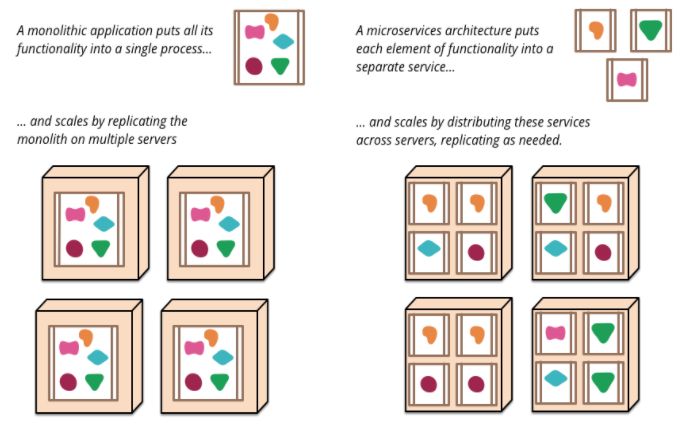
|  |
| --- |
|  |
| **아키텍처 연구** |
| **MSA 및 아키텍처 패턴(layer) 분석** |
|  |
| **KCC정보통신 정보기술연구소 아키텍처 연구팀** |
|  |
| **인턴 도 채 원** |

|  |
| --- |
|  |

1. **마이크로 서비스 아키텍처(MSA: Micro Service Architectures)**
2. **개념**

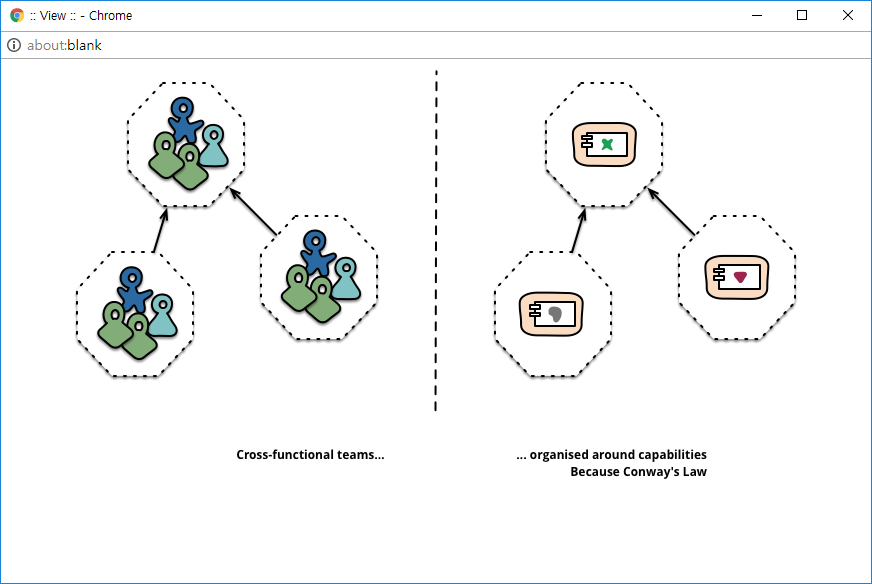
* 독립적이고 단순한 개병 서비스들로 전체의 서비스를 구성
* 단일 어플리케이션을 나누어서 작은 서비스들의 조합으로 구성
* 모놀리틱 문제의 해결



<그림1> 모놀리틱과 마이크로서비스 구성 방식

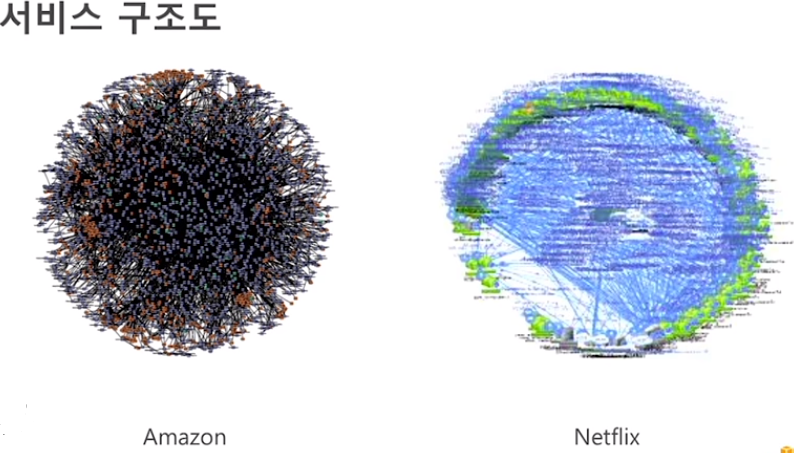
1. **특징**

* 컴포넌트를 독립적인 서비스 단위로 분리
* 명확한 업무 경계
* 데이터도 서비스별로 분리
* 서비스간의 통신은 가볍고 단순한 구조
* 중앙에서 관리하는 부분은 최소한으로
* Conway’s Law적용: 소프트웨어의 구조는 그 소프트웨어를 만드는 팀의 구조와 일치한다



