# 아키텍쳐 설계서

목차

## 1. 소프트웨어 아키텍처의 개념

가. 정의

λ 소프트웨어 컴포넌트들과 그들의 외부적으로 보여지는 특성, 그리고 그 들 상호간의 관계들로 구성되는 해당 시스템의 구조 또는 구조들

⌊ 외부적 특성 : 제공서비스, 장애처리 매커니즘, 공유자원 활용 등

⌊ 아키텍처는 소프트웨어의 요소를 정의

⌊ 시스템은 하나 이상의 구조로 구성될 수 있음

⌊ 소프트웨어가 들어간 시스템에는 전부 소프트웨어 아키텍처 존재

⌊ 시스템 요소의 행위를 다른 요소에서 인식할 수 있다면 이는 아키 텍처로 표현이 가능

나. 등장 배경

λ Stakeholder간의 관점의 조율을 통한 시스템 최적화

λ 요구사항들 간의 개념상의 충돌 조정

⌊ 성능, 유지보수성, 개발단가

λ 우선순위 결정과 요구사항들 간의 상반성 분석을 위한 작업 필요

λ 품질향상 필요 : 아키텍처 스타일 결정 -> 프레임워크 결정 -> 디자인 패턴 결정 -> 품질에 영향

다. 초기 설계 원칙

λ 아키텍처는 개발의 제약 사항을 정의

λ 아키텍처는 개발 구조 결정

λ 시스템의 품질 속성을 장려하거나 억제

λ 시스템 품질을 예측 가능해야 함

λ 시스템 변화 이유를 좀 더 쉽게 파악하거나 관리 가능

λ 점진적 프로토타입 개발에 도움

λ 정확한 비용과 일정 예측 가능

라. 이전 가능한 모델(재 사용성)

λ 소프트웨어 프로덕트 라인은 아키텍처를 공유

λ 시스템 요소를 외부에서 개발해 구현 가능

λ 과유.불급 – 아키텍처는 설계 방법을 적당한 수준으로 제한

λ 템플릿 기반의 개발이 가능해짐

λ 아키텍처는 훈련의 기본

마. 아키텍처 불일치

λ 호환 / 교환 용이성을 위해 도입한 컴포넌트 / 서브 시스템으로 인하여 개발이 많이 필요할 수도 있고, 아키텍처의 목적을 위배하여 아키텍처 의 복잡성을 유발 시키는 현상 – 데이비드 갈란

## 2. 쿠버네티스의 개념

### 쿠버네티스란 무엇인가?

이 페이지에서는 쿠버네티스 개요를 설명한다.

쿠버네티스는 컨테이너화된 워크로드와 서비스를 관리하기 위한 이식성이 있고, 확장가능한 오픈소스 플랫폼이다. 쿠버네티스는 선언적 구성과 자동화를 모두 용이하게 해준다. 쿠버네티스는 크고, 빠르게 성장하는 생태계를 가지고 있다. 쿠버네티스 서비스, 기술 지원 및 도구는 어디서나 쉽게 이용할 수 있다.

쿠버네티스란 명칭은 키잡이(helmsman)나 파일럿을 뜻하는 그리스어에서 유래했다. 구글이 2014 년에 쿠버네티스 프로젝트를 오픈소스화했다. 쿠버네티스는 프로덕션 워크로드를 대규모로 운영하는 15 년 이상의 구글 경험과 커뮤니티의 최고의 아이디어와 적용 사례가 결합되어 있다.

#### 여정 돌아보기

시간이 지나면서 쿠버네티스가 왜 유용하게 되었는지 살펴보자.

**전통적인 배포 시대:** 초기 조직은 애플리케이션을 물리 서버에서 실행했었다. 한 물리 서버에서 여러 애플리케이션의 리소스 한계를 정의할 방법이 없었기에, 리소스 할당의 문제가 발생했다. 예를 들어 물리 서버 하나에서 여러 애플리케이션을 실행하면, 리소스 전부를 차지하는 애플리케이션 인스턴스가 있을 수 있고, 결과적으로는 다른 애플리케이션의 성능이 저하될 수 있었다. 이에 대한 해결책은 서로 다른 여러 물리 서버에서 각 애플리케이션을 실행하는 것이 있다. 그러나 이는 리소스가 충분히 활용되지 않는다는 점에서 확장 가능하지 않았으므로, 물리 서버를 많이 유지하기 위해서 조직에게 많은 비용이 들었다.

**가상화된 배포 시대:** 그 해결책으로 가상화가 도입되었다. 이는 단일 물리 서버의 CPU 에서 여러 가상 시스템 (VM)을 실행할 수 있게 한다. 가상화를 사용하면 VM 간에 애플리케이션을 격리하고 애플리케이션의 정보를 다른 애플리케이션에서 자유롭게 액세스 할 수 없으므로, 일정 수준의 보안성을 제공할 수 있다.

가상화를 사용하면 물리 서버에서 리소스를 보다 효율적으로 활용할 수 있으며, 쉽게 애플리케이션을 추가하거나 업데이트할 수 있고 하드웨어 비용을 절감할 수 있어 더 나은 확장성을 제공한다. 가상화를 통해 일련의 물리 리소스를 폐기 가능한(disposable) 가상 머신으로 구성된 클러스터로 만들 수 있다.

각 VM 은 가상화된 하드웨어 상에서 자체 운영체제를 포함한 모든 구성 요소를 실행하는 하나의 완전한 머신이다.

**컨테이너 개발 시대:** 컨테이너는 VM 과 유사하지만 격리 속성을 완화하여 애플리케이션 간에 운영체제(OS)를 공유한다. 그러므로 컨테이너는 가볍다고 여겨진다. VM 과 마찬가지로 컨테이너에는 자체 파일 시스템, CPU, 메모리, 프로세스 공간 등이 있다. 기본 인프라와의 종속성을 끊었기 때문에, 클라우드나 OS 배포본에 모두 이식할 수 있다.

컨테이너는 다음과 같은 추가적인 혜택을 제공하기 때문에 인기가 있다.

- 기민한 애플리케이션 생성과 배포: VM 이미지를 사용하는 것에 비해 컨테이너 이미지 생성이 보다 쉽고 효율적임.
- 지속적인 개발, 통합 및 배포: 안정적이고 주기적으로 컨테이너 이미지를 빌드해서 배포할 수 있고 (이미지의 불변성 덕에) 빠르고 쉽게 롤백할 수 있다.
- 개발과 운영의 관심사 분리: 배포 시점이 아닌 빌드/릴리스 시점에 애플리케이션 컨테이너 이미지를 만들기 때문에, 애플리케이션이 인프라스트럭처에서 분리된다.
- 가시성은 OS 수준의 정보와 메트릭에 머무르지 않고, 애플리케이션의 헬스와 그 밖의 시그널을 볼 수 있다.
- 개발, 테스팅 및 운영 환경에 걸친 일관성: 랩탑에서도 클라우드에서와 동일하게 구동된다.
- 클라우드 및 OS 배포판 간 이식성: Ubuntu, RHEL, CoreOS, 온-프레미스, 주요 퍼블릭 클라우드와 어디에서든 구동된다.
- 애플리케이션 중심 관리: 가상 하드웨어 상에서 OS 를 실행하는 수준에서 논리적인 리소스를 사용하는 OS 상에서 애플리케이션을 실행하는 수준으로 추상화 수준이 높아진다.
- 느슨하게 커플되고, 분산되고, 유연하며, 자유로운 마이크로서비스: 애플리케이션은 단일 목적의 머신에서 모놀리식 스택으로 구동되지 않고 보다 작고 독립적인 단위로 쪼개져서 동적으로 배포되고 관리될 수 있다.
- 리소스 격리: 애플리케이션 성능을 예측할 수 있다.
- 자원 사용량: 리소스 사용량: 고효율 고집적.

## 쿠버네티스가 왜 필요하고 무엇을 할 수 있나

컨테이너는 애플리케이션을 포장하고 실행하는 좋은 방법이다. 프로덕션 환경에서는 애플리케이션을 실행하는 컨테이너를 관리하고 가동 중지 시간이 없는지 확인해야한다. 예를 들어 컨테이너가 다운되면 다른 컨테이너를 다시 시작해야한다. 이 문제를 시스템에 의해 처리한다면 더 쉽지 않을까?

