|  |
| --- |
|  |
| MSA |
| Micro Service Architecture |
|  |
| 인턴 김 지 영 |
|  |
| 2018-09-14 |

|  |
| --- |
|  |

**목차**

[**1.** **MSA란?** 2](#_Toc524943095)

[**1.1.** **MSA가 나타난 배경** 2](#_Toc524943096)

[**1.2.** **MSA정의** 3](#_Toc524943097)

[**2.** **MSA 구조** 3](#_Toc524943098)

[**2.1.** **MSA 구조** 3](#_Toc524943099)

[**2.2.** **MSA 배포 구조** 4](#_Toc524943100)

[**2.3.** **데이터 저장 구조** 5](#_Toc524943101)

[**2.4.** **API Gateway** 5](#_Toc524943102)

[**3.** **MSA의 장점&단점** 8](#_Toc524943103)

[**3.1.** **MSA의 장점** 8](#_Toc524943104)

[**3.2.** **MSA의 단점** 8](#_Toc524943105)

[4. **MSA 사용법** 9](#_Toc524943106)

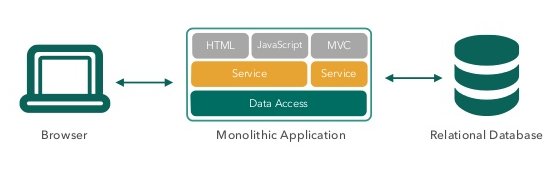
[**5.** **참고** 10](#_Toc524943107)

# **MSA란?**

## **MSA가 나타난 배경**

* **Monolithic Architecture**

하나의 애플리케이션이 한통으로 들어가는 구조를 가진다. 업무 컴포넌트들이 하나의 강한 결합 구조로 연결되어 있어, 통일성 있는 구조를 가진다.



[그림1] Monolithic Architecture 출처 : <https://readme.skplanet.com/?p=10700>

* **Monolithic Architecture의 장점**

1. 애플리케이션이 단순하게 구성되어있어, 로컬환경에서 개발하기에 편리하고 통합 시나리오 테스트를 수행하기 쉽다.
2. 모든 코드가 하나의 묶음으로 구성되어 있기 때문에 배포가 매우 간단하다.

하지만, Monolithic Architecture는 이러한 장점들이 있음에도 불구하고 서비스가 지속적으로 성장하고 규모가 커질 때 문제점들이 발생한다.

* **Monolithic Architecture의 문제점**

1. 하나의 모듈 수정을 반영하려면 전체 애플리케이션을 다시 배포해야 하기 때문에 서버 가동, 빌드, 배포 시간이 오래 걸린다.
2. 개발자의 규모가 커지고, 서비스의 복잡도가 증가되면 아주 간단한 기능을 하나 추가하기 위해서도 많은 줄의 코드를 수정해야 하기 때문에 오류가 발생할 확률이 높다.
3. 애플리케이션을 구성하는 프로그래밍 언어, 또는 프레임워크의 변경이 매우 어렵다.

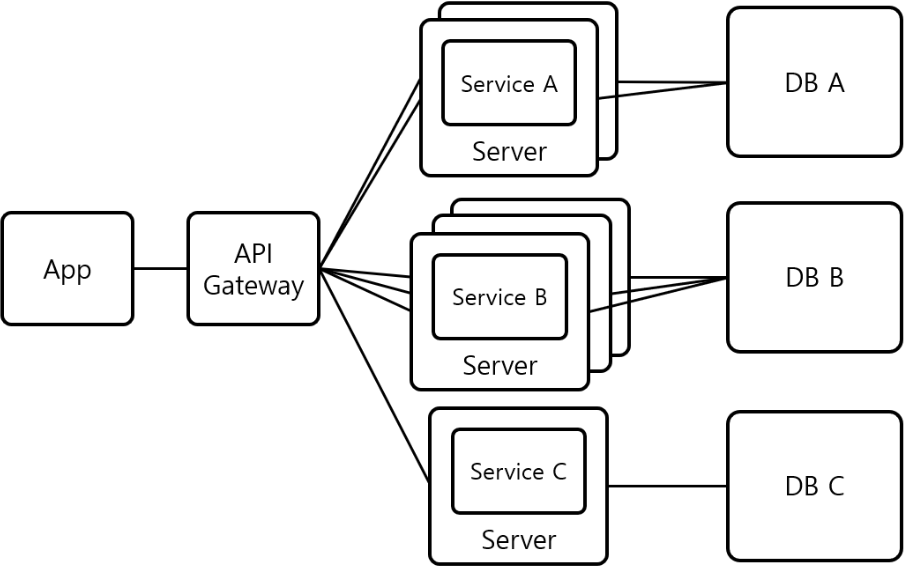
최근, 이러한 Monolithic Architecture의 문제점들을 보완하기 위해서 MSA가 등장하게 되었다.

