**MSA(Microservice Architecture)**:

전체 시스템을 독립적인 서비스의 집합으로 구성하는 방법이다.

각각의 소프트웨어는 그들만의 한정된 책임 범위를 갖는다.

어플리케이션 출시처럼 하나의 목표를 향해 일하지만 자기가 개발하는 서비스만 책임진다.

여러 어플리케이션에서 재사용할 수 있어야 한다.

어플리케이션은 항상 기술 중립적 프로토콜을 사용해 통신하므로 서비스 구현 기술과는 무관하다.

마이크로서비스 기반의 어플리케이션을 다양한 언어와 기술로 구축할 수 있다는 것을 의미한다.

REST 등 가벼운 통신 아키텍처, 또는 Kafka 등을 이용한 message stream을 주로 사용한다.

**MSA 장점:**

1. 조직적인 확장성이 높다.

2. Scale out이 쉽다.

3. 각 서비스에 다른 기술 스택을 쓸 수 있다.

4. 시스템 안정성이 높아진다.

(하나의 장애가 전체 시스템으로 번지지 않는다.)

**MSA의 어려움:**

1. RPC가 기반이 되어야 한다.

2. 통합 테스트가 까다롭다.

3. 성능 문제 및 버그 해결이 까다롭다.

4. 과도한 복잡성이 생길 수 있다.

**모놀리식으로 구성된 서비스를 MSA로의 전환 시,**

**마이크로 서비스의 경계를 정하는 원칙:**

**1. 각각의 마이크로 서비스가 응집성이 있어야 한다.**

응집성은 함께 변경되는 코드가 함께 유지된다는 것을 의미합니다.

서로 강하게 관련되어 있고, 함께 변경되는 요소들은 같은 서비스에 존재해야 한다.

**2. 서비스도 SRP를 따라야 한다.**

모든 마이크로 서비스는 한 가지 일을 해야 한다.

**3. 각 서비스는 느슨한 결합되어야 한다.**

서로 상호 연관성이 매우 적거나 전혀 없어야 한다.

각각의 서비스가 다른 서비스와 최소한의 통신으로 고유 기능을 수행해야 한다.

참고:

각 서비스의 크기는 그리 중요하지 않다.

가능한 작아야 한다는 것은 흔한 오해다.

**MSA로 분해하는 2가지 방법:**

**1. 비즈니스 기능에 따라 분리**

UI나 조직 구조에 맞춰 서비스를 나눈다.

고객이 보는 기능 단위이다.

**2. DDD ( Domain Driven Design )**

DDD는 도메인을 명확히 분리하고 일관된 언어를 사용하는 것에 중점을 두며,

이러한 과정을 통해 개발자들 간의 소통이 원활해진다.

예를 들어, DDD는 비즈니스 요구사항을 반영하여 서비스의 경계를 명확히 하여,

각 팀이 해당 도메인에 대해 독립적으로 작업할 수 있도록 지원한다.

**도메인이란?**

특정 비즈니스의 전체적인 개념(큰 범위)을 의미한다.

**하위 도메인이란?**

하위 도메인은 도메인을 구성하는 개별 기능을 의미한다.

모든 하위 도메인이 똑같이 중요한 것은 아니기에,

하위 도메인으로 3종류로 분류한다.

1. Core: 비즈니스 핵심 경쟁력

2. Supporting: 핵심을 보조하는 기능

3. Generic: 범용적인 기능

ex\_)

도메인: 이커머스 (온라인 쇼핑몰)

하위 도메인

* Core: 주문 처리, 상품 추천
* Supporting: 고객 지원, 배송 관리
* Generic: 결제 시스템, 인증

**DDD의 몇 가지 핵심 요소:**

**유비쿼터스 언어:**

도메인 전문가와 개발자가 공통으로 사용하는 언어를 정의하고,

이를 통해 의사소통의 명확성을 높입니다.

**Bounded Context:**

도메인 모델이 적용되는 특정 경계를 설정하여, 서로 다른 도메인 간의 혼동을 피한다

각 Bounded Context는 독립적으로 모델을 발전시키고 유지관리할 수 있다.

**DDD 진행 순서:**

1. SW 개발팀 공통 언어 정의 = 유비쿼터스 언어

2. 비즈니스 분석 = 핵심 개념 식별

3. 식별된 개념 분석 = Bounded Context 식별

3. 컨텍스트 관계정의 = Context Map 작성

4. 서비스 분할 및 통합 = 마이크로 서비스 후보 선정

**MSA로 마이그레이션 하는 방법:**

**1. Big Bang Approach (비추)**

업데이트 싹 다 중단하고,

모든 개발자들이 마이그레이션에 집중하여 끝내는 방법.

예상된 기간내로 끝나지 않았을 때 리스크가 있다.

단일 애플리케이션을 너무 많은 개발자가 함께 작업하면, 산으로 갈 수 있다.

**2. Incremental and continuous Approach**

점진적이고 지속적으로 이관하는 방법이다.

MSA로 이관했을 때 많은 이점을 얻을 수 있는 순으로 이관한다.

참고:

MSA로 이관 중에는 최대한 기술 스택을 유지하는게 안전하다.

**마이그레이션 스탭:**

1. 확실한 테스트 커버리지가 있는지 확인&추가

2. 명확하고 심사 숙고한 API 정의

3. 기존 코드에서 상호 의존성 제거

**스트랭글러 피그(strangler fig) 패턴:**

기존 모놀리식 애플리케이션을 점진적으로 MSA로 전환하는 방법이다.

기존 모놀리식 애플리케이션 앞에 Proxy를 둔다.

이 Proxy는 기존 시스템과 새로운 MSA 사이에서 트래픽을 라우팅하는 역할을 한다.

**이런 Proxy를 Strangler Facade로 명한다.**

약간 카나리 배포처럼 동작하는데,

MSA로 분리한 특정 기능이 있을 때,

해당 MSA를 기동하고, 모놀리식에서도 코드를 제거하지 않는다.

**‘Shadow Traffic’** (실제 서비스에는 영향 없이 트래픽만 복사해서 테스트)

방식으로 먼저 검증하고 문제가 없다고 판단되면,

Proxy가 특정 기능을 사용하는,

일부 트래픽을 MSA에 라우팅 하고,

해당 MSA가 문제가 있는지 개발자가 모니터링한다.

(MSA로 점차 트래픽을 옮긴다.)

문제가 없다면,

MSA로 이관된 특정 기능을

모놀리식 코드에서 완전히 제거하고

Proxy는 특정 기능을 사용한다면,

무조건 MSA에 라우팅 되도록 한다.

