대용량 웹서비스를 위한 마이크로 서비스 아키텍쳐의 이해

조대협

2014.08.27 21:54

마이크로 서비스 아키텍쳐 (MSA의 이해)

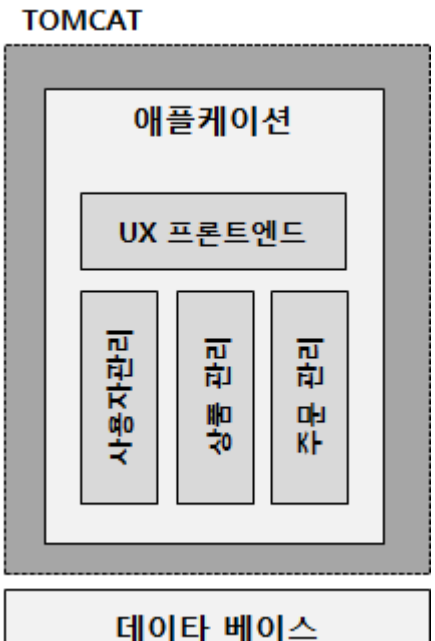
조대협(http://bcho.tistory.com)

배경

마이크로 서비스 아키텍쳐(이하 MSA)는 근래의 웹기반의 분산 시스템의 디자인에 많이 반영되고 있는 아키텍쳐 스타일로, 특정 사람이 정의한 아키텍쳐가 아니라, 분산 웹 시스템의 구조가 유사한 구조로 설계 되면서, 개념적으로만 존재하던 개념이다.
얼마전 마틴파울러(Martin folwer)가 이에 대한 MSA에 대한 개념을 글로 정리하여, 개념을 정립 시키는데 일조를 하였다.
이 글에서는 대규모 분산 웹시스템의 아키텍쳐 스타일로 주목 받고 있는 MSA에 대한 개념에 대해서 알아보도록 한다.

모노리틱 아키텍쳐(Monolithic Architecture)

마이크로 서비스 아키텍쳐를 이해하려면 먼저 모노리틱 아키텍쳐 스타일에 대해서 이해해야 한다
모노리틱 아키텍쳐 스타일은 기존의 전통적인 웹 시스템 개발 스타일로, 하나의 애플리케이션 내에 모든 로직들이 모두 들어 가 있는 “통짜 구조” 이다.
예를 들어, 온라인 쇼핑몰 애플리케이션이 있을때, 톰캣 서버에서 도는 WAR 파일(웹 애플리케이션 패키징 파일)내에, 사용자 관리,상품,주문 관리 모든 컴포넌트들이 들어 있고 이를 처리하는 UX 로직까지 하나로 포장되서 들어가 있는 구조이다.



각 컴포넌트들은 상호 호출을 함수를 이용한 call-by-reference 구조를 취한다.
전체 애플리케이션을 하나로 처리하기 때문에, 개발툴 등에서 하나의 애플리케이션만 개발하면 되고, 배포 역시 간편하며 테스트도 하나의 애플리케이션만 수행하면 되기 때문에 편리하다.
문제점
그러나 이러한 모노리틱 아키텍쳐 시스템은 대형 시스템 개발시 몇가지 문제점을 갖는다.
모노리틱 구조의 경우 작은 크기의 애플리케이션에서는 용이 하지만, 규모가 큰 애플리케이션에서는 불리한 점이 많다.
크기가 크기 때문에, 빌드 및 배포 시간, 서버의 기동 시간이 오래 걸리며 (서버 기동에만 2시간까지 걸리는 사례도 경험해봤음)
프로젝트를 진행하는 관점에서도, 한두사람의 실수는 전체 시스템의 빌드 실패를 유발하기 때문에, 프로젝트가 커질 수 록, 여러 사람들이 협업 개발하기가 쉽지 않다
또한 시스템 컴포넌트들이 서로 로컬 콜 (call-by-reference)기반으로 타이트하게 연결되어 있기 때문에, 전체 시스템의 구조를 제대로 파악하지 않고 개발을 진행하면, 특정 컴포넌트나 모듈에서의 성능 문제나 장애가 다른 컴포넌트에까지 영향을 주게 되며, 이런 문제를 예방하기 위해서는 개발자가 대략적인 전체 시스템의 구조 등을 이해 해야 하는데, 시스템의 구조가 커질 수 록, 개인이 전체 시스템의 구조와 특성을 이해하는 것은 어려워진다.
특정 컴포넌트를 수정하고자 했을때, 컴포넌트 재 배포시 수정된 컴포넌트만 재 배포 하는 것이 아니라 전체 애플리케이션을 재 컴파일 하여 전체를 다시 통으로 재배포 해야하기 때문에 잦은 배포가 있는 시스템의 경우 불리하며,
컴포넌트 별로, 기능/비기능적 특성에 맞춰서 다른 기술을 도입하고자 할때 유연하지 않다. 예를 들어서, 전체 애플리케이션을 자바로 개발했다고 했을 때, 파일 업로드/다운 로드와 같이 IO 작업이 많은 컴포넌트의 경우 node.js를 사용하는 것이 좋을 수 있으나, 애플리케이션이 자바로 개발되었기 때문에 다른 기술을 집어 넣기가 매우 어렵다.
※ 모노리틱 아키텍쳐가 꼭 나쁘다는 것이 아니다. 규모가 작은 애플리케이션에서는 배포가 용이하고, 규모가 크더라도, call-by-reference call에 의해서 컴포넌트간 호출시 성능에 제약이 덜하며, 운영 관리가 용이하다. 또한 하나의 구조로 되어 있기 때문에, 트렌젝션 관리등이 용이하다는 장점이 있다. 즉 마이크로 서비스 아키텍쳐가 모든 부분에 통용되는 정답은 아니며, 상황과 필요에 따라서 마이크로 서비스 아키텍쳐나 모노리틱 아키텍쳐를 적절하게 선별 선택 또는 변형화 해서 사용할 필요가 있다.