<그림2> Conway’s Law

1. **MSA적용사례**



<그림3> Amazon과 Netflix의 서비스 구조도

1. Netflix

* 매주 반복적으로 배포되는 애플리케이션 코드베이스에 모든 사람들이 한꺼번에 붙어서 작업 => 수백 개의 microservices 쪼개고 팀도 소규모로 변경
* 모든 인프라에 대한 실패를 가정하고 인프라를 운영함 (FIT: Fault Injection Testing)

1. Uber

* 비교적 초기부터 microservices 도입. 대부분 python으로 구현되어 있으나 DB는 다양하게 많이 쪼개서 사용 중

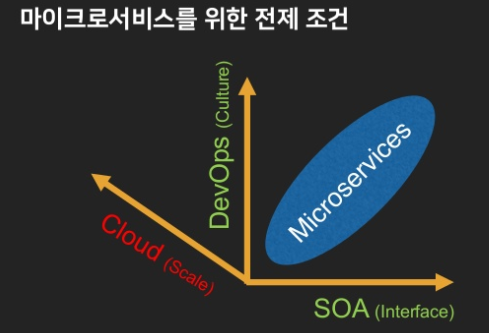
1. Nike의 클라우드 기반 마이크로서비스 구축사례

* https://www.youtube.com/watch?v=k-xp-cYrSTg

1. Amazon.com

* 수 천 개 팀 (자율적 DevOps팀) X 마이크로서비스 아키텍처 X 지속적 배포 X 다양한 개발 환경 = 연간 5천만회 배포

1. Google, MicroSoft, Ebay, Twitter 등
2. **MSA 적용기술**



<그림4> MS를 위한 전제조건

* 수많은 배포작업이 수반됨. 릴리즈가 개별적으로 이루어지는 특성상 이를 별도의 운영 팀에서 일괄적으로 관리하는 것은 불가능에 가까우며 일련의 작업들을 자동화 시키지 않으면 과다한 업무량이 발생하게 되며, 이를 해결하기 위해 마이크로서비스 개발에 있어서 DevOps의 도입은 필수요소임.

1. **마이크로 서비스 아키텍처의 장/단점**
   * 1. 장점

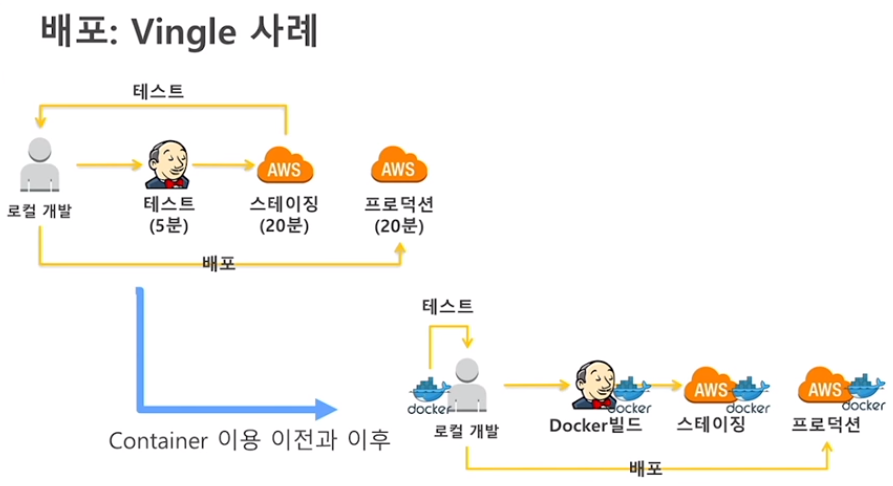
* 모듈을 명확히 분리하기 쉬움
* 시스템 전체가 죽는 경우를 최소화
* 빌드/배포 용이, 확장 용이, 장애로 인한 영향 격리
* 기술의 다양성, 팀 운영의 자율성이 높아짐
  + 1. 단점
* 서비스간 커뮤니케이션 이슈
* 시스템 전체 정합성, 무결성 관리 어려움
* 설계 시의 고려해야 할 사항이 많아짐
* 서비스의 세분화됨에 따라, 서비스 간의 코디네이션(Chief Architect)이 필요
* 배포와 운영에 자동화를 사용하지 않으면, 오히려 기존 방식보다 더 어려움(개별 서비스이기 때문에 수량이 많아짐)