**MSA 핵심 원칙:마이크로 서비스마다 DB가 있어야 한다.**

DB를 공유하면 스키마 변경 시 서로 다른 서비스에 영향이 가고,

DB에서 병목이 발생할 수 있다.

다른 서비스가 필요한 데이터가 있는 경우,

추가적인 지연시간 있더라도,

직접 DB에 연결하지 못하게 해야 한다.

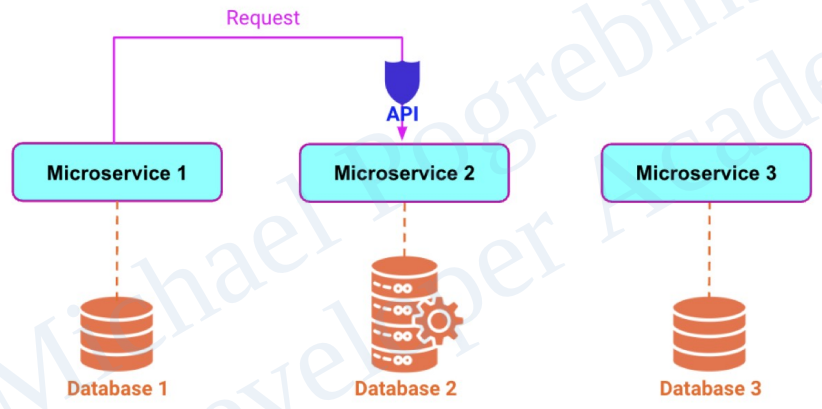
이런 요청은 대상 서비스의 API를 통해 이루어져야 한다.

**단점:**

1. 지연시간

2. DB수준에서 조인 같은거 불가능

3. 분산 트랜잭션을 사용해야 함 (복잡성 up)

****

**MSA 에서의 DRY 원칙:**

일반적으로 동일 로직 및 데이터를 반복 사용한다면,

이를 공유할 수 있게 하는게 좋다. ex\_) 공유 라이브러리

하지만,

**MSA 에서는 항상 DRY 원칙이 적용되지 않을 수 있다.**

공유 라이브러리를 서로 다른 서비스에서 사용하면,

이는 서비스 간의 느슨한 결합을 깨트릴 수 있다.

**MSA에서 서비스 간 중복코드 해결 방안:**

**비즈니스 로직이 중복되는 경우:**

1.

도메인의 경계를 다시 나누어, 코드를 한 곳에만 추가한다.

도메인 경계가 잘 나뉘어 졌으면 이런 일이 발생하지 않았을 것이다.

2.

해당 코드만 담당하는 별도 서비스를 만든다.

중복 코드의 로직이 너무 복잡할 경우 해당한다.

3.

사이드카 패턴을 사용한다.

공용 로직에 해당하는 부분을 별도 프로세스로 띄우고

서비스와 같은 호스트에서 실행되게 한다.

4. (마지막 수단)

재시도 전략 등 매우 일반적이고

앞으로 변경이 되지 않을 것 같은 코드는

공유 라이브러리를 써도 괜찮다.

5.

통신을 위한 데이터는 중복될 수 있고 바람직하다.

이 경우, 공유 스키마를 통해 각자의 코드에 맞게 생성하는 것이 좋다.

테스트에서 빨리 발견되게 조치를 하는게 좋다.

**DB 데이터 중복인 경우:**

Owner를 정하고, 캐싱해서 쓰는 것도 좋다.

CQRS를 사용하는 것도 상황에 따라 좋다. (일관성을 고려하자)

성능 때문에 어쩔 수 없이 중복 저장하는 경우도 있다.

이 경우 일관성을 잘 고려해야 한다.

**구조화된 자율성 (Structure Autonomy):**

각 서비스가 완전한 자율성을 가지고 만들어지면 안된다.

인프라를 설정이 어려워진다.

전담 DevOps, QA가 너무 힘들 수 있고,

시스템 전반을 이해하기 어려워진다.

개발자 간 협업이 어려워질 수 있다.

**자율성을 가질 수 없는 범위**

1. 인프라

- CI/CD

- 모니터링, 알람 시스템

2. public/private API 사용 방법

3. 보안 및 데이터 규정

**아래의 일정한 자율성이 부여되는 범위 (승인 필요):**

1. 프로그래밍 언어

2. DB 종류

**완전한 자율성이 부여되는 범위:**

1. 출시 프로세스

2. 업데이트 일정 및 빈도

3. 툴

4. 문서

**마이크로 프론트엔드 아키텍처 패턴:**

모놀리식 웹 애플리케이션을

독립적인 단일 페이지 애플리케이션으로 작동하는

여러 프런트엔드 모듈이나 라이브러리로 분할한다.

**Build-Time Integration 방식:**

여러 프런트엔드 모듈을 빌드 시점에 하나의 번들로 합치는 방식.

각 모듈은 독립적인 코드베이스를 가지지만, 최종 배포물은 하나.

**Run-Time Integration방식:**

브라우저에서 컨테이너 역할을 하는 메인 앱(Host, Shell)이 있고,

해당 앱에서 여러 개의 마이크로 프론트엔드 애플리케이션을 실행한 다음,

이걸 조합해서 최종 UI를 구성하는 방식이다.

각각의 마이크로 프론트엔드는 독립적으로 실행되지만,

컨테이너가 이를 불러와서 하나의 UI로 조합한다.

ex\_)

+------------------------------------------------+

| [Header - Shell (공통 레이아웃)] |

+------------------------------------------------+

[ 마이크로 프론트엔드 1 - 추천 콘텐츠 ]

+------------------------------------------------+

[ 마이크로 프론트엔드 2 - 최근 본 영상 ]

