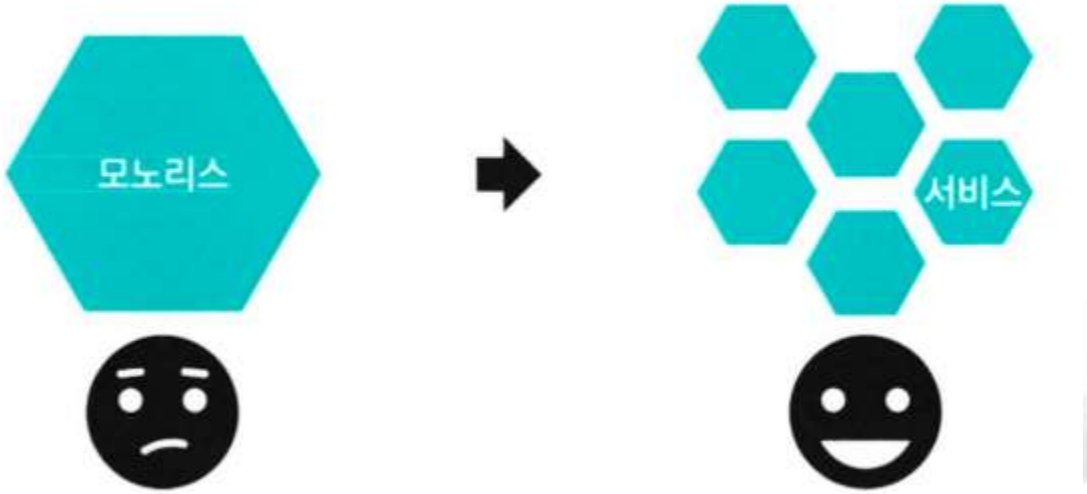
- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라에 어울리는 애플리케이션의 조건
    - ● 특정 서비스만 탄력성 있게 확장(스케일 아웃)
      - ◆ 시스템 운영자의 시나리오
        - ▪ 첫 번째 단계에서 운영자는 타임세일 서비스만 분리돼 있으므로 타임세일 서비스의 용량만 고려해서 증설(스케일 업)
        - ▪ 두 번째 단계에서는 독립된 타임세일 서비스의 사용량을 고려해서 트래픽이 증가하면 타임세일 서비스 인스턴스만 복제되도록 설정(스케일 아웃)
        - ▪ 전체 시스템이 사용할 메모리 크기가 16GB였다면 타임세일 서비스에 필요한 용량은 3~ 4GB 정도에 불과한데 이전 시나리오에서는 전체 시스템에 할당된 16GB의 메모리 크기를 32GB로 증설한 후 32GB가 여러 개로 복제된 비용을 지불했는데 이제는 타임세일 서비스에 필요한 3GB를 8GB로 증설하고 8GB가 여러 개로 복제된 비용만 지불
        - ▪ 조각으로 세일 서비스만 분리돼 있기 때문에 나머지 서비스는 가동된 상태에서 변경하고 싶은 일부분 만 변경해 배포할 수도 있음

비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라에 어울리는 애플리케이션의 조건
    - ● Cloud Friendly 와 Cloud Native
      - ◆ 작은 단위의 서비스 연계로 시스템을 구성하지 않고 전체 시스템을 하나의 덩어리로 만들어 Cloud 인프라에 올려도 비즈니스를 제공하는 데 전혀 문제가 없지만 특정 기능만 확장하거나 배포할 수 없는 비효율을 감수해야 함
      - ◆ Cloud 플랫폼 중 하나인 Cloud Foundry를 서비스하는 Pivotal에서는 이처럼 큰 덩어리로 Cloud 환경에 올라갈 수 있게만 한 애플리케이션을 Cloud 친화 애플리케이션(Cloud Friendly Application)이라 하고 독립적으로 분리되어 배포될 수 있는 조각으로 구성된 애플리케이션을 Cloud 인프라에 가장 어울리고 효과적이라는 의미로 Cloud Native 애플리케이션(Cloud Native Application)이라 부르고 궁극적으로 Cloud Friendly에서 Cloud Native로 전이해야 한다고 함
      - ◆ 시스템이 비즈니스 민첩성을 잘 지원하기 위해서는 Cloud 인프라를 효율적으로 활용하도록 애플리케이션 조각으로 구성된 Cloud Native 애플리케이션이 가장 효과적인데 아마존의 사례가 그것을 증명하며 비즈니스 개선 속도인 0.66초를 만들어 낸 것

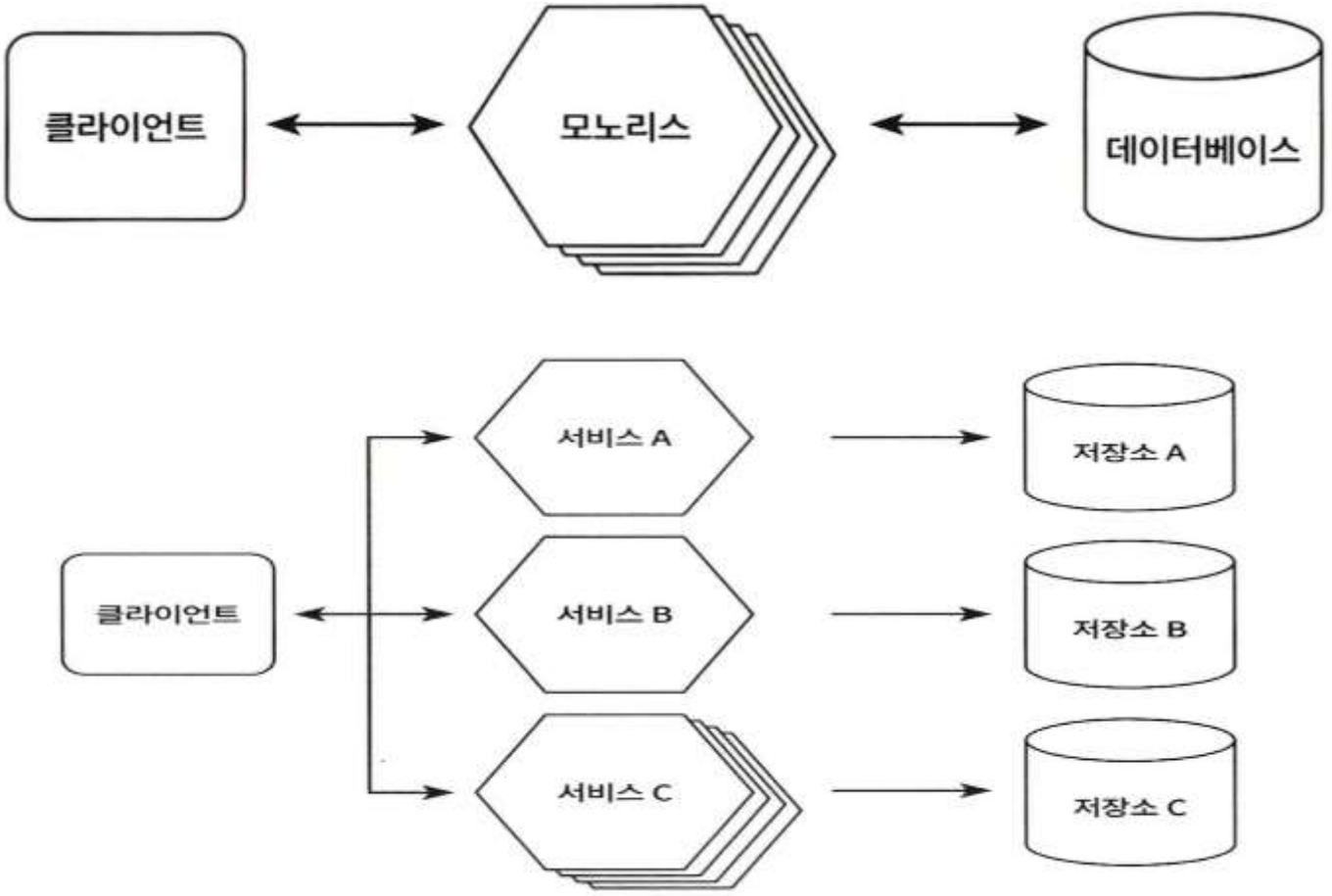
비지니스 민첩성

- ❖ 성공한 인터넷 기업들과 비지니스 민첩성
  - ✓ Cloud 인프라에 어울리는 애플리케이션의 조건
    - ● Cloud Friendly 와 Cloud Native



비지니스 민첩성

- Micro Service
  - ✓ Monolithic 와 Micro Service



비지니스 민첩성

- ❖ Micro Service
  - ✓ Monolithic 와 Micro Service
    - ● Monolithic
      - ◆ Monolithic는 하나의 단위로 개발되는 일체식 애플리케이션
      - ◆ 보통 3 티어라 불리는 사용자 인터페이스와 데이터베이스 그리고 서버 쪽 애플리케이션의 3개 부분으로 구성
      - ◆ 서버 측 애플리케이션이 일체 즉 논리적인 단일체로서 아무리 작은 변화에도 새로운 버전으로 전체를 빌드해서 배포해야 하고 일체식 애플리케이션은 단일 프로세스에서 실행되기 때문에 확장이 필요할 경우 특정 기능만 확장할 수 없고 반드시 전체 애플리케이션을 동시에 확장해야 하는데 보통 로드 밸런서를 앞에 두고 여러 인스턴스 위에 큰 덩어리를 복제해 수평으로 확장
      - ◆ 변경이 발생하면 Monolithic 시스템의 단점이 극대화되는데 여러 개의 Monolithic 시스템이 수평으로 확장된 상태이므로 여러 개의 Monolithic 시스템 모두를 전부 다시 빌드하고 배포해야 하고 확장 시 애플리케이션이 병렬로 확장되어 사용량 증가에 대응할 수 있지만 데이터베이스는 통합되어 하나이므로 탄력적으로 대응할 수 없기 때문에 사전에 성능을 감당하기 위해 스케일 업을 통해 용량을 증설해야 함

비지니스 민첩성

- ❖ Micro Service
  - ✓ Monolithic 와 Micro Service
    - ● Micro Service
      - ◆ Micro Service는 서버 측이 여러 개의 조각으로 구성돼 각 서비스가 별개의 인스턴스로 로딩되는데 여러 서비스 인스턴스가 모여 하나의 비즈니스 애플리케이션을 구성하고 각기 저장소가 다르므로 업무 단위로 모듈 경계를 명확하게 구분
      - ◆ 확장 시에는 특정 기능별로 독립적으로 확장할 수 있고 특정 서비스를 변경할 필요가 있다면 해당 서비스만 빌드해서 배포하면 됨
      - ◆ 각 서비스가 독립적이어서 서로 다른 언어로 개발하는 것도 가능하므로 각 서비스의 소유권을 분리해 서로 다른 팀이 개발 및 운영할 수 있음

비지니스 민첩성

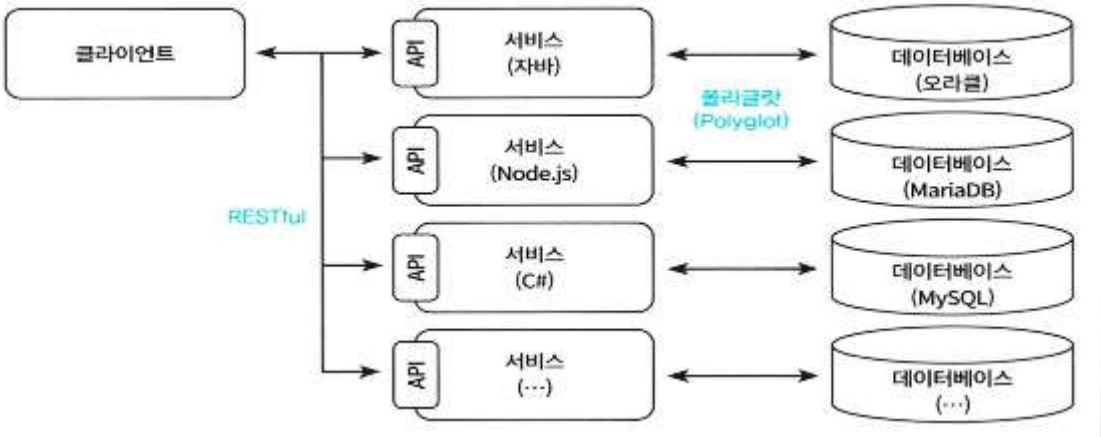
- ❖ Micro Service
  - ✓ SOA 와 Micro Service
    - 소프트웨어 공학에서 말하는 모듈화(modularity) 개념의 발전 흐름을 보면 단순히 기능을 하향식 분해해서 설계해 나가는 구조적(structured) 방법론부터 시작해 객체 단위로 모듈화하기 위한 객체지향(object-oriented) 방법론, 모듈화의 단위가 기능별로 재사용할 수 있는 좀 더 큰 컴포넌트가 되는 CBD(Component Based Development), 그리고 컴포넌트를 모아 비즈니스적으로 의미있고 완결적인 서비스 단위로 모듈화하는 SOA(Service Oriented Architecture)로 이어지는 발전 과정을 거침
    - CBD와 SOA도 넓게 보면 단위 컴포넌트나 서비스를 구성해서 시스템을 만드는 개념이고 Micro Service 시스템의 구조와 매우 유사
    - Micro Service 기반으로 시스템을 개발하는 아키텍처 및 개발 방식을 Micro Service 아키텍처(MSA; Microservice Architecture) 함
    - 넓게 보면 여러 개의 응집된 비즈니스 서비스의 집합으로 시스템을 개발한다는 점에서 SOA와 MSA는 개념적으로는 큰 차이가 없지만 SOA는 구체적이지 않고 이론적이며 실제 비즈니스 성공 사례가 많지 않았지만 MSA는 Cloud 인프라 기술의 발전과 접목되어 아마존 과 넷플릭스에 의해 구체화되고 비즈니스 성공 사례로 널리 공유된 바 있는데 이상적이었지만 성공을 증명하지 못했던 SOA가 Cloud 인프라의 등장으로 하드웨어를 유연하게 다룰 수 있게 되면서 비로소 실현되어 성공적으로 증명된 시스템 구조가 MSA

비지니스 민첩성

- Micro Service
  - ✓ SOA 와 Micro Service
    - Amazon 과 Netflix로 일컬어지는 인터넷 Unicorn의 성공 과정을 계속 지켜본 마틴 파울러는 2014년에 이러한 여러 성공 사례의 특징을 뽑아 Micro Service를 다음과 같이 정의
      - Micro Service는 여러 개의 작은 서비스 집합으로 개발하는 접근 방법
      - 각 서비스는 개별 프로세스에서 실행되고 HTTP 자원 API 같은 가벼운 수단을 사용해서 통신
      - 서비스는 비즈니스 기능 단위로 구성되고 자동화된 배포 방식을 이용하여 독립적으로 배포
      - 서비스에 대한 중앙 집중적인 관리는 최소화하고 각 서비스는 서로 다른 언어와 데이터 저장 기술을 사용할 수 있음

비지니스 민첩성

- ❖ Micro Service
  - ✓ SOA 와 Micro Service
    - ● 마틴 파울러가 정의한 Micro Service 개념을 표현한 개념도



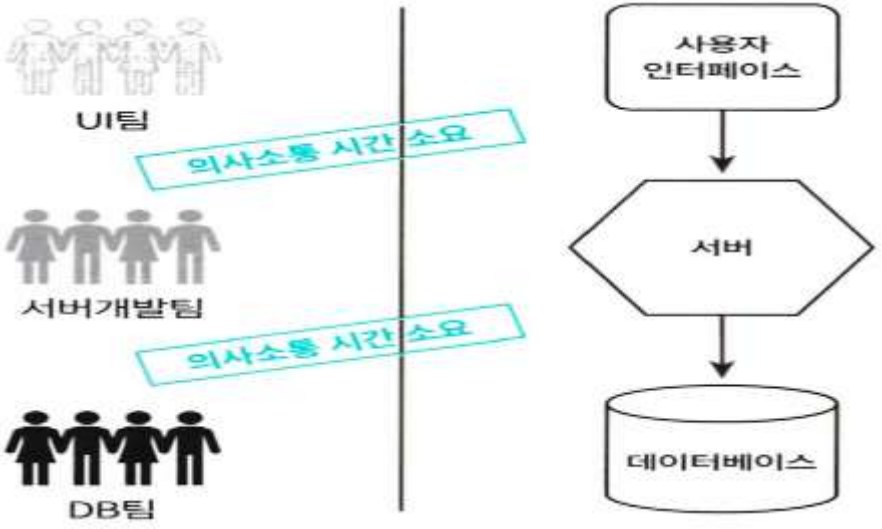
- ◆ 각 서비스와 저장소는 다른 서비스 및 저장소와 격리돼 있으며 API를 통해서만 느슨하게 연계되기 때문에 독립적으로 확장 가능하고 하나의 서비스만 독립적으로 배포 가능하고 다른 서비스와 연계된 API에 영향을 주지 않는다면 내부의 언어나 저장소는 자율적으로 선택할 수 있음
- ◆ 한 서비스에는 자바 와 오라클을 다른 서비스에는 Node.js 와 Maria DB를 선택한 것을 볼 수 있음

비지니스 민첩성

- Micro Service
  - ✓ SOA 와 Micro Service
    - 특정 서비스를 구축하는 데 사용되는 언어나 저장소를 자율적으로 선택할 수 있는 방식을 가리켜 Polyglot 하다로 표현
    - Cloud 등의 가상 인프라가 지닌 유연성이 이를 가능하도록 지원
    - Polyglot 저장소 같은 특성을 통해 CBD/SOA가 추구했지만 미흡했던 모듈화를 비로소 실현할 수 있었던 방식이 MSA
    - CBD/SOA의 접근법에서는 애플리케이션은 모듈 별로 분리했으나 데이터 저장소까지는 분리하지 못함(여러 애플리케이션이 하나의 저장소를 통합해서 공유)
    - 데이터의 강한 결합으로 애플리케이션도 독립적으로 사용하기가 힘들었지만 MSA에서는 SOA에는 없었던 다음의 두 가지 개념으로 모듈화 방식을 강화했고 이를 진정으로 실현
      - ◆ 서비스 별 저장소를 분리해서 다른 서비스가 저장소를 직접 호출하지 못하도록 캡슐화 하는데 다른 서비스의 저장소에 접근하는 수단은 API밖에 없음
      - ◆ REST API 같은 가벼운 개방형 표준을 사용해 각 서비스가 느슨하게 연계되고 누구나 쉽게 사용할 수 있음

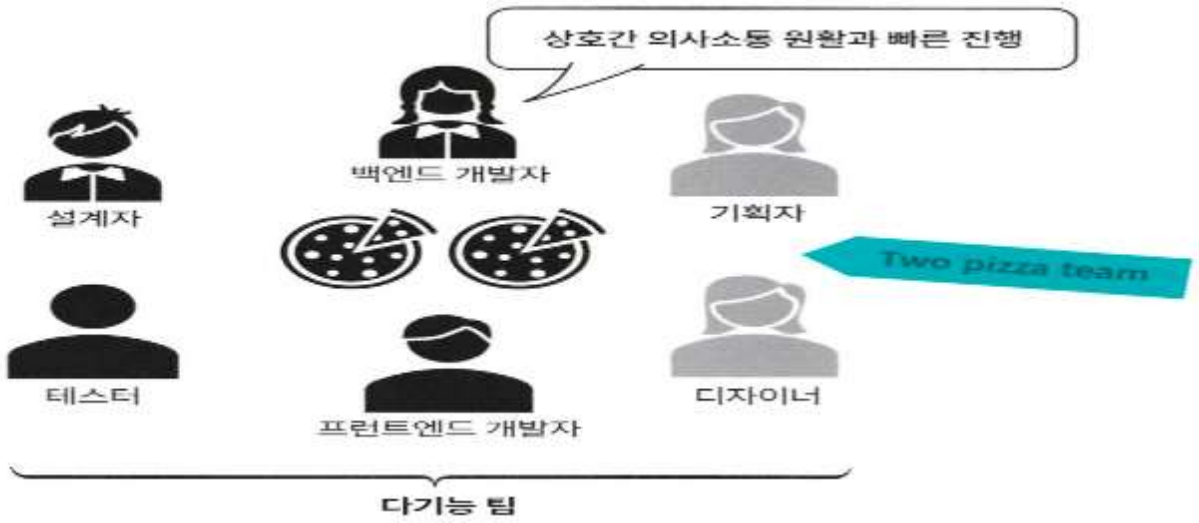
비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 조직의 변화: 업무 기능 중심 팀
    - 콘웨이 법칙(Conway's law)은 멜빈 콘웨이(Melvin E. Conway)가 정의한 조직 과 조직이 개발한 소프트웨어의 관계를 정의한 법칙으로 시스템을 개발할 때 항상 시스템의 모양이 팀의 의사 소통 구조를 반영하도록 하는 것
    - 예전의 일하는 방식을 보면 하나의 애플리케이션을 만드는 데 UI팀, 서버 개발팀, DB 팀과 같은 기술 별로 팀이 나눠져 있고 하나의 애플리케이션을 만드는 데는 세 팀 간의 의사 소통이 필요하기 때문에 시스템도 이러한 의사소통 구조를 그대로 반영하고 이러한 팀 구조에서는 팀 간의 의사결정도 느리고 의사 소통도 어려움



비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 조직의 변화: 업무 기능 중심 팀
    - Micro Service를 만드는 팀은 업무 기능 중심의 팀 이어야 하는데 업무 기능 중심 팀은 역할 또는 기술 별로 팀이 분리되는 것이 아니라 업무 기능을 중심으로 기술이 다양한 사람들이 하나의 팀이 되어 서비스를 만드는 것을 의미
    - 업무 기능 중심 팀은 다양한 역할(기획자, 디자이너, Front End 개발자, Back End 개발자. 설계자, 테스터 등)로 구성되고 이 팀은 서비스를 처음부터 끝까지 만들기 위한 모든 단계의 역할을 모두 갖추고 있음
    - 이 팀은 같은 공간, 같은 시간을 공유하기 때문에 의사 소통도 원활하고 의사 결정도 빠르게 진행될 수 있음

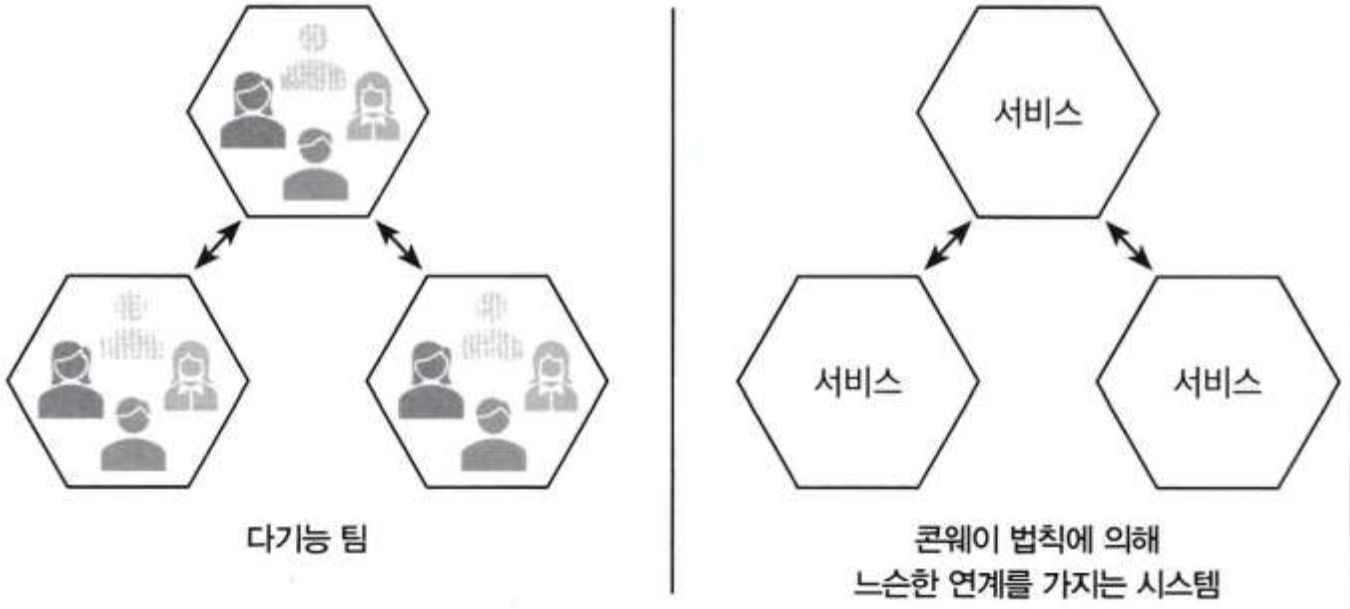


비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 조직의 변화: 업무 기능 중심 팀
    - 이 팀을 여러 기능들이 모여 있다는 의미에서 다기능 팀(Cross-Functional Team)이라고도 함
    - 이 팀은 자율적으로 담당 비즈니스에 관련된 서비스를 만들 뿐 만 아니라 개발 이후에 운영할 책임까지 짐
    - 아마존에서는 이런 팀을 two pizza team 이라고 표현하는데 이는 피자 두 판으로 서로 빈번히 의사소통하며 함께 식사할 수 있는 정도의 팀원 수를 의미하며 한 팀에 다양한 역할을 수행하는 사람들이 모여 있으며 개발과 운영을 동시에 수행하는 팀
    - Micro Service를 만드는 데 필요한 기능과 기술을 팀 내부에 모두 가지고 있으므로 다른 Micro Service 팀과는 협력할 일이 적을 수밖에 없기 때문에 콘웨이 법칙에 의해 이 팀이 만든 Micro Service도 다른 팀이 만든 Micro Service와는 느슨하게 연계

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 조직의 변화: 업무 기능 중심 팀

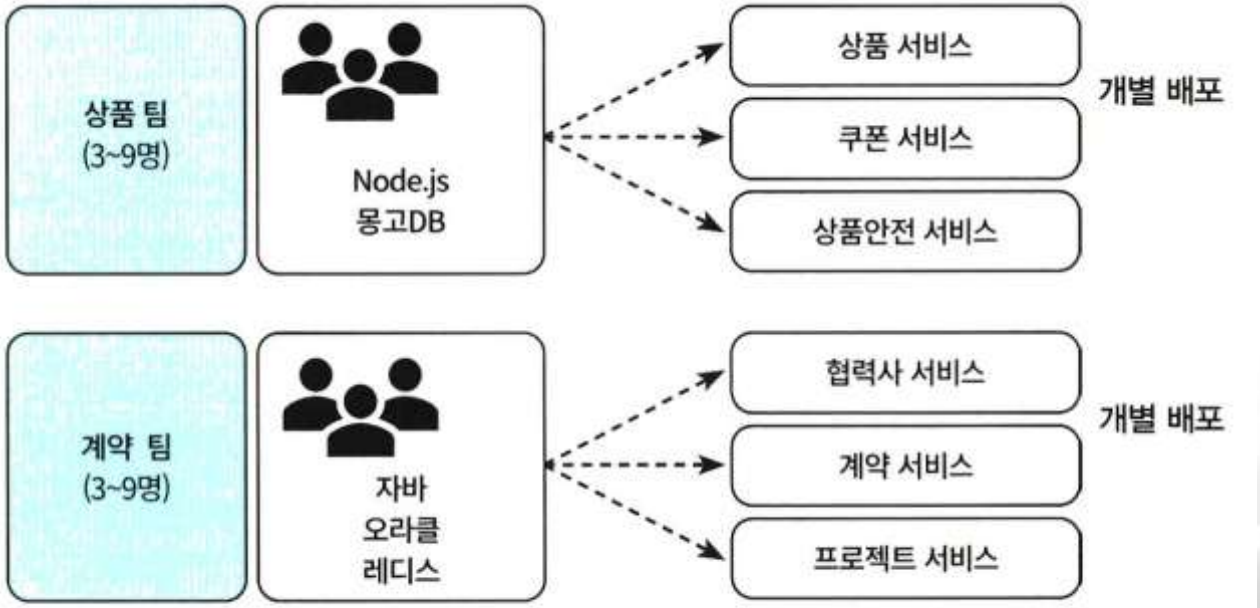


비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 관리 체계의 변화: 자율적인 분권 거버넌스, 폴리글랏
    - 아마존에서는 다기능 팀이 개발과 운영을 책임진다고 했는데 이러한 조직문화를 가리켜 you built it, you run it 이라는 모토로 표현하는데 우리가 만들고 우리가 직접 운영한다 라는 의미
    - Micro Service를 만드는 조직은 중앙의 강력한 거버넌스를 추구하지 않기 때문에 Micro Service팀으로 구성된 조직은 중앙의 강력한 표준이나 절차 준수를 강요하지 않음
    - 각 Micro Service팀은 빠르게 서비스를 만드는 것을 최우선 목적으로 두고 스스로 효율적인 방법론과 도구, 기술을 찾아 적용
    - 현실 세계에 비유하자면 작은 정부나 지방 자치 제도와 유사
    - 각 Micro Service팀이 자신의 서비스 성격에 맞는 최적의 언어와 저장소를 자율적으로 선택
    - 상품팀은 상품 검색 서비스를 위해 빠른 검색에 특화된 NoSQL인 몽고디비(MongoDB)와 Node.js를 선택했고 계약팀은 계약 서비스를 위해 자바 언어와 오라클, 레디스(Redis)를 선택하는데 각 서비스 팀이 팀에 맞는 개발 언어 및 저장소를 선택하는 것을 각각 폴리글랏 프로그래밍, 폴리글랏 저장소라고 함

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 관리 체계의 변화: 자율적인 분권 거버넌스, 폴리글랏



비지니스 민첩성

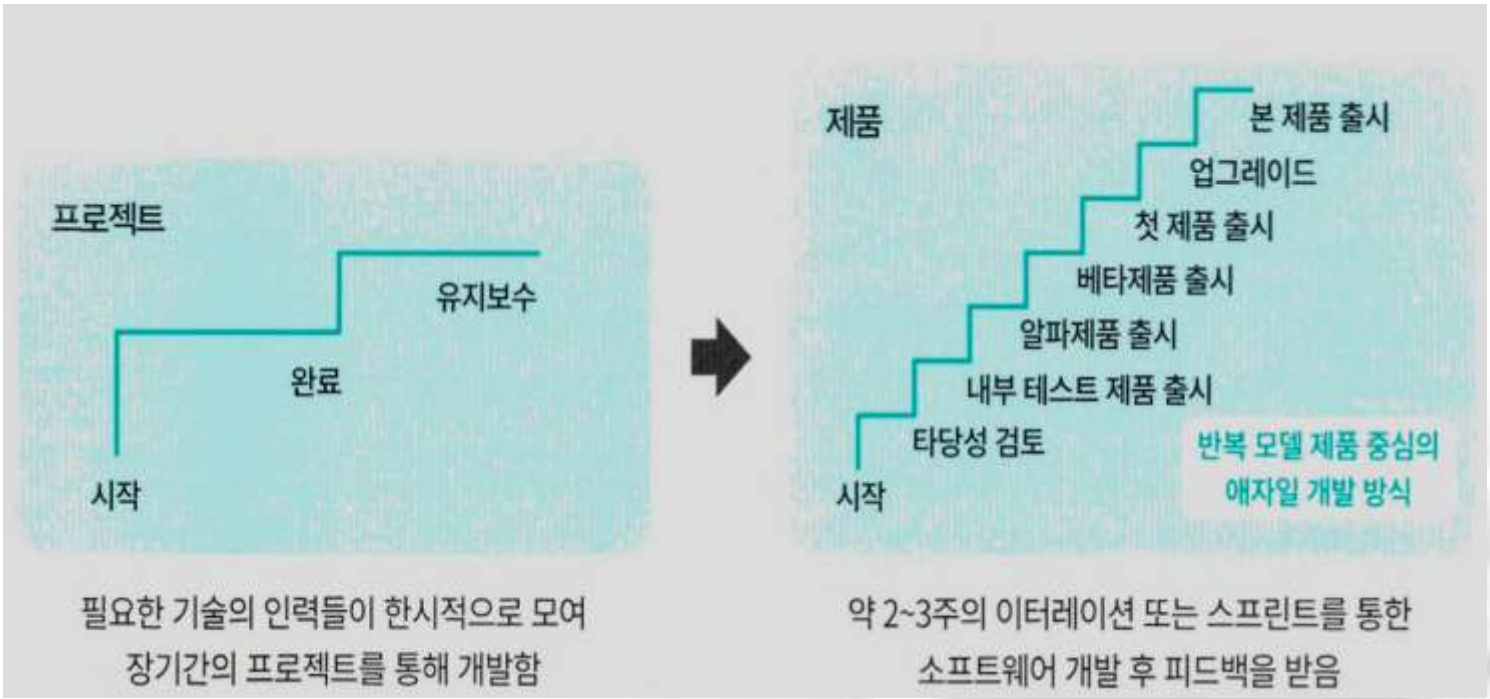
- Micro Service를 위한 조건
  - ✓ 개발 생명 주기의 변화: 프로젝트가 아니라 제품 중심
    - 기존에는 대부분의 애플리케이션 개발 모델이 프로젝트 단위였기 때문에 필요한 기술을 사용하는 인력들이 한시적으로 모여 장기간의 프로젝트를 통해 개발을 완료하고 나면 이를 운영 조직에 넘기는 방식으로 진행되어서 개발 조직 과 운영 조직이 분리돼 있음
    - 초기에 모든 일정을 계획했는데 요구사항 정의를 통해 개발할 기능을 나열하고 이에 대한 설계를 진행하며 설계가 완료돼야 개발이 진행되고 각 단계는 완료 데드라인이 있어 그 일정을 완료함으로써 최종 기능을 제공하기 때문에 프로젝트 기간 중에 발생한 변경이나 새로운 아이디어를 포용하지 못함
    - Micro Service 팀의 개발은 비즈니스의 갑작스런 트렌드 변화에 유연하게 대처해야 하고 개발 뿐 아니라 운영을 포함한 소프트웨어의 전체 생명주기를 책임져야 하기 때문에 소프트웨어를 완성해야 할 기능들의 집합으로 보는 것이 아니라 비즈니스를 제공하는 제품(product)으로 바라보고 개발한 뒤에 반응을 보고 개선하는 방식으로 소프트웨어를 개발
    - 소프트웨어를 개발하는 방식 측면에서 프로젝트 형태의 폭포수 모델 또는 빅뱅 방식으로 진행하는 것이 아니라 점진 반복적인 모델 제품 중심의 애자일(agile) 개발 방식을 채용하는데 이 방식은 약 2~3주 단위의 스프린트(Sprint)를 통해 소프트웨어를 개발 및 배포해서 바로 피드백을 받아 소프트웨어에 반영할 수 있게 해주기 때문에 소프트웨어를 한시적 프로젝트를 통해 고객의 고정된 요건을 받아 기능이 만족되면 제공 및 전달하는 개념으로 보지 않고 요건의 변화에 따라 지속적으로 개선되고 발전시킬 제품으로 바라봄