그것이 쿠버네티스가 필요한 이유이다! 쿠버네티스는 분산 시스템을 탄력적으로 실행하기 위한 프레임 워크를 제공한다. 애플리케이션의 확장과 장애 조치를 처리하고, 배포 패턴 등을 제공한다. 예를 들어, 쿠버네티스는 시스템의 카나리아 배포를 쉽게 관리 할 수 있다.

쿠버네티스는 다음을 제공한다.

- **서비스 디스커버리와 로드 밸런싱** 쿠버네티스는 DNS 이름을 사용하거나 자체 IP 주소를 사용하여 컨테이너를 노출할 수 있다. 컨테이너에 대한 트래픽이 많으면, 쿠버네티스는 네트워크 트래픽을 로드밸런싱하고 배포하여 배포가 안정적으로 이루어질 수 있다.
- **스토리지 오케스트레이션** 쿠버네티스를 사용하면 로컬 저장소, 공용 클라우드 공급자 등과 같이 원하는 저장소 시스템을 자동으로 탑재 할 수 있다.
- **자동화된 롤아웃과 롤백** 쿠버네티스를 사용하여 배포된 컨테이너의 원하는 상태를 서술할 수 있으며 현재 상태를 원하는 상태로 설정한 속도에 따라 변경할 수 있다. 예를 들어 쿠버네티스를 자동화해서 배포용 새 컨테이너를 만들고, 기존 컨테이너를 제거하고, 모든 리소스를 새 컨테이너에 적용할 수 있다.
- **자동화된 빈 패킹(bin packing)** 컨테이너화된 작업을 실행하는데 사용할 수 있는 쿠버네티스 클러스터 노드를 제공한다. 각 컨테이너가 필요로 하는 CPU 와 메모리(RAM)를 쿠버네티스에게 지시한다. 쿠버네티스는 컨테이너를 노드에 맞추어서 리소스를 가장 잘 사용할 수 있도록 해준다.
- **자동화된 복구(self-healing)** 쿠버네티스는 실패한 컨테이너를 다시 시작하고, 컨테이너를 교체하며, '사용자 정의 상태 검사'에 응답하지 않는 컨테이너를 죽이고, 서비스 준비가 끝날 때까지 그러한 과정을 클라이언트에 보여주지 않는다.
- **시크릿과 구성 관리** 쿠버네티스를 사용하면 암호, OAuth 토큰 및 SSH 키와 같은 중요한 정보를 저장하고 관리 할 수 있다. 컨테이너 이미지를 재구성하지 않고 스택 구성에 시크릿을 노출하지 않고도 시크릿 및 애플리케이션 구성을 배포 및 업데이트 할 수 있다.

## 쿠버네티스가 아닌 것

쿠버네티스는 전통적인, 모든 것이 포함된 Platform as a Service(PaaS)가 아니다. 쿠버네티스는 하드웨어 수준보다는 컨테이너 수준에서 운영되기 때문에, PaaS 가 일반적으로 제공하는 배포, 스케일링, 로드 밸런싱과 같은 기능을 제공하며, 사용자가 로깅, 모니터링 및 알림 솔루션을 통합할 수 있다. 하지만, 쿠버네티스는 모놀리식(monolithic)이 아니어서, 이런 기본 솔루션이 선택적이며 추가나 제거가 용이하다. 쿠버네티스는 개발자 플랫폼을 만드는 구성 요소를 제공하지만, 필요한 경우 사용자의 선택권과 유연성을 지켜준다.

쿠버네티스는:

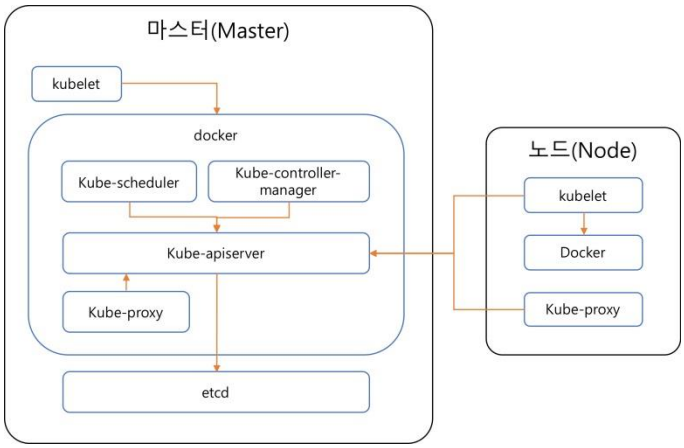
- 지원하는 애플리케이션의 유형을 제약하지 않는다. 쿠버네티스는 상태 유지가 필요 없는(stateless) 워크로드, 상태 유지가 필요한(stateful) 워크로드, 데이터 처리를 위한 워크로드를 포함해서 극단적으로 다양한 워크로드를 지원하는 것을 목표로 한다. 애플리케이션이 컨테이너에서 구동될 수 있다면, 쿠버네티스에서도 잘 동작할 것이다.
- 소스 코드를 배포하지 않으며 애플리케이션을 빌드하지 않는다. 지속적인 통합과 전달과 배포, 곧 CI/CD 워크플로우는 조직 문화와 취향에 따를 뿐만 아니라 기술적인 요구사항으로 결정된다.
- 애플리케이션 레벨의 서비스를 제공하지 않는다. 애플리케이션 레벨의 서비스에는 미들웨어(예, 메시지 버스), 데이터 처리 프레임워크(예, Spark), 데이터베이스(예, MySQL), 캐시 또는 클러스터 스토리지 시스템(예, Ceph) 등이 있다. 이런 컴포넌트는 쿠버네티스 상에서 구동될 수 있고, 쿠버네티스 상에서 구동 중인 애플리케이션이 Open Service Broker 와 같은 이식 가능한 메커니즘을 통해 접근할 수도 있다.
- 로깅, 모니터링 또는 경보 솔루션을 포함하지 않는다. 개념 증명을 위한 일부 통합이나, 메트릭을 수집하고 노출하는 메커니즘을 제공한다.
- 기본 설정 언어/시스템(예, Jsonnet)을 제공하거나 요구하지 않는다. 선언적 명세의 임의적인 형식을 목적으로 하는 선언적 API 를 제공한다.
- 포괄적인 머신 설정, 유지보수, 관리, 자동 복구 시스템을 제공하거나 채택하지 않는다.
- 추가로, 쿠버네티스는 단순한 오케스트레이션 시스템이 아니다. 사실, 쿠버네티스는 오케스트레이션의 필요성을 없애준다. 오케스트레이션의 기술적인 정의는 A 를 먼저 한 다음, B 를 하고, C 를 하는 것과 같이 정의된 워크플로우를 수행하는 것이다. 반면에, 쿠버네티스는 독립적이고 조합 가능한 제어 프로세스들로 구성되어 있다. 이 프로세스는 지속적으로 현재 상태를 입력받은 의도한 상태로 나아가도록 한다. A 에서 C 로 어떻게 갔는지는 상관이 없다. 중앙화된 제어도 필요치 않다. 이로써 시스템이 보다 더 사용하기 쉬워지고, 강력해지며, 견고하고, 회복력을 갖추게 되며, 확장 가능해진다.

## 3. 쿠버네티스의 중요성 및 특징

쿠버네티스 클러스터는 크게 2 가지 종류로 구성됩니다. 클러스터를 관리하는 역할을 하는 마스터(master)와 실제 컨테이너를 실행시키는 작업을 하는 노드(node)입니다. 구성도는 다음과 같습니다.

마스터에는 etcd, kube-apiserver, kube-scheduler, kube-controller-manager, kubelet, kube-proxy, docker 등이 실행됩니다. 각 프로세스들이 다른 장비에 별개로 떠도 실제 쿠버네티스 클러스터를 운영하는데 이상은 없지만 마스터 장비 1 대에는 위에 있는 프로세스 한 묶음을 같이 실행하는게 일반적인 구성입니다. 마스터는 고가용성을 위해서 3 대 정도를 실행해서 운영합니다. 평소에 실제 리더로서 클러스터를 관리하는 마스터는 1 대이고 나머지 2 대는 대기중입니다. 그러다가 리더 마스터에 장애가 발생하면 자연스럽게 나머지 2 대중 1 대로 리더역할이 넘어가게 됩니다. 클러스터를 좀 더 안정적으로 운영하려면 마스터를 5 대로 운영할 수도 있습니다.
노드는 쿠버네티스 초기에는 미니언(minion)이라고 불렸었는데, 현재는 노드라고 부릅니다. 쿠버네티스 소스나 데이터 저장구조를 보면 아직까지 미니언이라고 쓰여 있는 부분들이 남아 있습니다. 노드에는 kubelet, kube-proxy, docker 등이 실행됩니다. 실제 사용자가 사용하는 컨테이너들은 대부분 노드에서 실행됩니다

다음 그림을 보면 마스터와 노드가 어떤 구조로 통신하는지 알 수 있습니다.