마이크로 서비스 아키텍쳐

마이크로 서비스 아키텍쳐는 대용량 웹서비스가 많아짐에 따라 정의된 아키텍쳐인데, 그 근간은 SOA (Service Oriented Architecture : 서비스 지향 아키텍쳐)에 두고 있다.
SOA는 엔터프라이즈 시스템을 중심으로 고안된 아키텍쳐라면, 마이크로 서비스 아키텍쳐는 SOA 사상에 근간을 두고, 대용량 웹서비스 개발에 맞는 구조로 사상이 경량화 되고, 대규모 개발팀의 조직 구조에 맞도록 변형된 아키텍쳐이다.

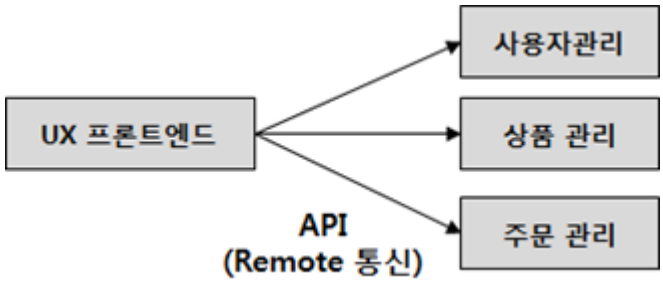
아키텍쳐 구조

서비스

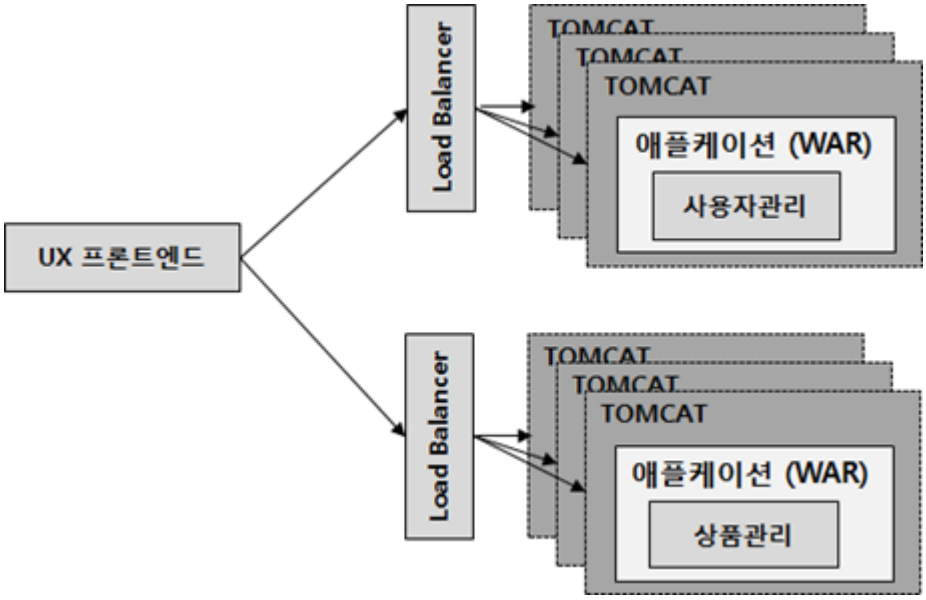
마이크로 서비스 아키텍쳐에서는 각 컴포넌트를 서비스라는 개념으로 정의한다. 서비스는 데이타에서 부터 비지니스 로직까지 독립적으로 상호 컴포넌트간의 의존성이 없이 개발된 컴포넌트(이를 버티컬 슬라이싱/Vertical Slicing-수직적 분할이라고 한다.)로 REST API와 같은 표준 인터페이스로 그 기능을 외부로 제공한다.
서비스 경계는 구문 또는 도메인(업무)의 경계를 따른다. 예를 들어 사용자 관리, 상품 관리, 주문 관리와 같은 각 업무 별로 서비스를 나눠서 정의한다. 사용자/상품 관리 처럼 여러개의 업무를 동시에 하나의 서비스로 섞어서 정의하지 않는다.
REST API에서 /users, /products와 같이 주요 URI도 하나의 서비스 정의의 범위로 좋은 예가 된다.

마이크로 서비스 아키텍쳐의 구조

마이크로 서비스 아키텍쳐의 구조는 다음과 같은 모양을 따른다.
각 컴포넌트는 서비스라는 형태로 구현되고 API를 이용하여 타 서비스와 통신을 한다.



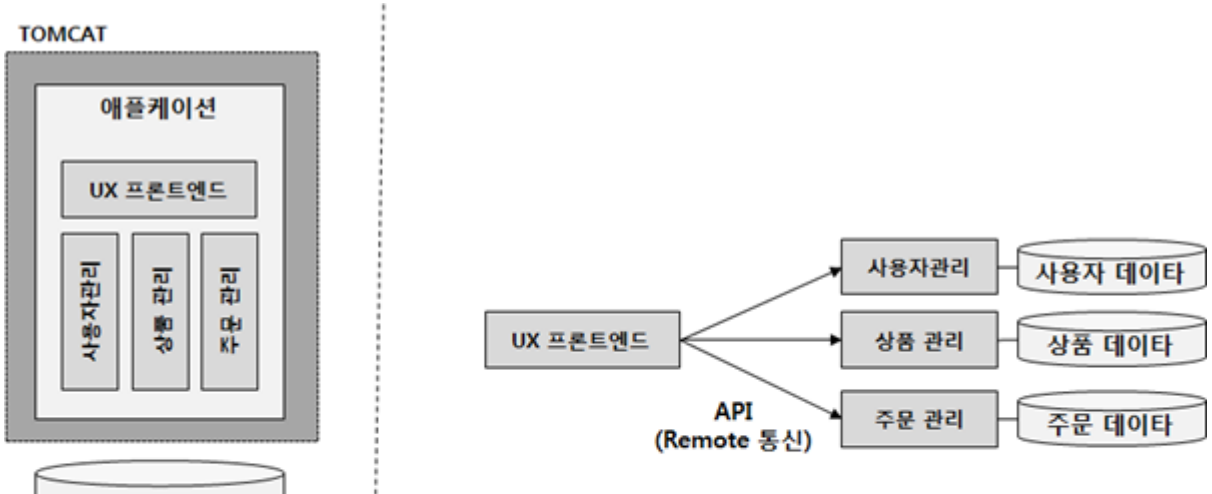
배포 구조관점에서도 각 서비스는 독립된 서버로 타 컴포넌트와의 의존성이 없이 독립적으로 배포 된다.
예를 들어 사용자 관리 서비스는 독립적인 war파일로 개발되어, 독립된 톰캣 인스턴스에 배치된다. 확장을 위해서 서비스가 배치된 톰캣 인스턴스는 횡적으로 스케일 (인스턴스 수를 더함으로써)이 가능하고, 앞단에 로드 밸런서를 배치하여 서비스간의 로드를 분산 시킨다.