비지니스 민첩성

❖ Micro Service를 위한 조건

✓ 개발 생명 주기의 변화: 프로젝트가 아니라 제품 중심

- Micro Service는 계속 피드백을 받아 지속적으로 변화, 개선되고 향상되는 존재

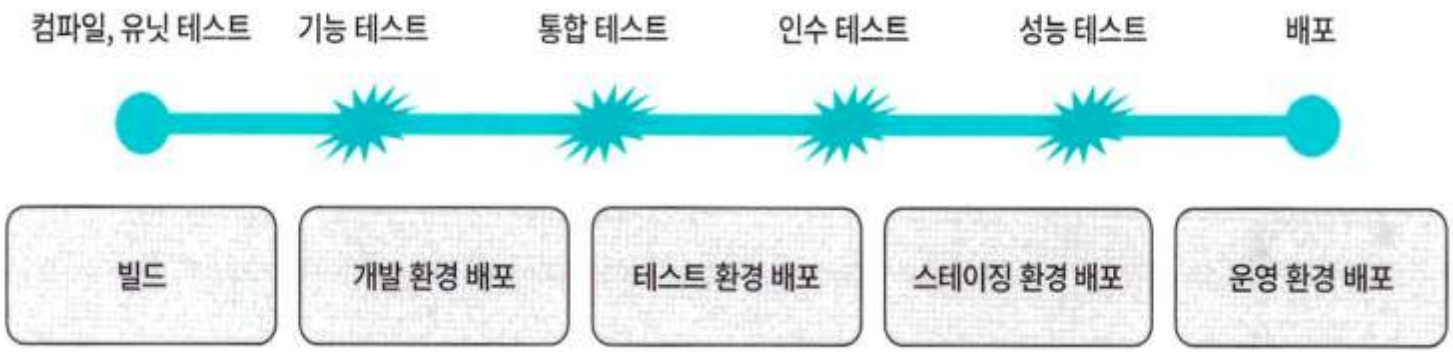


비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 개발 환경의 변화: 인프라 자동화
    - Micro Service는 독립적으로 배포
    - Monolithic처럼 한 덩어리로 배포한다면 수동으로 배포하는 방식도 크게 문제가 없겠지만 Micro Service처럼 여러 개로 쪼개진 상태에서는 수동으로 배포하는 방식은 바람직하지 않기 때문에 여러 개의 Micro Service를 빠르게 배포하기 위한 방법이 필요
    - Micro Service가 화려하게 등장한 이유는 Cloud라는 가상 인프라 발전에 기인
    - 전체 소프트웨어를 구현하는 과정은 개발 환경을 준비하는 과정, 실제로 소프트웨어를 개발하는 과정, 개발 완료된 소프트웨어를 빌드, 테스트, 배포하는 개발 지원 과정으로 구분
    - 첫 단계인 개발 환경 준비 과정에 개발 환경으로 Cloud 인프라를 활용하면 쉽고 빠르게 개발 환경을 준비할 수 있어서 팀의 개발 속도가 높아지는데 개발 지원 과정의 속도를 높이는 가장 좋은 방법은 자동화
    - 개발 지원 환경을 자동화하는 데는 소스 코드를 빌드하는 도구 와 빌드와 동시에 테스트하는 도구, 가상화된 인프라에 배포하는 도구가 모두 필요
    - Micro Service팀이 단기간에 제품을 빨리 개발하고 피드백을 받기 위해서는 이러한 개발 지원 환경의 자동화가 반드시 갖춰져야 함
    - 이 같은 환경은 개발과 운영을 동시에 수행하는 데브옵스(DevOps)를 궁극적으로 가능하게 하므로 데브옵스 개발 환경이라 속칭하기도 하며 이러한 개발 환경, 개발 지원 환경을 자동화하는 것을 모두 통틀어 인프라 자동화라고 하기도 하는데 인프라 자동화는 Micro Service 개발 과정의 필수 조건

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 개발 환경의 변화: 인프라 자동화
    - 배포 파이프라인의 자동화



  - ◆ 빌드/배포 파이프라인은 일반적으로 소스 코드 빌드 -> 개발 환경 배포 -> Staging 환경(운영 환경과 거의 동일한 환경을 만들어 놓고 운영환경으로 이전하기 전에 Security, 성능, 장애 등의 비기능적 부분을 검증하는 환경) 배포 - 운영 환경 배포로 구성

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 개발 환경의 변화: 인프라 자동화
    - 배포 파이프라인의 자동화
      - 빌드/배포 파이프라인 프로세스는 도구를 통해 자동화해야 하는데 최근에는 배포 환경이 Micro Service 개수에 따라 급격하게 늘어나기 때문에 이를 효율적으로 관리하기 위해 인프라 구성과 자동화를 마치 소프트웨어처럼 코드로 처리하는 빙식인 Infrastructure as Code가 각광받고 있음
      - Infrastructure as Code란 코드를 이용해 인프라 구성부터 애플리케이션 빌드, 배포를 정의하는 것을 의미하는데 이렇게 되면 수많은 하드웨어 리소스 설정을 동일하게 통제할 수 있으며 상황에 따른 검증되고 적절한 설정을 쉽게 복제하고 누구한테나 공유할 수 있게 돼서 인프라를 매우 효율적으로 관리할 수 있음

비지니스 민첩성

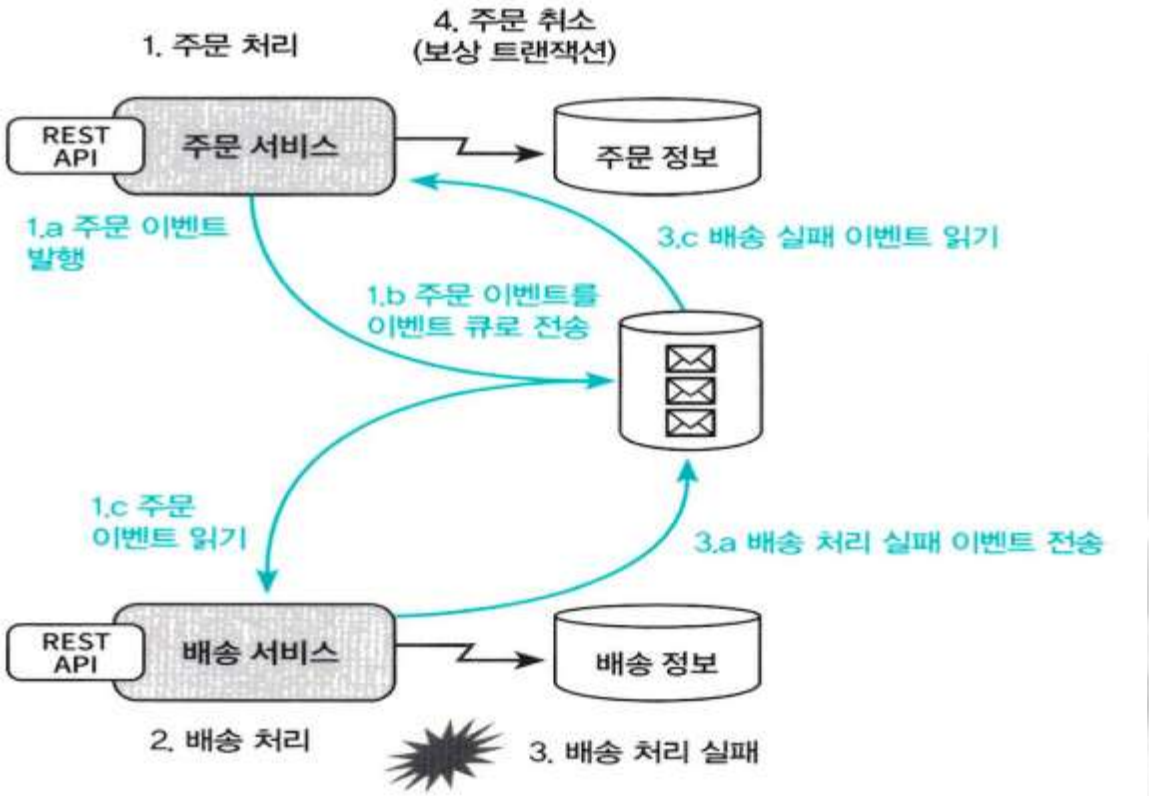
- Micro Service를 위한 조건
  - ✓ 저장소의 변화: 통합 저장소가 아닌 분권 데이터 관리
    - Monolithic 시스템을 살펴보면 단일 통합 데이터베이스를 사용
    - 단일 데이터베이스를 유지하는 방식은 과거 스토리지 가격 및 네트워크 속도에 따른 데이터의 안정성과 효율성을 추구한 결과로 데이터를 잘 정리하는 정규화가 반드시 추구해야 할 가치였지만 지금은 스토리지 가격이 저렴하고 네트워크 대역폭이 매우 커져서 데이터를 억지로 꾸깃꾸깃 뭉쳐서 작은 공간에 넣을 필요가 없음
    - Micro Service는 폴리글랏 저장소(polyglot persistence) 접근법을 선택하며 서비스 별로 데이터베이스를 갖도록 설계하는데 각 저장소가 서비스 별로 분산돼 있어야 하며 다른 서비스의 저장소를 직접 호출할 수가 없고 API를 통해서만 접근해야 한다는 의미
    - 이러한 구조에서는 비즈니스 처리를 위해 일부 데이터의 복제와 중복 허용이 필요한데 여기서 반드시 등장하는 문제가 있는데 바로 각 Micro Service의 저장소에 담긴 데이터의 비즈니스 정합성을 맞춰야 하는 데이터 일관성 문제
    - 데이터 일관성 처리를 위해서는 보통 2단계 커밋(two-phase commit) 같은 분산 트랜잭션 기법을 사용하는데 각각 다른 서비스를 하나의 트랜잭션으로 묶다 보면 각 서비스의 독립성도 침해되고 NoSQL 저장소처럼 2단계 커밋을 지원하지 않는 경우도 있기 때문에 Micro Service는 데이터 일관성 문제를 해결하기 위해 두 서비스를 단일 트랜잭션으로 묶는 방법이 아닌 비동기 이벤트 처리를 통한 협업을 강조하는데 이를 가리켜 결과적 일관성(Eventual Consistency)이라는 개념으로 표현하기도 하는데 간단히 말해 두 서비스의 데이터가 일시적으로 불일치하는 시점에 있고 일관성이 없는 상태지만 결국에는 두 데이터가 같아진다는 개념

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 저장소의 변화: 통합 저장소가 아닌 분권 데이터 관리
    - 여러 트랜잭션을 하나로 묶지 않고 별도의 로컬 트랜잭션을 각각 수행하고 일관성이 달라진 부분은 체크해서 보상 트랜잭션으로 일관성을 맞추는 개념
    - 주문 서비스와 배송 서비스가 있고 각 저장소가 분리돼 있음
    - 주문이 발생하면 반드시 배송 처리가 돼야 하는 비즈니스가 있다고 생각해 보면 두 업무를 2단계 커밋 과 같은 분산 트랜잭션 기법을 사용해 하나로 묶으면 배송 서비스가 실패했을 때 트랜잭션에 포함되는 주문 서비스도 함께 롤백(rollback) 처리해서 데이터의 일관성을 맞출 수 있을 것이지만 하나의 데이터 저장소가 2단계 커밋을 지원하지 않는 저장소라면 이렇게 하기가 불가능할 것
    - 분산 트랜잭션으로 묶는다면 두 서비스가 강하게 결합돼 있어 서비스를 분리한 효과를 누리기 힘들 것

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 저장소의 변화: 통합 저장소가 아닌 분권 데이터 관리



비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 저장소의 변화: 통합 저장소가 아닌 분권 데이터 관리
    - Micro Service가 추구하는 다른 방법은 각 트랜잭션을 분리하고 큐(queue) 메커니즘을 이용해 보상 트랜잭션을 활용하는 방법
      - 주문 서비스가 주문 처리 트랜잭션을 수행
        - 동시에 주문 이벤트를 발행
        - 주문 이벤트가 메시지 큐로 전송
        - 배송 서비스가 주문 이벤트를 인식
      - 배송 서비스가 주문 처리에 맞는 배송 처리 트랜잭션을 수행(비즈니스 일관성 만족)
      - 배송 처리 트랜잭션 중 오류로 트랜잭션을 실패
        - 배송 처리 실패 이벤트를 발행
        - 배송 처리 실패 이벤트가 메시지 큐로 전송
        - 주문 서비스가 배송 처리 실패 이벤트를 인식
      - 주문 서비스는 주문 취소(보상 트랜잭션)를 수행(비즈니스 일관성 만족)

비지니스 민첩성

- Micro Service를 위한 조건
  - ✓ 위기 대응 방식의 변화: 실패를 고려한 설계
    - 아마존의 부사장인 버너 보겔스(Werner Vogels)는 소프트웨어는 모두 실패한다 라고 말한 바 있는데 시스템은 언제든 실패할 수 있으며 실패해서 더는 진행할 수 없을 때도 자연스럽게 대응할 수 있도록 설계해야 한다는 의미로 이러한 성격을 내결함성(fault tolerance)이라 함
    - 예전의 시스템 아키텍처는 무결함이나 실패 무결성을 추구했는데 시스템이 다운되지 않고 중단되지 않기 위해서는 완벽을 추구해야 하며 강건해야 했지만 버너 보겔스의 말처럼 어떤 시스템도 실패하지 않을 수 없는데 실패하지 않는 시스템을 만드는 것보다 실패에 빠르게 대응할 수 있는 시스템을 만드는 편이 더 쉽고 효율적이라는 것인데 이를 위해서는 다양한 실패에 대비해서 완벽히 테스트할 수 있는 환경을 마련해야 하고 시스템의 실패를 감지하고 대응하기 위해 실시간 모니터링 체계도 갖춰야 함
    - 이러한 예로 서킷 브레이커(circuit breaker) 패턴을 들 수 있는데 이 패턴은 회로 차단기처럼 각 서비스를 모니터링하고 있다가 한 서비스가 다운되거나 실패하면 이를 호출하는 서비스의 연계를 차단하고 적절하게 대응하는 것을 의미하는데 이러한 설계는 서비스가 긴급 장애 상황에 빠르고 유연하고 탄력 적으로 대응할 수 있게 함
    - 넷플릭스에서는 카오스 몽키(chaos monkey)라는 장애를 일부러 발생시키는 도구를 만들어 이러한 탄력적인 아키텍처가 제대로 동작하는지 점검

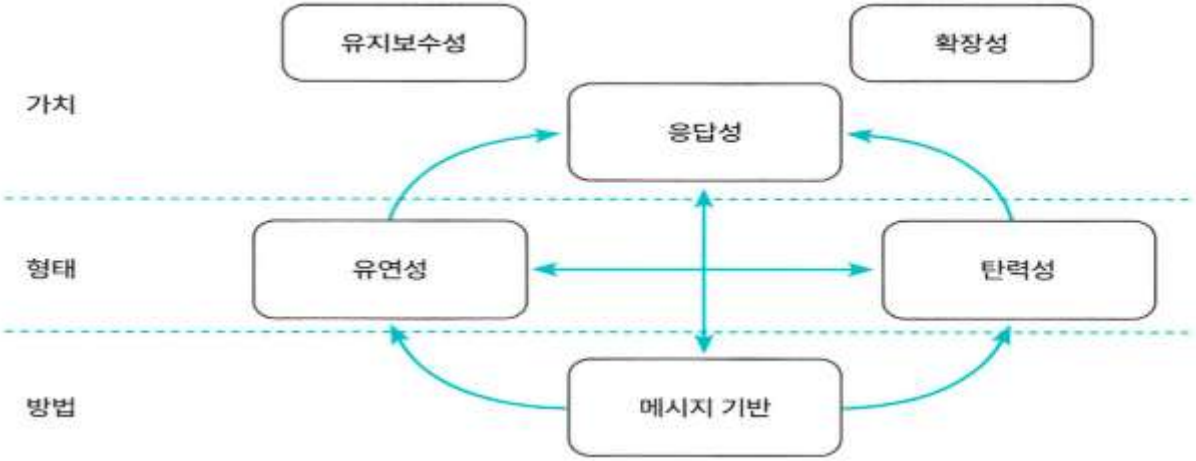
MSA의 이해

- 리액티브 선언: 현대 애플리케이션이 갖춰야 할 바람직한 속성들
  - ✓ 소프트웨어 아키텍처란 소프트웨어를 구성하는 요소 와 그 구성 요소 간의 관계를 정의한 것
  - ✓ 아키텍처를 정의하는 과정은 시스템 구축을 위한 여러 가지 비기능 요건(성능, 가용성, 보안, 유지 보수성, 확장성 등)을 만족하는 다양한 해결 방법을 찾는 과정으로 여러 문제 영역에 대한 해결책을 찾는 과정
  - ✓ Micro Service 아키텍처는 Cloud라는 가상화된 인프라를 활용해서 구조화하는 것이기 때문에 가상화된 인프라의 특징을 고려해서 설계해야 함
  - ✓ 현대의 애플리케이션은 도처에 존재하는데 가깝게는 스마트폰이나 데스크톱 컴퓨터에 탑재되는 애플리케이션부터 멀게는 생활가전이나 IoT 기기에 탑재되는 애플리케이션 등이 있음
  - ✓ 사람들은 이러한 기기에 포함된 애플리케이션이 요청에 즉각 응답하고 항상 가동되길 기대
  - ✓ 현대 애플리케이션에 대한 기대를 잘 표현한 문서가 있는데 2014년 요나스 보네(Jonas Boner) 등이 선언한 리액티브 선언문(The Reactive Manifesto)으로 리액티브 선언문에서는 응답성(Responsive), 탄력성(Resilient), 유연성(Elastic), 메시지 기반(Message Driven)이라는 4가지 특성을 강조하고 이러한 요건을 만족하는 시스템을 리액티브 시스템이라고 정의

MSA의 이해

❖ 리액티브 선언: 현대 애플리케이션이 갖춰야 할 바람직한 속성들

✓ 리액티브 선언의 4가지 요소



- 응답성(Responsive): 사용자에게 신뢰성 있는 응답을 빠르고 적절하게 제공하는 것을 의미
- 탄력성(Resilient): 장애가 발생하거나 부분적으로 고장나더라도 시스템 전체가 고장 나지 않고 빠르게 복구하는 능력을 의미
- 유연성(Elastic): 시스템의 사용량에 변화가 있더라도 균일한 응답성을 제공하는 것을 의미하며 시스템 사용량에 비례해서 자원을 늘리거나 줄이는 능력
- 메시지 기반(Message Driven): 비동기 메시지 전달을 통해 위치 투명성, 느슨한 결합, 논블로킹 통신을 지향하는 것을 의미

MSA의 이해

- 리액티브 선언: 현대 애플리케이션이 갖춰야 할 바람직한 속성들
  - ✓ 4가지 요소는 모두 리액티브 시스템을 만들기 위한 요소이고 각 요소는 상호 보완적
  - ✓ 리액티브 시스템은 첫 번째 요소인 사용자에게 가장 빠르고 적절한 응답을 제공하기 위해 장애로부터 빠르게 회복하고(탄력성) 시스템은 사용량 트래픽에 반응해 시스템 자원 조절을 유연하게 수행하며(유연성) 메시지 기반 통신을 통해 느슨한 결합 과 위치 투명성을 제공해야 한다는 의미
  - ✓ reactive의 사전적 의미는 반응을 보이는 인데 이는 다양한 상황에 따라 빠르고 적절하게 반응하는 시스템을 의미하는데 다양한 상황이란 여러 프런트 엔드 장비나 시스템이 연계하는 레거시 시스템과 내부 장비들 그리고 빈번히 발생하는 장애나 트래픽 증감을 의미하는 것으로 급변하는 상황에 적응할 수 있는 시스템을 요구하는 것
  - ✓ 여기서 리액티브 시스템이 반드시 갖춰야 할 공통적인 특성 하나를 발견할 수 있는데 바로 아키텍처 유연성(flexibility)
  - ✓ 리액티브 시스템이 신뢰성 있는 응답을 빠르게 제공하고 부분적 장애가 빨리 복구되고 수요 증가에 탄력적으로 대응하기 위해서는 시스템 자체가 변화와 확장에 언제든지 대응할 수 있는 아키텍처 유연성을 갖추는 것이 필수
  - ✓ 아키텍처 유연성은 시스템을 구성하는 구성 요소 간의 관계들이 느슨하게 맺어져 있어 언제든지 대체되거나 추가 확장될 수 있는 특성을 의미하는데 특히 Cloud 인프라 자체가 변화무쌍한 비즈니스 환경에 대응할 수 있는 유연성과 확장성을 갖추고 있기 때문에 그것을 사용하는 애플리케이션 아키텍처도 반드시 이 같은 아키텍처 유연성을 갖춰야 함
  - ✓ 메시지 기반 이라는 요소가 이러한 아키텍처 유연성을 만족시키는 요소라 할 수 있는데 Micro Service 아키텍처에서도 Micro Service 간의 의존성을 줄이는 중요한 특성

MSA의 이해

- 강 결합에서 느슨한 결합의 아키텍처로의 변화
  - ✓ 예전에는 아키텍처의 구성 요소들을 각 기업이나 특정 벤더의 제품에 전적으로 의존해서 구축하거나 수정이 필요한 부분만 별도로 직접 개발하는 경우가 많았기 때문에 특정 벤더 솔루션이나 프레임워크가 변경될 경우 그것에 의존하는 애플리케이션의 많은 부분들을 변경해야 할 정도로 강 결합돼 있었는데 이처럼 특정 벤더 중심의 아키텍처는 검증된 유명 제품군을 사용한다는 점에서 품질이 보장된다고 생각할 수 있는 반면 특정 벤더에 의존한다는 점에서 특정 기술에 락인(lock-in)되어 쉽게 변경하거나 확장하지 못한다는 단점이 있음
  - ✓ 최근에 이러한 분위기를 바꿔 하나의 벤더에 의존하거나 직접 구축할 필요가 적어졌는데 Cloud 환경하에서 사용되는 오픈소스 또는 오픈소스를 기반으로 한 상용 제품들이 이전의 유명 벤더의 제품군 만큼이나 품질이 높아지고 다양한 기능을 지원하면서 서로 다른 오픈소스 제품 간에도 충분한 호환성을 제공하기 때문
  - ✓ 이러한 흐름은 아키텍처 설계 활동에도 변화를 가져왔는데 예전에는 검증된 기술이나 솔루션을 기반으로 기술을 직접 구현하는 폐쇄적인 방식이었던 것에 비해 최근의 아키텍처 설계는 필요한 영역에 적절한 솔루션을 선택하고 조합하는 개방적인 방식으로 바뀌고 있음
  - ✓ 최근 아키텍처 설계 문서들을 보면 각 솔루션의 로고로 채워지는 경향이 있는 데 이것은 아키텍처링이 유연하고 호환성 있는 적절한 솔루션 및 오픈소스를 선택하는 과정 임을 보여줌

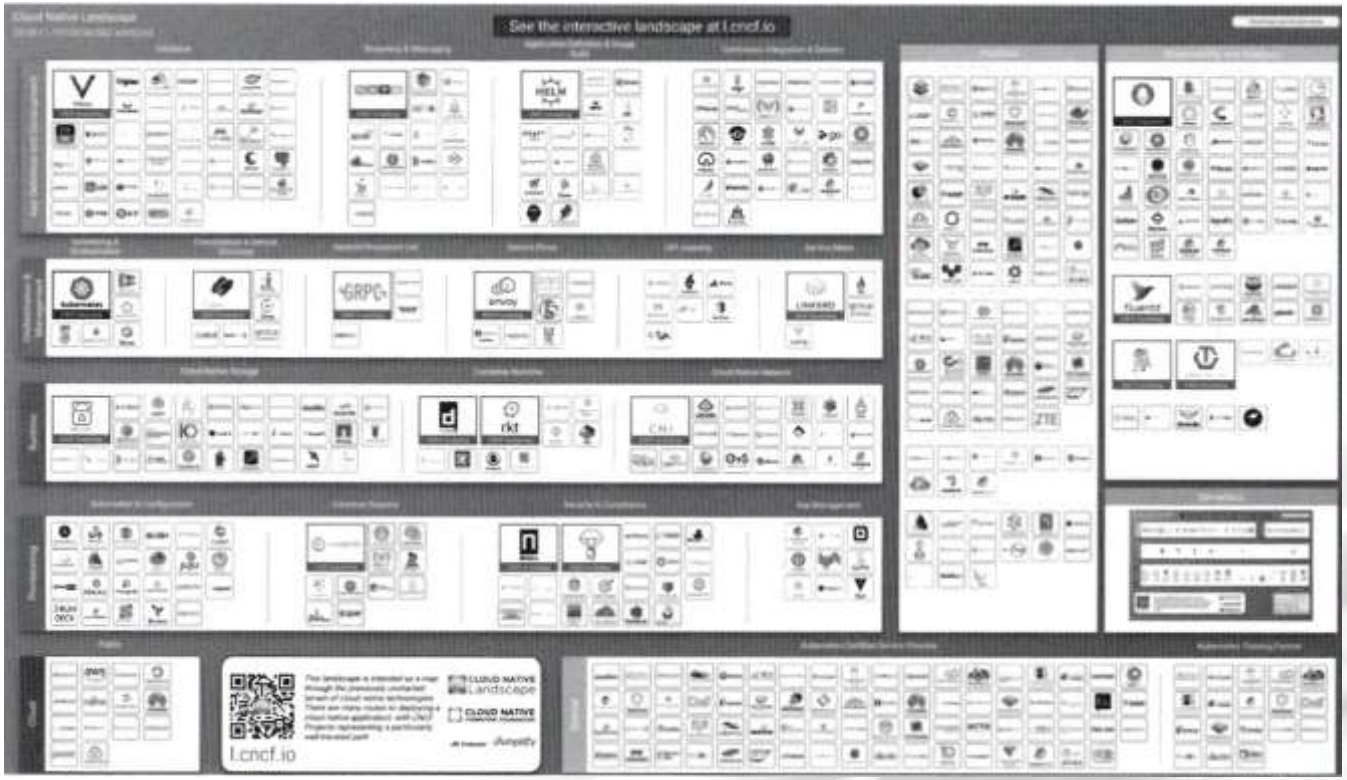
MSA의 이해

❖ 강 결합에서 느슨한 결합의 아키텍처로의 변화

✓ 레이어 별로 선택된 애플리케이션

MSA의 이해

- ❖ 강 결합에서 느슨한 결합의 아키텍처로의 변화
  - ✓ Cloud Native 컴퓨팅 재단(CNCF)에서 제공하는 Cloud Native 지형도(cloud native landscape)

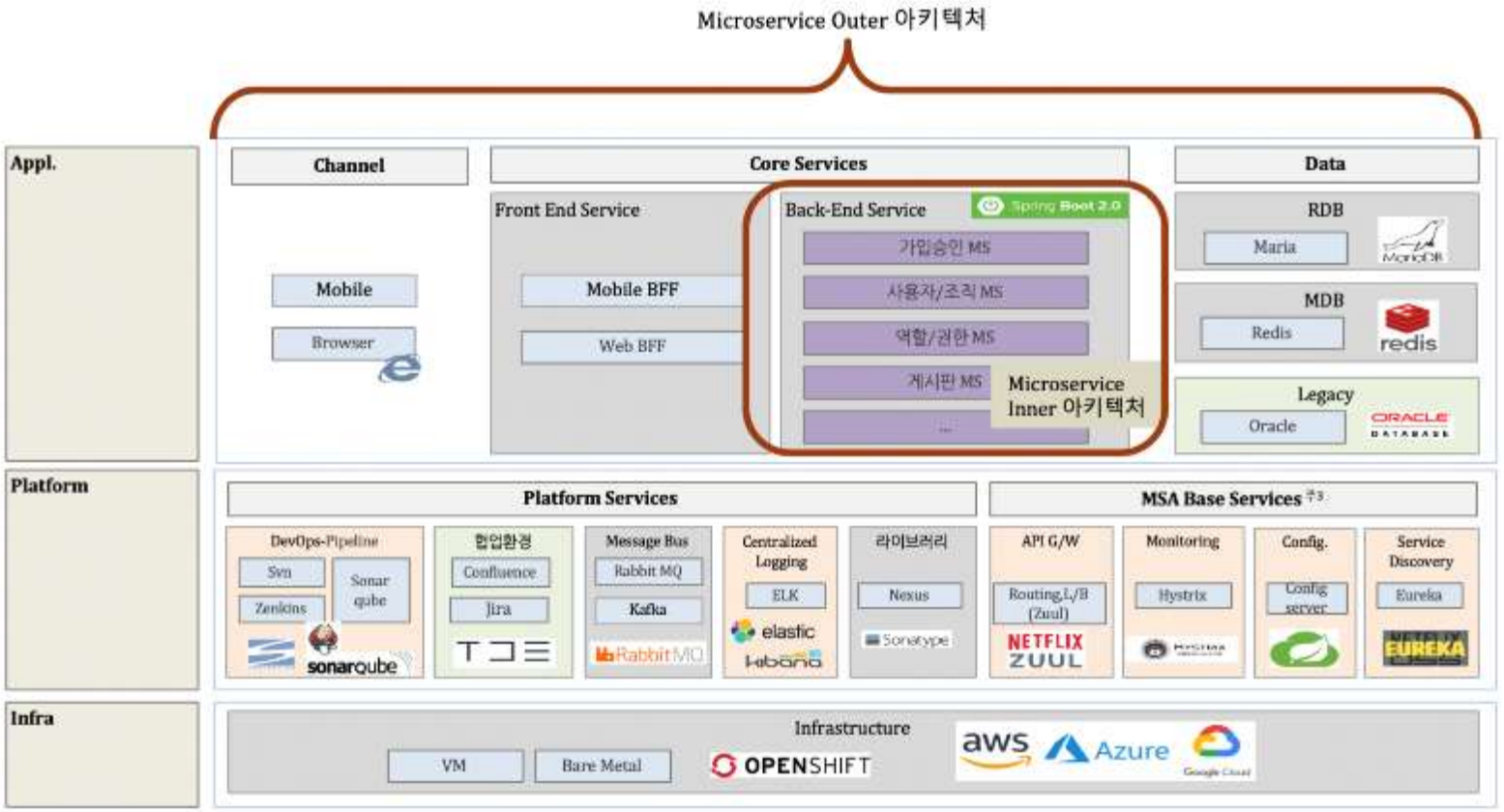


MSA의 이해

- 강 결합에서 느슨한 결합의 아키텍처로의 변화
  - ✓ Cloud 기반의 애플리케이션을 구축하는 데 필요한 인프라 및 애플리케이션 영역에서 다양한 오픈소스 제품이나 상용 제품이 사용되고 있음
  - ✓ Cloud Native 지형도는 매년 매분기 마다 업데이트되고 있는데 얼마나 많은 영역에서 다양한 오픈소스 및 제품들이 생겨나고 포진돼 있는지 알 수 있음
  - ✓ 아키텍트는 이처럼 다양한 기술 영역의 변화 흐름을 이해하고 따라가야 하며 최적의 Cloud 환경을 구성할 적절한 제품 및 솔루션을 선택해서 조합할 필요가 있음
  - ✓ 이러한 모습은 강 결합 위주의 Monolithic 아키텍처가 유연하고 느슨하게 결합되는 Micro Service 기반 아키텍처로 변화되고 있음을 보여주기도 함

MSA의 이해

- Micro Service의 외부 아키텍처와 내부 아키텍처
  - ✓ Micro Service 아키텍쳐 – 각 구성 요소들은 대체하거나 변경할 수 있도록 구성



MSA의 이해

- ❖ Micro Service의 외부 아키텍처와 내부 아키텍처
  - ✓ Micro Service 아키텍쳐 – 각 구성 요소들은 대체하거나 변경할 수 있도록 구성
    - 밑에서부터 인프라, 플랫폼, 애플리케이션 영역으로 구분되며 각 관계를 설명하면 맨 아래에 기반이 되는 하드웨어 인프라가 있고, 인프라 영역 위에 애플리케이션을 운영 및 구동하기 위한 플랫폼이 올라가며 플랫폼 위에 애플리케이션인 서비스가 구동됨
    - 아래의 인프라 영역 과 플랫폼 영역, 애플리케이션 영역에 있는 구성 요소 및 그것들의 관계를 정의하는 것이 MSA 외부 아키텍처(outer architecture)
    - 외부 아키텍처는 Micro Service가 운영되는 환경을 정의하는데 여기에는 인프라 환경, 플랫폼 환경, Micro Service가 운영되는 애플리케이션 환경이 모두 포함되며 특히 애플리케이션 측면에서는 여러 개의 Micro Service를 관리하고 운영하기 위한 애플리케이션도 모두 포함되는데 이러한 환경을 Cloud 아키텍처가 요구하는 유연성과 확장성을 고려해서 정의해야 하는 것
    - 실제로 비즈니스가 실행되는 비즈니스 애플리케이션 즉 각 Micro Service의 내부 구조도 정의해야 하는데 이를 MSA 내부 아키텍처(inner architecture) 라고 함
    - 내부 아키텍처는 Micro Service가 제공하는 API, 비즈니스 로직, 이벤트 발행, 데이터 저장 처리 등을 어떻게 구조화해야 하는가에 관한 내용이며 당연히 이 구조도 변화에 적응 가능하도록 유연하고 확장성 있게 구현해야 함

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 저명한 소프트웨어 아키텍트인 크리스 리처드슨(Chris Richardson)은 Micro Service 관련 기술을 설명하는 자신의 온라인 매체인 microservices.io 에서 이러한 Micro Service 아키텍처 패턴을 인프라 패턴, 애플리케이션 인프라 패턴, 애플리케이션 패턴 등으로 분류해서 정의
  - ✓ 인프라가 구축돼야 하고 그 위에 미들웨어가 올라가고 미들웨어 위에서 애플리케이션이 동작
  - ✓ MSA 구성 요소 및 패턴의 유형
    - 인프라 구성 요소: Micro Service를 지탱하는 하부 구조 인프라를 구축하는 데 필요한 구성요소
    - 플랫폼 패턴: 인프라 위에서 Micro Service의 운영과 관리를 지원하는 플랫폼 차원의 패턴
    - 애플리케이션 패턴: Micro Service 애플리케이션을 구성하는 데 필요한 패턴

