초보를 위한 쿠버네티스 안내서

쿠버네티스 시작하기

컨테이너 오케스트레이션

서버를 관리한다는 것

서버의 상태를 관리하기 위한 노력

> **⇒ 문서화를 잘 해보자!**
> (A-Z까지 화면 하나 하나 캡쳐)

> **⇒ 도구 사용** CHEF, puppet labs, ANSIBLE
> (스크립트 직접 입력할 필요 X, but 설치된 것과 충돌날 수 있음 결국 난이도 高)

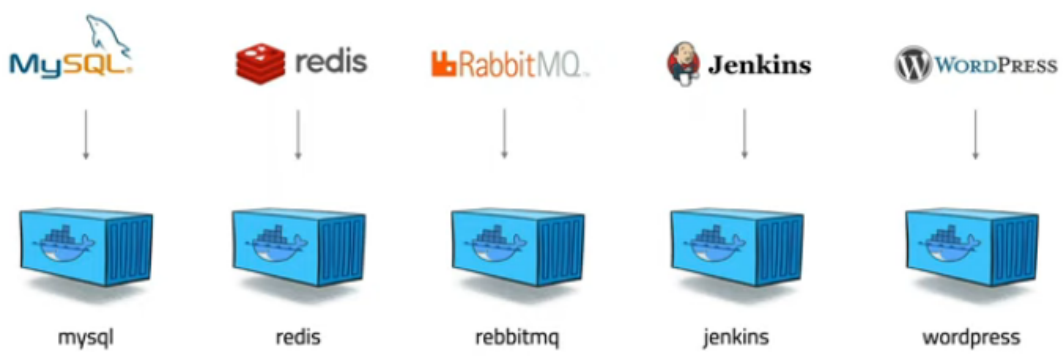
> **⇒ 가상머신 사용**
> (문제, 클라우드에 안맞고 특정 벤더에 dependency가 생김, 멀티 클라우드에 부적합, 느림)

> **⇒ 도커 컨테이너**
> (어디서든 동작하고 쉽고 효율적)

도커의 등장

컨테이너의 특징

- 가상머신과 비교하여 컨테이너 생성이 쉽고 효율적
- 컨테이너 이미지를 이용한 배포와 롤백이 간단
- 언어나 프레임워크에 상관없이 애플리케이션을 동일한 방식으로 관리
- 개발, 테스팅, 운영환경은 물론 로컬 PC와 클라우드까지 동일한 환경을 구축
- 특정 클라우드 벤더에 종속적이지 않음



`containerization` : 프로그램을 컨테이너화 해서 사용하는것

Dockerfile —(Build)→ Image —(Create)→ Container

개발자 (코드작성) → Build → Ship → Run

모든 앱을 컨테이너로 만들기 시작. app 많아지고 늘어나기 시작. 결국 수천개의 컨테이너를 관리하려면?

도커 그 이후

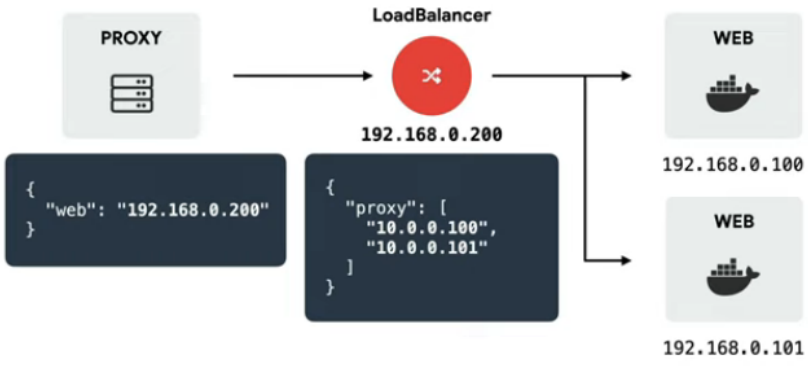
1. **배포는 어떻게 할까?**

문제1) 하나하나 ssh 접속하기 힘들다. 하나하나 IP 관리하고 실행할때도 하나하나 접속하기 힘들다.

문제2) 도커를 많이 사용하면 빈공간이 생긴다. 빈공간은 서버가 여러개 있는데 컨테이너가 실행이 안되어 있는 서버다. 새로운 컨테이너를 실행하려면 빈서버에 실행하는게 좋은데 어떤 서버가 여유가 있는지 보려면 모니터링 도구를 만들어야 하거나 하나하나 접속해봐야한다.

문제3) 컨테이너를 많이 사용하면서 모두 v1 → 배포를 해서 일부를 v2로 → 전체를 v2로 → 문제가 생겨서 v1으로 롤백해야하는데 손이 너무 많이감. 중앙에서 모든 컨테이너를 버전업하고 싶은 욕구가 생김

2. **서비스 검색은 어떻게 할까?**



동작은 잘한다. 근데 하다보면 굉장히 많아짐. 마이크로서비스가 유행하면서 내부서비스간 통신이 많아지는데 그럴때마다 로드밸런서 설치해주고 IP 변경시마다 업데이트하는 작업이 필요함 (서비스 검색)

3. **서비스 노출은 어떻게 할까?**

Public영역에 nginx와 같은 proxy서버를 하나 두고 거기에 들어오는 host가 뭐냐에 따라서 내부의 private의 컨테이너를 연결

관리하다보면 귀찮음. 내부적으로 어떤 서버를 띄우면 요청이 옴. 도메인 등록했으니 컨테이너 연결해주세요! nginx 설정 바꾸면 되긴 하는데 귀찮음

4. **서비스 이상, 부하 모니터링은 어떻게 할까?**

총 12개의 컨테이너를 관리하고 있는데 5개의 컨테이너가 갑자기 죽으면? 접속해서 컨테이너 로그 보고 다시 띄우고 그래야하는데 자동화되거나 쉬운 방법을 찾아야함. 갑자기 부하를 많이 받으면? 응답 속도가 느려지니 개수를 늘려야하는데 이것도 자동화를 해야..!

`컨테이너 오케스트레이션(Container Orchestraion)`

복잡한 컨테이너 환경을 **효과적으로 관리**하기 위한 도구

서버 관리자가 하는 일들을 대신해주는 프로그램을 만든다.