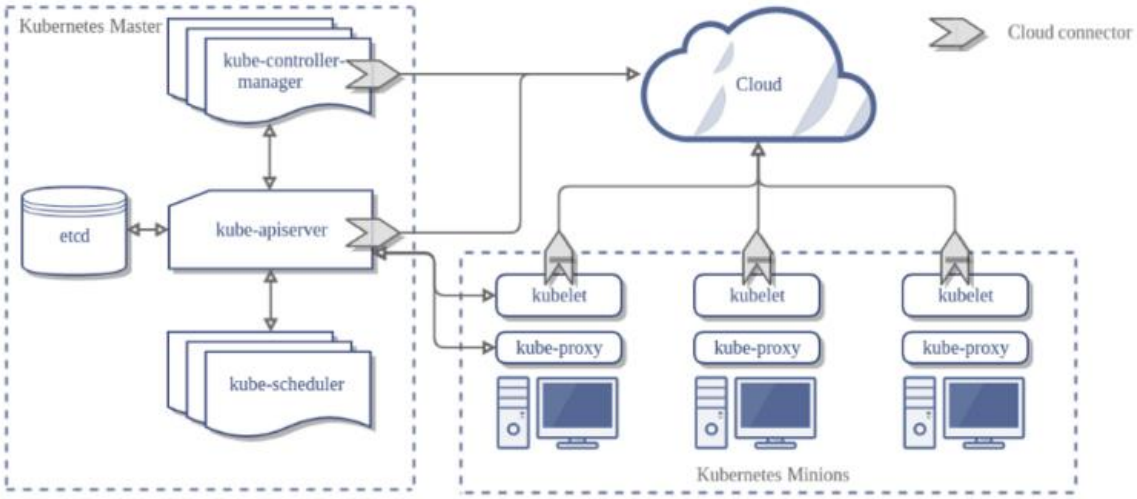



그림을 잘 보시면 각각의 프로세스의 중심에 kube-apiserver 가 있는걸 볼 수 있습니다. 쿠버네티스의 모든 통신의 중심에는 apiserver 가 있습니다. apiserver 를 통해서 다른 프로세스들이 서로 필요한 정보를 주고 받게 됩니다. 그리고 etcd 에 대한 접근도 다른 프로세스는 하지 않고 apiserver 만 하고 있는걸 볼 수 있습니다. 먼저 마스터 서버를 보면 kubelet 이 마스터에 있는 도커를 관리한다는걸 알 수 있고 도커내에 컨테이너로 kuber-scheduler, kube-controller-manager, kube-apiserver, kube-proxy 가 떠 있는걸 볼 수 있습니다.
쿠버네티스 초기에는 이런 쿠버네티스 관리용 프로세스들은 컨테이너가 아니라 직접 서버내의 프로세스로 실행했었는데 최근에는 이렇게 관리용 프로세스들 조차 컨테이너로 실행하고 있습니다.

이 프로세스들은 모두 하이퍼큐브(hyperkube)라는 하나의 바이너리 파일로 컴파일되어 있고 실행할때 각각 다른 옵션을 줌으로써 각각의 역할을 하도록 되어 있습니다.
etcd 는 컨테이너가 아니라 프로세스로 빠져 있는걸 볼 수 있습니다. etcd 까지 컨테이너로 설정할 수 있긴 하지만 여기서는 컨테이너가 아닌 서버 프로세스로 실행해도 구성이 가능하다는걸 보여주기 위해서 따로 실행하는 형식으로 보여드립니다.
마찬가지로 노드를 보면 kubelet 이 도커를 관리하는걸 볼 수 있습니다. kubelet 은 마스터의 apiserver 를 바라보고 있으면서 포드의 생성/관리/삭제를 담당하게 됩니다. 노드의 kube-proxy 는 마스터와는 다르게 컨테이너가 아니라 프로세스로 띄우도록 했는데요. 마스터와 마찬가지로 컨테이너로 띄울 수도 있습니다.

## 4. 쿠버네티스의 구조 및 스타일

쿠버네티스에 대한 개념 이해가 끝났으면, 이제 쿠버네티스가 실제로 어떤 구조로 구현이 되어 있는지 아키텍쳐를 살펴보도록 합시다. 아키텍쳐를 이용하면 동작 원리를 이해할 수 있기 때문에, 쿠버네티스의 사용법을 이해하는데 도움이 됩니다. 특히 쿠버네티스의 구조 및 스타일에서는 위에 내용인 중요성 및 특징과 유사하게 사용된 개념들이 있기 때문에 잘 분리하거나 이해하는 것이 좋습니다.




<그림. 쿠버네티스 아키텍쳐>

출처 - https://kubernetes.io/docs/concepts/architecture/

**마스터와 노드**

쿠버네티스는 크게 마스터(Master)와 노드(Node) 두 개의 컴포넌트로 분리된다.

마스터는 쿠버네티스의 설정 환경을 저장하고 전체 클러스터를 관리하는 역할을 맡고있고, 노드는 파드나 컨테이너 처럼 쿠버네티스 위에서 동작하는 워크로드를 호스팅하는 역할을 한다.

**마스터**

쿠버네티스 클러스터 전체를 컨트럴 하는 시스템으로, 크게 다음과 API 서버, 스케쥴러, 컨트롤러

매니저, etcd 로 구성되어 있다.

**API 서버**

쿠버네티스는 모든 명령과 통신을 API를 통해서 하는데, 그 중심이 되는 서버가 API서버이다.

쿠버네티스의 모든 기능들을 REST API로 제공하고 그에 대한 명령을 처리한다.

**Etcd**

API 서버가 명령을 주고 받는 서버라면, 쿠버네티스 클러스터의 데이타 베이스 역할이 되는 서버로 설정값이나 클러스터의 상태를 저장하는 서버이다. etcd라는 분산형 키/밸류 스토어 오픈소스()https://github.com/coreos/etcd) 로 쿠버네티스 클러스터의 상태나 설정 정보를 저장한다.

**스케쥴러**

스케쥴러는 Pod,서비스등 각 리소스들을 적절한 노드에 할당하는 역할을 한다.

**컨트롤러 매니져**

컨트롤러 매니저는 컨트롤러(Replica controller, Service controller, Volume Controller, Node controller 등)를 생성하고 이를 각 노드에 배포하며 이를 관리하는 역할을 한다.

**DNS**

그림에는 빠져있는데, 쿠버네티스는 리소스의 엔드포인트(Endpoint)를 DNS로 맵핑하고 관리한다. Pod나 서비스등은 IP를 배정받는데, 동적으로 생성되는 리소스이기 때문에 그 IP 주소가 그때마다 변경이 되기 때문에, 그 리소스에 대한 위치 정보가 필요한데, 이러한 패턴을 Service discovery 패턴이라고 하는데, 쿠버네티스에서는 이를 내부 DNS서버를 두는 방식으로 해결하였다.

새로운 리소스가 생기면, 그 리소스에 대한 IP와 DNS 이름을 등록하여, DNS 이름을 기반으로 리소스에 접근할 수 있도록 한다.

**노드**

노드는 마스터에 의해 명령을 받고 실제 워크로드를 생성하여 서비스 하는 컴포넌트이다.

노드에는 Kubelet, Kube-proxy,cAdvisor 그리고 컨테이너 런타임이 배포된다.

**Kubelet**

노드에 배포되는 에이전트로, 마스터의 API서버와 통신을 하면서, 노드가 수행해야 할 명령을 받아서 수행하고, 반대로 노드의 상태등을 마스터로 전달하는 역할을 한다.