가장 큰 특징이, 애플리케이션 로직을 분리해서 여러개의 애플리케이션으로 나눠서 서비스화하고, 각 서비스별로 톰캣을 분산 배치한 것이 핵심이다.

데이타 분리

데이타 저장관점에서는 중앙 집중화된 하나의 통 데이타 베이스를 사용하는 것이 아니라 서비스 별로 별도의 데이타 베이스를 사용한다. 보통 모노리틱 서비스의 경우에는 하나의 통 데이타 베이스 (보통 RDBMS를 사용) 하는 경우가 일반적이지만, 마이크로 서비스 아키텍쳐의 경우, 서비스가 API에서 부터 데이타 베이스까지 분리되는 수직 분할 원칙 (Vertical Slicing)에 따라서 독립된 데이타 베이스를 갖는다.



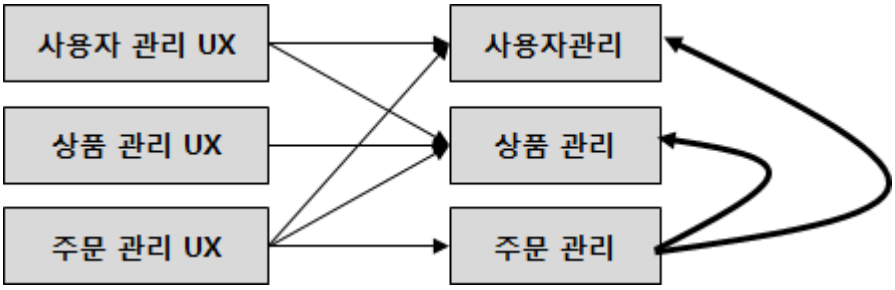
모노리틱 구조에서
데이타 저장 방식

마이크로 서비스 아키텍처에서
데이타 저장 방식

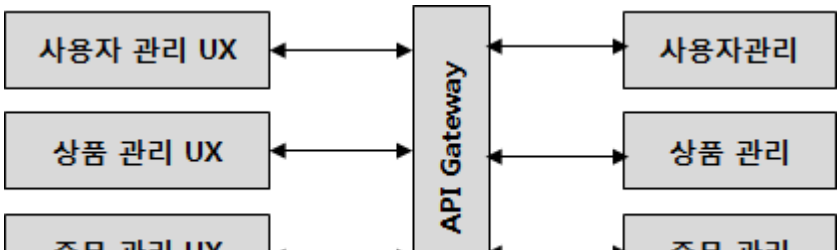
데이타 베이스의 종류 자체를 다른 데이타 베이스를 사용할 수 도 있지만, 같은 데이타 베이스를 사용하더라도 db를 나누는 방법을 사용한다.
이 경우, 다른 서비스 컴포넌트에 대한 의존성이 없이 서비스를 독립적으로 개발 및 배포/운영할 수 있다는 장점을 가지고 있으나, 다른 컴포넌트의 데이타를 API 통신을 통해서만 가지고 와야 하기 때문에 성능상 문제를 야기할 수 있고, 또한 이 기종 데이타 베이스간의 트렌젝션을 묶을 수 없는 문제점을 가지고 있다. (이러한 데이타 분산에 의한 트렌젝션 문제는 SOA 때부터 있어 왔다.) 데이타 분산으로 인한 트렌젝션 문제는 뒤에서 조금 더 자세하게 설명하도록 한다.

API Gateway

마이크로 서비스 아키텍쳐 설계에 있어서 많이 언급되는 컴포넌트 중의 하나가 api gateway 라는 컴포넌트 이다. api gateway는 마치 프록시 서버 처럼 api들 앞에서 모든 api에 대한 end point를 통합하고, 몇가지 추가적인 기능을 제공하는 미들웨어로, SOA의 ESB (Enterprise Service Bus)의 경량화 버전이다. Apigateway가 마이크로 서비스 아키텍쳐 상에서 수행하는 주요 기능을 살펴보면 다음과 같다.
EndPoint 통합 및 토폴로지 정리
마이크로 서비스 아키텍쳐의 문제점 중의 하나는 각 서비스가 다른 서버에 분리 배포 되기 때문에, API의 End point 즉, 서버의 URL이 각기 다르다는 것이다.
사용자 컴포넌트는 http://user.server.com, 상품 컴포넌트는 http://product.server.com 과 같은 분리된 URL을 사용하는데, 이는 API 사용자 경험 관점에서도 사용하기가 불편하다. 특히 마이크로 서비스 아키텍쳐는 컴포넌트를 되도록이면 업무 단위로 잘게 짜르는 fine grained (작은 덩어리)의 서비스를 지향하기 때문에, 컴포넌트의 URL 수는 더 많이 늘어 날 수 있다.
API를 사용하는 클라이언트에서 서버간의 통신이나, 서버간의 API 통신의 경우 p2p(Point to Point)형태로 토폴로지가 복잡해지고 거미줄 모양의 서비스 컴포넌트간의 호출 구조는 향후 관리의 문제를 일으킬 수 있다. 하나의 end point를 변경하였을때, 제대로 관리가 되지 않을 경우가 있다.



<그림. P2P 형태의 토폴리지>
이러한 토폴로지상의 문제점을 해결하기 위해서 중앙에 서비스 버스와 같은 역할을 하는 채널을 배치 시켜서, 전체 토폴로지를 p2p에서 hub & spoke 방식으로 변환 시켜서, 서비스간 호출을 단순화 시킬 수 있다.