- **CLUSTER**
  - 중앙제어 (master-node) : master → cluster(containers …)
    노드를 하나하나 관리하는게 아니라 클러스터 단위로 추상화해서 관리
    클러스터 하나하나 노드에 ssh 접속하기 어렵기 때문에 master서버를 하나 두고 관리자는 마스터 서버에 명령어를 던지면 알아서 해줌
  - 네트워킹 : 클러스터 내의 노드들끼리는 네트워크 통신이 잘 되어야한다. 설정을 가상네트워크라든가 잘 연결해서 서버들끼리 통신에 문제가 없도록 설정하는것이 필요
  - 노드 스케일 : 노드의 개수가 수천개가 되더라도 잘 돌아야한다. 처음 설계할때 잘 설계해야 부하를 감당할 수 있음
- **STATE** 상태 관리:

```
{
  image: "app1"
  replicas: 2 # 2 -> 3
}
```

자동으로 컨테이너 3개로 띄우도록 해줌 (직접 조치하지 않아도 컨테이너 오케스트레이션이 알아서 해줌) 만약 하나에 문제 생기면 죽이고 다시 3개가 되도록 해줌

- **SCHEDULING** 배포 관리 :

  서버의 여유공간을 보고 어디에 새로운 앱을 어떤 서버에 띄울지 스케쥴링 기능 하나 더 띄우고 싶은데 서버가 부족할 때 서버를 하나 더 띄우고 거기에 컨테이너를 띄우는 기능이 필요

- **ROLLOUT ROLLBACK** 배포 버전관리 :

  `{ version: “v1” }` —RollOut→ `{ version: “v2” }` —RollBack→ `{ version: “v1” }`

  이걸 하나하나 관리하는게 아니라 중앙에서 관리

- **SERVICE DISCOVERY** 서비스 등록 및 조회 :

  WEB 192.168.0.101 — **등록** —> 서버 (WEB IP들 등록됨) —**조회 or Watch**—> Proxy (1. 설정변경 2. 프로세스 재시작)
  proxy 서버는 저장소를 계속 관찰한다. 바뀔때마다 설정을 내부적으로 변경하고 재시작하면 관리자가 하나하나 IP를 바꿀 필요없이 프로그램이 자동으로 설정해줌

- **VOLUME** 볼륨 스토리지 :
  노드 3개에 각각 필요한 볼륨을 마운트 해야할 수 있음

  e.g.) node1 - NFS, node2 - AWS EBS, node3 - GCE PD

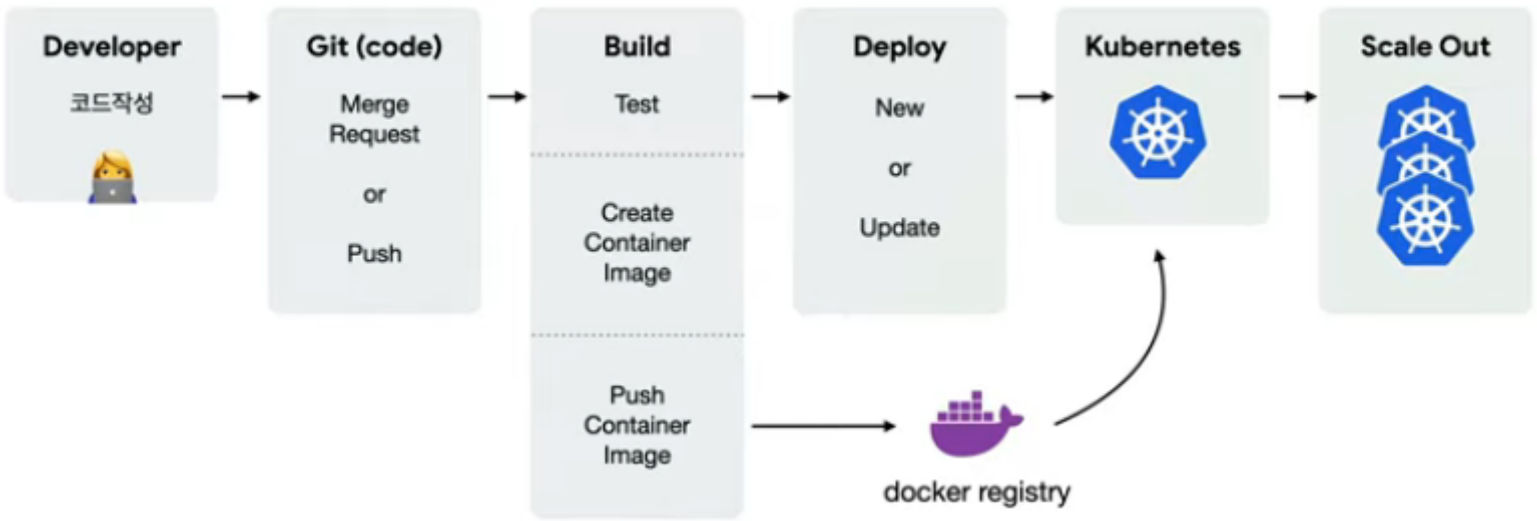
  당연히 손으로 이걸 연결할 수 있는데 추상적으로!

왜 쿠버네티스인가?

쿠버네티스 : 컨테이너를 쉽고 빠르게 배포/확장하고 관리를 자동화해주는 오픈소스 플랫폼

1. 오픈소스
2. 엄청난 인기
3. 무한한 확장성 (머신러닝, CI/CD, 서비스메시, 서버리스)
4. 사실상의 표준 (De facto: 데팍토)

어떤걸 배울까?