1. **마이크로서비스 설계시 고려사항들**

* 각 개별 서비스에 대한 기능 적합성 및 성능에 대한 효율성
* 서비스 간의 호환성
* 각 서비스에 대한 신뢰성 및 서비스 간의 통신 시의 보안
* 개별 서비스에 대한 유지보수성 및 이식성

**사. 결론**

* 마이크로 서비스 아키텍처는 이미 기존 서비스 운영을 잘하고 있고, 장애 처리나 모니터링 등을 잘하고 있는 조직이 다음 단계로 진화하는 아키텍처로 선정하는 것이 좋음
* MSA로 만들어진 시스템을 운영, 개발하는 입장에서 구조적으로 인터페이스와, 책임이 강제되기 때문에 오히려 서브시스템의 설계나 명세가 명확하지 않아도 다이나믹하게 최적의 구조를 찾아가는 속도가 빠르고 이터레이션이 빠르기 때문에 하루에 여러 번 배포가 가능함



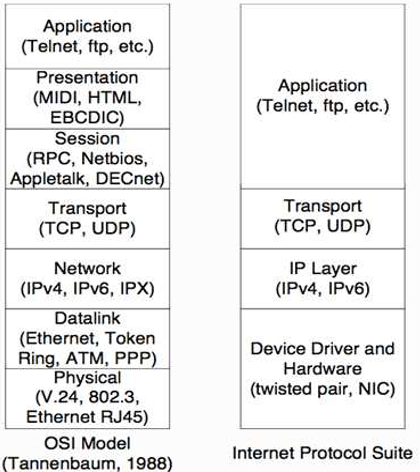
<그림5> Docker를 이용한 배포

1. **아키텍처 패턴 Layer**
2. **개념**

* 특정 추상 레벨에 있는 서브태스크들끼리 서로 묶어서 하나의 그룹으로 분류하는 방식
* 하위 수준의 이슈를 상위 수준에 이슈와 분리시켜 소프트웨어의 재사용성을 높여주는 패턴

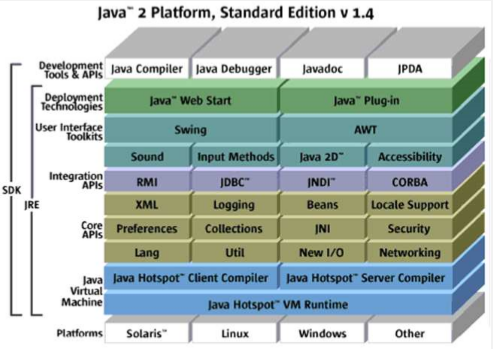
1. **특징**
   * 시스템의 일부분으로 변경을 지역화하여 시스템의 다른 부분에 대한 영향을 최소화함으로써, 애플리케이션의 관리를 쉽게 하여 전반적인 애플리케이션의 유연성(flexibility)을 향상시킨다.
   * 컴포넌트 사이에 관심을 분리함으로써 유연성(flexibility), 유지보수성(maintainability), 확장성(scalability)을 향상시킨다.
   * 컴포넌트를 여러 애플리케이션에 재사용할 수 있다.
   * 다른 팀과의 종속성을 최소화한 상태로 각 팀은 자신의 작업을 수행할 수 있으며, 각 팀은 자신의 능력에 맞는 분야의 작업을 효율적으로 진행할 수 있다.
   * 각 컴포넌트는 내부적으로 강한 응집력(cohesion)을 가져야 한다.
   * 컴포넌트 사이에는 느슨한 결합도(loosely-coupled)를 가져야 한다.
   * 다양한 컴포넌트들은 독립적으로 배포되며 관리되어야 한다.
   * 너무 많은 레이어를 사용하면 성능에 부정적인 영향을 미치게 된다.
2. **적용사례**

* 네트워크, 프로토콜 아키텍처(e.g. OSI 7 layer)



<그림6> OSI7Layer예제

* 가상 머신(e.g. interpreters, JVM)