**Kube-proxy**

노드로 들어오거는 네트워크 트래픽을 적절한 컨테이너로 라우팅하고, 로드밸런싱등 노드로 들어오고 나가는 네트워크 트래픽을 프록시하고, 노드와 마스터간의 네트워크 통신을 관리한다.

**Container runtime (컨테이너 런타임)**

Pod를 통해서 배포된 컨테이너를 실행하는 컨테이너 런타임이다. 컨테이너 런타임은 보통 도커 컨테이너를 생각하기 쉬운데, 도커 이외에도 rkt (보안이 강화된 컨테이너), Hyper container 등 다양한 런타임이 있다.

**cAdvisor**

cAdvisor는 각 노드에서 기동되는 모니터링 에이전트로, 노드내에서 가동되는 컨테이너들의 상태와 성능등의 정보를 수집하여, 마스터 서버의 API 서버로 전달한다.

이 데이타들은 주로 모니터링을 위해서 사용된다.

전체적인 구조 자체는 복잡하지 않다. 모듈화가 되어 있고, 기능 확장을 위해서 플러그인을 설치할 수 있는 구조로 되어 있다. 예를 들어 나중에 설명하겠지만 모니터링 정보를 저장하는 데이타베이스로는 많이 사용되는 Influx 데이타 베이스 또는 Prometheus 와 같은 데이타 베이스를 선택해서 설치할 수 있고 또는 커스텀 인터페이스 개발을 통해서, 알맞은 저장소를 개발하여 연결이 가능하다.

## 5. GIS 및 RESTful API 구조 및 스타일

### GIS 정의

- GIS(Geographic Information System) 란 인간생활에 필요한 지리정보를 컴퓨터 데이터로 변환하여 효율적으로 활용하기 위한 정보시스템이다.
- 정보시스템이란 의사결정에 필요한 정보를 생성하기 위한 제반 과정으로서 정보를 수집, 관측, 측정하고 컴퓨터에 입력하여 저장, 관리하며 저장된 정보를 분석하여 의사결정에 반영할 수 있는 시스템이다.
- GIS 는 지리적 위치를 갖고 있는 대상에 대한 위치자료와 (spatial data)와 속성자료(attribute data)를 통합·관리하여 지도, 도표 및 그림들과 같은 여러 형태의 정보를 제공한다.
- 즉 GIS 란 넓은 의미에서 인간의 의사결정능력 지원에 필요한 지리정보의 관측과 수집에서부터 보존과 분석, 출력에 이르기까지의 일련의 조작을 위한 정보시스템을 의미한다.
- GIS 는 인간의 현실생활과 밀접한 관계가 있는 모든 자료를 취급하므로서 광범위한 활용분야를 가지고 있다. GIS 의 활용분야는 토지, 자원, 도시, 환경, 교통, 농업, 해양 및 국방에 이르기까지 다양한 산업 전반에 걸쳐 빠르게 발전하고 있다.

### GIS 주요기능

- GIS 는 모든 정보를 수치의 형태로 표현한다. 모든 지리정보가 수치데이터의 형태로 저장되어 사용자가 원하는 정보를 선택하여 필요한 형식에 맞추어 출력할 수 있다. 이것은 기존의 종이지도의 한계를 넘어 이차원 개념의 정적인 상태를 삼차원 이상의 동적인 지리정보의 제공이 가능하다.
- GIS 는 다량의 자료를 컴퓨터 기반으로 구축하여 정보를 빠르게 검색할 수 있으며 도형자료와 속성자료를 쉽게 결합시키고 통합 분석 환경을 제공한다.
- GIS 에서 제공하는 공간분석의 수행 과정을 통하여 다양한 계획이나 정책수립을 위한 시나리오의 분석, 의사결정 모형의 운영, 변화의 탐지 및 분석기능에 활용한다.
- 다양한 도형자료와 속성자료를 가지고 있는 수많은 데이터 파일에서 필요한 도형이나 속성정보를 추출하고 결합하여 종합적인 정보를 분석, 처리할 수 있는 환경을 제공하는 것이 GIS 의 핵심 기능이다.

### 1. REST API의 탄생

---

REST 는 Representational State Transfer 라는 용어의 약자로서 2000 년도에 로이 필딩 (Roy Fielding)의 박사학위 논문에서 최초로 소개되었습니다. 로이 필딩은 HTTP 의 주요 저자 중 한

사람으로 그 당시 웹(HTTP) 설계의 우수성에 비해 제대로 사용되어지지 못하는 모습에 안타까워하며 웹의 장점을 최대한 활용할 수 있는 아키텍처로써 REST 를 발표했다고 합니다.

## 2. REST 구성

쉽게 말해 REST API 는 다음의 구성으로 이루어져있습니다. 자세한 내용은 밑에서 설명하도록 하겠습니다.

- **자원(RESOURCE)** - URI
- **행위(Verb)** - HTTP METHOD
- **표현(Representations)**

## 3. REST 의 특징

1) Uniform (유니폼 인터페이스)

Uniform Interface 는 URI 로 지정한 리소스에 대한 조작을 통일되고 한정적인 인터페이스로 수행하는 아키텍처 스타일을 말합니다.

2) Stateless (무상태성)

REST 는 무상태성 성격을 갖습니다. 다시 말해 작업을 위한 상태정보를 따로 저장하고 관리하지 않습니다. 세션 정보나 쿠키정보를 별도로 저장하고 관리하지 않기 때문에 API 서버는 들어오는 요청만을 단순히 처리하면 됩니다. 때문에 서비스의 자유도가 높아지고 서버에서 불필요한 정보를 관리하지 않음으로써 구현이 단순해집니다.

3) Cacheable (캐시 가능)

REST 의 가장 큰 특징 중 하나는 HTTP 라는 기존 웹표준을 그대로 사용하기 때문에, 웹에서 사용하는 기존 인프라를 그대로 활용이 가능합니다. 따라서 HTTP 가 가진 캐싱 기능이 적용 가능합니다. HTTP 프로토콜 표준에서 사용하는 Last-Modified 태그나 E-Tag 를 이용하면 캐싱 구현이 가능합니다.

4) Self-descriptiveness (자체 표현 구조)

REST 의 또 다른 큰 특징 중 하나는 REST API 메시지만 보고도 이를 쉽게 이해 할 수 있는 자체 표현 구조로 되어 있다는 것입니다.

5) Client - Server 구조

REST 서버는 API 제공, 클라이언트는 사용자 인증이나 컨텍스트(세션, 로그인 정보)등을 직접 관리하는 구조로 각각의 역할이 확실히 구분되기 때문에 클라이언트와 서버에서 개발해야 할 내용이 명확해지고 서로간 의존성이 줄어들게 됩니다.

6) 계층형 구조

REST 서버는 다중 계층으로 구성될 수 있으며 보안, 로드 밸런싱, 암호화 계층을 추가해 구조상의 유연성을 둘 수 있고 PROXY, 게이트웨이 같은 네트워크 기반의 중간매체를 사용할 수 있게 합니다.

## 4. REST API 디자인 가이드

---

REST API 설계 시 가장 중요한 항목은 다음의 2 가지로 요약할 수 있습니다.

**첫 번째,** URI 는 정보의 자원을 표현해야 한다.
**두 번째,** 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE)로 표현한다.

다른 것은 다 잊어도 위 내용은 꼭 기억하시길 바랍니다.

### 4-1. REST API 중심 규칙

1) URI는 정보의 자원을 표현해야 한다. (리소스명은 동사보다는 명사를 사용)

```
GET /members/delete/1
```

위와 같은 방식은 REST 를 제대로 적용하지 않은 URI 입니다. URI 는 자원을 표현하는데 중점을 두어야 합니다. delete 와 같은 행위에 대한 표현이 들어가서는 안됩니다.

2) 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE 등)로 표현

위의 잘못 된 URI 를 HTTP Method 를 통해 수정해 보면

```
DELETE /members/1
```

으로 수정할 수 있겠습니다.
회원정보를 가져올 때는 GET, 회원 추가 시의 행위를 표현하고자 할 때는 POST METHOD 를 사용하여 표현합니다.

