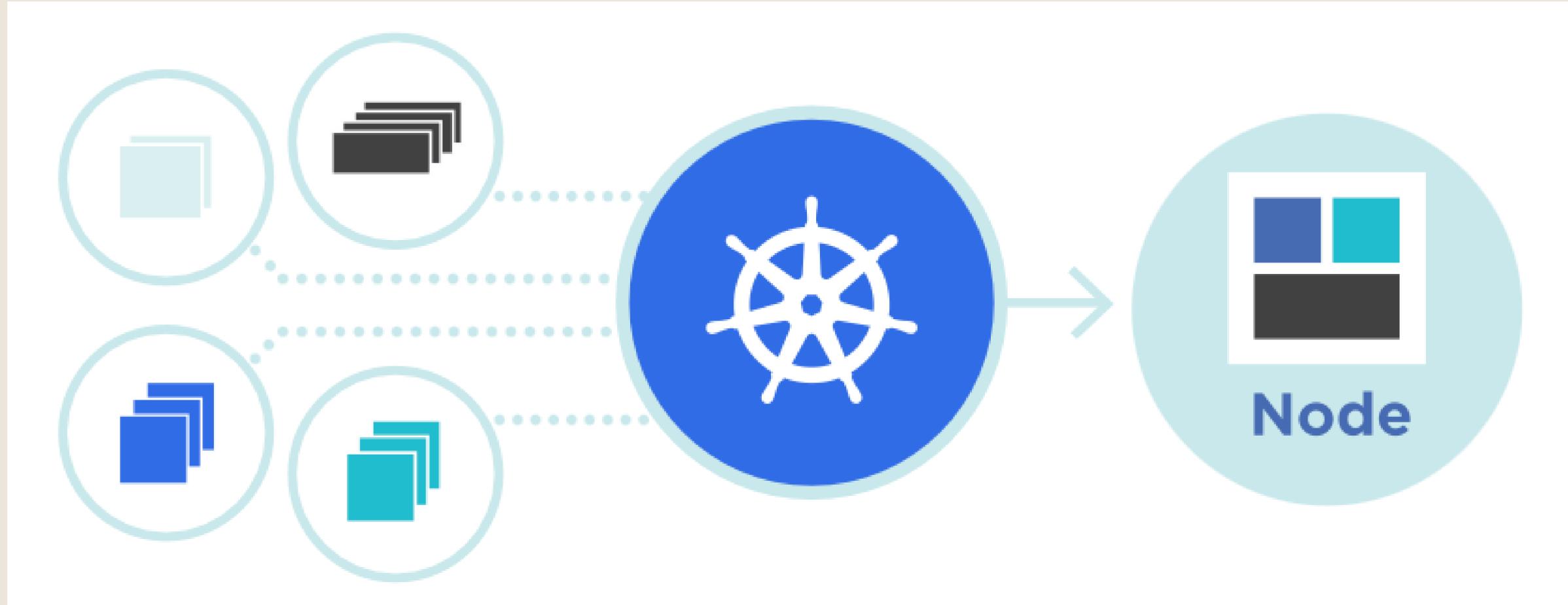


23.07.24

# Kubernetes Basic

풀잎스쿨 이보경

# About Kubernetes



- 도커 컨테이너를 위한 오픈소스 오케스트레이션 프레임워크
- **컨테이너화된 애플리케이션을 자동으로 배포, 스케일링 및 관리해주는 오픈소스 시스템**
- 하나의 머신에서 여러 컨테이너를 실행할 수 있고, 해당 머신이 클러스터를 구성
- 웹 애플리케이션과 같은 장기 실행 서비스 실행 가능 → 이러한 컨테이너의 상태 관리
- 하나의 호스트에서 몇 개의 도커 컨테이너를 수동으로 실행하는 대신, 하나의 노드에서 시작해 최대 수천개의 노드로 올라갈 수 있음
- 자체 데이터센터의 온프레미스, 공용 구글 클라우드, AWS, 다른 클라우드나 하이브리드 어디에서나 실행 가능