- 도커 컨테이너 실행하기
- 쿠버네티스에 컨테이너 배포하기

- 외부 접속 설정 하기
- 스케일 아웃 하기
- 그외 고급기능 소개

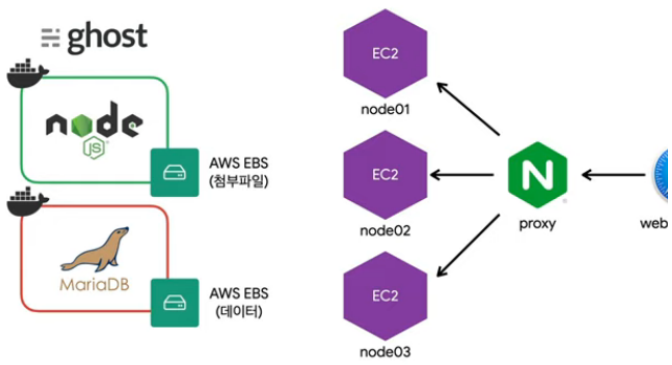
쿠버네티스 알아보기

쿠버네티스 소개

CloudNative : 클라우드 이전은 리소스를 한 땀 한 땀 직접 관리. 클라우드 이후 수많은 리소스를 자유롭게 사용하고 추상적으로 관리

클라우드 환경에서 어떻게 애플리케이션을 배포하는게 좋은걸까? ⇒ CloudNative(컨테이너, 서비스메시, 마이크로서비스, API, DevOps, …)

쿠버네티스 데모



AWS EBS 만들고 EC2 마운트하고 이거 귀찮음

서버들 마다 도커를 실행하고 하는거

프록시에서 포트 확인하고 ALB 설정 해줘야함

노드 서버 1번 DB서버 3일때 정확하게 알아야함 설정을 하나하나 바꿔줘야함

⇒ 쿠버네티스에서는 어떻게 처리할지?

쿠버네티스 아키텍처

1️⃣ 구성/설계

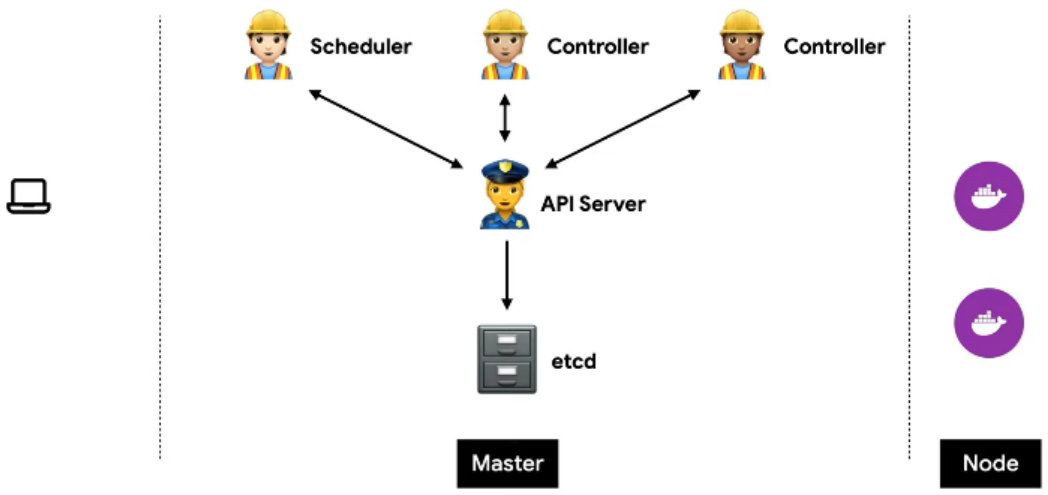
💻 컨테이너 생성해달라 요청하면 그걸 적어놓고 생성을 할 것이다. 방금 적어놓은 내용과 실제 컨테이너가 몇개인지 비교를 한다. 컨테이너 1개여야하는데 없으면 당연히 다시 만든다. 그러면 안정적인 상태

`Desired State` : 쿠버네티스도 내부적으로 이 방식을 사용

상태 체크 (현재 상태 == 원하는 상태) → **차이점 발견** (현재 상태 ≠ 원하는 상태) → **조치** (현재 상태 → 원하는 상태) → 반복

서버를 2대로 넉넉히 유지해달라 요청. 그럼 서버 2대 요청 적어놓고 추가할건데 컨테이너 요청하면 1개 떠있으니 거기에 하면 되는데 새 컨테이너는 어디에 배포? 따로 체크하는 사람이 있으면 좋겠다 싶어서 Scheduler를 만들어 그것만 체크해주는 사람 고용. 컨테이너 상태 체크할 사람도 뽑는 Controller 또 노드를 체크하는 Controller , ….

Replication Controller (복제만 잘되었는지 체크하는 컨트롤러), Endpoint Controller, Namespace Controller, Custom Controller, ML Controller, CI/CD Controller, …



🔎 Master 상세

- `etcd` : 모든 데이터를 확실하게 관리
  - 모든 상태와 데이터를 저장
  - 분산 시스템으로 구성하여 안정성을 높임 (고가용성)
  - 가볍고 빠르면서 정확하게 설계 (일관성)
  - Key-Value 형태로 데이터 저장
  - TTL, watch같은 부가 기능제공
  - 백업은 필수!

- `API Server` : 조회나 요청은 모두 API server를 통해서
  - 상태를 바꾸거나 조회
  - etcd와 유일하게 통신하는 모듈
  - REST API 형태로 제공
  - 권한을 체크하여 적절한 권한이 없을 경우 요청을 차단
  - 관리자 요청 뿐 아니라 다양한 내부 모듈과 통신
  - 수평으로 확장되도록 디자인

- `Schduler` : 어떤 노드에 어떤 컨테이너를?
  - 새로 생성된 Pod을 감지하고 실행할 노드를 선택
  - 노드의 현재 상태와 Pod의 요구사항을 체크
    - 노드에 라벨을 부여 e.g.) a-zone, b-zone 또는 gpu-enabled, …

- `Controller` : LOOP 돌고 무한반복
  - 논리적으로 다양한 컨트롤러가 존재
  - 끊임 없이 상태를 체크하고 원하는 상태를 유지
  - 복잡성을 낮추기 위해 하나의 프로세스로 실행