<그림. 버스 기반의 Hub & Spoke 토폴리지>

Orchestration

다른 기능으로는 orchestration 이라는 개념이 있다. 기존 open api의 mash up과 같은 개념으로, 여러개의 서비스를 묶어서 하나의 새로운 서비스를 만드는 개념이다.
예를 들어, 포인트 적립과, 물품 구매라는 서비스가 있을때, 이 두개의 서비스를 묶어서 “물품 구입시 포인트 적립”이라는 새로운 서비스를 만들어 낼 수 있다. 이러한 orchestration 기능은, api gateway를 통해서 구현될 수 있다.
이는 마이크로 서비스 아키텍쳐가 서비스 자체가 fine grained 형태로 잘게 쪼게졌기 때문에 가능한 일인데, 사실 orchestration을 api gateway 계층에서 하는 것은 gateway 입장에서 부담이 되는 일이다. 실제로 과거의 SOA 시절에 많은 ESB(Enterprise Service Bus) 프로젝트가 실패한 원인 중의 하나가 과도한 orchestration 로직을 넣어서 전체적인 성능 문제를 유발한 경우가 많았다. 그래서 orchestration 서비스의 활용은 마이크로 서비스 아키텍쳐에 대한 높은 이해와 api gateway 자체에 대한 높은 수준의 기술적인 이해를 필요로 한다.
실제로 넷플릭스의 경우 마이크로 서비스 아키텍쳐를 사용하면서, 여러개의 서비스들을 gateway 계층을 통해서 orchestration 하는 모델을 사용하고 있다.

공통 기능 처리 (Cross cutting function handling)

또한 API에 대한 인증 (Authentication)이나, Logging과 같은 공통 기능에 대해서 서비스 컴포넌트 별로 중복 개발해야 하는 비효율성을 유발할 수 있다. api gateway에서 이러한 공통 기능을 처리하기 되면, api 자체는 비지니스 로직에만 집중을 하여 개발에 있어서의 중복등을 방지 할 수 있다.
mediation
이외에도 XML이나 네이티브 메세지 포맷을 json등으로 상호 변환해주는 message transformation 기능이나, 프로토콜을 변환하는 기능, 서비스간의 메세지를 라우팅해주는 기능등 여러가지 고급 mediation 기능을 제공을 하지만, api gateway를 최대한 가볍게 가져간다는 설계 원칙 아래서 가급 적이면 고급적인 mediation 기능을 사용할 때는 높은 수준의 설계와 기술적인 노하우를 동반해야 한다.

※ ESB vs APIgateway
SOA 프로젝트의 실패중의 하나가 ESB로 꼽히는 경우가 많은데, 이는 ESB를 Proxy나 Gateway처럼 가벼운 연산만이 아니라, 여러개의 서비스를 묶는 로직에 무겁게 사용했기 때문이다. (사용하면 안된다는 것이 아니라 잘 사용해야 한다는 것이다.) ESB는 메세지를 내부적으로 XML로 변환하여 처리하는데, XML 처리는 생각하는것 보다 파싱에 대한 오버헤드가 매우 크다. 또한 ESB의 고유적인 버스나 게이트웨이로써의 특성이 아니라 타 시스템을 통합 하기 위한 EAI적인 역할을 ESB를 이용해서 구현함으로써 많은 실패 사례를 만들어 내었다. 그래서 종종 ESB는 Enterprise Service Bus가 아니라 EnterpriSe nightmare Bus로 불리기도 한다. J
이러한 개념적인 문제를 해결하기 위해서 나온 제품군이 apigateway라는 미들웨어 제품군들인데, ESB와 기본적인 특성은 유사하나 기능을 낮추고 EAI의 통합 기능을 제거하고 API 처리에만 집중한 제품군들로, 클라우드상에서 작동하는 PaaS (Platform As A Service)형태의 서비스로는 apigee.com이나 3scale.com 등이 있고, 설치형 제품으로는 상용 제품인 CA社의 Layer7이나 오픈소스인 Apache Service Mix, MuleSoft의 ESB 제품 그리고 WSO2의 API Platform 등이 있다.
Apigateway 부분에 마이크로 서비스 아키텍쳐의 다른 부분 보다 많은 부분을 할애한 이유는, 컴포넌트를 서비스화하는 부분에 까지는 대부분 큰 문제가 없이 적응을 하지만 apigateway의 도입 부분의 경우, 내부적인 많은 잡음이 날 수 있고, 또한 도입을 했더라도 잘못된 설계나 구현으로 인해서 실패 가능성이 비교적 높은 모듈이기 때문이다. 마이크로 서비스 아키텍쳐의 핵심 컴포넌트이기도 하지만, 도입을 위해서는 팀의 상당 수준의 높은 기술적인 이해와 개발 능력을 필요로 한다.

배포

마이크로 서비스 아키텍쳐의 가장 큰 장점 중의 하나가 유연한 배포 모델이다. 각 서비스가 다른 서비스와 물리적으

로 완벽하게 분리되기 때문에 변경이 있는 서비스 부분만 부분 배포가 가능하다 예를 들어서, 사용자 관리 서비스 로직이 변경되었을 때, 모노리틱 아키텍쳐의 경우에는 전체 시스템을 재 배포해야 하지만, 마이크로 서비스 아키텍쳐의 경우에는 변경이 있는 사용자 관리 서비스 부분만 재 배포 하면 되기 때문에, 빠르고 전체 시스템의 영향도를 최소화한 수준에서 배포를 진행할 수 있다.

확장성

서비스 별로 독립된 배포 구조는 확장성에 있어서도 많은 장점을 가지고 오는데, 부하가 많은 특정 서비스에 대해서만 확장이 가능하여 조금 더 유연한 확장 모델을 가질 수 있다. 모노리틱 아키텍쳐의 경우에는 특정 서비스의 부하가 많아서 성능 확장이 필요할때, 전체 서버의 수를 늘리거나 각 서버의 CPU 수를 늘려줘야 하지만, 마이크로 서비스 아키텍쳐의 경우에는 부하를 많이 받는 서비스 컴포넌트 만 확장을 해주면 된다.