**회원정보를 가져오는 URI**

```
GET /members/show/1        (x)
GET /members/1             (o)
```

**회원을 추가할 때**

```
GET /members/insert/2 (x)   - GET 메서드는 리소스 생성에 맞지 않습니다.
POST /members/2            (o)
```

**[참고]HTTP METHOD 의 알맞은 역할**

POST, GET, PUT, DELETE 이 4 가지의 Method 를 가지고 CRUD 를 할 수 있습니다.

| METHOD | 역할 |
| --- | --- |
| POST | POST를 통해 해당 URI를 요청하면 리소스를 생성합니다. |
| GET | GET를 통해 해당 리소스를 조회합니다. 리소스를 조회하고 해당 도큐먼트에 대한 자세한 정보를 |
| PUT | PUT를 통해 해당 리소스를 수정합니다. |
| DELETE | DELETE를 통해 리소스를 삭제합니다. |

다음과 같은 식으로 URI 는 자원을 표현하는 데에 집중하고 행위에 대한 정의는 HTTP METHOD 를 통해 하는 것이 REST 한 API 를 설계하는 중심 규칙입니다.

4-2. URI 설계 시 주의할 점

1) 슬래시 구분자(/)는 계층 관계를 나타내는 데 사용

```
http://restapi.example.com/houses/apartments
http://restapi.example.com/animals/mammals/whales
```

2) URI 마지막 문자로 슬래시(/)를 포함하지 않는다.

URI 에 포함되는 모든 글자는 리소스의 유일한 식별자로 사용되어야 하며 URI 가 다르다는 것은 리소스가 다르다는 것이고, 역으로 리소스가 다르면 URI 도 달라져야 합니다. REST API 는 분명한 URI 를 만들어 통신을 해야 하기 때문에 혼동을 주지 않도록 URI 경로의 마지막에는 슬래시(/)를 사용하지 않습니다.

```
http://restapi.example.com/houses/apartments/ (X)
http://restapi.example.com/houses/apartments  (0)
```

3) 하이픈(-)은 URI 가독성을 높이는데 사용

URI 를 쉽게 읽고 해석하기 위해, 불가피하게 긴 URI 경로를 사용하게 된다면 하이픈을 사용해 가독성을 높일 수 있습니다.

4) 밑줄(_)은 URI에 사용하지 않는다.

글꼴에 따라 다르긴 하지만 밑줄은 보기 어렵거나 밑줄 때문에 문자가 가려지기도 합니다. 이런 문제를 피하기 위해 밑줄 대신 하이픈(-)을 사용하는 것이 좋습니다.(가독성)

5) URI 경로에는 소문자가 적합하다.

URI 경로에 대문자 사용은 피하도록 해야 합니다. 대소문자에 따라 다른 리소스로 인식하게 되기 때문입니다. RFC 3986(URI 문법 형식)은 URI 스키마와 호스트를 제외하고는 대소문자를 구별하도록 규정하기 때문이지요.

```
RFC 3986 is the URI (Unified Resource Identifier) Syntax document
```

6) 파일 확장자는 URI에 포함시키지 않는다.

```
http://restapi.example.com/members/soccer/345/photo.jpg (X)
```

REST API 에서는 메시지 바디 내용의 포맷을 나타내기 위한 파일 확장자를 URI 안에 포함시키지 않습니다. Accept header 를 사용하도록 합시다.

```
GET / members/soccer/345/photo HTTP/1.1 Host: restapi.example.com Accept: image/jpg
```

4-3. 리소스 간의 관계를 표현하는 방법

REST 리소스 간에는 연관 관계가 있을 수 있고, 이런 경우 다음과 같은 표현방법으로 사용합니다.

/리소스명/리소스 ID/관계가 있는 다른 리소스명

ex) GET : /users/{userid}/devices (일반적으로 소유 'has'의 관계를 표현할 때)

만약에 관계명이 복잡하다면 이를 서브 리소스에 명시적으로 표현하는 방법이 있습니다. 예를 들어 사용자가 '좋아하는' 디바이스 목록을 표현해야 할 경우 다음과 같은 형태로 사용될 수 있습니다.

GET : /users/{userid}/likes/devices (관계명이 애매하거나 구체적 표현이 필요할 때)

4-4. 자원을 표현하는 Colllection 과 Document

Collection 과 Document 에 대해 알면 URI 설계가 한 층 더 쉬워집니다. DOCUMENT 는 단순히 문서로 이해해도 되고, 한 객체라고 이해하셔도 될 것 같습니다. 컬렉션은 문서들의 집합, 객체들의 집합이라고 생각하시면 이해하시는데 좀더 편하실 것 같습니다. 컬렉션과 도큐먼트는 모두 리소스라고 표현할 수 있으며 URI 에 표현됩니다. 예를 살펴보도록 하겠습니다.

http:// restapi.example.com/sports/soccer

위 URI 를 보시면 sports 라는 컬렉션과 soccer 라는 도큐먼트로 표현되고 있다고 생각하면 됩니다. 좀 더 예를 들어보자면

http:// restapi.example.com/sports/soccer/players/13

sports, players 컬렉션과 soccer, 13(13 번인 선수)를 의미하는 도큐먼트로 URI 가 이루어지게 됩니다. 여기서 중요한 점은 **컬렉션은 복수**로 사용하고 있다는 점입니다. 좀 더 직관적인 REST API 를 위해서는 컬렉션과 도큐먼트를 사용할 때 단수 복수도 지켜준다면 좀 더 이해하기 쉬운 URI 를 설계할 수 있습니다.

**5. HTTP 응답 상태 코드**

---

마지막으로 응답 상태코드를 간단히 살펴보도록 하겠습니다. 잘 설계된 REST API 는 URI 만 잘 설계된 것이 아닌 그 리소스에 대한 응답을 잘 내어주는 것까지 포함되어야 합니다. 정확한 응답의 상태코드만으로도 많은 정보를 전달할 수가 있기 때문에 응답의 상태코드 값을 명확히 돌려주는 것은 생각보다 중요한 일이 될 수도 있습니다. 혹시 200 이나 4XX 관련 특정 코드 정도만 사용하고 있다면 처리 상태에 대한 좀 더 명확한 상태코드 값을 사용할 수 있기를 권장하는 바입니다.
상태코드에 대해서는 몇 가지만 정리하도록 하겠습니다.

| 상태 코드 | |
|---|---|
| 200 | 클라이언트의 요청을 정상적으로 수행함 |
| 201 | 클라이언트가 어떠한 리소스 생성을 요청, 해당 리소스가 성공적으로 생성됨(POST를 통한 리소스 생성 작업 시) |

| 상태 코드 | |
|---|---|
| 400 | 클라이언트의 요청이 부적절 할 경우 사용하는 응답 코드 |
| 401 | 클라이언트가 인증되지 않은 상태에서 보호된 리소스를 요청했을 때 사용하는 응답 코드 |
| | (로그인 하지 않은 유저가 로그인 했을 때, 요청 가능한 리소스를 요청했을 때) |
| 403 | 유저 인증상태와 관계 없이 응답하고 싶지 않은 리소스를 클라이언트가 요청했을 때 사용하는 응답 코드 |
| | (403 보다는 400이나 404를 사용할 것을 권고. 403 자체가 리소스가 존재한다는 뜻이기 때문에) |
| 405 | 클라이언트가 요청한 리소스에서는 사용 불가능한 Method를 이용했을 경우 사용하는 응답 코드 |

| 상태코드 | |
|---|---|
| 301 | 클라이언트가 요청한 리소스에 대한 URI가 변경 되었을 때 사용하는 응답 코드 |
| | (응답 시 Location header에 변경된 URI를 적어 줘야 합니다.) |
| 500 | 서버에 문제가 있을 경우 사용하는 응답 코드 |

## 6. 쿠버네티스 기반의 마이크로서비스 구축 관련 아키텍처 프로세스

쿠버네티스와 컨테이너 기술은 마이크로서비스아키텍처(MSA) 환경을 구현하기에 가장 적합한 기술로 지목되며 최근 각광을 받고 있다. 비즈니스 민첩성이 점점 더 중요해지는 환경에서 컨테이너와 쿠버네티스는 하이브리드 클라우드, 나아가 멀티 클라우드 서비스를 모두 사용하길 원하는 기업들의 요구를 충족하는 대안으로도 떠올랐다.