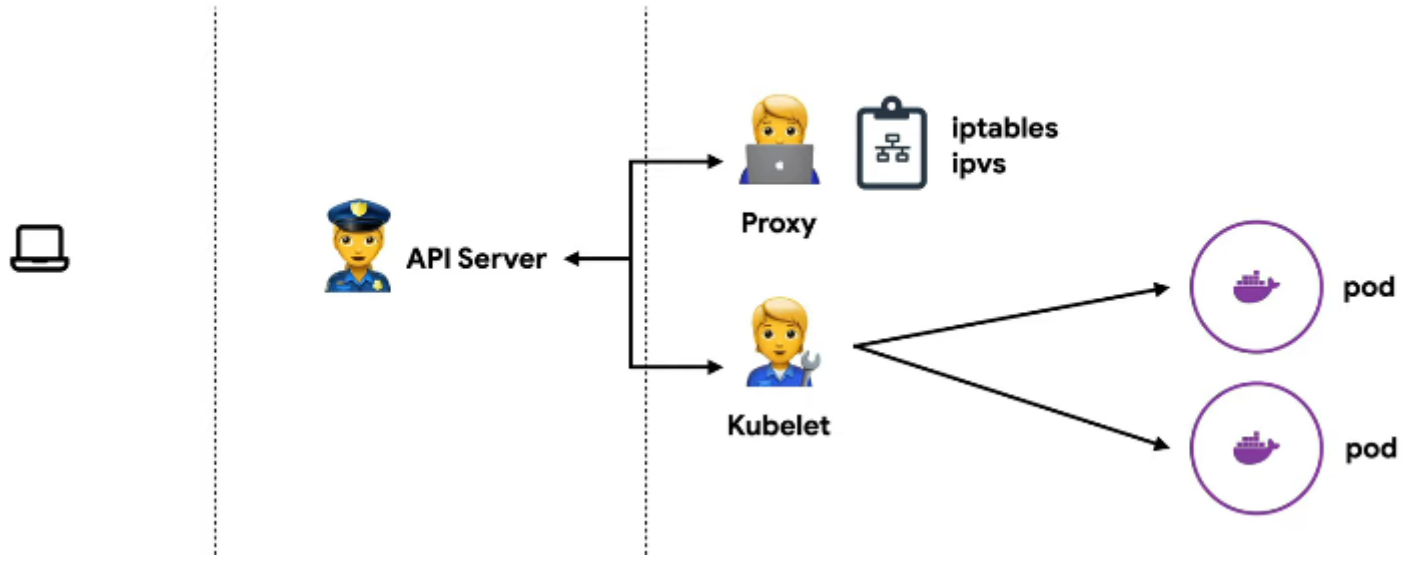
🔀 [조회 흐름]

`Controller` — 정보 조회 —> `API Server` — 정보 조회 권한 체크 —> `etcd`

`etcd` — 원하는 상태 변경—> `API Server` —> 원하는 상태 변경 —> `Controller` —> 원하는 상태로 리소스 변경

`Controller` —변경 사항 전달—> `API Server` —> 정보 갱신 권한 체크 —> etcd —> 정보 갱신

🔎 Node 상세

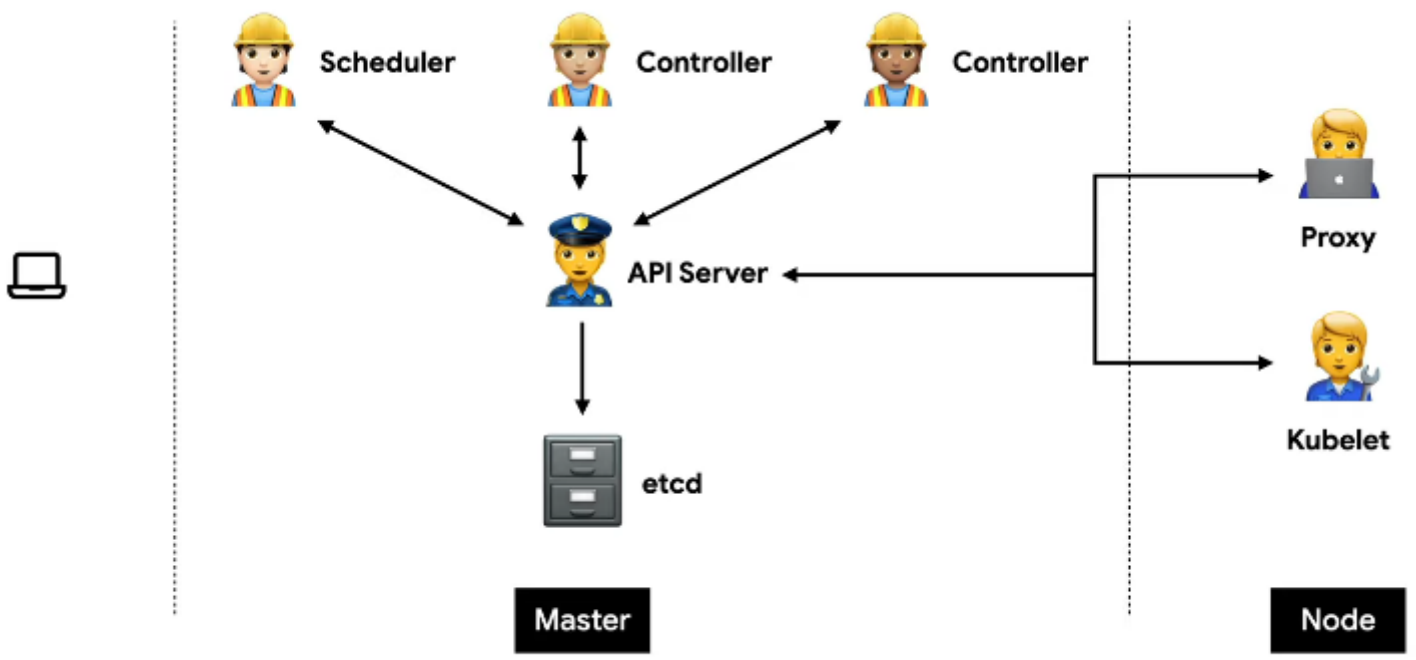


- `kubelet` : 컨테이너 관리 확실하게
  - 각 노드에서 실행
  - Pod을 실행/중지하고 상태를 체크 (컨테이너를 사용할 수 있도록 Pod로 감싸 사용)
  - CRI (Container Runtime Interface) : docker, Containerd, CRI-O

- `proxy` : 내/외부 통신 설정하기
  - 네트워크 프록시와 부하 분산 역할
  - 성능상의 이유로 별도의 프록시 프로그램 대신 iptables or IPVS를 사용

🔀 [쿠버네티스 흐름]

어떤 Pod이 하나 생성되기 위해 어떤 과정을 거치는지 알아보자!