Conway's Law (컨웨이의 법칙)

마이크로 서비스 아키텍쳐의 흥미로운 점중의 하나는 아키텍쳐 스타일의 조직 구조나 팀 운영 방식에 영향을 준다는 것인데, 마이크로 서비스 아키텍쳐는 컨웨이의 법칙에 근간을 두고 있다.
컨웨이의 법칙은 "소프트웨어의 구조는 그 소프트웨어를 만드는 조직의 구조와 일치한다"는 이론이다.
현대의 소프트웨어 개발은 주로 애자일 방법론을 기반으로 하는 경우가 많다. 애자일 팀의 구조는 2 피자팀(한팀의 인원수는 피자 두판을 먹을 수 있는 정도의 인원 수가 적절하다.)의 모델을 많이 따르는데, 한 팀이 7~10명정도로 이루어지고, 이 인원 수가 넘어가면 팀을 분리하는 모델이다.
마이크로 서비스 아키텍쳐는 각 컴포넌트를 팀에 배치해서 책임지고 개발하는 것을 근간으로 하며, 팀간의 의존성을 제거해서 각 팀이 컴포넌트 개발을 독립적으로할 수 있는 구조로 잡혀있다.

마이크로 서비스 아키텍쳐의 문제점

분홍빛 미래 처럼 보이는 마이크로 서비스 아키텍쳐는 아무런 문제가 없는 것일까? 당연히 여러가지 장점을 제공하는 대신에 그만한 단점을 가지고 있다.

성능

모노리틱 아키텍쳐는 하나의 프로세스 내에서 서비스간의 호출을 call-by-reference 모델을 이용한다. 반면 마이크로 서비스 아키텍쳐는 서비스간의 호출을 API 통신을 이용하기 때문에 값을 json이나 xml에서 프로그래밍에서 사용하는 데이타 모델 (java object등)으로 변환하는 marsharing 오버헤드가 발생하고 호출을 위해서 이 메세지들이 네트워크를 통해서 전송되기 때문에 그만한 시간이 더 추가로 소요된다.

메모리

마이크로 서비스 아키텍쳐는 각 서비스를 독립된 서버에 분할 배치하기 때문에, 중복되는 모듈에 대해서 그만큼 메모리 사용량이 늘어난다.
예를 들어 하나의 톰캣 인스턴스에서 사용자 관리와 상품 관리를 배포하여 운용할 경우, 하나의 톰캣을 운영하는데 드는 메모리와, 스프링 프레임웍과 같은 라이브러리를 사용하는데 소요되는 메모리 그리고 각각의 서비스 애플리케이션이 기동하는 메모리가 필요하다.
그러나 마이크로 서비스 아키텍쳐로 서비스를 배포할 경우 사용자 관리 서비스 배포와 상품 관리 서비스 배포를 위한 각각의 별도의 톰캣 인스턴스를 운용해야 하고, 스프링 프레임웍과 같은 공통 라이브러리도 각각 필요하기 때문에, 배포하고자 하는 서비스의 수 만큼 중복된 양의 메모리가 필요하게 된다.
위의 두 문제는 반드시 발생하는 문제점이기는 하나 현대의 인프라 환경에서는 크게 문제는 되지 않는다. (기존에 비해 상대적으로). 현대의 컴퓨팅 파워 자체가 워낙 발달하였고, 네트워크 인프라 역시 기존에 1G등에 비해서 내부 네트워크는 10G를 사용하는 등, 많은 성능상 발전이 있었다. 또한 메모리 역시 비용이 많이 낮춰지고 32bit에서 64bit로 OS들이 바뀌면서, 가용 메모리 용량이 크게 늘어나서 큰 문제는 되지 않는다. 또한 성능상의 문제는 비동기 패턴이나 캐싱등을 이용해서 해결할 수 있는 다른 방안이 많기 때문에 이 자체는 큰 문제가 되지 않는다.
그보다 더 문제점은 아래에서 언급하는 내용들인데

테스팅이 더 어려움

마이크로 서비스 아키텍쳐의 경우 서비스들이 각각 분리가 되어 있고, 다른 서비스에 대한 종속성을 가지고 있기 때문에, 특정 사용자 시나리오나 기능을 테스트하고자 할 경우 여러 서비스에 걸쳐서 테스트를 진행해야 하기 때문에 테스트 환경 구축이나 문제 발생시 분리된 여러개의 시스템을 동시에 봐야 하기 때문에 테스팅의 복잡도가 올라간다.
운영 관점의 문제

운영 관점에서는 서비스 별로 서로 다른 기술을 사용할 수 있으며, 시스템이 아주 잘게 서비스 단위로 쪼게 지기 때문에 운영을 해야할 대상 시스템의 개수가 늘어나고, 필요한 기술의 수도 늘어나게 된다.