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - IT 업계에서 인프라의 의미는 엔터프라이즈 IT 환경을 운영하고 관리하는데 필요한 근간이 되는 하드웨어, 소프트웨어, 네트워킹 구성요소, 운영체제, 데이터 스토리지 등을 모두 포괄
    - Cloud 환경에서는 이러한 인프라 구성요소가 가상화 되어 제공
    - 퍼블릭 Cloud와 베어 메탈, 프라이빗 Cloud 환경
      - 예전에 오랜 시간에 걸쳐 힘들게 구축했던 인프라를 이제는 AWS, 구글, 마이크로소프트, IBM 등 세계적인 플랫폼 사업자들이 자동화된 IaaS(Infrastructure as a Service), PaaS(Platform as a Service) 서비스를 통해 쉽고 편하게 이용할 수 있게 제공
      - 예를 들면 시스템의 자원 구성, 할당. 관리, 모니터링 등이 Cloud 서비스로 구성돼 있어 일련의 설정 작업을 몇 번의 버튼 클릭만으로 처리할 수 있음
      - 이 같은 환경에서 아키텍트가 해야 할 일을 하나씩 살펴보면 먼저 맨 하부의 시스템의 기반이 되는 인프라를 구축해야 하는데 여기서 고민은 기존의 물리적인 베어 메탈 장비를 구매해서 구축 해야 하느냐 아니면 가상화 환경을 선택해서 이용하느냐 이며 또한 가상화 환경에 구축하기로 결정했더라도 Cloud 사업자가 서비스로 제공하는 퍼블릭 IaaS, PaaS를 선택할지 또는 직접 구매 하거나 기존의 보유한 베어 메탈 서버에 프라이빗 PaaS를 구축할지를 고민해야 함

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● 퍼블릭 Cloud와 베어 메탈, 프라이빗 Cloud 환경
      - ◆ Micro Service는 어떠한 장비에도 구동될 수 있는데 Micro Service는 어떠한 환경에서도 유연하도록 구성돼야 하므로 특정 인프라를 고집하지 않지만 가상화 장치없이 베어 메탈 장비로 Micro Service 애플리케이션을 구동한다면 Micro Service마다 베어 메탈 장비를 구축해야 하고 인프라의 유연한 확장/축소를 기대하기 힘든 무모한 작업이 될 것이므로 당연히 가상 인프라 환경을 검토할 필요가 있으며 MSA 시스템을 위한 베어 메탈을 고려한다면 그것은 베어 메탈에 별도의 프라이빗 Cloud 환경을 구축하는 것을 의미
      - ◆ 예전 같으면 이러한 하위의 인프라 선택이 그 위에 올라가는 다른 소프트웨어 구성 요소에 영향을 끼쳤겠지만 요즘은 인프라환경으로 퍼블릭 Cloud 환경이나 프라이빗 Cloud 환경이나 어떤 것을 선택하든 다른 아키텍처 요소와 유연하게 결합할 수 있어 크게 신경 쓸 필요가 없는데 이러한 모든 소프트웨어들이 상호 독립적이고 서로 호환되도록 유연한 구조로 만들어졌기 때문
      - ◆ 이처럼 유연성이 Cloud 기반 소프트웨어의 필수 요소로 자리 잡고 있는 것이 요즘 추세

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● VM 과 컨테이너
      - ◆ 가상 인프라 환경을 활용하기로 선택했다면 그 다음으로 가장 먼저 고민해야 할 사항은 가상 머신(VM; Virtual Machine) 제품과 컨테이너 기반 제품 중 하나를 선택하는 문제
      - ◆ 차이점을 살펴보면 가상 머신은 하이퍼바이저(hypervisor)라는 소프트웨어를 이용해 하나의 시스템에서 여러 개의 운영체제를 사용하는 기술인 반면 컨테이너는 하이퍼바이저 없이 컨테이너 엔진을 사용해 가상의 격리된 공간을 생성

MSA의 이해

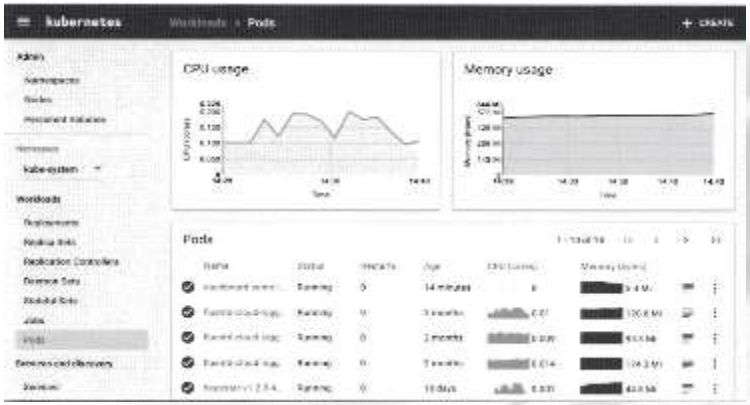
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● VM 과 컨테이너
      - ◆ 차이점은 게스트 OS의 유무로 볼 수 있는데 게스트 OS를 사용하는 가상 머신에서는 운영체제 패치 설치나 관련 라이브러리 설치 같은 오버헤드가 지속적으로 발생하기 때문에 Micro Service 같은 작은 서비스를 패키지하고 배포하기에는 컨테이너 환경이 더 적합
      - ◆ 가장 대표적인 컨테이너 기술로는 필요 라이브러리나 실행 파일을 여러 개의 레이어 이미지로 추가하거나 변경할 수 있는 도커(Docker)가 유명하고 가장 많이 사용됨
      - ◆ 도커 컨테이너는 레이어 단위의 이미지를 포개는 방식으로 구성되며 밑에서 부터 애플리케이션 구동을 위한 기반 이미지, 운영체제, 런타임, 애플리케이션이 이미지로 정의

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● VM 과 컨테이너
      - ◆ 도커 컨테이너의 이점
        - ▪ 이식성: 어떠한 호스트 커널이나 플랫폼 버전에 상관없이 도커만 실행할 수 있으면 사용 가능하며 동일하게 동작
        - ▪ 신속성: 크기가 작고 가볍기 때문에 빠르게 배포 가능하며 문제 발생 시 수정할 필요 없이 새로 기동하면 됨
        - ▪ 재사용성: 동일한 환경을 재사용해서 쉽게 설정 가능하기 때문에 개발, 테스트, 스테이징, 프로덕트 환경을 동일한 환경으로 구축하기가 쉬움
      - ◆ Micro Service 같이 독립적으로 배포되고 수정되기 위한 환경은 가상 머신보다는 컨테이너가 더 적절한데 Micro Service의 가변적이고 유연한 속성을 컨테이너가 쉽고 빠르게 지원할 수 있기 때문
      - ◆ 가장 많이 사용하는 가상 머신 제품군으로는 AWS EC2, 애저(Azure) VM 등이 있고 컨테이너 기술로는 도커 외에 Unikernels, LXD, OpenVZ, RKt 등이 있으나 도커가 가장 유명하고 실질적 표준으로 자리 잡고 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● 컨테이너 오케스트레이션
      - ◆ 컨테이너 기술을 선택했다면 컨테이너를 관리하기 위한 기술 또한 필요
      - ◆ 컨테이너가 많아지면 그에 따라 컨테이너의 자동 배치 및 복제, 장애 복구, 확장 및 축소, 컨테이너 간 통신, 로드 밸런싱 등의 컨테이너 관리를 위한 기능이 필요한데 이러한 기술을 컨테이너 오케스트레이션(Container Orchestration)이라 함
      - ◆ 컨테이너 오케스트레이션 도구로는 도커 스웜(Docker Swarm), 아파치 메소스(Apache Mesos) 등이 있으며 최근에는 구글이 자사의 도커 컨테이너 관리 노하우를 CNCF 재단에 제공해서 공개한 쿠버네티스(Kubernetes8가 큰 인기를 끌고 있는데 쿠버네티스를 이용하면 손쉽게 사설 컨테이너 플랫폼 환경을 구축할 수 있기 때문
      - ◆ 쿠버네티스 대시보드



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● 컨테이너 오케스트레이션
      - ◆ 쿠버네티스나 AWS 같은 플랫폼 사업자가 제공하는 대부분의 PaaS UI는 대부분 이와 유사한 시각적인 대시보드를 제공하는데 이곳에서 컨테이너 배포의 기본 단위에 해당하는 파드(Pod), 디플로이먼트(Deployment), 레플리카셋(Replica Sets) 정보를 확인하고 설정할 수 있음
      - ◆ 쿠버네티스는 다음과 같은 주요 기능을 제공
        - ▪ 자동화된 자원 배정(Automatic binpacking): 각 컨테이너가 필요로 하는 CPU와 메모리를 쿠버네티스에 요청하면 컨테이너를 노드에 맞춰 자동으로 배치
        - ▪ 셀프 치유(Self-healing): 컨테이너의 이상 유무를 점검(health check)해서 실패한 경우 자동으로 교체하고 재스케줄링
        - ▪ 수평 확장(Horizontal scaling): CPU 및 메모리 사용량을 일정량을 초과하면 자동으로 확장
      - ◆ 쿠버네티스는 애플리케이션으로 구현 가능한 일부 Micro Service 운영/관리 패턴을 자체적으로 내장하고 있기도 하기 때문에 이러한 대중성과 인기에 힘입어 최근에는 AWS, 애저, Pivotal, IBM 등 자체적인 컨테이너 기반의 PaaS 서비스를 제공하는 CSP 사업자들도 별도로 쿠버네티스 기반의 컨테이너 서비스를 제공

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - Backing Service
      - ◆ 마이크로서비스에서 데이터에 대해 정의 하는 부분
      - ◆ Persistence: RDBMS와 NoSQL을 의미(Oracle, mysql, mariaDB, mongoDB, postgreSQL ...)
      - ◆ Cache: 데이터 캐쉬를 정의(redis)
      - ◆ Message Broker: Queue등을 이용한 메시지 전달 매체와 방법을 의미 (RabbitMQ, Kafka 등)

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - Telemetry
      - ◆ 마이크로서비스는 각 서비스들이 분리되어 있기 때문에 각 서비스간의 로그를 한데 모아야 하는 이슈, 각 서비스가 어떻게 호출되는지 추적해야 하는 이슈, 그리고 각 서비스에 대한 모니터링 이슈가 생길 수 밖에 없음
      - ◆ Telemetry는 바로 Log, Trace, Monitoring을 의미
      - ◆ Logging
        - ▪ 여러 개로 분리된 서비스에서 생성되는 분산된 로그 파일에 대한 중앙화, 집중화 방안
        - ▪ ELK(Elastic Search, Logstash, Kibana), EFK(Elastic Search, Fluentd, Kibana)
      - ◆ Trace
        - ▪ 클라이언트의 단일 요청에 대해 마이크로서비스간 다중 호출 관계가 있을 때 이에 대한 추적 방안
        - ▪ sluth/zipkin
      - ◆ Moinitoring
        - ▪ 다중화 된 서비스에 대한 모니터링 방안
        - ▪ Prometheus/Grafana

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - CI/CD
      - 마이크로서비스는 많은 구성요소들이 유기적으로 통합되어 하나의 서비스를 구성
      - 너무나 많은 구성 요소들로 되어 있기때문에 개발, 배포, 테스트 등이 쉽지 않음
      - CI/CD는 이를 위해 많은 부분을 자동화시켜 지속적인 통합과 배포(Delivery)를 지원해 주는 개념
      - Nexus, Habor, Helm, Jenkins, Git, gradle, maven, junit등으로 구성될 수 있음

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - 그 밖의 다양한 Cloud 인프라 서비스
      - AWS, Azure, GCP 등의 CSP 사업자 모두 기존의 물리 서버, 네트워크. 스토리지, DB를 대체 할 다양한 가상 서버, 가상 네트워크 및 가상 스토리지, 가상 DB 등을 제공하고 있으며 이들의 연계는 퍼블릭/프라이빗 간, 가상/물리 레거시 시스템 간에 모두 가능
      - Cloud 인프라 선택지는 다양
      - 애플리케이션 형태를 MSA가 아닌 Monolithic 시스템으로 구축하기 위해서는 가상 머신 Cloud 제품군인 AWS EC2, Azure VM을 사용할 수도 있고 Monolithic 시스템이 아니라 MSA 시스템으로 간다고 해도 쿠버네티스는 아니지만 동일하게 컨테이너 기반인 AWS Elastic Beanstalk나 Elastic Container Service(ECS), Azure Web App, Google App Engine 등의 PaaS를 고려할 수도 있음
      - 처음에 Cloud 서비스 제품군을 접하면 각 벤더가 위와 같이 매우 다양한 제품군을 제공하기 때문에 선택하기가 어렵지만 이러한 서비스들을 Cloud 서비스 유형(IaaS. PaaS, CaaS)으로 묶어서 보면 단순해짐

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 인프라 구성 요소
    - ● 그 밖의 다양한 Cloud 인프라 서비스
      - ◆ 서비스 유형별 대표적인 Cloud 서비스
        - ▪ IaaS(Infrastructure as a Service): 가상 머신, 스토리지, 네트워크 같은 인프라를 필요한 만큼 적시에 제공하는 서비스로서 사용자는 이러한 인프라를 이용해 개발 환경을 구성한 후 애플리케이션을 배포 - AWS EC2(Elastic Compute Cloud), GCP Compute Engine, Azure VM
        - ▪ CaaS(Container as a Service): 컨테이너 기반 가상화를 사용해 컨테이너를 업로드, 구성, 실행. 확장, 중지할 수 있는 서비스로 애플리케이션을 바로 구동할 수 있는 환경을 제공한다는 점에서 PaaS와 유사 하지만 다른 환경에도 이식 가능한 컨테이너 기반 가상화를 제공한다는 점이 다름 - 마이크로소프트의 Azure Kubernetes Service(AKS), 아마존의 Elastic Kubernetes Service(EKS), Google Kubernetes Engine(GKE), AWS ECS
        - ▪ PaaS(Platform as a Service): 복잡함 없이 애플리케이션을 곧바로 개발, 실행, 관리할 수 있는 플랫폼 환경을 서비스 형태로 제공하는 것으로 IaaS 위에 실제로 애플리케이션이 실행될 수 있는 미들웨어나 런타임까지 탑재된 환경 - Azure Web App, Google App Engine, Cloud Foundry, Heroku, AWS Elastic Beanstalk

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - 애플리케이션이 실제로 구동되는 인프라 환경을 결정했다면 그 다음으로 선택한 인프라 환경 위에서 애플리케이션을 운영하고 관리하는 환경을 구성하는 방법을 생각해야 하는데 특히 애플리케이션을 빌드하고 인프라에 배포할 수 있는 환경이 중요
    - Micro Service의 개념에서 마틴 파울러도 강조했지만 MSA 시스템을 구성하는 수많은 Micro Service를 하나하나 수동으로 빌드하고 배포한다면 비효율적이고 큰 혼란을 가져올 것이기 때문에 이러한 과정을 하나 하나 통제하고 자동화하는 것이 중요해졌음
    - 개발 지원 환경: 데브옵스 인프라 구성
      - 그래서 필요한 요소가 Micro Service를 빌드하고 테스트한 뒤 배포할 수 있게 도와주는 개발 지원 환경인 데브옵스(DevOps) 환경
      - 데브옵스는 앞에서 언급한 것처럼 개발과 운영이 분리되지 않은 개발 및 운영을 병행할 수 있는 조직 또는 그 문화를 일컫는데 여기서는 협의의 의미로 개발과 운영을 병행 가능하게끔 높은 품질로 소프트웨어를 빠르게 개발하도록 지원하는 빌드, 테스트, 배포를 위한 자동화 환경

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 개발 지원 환경: 데브옵스 인프라 구성
      - ◆ 수동 빌드 및 배포 과정

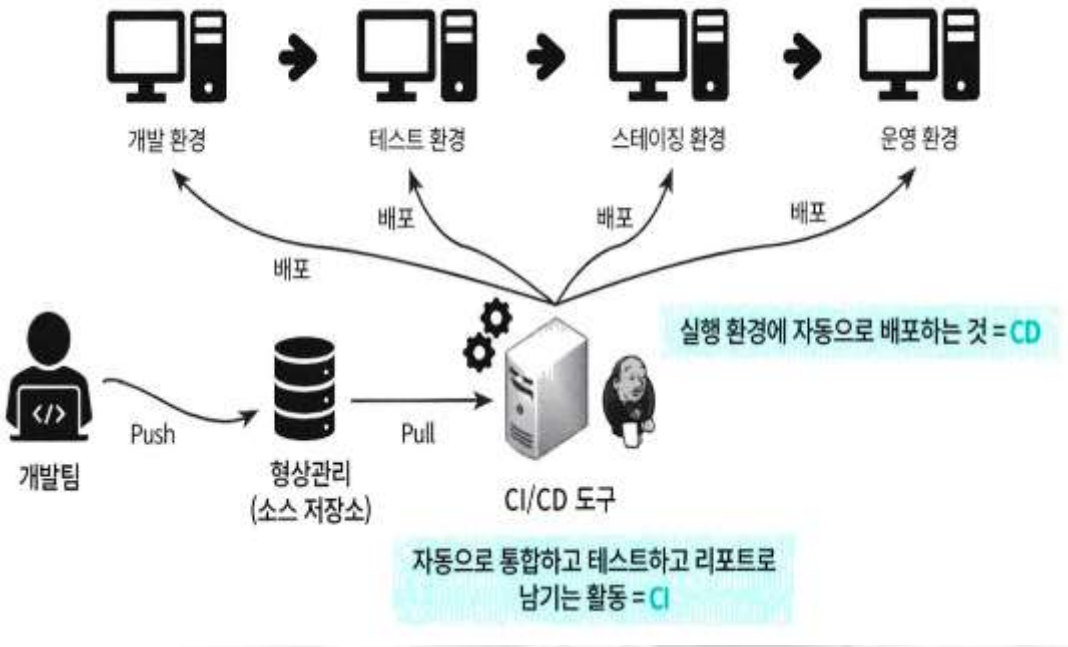
- 개발자는 개발 환경에서 애플리케이션을 완성한 후 컴파일하고 수동으로 테스트한 후 발생한 오류를 수정 한 뒤 스테이징 환경에 배포
- 개발자는 운영 환경에 배포하기 전에 스테이징 환경에서 다시 수동으로 테스트한 후 오류가 발생하면 첫 환경인 개발 환경으로 돌아가 오류를 수정한 뒤 스테이징 환경에서 다시 테스트를 수행
- 이러한 과정이 무사히 끝나면 배포 승인을 받고 승인 완료 후 배포 담당자가 애플리케이션을 운영 환경에 배포

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - 개발 지원 환경: 데브옵스 인프라 구성
      - 수동 빌드/배포 과정에는 정말 많은 시간이 소요되며 마지막에 배포를 처리하는 배포 담당자는 보통 시스템 사용률이 낮은 야간에 시스템을 장시간 멈추고 배포 작업을 진행하는 경우가 많음
      - 당연히 이러한 환경에서는 비즈니스 민첩성이 높을 수 없기 때문에 이 같은 활동을 자동화할 필요가 생기는데 특히 여러 개의 Micro Service를 배포해야 하는 환경에서는 배포가 잦을 수밖에 없기 때문에 자동화가 절실
      - 자동화된 빌드나 배포 작업을 보통 CI/CD 라고 하며 여기서 CI는 지속적 통합(Continuous Integration)을 가리키는데 원래 지속적 통합은 애자일 방법론 중 켄트 벡(Kent Back)이 만든 XP(eXtreme Programming)의 주요 프랙티스로 시작됐으며 오랜 시간이 걸리는 빌드를 매일 자동화해서 수행한다면 개발 생산성이나 소스 코드 품질이 높아진다는 경험에서 출발

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 개발 지원 환경: 데브옵스 인프라 구성
      - ◆ 지속적 통합/배포가 진행되는 과정



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 개발 지원 환경: 데브옵스 인프라 구성
      - ◆ 지속적 통합/배포가 진행되는 과정
        - ▪ 작성한 소스 코드와 그것을 테스트한 테스트 코드를 형상관리 시스템에 전송(Push)
        - ▪ 빌드 도구에서 형상관리 서버의 코드를 가져와(Pull) 통합한 다음 자동으로 빌드하고 테스트 코드를 실행해 테스트를 수행
        - ▪ 테스트 수행 결과를 리포트 문서로 기록하고 빌드된 소스코드를 스테이징 환경에 자동으로 배포
        - ▪ 테스터가 스테이징 환경에서 테스트를 수행하거나 또는 빌드 및 단위 테스트 결과를 개발자가 혹인하고 문제가 있다면 즉시 소스코드를 수정

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 개발 지원 환경: 데브옵스 인프라 구성
      - ◆ 자동으로 통합 및 테스트하고 그 결과를 리포트로 기록하는 활동을 CI라 하고 실행 환경에 내보내는 활동이 CD
      - ◆ CD는 지속적 제공(Continuous Delivery) 및 지속적 배포(Continuous Deployment)를 모두 포함
      - ◆ 지속적 제공은 빌드된 소스코드의 실행 파일을 실행 환경에 반영하기 전 단계까지 진행하는 방식이므로 아직 실행 환경에 배포되지 않았고 실행 환경에 반영하기 위해서는 승인 및 배포 담당자의 허가를 받아야 하며 배포도 수동으로 처리하는데 다소 엄격한 배포 절차를 밟는다고 할 수 있는 반면 지속적 배포는 소스코드 저장소에서 빌드한 소스코드의 실행 파일을 실행 환경까지 자동으로 배포하는 방식을 말하며 모든 영역을 자동화하는 것에 해당
      - ◆ 더 넓은 의미로 살펴보면 지속적 제공은 비즈니스 속도에 민첩하게 대응하기 위해 소프트웨어를 짧은 주기로 빈번하게 빌드, 테스트, 시장에 출시하는 모든 활동을 의미하기도 함

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 개발 지원 환경: 데브옵스 인프라 구성
      - ◆ Continuous Delivery와 Continuous Deployment

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 빌드/배포 파이프라인 설계
      - ◆ 빌드/배포되는 과정 동안 수행해야 할 태스크가 정의된 것을 빌드/배포 파이프라인이라고 하는데 빌드/배포 파이프라인은 통합 및 배포까지 이어지는 일련의 프로세스를 하나로 연계해서 자동화하고 시각화된 절차로 구축하는 것인데 어떤 단계를 거치고, 어떤 도구로 그 단계를 구성할지 결정해야 함

MSA의 이해

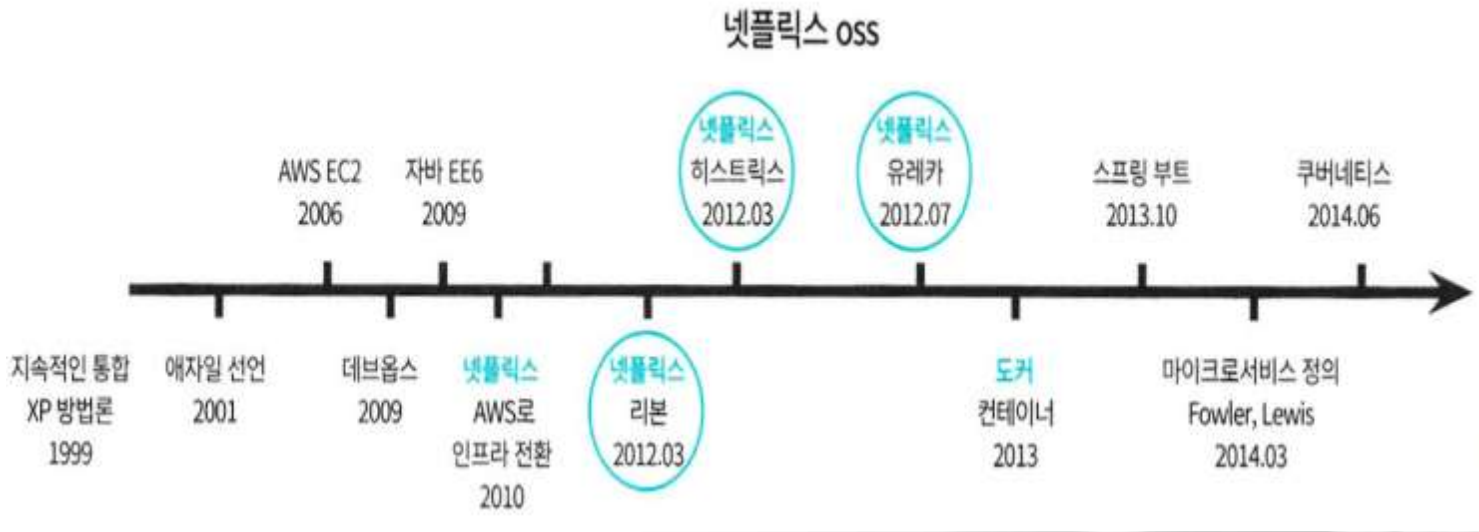
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 빌드/배포 파이프라인 설계
      - ◆ Cloud 환경이 활성화되기 이전에는 배포 환경으로 베어 메탈 하드웨어를 사용하는 경우가 있어 일부 과정에서 수동으로 처리하는 절차가 있었지만 최근 Cloud 같은 가상 환경이 대중 화되면서 완전한 자동화가 가능해졌는데 인프라 구성을 마치 프로그래밍하는 것처럼 처리하고 소수의 인원으로 많은 컨테이너 배포 처리를 할 수 있게 됐는데 이를 가리켜 Infrastructure as Code라 함
      - ◆ Infrastructure as Code를 이용하면 배포 파이프라인 절차를 완벽하게 자동화 할 수 있으며 대규모 인프라 관리를 수행할 수 있고 코드이기 때문에 쉽게 공유 및 재사용이 가능
      - ◆ 형상관리 리포지토리에서 소스코드를 가져와 빌드해서 실행 파일을 만드는 작업, 실행 파일을 실행 환경에 배 포하는 작업, 그리고 이런 작업들을 통제하고 연결해서 전 작업이 성공하면 다음 작업이 자동으로 수행되게 하는 연계 자동화 작업으로 나눌 수 있는데 이를 모두 코드로 정의 및 설정할 수 있고 이를 지원하기 위한 여러 오픈소스나 솔루션을 이용할 수 있음
      - ◆ 모든 과정을 자동화하는 과정은 매우 힘든 작업이기 때문에 일부는 자동화, 일부는 수동으로 처리할 수도 있으며 어떤 애플리케이션은 성격에 따라 이 같은 전체적인 자동화가 필요할 수도 있고 아닐 수도 있는데 이에 맞게 빌드/배포 파이프라인도 애플리케이션마다 다르게 설정할 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 빌드/배포 파이프라인 설계
      - ◆ MSA 시스템도 마찬가지인데 Micro Service는 각각 별도의 리포지토리를 가지고 있고 독립적으로 수정 및 빌드하고 배포해야 하므로 빌드/배포 파이프라인도 Micro Service 별로 별도로 설계

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● Micro Service 생태계와 운영 관리 요소의 탄생
      - ◆ Micro Service 생태계가 어떻게 발전 했는지를 나타내는 발전 흐름



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● Micro Service 생태계와 운영 관리 요소의 탄생
      - ◆ CI의 개념이 켄트 벡에 의해 XP 방법론의 프랙티스로 1999년에 소개되고 켄 트 벡, 스크럼의 켄 슈와버, 마틴 파울러 등 여러 소프트웨어 업계의 구루들이 모여 2001년에 애자일 선언을 했는데 그때부터 소프트웨어 업계가 장기적인 계획이나 단계적 프로세스 대신 빠 른 실패와 피드백을 기반으로 하는 실용적인 실천법을 선호하게 됨
      - ◆ 한편 다른 쪽에서는 아마존이 2006년에 IaaS 서비스인 EC2를 발표하며 최초로 Cloud 사업을 시작했는데 그 즈음 DVD 대여 서비스로 출발했던 넷플릭스가 스트리밍 사업을 시작했는데 얼마 후에 스트리밍 데이터베이스의 스토리지가 손실되는 대규모 서비스 장애를 겪게 되었는데 이를 계기로 넷플릭스는 한 덩어리의 Monolithic 시스템에서 Micro Service 기반의 시스템으로 전환하는 작업을 시작했는데 이때 선택한 것이 AWS의 EC2
      - ◆ Cloud 기반에서 Micro Service로 전환하는 과정은 쉽지 않았는데 우선 애플리케이션이 한 덩어리였을 때 발생하지 않았던 여러 문제점이 불거졌으며 전체 서비스를 여러 개의 서비스로 분산 구성했을 때 한 서비스에서 발생한 장애가 다른 서비스로 전파된다거나 여러 서비스에 분산된 로그를 관리해야 하는 불편함, 서비스 하나가 동작하지 않아 시스템의 일부 기능이 동작하지 않아도 그것을 알아채지 못하고 장애가 방치되는 문제들이 발생

MSA의 이해

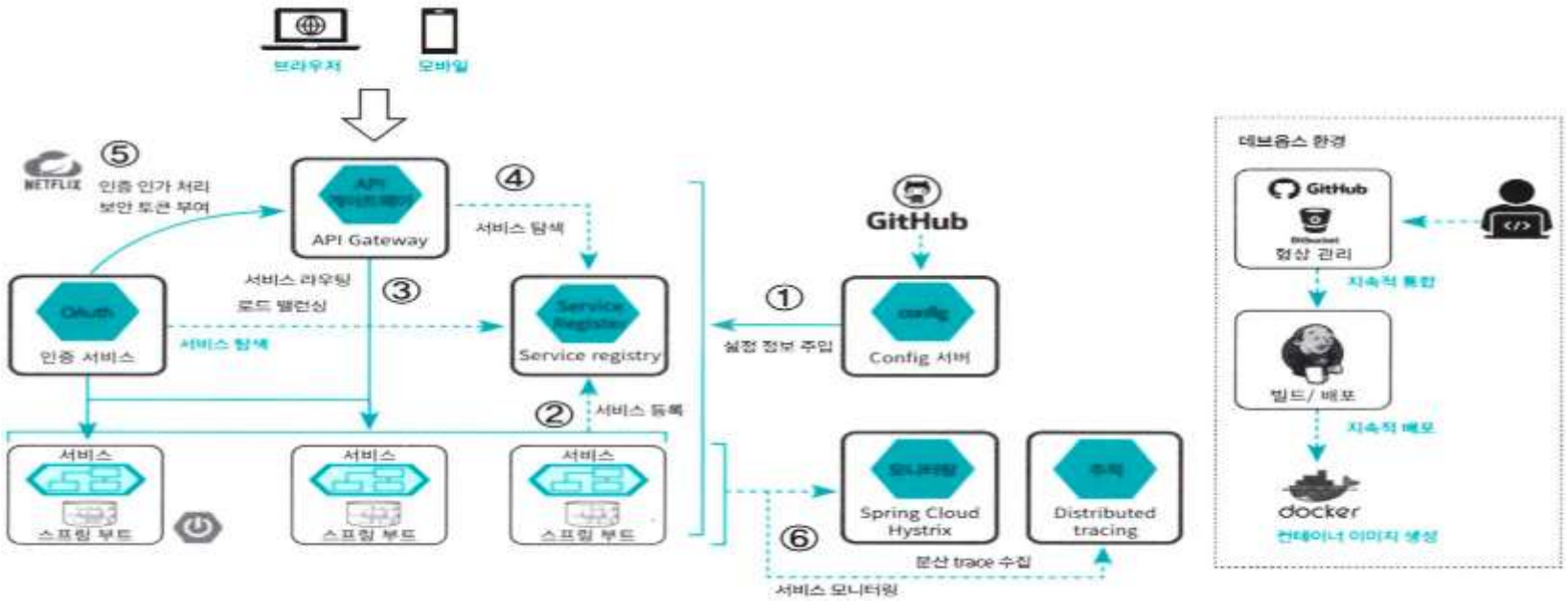
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● Micro Service 생태계와 운영 관리 요소의 탄생
      - ◆ 넷플릭스는 이러한 문제의 해결법으로 다양한 서비스와 도구를 개발하게 되었고 넷플릭스의 기술력에 의구심을 갖는 사람들에게 보란듯이 오픈소스로 공개했는데 그것이 바로 넷플릭스 OSS
      - ◆ 넷플릭스 OSS에는 여러 Micro Service 간의 라우팅과 로드 밸런싱을 위한 줄(Zuul)과 리본(Ribbon), 모니터링을 위한 히스트릭스(Hystrix), 서비스 등록을 위한 유레카(Eureka) 등이 포함돼 있는데 이러한 넷플릭스의 공유 활동이 Micro Service 기술을 발전시키는 시초가 됐는데 이를 기반으로 주요 Micro Service 운영 관리를 위한 문제 영역들이 활발히 논의되고 이를 해결한 기술들이 공유되고 자연스럽게 개발자 간 협업 및 오픈소스에 대한 기여가 이뤄졌고 이로 인해 업계가 함께 발전하게 됨
      - ◆ 이후 2013년에는 가장 유명한 컨테이너 기술인 도커가 세상에 등장했고 또한 이쯤에 스프링 진영에서는 Micro Service를 쉽게 개발할 수 있는 프레임워크인 스프링 부트를 발표하고 최근에는 컨테이너 오케스트레이션 기술인 구글의 쿠버네티스가 등장했는데 Micro Service의 개념을 마틴 파울러가 정리한 시점이 바로 이 시점
      - ◆ 간단히 정리하면 AWS의 Cloud 환경, 도커 컨테이너, 넷플릭스가 공유한 오픈소스, 스프링 프레임워크, 구글의 쿠버네티스 같은 것들이 Micro Service 생태계의 발전을 계속 이끌었고, 이러한 과정을 거쳐 Micro Service 아키텍처의 주요 문제 영역들이 논의되고 그에 대한 해결책이 제시돼 왔음을 알 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 경험으로 획득한 지혜: Micro Service 관리/운영 패턴
      - ◆ Micro Service 구축 시 발생하는 문제는 주로 시스템을 여러 개의 서비스로 구성하기 때문에 발생하는 문제로 넷플릭스가 이 문제를 해결하는 데 크게 기여했는데 넷플릭스 OSS는 넷플릭스가 Micro Service를 개발하고 운영하면서 생긴 노하우를 다른 사람들도 쉽게 사용할 수 있도록 공유한 오픈소스
      - ◆ Micro Service 생태계에 크게 도움이 됐고 특히 Micro Service 관리와 운영을 지원하는 전형적인 Micro Service 애플리케이션 패턴으로 자리 잡음
      - ◆ API 게이트웨이, 서비스 디스커버리, 모니터링, 트레이싱 등이 다수의 Micro Service를 관리 및 운영하기 위한 플랫폼 패턴으로서 넷플릭스에서 소스를 공개하고 나서 패턴으로 정착되고 나중에 이러한 패턴을 적용한 다른 여러 도구와 오픈소스들이 생겨나는 밑거름으로 작용
      - ◆ 넷플릭스 OSS를 더 쉽게 쓸 수 있도록 스프링 진영의 피보탈에서는 기존의 스프링 부트 프레임워크에서 잘 돌아갈 수 있도록 넷플릭스 OSS 모듈들을 스프링 프레임워크로 감싸서 스프링 Cloud(Spring Cloud)19라는 명칭으로 발표했는데 이를 통해 스프링 부트와 스프링 Cloud는 Micro Service를 개발하기 위한 가장 대중적인 기본 프레임워크로 자리매김
      - ◆ 스프링 부트와 스프링 Cloud를 이용하면 Micro Service 애플리케이션의 운영 환경을 쉽게 구축 할 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 스프링 Cloud: 스프링 부트 + 넷플릭스 OSS
      - ◆ 스프링 Cloud는 스프링 프레임워크를 개발하고 있는 피보탈에서 넷플릭스가 공개한 줄, 유레카, 히스트릭스, 리본 등의 넷플릭스 오픈소스를 스프링 부트 프레임워크 기반으로 사용하기 쉽게 통합한 것
      - ◆ 스프링 Cloud를 기반으로 한 아키텍쳐