# Features of Kubernetes

## 1. 선언적 구성과 자동화 용이

### • 명령적 접근법

- 원하는 상태를 만들기 위해 필요한 동작을 지시하는 방식
- 쿠버네티스: CLI 환경에서 `kubectl` 을 통한 구성요소 생성/수정/삭제 명령어를 수행하는 방식
- 필요한 요소를 명령어 한 줄로 즉시 생성하여 다룰 수 있게 해주는 장점
- 한계점
  - 명령어 만으로 수행 가능한 작업이 제한적
  - 협업 환경에서 작업 내역 추적 어려움
  - 현재 작업 환경의 설정사항을 직접 파악해야 함

```
kubectl run nginx --image=nginx
kubectl create deployment nginx --image=nginx
kubectl expose deployment nginx --port=80
kubectl edit deployment nginx
kubectl scale deployment nginx --replicas=5
kubectl set image deployment nginx nginx:nginx:1.18
kubectl <create|replace|delete> -f nginx.yaml
```

<https://seongjin.me/kubernetes-imperative-vs-declarative/>

# Features of Kubernetes

## 1. 선언적 구성과 자동화 용이

- **선언적 접근법**

- 원하는 상태 그 자체를 선언하는 방식
- 쿠버네티스: YAML 파일을 통해 원하는 구성요소의 상태 기술 -> `**kubectl apply -f <파일명.yaml>**` 형태의 명령어로 적용
- 관리자가 선언한 특정한 형태를 시스템이 스스로 파악하여 반영하는 프로세스

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx:1.20.2
```

- 적용하고자 하는 YAML 파일이
  - 요구되는 문법에 맞게 작성되었는지
  - 해당되는 요소가 기존 클러스터에 이미 배포되었는지
  - 있다면 업데이트,
  - 없다면 신규 생성
- 알아서 진행