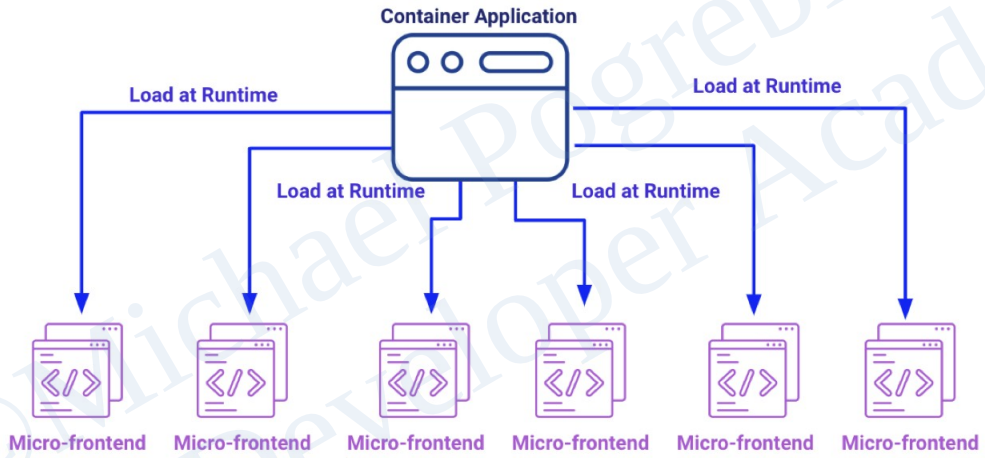
+------------------------------------------------+

[ 마이크로 프론트엔드 3 - 인기 콘텐츠 ]

+------------------------------------------------+

| [Footer - Shell (공통 레이아웃)] |

+------------------------------------------------+



**MSA API 관리:**

클라 입장에서 각 서비스마다 따로 요청/응답을 받는 것은 번거로운 일이다.

개별 서비스에 인증을 따로 해야 하고, API 변경을 다 맞춰줘야 한다.

클라가 어떤 서비스가 있는지 다 이해하고 이를 요청해야 한다.

클라가 API 관리를 쉽게 하기 위해 API 게이트웨이 패턴을 쓸 수 있다.

**API 게이트웨이 패턴:**

시스템 진입점에 API 게이트웨이라는 구성 요소를 배치한다.

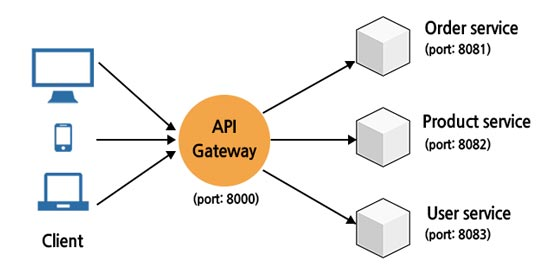
클라는 API 게이트웨이를 통해 API를 요청한다.

API 게이트웨이는 특정한 API를 이를 처리하는 서비스로 라우팅한다.

라우팅 말고 트래픽 관리, 권한 제한 기능을 수행할 수도 있다.

클라이언트와 서버 쪽 서비스의 구현이 분리되어,

서비스 구성을 조금 더 자유롭게 변경할 수 있다.



**분산 트랜잭션이 필요한 경우 처리 방법:**

1. 연결 체인 (직렬 처리)

2. Orchestration SAGA (병렬 처리)

3. Event Driven (코레오그래피 SAGA 말하는 듯)

**이벤트 중심 아키텍처 (Event Driven Archietectrue):**

MQ에 Event를 넣고, 언제든지 메시지 브로커에 이를 찾을 수 있다.

이벤트가 동작, 상태를 변경하는 것을 중심으로 한다.

MQ에서 개발자가 원하는 만큼 많은 수신자에게 이벤트를 라우팅할 수 있게 해주고,

시스템에 중복성을 추가한다.

**요청-응답 모델 vs 이벤트 중심 모델:**

**요청-응답 모델:**

보통 동기적으로 구현된다.

한 번 소비되고, 한 대의 서버에서만 소비된다.

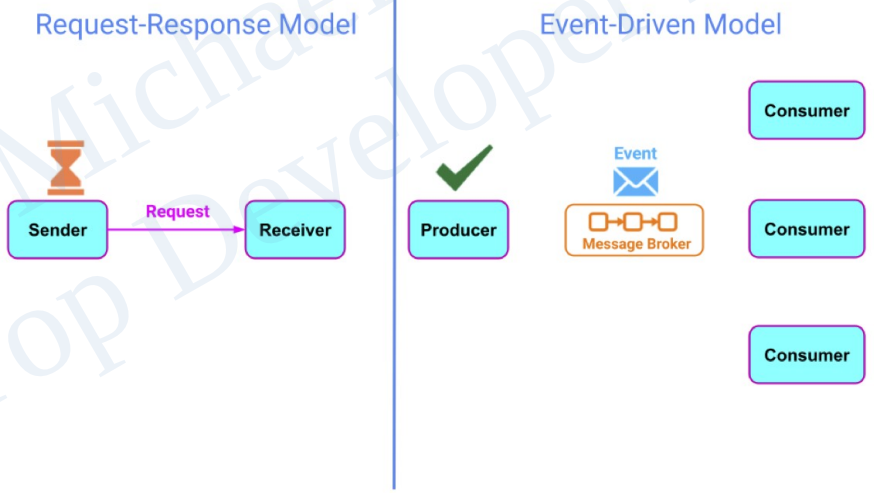
Sender가 각 Serivce를 알아야 한다.

**이벤트 중심 모델(MQ):**

비동기적으로 구현된다.

다른 서비스에서 여러 번 소비될 수 있다.

Sender 각 서비스를 몰라도 된다.



**이벤트 중심 아키텍쳐를 쓰기 좋은 경우:**

1. fire and forget이 가능한 경우

2. 메시지 손실이 발생하면 안되는 경우

3. 계속해서 데이터 및 이벤트를 스트림해야 하는 경우

4. 이상 감지 또는 패턴을 인식해야 하는 경우

(MQ에 초당 요청 사항이 증가할 때, ScaleOut을 고려할 수 있다.)

5. 브로드캐스팅 해야하는 경우

6. 트래픽 급증을 견딜 수 있게 버퍼링을 해야하는 경우

**요청-응답 모델를 쓰기 좋은 경우:**

1. 사용자에게 즉각적으로 응답해야 할 경우

2. 이벤트 중심 모델로 얻을 이점이 없는 경우

**자주 사용하는 이벤트 중심 아키텍쳐 패턴 2가지:**

**1. Event Streaming**

MQ가 이벤트의 임시 저장소 또는 영구 저장소로 사용된다.

동일한 소비자 또는 다른 소비자에 의해 이미 소비됐더라도,

소비자는 이벤트의 로그에 완전히 접근할 수 있다.

신뢰성 있는 전달을 위해 좋다.

이상 감지에도 좋은 역할이 될 수 있다.

**2. Pub/Sub**

특정 주제의 채널에 구독하면,

구독 이후에 새로운 이벤트들만 받게 된다.

구독자가 이벤트를 받으면 MQ에서 이벤트를 삭제한다.