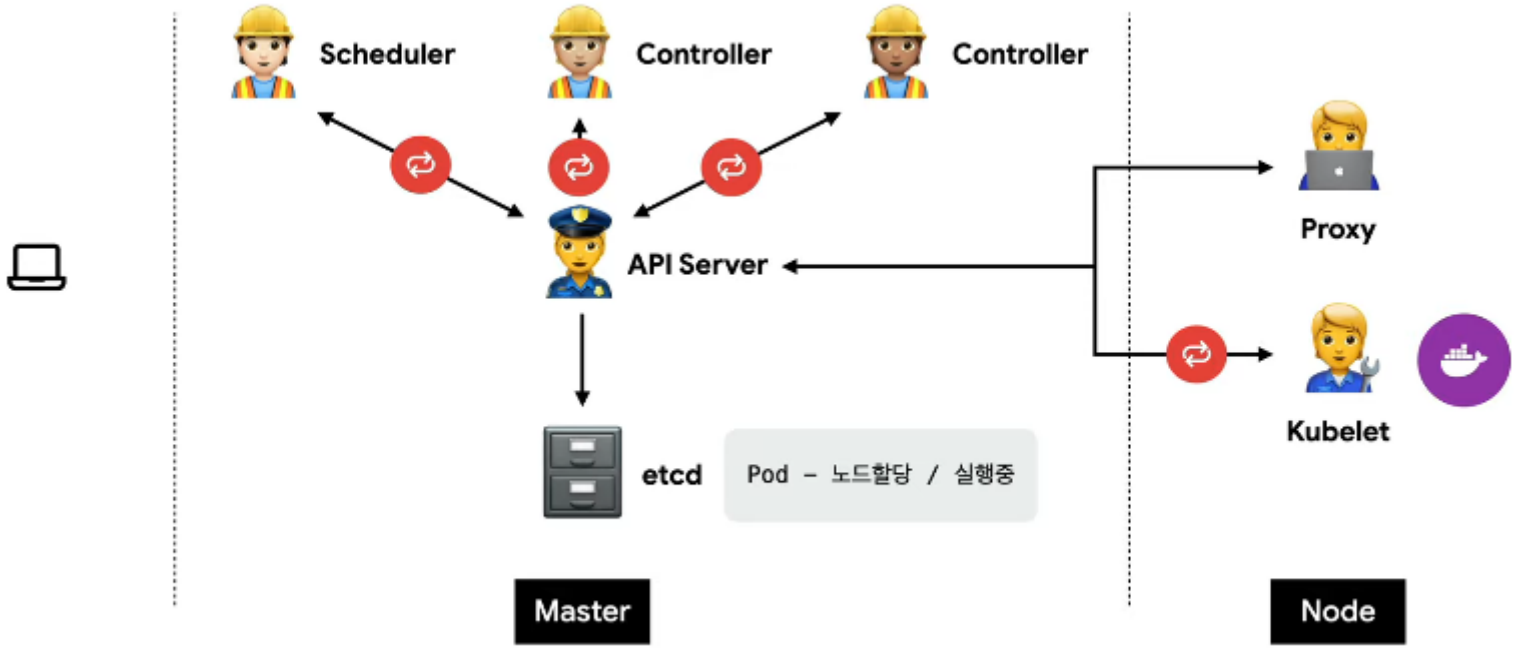
- `관리자` **—1. Pod 요청—>** `API Server`
- `API Server` **—2. Pod 요청—>** `etcd` (Pod - 생성 요청)
- `API Server` **<—3. 새 Pod 확인—>** `Controller`

  새로 생긴 Pod가 있는지 계속 체크하는데 지금 새 Pod 생성 요청을 발견
- `API Server` **<—4. Pod 할당—** `Controller`
- `API Server` **—5. Pod 할당 요청—>** `etcd` (Pod - 할당 요청)
- `API Server` **<—6. 새 Pod 할당 확인—>** `Scheduler`

  할당 요청을 받은 새로운 Pod가 있는지 계속 체크하는데 지금 새 할당이 들어온거임 어디에 할당할지 고민하다가~~
- `API Server` **—7. Pod 노드 할당—>** `Scheduler`

  특정 노드에 Pod을 할당
- `API Server` **—8. Pod 노드 할당—>** `etcd` (Pod - 노드 할당 / 미실행)
- `API Server` **<—9. 미실행 Pod 확인—>** `Kubelet`

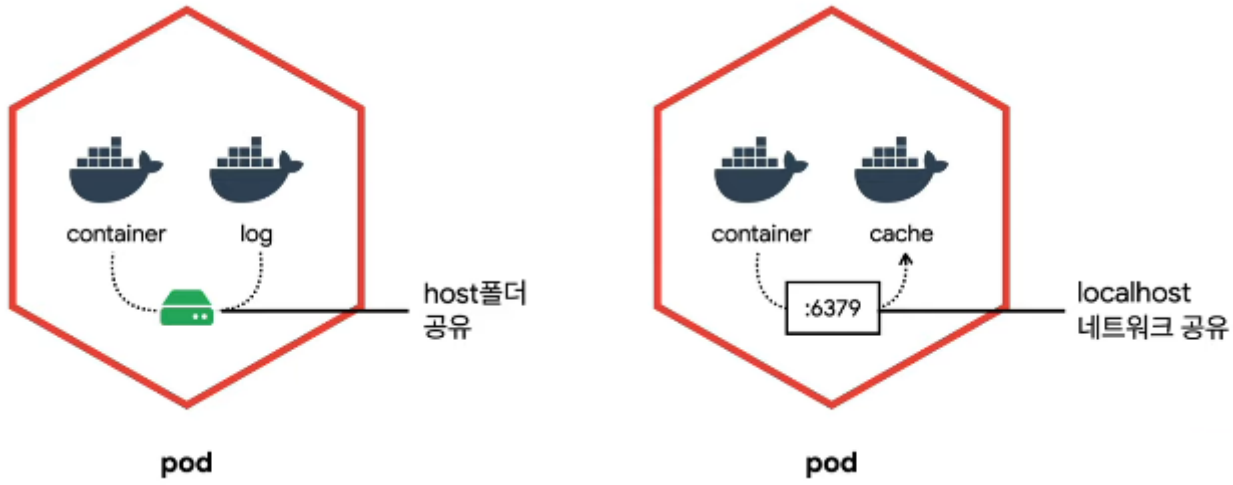
  내 노드에 할당된 Pod 중에서 실행이 안된 Pod있는 지 계속 체크하는데 지금 있음
- `API Server` **<—10. Pod 생성—** `Kubelet`
- `API Server` **—11. Pod 생성—>** `etcd` (Pod - 노드 할당 / 실행중)