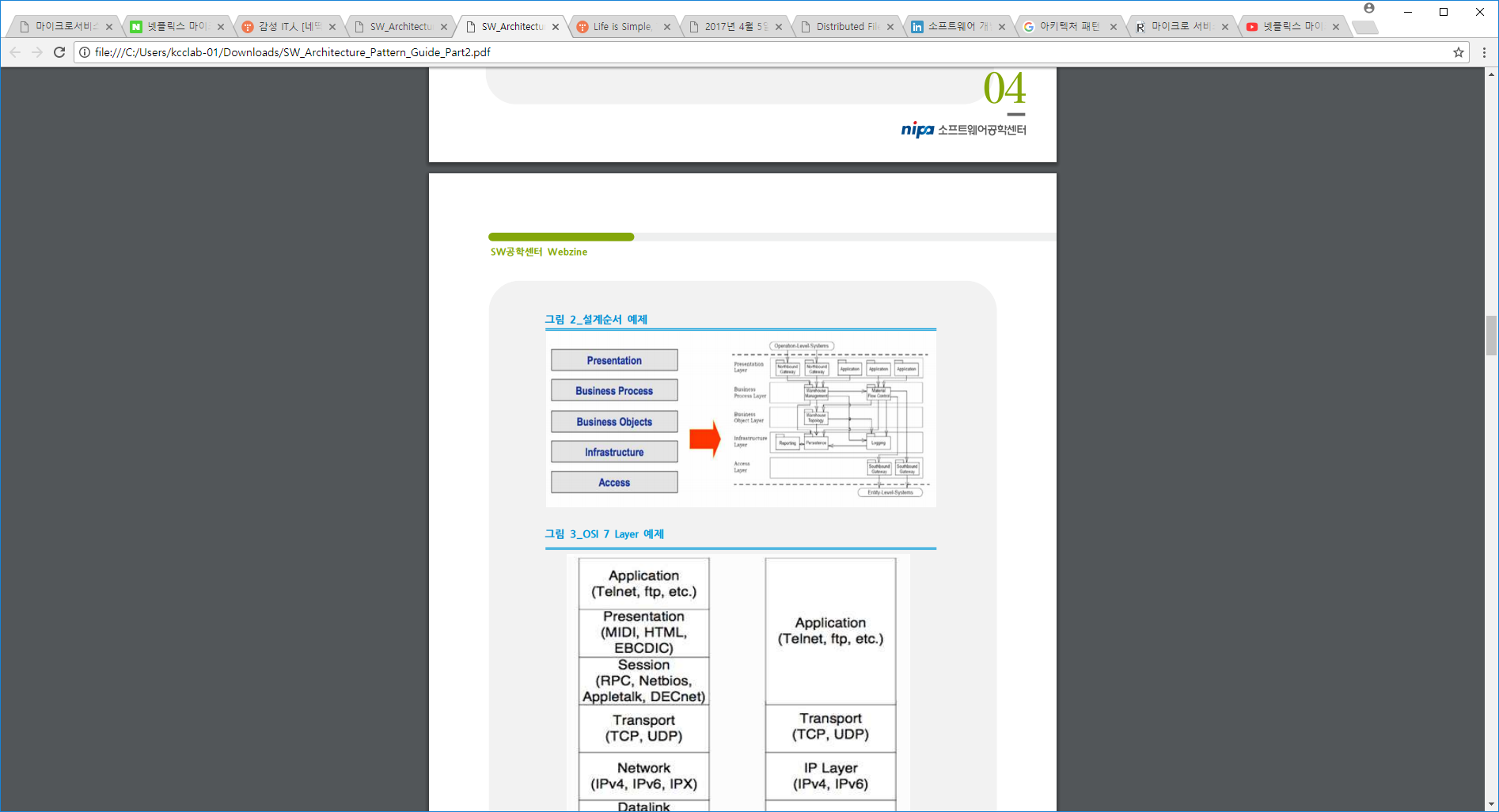


<그림7> JavaVirtual Machine예제

1. **설계순서**
2. 계층 별로 모듈을 묶는 추상 기준을 정의
3. 추상기준에 따라 계층을 몇 레벨로 나눌지 결정
4. 계층마다 역할 및 태스크 부여
5. 계층별 제공서비스를 상세히 정의
6. 계층별 상세 인터페이스 정의
7. 시스템 기능이 계층에서 동작하는 것이 가능한지 확인

(예: 유스케이스 시나리오를 시뮬레이션 하는 방식)

1. 계층 내부에 대한 구조 정의
2. 인접한 계층 간의 통신 방식 정의
3. 예외 처리 방식을 정의



<그림4> layer패턴을 이용한 설계순서 예제

1. **Layer패턴의 장/단점**
2. 장점

* 계층별 연동을 한정할 수 있어 Loosely coupled 원칙을 지킬 수 있음
* 변화에 대한 영향력을 한정할 수 있어 코딩이나 테스트를 계층별로 진행할 수 있음
* 인터페이스 정의가 잘 되어 있다면 계층을 통째로 교체할 수 있음
* 모듈의 재사용성을 높여 유지보수성이나 이식성이 좋은 패턴임

1. 단점

* 계층의 원칙을 지키기 위해 각 계층을 모두 거쳐야 하므로 성능 측면에 불이익을 받을 수 있음
* 계층을 구분하기 어렵고 잘못 구분할 경우 설계 수정이 빈번히 발생할 수 있음
* 계층의 적절한 개수 및 규모를 정의하는 것이 어려움