Fire and Forget 특성을 가질 때,

이벤트를 무한히 받아야할 때 좋다.

**respose를 못받았을 때 대처:**

**맹점:**

request를 처리했지만, response만 못했을 경우,

requset를 처리 못했을 경우 2가지가 존재한다.

이때 Sender는 receiver가 처리 했는지 모르는데,

메시지를 재전송 해야 하는가?

**메시지 전달 의미론:**

**1. 최대 한 번만 전달(At-Most-Once)**

발신자가 일정 기간 내에 확인 응답을 받지 못해도, 재전송 하지 않는다.

일부 데이터의 손실은 괜찮지만, 이벤트 중복은 피하고 싶은 경우에 사용.

가장 낮은 오버헤드와 가장 낮은 레이턴시를 가진다.

**2. 최소 한 번만 전달(At-Least-Once)**

발신자가 일정 기간 내에 확인 응답을 받지 못하면, 이벤트를 재전송 한다.

데이터 손실이 있으면 안되고, 이벤트가 중복 처리되어도 문제 없을 때 사용.

응답을 기다리는 로직이 필요하고 레이턴시가 발생하기 때문에,

실시간, 고 처리량 시스템에는 좋지 않다.

**3. 딱 한 번만 전달 (Exactly-Once)**

At-Least-Once와 방식은 같은데,

EventId를 같이 넘기고, 이를 확인하여 딱 한번만 실행되게 한다.

이미 제공한 적 있는 id면 그냥 로그 남기고 무시하는 거다.

실제로는 1번 이상 전송 될 수 있지만, Id 비교로 딱 한번만 동작하게 한다.

오버해드와 대기시간이 제일 길다.

**서킷 브레이커:**

서킷 브레이커는 장애 전파를 막기 위한 보호 장치로, 실패가 일정 기준을 넘으면 호출을 차단한다.

정상 상태로 복구할 기회를 주기 위해 일정 시간 후 다시 시도하는 구조이다.

**주요 개념:**

* **연계 고장(Cascading Failure)**:  
  하나의 컴포넌트 고장이 연쇄적으로 다른 서비스에 영향을 줌
* **fail-fast**:  
  문제가 생기면 빠르게 실패하고 더 이상 진행하지 않음
* **백 프레셔(Back Pressure)**:  
  처리 불가능한 요청은 upstream에 알리고, 부하를 조절함

**동작 구조 (상태 변화)**

**Closed (닫힘):**

정상 동작. 요청을 그대로 보내고 실패율을 감시.

**Open (열림):**

실패율이 임계치를 넘으면 차단. 일정 시간 동안 외부 호출 중단 후 바로 실패 반환.

**Half-Open (반열림):**

대기 시간 후 일부 요청만 테스트로 보내 성공하면 Closed로 복귀, 실패하면 다시 Open.

**서킷 브레이커 대표 라이브러리:**

Hystrix by 넷플릭스

모든 마이크로 서비스 마다 이를 구현해야 하고,

사용할 수 없는 프로그래밍 언어도 있어서,

요즘 안쓰는 추세라고 한다.

**업스트림과 다운스트림:**

[Client] → [Service A] → [Service B] → [DB]

서비스 호출 기준:

* A 입장에서 B는 Downstream (A가 B에 요청을 보내니까)
* A 입장에서 Client는 Upstream (Client가 A에 요청을 보내니까)

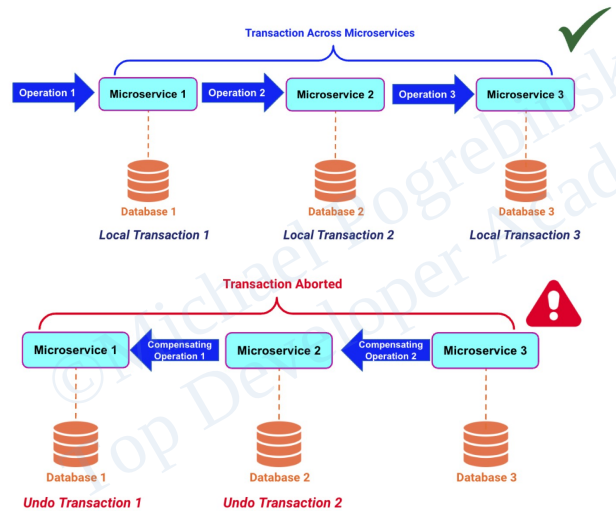
데이터 흐름 기준 (예: Stream 처리, Back Pressure):

* B → A로 데이터가 올라오면, B는 Upstream, A는 Downstream
* 이 기준에서 Back Pressure는 Downstream(A) 이 Upstream(B) 에게 “속도 줄여”라고 말함

**MSA 디자인 패턴**

**SAGA 패턴:**

자세한 내용은 [21) 분산 트렌젝션](https://docs.google.com/document/u/0/d/10zrhuTgr7VpfnivzgFTq9_Kd37LZWbAyYgBxfcX_jL0/edit) 참고



**CQRS 패턴 (Command and Query Responsibility Segregation):**

Write 하는 DB와 Read 하는 DB를 분리한다.

Write 하는 DB가 오르지날 이고, Read 하는 DB는 복제본이다.

WriteDB와 ReadDB 사이에 MQ를 둬서 꾸준히 복사되게 한다.

WriteDB와 ReadDB가 꼭 같은 유형의 DB일 필요는 없다.

**장점:**

1. SRP에 좋다.

2. 성능상 유리하다.

WriteDB와 ReadDB는 각 목적에 유리한 스키마를 가져갈 수 있다.

3. 확장성에 좋다.

4. 통합 뷰로써 사용될 수 있다.

별도의 쿼리 서비스로,

여러 서비스의 DB의 Read를 하나의 DB로 합쳐서, 빠른 쿼리만을 위한 서비스를 제공할 수 있다.

MSA에서는 JOIN에 어려움을 느낄 수 있기 때문에 이는 도움이 된다.