```
kubectl apply -f nginx.yaml
```

# Features of Kubernetes

## 2. 자동화된 롤아웃과 롤백

- 애플리케이션 변경시 점진적으로 롤아웃하는 동시에, 애플리케이션을 모니터링해서 모든 인스턴스가 동시에 종료되지 않도록 보장
- 문제가 발생하면 변경사항 롤백

## 3. 스토리지 오케스트레이션

- 로컬 스토리지, AWS, GCP와 같은 퍼블릭 클라우드, 또는 NFS같은 네트워크 스토리지 시스템에서 원하는 시스템 자동으로 마운트

## 4. 서비스 디스커버리와 로드 밸런싱

- 익숙하지 않은 서비스 디스커버리 매커니즘을 사용하기 위해 애플리케이션 수정할 필요 없음
- pod에 고유한 IP 주소와 pod 집합에 대한 단일 DNS명 부여
- 그것들 간에 로드밸런스 수행할 수 있음

## 5. 자가 치유

- 실패한 컨테이너를 재시작하고, 노드가 죽는 경우 컨테이너들을 교체하기 위해 다시 스케줄링
- 사용자가 정의한 상태 체크에 응답하지 않는 컨테이너들 종료
- 서비스를 제공할 준비가 완료될 때까지 해당 컨테이너를 클라이언트에 알리지 않음