2️⃣ 오브젝트

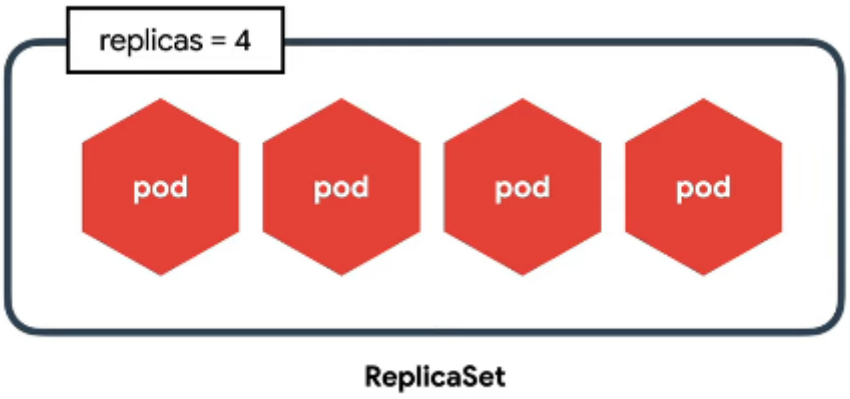
Pod

- 가장 작은 배포 단위
- 전체 클러스터에서 고유한 IP를 할당 (즉, 내부적으로 통신 가능)
- 여러개의 컨테이너가 하나의 Pod에 속할 수 있음
  보통 하나가 존재하지만 두개가 존재할 수도 있음
  예시1) 컨테이너 + 로그 수집 컨테이너 (2개)
  예시2) 컨테이너 + 캐시 (2개)



ReplicaSet

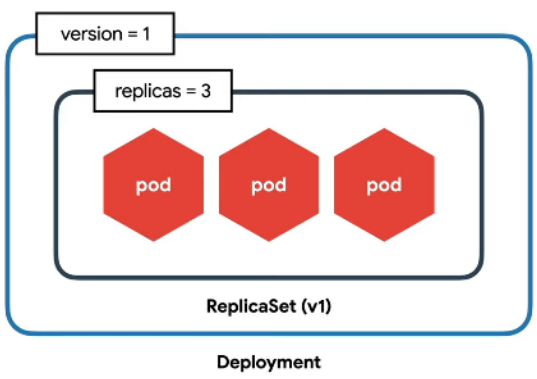
- 여러개의 Pod을 관리
- 새로운 Pod은 Template을 참고해서 생성
- 신규 Pod을 생성하거나 기존 Pod을 제거하여 원하는 수 (Replicas)를 유지

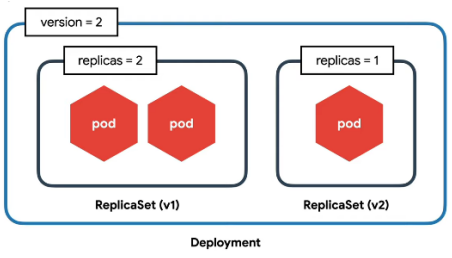


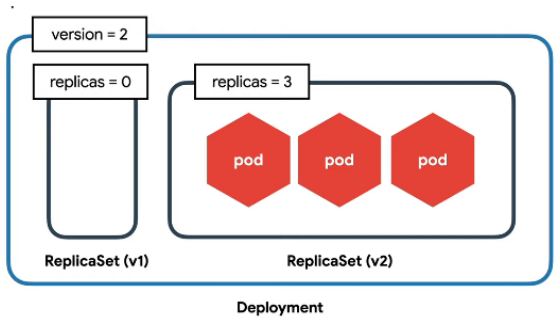
`Deployment`

- 배포 버전을관리

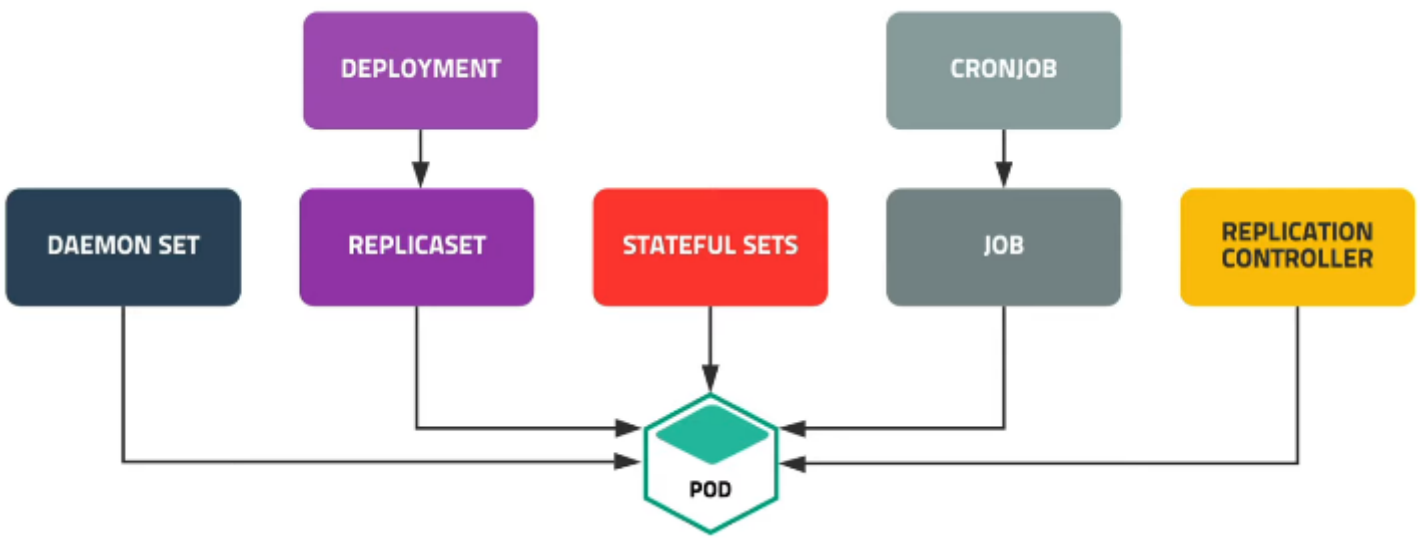
v1을 v2로 바꾸고 싶은데 무중단으로 배포해야하는데 그래서 ReplicaSet을 사용한다.