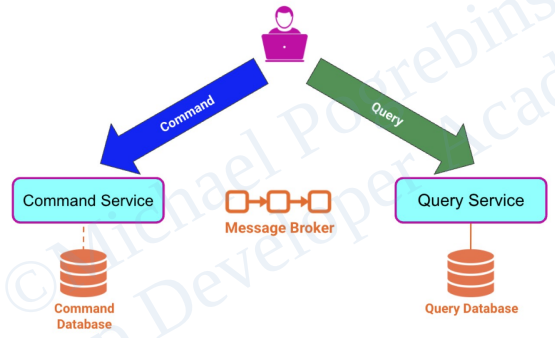
**주의점:**

읽기와 쓰기 사이에 최종적인 일관성만을 가지기에,

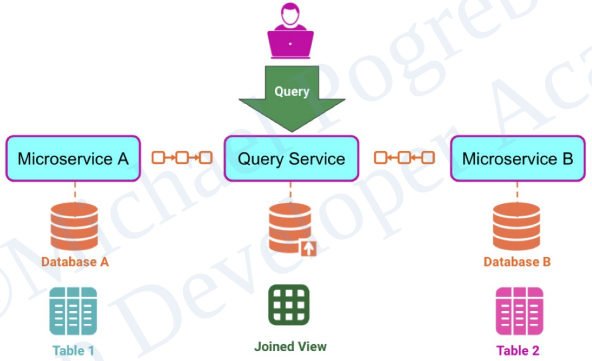
ReadDB에 복사되기 전에, 데이터를 읽으면,

Write 명령에 성공 했지만 이전 데이터를 읽어올 수 있다.

ex\_1)



ex\_2)



**Event Sourcing 패턴:**

현재 상태 뿐만 아니라, 이전 상태가 모두 필요할 때 쓴다.

DB에 현재 상태를 저장하는 것이 아니라, 이벤트를 저장한다.

한 번 저장한 이벤트는 다시 변경하지 않는다.

만약 현재 상태를 얻고 싶다면,

엔티티에서 발생한 모든 이벤트를 재적용하면 된다.

이를 **리플레이** 라고 한다.

ex\_) 통장 거래 내역

**이벤트를 저장하는 방법:**

1. 각 이벤트를 DB에 개별 레코드로 저장하는 방법

2. 영속성을 가진 MQ에 저장하는 방법

**장점:**

1. 히스토리를 파악할 수 있다.

2. 검증 및 분석에 좋다.

3. 새로운 레코드만 추가 하기 때문에,

동일 엔티티에 대한 경합이 없다.

이에 따라 성능 향상을 기대할 수 있다.

**단점:**

1. 이벤트가 많아지면, 리플레이 하는데 오래 걸린다. O(n)

2. 이벤트를 처리하는 로직이 바뀌었다면,

리플레이 했을 때 현재 결과가 달라질 수 있다.

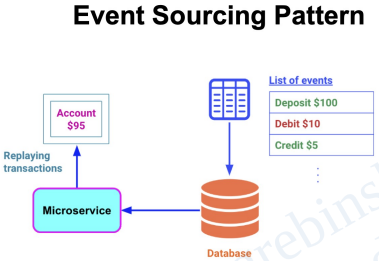
**단점을 해결하는 방법:**

1. 특정 지점에 **스냅샷**을 찍는다.

가장 마지막 스냅샷을 기준으로 리플레이 한다.

2. **CQRS를 이용**하여, WriteDB는 이벤트 소싱으로하고,

ReadDB는 최신 상태만 유지하는 등으로 유연하게 처리할 수 있다.



**테스팅 피라미드:**

빠르고 비용 적은 테스트를 많이, 느리고 비용 큰 테스트는 적게 하자.

**종단간(EndToEnd) 테스트:**

UI, 전체 애플리케이션 포함한 전체 시스템을 실행하고 테스트.

엔드 유저 입장에서, 각각의 테스트는 사양과 일치하는지 확인.

신뢰성이 가장 높음.

**통합 테스트:**

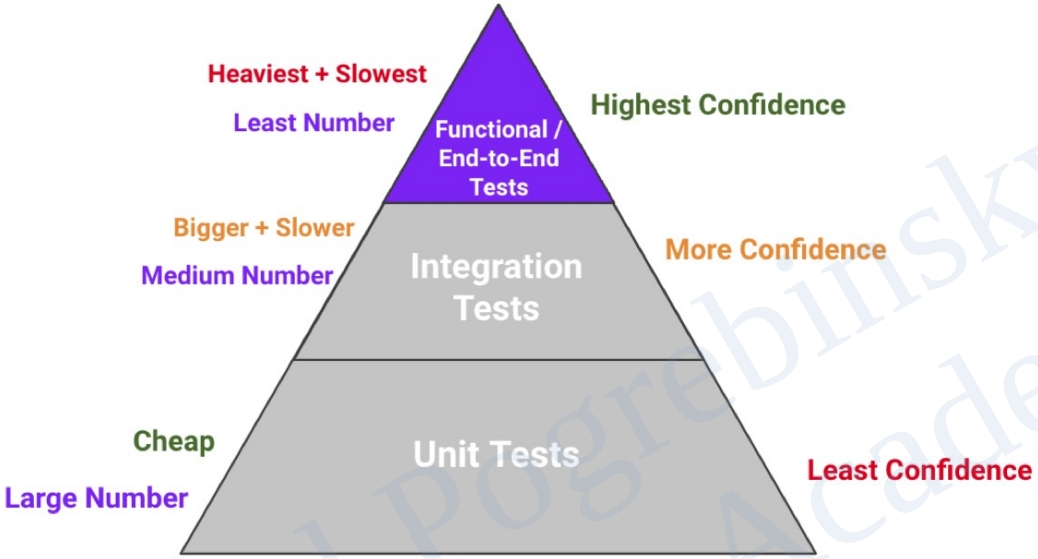
DB, 메시지 브로커와 같은 여러 단위와 시스템이 실제로 작동하는지 테스트.

유닛 테스트 보다는 신뢰도가 높음

**유닛 테스트:**

클래스나 모듈과 같은 작은 논리 단위를 독립적으로 테스트.