## **MSA 정의**

대규모 소프트웨어 개발에 적용하기 위한 것으로 단독으로 실행 가능하고 독립적으로 배치될 수 있는 작은 단위(모듈)로 기능을 분해하여 서비스하는 아키텍처이다. 서비스 지향 아키텍처 라고도 한다.

* 서비스를 기능별로 작게 쪼개는 서버 아키텍처의 디자인 패턴이다.
* 하나의 서비스는 한가지 일에 초점을 맞춘다.
* API(REST API와 같은 표준 인터페이스)로 다른 서비스와 연계할 수 있다.

\*마이크로 서비스 아키텍처에는 각 컴포넌트를 서비스라는 개념으로 정의한다.



[그림2] MSA 출처 : https://readme.skplanet.com/?p=10700

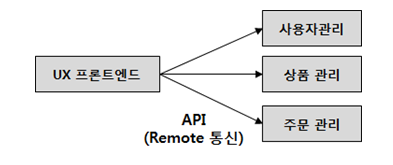
\*Micro Service : 애플리케이션을 분할 할 때 수직 방향의 기능별로 절단하는데, 절단된 독립적인 작은 모듈을 마이크로 서비스라고 한다. 각 마이크로 서비스는 공유나 프로세스 간 통신이 없어도 독립적으로 실행되며 운영 관리된다. 마이크로서비스 간 연결은 응용 프로그래밍 인터페이스(API)를 이용한다.

\*트래픽 : 인터넷 상에서 송수신 되는 모든 통신의 양. 홈페이지를 열고 이미지나 동영상을 다운받을 때 등등 모든 파일이 들어오고 나가는 것을 트래픽이라고 한다.

# **MSA 구조**

## **MSA 구조**

마이크로 서비스 아키텍처의 구조는 다음과 같다. 각 컴포넌트는 서비스라는 형태로 구현되고 API를 이용하여 타 서비스와 통신한다.



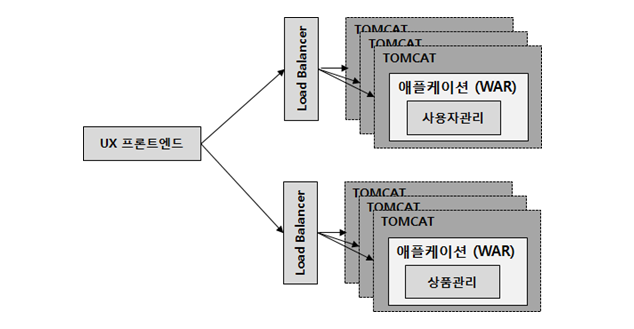
서비스

[그림 3] MSA구조 출처 : <http://bcho.tistory.com/948>

예를 들어, 사용자 관리, 상품 관리, 주문 관리와 같은 각 업무별로 서비스를 나눠서 정의한다. 여러 개의 업무를 동시에 하나의 서비스로 섞어서 정의하지 않는다. REST API에서 /users, /products와 같이 주요 URI도 하나의 서비스 정의의 범위가 될 수 있다.

## **MSA 배포 구조**

배포 구조관점에서, 각 서비스는 독립된 서버이기 때문에 다른 컴포넌트와 독립적으로 배포될 수 있다.



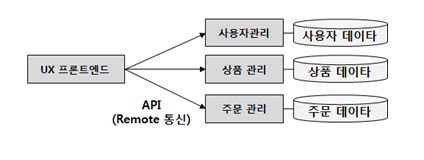
[그림4]MSA 배포 구조 관점 출처 : <http://bcho.tistory.com/948>

예를 들어, 사용자 관리 서비스는 독립적인 파일로 개발되어, 독립된 서버에 배치된다. 확장을 위해서 서비스가 배치된 서버 인스턴스는 인스턴스끼리 더하는 것이 가능하고, 앞에 로드 밸런서(서버측 시스템)를 배치하여 서비스간의 로드를 분산시킨다. 애플리케이션 로직을 분리해서 여러 개의 애플리케이션으로 나눠서 서비스화하고, 각 서비스 별로 서버를 분산 배치한 것이 핵심이다.