프라이빗 클라우드 환경에서 컨테이너 애플리케이션 서비스를 구현하고 운영, 관리할 수 있는 오케스트레이션 플랫폼인 쿠버네티스를 구축하고자 할 경우, 인프라 환경을 고려할 수밖에 없다. 하지만 다양한 오픈소스•상용 컨테이너와 쿠버네티스 플랫폼, 그리고 이를 구동할 x86 서버 기반 베어메탈 시스템, 하이퍼컨버지드인프라스트럭처(HCI), 가상화(VM), 네트워크, 스토리지까지 고려해야 할 영역이 방대하고 수많은 기술이 혼재하는 상황에서 기업 환경과 요구에 맞는 방식을 선택하고 구성하는 것이 그리 간단한 일은 아니다.

그러다보니 기업에서 사용하는 애플리케이션들을 컨테이너화해 사용하는데 그치는 경우도 있고, 서비스 계층에만 초점을 두고 프로젝트를 진행하다 네트워크 인프라 계층에서 예기치 않는 문제가 발생하는 일도 생겨나고 있다. 인프라 구성이 복잡해지고 높은 성능과 가용성이 뒷받침되지 않아 장애가 발생하거나, 가시성이 확보돼 있지 않아 뒤늦게 인프라를 변경하는 경우다.

현업, 개발조직의 요구에 따라 애플리케이션을 빠르고 유연하게 개발하고 배포할 수 있는 CICD(Continuous Integration and Continuous Delivery) 환경을 구축하는 것에만 신경쓰다보면 오히려 전체 플랫폼을 구축하는데 더 오랜 시간이 걸릴 수 있다. 실제 컨테이너와 쿠버네티스 플랫폼을 구축하고 있지만 MSA를 구현하지 못하고 있는 사례가 많은 이유이기도 하다.

이같은 상황은 기업 내 조직 운영 방식과 연관돼 있다. 서비스 개발부터 배포하고 구동하는 과정에 개발자, 플랫폼 담당자, 인프라 운영자 조직이 구분돼 있고 성과와 목표가 저마다 다르기 때문이다.

시스코코리아의 데이터센터 테크니컬 솔루션 전문가는 "비즈니스 민첩성이 매우 중요해지면서 컨테이너와 쿠버네티스로 MSA를 구축하려는 움직임이 활발하다"라면서 "컨테이너는 기업의 웹, 웹 애플리케이션 서버(WAS), 데이터베이스(DB)로 구성되는 IT 서비스 인프라가 한 덩어리로 구현돼 움직이는 모놀로틱(monolithic) 구조를 탈피해 민첩성을 확보하고 종속성을 제거할 수 있게 해 MSA 트렌드에 적합하다"고 말했다.

이 전문가는 "MSA를 구축하기 위해서는 기술뿐 아니라 조직의 변화까지 이뤄져야 한다. 그리고 서비스 계층만이 아니라 인프라 계층까지 아우르는 통찰력을 바탕으로 설계해야 하지만 쉬운 일

이 아니다"라면서 "MSA를 구현하기 위한 플랫폼과 이를 지원하는 최적의 인프라를 엔드투엔드(end to end)로 제공하는 기업은 시스코밖에 없다"고 강조했다.

네트워크, 컴퓨팅, 스토리지 영역을 아우르는 시스코 SDx 플랫폼

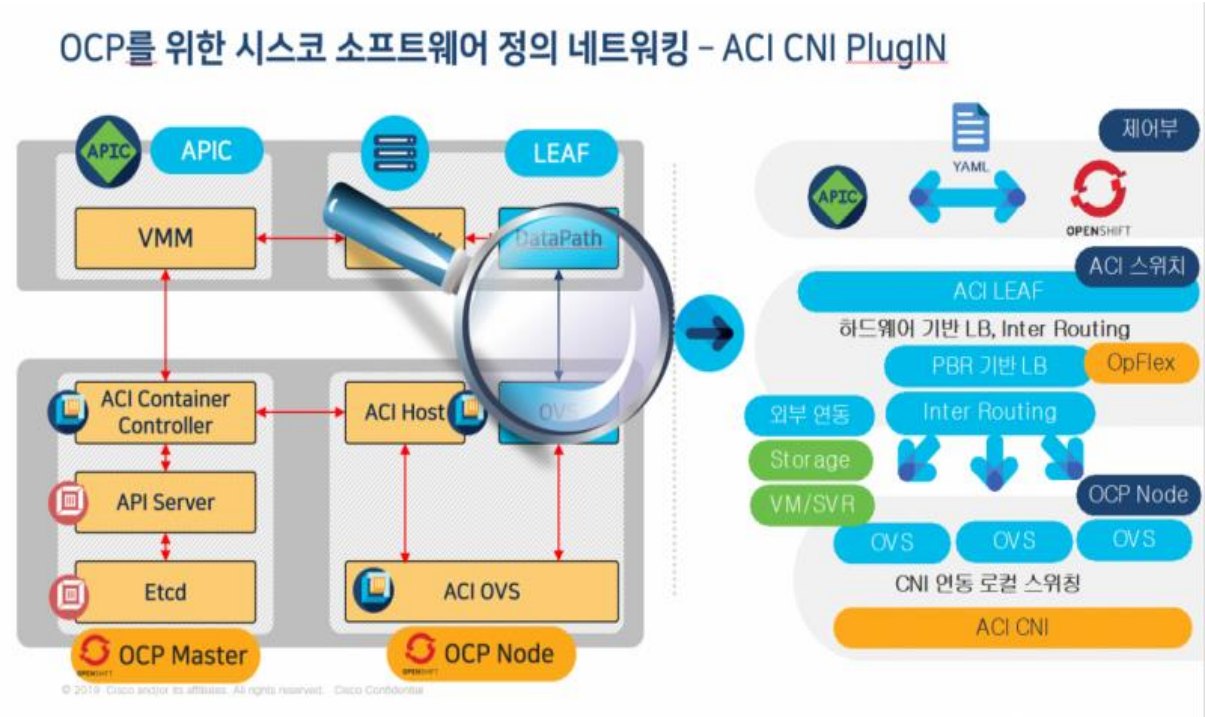
컨테이너와 쿠버네티스 플랫폼을 구축하려면 소프트웨어정의(SDx) 방식의 컴퓨팅, 네트워크, 스토리지 인프라가 필수다. 시스코는 기업이 원하는 컨테이너 애플리케이션 서비스에 최적화된 SDx 인프라를 모두 지원하고 있다.

소프트웨어정의네트워킹(SDN)은 시스코 ACI(Application-Centric Infrastructure), 소프트웨어정의컴퓨팅(SDC) 구현 플랫폼은 UCS(Unified Computing System), 소프트웨어정의스토리지(SDS) 플랫폼이 탑재돼 있는 HCI는 하이퍼플렉스(HyperFlex)를 활용한다.

이들 시스코 플랫폼은 컨테이너 네트워크 인터페이스(CNI), 컨테이너 스토리지 인터페이스(CSI)를 지원해 컨테이너 애플리케이션 서비스 인프라에서 요구되는 손쉬운 구성과 운영, 가시성을 충족한다. CNI, CSI는 리눅스재단의 클라우드네이티브컴퓨팅재단(CNCF)에서 컨테이너, 쿠버네티스 환경을 손쉽게 구축하기 위해 만든 플러그인 표준규격이다.

시스코는 구글을 비롯해 레드햇, 피보탈 등이 제공하는 쿠버네티스, 오픈시프트 컨테이너 플랫폼, 클라우드 파운드리와 같은 다양한 오픈소스•상용 쿠버네티스 플랫폼을 모두 지원하고 있다. 시스코 컨테이너 플랫폼(CCP)도 제공한다.

컨테이너 네트워크 인터페이스(CNI)와 컨테이너 스토리지 인터페이스(CSI) 지원



CNI 플러그인을 지원하는 시스코 ACI는 컨테이너와 쿠버네티스 플랫폼의 제어부에서 수행해야 하는 컴퓨팅과 스토리지 영역, 쿠버네티스와 컨테이너 플랫폼과 각각의 노드 사이에서 필요한 동-서, 북-남 트래픽을 처리하는 모든 네트워크 기능을 관장한다.