각 유닛을 독립적으로 테스트하기 때문에, 전체 시스템에 대한 신뢰도가 낮음



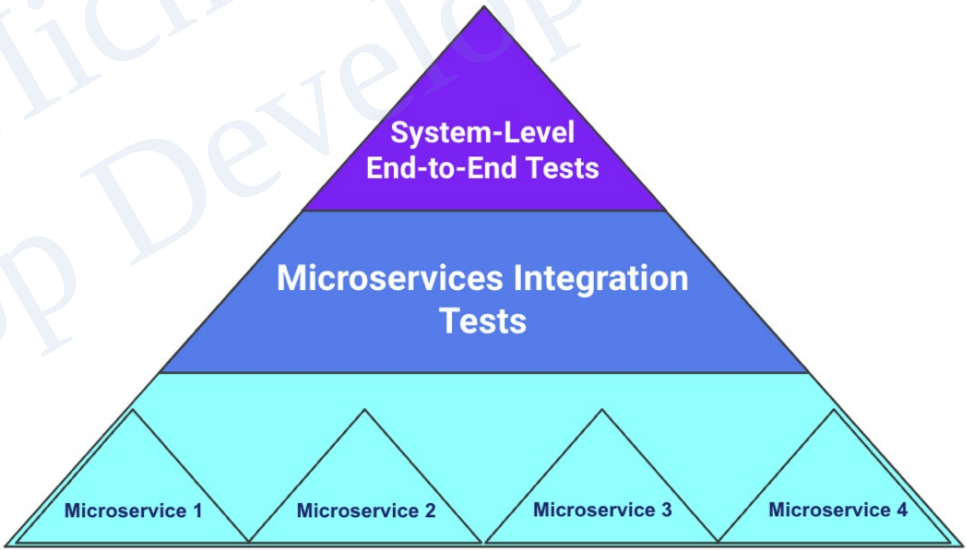
(테스트 양은 유닛이 제일 많고, 종단간이 제일 적어야 한다.)

**MSA에서 테스팅 피라미드 사용 방법:**

1. 각 서비스에 대해서 유닛 테스트, 통합 테스트, 종단간 테스트를 진행한다.

2. 각 서비스를 큰 시스템의 일부인 작은 단위로 취급하고,

더 큰 피라미드 수준에서 통합 테스트, 종단간 테스트를 진행한다.



**MAS에서의 테스팅 피라미드의 한계:**

1. 종단간 테스트는 특히 달성하기 어렵다.

2. 공수가 많이 든다.

3. 통합 테스트 진행을 하기 위해 팀간 결합점이 발생한다.

통합 테스트 진행을 위해 다른 서비스 담당자와 계속 합을 맞춰야 한다.

**MSA의 통합 테스트 해결책**

**1. 통합 테스트 진행이 어렵다면, Mock을 사용한다.**

Mock은 모의 객체를 의미한다.

다른 서비스와 연결하는 데신 Mock을 사용하고,

해당 Mock이 기대하는 응답을 내도록 만든다.

**문제점:**

API의 제공자와 사용자 간의 동기화 깨짐 문제.

ex\_) 제공자는 API를 갱신해서, 이제 다른 res를 던지는데,

사용자는 이를 몰라 이전 버전에 맞게 Mock을 사용할 수 있다.

**2. 통합 테스트 진행이 어렵다면, 계약 테스트(Contract Testing)를 진행한다.**

전용 도구를 이용하여, API 제공자와 소비자를 공유 계약을 통해 동기화하는 방식이다.

**계약 테스트 진행 방법:  
ㄴ 소비자 중심 계약 테스트(Consumer-Driven Contract, CDC) 방식**

1. 소비자가 API 계약을 정의 (예: "GET /users/{id} 요청 시 특정 JSON 응답을 반환해야 함").
2. 계약을 제공자에게 전달하여 검증.
3. 제공자는 계약을 기반으로 자체 테스트 수행 → 변경이 발생해도 계약이 깨지지 않는지 확인.
4. 계약이 깨지면 배포 차단

이를 통해 소비자는 Mock, MockMQ을 통해서도 안전히 테스트 가능하다.

**3. 전체 테스트 진행이 어렵다면,**

STG 환경에서 테스트를 하거나,

카나리 배포, 블루/그린 배포를 이용하는 것도 좋다.

**계약 테스트 솔루션:**

Pact:

API 소비자와 API 제공자가 API 계약을 정의하고

검증할 수 있게 해주는 오픈 소스 계약 테스트 도구이다.

다양한 프로그래밍 언어와 프레임워크를 지원한다.

**관측 가능성과 모니터링은 어떻게 다른가?**

**모니터링:**

미리 정의된 일련의 메트릭을 수집, 분석 및 표시 한다.

해당 메트릭에 경고를 부착함으로써, 문제가 발생했는지 알 수 있다.

다만, 문제가 무엇인지와 문제를 해결하는 방법은 제공되지 않는다.

**관측 가능성(Observability):**

능동적 디버깅, 패턴 찾기, MSA 입출력을 따라가며

시스템 동작에 대한 통찰력을 얻게한다.

시스템에서 개별 요청, 트랜잭션 또는 이벤트 흐름을 따라갈 수 있게 하고,

병목 현상을 발견하고 격리시킬 수 있다.

문제가 존재한다고만 알려주는 것이 아니다.

**관측 가능성의 3가지 주요 사항:**

1. 분산 로깅

2. 메트릭

3. 분산 추적:

요청이 여러 개의 서비스를 통해 이동하는 경로와

해당 요청을 처리하는데 걸리는 시간

(뒤에서 관측 가능성의 3가지 주요 사항을 더 다룬다.)

**분산 추적 도구 프레임워크**

**OpenTelemetry:**

계측, 수집, 그리고 지표 로그, 추적을 내보내는 데 사용되는 API, SDK, 도구들의 집합이다.

오픈 소스로 제공되며 C++, #/.NET, Erlang/Elixir, Go, Java, JavaScript, PHP, Python, Ruby, Rust, Swift 등 다양한 프로그래밍 언어로 사용할 수 있다.

**분산 추적 백엔드**

**Jaeger:**

오픈 소스의 분산 추적 플랫폼으로, 클라우드 네이티브하며 무한히 확장 가능하고 100% 무료이다. 분산 워크플로우 모니터링, 성능 병목의 근본 원인 추적, 서비스 의존성 분석 등을 가능하게 한다.

계측을 위해 OpenTelemetry가 필요하다.

**Zipkin:**

또 다른 오픈 소스 분산 추적 시스템이다.

UI로 제공되는 데이터는 메모리에 저장되거나 Apache Cassandra 또는 Elasticsearch 내에 영구적으로 저장된다. 원래 2010년에 Twitter에서 개발되었으며 Google의 Dapper 논문을 기반으로 하고 있다.

**분산 로깅:**

MSA 환경에서 로그가 각 서비스 별로 분산되어 있으면,

이를 확인하는데 어려움을 느낄 수 있다.

1.

로그를 중앙 집중화 하고, 확장 가능한 시스템으로 하면 좋다.

이 시스템에서 로그를 구문 분석하고, 색인화 하고,

패턴이나 텍스트 검색으로 쉽게 검색되게 해야 한다.

로그는 사람도 컴퓨터도 쉽게 읽을 수 있어야 한다.

ex\_) [date][logLevel][pid, tid][ip][message]

2.

Json, XML 같은 구조화된 로그도 좋다.

3.