# Terms of Kubernetes

## Cluster

- Node라고 불리는 머신들의 집합으로, k8s가 관리하는 컨테이너화된 애플리케이션들 기동함

## Deployment

- 복제된 애플리케이션을 관리하는 API 객체
- 각 레플리카는 각각 하나의 pod로 대표되며, 지정된 레플리카 수만큼 pod 수를 생성 및 유지

## ReplicaSet

- 특정 수의 pod replica들이 동시에 구동되도록 하는 객체

## Node

- k8s의 워커 머신
- pod를 구동하기 위해 필요한 서비스들을 가지며, master 컴포넌트에 의해 관리됨

## Pod

- k8s의 최소 단위 객체
- 클러스터 상에서 동작하는 컨테이너 집합

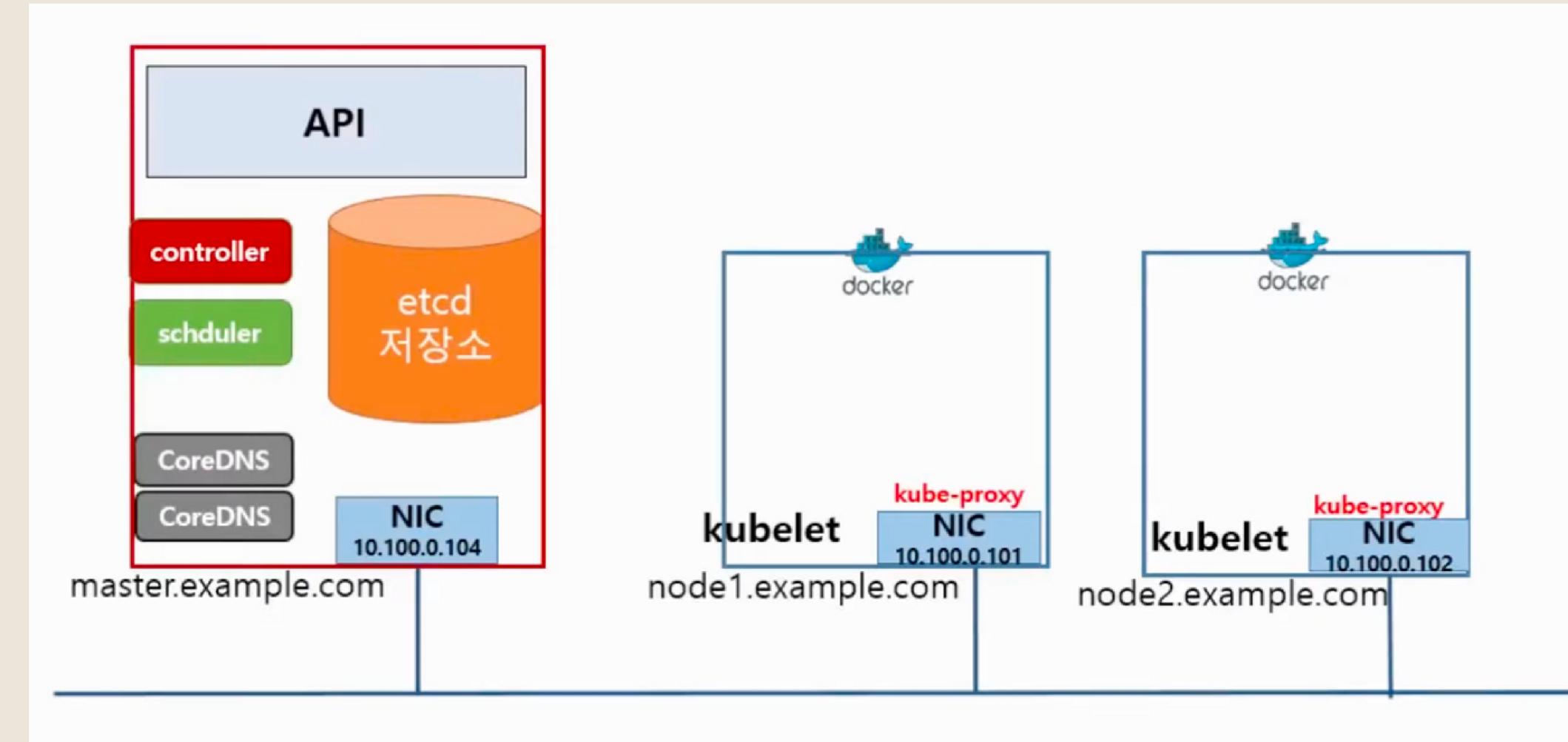
## Service

- pod 집합과 같은 애플리케이션들에 접근하는 방법을 기술하는 API 객체

## Namespace

- 동일한 물리 클러스터에서 여러 가상 클러스터를 지원하기 위해 k8s가 사용하는 추상화