\*톰캣 : 서블릿 컨테이너 (또는 웹 컨테이너)만 있는 웹 애플리케이션 서버

## **데이터 저장 구조**

데이터를 저장할 때, 중앙 집중화된 하나의 데이터베이스를 사용하는 것이 아니라 서비스 별로 별도의 데이터베이스를 사용한다. 보통 Monolithic Architecture에서는 하나의 데이터베이스(보통 RDBMS를 사용)를 사용하는 경우가 일반적이었다. 하지만, MSA의 경우 서비스가 API에서부터 데이터베이스까지 분리되는 수직 분할 원칙에 따라 독립된 데이터베이스를 갖는다.



서비스

데이터베이스

[그림5]MSA에서 데이터 저장 방식 출처 : <http://bcho.tistory.com/948>

이러한 경우에, **다른 서비스 컴포넌트에 대한 의존성이 없어 서비스를 독립적으로 개발 및 배포/운영할 수 있다는 장점을** 가지고 있다. 하지만, **다른 컴포넌트의 데이터를 사용하고 싶을 때, API 통신을 통해서만 가지고 와야 하기 때문에 성능상 문제를 야기할 수 있다. 또한, 데이터베이스간의 트렌젝션을 묶을 수 없는 문제점**을 가지고 있다. 이러한 문제점들을 개선하기 위해서 **API Gateway**를 사용한다.

\*수직분할원칙 : 테이블의 일부 열을 빼내는 형태로 분할하는 것.

## **API Gateway**

* **API Gateway 사용 이유**

1. 각 서비스가 다른 서버에 분리, 배포되어있기 때문에 API의 EndPoint(서버의 URL)가 각각 다르다. –MSA의 특성상 컴포넌트의 URL 수가 매우 많아진다.
2. API를 사용하는 클라이언트에서 서버 간의 통신이나, 서버 간의 API 통신의 경우 p2p(Point to Point)형태로 토폴로지가 복잡해지고 서비스 컴포넌트간의 호출 구조는 향후 관리의 문제를 일으킬 수 있다.
3. 하나의 end point를 변경하였을 때, 제대로 관리가 되지 않을 경우가 있다.
4. API에 대한 인증이나 Logging과 같은 공통 기능에 대해서 서비스 컴포넌트 별로 중복 개발해야한다.

MSA에서는 이러한 문제점들을 해결하기 API Gateway를 배치시킨다.

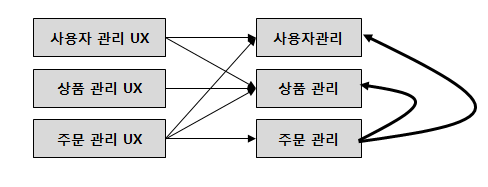
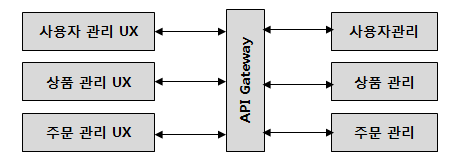
\*Gateway : 컴퓨터 네트워크에서 서로 다른 통신망, 프로토콜을 사용하는 네트워크 간의 통신을 가능하게 하는 컴퓨터나 소프트웨어이다.

\*Proxy server : 클라이언트가 자신을 통해서 다른 네트워크 서비스에 간접적으로 접속할 수 있게 해주는 컴퓨터나 응용프로그램이다. 서버와 클라이언트 사이에서 중계기로서 대리로 통신을 수행하는 기능을 가리켜 ‘프록시’, 그 중계 기능을 하는 것을 프록시 서버라고 한다.

* **API Gateway 역할**

1. **EndPoint 통합 및 토폴로지 정리**

전체 토폴로지를 p2p에서 Hub&Spoke 방식으로 변환시켜서 서비스간의 호출을 단순화 시킨다.

[그림6]P2P 형태의 토폴리지-> Hub&Spoke 토폴리지 출처 : <http://bcho.tistory.com/948>

\*토폴로지(topology) : 컴퓨터 네트워크의 요소들을 물리적으로 연결해 놓은 것, 또는 그 연결 방식.