컨테이너들로 구성된 포드(Pod) 간 가상 오버레이 네트워크(VxLAN) 터널링 기반 통신과 내외부 라우팅 경로를 지원하고, 컨테이너 서비스 환경에서 필요한 성능과 가시성을 확보할 수 있게 만든다. 컨테이너 애플리케이션 서비스에서 필요한 로드밸런싱도 하드웨어 스위치에서 정책 기반 라우팅(PBR) 로드밸런싱 기능을 제공한다. 컨테이너로 로드밸런싱을 수행하는 방식에 비해 성능에 미치는 영향을 최소화할 수 있다는 게 시스코의 설명이다. 아울러 기업이 요구하는 단일한 보안 정책도 적용할 수 있다.

핵심 역할은 ACI의 APIC(Application Policy Infrastructure Controller)이 수행한다. SDN 환경에서 컨트롤러가 모든 데이터 경로(네트워크 장비)를 제어하는 방식의 시스코 ACI의 구조가 ACI CNI를 통해 컨테이너, 쿠버네티스 플랫폼 안에서 그대로 구현된다. 모든 네트워킹 영역을 관장한다는 얘기다. 플랫폼의 마스터 노드에 API로 연동된 ACI 컨테이너 컨트롤러가 구동되고, 워커 노드에서 구현되는 오픈가상스위치(OVS)를 컨테이너 ACI OVS가 대신 처리한다. 이 노드에는 ACI 호스트가 OVS와 ACI OVS, 그리고 ACI 컨테이너 컨트롤러와 통신한다.

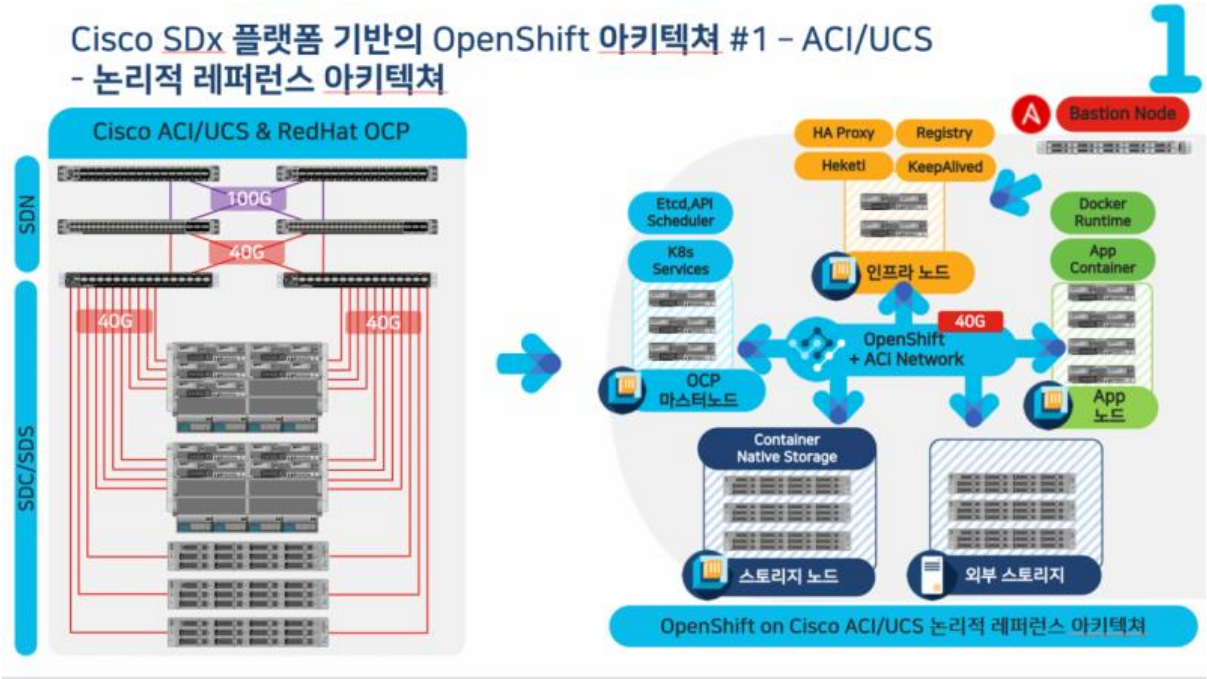
프로세스로 동작하는 컨테이너 서비스 운영 환경에서는 컨테이너의 특성상 데이터 연속성을 가져가기가 힘들다. 이로 인해 지속적인 볼륨(Persistent Volume, PV)을 운영할 수 있는 환경을 만들 수 있도록 CSI가 사용되고 있다. 스토리지 컨트롤러를 컨테이너로 플랫폼 노드에 탑재하는 방식으로 구현된다. 이에 따라 사용하고 있는 네임스페이스(NameSpace)나 클러스터 단위로 현업 담당자들이 필요시 중복제거, 압축 등의 데이터 관리를 유연하게 수행할 수 있다.

시스코는 UCS와 하이퍼플렉스에서 SDS 구성을 최적화할 수 있도록 지원하고 있다. 시스코 하이퍼플렉스는 PV 스토리지를 위한 CSI 플러그인을 지원한다. CSI가 등장하기 전에 구글과 협력해 개발한 플렉스볼륨(Flexvolume) 스토리지 인터페이스도 제공한다.

ACI+UCS, ACI+하이퍼플렉스 기반 레드햇 오픈시프트 아키텍처

시스코는 SDx 플랫폼과 오픈시프트 컨테이너 플랫폼(이하 OCP)을 기반으로 검증된 아키텍처를 제공하고 있다.

먼저 시스코 넥서스 스위치 기반의 ACI와 UCS를 기반으로 SDN과 SDC, SDS를 구현할 수 있다. 기존의 데이터센터 인프라와 비슷하지만 필요에 따라 프로그래밍 기능을 구현할 수 있는 아키텍처다. UCS는 베어메탈과 가상화(리눅스 KVM) 환경을 모두 지원한다. 그 위에 OCP가 탑재된다. 스토리지는 UCS에 레드햇 Ceph 스토리지를 비롯해 다양한 상용 스토리지와 셀프 스토리지를 구현할 수 있다.




그림을 보면, OCP의 마스터노드가 세 대의 블레이드 서버에 분산돼 고가용성(HA) 클러스터로 구성돼 있다. 인프라 노드는 블레이드 서버 두 대에 고가용성 프록시, 레지스트리 등이 탑재돼 있다. 애플리케이션 컨테이너가 구동되는 노드가 네 대의 블레이드 서버로, 그 아래 컨테이너 스토리지 노드까지 블레이드 서버에 구현돼 있다. 외부 물리적 스토리지는 OCP와 ACI CNI 플러그인이 연동돼 연결을 제어한다.

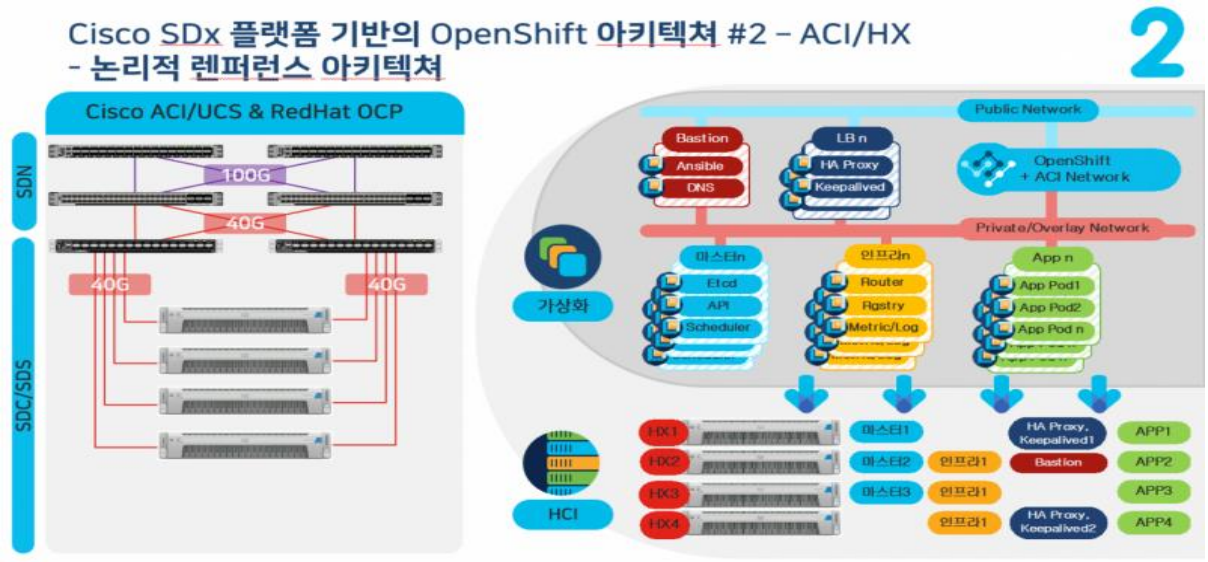
시스코 ACI APIC이 전체 네트워크 영역을 제어하는 역할을 수행한다. UCS 매니저는 클러스터 형태로 구성되는 컴퓨팅과 스토리지 영역을 한꺼번에 제어할 수 있다. APIC과 UCS 매니저가 지원하는 REST API로 앤서블과 연동해 플러그인 설치와 구성까지 자동화된 워크플로우를 구현할 수 있다.