MSA의 이해

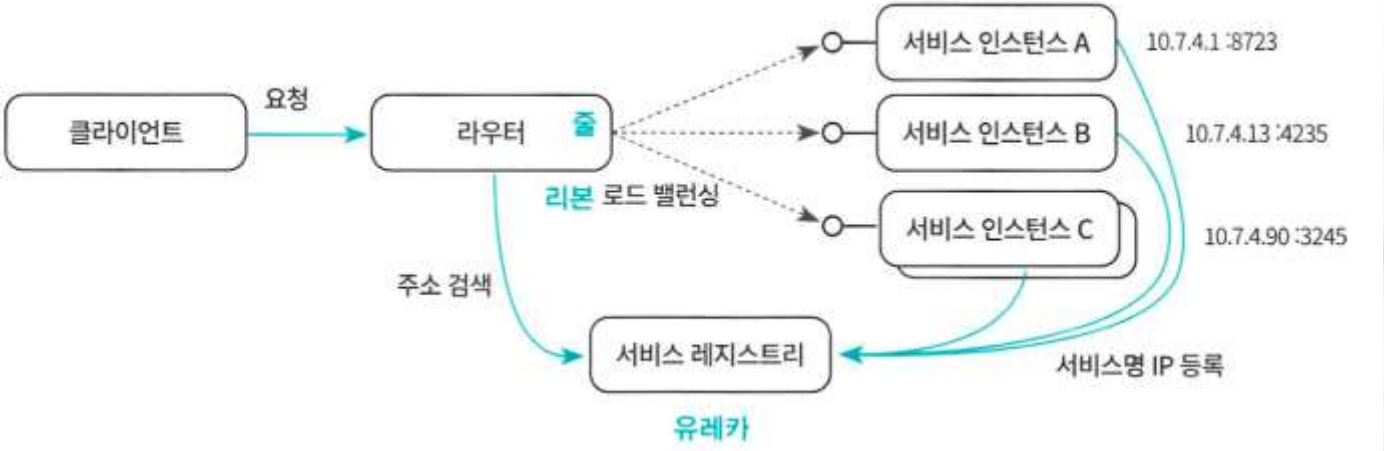
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 스프링 Cloud: 스프링 부트 + 넷플릭스 OSS
      - ◆ 데브옵스 환경과 스프링 Cloud 서비스가 있고 그것을 기반으로 업무 처리 Micro Service가 운용
      - ◆ 비니지스를 구현한 Micro Service 와 스프링 Cloud 서비스의 연계 흐름
        - ▪ 모든 Micro Service(스프링 Cloud 서비스를 포함한)는 인프라에 종속되지 않도록 데이터베이스, 파일 등에 저장된 환경 설정 정보를 형상관리 시스템에 연계된 Config 서비스에서 가져와 설정 정보를 주입한 후 Cloud 인프라의 개별 인스턴스로 로딩
        - ▪ 로딩과 동시에 서비스 레지스트리에 자신의 서비스명과 Cloud 인프라부터 할당받은 물리 주소를 매핑 해서 등록
        - ▪ 클라이언트가 API 게이트웨이를 통해 Micro Service에 접근하고 이때 API 게이트웨이는 적절한 라우팅 및 부하 관리를 위한 로드 밸런싱을 수행
        - ▪ API 게이트웨이에서 클라이언트가 Micro Service에 접근하기 위한 주소를 알기 위해 서비스 레지스트리 검색을 통해 서비스의 위치를 가져옴
        - ▪ API 게이트웨이는 클라이언트가 각 서비스에 접근할 수 있는 권한이 있는지 권한 서비스 와 연계 해 인증/인가 처리를 수행
        - ▪ 모든 Micro Service 간의 호출 흐름은 모니터링 서비스 와 추적 서비스에 의해 모니터링되고 추적

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 다양한 서비스의 등록 및 탐색을 위한 서비스 레지스트리, 서비스 디스커버리 패턴
      - ◆ Front End 클라이언트가 여러 개의 Back End Micro Service를 어떻게 호출하고 스케일 아웃을 통해 인스턴스가 여러 개로 복제된 경우 어떻게 부하를 적절히 분한 할 까 하는 것들을 위한 패턴이 서비스 디스커버리(Service Discovery) 패턴
      - ◆ 클라이언트가 여러 개의 Micro Service를 호출하기 위해서는 최적 경로를 찾아주는 라우팅 기능과 적절한 부하 분산을 위한 로드 밸런싱 기능이 제공돼야 하는데 넷플릭스의 OSS로 예를 들면 라우팅 기능은 줄(Zuiil)이 로드 밸런싱은 리본(Ribbon)이 담당
      - ◆ 라우터는 최적 경로를 탐색하기 위해 서비스 명칭에 해당하는 IP 주소를 알아야 하는데 이러한 라우팅 정보를 클라이언트가 가지고 있으면 Cloud 환경에서 동적으로 변경되는 백 엔드의 유동 IP 정보를 매번 전송받아 변경해야 하기 때문에 제3의 공간에서 이러한 정보를 관리 하는 것이 좋은데 Back End Micro Service 서비스의 명칭과 유동적인 IP 정보를 매핑해서 보관할 저장소가 필요한데 넷플릭스 OSS의 유레카(Eureka)가 그 기능을 담당하고. 이러한 패턴을 서비스 레지스트리 패턴이라 함

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 다양한 서비스의 등록 및 탐색을 위한 서비스 레지스트리, 서비스 디스커버리 패턴
      - ◆ 각 서비스 인스턴스가 로딩될 때 자신의 서비스 이름과 할당된 IP 주소를 레지스트리 서비스에 등록한 다음 클라이언트가 해당 서비스명을 호출할 때 라우터가 레지스트리 서비스를 검색해 해당 서비스의 이름과 매핑된 IP 정보를 확인한 후 호출
      - ◆ 레지스트리 서비스는 모든 Micro Service의 인스턴스의 주소를 알고 있는 서비스 매핑 저장소가 되는데 모든 Micro Service가 처음 기동할 때 자신의 위치 정보를 저장하고 서비스가 종료될 때 위치 정보가 삭제됨

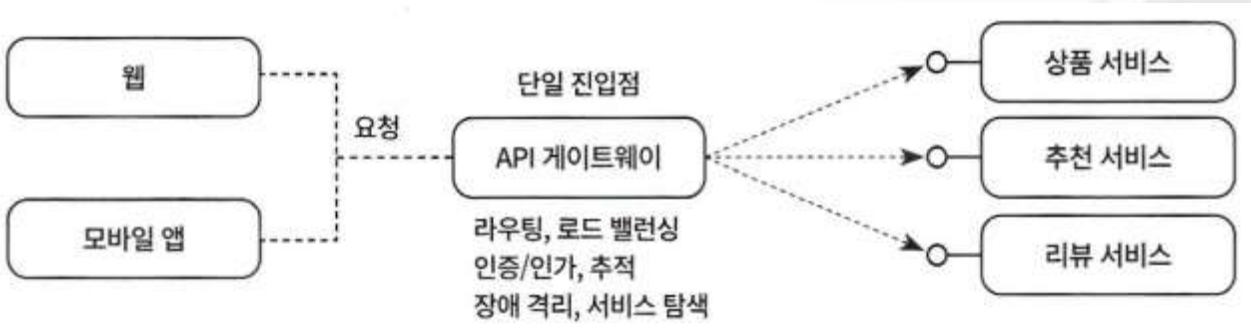


MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 다양한 서비스의 등록 및 탐색을 위한 서비스 레지스트리, 서비스 디스커버리 패턴
      - ◆ 서비스 레지스트리에는 업무 처리를 위한 Micro Service 뿐만 아니라 관리와 운영을 위한 기반 서비스의 주소도 함께 보관하는데 예를 들면, Config 서비스, 모니터 링 서비스, 추적 서비스도 모두 이름을 가지고 있기 때문에 주소를 가지고 있어야 함
      - ◆ 스프링 유레카로 레지스트리 서비스를 구현한 경우 서비스 이름 과 IP 포트 정보가 매핑된 것을 확인할 수 있음
      - ◆ 특히 다수의 인스턴스가 하나의 서비스 이름으로 등록될 때 다수의 IP 주소와 포트 정보가 매핑되고 라우터는 이 정보를 질의해서 로드 밸런싱도 할 수 있음
      - ◆ 이 패턴을 다른 솔루션에서도 제공하는데 쿠버네티스의 경우 이 같은 서비스 레지스트리, 디스커버리 기능을 자체 기능인 쿠버네티스 DNS 및 서비스 (Kubernetes Service)로 제공

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 단일 진입을 위한 API 게이트웨이 패턴
      - ◆ 여러 클라이언트가 여러 개의 서버 서비스를 각각 호출하게 된다면 매우 복잡한 호출 관계가 만들어질 것인데 이러한 복잡성을 통제하기 위한 방법이 필요
      - ◆ 한 가지 해결책은 API 게이트웨이(Gateway)
      - ◆ 다양한 클라이언트가 다양한 서비스에 접근하기 위해서는 단일 진입점을 만들어 놓으면 여러모로 효율적
      - ◆ 다른 유형의 클라이언트에게 서로 다른 API 조합을 제공할 수도 있고 각 서비스에 접근할 때 필요한 인증/인가 기능을 한 번에 처리할 수도 있으며 정상적으로 동작하던 서비스에 문제가 생겨 서비스 요청에 대한 응답 지연이 발생하면 정상적인 다른 서비스로 요청 경로를 변경하는 기능이 작동되게 할 수도 있음



MSA의 이해

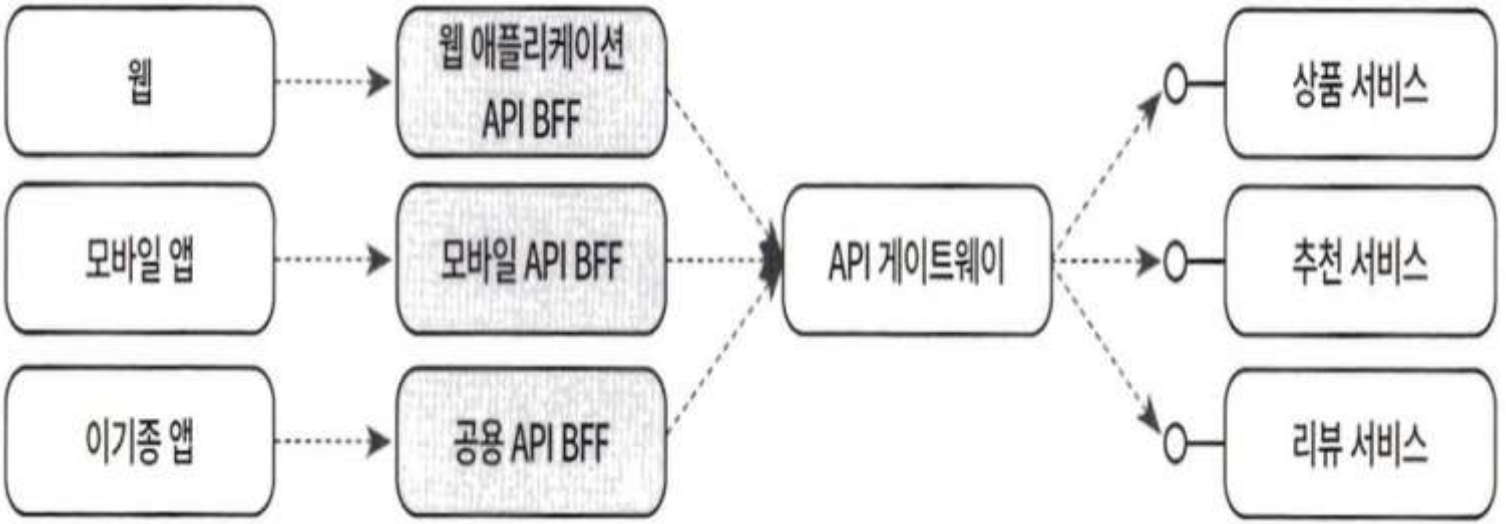
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 단일 진입을 위한 API 게이트웨이 패턴
      - ◆ 이러한 서비스 흐름 제어를 위한 서비스 라우팅 기능은 L4 같은 하드웨어 장비로 구현할 수도 있고 소프트웨어로 구현할 수도 있는데 소프트웨어로 구현할 경우 API 게이트웨이가 애플리케이션 레벨의 라우팅 기능을 수행하며 또한 여러 인스턴스로 부하를 분산하는 로드 밸런싱도 수행하고 라우팅 시 필터를 둬서 라우팅 전과 후에 각각 수행되는 선행 처리와 후행 처리, 에러 처리 등을 손쉽게 구현할 수 있음
      - ◆ API 게이트웨이는 다른 서비스와 연계해서 다음과 같은 기능을 제공
        - ▪ 레지스트리 서비스와 연계한 동적 라우팅, 로드 밸런싱
        - ▪ 보안: 권한 서비스와 연계한 인증/인가
        - ▪ 로그 집계 서비스와 연계한 로깅 - API 소비자 정보, 요청/응답 데이터
        - ▪ 메트릭(Metrics) - 에러율, 평균/최고 지연시간, 호출 빈도 등
        - ▪ 트레이싱 서비스와 연계한 서비스 추적 - 트래킹 ID 기록
        - ▪ 모니터링 서비스와 연계한 장애 격리(서킷 브레이커 패턴)

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 단일 진입을 위한 API 게이트웨이 패턴
      - ◆ 이러한 API 게이트웨이 패턴은 스프링 Cloud의 스프링 API 게이트웨이 서비스(Spring API Gateway Service)라는 제품으로 구현할 수 있는데 스프링 게이트웨이 서비스는 스프링 웹사이트(spring.io)에서 내려받아 간단한 스프링 애너테이션(Annotation – 자바 코드에 메타 데이터 형태의 주석을 달아 컴파일 혹은 런타임 시 해석되게 하는 기법으로 코딩 양을 줄이고 기술 설정 등을 추상화해서 코드 복잡도를 줄이는 효과) 설정만으로 손쉽게 적용할 수 있음
      - ◆ 다른 Cloud 플랫폼에서도 이러한 API 게이트웨이 패턴을 지원하는데 쿠버네티스의 경우 자체 기능인 쿠버네티스 서비스(Kubernetes Service)와 인그레스 리소스(ingress resources)로 제공
      - ◆ API 게이트웨이는 Front End가 Back End를 호출할 때 필요하다고 했는데 그뿐만 아니라 외부 레거시 시스템과 단일 지점에서 서로 다른 형태의 API를 연계하는 용도로도 사용되기도 함

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● BFF 패턴



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● BFF 패턴
      - ◆ 최근에는 PC뿐만 아니라 다양한 모바일 장비를 사용하기 때문에 다양한 클라이언트를 고려해야 함
      - ◆ 이처럼 다양한 클라이언트를 위해서는 특화된 처리를 위한 API 조합이나 처리가 필요한데 이를 위한 해결 방법으로 BFF(Backend for Frontend) 패턴이 있음
      - ◆ BFF 패턴은 API 게이트웨이와 같은 진입점을 하나로 두지 않고 Front End의 유형에 따라 각각 두는 패턴
      - ◆ Front End를 위한 Back End라는 의미로 BFF(Backend For Frontend)라고 부름
      - ◆ 웹을 위한 API 게이트웨이, 모바일을 위한 API 게이트웨이 등 클라이언트 종류에 따라 최적화된 처리를 수행할 수 있게 구성할 수 있는데 모바일을 위한 API만 선택해서 제공하거나 웹을 위한 API만 적절하게 제공할 수 있으며 또한 각 Front End에 대한 처리만 수행하는 BFF를 두고 이후에 통합적인 API 게이트웨이를 둠으로써 공통적인 인증/인가, 로깅 등의 처리를 통제하는 구조로 구성할 수도 있음

MSA의 이해

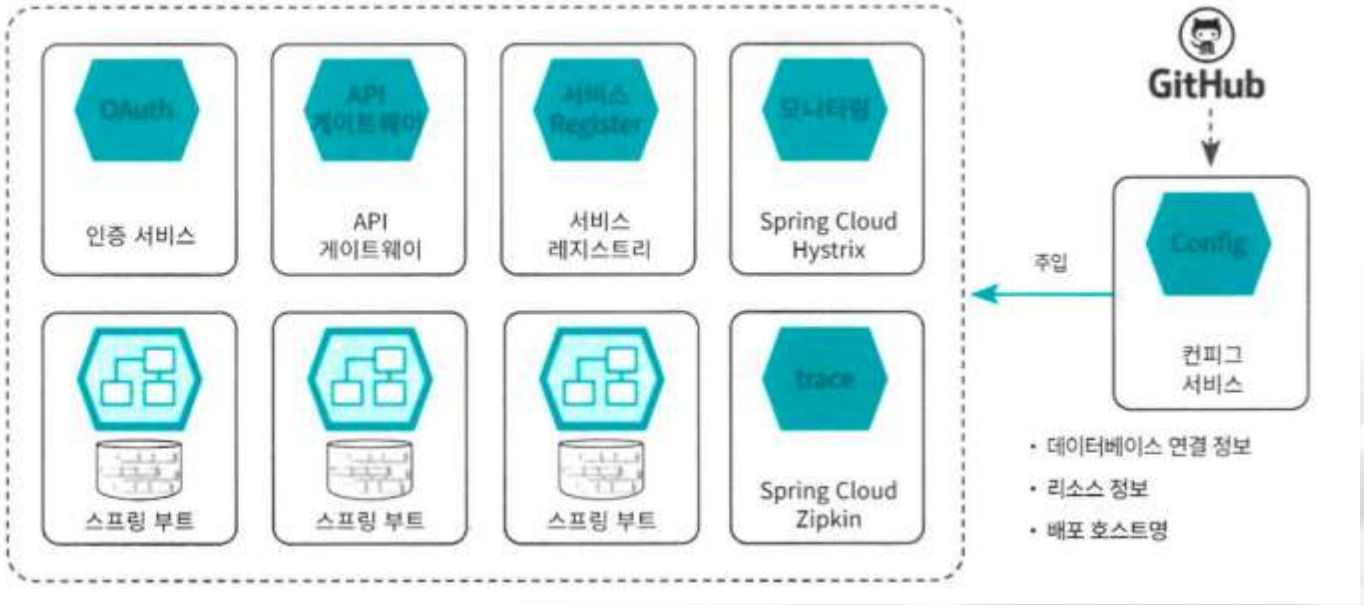
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 외부 구성 저장소 패턴
      - ◆ Cloud 인프라와 같이 유연한 인프라를 사용하는 상황에서 데이터베이스 연결 정보, 파일 스토리지 정보 같은 내용을 애플리케이션에 포함하면 변경 시 반드시 재배포해야 하는데 이 경우 서비스를 중단해야 하고 여러 Micro Service가 동일한 구성 정보를 사용하는 경우에도 일일이 변경하기가 어렵고 변경 시점에 일부 Micro Service의 구성 정보가 불일치할 수도 있기 때문에 Micro Service가 사용하는 자원의 설정 정보를 쉽고 일관되게 변경 가능하도록 관리할 필요가 있음
      - ◆ 이를 위한 방법이 외부 저장소 패턴
      - ◆ 외부 저장소는 각 Micro Service의 외부 환경 설정 정보를 공동으로 저장하는 백업 저장소
      - ◆ PaaS 솔루션 개발사인 Heroku에서 발표한 Twelve-Factor라는 Cloud Native 애플리케이션을 만들기 위한 체크리스트가 있는데. 여기서 컨피그(Config)라는 원칙을 언급했는데 이것은 애플리케이션이 배포되는 환경(스테이징, 프로덕션, 개발, 테스트 환경)이 매번 달라지기 때문에 코드에서 사용하는 환경 설정 정보는 코드와 완전히 분리되어 관리해야 한다는 원칙을 가리킴
      - ◆ Cloud에서 운영되는 애플리케이션은 특정한 배포 환경에 종속된 정보를 코드에 두면 안 된다는 원칙으로 왜냐하면 이러한 정보를 애플리케이션에 두면 배포 환경이 변경됐을 때 애플리케이션 또한 변경해야 하기 때문

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 외부 구성 저장소 패턴
      - ◆ 분리해야 할 환경 정보로는 데이터베이스 연결 정보, 배포 시 변경해야 할 호스트명, Back End 서비스의 연결을 위한 리소스 정보 등이 있으며 또한 서비스가 기동되는 개발 서버, 테스트, 운영 서버의 IP 주소와 포트 정보 등을 분리해서 환경 변수로 사용
      - ◆ 스프링 Cloud 컨피그(Spring Cloud Config)를 이용하면 이러한 환경 정보를 코드에서 분리하고 컨피그 서비스를 통해 런타임 시 주입되게 할 수 있음
      - ◆ 환경 정보는 Git 같은 별도의 형상관리 리포지토리에 보관하고 컨피그 서비스는 해당 서비스에 특정 환경에 배포될 때 적절한 환경 정보를 형상관리 리포지토리에서 가져와 해당 서비스에 주입
      - ◆ 쿠버네티스에서는 이러한 외부 구성 저장소 패턴을 쿠버네티스 컨피그맵(ConfigMap)으로 제공

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 외부 구성 저장소 패턴



MSA의 이해

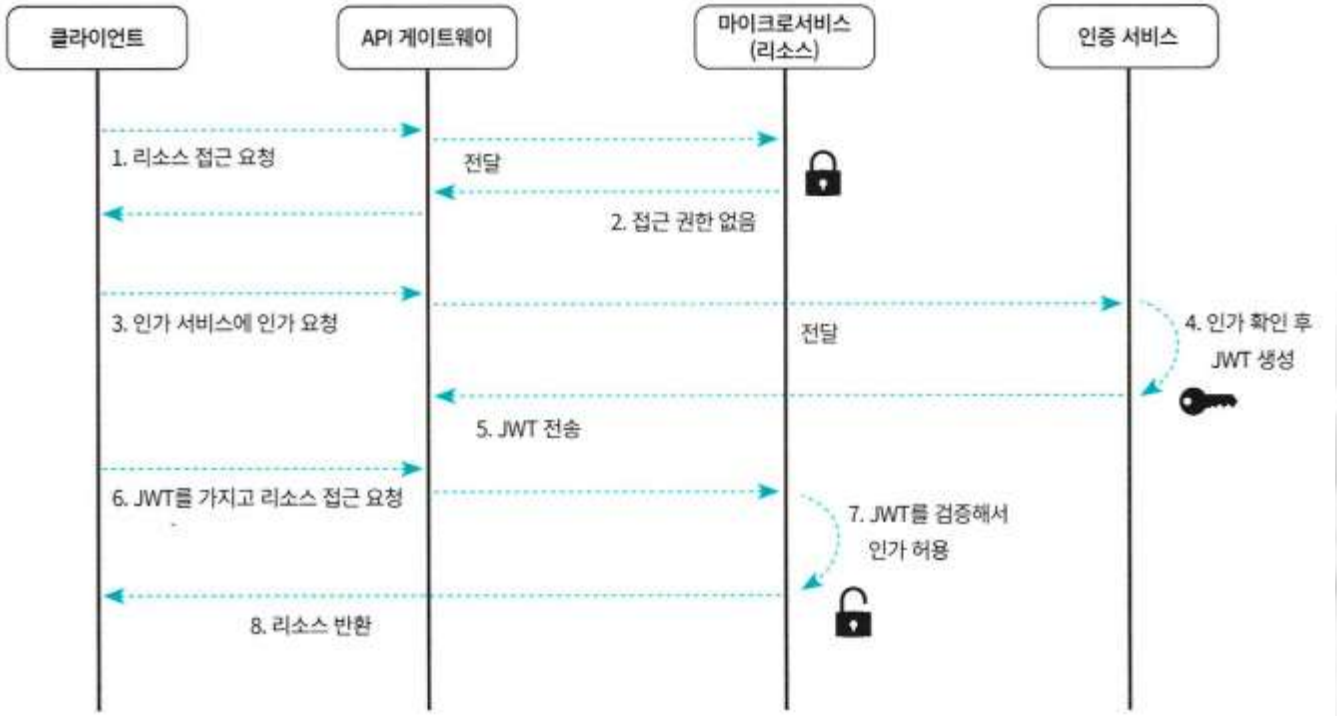
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 인증/인가 패턴
      - ◆ 각 서비스가 모두 인증/인가를 중복으로 구현한다면 비효율적
      - ◆ Micro Service 인증/인가를 처리하기 위해서는 일반적으로 다음과 같은 패턴을 활용
      - ◆ 중앙 집중식 세션 관리
        - ▪ 기존 Monolithic 방식에서 가장 많이 사용했던 방식은 서버 세션에 사용자의 로그인 정보 및 권한 정보를 저장하고 이를 통해 애플리케이션의 인증/인가를 판단하는 것으로 Micro Service는 사용량에 따라 수시로 수평 확장할 수 있고 로드 밸런싱 처리가 되기 때문에 세션 데이터가 손실될 수 있기 때문에 Micro Service는 각자의 서비스에 세션을 저장하지 않고 공유 저장소에 세션을 저장하고 모든 서비스가 동일한 사용자 데이터를 얻게 하는데 이때 세션 저장소로 보통 레디스(Redis)나 멤캐시드(Memcached)를 사용
      - ◆ 클라이언트 토큰
        - ▪ 세션은 중앙 서버에 저장되고 토큰은 사용자의 브라우저에 저장되는 방식으로 토큰은 사용자의 신원 정보를 가지고 있고 서버로 요청을 보낼 때 전송되기 때문에 서버에서 인가 처리를 할 수 있는데 JWT(JSON Web Token)는 토큰 형식을 정의하고 암호화하며 다양한 언어에 라이브러리를 제공하는 공개 표준(RFC 7519)

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 인증/인가 패턴
      - ◆ API 게이트웨이를 사용한 클라이언트 토큰
        - ▪ 사용자 인증 프로세스는 토큰 인증 프로세스와 유사
        - ▪ 차이점은 API 게이트웨이가 외부 요청의 입구로 추가된다는 것 과 인증/인가를 처리하기 위한 별도의 전담 서비스를 만들어서 다른 서비스의 인증/인가 처리를 위임할 수 있음
        - ▪ 이러한 서비스를 인증 서비스(auth service)라 하는데 API 게이트웨이와 연동해서 인증/인가를 처리
        - ▪ 인증 서비스를 이용하면 각 리소스 서비스가 자체적으로 인증/인가를 처리하지 않고 업무 처리에 집중할 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 인증/인가 패턴
      - ◆ API 게이트웨이를 사용한 클라이언트 토큰



MSA의 이해

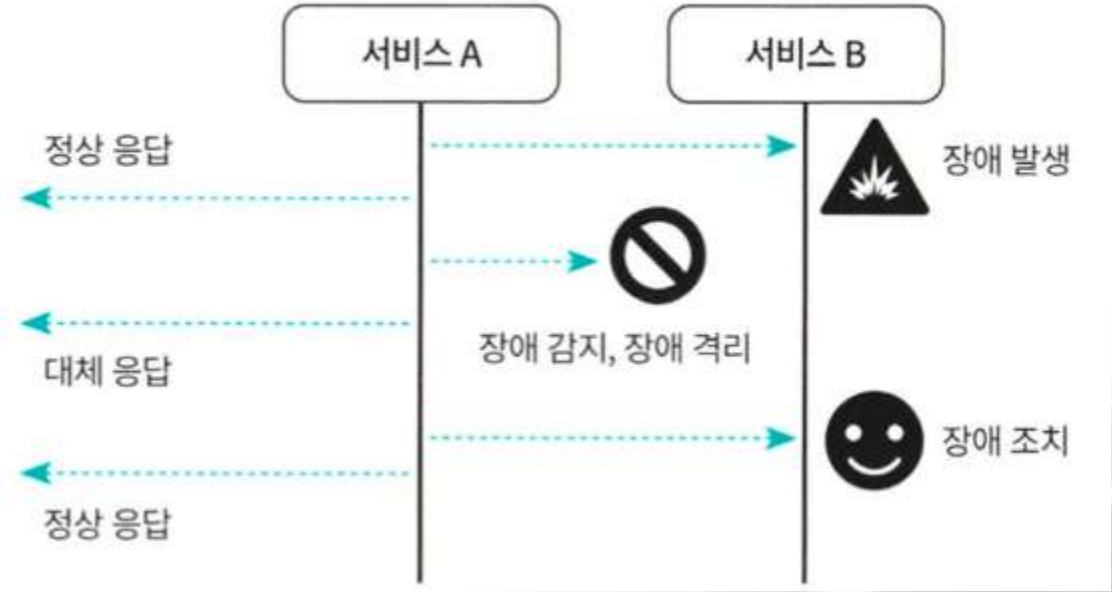
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 인증/인가 패턴
      - ◆ API 게이트웨이를 사용한 클라이언트 토큰
        - ▪ 클라이언트가 리소스 서비스에 접근을 요청하면 API 게이트웨이는 인증 서비스에게 전달
        - ▪ 인증 서비스는 해당 요청이 인증된 사용자가 보낸 것인지(인증), 해당 리소스에 대한 접근 권한이 있는지(인가) 확인하고 모두 확인하고 나면 리소스에 접근 가능한 증명서인 액세스 토큰을 발급
        - ▪ 클라이언트는 다시 액세스 토큰을 활용해 접근을 요청
        - ▪ 각 리소스 서비스는 이러한 요청이 액세스 토큰을 포함하고 있는지 판단해서 리소스에 대한 접근을 허용

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 장애 및 실패 처리를 위한 서킷 브레이커 패턴
      - ◆ 여러 서비스로 구성된 시스템에서는 한 서비스에 장애가 발생했을 때 다른 서비스가 영향을 받을 수 있는데 이때 장애가 발생한 서비스를 격리해서 유연하게 처리할 수 있는 방법이 필요한데 이를 위한 한 가지 방법이 서킷 브레이커 패턴
      - ◆ 서비스의 수가 많아지면 장점도 있지만 단점도 있는데 예를 들면 사용자가 접하는 전체 시스템은 정상적인데 특정 기능을 선택하면 즉각 에러가 발생하지도 않고 한참 동안 대기하는 상황이 발생할 수 있음
      - ◆ 사용자는 이 같은 상황이 장애인지 아닌지 파악도 안 되고 단순히 시스템이 느려졌다고 판단할 수도 있는데 이러한 상황은 정상적인 서비스가 장애가 발생한 서비스에 의존해서 서비스를 제공할 때 발생하는 상황으로서 장애가 다른 서비스로 전이된 상태
      - ◆ 시스템 과부하나 특정 서비스에 문제가 생겼을 때 자연스럽게 다른 정상적인 서비스로 요청 흐름이 변경되게 해야 하는데 그러자면 서비스 상태를 항상 실시간으로 관리해서 시각화하고 모니터링할 수 있어야 하고 특정 서비스에서 장애가 감지되면 장애가 다른 서비스로 전이되지 않게 하는 방법이 반드시 필요
      - ◆ 이를 전기회로 차단기와 비슷하다고 해서 서킷 브레이커 패턴이라고 함

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - 장애 및 실패 처리를 위한 서킷 브레이커 패턴



MSA의 이해

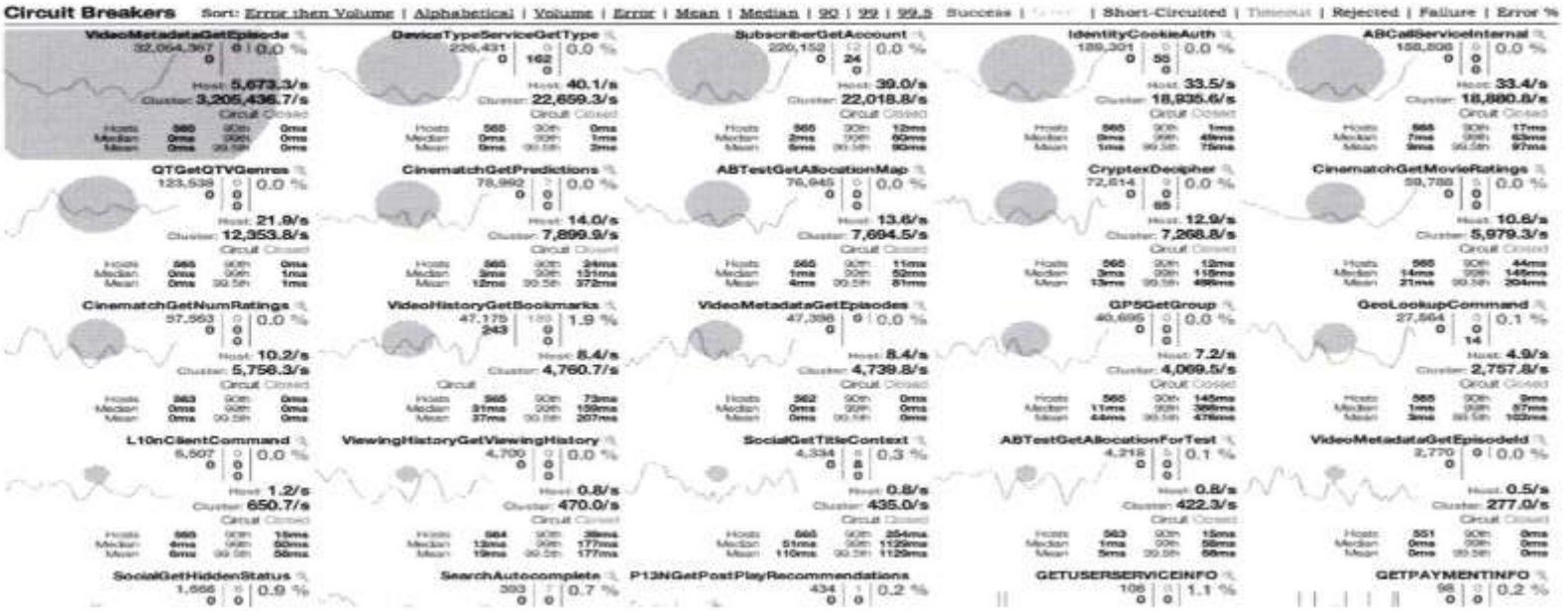
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 장애 및 실패 처리를 위한 서킷 브레이커 패턴
      - ◆ A라는 서비스가 B라는 서비스를 호출해서 자신의 서비스를 제공하는데 B 서비스에서 장애가 발생하면 이러한 동기 요청(request)의 성격상 A는 계속 기다리게 되는데 이 경우 A라는 서비스까지도 장애가 발생한 것처럼 사용자가 느끼게 됨
      - ◆ 서킷 브레이커 패턴은 이 같은 경우에 B 서비스 호출에 대한 연속 실패 횟수가 임곗값을 초과하면 회로 차단기가 작동해서 이후에 서비스를 호출하려는 모든 시도를 즉시 실패하게 만듬
      - ◆ 폴백(fallback) 메서드를 지정해 두면 장애가 발생했을 때 폴백 메서드가 자연스럽게 처리를 진행
      - ◆ 사용자는 특정 서비스에 장애가 발생했는지 눈치채지 못하고 시간이 흘러 장애가 복구됐을 때 다시 호출을 정상화하면 됨