1. **공통 기능 처리**

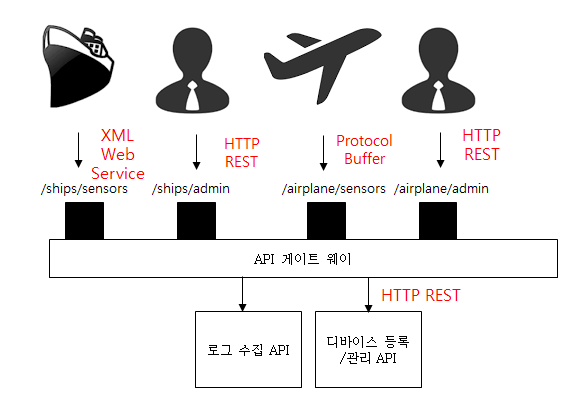
API에 대한 인증이나 Logging과 같은 공통 기능에 대해서 서비스 컴포넌트 별로 중복 개발해야하는 문제점을 해결한다. API Gateway 에서 공통 기능들을 처리하게 되면 중복을 방지 할 수 있어 효율적으로 개발할 수 있다.

1. **Message transformation**

XML이나 네이티브 메시지 포맷을 json등으로 상호 변환해 준다.

1. **프로토콜을 변환하는 기능**

다양한 서비스나 클라이언트를 지원하게 되면, 클라이언트나 서비스 별로 다른 통신 프로토콜을 사용해야 하는 경우가 있다. 보통 웹에서는 JSON 기반의 REST가 많이 사용되지만, 배나 비행기에 사용되는 센서들의 경우에는 가벼운 프로토콜을 사용해야 한다. 이렇게 다양한 타입의 프로토콜을 지원하기 위해서, 각 서비스들을 새롭게 구현하는게 아니라 API Gateway에서 프로토콜 변환을 통해서 같은 API를 다른 프로토콜로 서비스 할 수 있도록 할 수 있다.

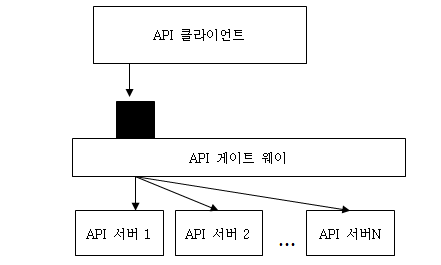


[그림7]API Gateway를 통한 프로토콜 변환 출처 : <http://bcho.tistory.com/1005>

\*프로토콜 : 복수의 컴퓨터 사이나 중앙 컴퓨터와 단말기 사이에서 데이터 통신을 원활하게 하기 위해 필요한 통신 규약.

1. **Backend API 서버로의 Load Balancing**

API Gateway 뒤에 다수의 API 서버가 있다고 할 때, 여러 개의 API 서버로 부하를 분산하는 기능이 필요하다.



[그림8]API Gateway를 통한 API 서버로의 로드 밸런싱

\*Load Balancer : 여러 대의 Server에게 균등하게 트래픽을 분산시켜주는 역할을 하는 것.

\*Load Balancing : 하나의 인터넷 서비스가 발생하는 트래픽이 많을 때 여러 대의 서버가 분산 처리하여 서버의 로드율 증가, 부하량, 속도저하 등을 고려하여 적절히 분산 처리하여 해결해주는 서비스

# **MSA의 장점&단점**

## **MSA의 장점**

1. 서비스들을 독립적으로 배포할 수 있다.
   * 하나의 서비스에 변화가 생기면 이 서비스에게 필요한 최소한의 부분들(이 기능과 직접적으로 관련이 있는 부분들)에게만 변경사항을 업데이트 할 수 있다. 이렇게 하면 배포 과정이 더 빨라지고, 개발과 실제 서비스 중인 기능의 차이를 줄일 수 있다.
2. 기능들을 독립된 서비스들로 격리시키면, 새로운 기능들을 추가할 때 기존 기능들이 불안정해지는 등의 문제를 피할 수 있다.
   * 하나의 컨테이너에서 에러가 발생해도 다른 중요한 컨테이너에 영향을 주지 않는다. 따라서 문제가 일어나도 유연하게 대처할 수 있다.
3. 서비스들을 분리시킴으로써 보안이 강화될 수 있다.
   * 아마존 EC2 컨테이너 서비스는 각각의 서비스 컨테이너를 서로 다른 AWS(Amazon Web Service)로 실행할 수 있게 해준다. (MSA) 이렇게 하면 각각의 서비스가 자원(resource)에 대해 자신만의 일시적인 접근 자격을 가지게 된다. 따라서 원격 코드 실행 등의 공격으로부터 유저들의 개인정보를 지킬 수 있다.

\*Amazon EC2 : Amazon EC2를 통해 원하는 만큼 가상 서버를 구축하고 보안 및 네트워크 구성과 스토리지 관리가 가능하다. 하드웨어에 선투자할 필요가 없어 더 빠르게 애플리케이션을 개발하고 배포할 수 있다.