로그 레벨(심각도)를 할당 하는 것도 좋다.

로그 레벨은 일반적으로 다음과 같이 분류될 수 있다.

1. TRACE

2. DEBUG

3. INFO

4. WARN

5. ERROR

6. FATAL

이 로그 레벨을 통해서

로그를 필터링 하거나,

주기적으로 심각도 수준에 따른

이벤트를 탐색 및 그룹화 할 수 있다.

4.

가능한 많은 컨택스트 정보를 로그에 남기는 것이 좋다.

ex\_) requsetId, serviceId, ipAddress, stack Trace, query, 매개 변수 등..

5. 개인정보와 불필요한 정보는 남기지 않는다.

저장 공간이랑 처리 비용이 매우 비싸다.

**메트릭:**

지표는 정기적으로 샘플링되어 계속적인 진행을 볼 수 있음으로,

시스템을 모니터링하고 이상 사례를 감지하는데 도움을 준다.

모든 것을 수집하려 하는 것은 좋지 않다.

많은 비용이 들고, 정보가 과부하 된다.

**5가지 골든 타입 시그널:**

**1. Traffic:**

단위 시간 당 요구되는 수요의 양

- RPS

- TPS

- Events Recv/Sec

- Events Send/Sec

- Incoming Req + Outgoing Req/Sec

**2. Errors:**

오류율은 경보를 설정하는데 매우 좋은 신호가 될 수 있다.

- Exception

- 다른 서비스로 부터 Res를 SUCCESS로 받지 못한 경우

- Res 보내는데 까지 지연 시간 임계값을 넘었을 경우

- 처리에 실패한 Event 수

- 송신에 실패한 수

- 실패한 Transaction 수

- Disk 저장 실패 수

**3. Latency**

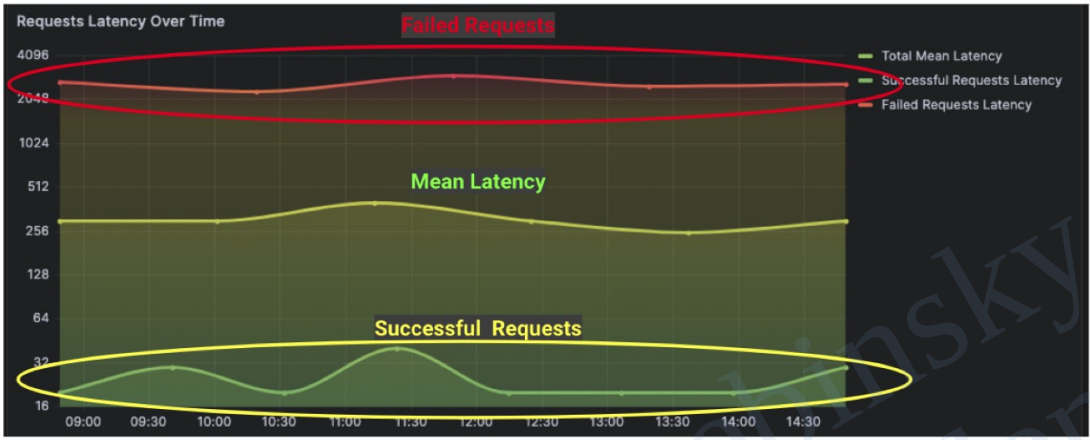
정확히 측정하기 위해서 고려할 것이 있다.

1.

평균 대기 시간 뿐만 아니라, 전체적인 분포를 봐야 한다.

분포가 평평해야 한다. 스파이크가 없는게 좋다.

2.

성공한 응답에 대한 지연시간과

실패한 응답에 대한 지연시간을 분리해야 한다.

실패한 응답 지연시간이,

전체적인 응답 시간이 빠르거나, 느리게 보이게 만든다.

**4. Saturation(포화)**

- MQ에 큐잉 되어 있는 항목 수

- 서비스에 예약되어 있는 작업 수

**5. Utilization (활용도)**

특정 자원이 얼마나 바쁜지 확인할 수 있다.

대부분 CPU 사용량을 100% 채우기 전에, 성능 저하가 발생한다.

- CPU 사용량 [0~100%]

- 메모리 사용량 [0~100%]

- Disk/DB 용량 [0~100%]

**분산 추적:**

클라 장치에서부터 전체 시스템을 흐르는 요청을 추적한다.

백엔드 서비스와 DB까지 모두 포함한다.

요청이 추적되는 동안, 우리는 시스템 각 부분이 처리하는데

걸리는 시간에 대한 중요한 성능 정보를 수집한다.

일반적으로 분산 추적은 정확히 무엇이 일어나고 있는지

알려주기에는 충분하지 않지만,

특정 구성 요소나 두 구성 요소 간의 통신 문제로 범위를 좁힐 수는 있다.

Trace는 Span의 그룹이며, 요청이 시스템의 서비스들을 통과하는 것을 추적한 것을 말한다.

Span은 각 서비스에서 이뤄지는 작업 단위를 의미한다.

Span은 지연 시간 같은 것을 가질 수 있다.

이런 Span들을 싹 모으면 전체 흐름과 지연 시간이 나온다.

(Span은 계층적으로 구조화할 수도 있다.)

보통 이러한 수집은 서비스 인스턴스와

동일한 호스트에서 별도의 프로세스로 실행되는 에이전트에 의해 수집된다.

이런 에이전트들은 그 추적 데이터를 MQ에 전송한다.

이런 추적 데이터는 전용 서비스에 의해 DB에 저장되고,

나중에 개발자는 이를 UI로 살펴볼 수 있다.