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 모니터링 과 추적 패턴
      - ◆ 서킷 브레이커 패턴을 가능하게 하려면 각 Micro Service의 장애를 실시간으로 감지해야 하고 서비스 간의 호출이 어떤지 알아야 하는데 모니터링하고 추적하는 패턴이 필요
      - ◆ 스프링 Cloud에서는 히스트릭스라는 라이브러리를 제공하고 히스트릭스 라이브러리가 배포된 서비스를 모니터링할 수 있는 히스트릭스 대시보드(Hystrix Dashboard)를 제공함으로써 Micro Service의 요청을 실시간으로 모니터링 할수 있음

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - 모니터링 과 추적 패턴
      - 각 요청이 차지하는 트래픽이 원형으로 표현되는데 원이 클수록 트래픽이 많다는 의미
      - 서비스 성능에 문제가 발생하면 서킷이 오픈되어 서킷 브레이커가 발동하고 적색으로 표현되는데 이를 통해 Micro Service를 모니터링 하다가 문제가 발생했을 때 적절한 조치를 취할 수 있게 됨



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 모니터링 과 추적 패턴
      - ◆ 모니터링과 함께 각 서비스 트랜잭션의 호출을 추적하면 Micro Service 운영에 매우 유용 – 분산 트레이싱 서비스
      - ◆ 트위터에서 공개한 집킨(Zipkin)이라는 오픈소스 프로젝트의 대시보드로서 분산된 서비스 간의 호출이나 지연 구간별 장애 포인트를 확인할 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 모니터링 과 추적 패턴
      - ◆ 서비스 API를 선택하면 그림의 왼쪽과 같이 각 API가 다른 API를 어떻게 호출하는지 호 출 시의 반응 시간이나 지연 구간 등을 확인할 수 있다. 또한 정적인 다이어그램을 통해 전체적 인 API 간의 호출 빈도도 확인할 수 있음
      - ◆ 운영자가 이 다이어그램을 실시간으로 보다가 만약 호출 빈도가 너무 많은 API나 반응 시간이 높은 API가 있다면 이를 개선할 수 있을 것

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 중앙화 된 로그 집계 패턴
      - ◆ Micro Service가 사용량에 따라 탄력적으로 변화하면서 언제든지 인스턴스가 생성/삭제되는 과정에서 로컬 로그가 초기화될 수 있음
      - ◆ Twelve-Factor의 로그(Logs) 원칙을 보면 로그를 이벤트 스트림(event streams)으로 처리하라고 함
      - ◆ 로그는 시작과 끝이 고정된 것이 아니라 서비스가 실행되는 동안 계속 흐르는 흐름이라는 의미이고 서비스는 스트림의 전달이나 저장에 절대 관여하지 않아야 한다고 하는데 왜냐하면 로그를 전달 및 저장하는 메커니즘 자체가 특정 기술이나 인프라에 의존할 수밖에 없고 이러한 메커니즘을 직접 Micro Service에서 구현한다면 유연성이 떨어지기 때문인데 그래서 필요한 것이 중앙화된 로그 집계 패턴
      - ◆ 서비스에서 발생한 이벤트 스트림 형태의 로그를 수집하고 살펴볼 도구가 필요한데 대표적으로 많이 쓰이는 기술이 ELK 스택
      - ◆ ELK 스택은 엘라스틱서치(Elasticsearch), 로그스태시(Logstash), 키바나(Kibana)라는 세 가지 오픈소스 프로젝트를 기반으로 데이터 분석 환경을 구성한 것
      - ◆ 엘라스틱서치는 분석 엔진이고 로그스태시는 서버 측 로그 집합기이며 키바나는 시각적으로 로그 내역을 보여주는 대시 보드

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 중앙화 된 로그 집계 패턴

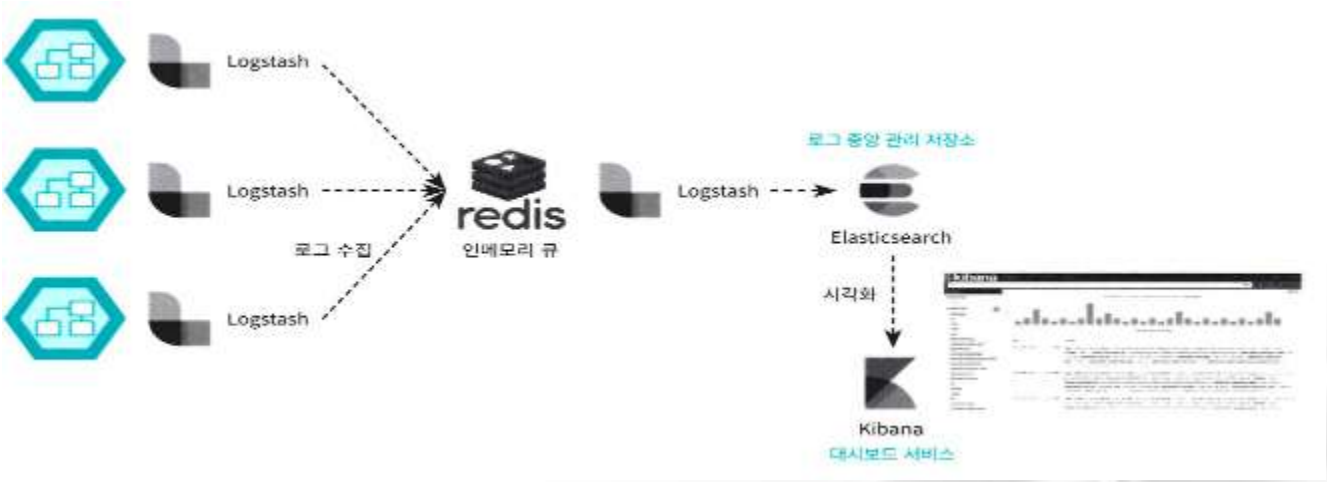
Elasticsearch	**분산형 검색 · 분석 엔진** 정형, 비정형, 위치 정보, 메트릭 등 원하는 방법으로 검색을 수행 · 결합 가능
Logstash	**로그 집합기** 데이터 처리 파이프라인. 다양한 소스에서 동시에 데이터를 수집해 변환한 뒤 특정 보관소로 데이터를 보냄
Kibana	**시각화** 히스토그램, 막대 그래프, 파이차트 등 표현. 위치 데이터, 시계열 분석, 그래프 관계 탐색 등 지원

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - 중앙화 된 로그 집계 패턴
      - ELK 스택을 이용하면 각 서비스의 인스턴스 로그를 집계해서 중앙에서 집중 관리할 수 있으며 손쉽게 특정 로그를 검색 및 분석할 수 있으며 또한 특정 메시지가 로그에 나타나거나 특정 예외가 발생할 때 운영자나 개발자에게 직접 통보하게 할 수도 있음
      - Micro Service의 로그 수집을 위한 ELK 스택 기반 아키텍처
        - 각 서비스에 로그 스태시가 설치되어 각 로그를 수집해서 레디스 저장소에 보내고 다른 서비스에서는 엘라스틱서치와 키바나로 로그 중앙 관리 저장소와 대시보드 서비스를 각각 구축

MSA의 이해

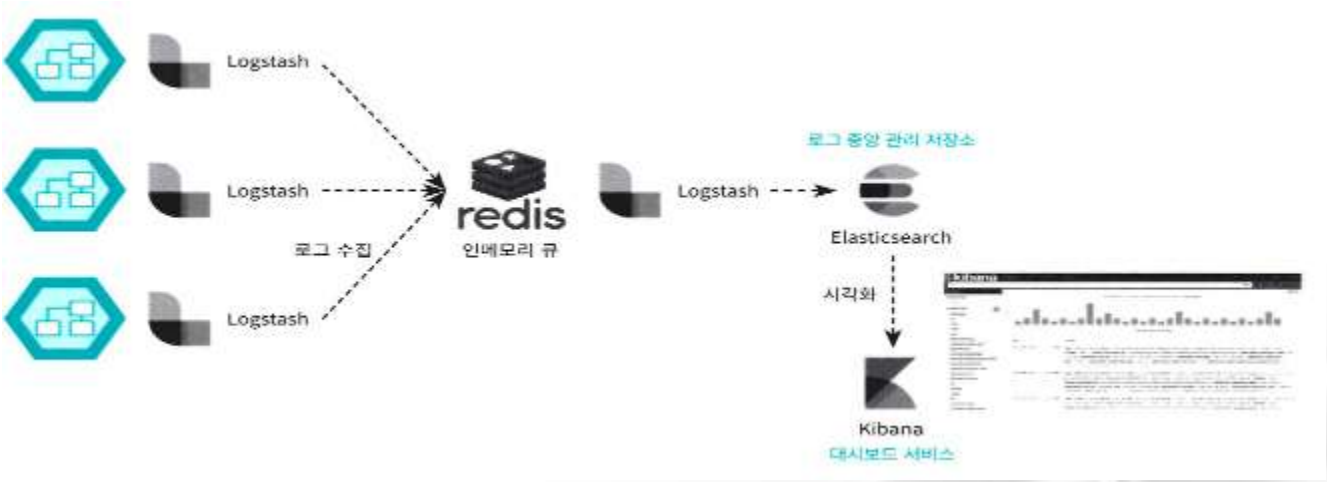
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 중앙화 된 로그 집계 패턴
      - ◆ Micro Service의 로그 수집을 위한 ELK 스택 기반 아키텍처



- ▪ Micro Service에서 보낸 로그가 중앙 레디스에 쌓이면 레디스에서 중앙 관리 저장소에 로그를 보내고 이 로그 저장소에 엘라스틱서치 엔진이 로그를 인덱싱하고 해당 로그 정보가 키바나 대시보드를 통해 보여지는데 중간에 레디스 데이터베이스를 둔 까닭은 Micro Service의 로그스태시가 바로 로그 저장소에 로그를 보낼 수 있지만 로그 스트림이 너무 몰리면 로그 저장소 서비 스에도 성능 문제가 생기기 때문에 중간에 임시 저장소를 추가한 것

MSA의 이해

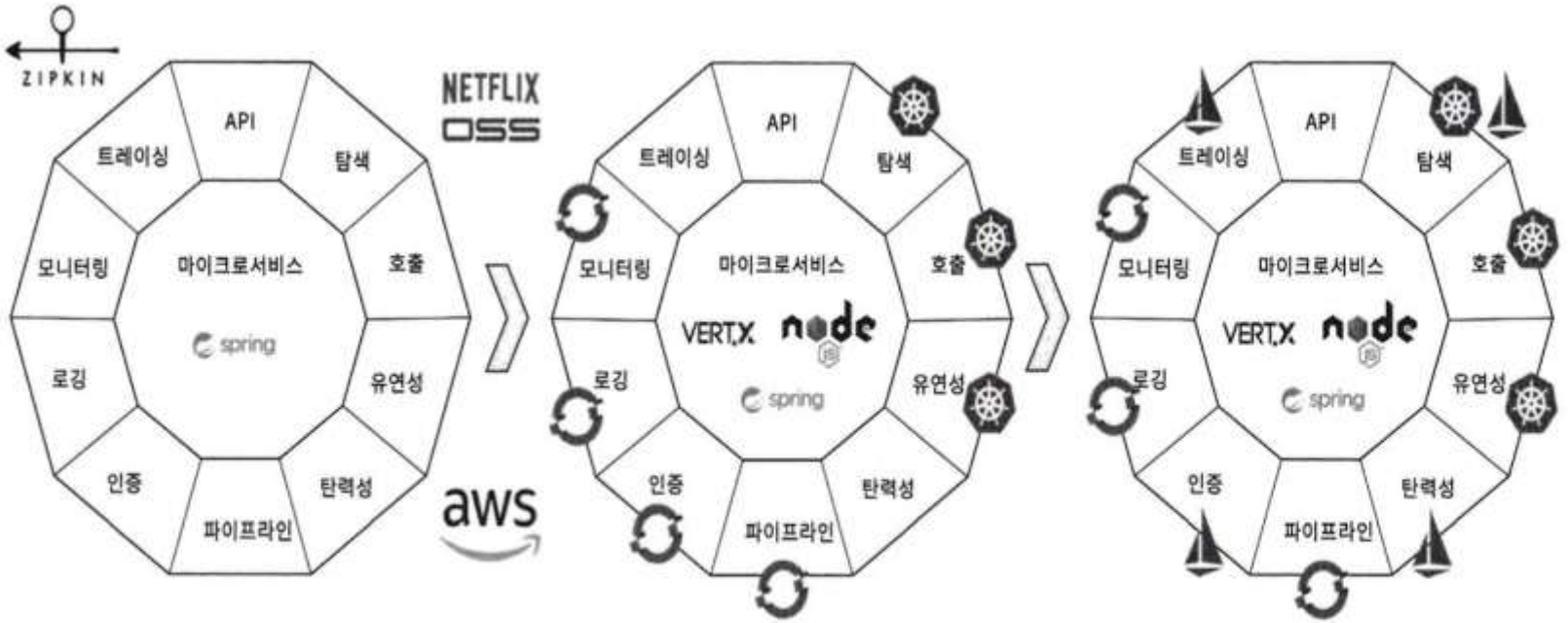
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 중앙화 된 로그 집계 패턴
      - ◆ Micro Service의 로그 수집을 위한 ELK 스택 기반 아키텍처



- ▪ Micro Service에서 보낸 로그가 중앙 레디스에 쌓이면 레디스에서 중앙 관리 저장소에 로그를 보내고 이 로그 저장소에 엘라스틱서치 엔진이 로그를 인덱싱하고 해당 로그 정보가 키바나 대시보드를 통해 보여지는데 중간에 레디스 데이터베이스를 둔 까닭은 Micro Service의 로그스태시가 바로 로그 저장소에 로그를 보낼 수 있지만 로그 스트림이 너무 몰리면 로그 저장소 서비 스에도 성능 문제가 생기기 때문에 중간에 임시 저장소를 추가한 것

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - MSA 기술 변화 흐름

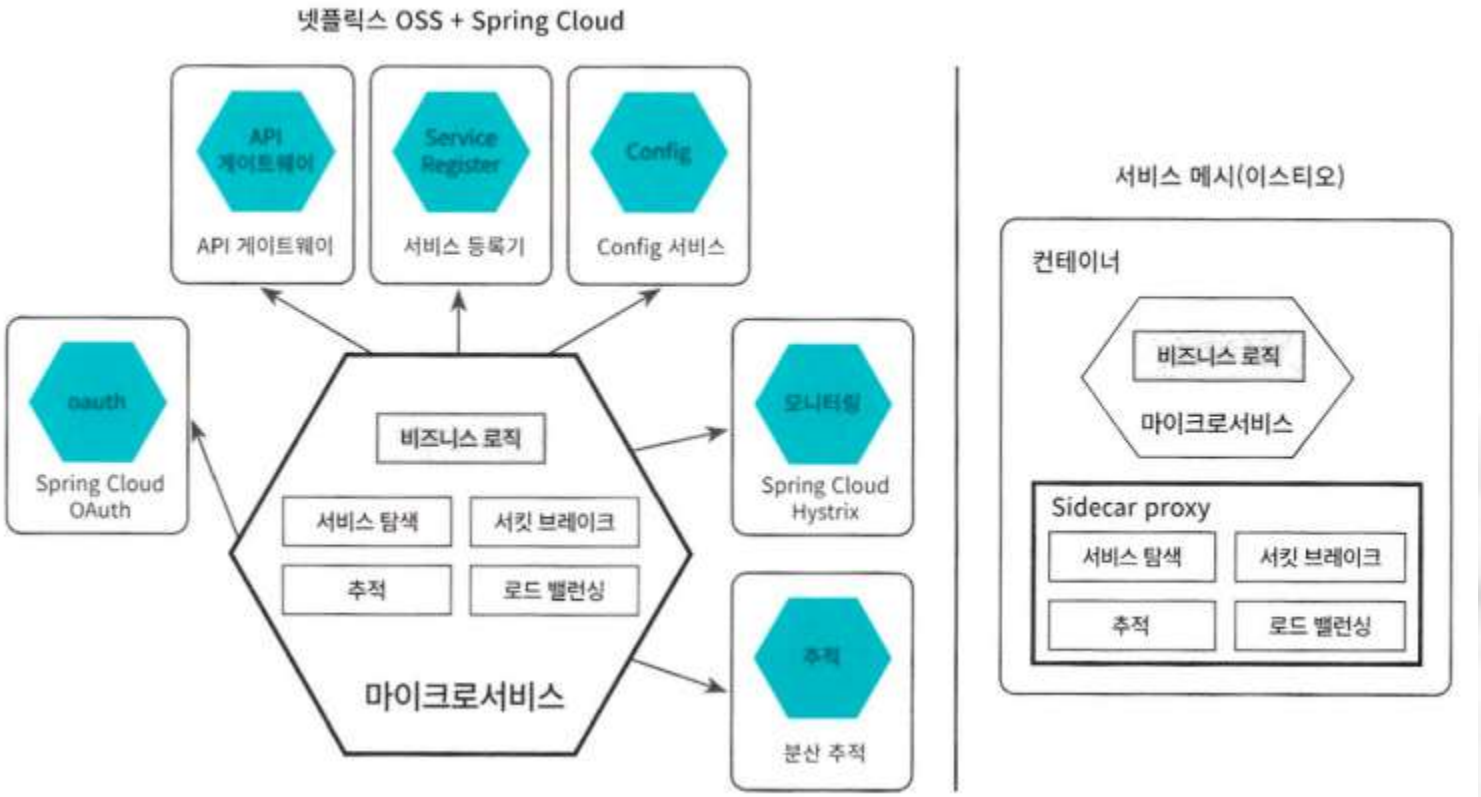


MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● MSA 기술 변화 흐름
      - ◆ 앞에서 언급한 패턴들은 모두 Monolithic 시스템이 여러 조각의 Micro Service로 나눠져서 발생하는 문제들을 해결하는데 추적, 모니터링, 로깅, 인증, 탐색, 유연성, 탄력성 등이 여기에 해당
      - ◆ 이러한 문제를 해결하기 위해 초기 MSA 생태계에서는 넷플릭스 OSS나 스프링 Cloud를 이용해 각각의 서비스를 별도로 만들어서 해결하거나 유연성처럼 수평 확장이 필요한 요소는 AWS IaaS 서비스를 이용해 해결했는데 문제마다 상이한 기술로 해결할 수 밖에 없었음
      - ◆ 이후 여러 문제의 해결책을 한꺼번에 제공하는 솔루션들이 등장했는데 바로 쿠버네티스 나 오픈시프트(OpenShift) 같은 제품으로 기존에 넷플릭스 OSS 기반의 여러 서비스로 처리했던 문제들을 묶어서 하나의 쿠버네티스 또는 오픈시프트로 해결할 수 있게 됨
      - ◆ 특히 인프라 유연성을 보장하기 위해 AWS IaaS의 인프라 차원에서 해결했던 역할을 쿠버네티스가 소프트웨어 차원, 즉 컨테이너의 레플리카 기술로 탐색, 호출 문제와 함께 통합해서 지원하면서 쿠버네티스가 각광받고 있음다.
      - ◆ 최근 동향은 쿠버네티스에 덧붙여 이스티오 기술이 함께 사용되고 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 매시 패턴



MSA의 이해

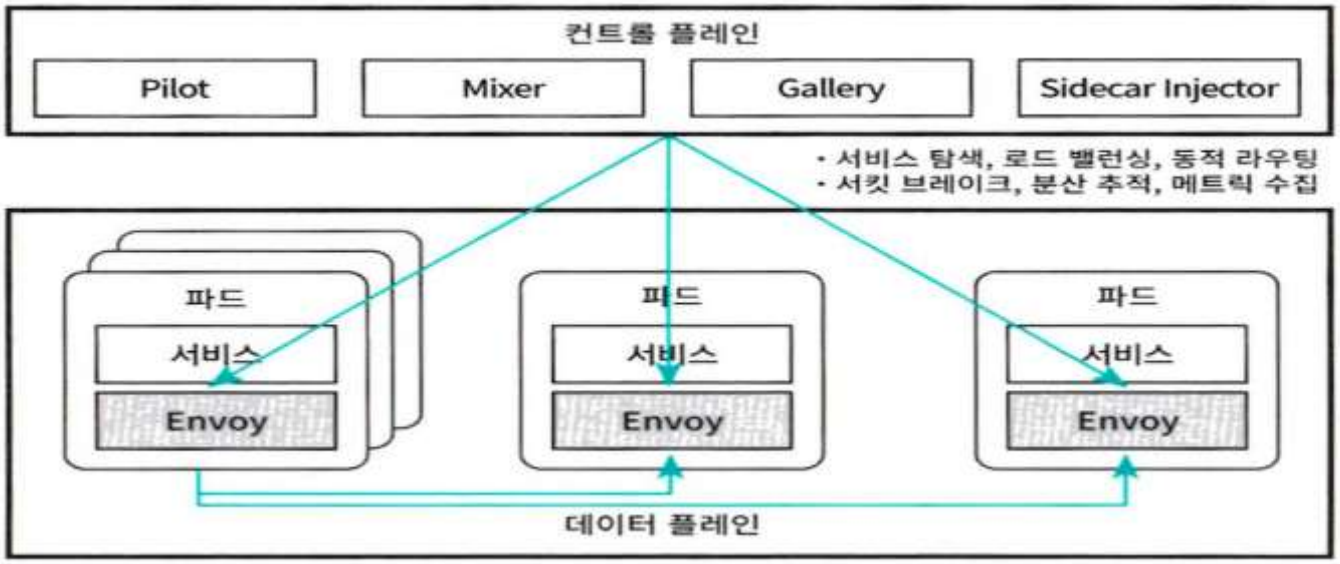
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 매시 패턴
      - ◆ 초창기 MSA 기술인 넷플릭스 OSS나 스프링 Cloud 기반의 서비스를 구축 및 운용할 때의 문제점은 API 게이트웨이, 서비스 레지스트리. 컨피그 서비스와 같이 운영 관리를 위한 여러 개 의 기반 서비스를 별도로 각각 만들어야 한다는 번거로움과 더불어 업무 처리 Micro Service에 스프링 Cloud 서비스를 사용하기 위한 라이브러리를 비즈니스 로직과 함께 탑재해야 한다는 점이었음
      - ◆ 기능 구현에 집중해야 하는 Micro Service 입장에서 이러한 코드까지 관리 및 운영해야 한다면 번거로울 수밖에 없음
      - ◆ 스프링 Cloud는 자바 기반이기 때문에 Micro Service가 자바 외의 다른 언어로 폴리글랏하게 구현된 경우에는 스프링 Cloud 서비스를 아예 사용할 수조차 없음
      - ◆ 최근에는 MSA 문제 영역 해결을 위한 기능(서비스 탐색, 서킷 브레이크, 추적, 로드 밸런싱 등)을 비즈니스 로직과 분리해서 네트워크 인프라 계층에서 수행하게 하는 서비스 메시(service mesh) 패턴이 선호되고 있음
      - ◆ 서비스 메시는 인프라 레이어로서 서비스 간의 통신을 처리하며 앞에서 언급한 여러 문제 해결 패턴을 포괄

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 매시 패턴
      - ◆ 돛단배 모양의 아이콘이 서비스 메시 패턴의 대표적 구현체인 구글의 이스티오(Istio)를 나타냄
      - ◆ 이스티오는 애플리케이션이 배포되는 컨테이너에 완전히 격리되어 별도의 컨테이너로 배포되는 사이드카(Sidecar) 패턴을 적용해서 서비스 디스커버리, 라우팅, 로드 밸런싱, 로깅, 모니터링, 보안, 트레이싱 등의 기능을 제공
      - ◆ 기본적으로 이스티오는 쿠버네티스에 탑재되어 이러한 서비스 메시 기능을 지원하며 스프링 Cloud와 넷플릭스 OSS를 이용한 경우에는 스프링 Cloud로 각 서비스를 먼저 구축하고 Micro Service 애플리케이션 자체도 코드 내부에 스프링 Cloud 사용을 위한 클라이언트 코드가 탑재돼야 하지만 서비스 메시를 적용하는 경우에는 오른쪽 그림처럼 Micro Service마다 함께 배포되는 사이드카 프락시에 운영 관리를 위한 기능이 별도로 담겨있기 때문에 Micro Service는 순수 비즈니스 로직에 집중할 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 매시 패턴
      - ◆ 컨트롤 플레인(Control Plain) 기능에 의해 중앙에서 통제되며 사이드카끼리 통신해서 관련 운영 관리 기능을 제공하는 모습을 보여주는데 이를 통해 Micro Service의 비즈니스 로직과는 완벽하게 독립적으로 운영
      - ◆ 쿠버네티스의 컨테이너 단위인 파드(pod)에 서비스 컨테이너와 사이드카 구현체인 엔보이(Envoy) 컨테이너가 함께 배포된 것을 볼 수 있음



MSA의 이해

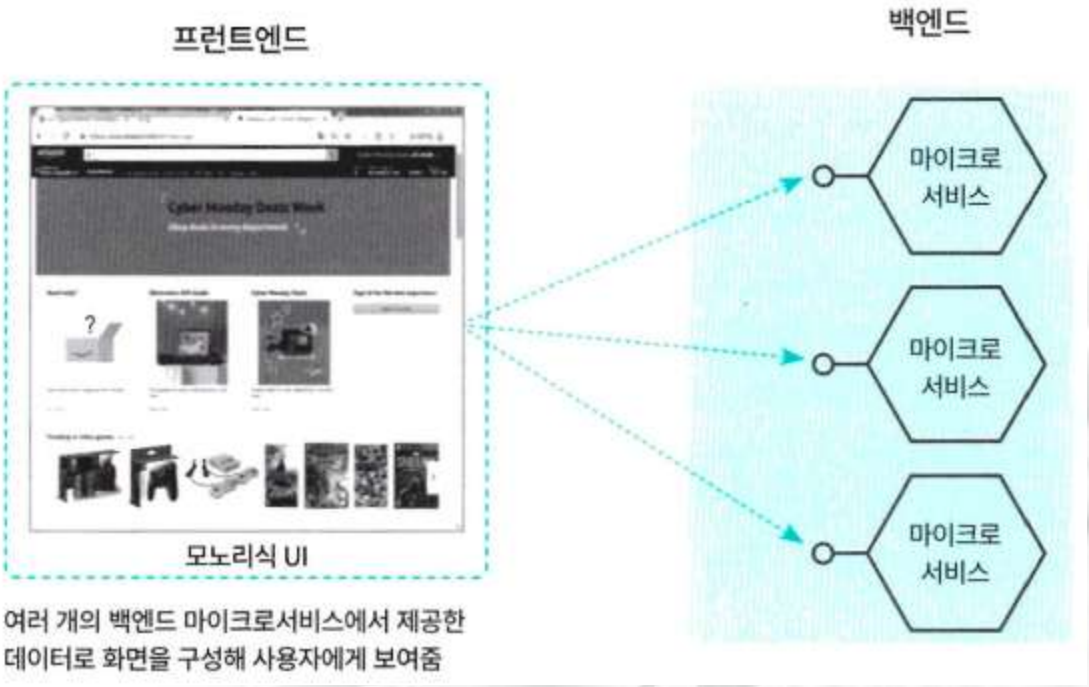
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 서비스 매시 패턴
      - ◆ 이스티오는 다음과 같이 스프링 Cloud와 넷플릭스 OSS에서 제공했던 대부분의 기능을 모두 제공하면서 동시에 차별점도 있음
        - ▪ 주요기능
          - 트래픽 관리(Traffic Management): 동적 라우팅. 로드 밸런싱
          - 보안: 보안 통신 채널(TLS), 인증/인가/암호화
          - 관측성(Observability): 메트릭, 분산 트레이싱, 로깅
        - ▪ 스프링 Cloud 및 넷플릭스 OSS와의 차별점
          - 애플리케이션 코드의 변경이 거의 없는데 스프링 Cloud나 넷플릭스 OSS 기반은 비즈니스 로직과 함께 코드로 표현돼야 하지만 이스티오는 완전히 사이드카로 격리되며 yaml 파일과 같은 설정 파일에 의해 정의됨
          - 폴리글랏 애플리케이션도 지원하는데 스프링 Cloud나 넷플릭스 기반은 자바 언어만 지원하나 이스티오는 각 Micro Service를 다른 언어(자바, Node.js, C#)로 작성한 경우에도 지원 가능
          - 이스티오는 쿠버네티스와 완벽하게 통합된 환경을 지원

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ Micro Service 운영과 관리를 위한 플랫폼 패턴
    - ● 쿠버네티스와 이스티오는 최근에 나온 기술이지만 모든 상황에 항상 적합하고 옳은 것은 아니며 각 문제 영역을 정확히 이해하고 시스템 상황에 맞게 적절한 기술을 적용하는 것이 중요

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - Monolithic Front End
      - ◆ 여러 API를 호출하고 조합한 후 화면으로 구성해서 보여주는 방식으로 이 경우 고민은 Front End가 한 덩어리일 경우 과연 Micro Service 기반 시스템의 장점인 서비스의 독립적인 변경과 배포가 가능하겠느냐는 것

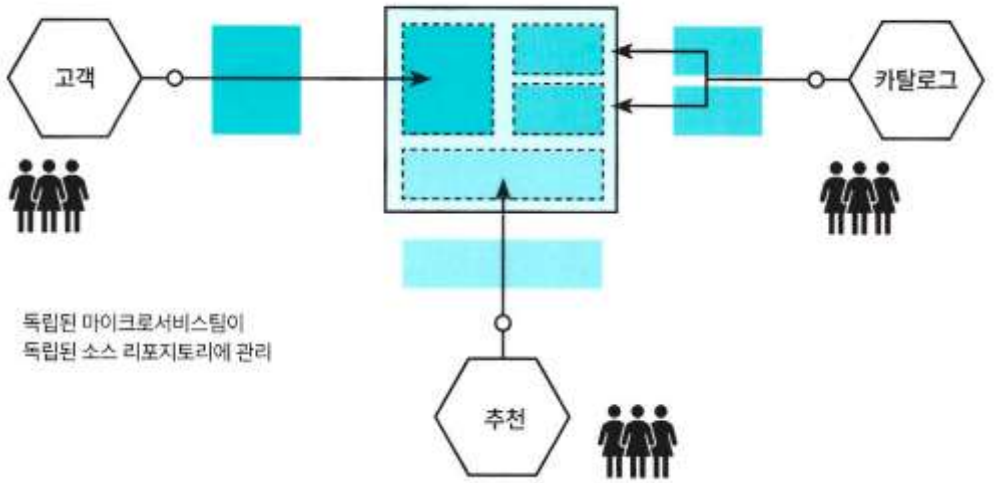


MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Monolithic Front End
      - ◆ 하나의 업무 기능은 보통 Front End와 Back End의 연계로 구현
      - ◆ 업무 기능 하나가 변경되어 재배포해야 할 상황을 생각해 보면 Back End는 수정해서 하나의 서비스로 독립적으로 배포 가능하지만 Front End는 하나의 덩어리이기 때문에 변경되지 않은 다른 기능들도 함께 빌드되고 배포해야 하기 때문에 이전의 Back End가 Monolithic였을 때 겪었던 문제(독립적인 기능 변경 및 배포 불가, 독립적인 기능 확장 불가)를 Front End의 Monolithic 서비스도 동일하게 겪을 수밖에 없는 것

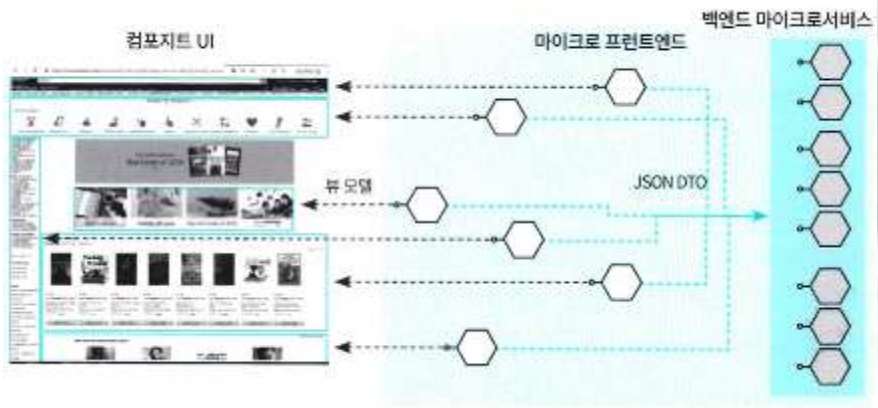
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● UI 컴포지트 패턴 또는 마이크로 Front End
      - ◆ Front End도 Back End Micro Service처럼 기능별로 분리하고 이를 조합하기 위한 프레임(frame) 형태의 부모 창을 통해 각 Front End를 조합해서 동작하게 작성
      - ◆ 부모 서비스는 틀만 가지고 있고 실제 각 기능 표현은 마이크로 Front End 조각이 구현하게 하는 방식으로 마이크로 Front End들은 비즈니스 구현을 위해 여 러 개의 Back End Micro Service API를 호출



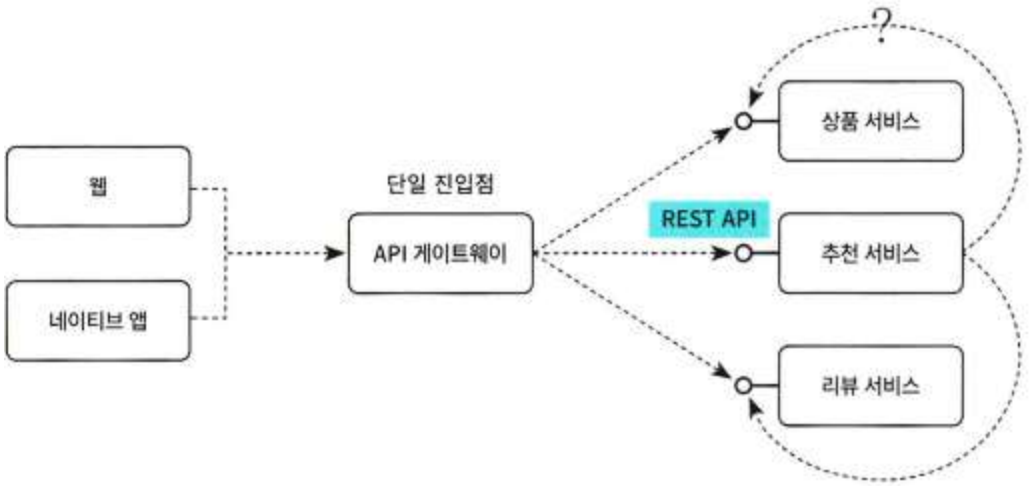
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● UI 컴포지트 패턴 또는 마이크로 Front End
      - ◆ 고객 서비스 팀은 별도의 독립된 소스 리포지토리에 Back End Micro Service와 마이크로 프런트를 관리하고 이를 독립적으로 빌드 및 배포할 수 있기 때문에 자율적으로 서비스를 개선할 수 있음
      - ◆ 아마존의 메인 화면은 여러 개의 조각으로 구성 돼 있고 각 조각은 여러 개의 마이크로 Front End의 조합으로 서비스를 제공
      - ◆ 하나의 기능을 변경했을 때 이를 제공하는 마이크로 Front End 와 Back End를 구성하는 Micro Service가 모두 변경되고 배포되는데 배포 시 잠시 반응이 없는 경우나 일부 Micro Service에 장애가 발생한 경우에도 메인 화면을 제공하는 부모 창은 사용자가 알아채지 못하게 화면 구성을 재배열하는 역할까지 수행
      - ◆ 아마존닷컴의 서비스 개선 속도는 이 같은 유연한 UI 구성을 통해서도 보장