서비스간 트렌젝션 처리

구현상의 가장 어려운 점중의 하나가, 트렌젝션 처리이다. 모노리틱 아키텍쳐에서는 RDBMS를 사용하면서 하나의 애플리케이션 내에서 트렌젝션이 문제가 있으면 쉽게 데이타베이스의 기능을 이용해서 rollback을 할 수 있었다. 여러개의 데이타베이스를 사용하더라도, 분산 트렌젝션을 지원하는 트렌젝션 코디네이터 (JTS – Java Transaction Service)등을 이용해서 쉽게 구현이 가능했는데, API 기반의 여러 서비스를 하나의 트렌젝션으로 묶는 것은 불가능 하다.
쉽게 예를 들어서 설명을 하면, 계좌에서 돈을 빼는 서비스와, 계좌에 돈을 넣는 서비스가 있다고 하자. 이 둘은 API를 expose했을 때, 계좌에서 돈을 뺀 후, 계좌에 돈을 넣기 전에 시스템이 장애가 나면, 뺀 돈은 없어지게 된다. 모노리틱 아키텍쳐를 사용했을 경우에는 이러한 문제를 트렌젝션 레벨에서 롤백으로 쉽게 해결할 수 있지만 API 기반의 마이크로 서비스 아키텍쳐에서는 거의불가능하다.
사실 이 문제는 마이크로 서비스 아키텍쳐 이전에도, 서비스와 API를 기본 컨셉으로 하는 SOA에도 있었던 문제이다.
이러한 문제를 해결하기 위해서 몇가지 방안이 있는데,
그 첫번째 방법으로는 아예 애플리케이션 디자인 단계에서 여러개의 API를 하나의 트렌젝션으로 묶는 분산 트렌젝션 시나리오 자체를 없애는 방안이다. 분산 트렌젝션이 아주 꼭 필요할 경우에는 차라리 모노리틱 아키텍쳐로 접근하는 것이 맞는 방법이다. 앞서도 언급했듯이 마이크로 서비스 아키텍쳐의 경우, 금융이나 제조와 같이 트렌젝션 보장이 중요한 엔터프라이즈 시스템보다는 대규모 처리가 필요한 B2C 형 서비스에 적합하기 때문에, 아키텍쳐 스타일 자체가 트렌젝션을 중요시 하는 시나리오에서는 적절하지 않다.
그럼에도 불구하고, 트렌젝션 처리가 필요할 경우, 트렌젝션 실패시 이를 애플리케이션 적으로 처리해 줘야 하는데, 이를 보상 트렌젝션(compensation transaction)이라고 한다. 앞의 계좌 이체 시나리오에서 돈을 뺀 후, 다른 계좌에 넣다가 에러가 났을 경우에, 명시적으로, 돈을 원래 계좌로 돌려주는 에러 처리 로직을 구현해야 한다.
마지막 방법으로 복합 서비스 (composite service)라는 것을 만들어서 활용하는 방법이 있는데, 복합 서비스란 트렌젝션을 묶어야 하는 두개의 시스템을 트렌젝션을 지원하는 네이티브 프로토콜을 이용해서 구현한 다음 이를 API로 노출 시키는 방법이다.
두개의 데이타 베이스는 XA(eXtended Architecture)와 같은 분산 트렌젝션 프로토콜을 써서 서비스를 개발하거나 또는 SAP나 Oracle 아답터와 같이 트렌젝션을 지원하는 네이티브 아답터를 사용하는 방법이다. 기존에 SOA에서 많이 했던 접근방법이기는 하나, 복합 서비스를 사용할 경우, 복합서비스가 서로 다른 두개의 서비스에 걸쳐서 tightly coupled하게 존재하기 때문에, 마이크로 서비스 아키텍쳐의 isolation(상호 독립적)인 사상에 위배되고 서비스 변경시에 이 부분을 항상 고려해야 하기 때문에 아키텍쳐상의 유연성이 훼손되기 때문에 꼭 필요하지 않은 경우라면 사용하지 않는 것이 좋다.

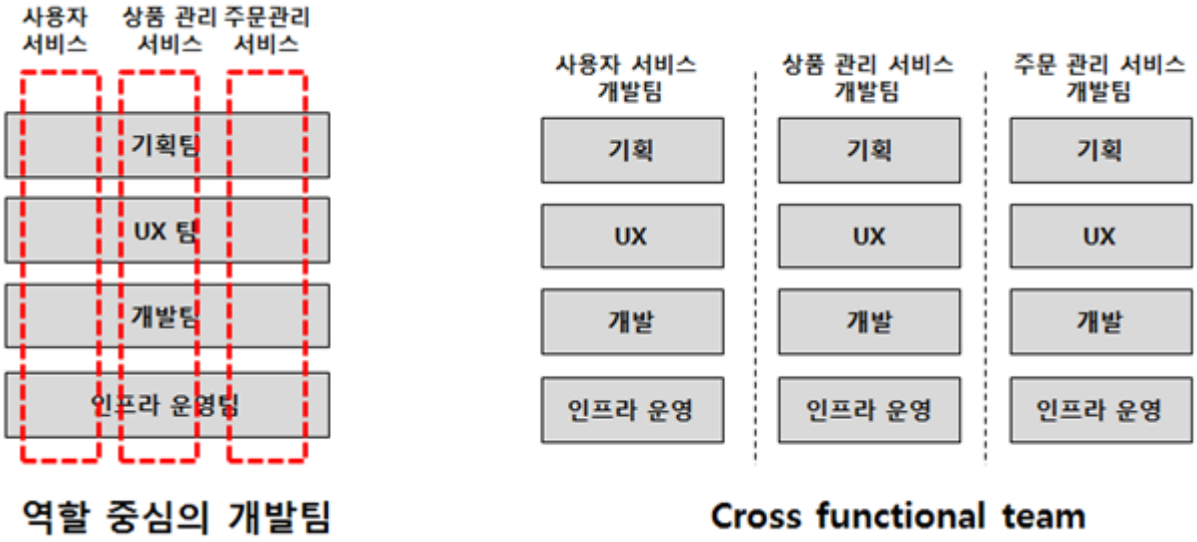
거버넌스 모델

거버넌스 (governance)란, 시스템을 개발하는 조직의 구조나 프로세스를 정의한 것으로, 일반적으로 중앙 집중화된 조직에서 표준 프로세스와 가이드를 기반으로 전체 팀을 운용하는 모델을 사용한다. 이를 중앙 집중형 거버넌스 모델 (Centralized governance model) 이라고 하는데, 이 경우 전체 시스템이 동일한 프로세스와 기술을 가지고 개발이 되기 때문에, 유지 보수가 용이하고 팀간의 인원 교체등이 편리하다는 장점을 가지고 있다. 전통적인 개발

모델들은 이러한 중앙 집중형 거버넌스 모델을 사용한다.
그러나 현대의 웹 개발의 경우, 오픈 소스 발달로 선택 가능한 기술들이 많고 각 요구 사항에 따라서 효율성 측면등을 고려할때 각각 최적화된 기술을 사용하는 것이 좋은 경우가 있다.
예를 들어, 전체 표준을 자바+RDBMS로 정했다 하더라도, 파일 업로드 다운로드 관련 컴포넌트는 io 성능과 많은 동시접속자를 처리할 수 있는 node.js가 유리하다던지, 데이타의 포맷은 복잡하지만, 복잡한 쿼리가 없을 경우에는 json document 기반의 mongodb와 같은 NoSQL등이 유리한 사례 등이 된다. 이러한 기술을 도입하기 위해서는 중앙 집중형 거버넌스 모델에서는 모든 개발팀을 교육 시키고, 운영 또한 준비를 해야하기 때문에 기술에 대한 적용 민첩성이 떨어지게 된다.
이러한 문제점을 해결 하는 거버넌스 모델이 분산형 거버넌스 모델 (De-Centralized governance model)인데, 이는 각 팀에 독립적인 프로세스와 기술 선택 권한을 주는 모델로, 각 서비스가 표준 API로 기능을 바깥으로 노출할 뿐 내부적인 구현 기술 구조는 추상화되어 가능한 사상이다
분산형 거버넌스 모델을 수행하려면, 이에 맞는 팀구조가 필요한데, 이 팀 구조는 다음과 같은 몇가지 특징을 가지고 있어야 한다.