1. 서비스들은 독립적인 자원 한도를 가진다.
   * 하나의 서비스가 갑자기 더 많은 양의 자원을 필요로 하게 된다고 하더라도 그 기능에 한해서만 latency(지연 속도)가 증가할 뿐, 다른 기능들은 영향을 받지 않는다.

## **MSA의 단점**

1. 배포할 부분들이 늘어나게 되어 배포 시 복잡성이 증가한다.
   * 마이크로서비스는 정상적인 배포를 위해서 언제나 자동화와 오케스트레이션(orchestration) 프레임 워크를 필요로 한다.

\*Orchestration : 여러 개의 서비스를 묶어서 하나의 새로운 서비스를 만드는 개념이다.

1. 서비스를 분리하므로 분산 컴퓨팅의 복잡성이 발생한다.
2. 하나의 서비스가 다른 서비스와 커뮤니케이션 할 때 latency(지연)가 발생한다.
3. 문제 발생의 근원지를 찾는 것이 Monolithic Architecture에 비해 어렵다.
   * 마이크로서비스 애플리케이션을 디버깅 하는 것은 어느 서비스가 문제를 발생시켰는지, 그 서비스의 어느 부분이 잘못되었는지를 찾아내야 하는 과정이므로 한 개의 시스템만 보면 됐던 모놀리스와는 차이가 있다.
   * 좋은 로깅(logging)/모니터링 솔루션을 필수적으로 갖춰야 한다.

\*logging : 시스템을 작동할 때 시스템의 작동 상태, 이용자의 습성 조사 및 시스템 동작의 분석 등을 하기 위한 작동 중의 각종 정보의 기록을 만드는 것.

\*분산 컴퓨팅 : 인터넷에 연결된 여러 컴퓨터들의 처리 능력을 이용하여 거대한 계산문제를 해결하려는 분산처리 모델.

# **MSA 사용법**

모든 애플리케이션이 MSA로 구성될 필요는 없다. MSA도 위에서 본 것과 같이 장점과 단점들이 있기 때문에 요즘 트렌드가 MSA라는 것만 보고 무작정 MSA를 도입하는 것이 아니다. MSA의 장점과 단점들을 잘 파악한 뒤, 자신이 개발할 애플리케이션이 어떤 구조에 더 적합할지 잘 판단하여 사용하는 것이 좋다.

1. 애플리케이션 배포 시간이 1시간 이상 걸린다.
2. 단순한 기능 하나를 수정해도 전체 기능에 대한 QA가 필요하다.
3. 단순한 버그 수정이 더 중대한 버그를 생산하는 일이 많아졌다.
4. 현재의 애플리케이션을 기능별로 나눈다고 가정했을 때 수십 개의 마이크로 서비스가 가능하다.

일반적으로 위의 항목들이 현재 상황에 해당한다고 생각되면 MSA를 사용하는 것이 좋다.

# **참고**

<https://medium.com/@jwyeom63/%EC%84%B8%EB%B6%84%ED%99%94-granular-%EC%95%A0%ED%94%8C%EB%A6%AC%EC%BC%80%EC%9D%B4%EC%85%98-%EC%95%84%ED%82%A4%ED%85%8D%EC%B2%98-%ED%8C%A8%ED%84%B4-1f6a2d720545>

[https://kihoonkim.github.io/2017/01/27/Microservices%20Architecture/[SDS%EC%82%AC%EB%82%B4%20%EA%B8%B0%EA%B3%A0]%20Microservice%20Architecture%EB%9E%80/](https://kihoonkim.github.io/2017/01/27/Microservices%20Architecture/%5bSDS%EC%82%AC%EB%82%B4%20%EA%B8%B0%EA%B3%A0%5d%20Microservice%20Architecture%EB%9E%80/)

<https://www.slideshare.net/prevdev/ss-65390862>

<http://bcho.tistory.com/948>

<https://terms.naver.com/entry.nhn?docId=3548871&cid=42346&categoryId=42346>

<https://terms.naver.com/entry.nhn?docId=796754&cid=42347&categoryId=42347>

<http://guruble.com/%EB%A7%88%EC%9D%B4%ED%81%AC%EB%A1%9C%EC%84%9C%EB%B9%84%EC%8A%A4microservice-%EC%95%84%ED%82%A4%ED%85%8D%EC%B2%98-%EA%B7%B8%EA%B2%83%EC%9D%B4-%EB%AD%A3%EC%9D%B4-%EC%A4%91%ED%97%8C%EB%94%94/>

<http://jodu.tistory.com/17>