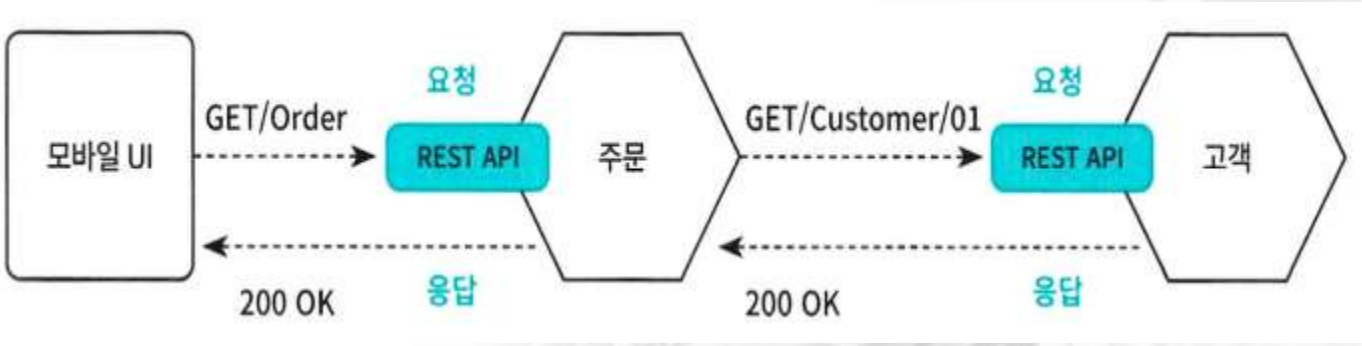
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 동기 통신 방식
        - ▪ 동기 호출 방식은 클라이언트에서 서버 측에 존재하는 Micro Service REST API를 호출할 때 사용되는 기본 통신 방법이며 다양한 클라이언트 채널 연계나 라우팅 및 로드 밸런싱을 원활하게 하기 위한 방법으로 중간에 API 게이트웨이를 둘 수 있음
        - ▪ 웹이나 앱에서 API 게이트웨이를 통해 상품, 추천, 리뷰 Micro Service를 동기 호출하는 구성



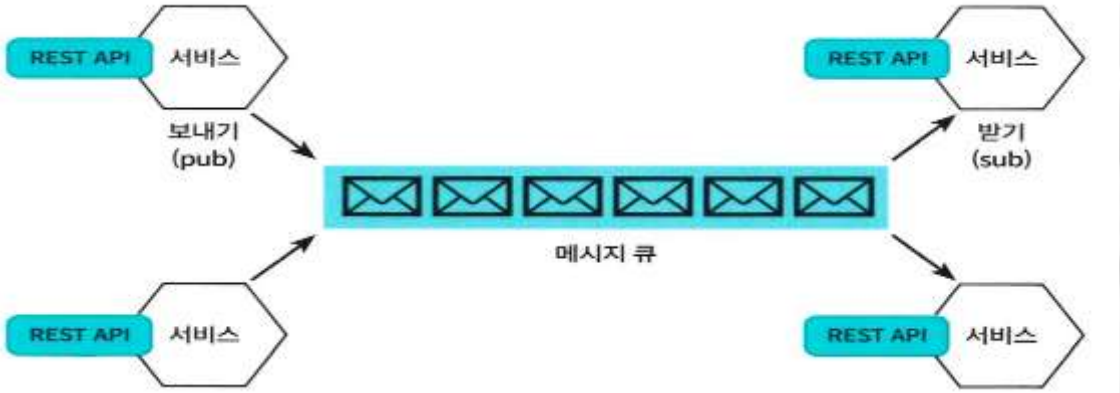
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 동기 통신 방식
        - ▪ Back End Micro Service 간의 호출에서도 REST API 같은 동기식 호출을 사용할 수 있는데 모바일 UI 고객의 주문 내역을 확인하기 위해 주문 서비스에 HTTP GET 방식의 요청을 보내면 주문 서비스는 고객 정보를 확인하기 위해 고객 서비스에 GET 방식의 동기 호출을 수행하고 그에 따라 바로 응답이 발생하고 성공 시 200 OK라는 성공 코드를 받아오도록 만들 수 있는데 이처럼 요청하면 응답이 오는 직관적인 방식이기 때문에 가장 많이 쓰이고 구현하기 쉽지만 호출을 받은 Micro Service에 장애가 생긴다면 요청을 보낸 서비스는 반응이 올 때까지 기다리게 되고. 반응이 오지 않으면 계속 기다리면서 재호출하게 됨



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 비동기 통신 방식
        - ▪ 메시지 기반의 비동기(asynchronous) 호출 방식은 동기 호출처럼 응답을 기다리지 않음
        - ▪ 메시지를 보낸 다음 응답을 기다리지 않고 다음 일을 처리하는데 보낸 결과가 어떻게 됐는지 응답을 받지 않으므로 동기식처럼 완결성을 보장할 수는 없기 때문에 이를 보장하기 위한 메커니즘이 필요한데 보통 아파치 카프카{Apache Kafka), 래빗엠큐(RabbitMQ), 액티브엠큐(ActiveMQ) 같은 메시지 브로커(message broker)를 활용



MSA의 이해

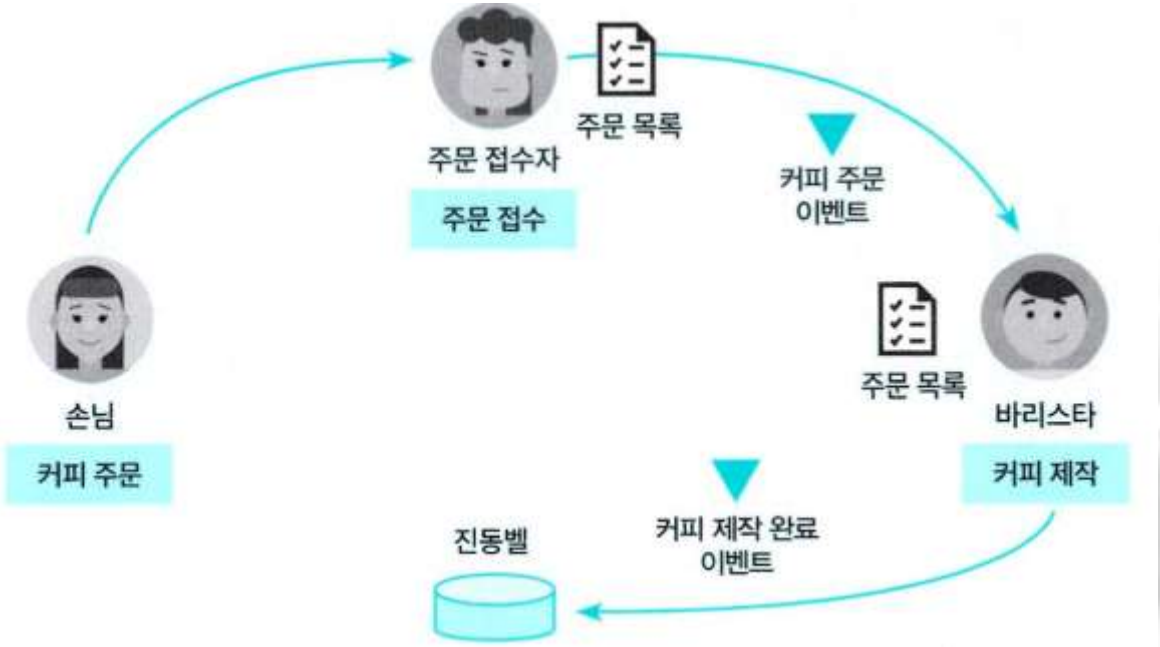
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 비동기 통신 방식
        - ▪ 이러한 메커니즘에서는 메시지를 보내는 생산자(producer)와 메시지를 가져다가 처리하는 소비자(consumer)가 서로 직접 접속하지 않고 메시지 브로커에 연결됨
        - ▪ 메시지 브로커에 메시지를 전달하고 자신의 일을 처리하면 메시지 브로커가 전송을 보장하게 되는데 여러 서비스에서 전달한 메시지를 처리하는 메시지 브로커 자체에 부하가 생길 수가 있으므로 이 경우 메시지 브로커는 메시지 처리 규모에 따라 확장이 가능해야 함
        - ▪ 이 방식은 메시지 브로커에 의해 중계되기 때문에 서로 통신하는 서비스들이 물리적으로 동일한 시스템에 위치할 필요도 없고 서로 프로세스를 공유할 필요도 없으며 심지어 동일한 시간 대에 동시에 동작하지 않아도 되기 때문에 서비스 요구에 따라 늘어나거나 줄어들 수 있는 탄력성이 높은 Cloud 플랫폼 환경에서 서비스가 다운됐을 때 또는 시스템을 더 확장해야 할 때 사용할 수 있는 매우 효과적인 방법
        - ▪ Cloud 벤더에서 완전관리형으로 제공하는 AWS의 SQS나 SNS, Azure Event Hub, Azure Event Grid 등도 많이 사용

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 비동기 방식의 이벤트 기반 아키텍쳐
        - ▪ 비동기 통신 방식을 이용해 느슨한 연계를 지향하는 아키텍처가 있는데 이벤트에 반응한다는 의미로 이벤트 기반 아키텍처(event driven architecture)라고 부르는데 이벤트 기반 아키텍처는 예전부터 사용된 개념으로 분산 시스템 간에 발신자가 이벤트를 생성 및 발행(publish)하고 해당 이벤트를 필요로 하는 수신자에게 전송하면 이벤트를 구독하고(subscribe) 있던 수신자가 이벤트를 받아 처리하는 형태의 시스템 아키텍처
        - ▪ 이벤트는 상태의 변화를 의미하며 기존의 순차적 방식의 아키텍처와 달리 특정 행동이 자동으로 순서에 따라 발생하는 것이 아닌 어떤 상태의 변경에 대한 반응으로 동작한다는 점이 차이점
        - ▪ 이러한 방식은 일반 실생활에서도 작업의 효율성을 위해 이미 많이 사용하는 방식

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 비동기 방식의 이벤트 기반 아키텍쳐



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 비동기 방식의 이벤트 기반 아키텍쳐
        - ▪ 커피숍에서 손님이 주문을 하면 주문 접수자는 주문을 받고 바리스타에게 커피 제작을 의뢰하는데 이때 주문 접수자는 하나의 주문이 바리스타에 의해 완료될 때까지 마냥 주문을 받지 않고 기다리지는 않는데 주문 접수 - 커피 제작 - 고객 전달이라는 하나의 무결하고 완결된 단위로 다루지 않음
        - ▪ 대신 주문이 들어오는 대로 꾸준히 주문 목록에 적고 동시에 커피 주문이 들어왔다는 이벤트를 바리스타에게 계속 전달하면 바리스타도 마찬가지로 커피 주문 이벤트를 순차적으로 제작 목록에 기입하고 주문에 해당하는 커피를 만들고 커피 제작이 완료되면 커피 제작 완료 이벤트를 보내고 이것이 진동벨을 통해 손님에게 통보되는 것
        - ▪ 주문은 많이 들어오는데 바리스타가 부족하다면 추가로 더 투입될 수도 있음
        - ▪ 이벤트 기반 방식은 여러 개의 주문을 받아 여러 개의 커피를 동시에 제작할 수 있는 효율성을 높이는 반면 병렬 처리를 하지 않고 하나의 커피 주문을 받고 커피가 제작될 때까지 다른 주문을 접수할 수 없다면 매우 비효율적일 것

MSA의 이해

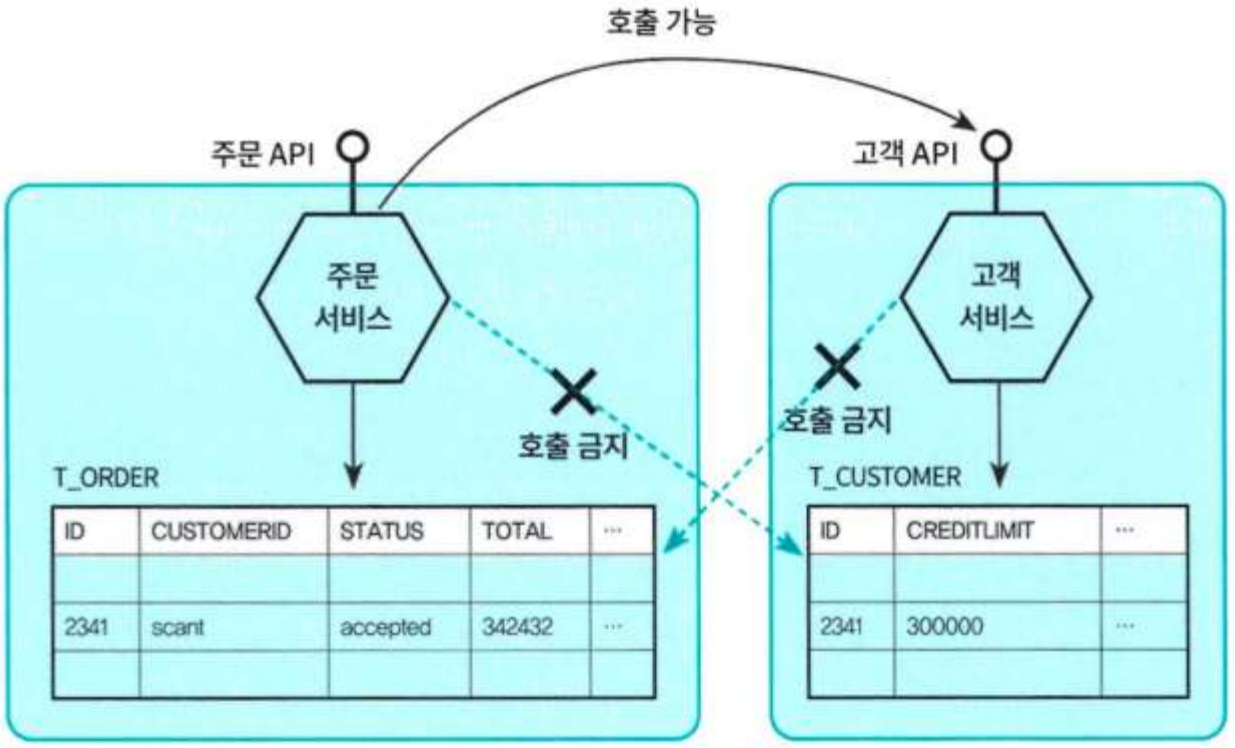
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● Micro Service 통신 패턴
      - ◆ 비동기 방식의 이벤트 기반 아키텍쳐
        - ▪ 이벤트 기반 아키텍처는 이벤트를 생산하는 모듈과 이벤트에 대응하는 모듈을 분리하고 상호 독립적으로 동작하게 함으로써 병렬 처리를 촉진
        - ▪ 이벤트 기반 아키텍처의 전달 메커니즘으로 앞에서 논의한 비동기 메시지 메커니즘을 선택하면 더욱 더 효과적
        - ▪ 이벤트 기반의 아키텍처와 비동기 통신 메커니즘을 함께 사용하는 Micro Service를 이벤트 기반 Micro Service(event-driven microservice)라고도 함
        - ▪ 이벤트 메시지를 사용하면 발신자와 수신자를 장소와 시간에서 쉽게 분리할 수 있으며 Micro Service가 추구하는 느슨한 결합으로 확장성, 탄력성 측면에서 이점이 많음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 저장소 분리 패턴
      - ◆ 기존 Monolithic 시스템의 저장소는 통합 저장소로 애플리케이션 모듈은 분리하되 저장 처리는 모듈 별로 격리하지 않고 다른 모듈에서의 호출을 허용하는 구조
      - ◆ 국내 엔터프라이즈 애플리케이션의 내부를 보면 모든 비즈니스 로직이 데이터베이스의 SQL 처리에 몰려있는 경우가 대부분인데 단순한 조사 방법이긴 하지만 소스코드 라인 수와 데이터 처리를 수행하는 SQL 구문의 라인 수를 세어봐도 알 수 있음
      - ◆ 업무 규칙 및 흐름 처리를 수행하는 애플리케이션의 코드 라인 수보다 SQL 코드의 라인 수가 몇 배 이상
      - ◆ 이러한 구조를 데이터 중심 애플리케이션이라 하는데 특정 관계형 데이터베이스 벤더에 구속되고 복잡해져 유지보수가 어려워지고 성능 문제가 발생했을 때 SQL 구문 튜닝이나 저장소 증설(스케일 업)에 의존할 수 밖에 없음
      - ◆ 이러한 구조의 애플리케이션은 아무리 여러 개의 Micro Service로 분리하더라도 요청이 증가할 경우 서비스는 한가하고 여러 서비스에서 호출되는 통합 데이터베이스만 여전히 바쁜 상황이 되어 Micro Service의 자동 확장(스케일 아웃) 기능이 별 소용이 없어질 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 저장소 분리 패턴



MSA의 이해

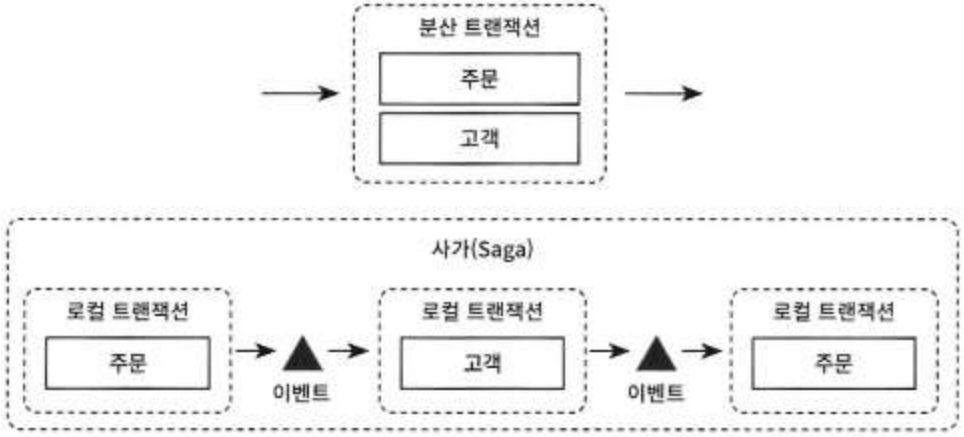
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 저장소 분리 패턴
      - ◆ 저장소 분리 패턴은 각 Micro Service는 각자의 비즈니스를 처리하기 위한 데이터를 직접 소유해야 한다는 것을 의미하기 때문에 자신이 소유한 데이터는 다른 서비스에 직접 노출하지 않고 각자가 공개한 API를 통해서만 접근할 수 있으며 (정보 은닉) 저장소가 격리돼 있기 때문에 각 저장소를 자율적으로 선택할 수 있는데(폴리글랏 저장소) 궁극적으로 이 같은 제약이 데이터를 통한 변경의 파급 효과(영향도)를 줄여 서비스를 독립적으로 만듬
      - ◆ 주문 서비스가 주문 수행을 위해 고객 정보를 필요로 하더라도 바로 고객 테이블에 질의할 수 없고 반드시 고객 서비스의 API를 통해서만 호출할 수 있음
      - ◆ Micro Service별로 기능을 분리하고 저장소를 격리함에 따라 이전에는 불거지 지 않았던 문제가 생기는데 여러 개의 분산된 서비스에 걸쳐 비즈니스 처리를 수행해야 하는 경우 비즈니스 정합성 및 데이터 일관성을 어떻게 보장할 것인가에 대한 문제

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 분산 트랜잭션 처리 패턴
      - ◆ 손쉽게 적용할 수 있는 한 가지 방법은 여러 개의 분산된 서비스를 하나의 일관된 트랜잭션으로 묶는 것
      - ◆ 분산 트랜잭션 처리에서는 여러 서비스 간의 비즈니스 및 데이터 일관성을 유지할 필요가 있는데 분산 트랜잭션 처리를 위한 전통적인 방법으로 2단계 커밋 같은 기법이 있음
      - ◆ 2단계 커밋은 분산 데이터베이스 환경에서 원자성(atomicity)을 보장하기 위해 분산 트랜잭션에 포함돼 있는 모든 노드가 커밋(commit)되거나 롤백(rollback)하는 메커니즘인데 각 서비스에 잠금(lock in)이 걸려 발생하는 성능 문제 탓에 효율적인 방법이 아닌데 특히 각 서비스가 다른 인스턴스로 로딩되기 때문에 통제하기 어렵고 서비스의 저장소가 각각 다를 경우 문제가 있으며 특히 MongoDB 같은 NoSQL 저장소는 2단계 커밋 자체를 지원하지 않고 Cloud의 가장 큰 장애는 네트워크 장애인 경우가 많은데 네트워크 장애 등으로 특정 서비스의 트랜잭션이 처리되지 않을 경우 트랜잭션에 묶인 서비스가 즉시 영향을 받기도 함
      - ◆ 2단계 커밋을 통한 분산 트랜잭션 처리는 독립적이지 않고 비자율적

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 분산 트랜잭션 처리 패턴
      - ◆ Micro Service의 독립적인 분산 트랜잭션 처리를 지원하는 패턴이 사가(Saga) 패턴
      - ◆ 사가 패턴은 각 서비스의 로컬 트랜잭션을 순차적으로 처리하는 패턴
      - ◆ 사가 패턴은 여러 개의 분산된 서비스를 하나의 트랜잭션으로 묶지 않고 각 로컬 트랜잭션과 보상 트랜잭션을 이용해 비즈니스 및 데이터의 정합성을 맞추는데 각 로컬 트랜잭션은 자신의 데이터베이스를 업데이트한 다음 사가 내에 있는 다음 로컬 트랜잭션을 트리거하는 메시지 또는 이벤트를 게시해서 데이터의 일관성을 맞추는 방식

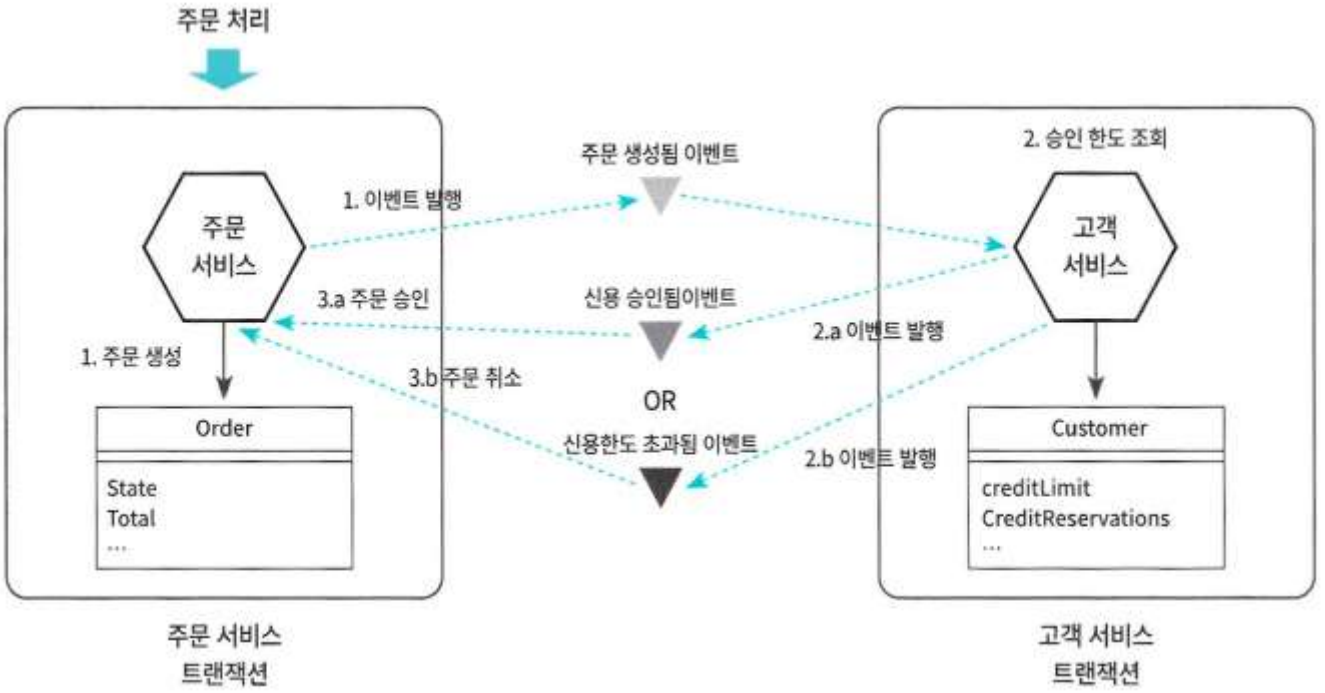


MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 분산 트랜잭션 처리 패턴
      - ◆ 롤백이 필요하다면 보상 트랜잭션의 개념을 이용
      - ◆ 보상 트랜잭션은 어떤 서비스에서 트랜잭션 처리에 실패할 경우 그 서비스의 앞선 다른 서비스에서 처리된 트랜잭션을 되돌리게 하는 트랜잭션
      - ◆ 사가는 일관성 유지가 필요한 트랜잭션을 모두 묶어 하나의 트랜잭션으로 처리하지 않고 각 로컬 트랜잭션으로 분리해서 순차적으로 처리하는 방법으로 트랜잭션이 실패한 경우 이전 로컬 트랜잭션이 작성한 변경 사항을 취소하는 일련의 보상 트랜잭션을 통해 비즈니스 처리의 일관성을 유지

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 분산 트랜잭션 처리 패턴
      - ◆ 주문을 처리할 때 고객의 신용 한도 정보에 따라 최종 주문을 승인하는 업무가 있는데 이 두 서비스의 트랜잭션을 하나로 묶지 않고 보상 트랜잭션과 이벤트를 활용해 처리할 수 있음



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 분산 트랜잭션 처리 패턴
      - ◆ 주문을 처리할 때 고객의 신용 한도 정보에 따라 최종 주문을 승인하는 업무가 있는데 이 두 서비스의 트랜잭션을 하나로 묶지 않고 보상 트랜잭션과 이벤트를 활용해 처리할 수 있음
        - ▪ 주문 처리가 시작되면 주문 서비스는 가주문을 생성하고 주문자 정보가 담긴 주문 생성됨 이벤트를 발행하고 트랜잭션을 종료
        - ▪ 고객 서비스가 주문 생성됨 이벤트를 확인한 뒤 다음 처리를 수행
          - 이벤트에 존재하는 주문자 정보로 고객의 신용한도를 조회해서 신용한도가 충족되면 신용 승인됨 이벤트를 발행
          - 신용한도가 충족되지 않는다면 신용한도 초과됨 이벤트를 발행
        - ▪ 주문 서비스는 고객 서비스가 발행한 이벤트를 확인해 다음 처리를 수행
          - 고객 서비스가 발행한 이벤트가 신용 승인됨 인 경우에는 주문 승인 처리를 수행
          - 신용한도 초과됨 이벤트인 경우에는 보상 트랜잭션인 주문 처리 취소를 수행
        - ▪ 이처럼 하나의 큰 트랜잭션으로 묶지 않고 4개의 분리된 로컬 트랜잭션으로 비즈니스의 정합성을 맞출 수 있음

MSA의 이해

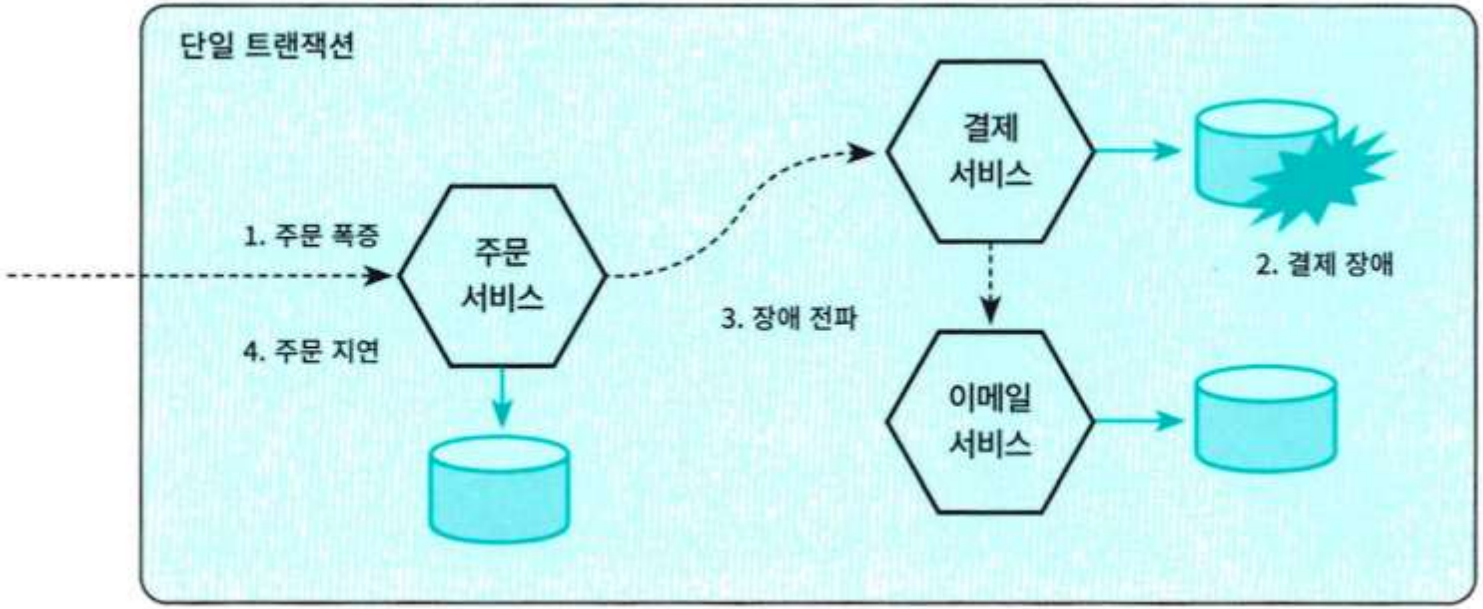
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 데이터 일관성에 대한 생각의 전환: 결과적 일관성
      - ◆ 모든 애플리케이션에는 비즈니스 처리를 위한 규칙이 있고 이러한 비즈니스 규칙을 만족하도록 데이터 일관성이 유지돼야 하는데 이전까지는 이 같은 데이터 일관성이 실시간으로 반드시 맞아야 한다는 생각이 일반적
      - ◆ 과연 모든 비즈니스 규칙들이 실시간으로 일관성을 맞춰야 할까 - 정말 그래야 하는가는 생각해 볼 문제
      - ◆ 쇼핑몰에서 주문을 하면 결제 처리가 돼야 하고 그 결제가 완료되면 결제 내용과 주문 처리 내역이 주문자에게 이메일로 전송돼야 하는데 일반적으로 주문된 다음에 결제가 되고 결제 내용이 이메일로 통보되는 것이 순차적인 일 처리 순서

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 데이터 일관성에 대한 생각의 전환: 결과적 일관성
      - ◆ 주문 지연 상황 발생
        - ▪ 주문자가 폭주한 경우를 생각해 보면 주문, 결제, 이메일 처리가 이러한 순차적인 동시 일관성을 추구하는 경우 주문이 폭주하면 결제 처리가 되지 않거나 지연되는 경우가 발생할 수 있는데 그럼 앞선 주문 처리 역시 지연될 것
        - ▪ 북미의 블랙 프라이데이(Black Friday) 같은 대규모 쇼핑 이벤트를 생각해 보면 수만 개의 주문이 발생하는데 결제 서비스에서 타사 외부 연동 장애가 발생해 더는 주문을 받을 수 없는 상황이 발생할 수도 있는데 이러한 상황을 고려했을 때 비즈니스 관점에서 보면 주문과 결제, 이메일 전송을 순차적으로 처리하기보다 무조건 먼저 주문을 많이 받아 놓는 것이 좋을 수 있음
        - ▪ 모든 비즈니스 처리가 반드시 실시간성을 요구하는 것은 아닌데 어떤 비즈니스는 데이터의 일관성이 실시간으로 맞지 않더라도 어느 일정 시점이 됐을 때 일관성을 만족해도 되는 것이 있는데 이러한 개념을 결과적 일관성(eventual consistency)이라고 함
        - ▪ 결과적 일관성의 개념은 고가용성을 극대화

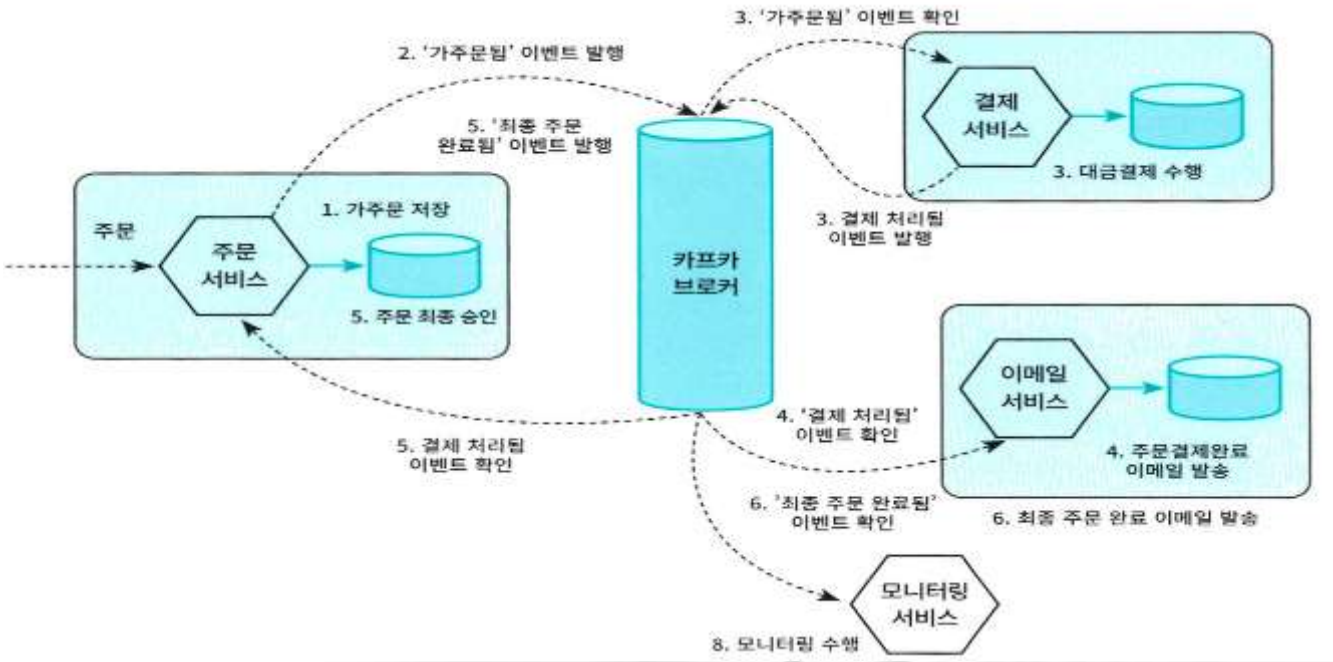
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 데이터 일관성에 대한 생각의 전환: 결과적 일관성
      - ◆ 주문 지연 상황 발생
        - ▪ 실시간성을 강조해서 결제 서비스에서 발생한 장애가 다른 서비스의 가용성을 떨어뜨린 사례로 결제 서비스의 장애로 주문 서비스마저 사용할 수가 없는 것