Cross functional team

기존의 팀 모델은 역할별로 나뉘어진 모델로 팀을 구분한다. 기획팀,UX팀,개발팀,인프라 운영팀 등 공통적인 특성으로 나누는 것이 기존의 팀 모델이다. 이런 팀 모델은 리소스의 운영에 유연성을 부여한다. 개발 인력이 모자르면 팀 내에서 개발인원을 다른 프로젝트에서 충당하는 등의 리소스 운영이 가능하지만 반대로 팀간의 커뮤니케이션이 팀이라는 경계에 막혀서 원할하지 않고 협의에 걸리는 시간으로 인해서 팀의 운영 속도가 떨어진다.
cross function team 모델은 하나의 팀에 UX, 개발팀,인프라팀등 소프트웨어 시스템을 개발하는데 필요한 모든 역할을 하나의 팀에 구성하고 움직이는 모델로, 각 서비스 개발팀이 cross functional team이 되서 움직인다.



<그림 역할 중심의 개발팀 과 cross functional team에 대한 모델 비교>
이 경우 서비스 기획에서 부터 설계,개발,운영이 가능해지고 다른 팀에 대한 의존성이 없어짐으로써 빠른 서비스 개발이 가능해진다.

You build,You run-Devops

기술에 대한 독립성을 가지려면 구현 뿐만 아니라 운영 또한 직접할 수 있는 능력을 가져야 한다. 그래서 개발과 운영을 하나의 조직에 합쳐 놓는 구조를 Devops라고 한다.
Devops는 Development와 Operation을 합성한 단어로, 개발팀과 운영팀이 다른 팀으로 분리되어 있어서 발생하는 의사 소통의 문제점을 해결하고, 개발이 운영을 고려하고, 운영에서 발생하는 여러 문제점과 고객으로부터의 피드백을 빠르게 수용하여 서비스 개선에 반영하는 개발 모델이다.
이런 모델이 가능해진 이유는 운영팀만의 고유 영역이었던 인프라에 대한 핸들링이 클라우드의 도입으로 인해서 쉬워져서, 애플리케이션 개발자도 웹사이트를 통해서 손쉽게 디스크나 네트워크 설정, 서버 설정등이 가능해졌기 때문이다.
Devops는 대단히 좋은 모델이고 아마존이나 넷플릭스등이 적용하고 있는 모델이기는 하나, 이 역시 대단히 높은

Devops는 대단히 좋은 모델이고 아마존이나 넷플릭스등이 적용하고 있는 모델이기는 하나, 이 역시 대단히 높은 수준의 팀의 성숙도가 필요하다.
개발자가 애플리케이션 개발 뿐만 아니라, 인프라에 대한 설계 및 운영까지 담당해야 하기 때문에 기존의 애플리케이션만 개발하던 입장에서는 대단히 부담이 되는 일이다.
좋은 모델이기는 하지만 충분히 준비가 되지 않은 상태에서 넘어가게 되면은 운영상의 많은 장애를 유발하기 때문에, 팀의 성숙도에 따라서 심각하게 고민해보고 적용을 해보기를 권장한다.

Project vs product

분산형 거버넌스 모델에서 중요한 점 중의 하나는 연속성이다. 거버넌스를 분산 시켜버렸기 때문에 팀별로 다른 형태의 표준과 기술 프로세스를 통해서 개발을 하기 때문에, 새로운 인원이 들어오거나 다른 팀으로 인원이 이동하였을 경우 팀에 맞는 형태의 재 교육이 필요하고 그간의 축적된 노하우가 100% 활용되지 못할 수 가 있기 때문에 가능하면 팀원들은 계속해서 해당 서비스 개발에 집중할 필요가 있다.
이를 위해서는 프로젝트의 컨셉 변화가 필요한데, 일반적으로 프로젝트란 일정 기간에 정해진 요구 사항을 구현하는데 목표가 잡혀 있으며, 프로젝트가 끝나면 인원은 다시 흩어져서 새로운 프로젝트에 투입 되는 형태로 역할 중심의 프로젝트팀 운용 방식에는 적절하다.
그러나 분산형 거버넌스 모델에서는 팀원의 영속성을 보장해줘야 하는데 이를 위해서는 프로젝트가 아니라 프로덕트(즉 상품)형태의 개념으로 개발 모델이 바뀌어야 한다. 팀은 상품에 대한 책임을 지고, 요구사항 정의 발굴에서 부터 개발 그리고 운영까지 책임을 지며, 계속해서 상품을 개선해 나가는 활동을 지속해야 한다. 이를 상품 중심의 개발팀 모델이라고 한다.

Self-organized team

이러한 요건등이 만족 되면, 팀은 독립적으로 서비스 개발을 할 수 있는 형태가 된다. 스스로 기획하고 개발하며 운영을 하며 스스로 서비스를 발전 시키는 하나의 회사와 같은 개념이 되는 것이다.
이렇게 독립적인 수행 능력을 가지고 있는 팀 모델을 self-organized team 모델이라고 한다.

Alignment

이러한 분산형 거버넌스 모델을 수행하기 전에 반드시 주의해야 할 점이 있는데, alignment 라는 개념이다
alignment는 각 팀간의 커뮤니케이션 방법이나 프로세스등 최소한 표준과 기술적인 수준을 맞추는 과정인데, 쉽게 이야기해서 개발 경험이 전혀 없는 대학을 갓졸업한 사람들로 팀을 만들고, 기존의 팀들은 4~5년차 경력 인원들만으로 팀을 만들어서 전체 팀을 운용하면 어떻게 될까?
마이크로 서비스 아키텍쳐는 각 서비스들이 상호 의존성을 가지고 있기 때문에, 개발경험이 없는 팀이 전체 팀의 개발 속도를 못 따라오고, 또한 품질등에도 심각한 문제가 생긴다. 그래서 어느 일정 수준 이상으로 팀의 능력을 끌어올려주고, 전체 팀에서 사용하는 최소한의 공통 프로세스등에 대해서는 서로 맞추어 놓을 필요가 있다. 이것이 바로 alignment 의 개념이다.
분산형 거버넌스 모델을 잘못 해석하거나 악용이 되면 팀에게 무조건적인 자치권을 부여하는 것으로 오역되서..
“분산형 거버넌스가 대세랍니다. 우리팀은 우리가 알아서 할테니 신경 끄세요.” 라는 형태의 잘못된 요청으로 전체 팀과 전체 시스템 아키텍쳐를 망쳐 버릴 수 있다.
제대로 된 해석은 “우리는 전체 팀이 나가야 할 방향과 비지니스 밸류에 대해서 이해를 하고 있습니다. 또한 이미 팀간의 커뮤니케이션이나 전체 시스템 구조에 대한 이해를 하고 있습니다. 이를 바탕으로 조금 더 빠른 개발과 효율성을 위한 모든 기능(역할)을 가지고 있는 팀을 운영하고자 합니다.” 가 제대로 된 해석이라고 볼 수 있겠다.