시스코는 네트워크와 컴퓨팅 물리적 노드를 직접 앤서블을 활용해 제어할 수 있는 방법을 권고한다. 서비스 플랫폼이 점점 커져 네트워크 스위치와 컴퓨팅 노드 역시 수십대씩 늘어날 경우 오히려 하드웨어가 서비스 확장 속도를 따라가지 못하는 경우가 있다는 이유에서다. APIC과 UCS 컨트롤러에서 앤서블 타워와 연동하게 되면 보다 간편하고 빠르게 정책 기반 자동화 환경을 운영할 수 있다.

시스코 ACI와 하이퍼플렉스로 구성하는 방식은 100% SDx를 구현한다. 컴퓨팅과 스토리지, 가상화(VM)가 통합된 HCI를 활용해 인프라 구조는 좀 더 단순하다.

기업 환경에서는 가상화, 특히 VM웨어 환경에서 컨테이너 서비스를 구현하는 경우가 실제로 많다. 그동안 VM을 많이 활용해 왔다는 이유도 있지만 베어메탈 환경에 비해 필요한 VM 노드를 빠르게 설치할 수 있고 드라이버도 손쉽게 적용할 수 있다는 장점을 제공한다는 게 주된 이유로

꼽힌다.




ACI와 하이퍼플렉스 기반으로 OCP를 구성한 아키텍처다. 컴퓨팅과 스토리지단은 4개의 하이퍼플렉스 노드로 구성하고 네트워킹도 단순화해 40Gbps로 연결돼 있다. 네트워킹은 ACI 기반 SDN이 구성돼 있다. 마스터, 인프라, 애플리케이션 노드들은 VM이다. 관리자들은 서비스 확장이 필요할 경우 손쉽게 필요한 CPU와 메모리를 할당해 자원을 늘려나가면 된다.

APIC과 하이퍼플렉스 컨트롤러에서 지원하는 REST API로 앤서블과 연동해 물리적 인프라 설치부터 플러그인 연동, 가상화 환경의 OCP 프로비저닝, MSA 정책까지 자동화된 방식으로 구성할 수 있다.

## 7. 쿠버네티스 기반의 마이크로서비스 구축 관련 아키텍처 설계 및 품질속성

**모놀리식에서 마이크로서비스 전환**
모놀리식 애플리케이션을 수평분리(데이터 / 로직 / 프레젠테이션)하면, 여러 기술 계층으로 나뉘고 세로로 분리하면 여러 비즈니스 도메인으로 나뉜다. 수평 각 계층은 상위 계층에 서비스를 제공한다. 데이터 계층을 상태를 저장하고, 로직 계층은 유용한 작업을 수행하고 프레젠테이션 계층은 사용자에게 결과를 반환합니다.

모놀리식 애플리케이션에서는 아키텍처가 애플리케이션 자체의 경계로 제한된다. 반면에 마이크로서비스 애플리케이션에서는 크기와 범위 측면에서 계속해서 진화하는 것에 대해 계획을 세운다. **마이크로서비스는 구성요소 자체뿐만 아니라 연결하는 방법, 위치, 그리고 어떻게 동시에 구축할지 등의 방법까지도 고려하는 것이 중요합니다.** 계획이 전체 애플리케이션 상에서 어떤 구조를 강요하거나 강제하기보다는 좋은 가이드라인을 따라 성장할 수 있게 북돋우기를 바랄 것입니다.

마이크로서비스는 격리되어 실행되지 않습니다. 각 마이크로서비스는 마이크로서비스를 다른 마이크로서비스와 함께 개발/배포/실행할 수 있는 환경에 존재합니다. 애플리케이션 아키텍처는 전체 환경을 포괄해야 합니다.

아키텍트의 역할은 설계를 강제하기보다는 진화를 가능하게 하는 것이다. 아키텍트는 애플리케이션의 기술적 근간이 빠른 속도와 유연성을 제공하도록 해야 한다네요. 아키텍트는 포괄적 시각을 가지고 애플리케이션의 요건이 충족되게하고 발전시켜 가이드해야합니다. 이를 달성하기 위해 아키텍트는 2 가지 방식으로 개발을 가이드해야 합니다..
- **원칙:** 고도의 기술적 목표나 조직의 목표를 달성하기 위해 팀이 따라야 하는 가이드라인
- **개념 모델:** 시스템의 관계를 보여주는 고수준의 모델과 애플리케이션 수준의 패턴

원칙은 유연합니다. 이는 비즈니스의 우선순위와 애플리케이션의 기술적 발전을 반영해 변경될 수 있어야 합니다. 예를 들어 개발 초기에는 제품의 시장 검증에 우선순위를 두지만, 애플리케이션이 성숙하면 성능과 확장성에 중점을 두어야 합니다.

-소프트웨어 아키텍처 설계

전체적인 소프트웨어 구조인 framework는 spring boot(스프링)를 사용하였으며, 주요기능은 gis위치기능과 RESTful api를 사용하였습니다.

그리고 spring mvc3도 조금 사용해서 개발하였습니다.

**가. 관점별 소프트웨어 아키텍처의 품질 속성 관점**

- 시스템

가용성 : 시스템이 필요할 때 작동하게될 가능성의 정도

변경가능성 : 시스템을 변경하는데 소요되는 시간과 노력 등의 비용과 관련

성능 : 시스템이 얼마나 빠르고 효율적으로 원하는 기능을 수행 하는가 하는 문제

보안 : 권한이 없는 사용자에 대해서는 서비스를 거부

테스트 용이성 : 테스트를 통해 소프트웨어가 결함(Fault)을 가지고 있음 을 얼마나 쉽게 증명할 수 있는가?

사용성 : 사용자가 얼마나 손쉽게 원하는 작업을 할 수 있는가와 관련

- 비즈니스

적시성 : 일정을 변경할 수 없거나, 시장의 주도권을 잡기 위해 경쟁사의 제품보다 먼저 출시

비용과 예산 : 일반적으로 유연성이 높은 아키텍처는 그렇지 않은 아키텍처보다 더 많은 예산 소요

내구연한 : 내구연한을 위한 변경 가능성,확장성,이식성 등의 시스템 품질 속성이 중요해짐

목표시장 : 범용성을 갖는 소프트웨어라면 시스템이 실행되는 플랫 폼도 잠재적인 시장의 크기를 결정하는 요소

첫공개일정 : 첫 버전에서는 기본적인 기능만 제공하고, 다른 기능은 이후 버전에서 릴리즈

기존시스템연동 : 적절한 통합 메커니즘을 정의하는데 제약 사항으로 작용

- 아키텍처

개념적무결성 : 모든 레벨에서 시스템의 설계를 단일화 하는 주제나 비전을 의미

정확성과완료성 : 아키텍처가 시스템의 모든 요구사항과 런타임

## 나. 유틸리티 트리

### Utility Tree

소프트웨어 아키텍처 등 품질을 기반으로 평가하는 모델에서 품질 특성을 기준으로 시나리오를 작성하는 분석법, 또는 그 구조이며 다른 말로 하면 즉, SA수립을 위해 다양한 관점의 요구사항으로부터 도출되는 품질 속성을 시나리오 기반의 품질 우선순위를 가시화 하여 도출하는 명세기법