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 데이터 일관성에 대한 생각의 전환: 결과적 일관성
      - ◆ 사가 패턴과 이벤트 메시지 기반 비동기 통신을 적용
        - ▪ 각 Micro Service의 트랜잭션은 독립적이고 각 트랜잭션이 성공했을 때 상태 변경 이벤트를 발행해 이 이벤트를 구독한 다른 서비스의 로컬 트랜잭션이 작동하도록 변경



MSA의 이해

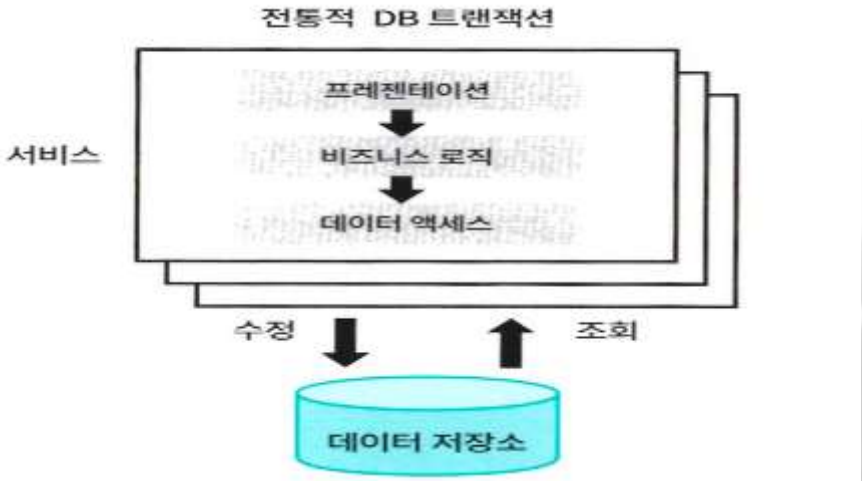
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 데이터 일관성에 대한 생각의 전환: 결과적 일관성
      - ◆ 사가 패턴과 이벤트 메시지 기반 비동기 통신을 적용
        - ▪ 변경된 아키텍쳐
          - 가주문이 생성되고 가주문됨 이벤트를 발행하는데 주문은 독립적 로컬 트랜잭션이기 때문에 끊임없이 받을 수 있는데 주문이 몰릴 경우 주문 서비스만 확장해서 가용성을 높일 수 있음
          - 가주문됨 이벤트는 메시지 브로커에 비동기로 전송
          - 결제 서비스는 발행된 가주문됨 이벤트를 확인하고 대금 결제 트랜잭션을 수행하고 결제 처리됨 이벤트를 발행
          - 이메일 서비스는 결제 처리됨 이벤트를 확인하고 주문 결제 완료 이메일을 사용자에게 발송
          - 주문 서비스는 결제 처리됨 이벤트를 확인하고 가주문으로 처리됐던 주문을 최종 승인하고 최종 주문 완료됨 이벤트를 발행
          - 이메일 서비스는 주문 서비스가 발행한 최종 주문 완료됨 이벤트를 확인해 최종적으로 주문이 완료됐다 는 이메일을 사용자에게 발송
          - 각 서비스는 각기 작업을 수행하다 오류가 발생하면 실패 이벤트를 발행해 다른 서비스가 비즈니스 정합성을 맞출 수 있게 함

MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 데이터 일관성에 대한 생각의 전환: 결과적 일관성
      - ◆ 사가 패턴과 이벤트 메시지 기반 비동기 통신을 적용
        - ▪ 변경된 아키텍쳐
          - • 별도로 메시지 큐에 쌓이는 이벤트들을 모니터링 서비스와 연계해 모니터링하고 추적해서 전체적인 비즈니스 정합성 여부를 관리자가 확인할 수도 있음
        - ▪ 이벤트 기반 아키텍처와 메시지 브로커, 사가 패턴으로 비즈니스 정합성을 결과적으로 보장할 수 있고 비즈니스 및 시스템 가용성을 극대화 할 수 있음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 읽기 와 쓰기 분리: CQRS 패턴
      - ◆ 비즈니스에 대한 기존 관념을 조금만 바꾸면 가용성을 높일 수 있는 방법이 다양
      - ◆ Micro Service의 핵심 설계 철학인 서비스 별 데이터 저장소 패러다임을 채택하면 서비스의 성능 향상을 위해 서비스 인스턴스를 스케일 아웃해서 여러 개로 실행한 경우 데이터 읽기/수정 작업으로 인한 리소스 교착상태가 발생할 수 있음

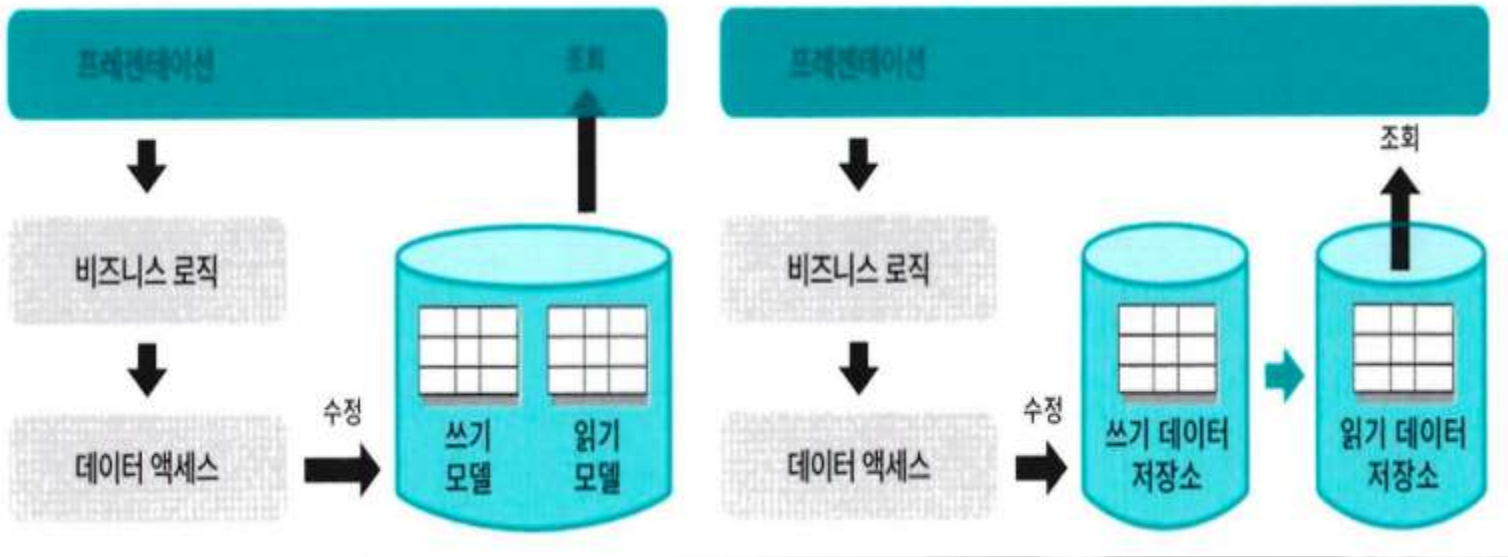


MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - 읽기 와 쓰기 분리: CQRS 패턴
      - 이 문제를 해결하기 위한 방법으로 CQRS 패턴이 있는데 CQRS(Command Query Responsibility Segregation)는 명령 조회 책임 분리를 의미
      - CQRS는 기존의 일반적이었던 동일한 저장소에 데이터를 넣고 입력, 조회, 수정, 삭제를 모두 처리하는 방식에 도전하는 흥미로운 패러다임
      - 일반적으로 사용자의 비즈니스 요청은 크게 시스템 상태를 변경하는 명령과 시스템의 상태를 조회하는 부분으로 나눌 수 있는데 일반적인 비즈니스 모델에서는 입력, 수정, 삭제가 조회보다 적게 쓰이고 조회 요청이 훨씬 많이 사용되는데 서비스 내에 이러한 모든 기능을 넣어 두면 조회 요청 빈도가 증가함에 따라 다른 명령 기능도 함께 확장해야 하므로 효율적이지 않음
      - 하나의 저장소에 쓰기 모델과 읽기 모델을 분리하는 방식으로 변화시켜 쓰기 서비스와 조회 서비스를 분리할 수도 있고 더 나아가 아예 물리적으로 쓰기 트랜잭션 용 저장소와 조회 용 저장소를 따로 준비할 수 있음
      - 쓰기 전략과 조회 전략을 각각 분리 하면 쓰기 시스템의 부하를 줄이고 조회 대기 시간을 줄이는 등 엄청난 이점을 누릴 수 있음

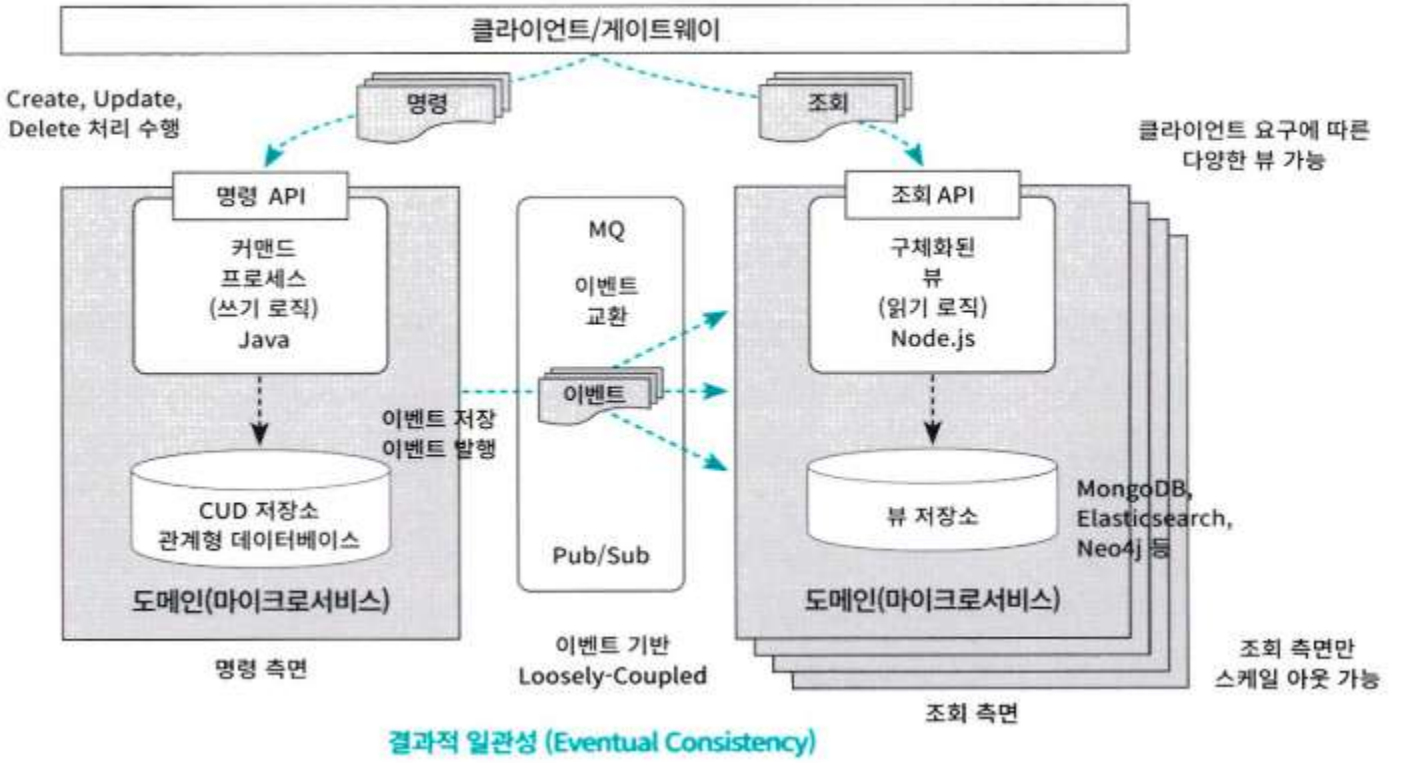
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 읽기 와 쓰기 분리: CQRS 패턴
      - ◆ 쓰기 와 읽기를 분리하는 과정



MSA의 이해

- MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - 읽기 와 쓰기 분리: CQRS 패턴
      - CQRS 방식을 이벤트 메시지 주도 아키텍처와 연계



MSA의 이해

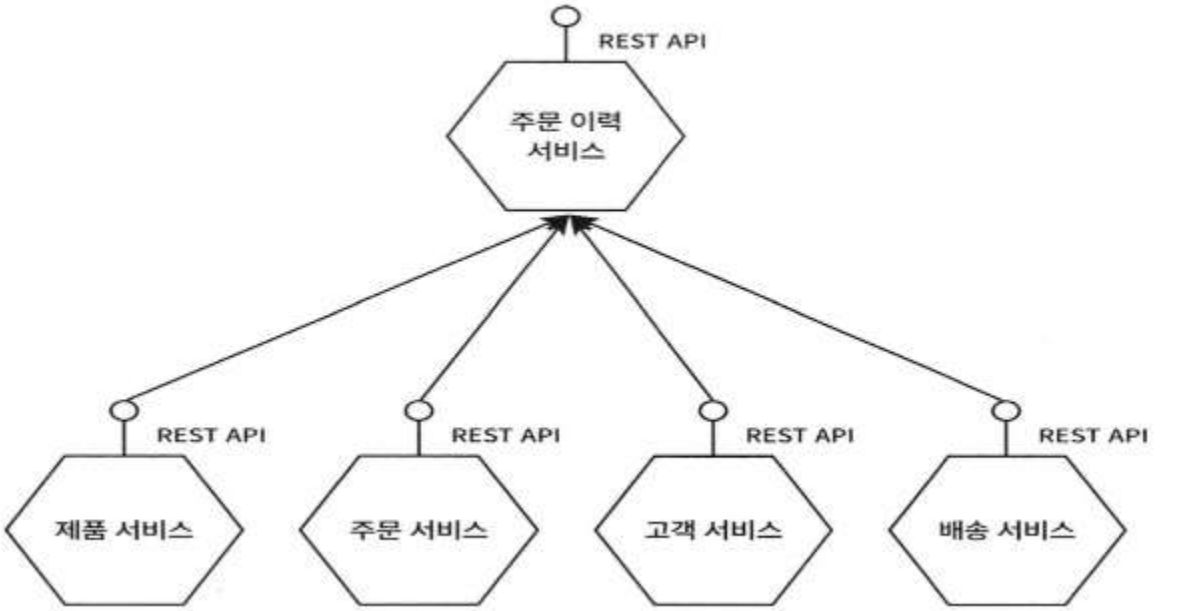
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 읽기 와 쓰기 분리: CQRS 패턴
      - ◆ CQRS 방식을 이벤트 메시지 주도 아키텍처와 연계
        - ▪ 명령 측면 Micro Service는 입력, 수정, 삭제(Create, Update, Delete) 처리를 수행하고 저장소로는 쓰기에 최적화된 관계형 데이터베이스를 사용하고 프로그래밍 언어도 업무 규칙을 표현하기 좋은 자바 언어를 사용
        - ▪ 조회 측면의 Micro Service에서는 조회 성능이 높은 몽고디비나 엘라스틱서치 같은 NoSQL 데이터베이스를 저장소로 사용하고 프로그래밍 언어도 조회를 간단하게 구현할 수 있는 스크립트 기반의 Node.js를 사용
        - ▪ 조회 서비스는 사용량이 많기 때문에 스케일 아웃해서 인스턴스를 증가시켜 놓음
        - ▪ 이러한 구조에서는 명령 서비스가 사용됨에 따라 조회 서비스 와의 데이터 일관성이 깨지게 되는데 이때 데이터 일관성 유지를 위해 필요한 것이 이벤트 주도 아키텍처
        - ▪ 명령 서비스는 저장소에 데이터를 쓰면서 저장한 내역이 담긴 이벤트를 발생시켜 메시지 브로커에 전달하고 조회 서비스는 메시지 브로커의 이벤트를 구독하고 있다가 이벤트 데이터를 가져와 데이터를 최신 상태로 동기화
        - ▪ 명령 서비스에 데이터가 들어간 즉시 조회 서비스의 데이터와 일치할 수 없고 시간적 간격이 있을 수 있지만 어느 시점이 되면 결과적으로 일치하게 되는데 이는 앞에서 언급한 결과적 일관성을 추구하는 것

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● API 조합과 CQRS
      - ◆ Micro Service의 저장소가 격리돼 있고 각 Micro Service마다 각기 다른 기능을 구현했을 때 여러 개의 Micro Service를 연계해서 서비스로 제공하는 경우에 사용할 수 있는 첫 번째 방법은 API 조합(composition)
        - ▪ 주문 이력 서비스는 제품 서비스가 제공하는 제품 정보, 주문 서비스의 주문 정보, 고객 서비스의 특정 고객 정보. 배송 서비스의 배송 정보가 모두 다 필요하기 때문에 각 기능을 제공하는 Micro Service를 조합하는 상위 Micro Service를 만들어 조합된 기능을 제공할 수 있음
        - ▪ 하위 서비스는 각자 독립적인 API를 제공하면서 연계 API를 위해 상위 서비스에 정보를 제공
        - ▪ 그렇지만 이러한 구조는 상위 서비스가 하위 서비스에 의존하는 결과를 가져오는데 상위 서비스가 제공하는 API에 정보를 제공하는 하위 서비스 중 하나라도 API를 변경하면 상위 서비스는 그에 따라 변경될 수밖에 없고 하위 서비스의 실패가 상위 서비스에 영향을 주는데 이러한 의존성을 줄이기 위한 방법이 필요

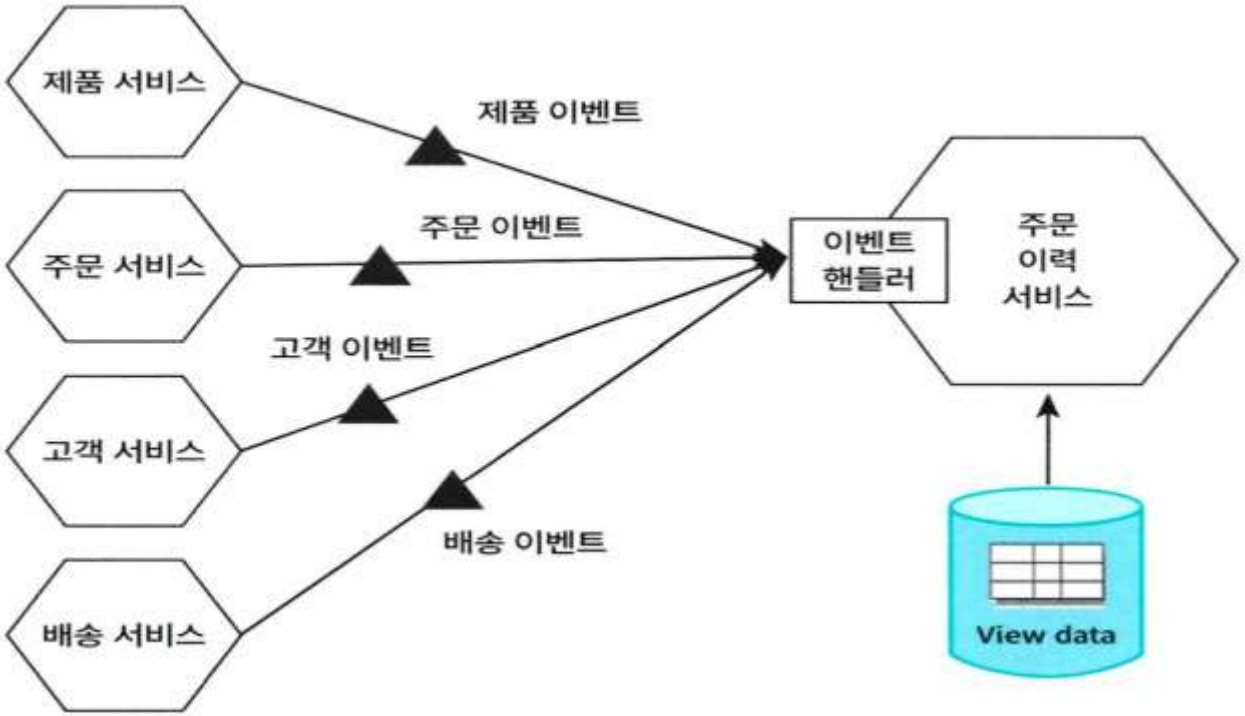
MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● API 조합과 CQRS
      - ◆ Micro Service의 저장소가 격리돼 있고 각 Micro Service마다 각기 다른 기능을 구현했을 때 여러 개의 Micro Service를 연계해서 서비스로 제공하는 경우에 사용할 수 있는 첫 번째 방법은 API 조합(composition)



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● API 조합과 CQRS
      - ◆ 두 번째 방법은 CQRS 적용



MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● API 조합과 CQRS
      - ◆ 두 번째 방법은 CQRS 적용
        - ▪ 주문 이력 서비스를 제공하는 Micro Service가 독자적인 저장소를 갖도록 만들고 주문 이력의 세세한 원천 정보를 보유하고 있는 각각의 서비스도 독자적으로 자신의 저장소를 가지고 서비스를 제공하고 있고 이 원천 정보를 보유한 여러 Micro Service는 자신의 서비스의 정보가 변경되는 시점에 변경 내역을 각자의 변경 이벤트로 발행
        - ▪ 그럼 주문 이력 Micro Service에서는 이 이벤트를 구독하고 있다가 이벤트를 가져와서 자신의 서비스의 저장소에 기록함으로써 다른 서비스의 데이터와 데이터 일관성을 맞추고 서비스의 주문 이력 조회 기능으로 제공
        - ▪ 이런 경우에는 다른 원천 서비스가 순간적인 장애가 생긴다고 해도 주문 이력 서비스가 영향을 받지 않는데 조회용 Micro Service를 별도로 생성하고 다른 서비스로부터 비동기 이벤트로 일관성을 맞춤으로써 API 조합 방식의 단점인 직접적인 의존성을 줄일 수 있는 것

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 쓰기 최적화: 이벤트 소싱 패턴
      - ◆ 사가 패턴 및 CQRS 패턴에서 비즈니스 불일치를 피하기 위해서는 저장소에 저장하는 것과 메시지를 보내는 것이 원자성을 지녀야 하는데 저장소에 저장하는 일과 메시지를 보내는 작업이 언제나 완전하게 진행되어 함께 실행돼야 하지만 객체의 상태 변화를 이벤트 메시지로 발행하고 또 객체 상태를 관계형 데이터베이스에 저장하는 경우 SQL 질의어로 변환해서 처리하기가 매우 번거롭고 까다롭고 메시지 발행과 저장 처리라는 두 가지 기능을 수행하므로 빠르지도 않음
      - ◆ 메시지 발행 및 저장 처리의 원자성을 보장하고 성능도 최적화하는 방법은 비즈니스 처리를 수행할 때 데이터 처리는 항상 처리 상태의 결괏값을 계산하고 데이터의 최종 상태를 확정해서 저장하는 방식으로 진행
      - ◆ 쇼핑몰의 장바구니에 품목을 추가, 삭제하는 과정을 보면 만약 일반적인 관계형 데이터베이스와 자바 언어를 사용한다면 장바구니, 품목 데이터 모델이 정의돼 있어야 하고 장바구니 상태를 변경할 때 매번 트랜잭션 결과를 반영해서 장바구니 데이터 모델의 결과를 계산해야 함

MSA의 이해

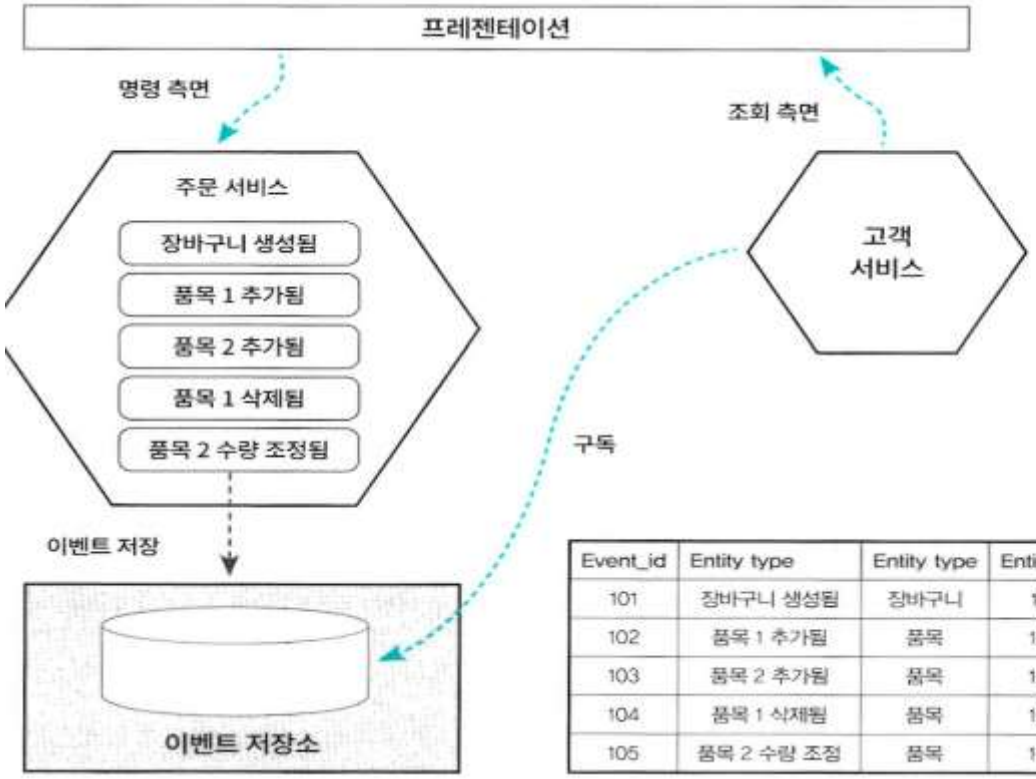
- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 쓰기 최적화: 이벤트 소싱 패턴
      - ◆ 객체의 상태 변화와 데이 터 모델의 결과의 변화 양상
        - ▪ 사용자 A의 장바구니 객체 생성 -> 장바구니 테이블에 사용자 A의 장바구니 로우(row)를 생성
        - ▪ 품목 1 객체 추가 -> 품목 테이블에 사용자 A 장바구니의 품목 1을 추가
        - ▪ 품목 2 객체 추가 -> 품목 테이블에 사용자 A 장바구니의 품목 2를 추가
        - ▪ 품목 1 객체 제거 -> 품목 테이블에 사용자 A 장바구니의 품목 1을 삭제
        - ▪ 품목 2 객체의 수량 변경 -> 품목 테이블에 사용자 A 장바구니의 품목 2의 수량 정보를 수정
      - ◆ 이처럼 객체의 상태 변경에 따라 데이터 모델로 처리되고 최종값이 반영돼야 하고 이 같은 과정은 복잡해지고 변환 처리로 느릴 수밖에 없는데 특히 인스턴스가 여러 개로 확장될 때 동시 업데이트 및 교착 상태로부터 안전하지 못할 수도 있음
      - ◆ 객체 상태를 데이터 모델에 맞춰 계산하지 않고 상태 트랜잭션 자체를 저장하는 방법을 사용할 수 있는데 이를 이벤트 소싱(event sourcing) 기법이라고 하며 트랜잭션 자체를 저장하자는 전략으로 상태 변경 이벤트를 계산해서 데이터 모델로 변경하지 않고 바로 이벤트 저장소에 그대로 저장

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 쓰기 최적화: 이벤트 소싱 패턴
      - ◆ 이렇게 하면 메시지 브로커와 데이터 저장소를 분리하지 않고 하나로 사용할 수 있으며 복잡한 과정이 없으니 쓰기 속도가 훨씬 빠름
      - ◆ 현재 시점의 상태가 필요할 때는 상태의 출발점부터 모든 기록된 상태 변경 트랜잭션을 순차적으로 계산
      - ◆ 처음부터 모든 트랜잭션을 처리하는 것이 부담된다면 매일 자정에 상태를 계산한 후 스냅숏으로 저장한 후 현재 상태 정보가 필요해지면 스냅숏 이후의 트랜잭션만 처리하면 되는데 이 같은 방식은 특정 시점의 상태가 필요하면 재현할 수도 있기 때문에 별도의 트랜잭션성 이력 로그 데이터를 기록할 필요도 없음
      - ◆ 또 한 가지 중요한 점은 명령 측면과 조회 측면의 서비스가 이벤트 저장소에 대한 CRUD를 모두 처리할 필요 없이 입력/조회(CR) 만 처리하면 된다는 것
      - ◆ 저장소에서 변경과 삭제가 발생하지 않기 때문에 명령 측면의 서비스를 여러 개 확장해도 동시 업데이트 및 교착상태가 발생하지 않음

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 쓰기 최적화: 이벤트 소싱 패턴
      - ◆ 이벤트 저장소의 데이터 형태



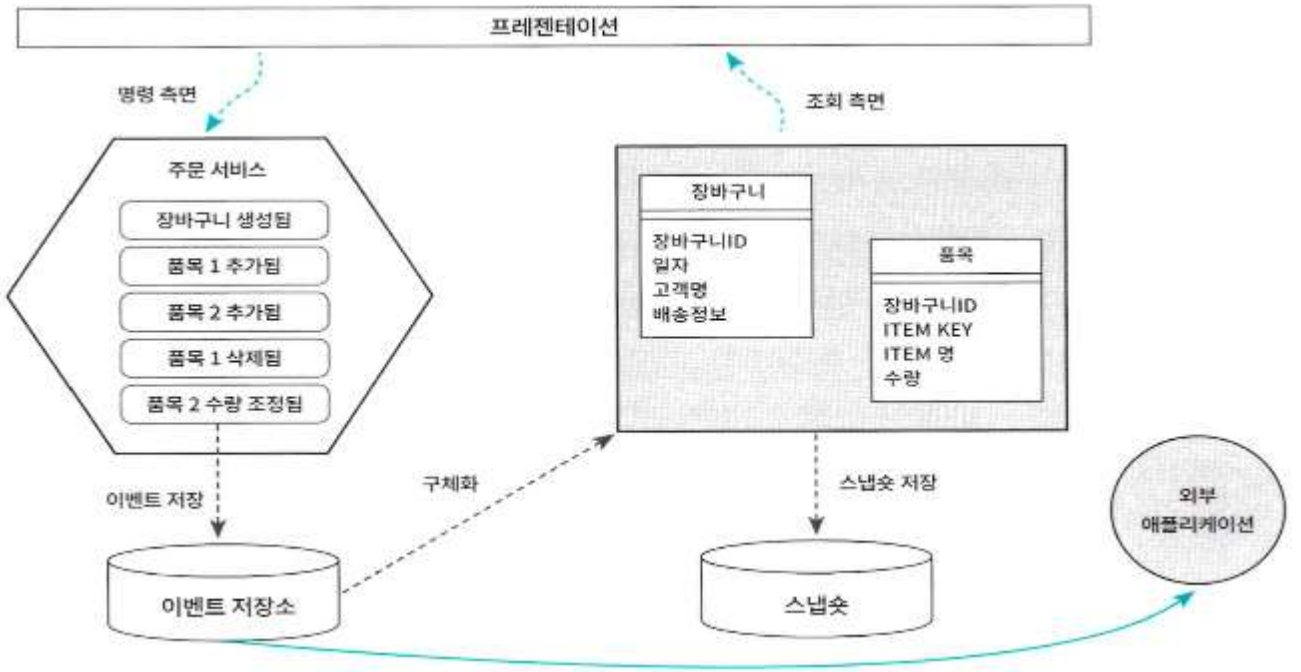
Event_id	Entity type	Entity type	Entity_id	Entity data
101	장바구니 생성됨	장바구니	101	{20190120,홍길동,서초동}
102	품목 1 추가됨	품목	102	{101,01,마우스,3개}
103	품목 2 추가됨	품목	102	{101,02,노트북,3개}
104	품목 1 삭제됨	품목	102	{……}
105	품목 2 수량 조정	품목	102	{101,02,노트북,2개}

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 쓰기 최적화: 이벤트 소싱 패턴
      - ◆ 이벤트 저장소의 데이터 형태
        - ▪ 이벤트 아이디가 있고 이벤트 타입으로 어떠한 상태인지 엔티티 타입으로 어떠한 객체의 이벤트인지 그리고 변경 내용이 엔티티 데이터 항목에 JSON 형태로 그대로 저장
        - ▪ JSON은 객체 형태라서 상태 객체가 그대로 들어간 것과 같음
        - ▪ 이벤트는 한번 발생한 후에 수정되지 않고 업데이트나 삭제 없이 입력만 되는 개념이라 동시성이나 정합성 등의 문제에 비교적 자유로움
        - ▪ 이벤트 소싱은 모든 트랜잭션의 상태를 바로바로 계산하지 않고 별도의 이벤트 스트림으로 이벤트 스트림 저장소에 저장하는 방식으로 이벤트 스트림 저장소는 오로지 추가만 가능하게끔 해서 계속 이벤트들이 쌓이게 만들고 실제로 내가 필요한 데이터를 구체화하는 시점에서는 그때까지 축적된 트랜잭션을 바탕으로 상태를 계산해서 구성
        - ▪ 이벤트 저장소는 이벤트 데이터베이스의 역할뿐 아니라 메시지 브로커처럼 작동하는데. 이는 데이터 저장 처리 메커니즘과 메시지 큐 같은 이벤트를 전달하기 위한 메커니즘을 통합해서 복잡성을 줄이고 특히 쓰기 성능을 최적화하고 상태를 저장하기 때문에 정확한 감사 로깅을 제공하고 객체의 예전 상태를 재구성하는 것이 간단해지며 외부 애플리케이션에 이벤트를 전달하는 것도 저장한 이벤트를 그대로 전송하면 되기 때문에 간편