Evolutionary Model (진화형 모델)

지금 까지 간략하게나마 마이크로 서비스 아키텍쳐에 대해서 알아보았다.
마이크로 서비스 아키텍쳐는 서비스의 재사용성, 유연한 아키텍쳐 구조, 대용량 웹 서비스를 지원할 수 있는 구조등으로 많은 장점을 가지고 있지만, 운영하는 팀에 대해서 높은 성숙도를 필요로 한다. 그래서 충분한 능력을 가지지 못한 팀이 마이크로 서비스 아키텍쳐로 시스템을 개발할 경우에는 많은 시행 착오를 겪을 수 있다.
마이크로 서비스 아키텍쳐를 적용할때는 처음 부터 시스템을 마이크로 서비스 아키텍쳐 형태로 설계해서 구현할 수

도 있겠지만, 모노리틱 시스템에서 부터 시작하여, 비지니스 운용시 오는 문제점을 기반으로 점차적으로 마이크로 서비스 아키텍쳐 형태로 진화 시켜 나가는 방안도 좋은 모델이 된다. 비지니스와 고객으로 부터 오는 피드백을 점차적으로 반영 시켜나가면서 동시에 팀의 성숙도를 올려가면서 아키텍쳐 스타일을 변화 시켜가는 모델로, 많은 기업들이 이런 접근 방법을 사용했다. 트위터의 경우에도, 모노리틱 아키텍쳐에서 시작해서 팀의 구조를 점차적으로 변환 시켜 가면서 시스템의 구조 역시 마이크로 서비스 아키텍쳐 형태로 전환을 하였고, 커머스 시장에서 유명한 이베이 같은 경우에도, 그 시대의 기술적 특성을 반영하면서 비지니스의 요구 사항을 적절히 반영 시켜가면서 시스템을 변화 시켜 나가는 진화형 모델로 아키텍쳐를 전환 하였다.

SOA와 비교

마이크로 서비스 아키텍쳐는 종종 SOA와 비교 되며, SOA는 틀리고 마이크로 서비스 아키텍쳐는맞다 흑백 논리 싸움이 벌어지고는 하는데,
SOA와 마이크로 서비스 아키텍쳐는 사실상 다른 개념이 아니라 SOA가 마이크로 서비스 아키텍쳐에 대한 조상 또는 큰 수퍼셋의 개념이다. 흔히 SOA가 잘못되었다고 이야기 하는 이유는 SOA를 아키텍쳐 사상으로 보는 것이 아니라 SOAP 기반의 웹서비스나, Enterprise Service Bus와 같은 특정 제품을 SOA로 인식하기 때문이다. SOA는 말 그대로 설계에 대한 사상이지 특정 기술을 바탕으로 한 구현 아키텍쳐가 아니다.
큰 의미에서 보자면 마이크로 서비스 아키텍쳐의 서비스 역시, SOA에서 정의한 서비스 중에서 fine grained 서비스로 정의되는 하나의 종류이며, api gateway역시 SOA 에서 정의한 ESB의 하나의 구현방식에 불과 하다.
만약에 기회가 된다면 마이크로 서비스 아키텍쳐에 대해 제대로 이해하기 위해서 SOA 를 반드시 공부해보기를 바란다.

결론

마이크로 서비스 아키텍쳐는 대용량 웹시스템에 맞춰 개발된 API 기반의 아키텍쳐 스타일이다. 대규모 웹서비스를 하는 많은 기업들이 이와 유사한 아키텍쳐 설계를 가지고 있지만, 마이크로 서비스 아키텍쳐가 무조건 정답은 아니다. 하나의 설계에 대한 레퍼런스 모델이고, 각 업무나 비지니스에 대한 특성 그리고 팀에 대한 성숙도와 가지고 있는 시간과 돈과 같은 자원에 따라서 적절한 아키텍쳐 스타일이 선택되어야 하며, 또한 아키텍쳐는 처음 부터 완벽한 그림을 그리기 보다는 상황에 맞게 점진적으로 진화 시켜 나가는 모델이 바람직하다.
 특히 근래의 아키텍쳐 모델은 시스템에 대한 설계 사상 뿐만 아니라 개발 조직의 구조나 프로젝트 관리 방법론에까지 영향을 미치기 때문에 단순히 기술적인 관점에서가 아니라 조금 더 거시적인 관점에서 고려를 해볼 필요가 있다.

참고 자료

Ebay 아키텍쳐 : http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf
Netflix 아키텍쳐 : http://techblog.netflix.com/2013/01/optimizing-netflix-api.html
infoQ Microservice Architecture : http://www.infoq.com/articles/microservices-intro
MicroService 개념 http://microservices.io/patterns/microservices.html
Martin folwer : http://martinfowler.com/articles/microservices.html
Dzone microservice architecture : http://java.dzone.com/articles/microservice-architecture
Thought works의 PPT : http://www.infoq.com/presentations/Micro-Services
node.js로 apigateway 만들기 : 정리 잘되어 있음. http://plainoldobjects.com/presentations/nodejs-the-good-parts-a-skeptics-view/

#API #api gateway #Micro service #MSA #REST #REST API #Service #대용량 웹서비스 #디자인 #마이크로 서비스 아키텍쳐 #마틴파울러 #분산 시스템 #설계 #아키텍쳐 #아키텍쳐 설계 방법