다양한 Workload



`Daemon Set` : 모든 노드에 꼭 하나씩만 떠있길 원하는 Pod을 만들고 싶을 때 사용. 로그 수집, 모니터링에 적합

`Stateful Sets` : 순서대로 Pod을 실행하고 싶거나 같은 볼륨을 재활용하고 싶을 때 사용

`Job` : 한번 실행하고 죽는 Pod

`Service - ClusterIP`

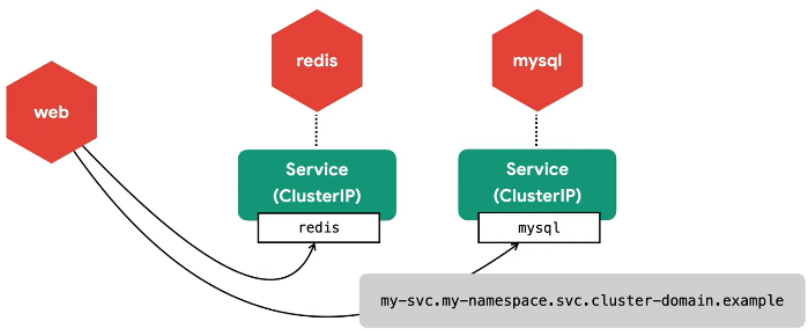
- **클러스터 내부에서 사용하는 프록시**

Pod이 세개가 있는데 세개를 로드밸런서하는 별도의 서비스. ClusterIP에 요청을 보내면 자동으로 세개 중 하나로 요청이 가는거

- **Pod는 동적이지만 서비스는 고유 IP를 가짐**

왜 이렇게 할까? Deployment나 ReplicaSet을 보면 Pod가 업데이트될때 IP를 유지한채로 업데이트되는게 아니라 Pod 죽고 새 Pod이 생기는 거기 때문에 요청을 보낼때 Pod으로 바로 보내는게 아니라 서비스라는 것을 먼저 만들고 (서비스 iP는 고정) Pod이 늘든 줄든 IP가 바뀌든 정상적으로 요청을 보낼 수 있음

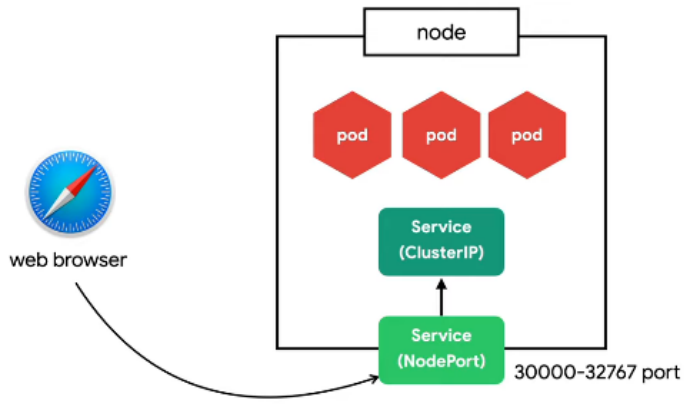
- **클러스터 내부에서 서비스 연결은 DNS를 이용**



웹 서버에서 redis로 통신하고 싶다? 내부 DNS에서 redis라는 도메인이 생기면 저 ClusterIP로 요청을 보내고 그 요청이 다시 안쪽의 Pod로 요청이 간다.

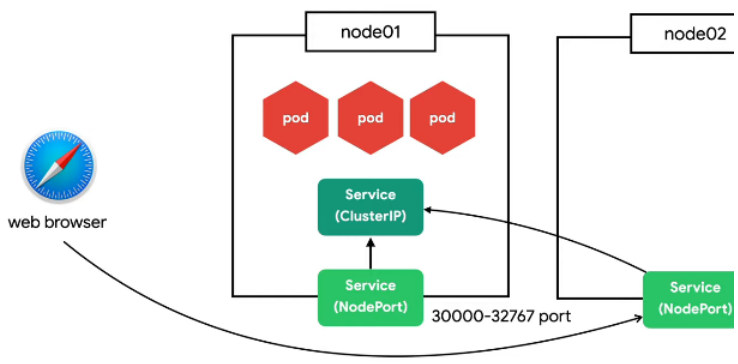
`Service - NodePort`

- **노드(host)에 노출되어 외부에서 접근 가능한 서비스**



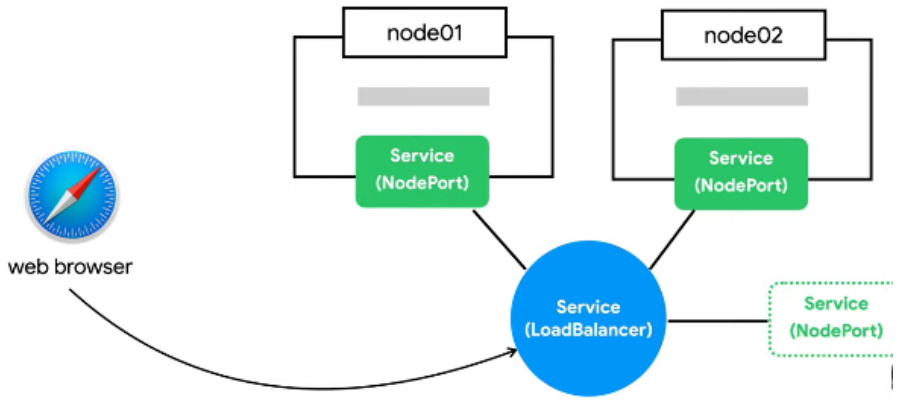
문제는 ClusterIP는 내부에서만 통신할 수 있다. 외부 브라우저로는 접근을 할 수 없음. 외부에서 접근하기 위해 NodePort 개념이 생긴다. 노드에 포트가 생기고 거기에 접속을 하면 그 요청이 ClusterIP로 가고 그 요청이 Pod으로 가도록!