MSA의 이해

- ❖ MSA 구성 요소 및 MSA 패턴
  - ✓ 애플리케이션 패턴
    - ● 쓰기 최적화: 이벤트 소싱 패턴
      - ◆ 이벤트 소싱 개념도



- ◆ 이벤트 스토밍 및 CQRS 구현을 지원하는 프레임워크로 자바 진영에서는 네덜란드의 Axon Framework 와 Eventuate 가 주로 사용

Micro Service Application Architecture

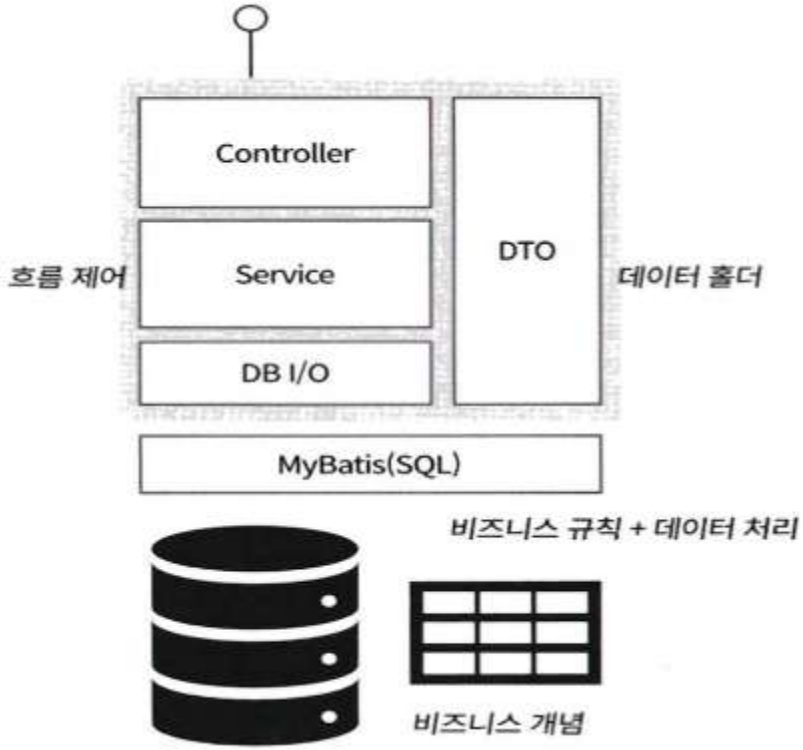
- Micro Service Application Architecture
  - ✓ 로버트 C. 마틴(Robert C. Martin)은 클린 아키텍처에서 소프트웨어의 가치는 행위 가치와 구조 가치로 나뉘고 소프트웨어를 정말로 부드럽게(Soft) 만드는 것은 구조 가치라고 언급한 바 있는데 여기서 행위 가치는 소프트웨어의 기능을 말하며 구조 가치는 소프트웨어 아키텍처를 의미하는데 그는 토끼와 거북이의 경주를 예로 들며 가장 빨리 가는 방법은 제대로 가는 것이며 코드와 설계의 구조를 깔끔하게 만들려는 생각을 하지 않고 기능 구현만 목적으로 삼으면 소프트웨어가 엉망이 된 상황에 대처하는데 더 많은 비용이 든다는 점을 강조
  - ✓ 단기간의 프로젝트 동안 애플리케이션 구조나 설계에 신경 쓰지 않고 오직 기능 구현 에만 몰두한 소프트웨어는 전혀 소프트하지 않은데 새로운 형태의 UI나 기술을 추가해야 한다고 했을 때 거의 처음부터 새로운 시스템을 만드는 것과 같은 수준으로 수정해야 하고 사소한 기능 변경에도 변경의 파급효과 및 영향도를 알 수 없어 그에 따른 실제 변경 작업보다 다른 모듈의 영향도를 파악하기 위한 테스트에 더 많은 시간을 투자해야 함
  - ✓ 개발과 운영을 모두 책임지고 있는 Micro Service팀 입장에서는 소프트웨어의 초기 개발 뿐만 아니라 지속적인 비즈니스의 변화에 빠르게 대응할 수 있는 구조가 필요한데 소프트웨어가 부드럽지 않다면 기민하게 대응하기가 어려울 것
  - ✓ Micro Service 또한 작은 애플리케이션이므로 기존의 애플리케이션 아키텍처의 연장선에서 고려해야 함

Micro Service Application Architecture

- 비지니스 로직은 어디에 – 관심사의 분리
  - ✓ 소프트웨어의 핵심은 비즈니스 로직인데 비즈니스 로직이란 보통 시스템의 목적인 비즈니스 영역의 업무 규칙(Rule), 흐름(Flow), 개념(Concept)을 표현하는 용어
  - ✓ 개발자의 역할은 문제 영역의 비즈니스 로직을 분석 및 이해하고 프로그래밍 언어라는 도구로 잘 표현하는 일인데 여기서 잘 표현한다는 것은 기능이 잘 동작하는 것과 더불어 이해하기 쉽고 변경하기 쉬운 시스템을 만드는 것을 의미
  - ✓ 설계 원칙 중 관심사의 분리(separation of concerns)라는 원칙이 있는데 이것은 시스템의 각 영역이 처리하는 관심사가 분리되어 잘 관리돼야 한다는 의미이고 이 원칙은 시스템을 이해하고 변경하기 쉽게 만들어 주는데 이 원칙에 따라 각 영역은 고유 관심사에 의해 분리되고 집중돼야 함
  - ✓ 모듈화 및 계층화도 이 같은 원칙에 기인하는데 특히 비즈니스를 표현하는 비즈니스 로직 영역과 기술 문제를 처리하기 위한 기술 영역은 철저히 분리하는 것이 좋은데 이것은 비즈니스 로직이 기술보다는 오랫동안 지속되고 안정적이어야 할 애플리케이션의 핵심 영역이기에 기술에 영향을 적게 받도록 설계하는 것을 강조한 데서 기인
  - ✓ 기술과 비즈니스 로직을 분리했을 때 복잡성이 낮아지고 유지보수성도 높아짐
  - ✓ 객체지향 분석설계(OOAD: Object Oriented Analysis and Design)에서는 비즈니스 로직을 누가 봐도 이해하기 쉽게 구조화하는 객체 모델로 표현하는 것을 강조
  - ✓ 애플리케이션의 유지보수성이 높다는 의미는 특정 개인에 의존하기보다는 어느 누구라도 손쉽게 애플리케이션을 이해하고 유지보수 할 수 있음을 의미
  - ✓ 유연하고 확장성 있는 MSA시스템을 만들기 위해서는 각 Micro Service의 관계들을 유연하게 만드는 MSA 외부 아키텍처 및 패턴도 중요하지만 Micro Service의 내부 구조를 어떻게 유연하게 만들 것인지도 중요

Micro Service Application Architecture

- ❖ 비지니스 로직은 어디에 – 관심사의 분리
  - ✓ 데이터베이스 중심 아키텍쳐의 문제점
    - 데이터베이스 중심 아키텍처란 특정 관계형 데이터베이스에 의존한 데이터 모델링을 수행한 다음 이 물리 테이블 모델을 중심에 두고 애플리케이션을 구현하기 위한 사고를 하는 방식



Micro Service Application Architecture

- ❖ 비지니스 로직은 어디에 – 관심사의 분리
  - ✓ 데이터베이스 중심 아키텍쳐의 문제점
    - 일반적으로 스프링 프레임워크를 활용한다면 컨트롤러(Controller), 서비스(Service), DB I/O(Database Input/Output), DTO(Data Transfer Object)로 구성되고. 데이터 처리는 SQL 매핑 프레임워크인 마이바티스(MyBatis)를 사용
    - 이러한 구조에서 일반적으로 비즈니스 로직은 서비스에 존재해야 한다고 말하지만 서비스에 존재하게 될 로직은 흐름 제어 로직 밖에 없고 그 밖의 비즈니스 개념과 규칙들은 앞에서 언급한 사례처럼 테이블과 SQL 질의에 존재
    - DTO는 질의를 통해 가져오는 정보 묶음(information holder)의 역할밖에 할 수 없음
    - 이 구조는 애플리케이션 로직 구성 패턴 중 하나인 트랜잭션 스크립트 구조와 비슷한데 간단한 처리 로직의 경우에는 편하지만 업무가 복잡해지면 점점 복잡성을 제어할 수 없게 된다는 단점이 있으며 업무 개념이 특정 저장 기술인 관계형 데이터베이스 테이블로 표현되고 업무가 복잡해질수록 업무 규칙이 데이터 질의 언어인 SQL과 섞여 표현
    - 비즈니스 민첩성을 위해서는 유연성과 확장성이 중요하다고 한 바 있는데 한 회사에서 비즈니스의 특정 기능을 위해 읽기에 최적화된 NoSQL 저장소로 교체하기로 결정했다고 하면 이러한 시스템 구조에서는 저장소를 변경하려고 해도 쉽게 변경할 수가 없는데 왜냐하면 저장 기술과 비즈니스 로직이 끈끈하게 붙어 있기 때문에 저장소를 변경했을 때 모든 것을 다시 구현해야 한다고 개발팀이 판단했기 때문

Micro Service Application Architecture

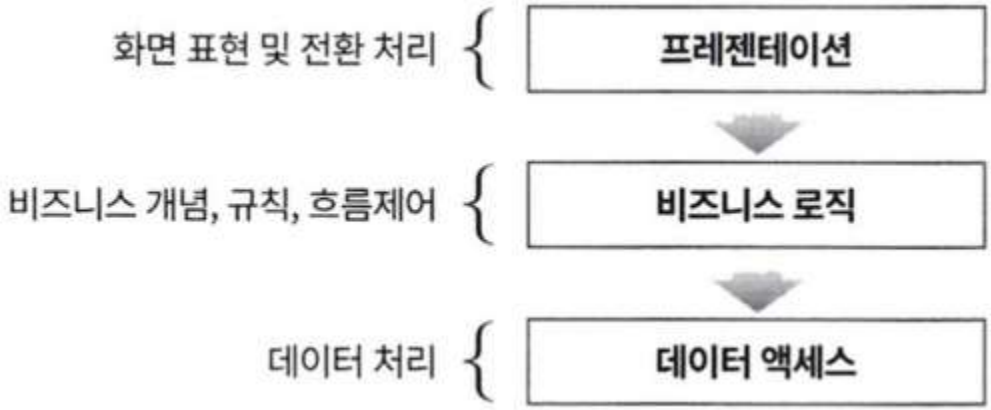
- ❖ 비지니스 로직은 어디에 – 관심사의 분리
  - ✓ 데이터베이스 중심 아키텍쳐의 문제점
    - 데이터베이스 중심 아키텍처의 성능 측면을 보면 이 같은 구조에서는 대부분의 성능을 데이터베이스에 의존
    - 서비스의 비즈니스 개념과 규칙이 대부분 데이터베이스에 표현되기 때문이며 애플리케이션에서는 별로 할 일이 없기 때문에 데이터가 늘어남에 따라 데이터베이스의 성능은 지속적으로 떨어지게 되고 이를 최적화할 방법으로 데이터베이스 서버의 사양과 용량을 계속 증가시키고 질의문 튜닝에 몰두할 수밖에 없는데 이렇게 되면 Cloud 인프라를 사용할 때 의 가장 큰 장점인 사용량에 유연하게 대응하는 자동 스케일 아웃이 의미가 없어지고 정작 바쁜 것은 데이터베이스이기 때문에 애플리케이션을 아무리 스케일 아웃해봐야 거둘 수 있는 효과가 미미
    - 데이터베이스의 본질은 데이터 저장 처리이고 여기에 최적화돼 있으며 SQL도 비즈니스 로직을 처리하기 위한 언어가 아니라 데이터 처리에 최적화된 언어이므로 스토리지가 비싸고 한정적인 인프라 상황에서 최적의 성능을 발휘하기 위해 데이터베이스 중심의 아키텍처가 필요했던 시기가 있었지만 Cloud 시대에는 필요한 만큼 인프라를 유연하게 이용할 수 있고 저장 기술 또한 선택지가 다양하며 이전처럼 웹과 관계형 데이터베이스만 고려해야 하는 상황도 아니며 웹, 모바일, 명령형 인터페이스(CLI), IoT 기기 등 여러 디바이스의 입출력을 지원해야 하고 관계형 데이터베이스, 메모리 데이터베이스, NoSQL, 메시지 큐까지 다양한 저장소와의 연계가 필요
    - Cloud의 풍부한 자원 환경에서는 애플리케이션 자체의 성능보다는 애플리케이션의 확장성과 유연함이 더 중요하므로 관심사의 분리 원칙에 따라 끈끈하게 결합돼 있던 비즈니스 로직 처리와 데이터 처리를 철저히 분리하는 것이 반드시 필요

Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - 레이어드 아키텍처(계층형 아키텍처)를 구성하는 레이어는 많은 사람들이 혼동하는 물리적인 티어의 개념과 달리 논리적인 개념
    - 티어는 물리적인 장비나 서버 컴퓨터 등의 물리층을 의미하고 레이어는 티어 내부의 논리적인 분할을 의미
    - 물리적인 서버 티어의 레이어를 프레젠테이션(presentation), 비즈니스 로직(business logic), 데이터 액세스(data access)의 3개의 논리적인 계층으로 구분할 수 있음
    - 티어 와 레이어

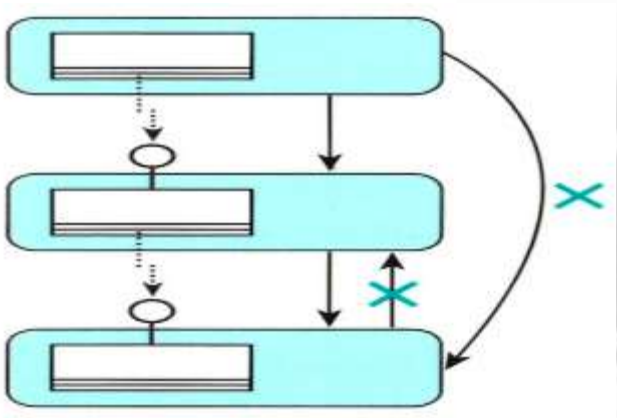
Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - 계층은 설계자들이 복잡한 시스템을 분리할 때 흔히 사용하는 패턴 중 하나로 애플리케이션이 내부에서 처리하는 관심사를 논리적으로 구분
    - 마틴 파울러가 엔터프라이즈 애플리케이션 아키텍처 패턴에서 구분한 레이어드 아키텍처 패턴의 전형적인 유형은 아키텍트가 의도하는 방향에 따라 여러 가지로 구분 가능하나 프레젠테이션, 비즈니스 로직, 데이터 액세스의 3계층으로 구분하는 것이 일반적



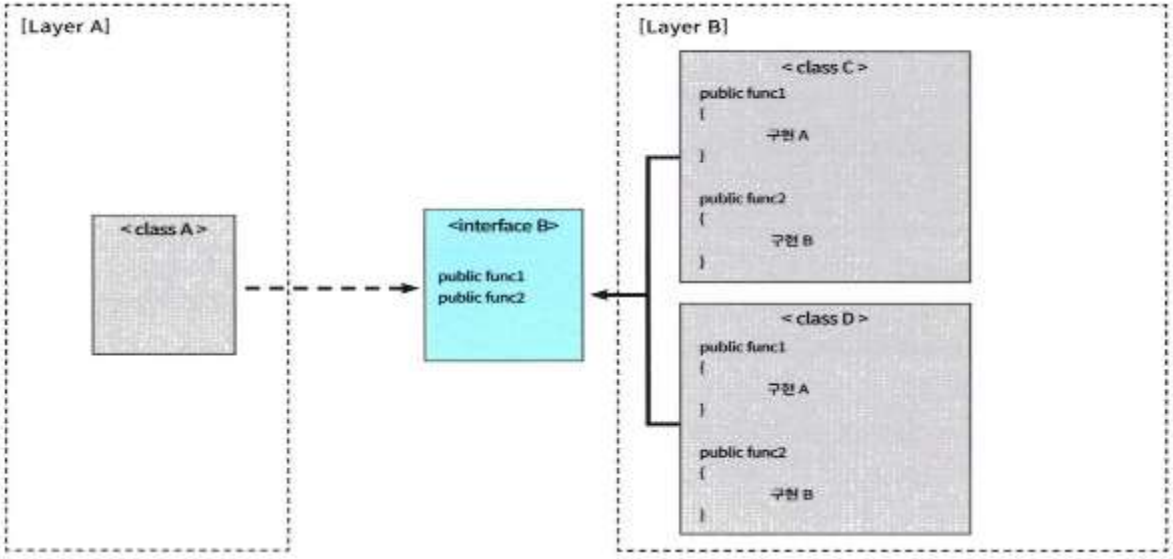
Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - ● 프레젠테이션 층의 관심사는 화면 표현 및 전환 처리이고 비즈니스 로직 층의 관심사는 비즈니스 개념 및 규칙, 흐름 제어이며 데이터 액세스 층의 관심사는 데이터 처리
    - ● 레이어드 아키텍처는 레이어 간 응집성을 높이고 의존도를 낮추기 위해 다음과 같은 몇 가지 규칙을 둠
      - ◆ 상위 계층이 하위 계층을 호출하는 단방향성을 유지
      - ◆ 상위 계층은 하위의 여러 계층을 모두 알 필요없이 바로 밑의 근접 계층만 활용
      - ◆ 상위 계층이 하위 계층에 영향을 받지않게 구성
      - ◆ 하위 계층은 자신을 사용하는 상위 계층을 알지 못하게 구성
      - ◆ 계층 간의 호출은 인터페이스를 통해 호출하는 것이 바람직 - 구현 클래스에 직접 의존하지 않음으로써 약한 결합을 유지



Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - 인터페이스를 통한 의존성 분리는 인터페이스를 구현하는 구현체를 다양하게 해주는 다형성을 추구함으로써 제어 흐름을 간접적으로 전환하게 해줌
    - 상위 계층은 직접적으로 하위 계층을 호출하지 않고 추상적인 인터페이스에 의존하는데 이 경우 하위 계층에서는 추상적 인터페이스를 만족하는 다양한 방식의 구현체를 선택적으로 적용할 수 있음
    - 인터페이스 호출을 통한 다형성 추구



Micro Service Application Architecture

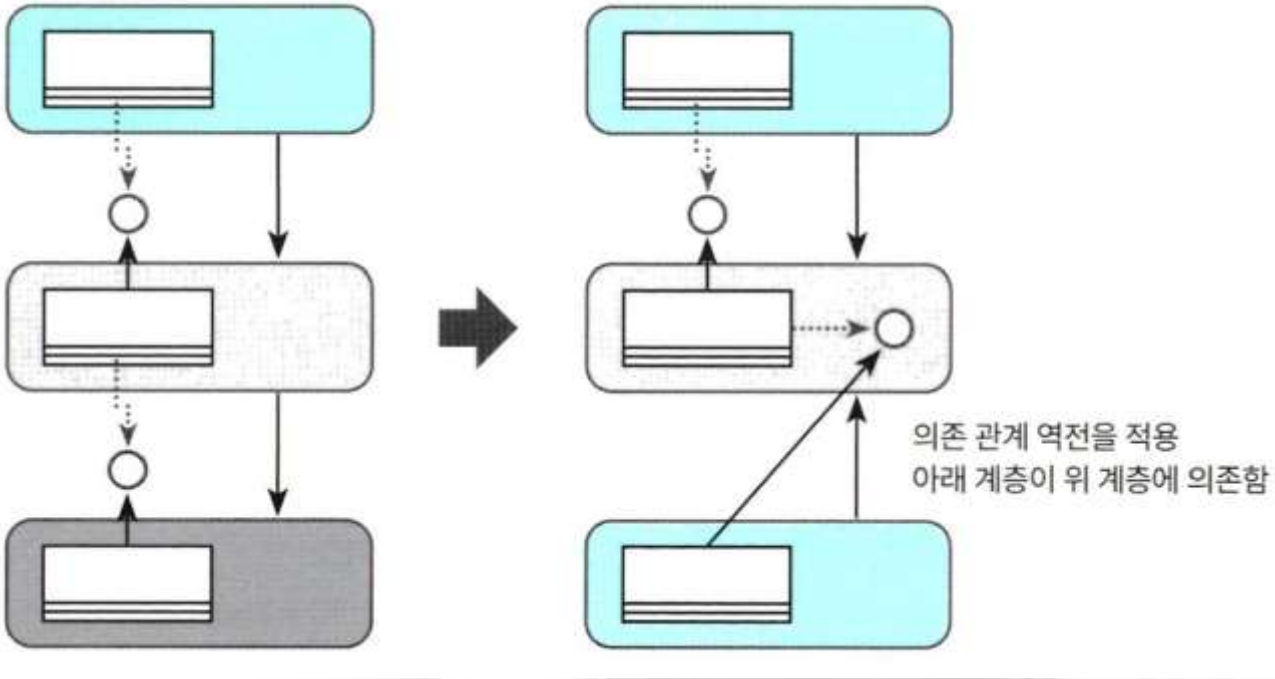
- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - 이러한 방식은 로버트 C. 마틴이 정의한 객체지향 설계 원칙 중 하나인 의존성 역전 원칙(DIP - Dependency Inversion Principle)을 만족하는 것처럼 보임
    - 의존성 역전 원칙은 유연성이 극대화된 시스템에서는 소스코드 의존성이 추상에 의존하며 구체에는 의존하지 않아야 한다 를 뜻하기 때문
    - 그렇지만 개방 폐쇄의 원칙(OCP; Open-Closed Principle)까지 살펴본다면 문제가 있는데 OCP는 소프트웨어 개체는 확장에는 열려 있어야 하고 변경에는 닫혀 있어야 한다는 원칙으로 이 원칙은 개체의 행위는 확장할 수 있어야 하지만 이때 개체를 변경해서는 안 된다는 의미
    - 일반적인 레이어드 아키텍처에서 OCP가 위배되는 까닭은 모든 계층이 각기 자신이 제공하는 기능에 대한 추상적인 인터페이스를 직접 정의하고 소유하고 있는 구조이기 때문인데 이런 구조에서는 제어 흐름(flow of control)이 상위 계층에서 하위 계층으로 흐르게 되고 이에 따른 소스 코드의 의존성은 제어 흐름의 방향대로 따를 수밖에 없음
    - 상위 계층은 하위 계층의 구체 클래스가 아닌 추상 인터페이스에 의존하고 인터페이스의 구현체를 달리하는 방법으로 의존성을 줄이고 다형성은 유지되나 인터페이스는 그 계층이 정의하는 추상 특성의 한계를 벗어날 수 없기 때문에 하위 계층의 유형이 추가되어 확장될 때 닫혀 있어야 할 상위 계층이 하위 계층에서 정의한 특성에 영향을 받게 됨

Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - 프레젠테이션, 비즈니스 로직, 데이터 액세스의 3계층으로 살펴보면 맨 마지막에 있는 데이터 액세스 계층이 변경됐을 때 비즈니스 로직 계층이 변경되면 안되는데 데이터 액세스 계층의 구현체가 클래스 C에서 클래스 D로 교체된 경우에는 비즈니스 로직 계층은 영향을 받지 않겠지만 데이터 액세스 계층의 인터페이스 B가 변경되면 비즈니스 로직 계층의 클래스가 데이터 액세스 계층의 인터페이스에 의존하기 때문에 변경의 영향을 받을 수밖에 없dma
    - 문제는 데이터 액세스 인터페이스의 위치인데 데이터 액세스 인터페이스는 데이터 액세스 계층에 존재하는데 일면 당연해 보이지만 이 위치 때문에 상위 계층이 하위 계층에 의존하게 됨
    - 애플리케이션에서는 비즈니스 로직이 핵심 영역이기 때문에 비즈니스 로직을 보통 고수준 영역이라고 하고 프레젠테이션 계층 및 데이터 액세스 계층을 저수준 영역이라고 하는데 고수준 영역은 핵심 영역이므로 보호를 받아야 하고 저수준 영역의 변경이나 확장에 영향을 받지 않아야 하지만 일반적인 레이어드 아키텍처의 규칙만 따르면 고수준 영역이 저수준 영역에 의존하게 되고 영향을 받게 되는데 여기서 의존성 역전 원칙(DIP)을 제대로 적용할 필요가 생김
    - DIP를 적용해 데이터 액세스 계층에서 정의한 인터페이스를 경계를 넘어 비즈니스 로직 계층으로 옮기면 데이터 액세스 계층의 구현체는 비즈니스 로직의 계층의 인터페이스를 바라볼 수밖에 없는데 데이터 액세스 계층이 구현해야 할 인터페이스를 좀 더 고수준의 비즈니스 로직 계층에서 정의하게 함으로써 기존의 위에서 아래로 흘렀던 의존관계를 역전시키고 고수준 영역이 저수준 영역의 변경에 영향을 받지 않게 하는 것

Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 레이어드 아키텍쳐
    - ● 의존 관계 역전의 적용

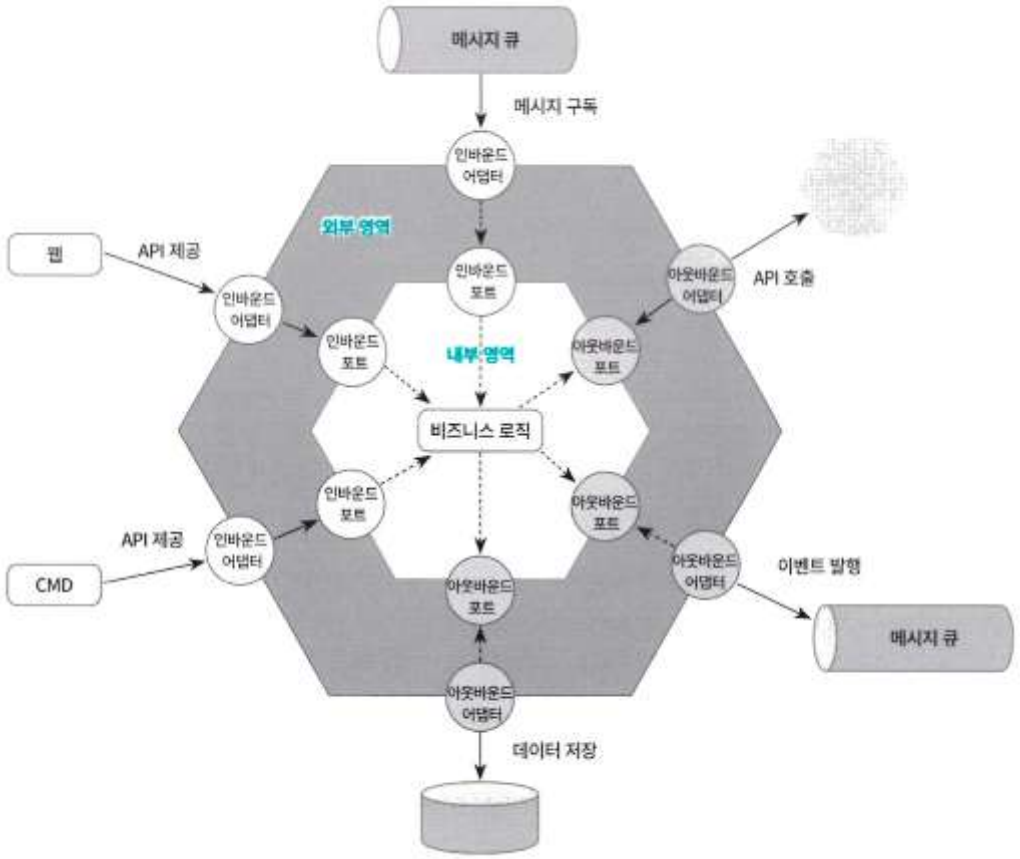


Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 헥사고날 아키텍쳐
    - 레이어드 아키텍처에 DIP를 적용해도 한계가 있는데 프레젠테이션 계층과 데이터 액세스 계층을 보통 저수준 계층으로 정의한다고 했는데 현대 애플리케이션에서는 이러한 두 가지 계층 말고도 다양한 인터페이스를 필요로 하는데 애플리케이션을 호출하는 다양한 시스템의 유형과 애플리케이션과 상호작용하는 다양한 저장소가 존재
    - 단방향 계층구조에서는 이러한 점을 지원하기 힘들기 때문에 다방면으로 열려있는 헥사고날 아키텍처로 이러한 문제점을 해결

Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 헥사고날 아키텍쳐
    - ● 헥사고날 아키텍쳐

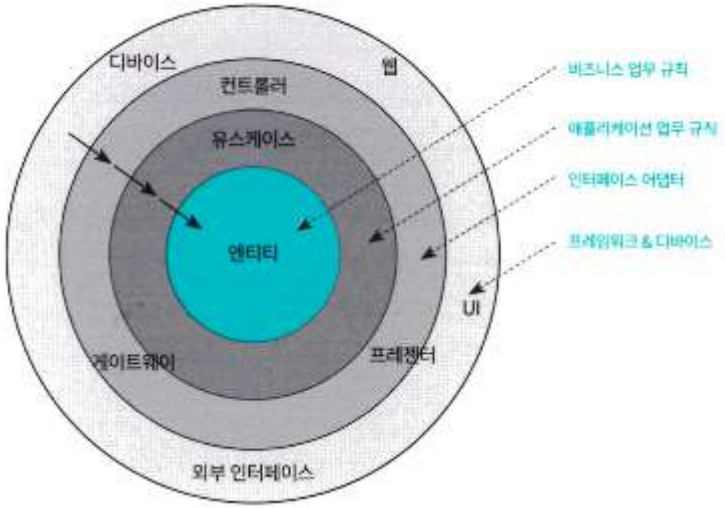


Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 헥사고날 아키텍쳐
    - 헥사고날 아키텍처는 앨리스테어 콕번(Alistair Cockburn)이 제시한 아키텍처로서 포트 앤드 어댑터 아키텍처(ports and adapters architecture)라고도 하는데 간단히 살펴보면 헥사고날 아키텍처에서는 고수준의 비즈니스 로직을 표현하는 내부 영역과 인터페이스 처리를 담당하는 저수준의 외부 영역으로 나누고 내부 영역은 순수한 비즈니스 로직을 표현하는 기술 독립적인 영역이고 외부 영역과 연계되는 포트를 가지고 있으며 외부 영역은 외부에서 들어오는 요청을 처리하는 인바운드 어댑터(inbound adapter)와 비즈니스 로직에 의해 호출되어 외부와 연계되는 아웃바운드 어댑터(outbound adapter)로 구성
    - 헥사고날 아키텍처의 가장 큰 특징은 고수준의 내부 영역이 외부의 구체 어댑터에 전혀 의존하지 않게 한다는 것으로 이를 가능하게 하는 것이 내부 영역에 구성되는 포트
    - 포트는 인바운드/아웃바운드 포트로 구분되는데 인바운드 포트는 내부 영역의 사용을 위해 표출된 API이며 외부 영역의 인바운드 어댑터가 호출하며 아웃바운드 포트는 내부 영역이 외부를 호출하는 방법을 정의하는데 여기서 DIP 원칙과 같이 아웃바운드 포트가 외부의 아웃바운드 어댑터를 호출해서 외부 시스템과 연계하는 것이 아니라 아웃바운드 어댑터가 아웃바운드 포트에 의존해서 구현
    - 외부 영역에 존재하는 어댑터의 종류를 살펴보면 인바운드 어댑터로는 REST API를 발행하는 컨트롤러, 웹 페이지를 구성하는 스프링 MVC 컨트롤러, 커맨드 핸들러, 이벤트 메시지 구독 핸들러 등이 될 수 있고 아웃바운드 어댑터로는 데이터 액세스 처리를 담당하는 DAO, 이벤트 메시지를 발행하는 클래스, 외부 서비스를 호출하는 프락시 등이 될 수 있음

Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 클린 아키텍쳐
    - 클린 아키텍처는 로버트 c. 마틴이 제안한 아키텍처로서 헥사고날 아키텍처의 아이디어와 매우 유사
    - 마틴은 소프트웨어는 행위 가치와 구조 가치의 두 종류의 가치를 가지며 구조 가치가 더 중요하다고 말하는데 소프트웨어를 부드럽게 만드는 것이 구조 가치이기 때문
    - 소프트웨어를 부드럽게 유지하는 방법은 구조 중에서 선택할 수 있는 것을 가능한 한 오랫동안 열어두는 것인데 열어둬야 할 선택 사항은 중요하지 않은 세부 사항
    - 마틴은 클린 아키텍처를 여러 겹으로 둘러싸인 영역으로 표현하며 중앙에서부터 밖으로 엔티티, 유스케이스, 그 외 세부 사항으로 구분



Micro Service Application Architecture

- ❖ 헥사고날 아키텍처와 클린 아키텍처
  - ✓ 클린 아키텍쳐
    - 정중앙에는 엔티티가 있는데 업무 규칙은 사업적으로 수익을 얻거나 비용을 줄일 수 있는 규칙 또는 절차를 의미하며 이러한 업무 규칙은 수동으로 처리할 수 있지만 시스템으로도 자동화할 수 있는데 예를 들면 쇼핑몰의 물건을 사고 파는 규칙, 은행의 이자 계산 규칙, 도서대출 시스템의 대출/반납 규칙 등 모든 시스템에는 해당 도메인의 업무를 규정하는 핵심 업무 규칙이 존재하고 핵심 업무 규칙은 보통 데이터를 요구하므로 핵심 규칙과 데이터는 본질적으로 결합돼 있기 때문에 객체로 쉽게 만들 수 있는데 이러한 유형을 엔티티 객체라 함
    - 엔티티를 감싸는 객체는 유스케이스(use case) 인데 유스케이스는 자동화된 시스템을 사용하는 처리 절차를 기술하는데 유스케이스는 애플리케이션에 특화된 업무 규칙을 표현하며 엔티티 내부의 핵심 업무 규칙을 호출하며 시스템을 사용하는 흐름을 담고 이때 엔티티 같은 고수준 영역은 저수준의 유스케이스 영역을 알게 해서는 안되고 엔티티는 간단한 객체여야 하며 프레임워크 데이터베이스 또는 기타 복잡한 것에 의존해서는 안 되고 유스케이스 객체를 통해서만 조작해야 하고 그 다음으로 유스케이스를 감싸고 있는 나머지 모든 영역이 세부사항
    - 세부사항으로는 입출력 장치, 저장소, 웹 시스템, 서버, 프레임워크, 통신 프로토콜이 될 수 있으며 세부 사항과 유스케이스의 관계를 의존 관계 역전의 원칙을 이용해 플러그인처럼 유연하게 처리해야 하는데 이처럼 명확한 결합의 분리는 테스트 용이성 및 개발 독립성, 배포 독립성을 강화할 수 있음