**단점:**

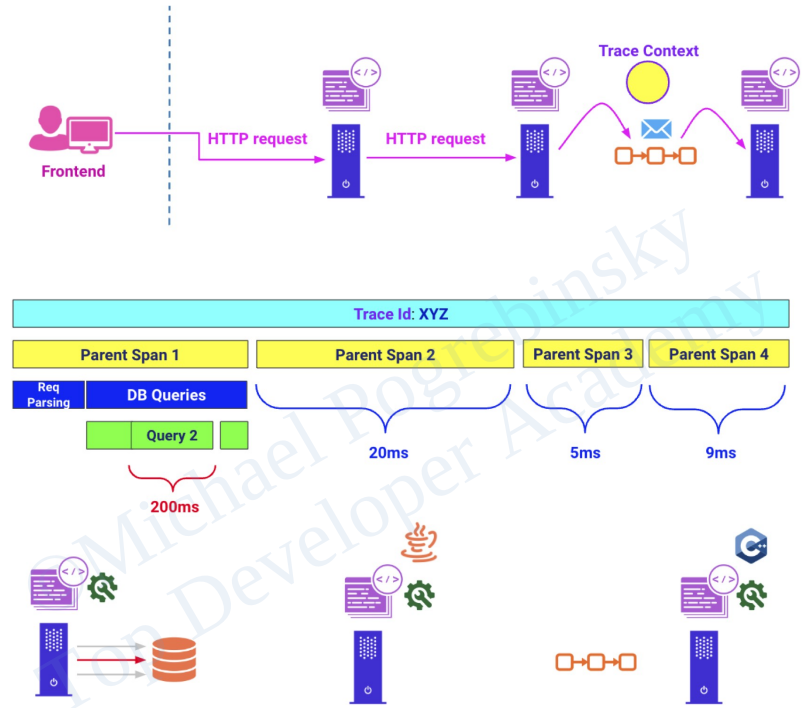
1. Span을 지정하는 것을 매번 수동으로 하게될 수 있으며,

Span의 범위를 잘 지정해야 한다.

2. 비용이 많이 발생할 수 있다.

(많은 데이터가 존재하여, 분석이 어려울 수도 있다.)

3. 성능도 먹는다.



**배포 관련:**

가상 머신은 자체 OS, 가상 리소스를 가진 가상 컴퓨터 처럼 동작한다.

가상 머신은 하이퍼 바이저에 의해 실제로 리소스를 할당되고 관리 된다.

멀티 테넌시는 게임, 스트리밍 서비스, 트레이딩 시스템에는 부적합 하다.

위와 같은 사용처는 싱글 테넌시를 사용해야 한다.

**멀티 테넌시의 단점:**

1. 보안 문제 발생할 수 있다. (가능성 희박)

2. Nosiy Neighbor 문제로 성능이 저하될 수 있다. (가능성 낮음)

호스트 머신의 자원을 정확하게 분리하기 어렵기 때문에,

같은 머신에 할당된 다른 사용자에 의해 성능 저하를 받을 수 있다.

**싱글 테넌시:**

비용이 더 비싸다.

보안, 성능 걱정 없다.

**전용 호스트:**

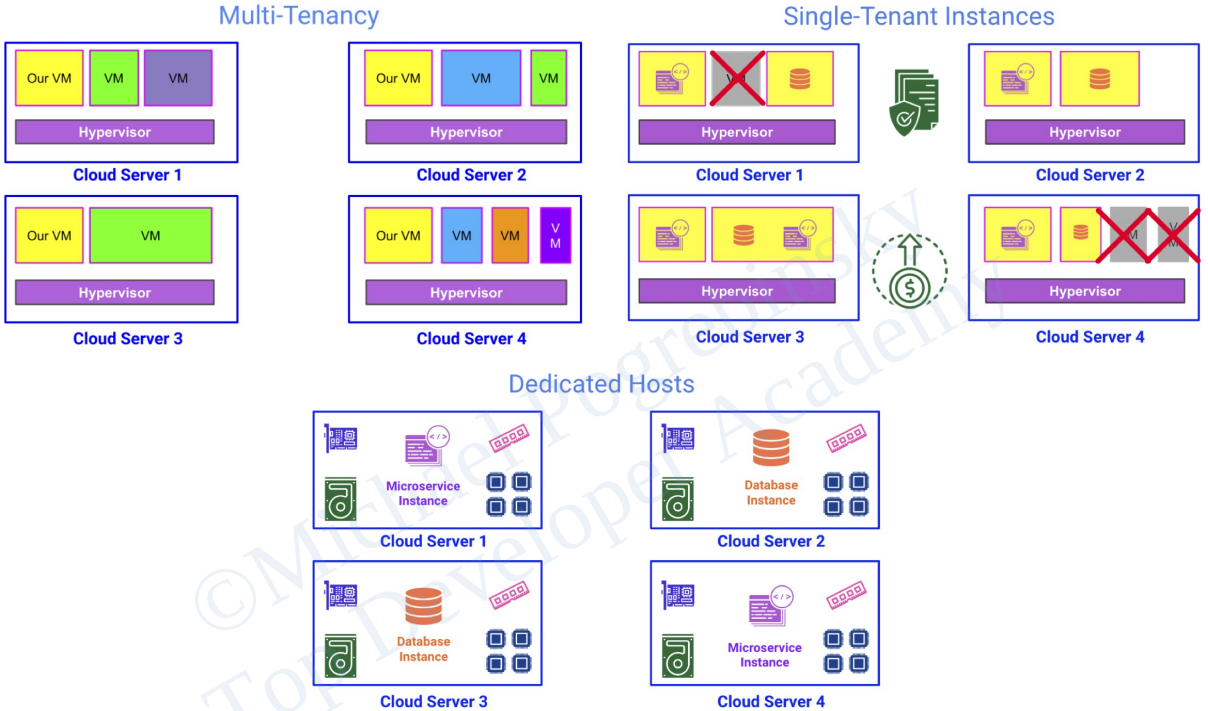
가장 비싸다.

보안, 성능 다 좋다.

**전용 호스트와 싱글 테넌시의 차이?**

싱글 테넌시는 애플리케이션 또는 가상 머신(VM) 수준에서 격리되지만,

전용 호스트는 물리적 서버를 완전히 점유한다는 차이가 있다.



**서버리스(Serverless):**

개발자가 서버 관리를 신경 쓰지 않고

코드 실행에 집중할 수 있는 클라우드 컴퓨팅 모델이다.

**FaaS(Function as a Service):**

서버리스의 한 형태로, 개별 함수를 필요할 때만 실행하는 방식이다.

**FaaS 장점**

1. 인프라 비용을 줄일 수 있다. (트래픽 많으면 인프라 비용 개폭발함)

2. ScaleOut 등을 관리하는 오버헤드를 줄일 수 있다.

**FaaS 단점**

1. 트래픽 패턴이 바뀌면 아주아주 큰 비용이 발생할 수 있다.

2. 서버리스 배포 성능을 예측하기 어렵다.

3. 보안 수준이 낮다 (멀티 테넌시 방식)

**컨테이너 장점:**

개발 환경과 운영 환경을 맞추기 위해 컨테이너를 쓰는게 좋다.

컨테이너를 쓰면 클라우드 벤더에 락인되지 않을 수 있다.

환경간 이동성이 엄청 좋아진다.

배포가 빨라진다.

**컨테이너 오케스트레이션 사용 시:**