- **모든 노드에 동일한 포트로 생성**



노드가 10개여도 아무 노드들어가도 잘 찾아서 간다

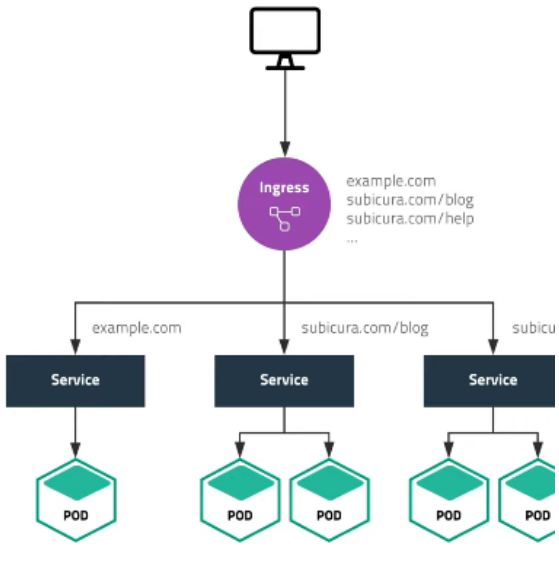
- **하나의 IP주소를 외부에 노출**



문제가 1번 노드를 연결해놨는데 도메인을 서버 자체가 사라질수도 있으니 2번 노드로 붙어야하는데 순간적으로 접속 안된다. 이를 방지하기 위해 로드밸런서를 별도로 둬서 사용자의 웹브라우저로 요청하면 → 로드밸런서→노드포트→ClusterIP→Pod로 요청이 흐름

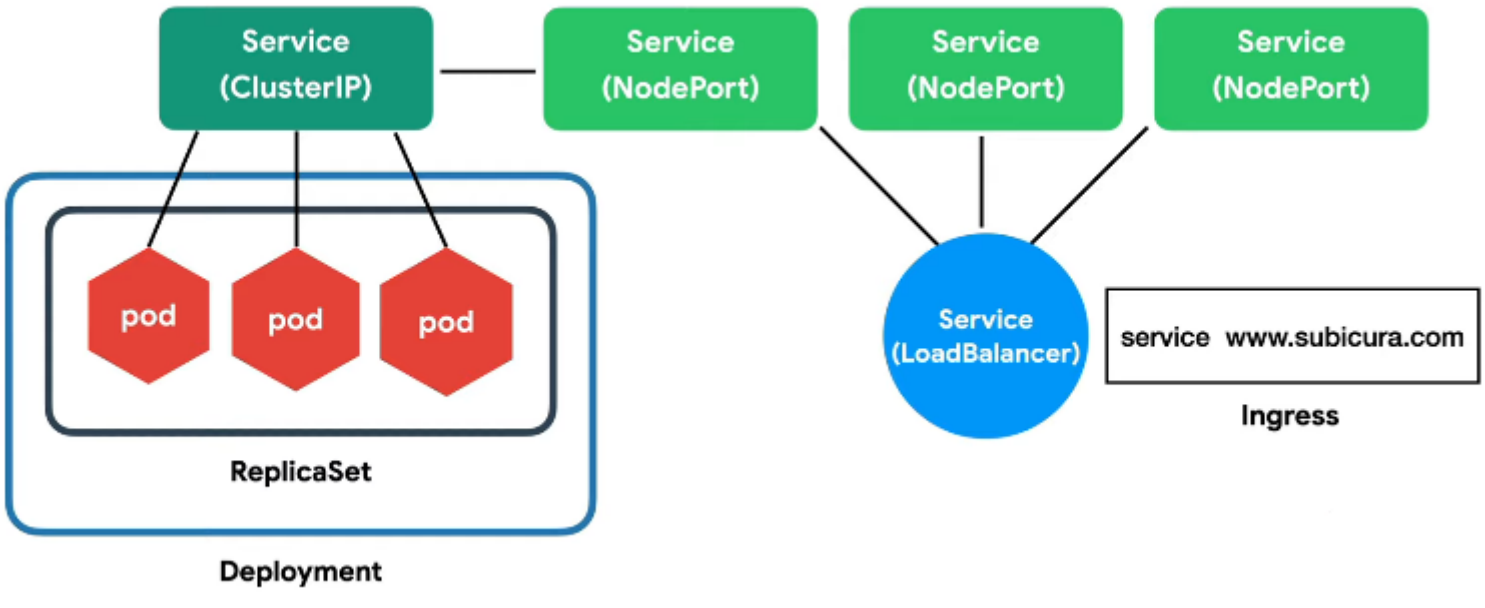
`Ingress`

- **도메인 또는 경로별 라우팅 (Nginx, HAProxy, ALB, …)**



아까는 IP 포트로만 접속하는거라면 인그레스는 도메인 이름, 패스에 따라서 내부에있는 ClusterIP와 연결할 수 있다. 이런 요청이 들어왔을 때 세개를 전부다 로드밸런서, 노드포트를 만드면 자원 낭비이므로 인그레스 하나만 만들고 그 인그레스가 내부적으로 서비스를 분기할 수 있다. 인그레스는 새로 만들기 보다는 기존의 nginx 등등을 재활용해서 쿠버네티스에 맞게 포장해서 사용중

🔢 [일반적인 구성]



그 외 기본 오브젝트

- Volume - Storage (EBS, NFS, …)
- Namespace - 논리적인 리소스 구분
- ConfigMap/Secret - 설정
- ServiceAccount - 권한 계정
- Role/ClusterRole - 권한 설정 (get, list, watch, create, …)

3️⃣ API 호출

`Object Spec - YAML`

- apiVersion
- kind
- metadata
  - name, label, namespace, ..

- spec
  - 각종 설정 (docs..)
- status : 시스템에서 관리하는 최신 상태

API 호출하기 : 원하는 상태(desired state)를 다양한 오브젝트(object)로 정의(spec)하고 API 서버에 yaml 형식으로 전달