# Kubernetes Architectures

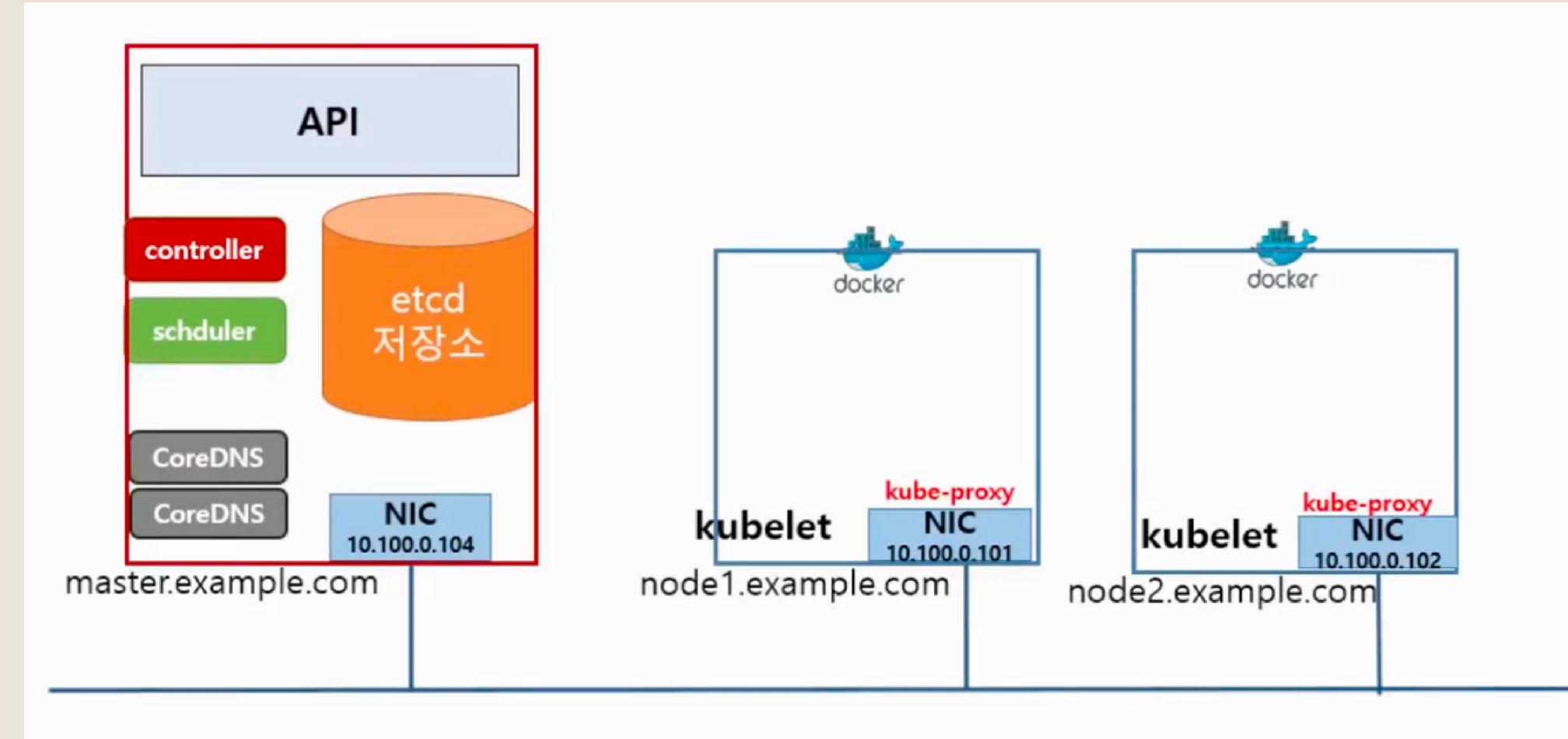


**Master component = Control-plane**

## 1) API component

- `kubectl` 명령으로 요청 받음
- 요청에 대한 문법, 권한 등이 합당한지 검사
- 합당하면 실행을 위해 문법에 맞춰 여러 컴포넌트들과 유기적으로 실행

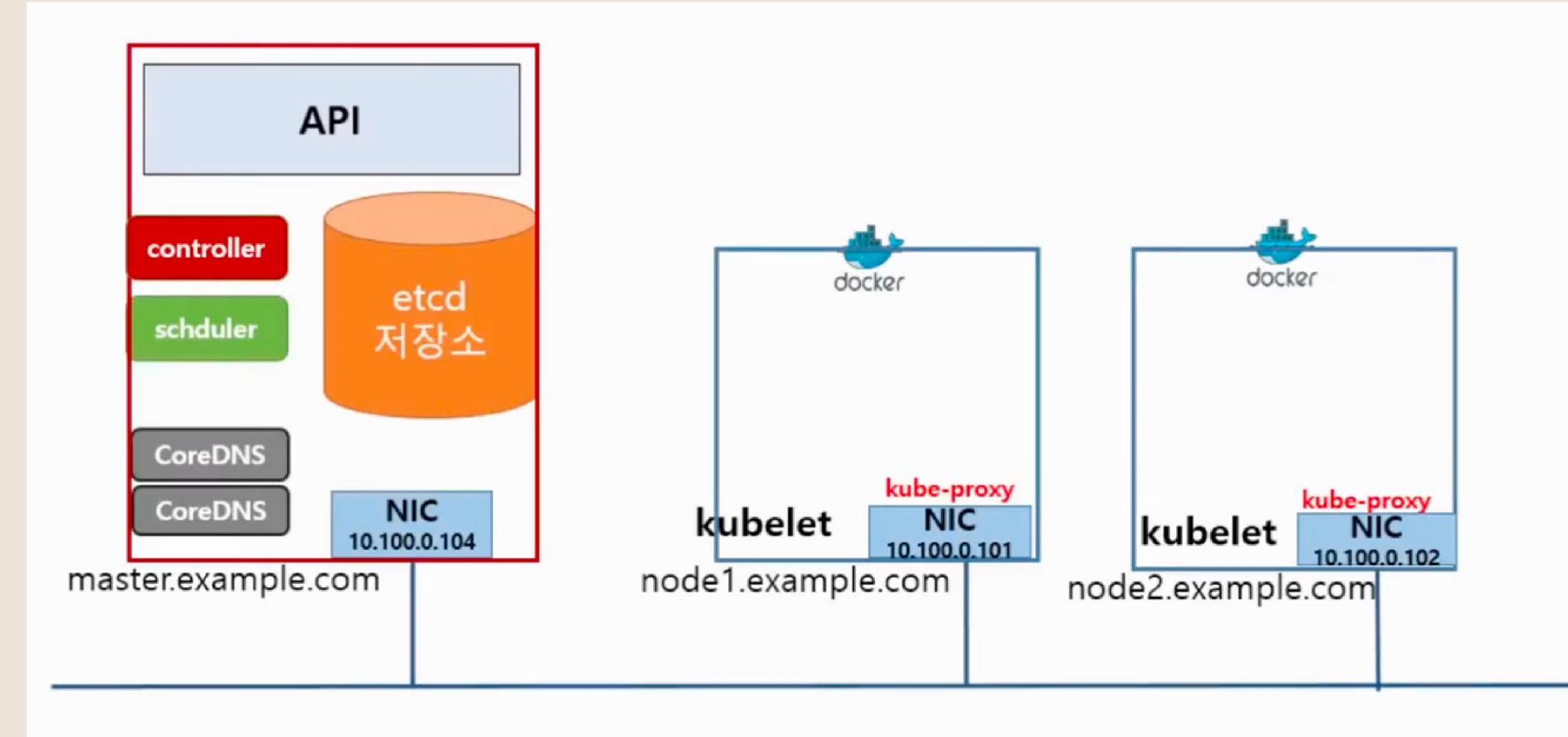
# Kubernetes Architectures



## 2) etcd 저장소

- key:value 타입으로 워커 노드들에 대한 상태 정보 저장
- HW 리소스 어떤식으로 사용중인지, 도커 컨테이너의 상태, 다운받은 image의 상태
- 워커노드 내 kubelet에 수집된 정보들 저장
  - kubelet 안에 cAdvisor라는 컨테이너 모델링 툴 포함되어 있음
  - 현재 워커노드의 컨테이너 기반의 상태 정보 + HW 정보 수집
- kubectl 명령이 어떻게 실행중인지 쿠버네티스 상태 정보도 포함

# Kubernetes Architectures



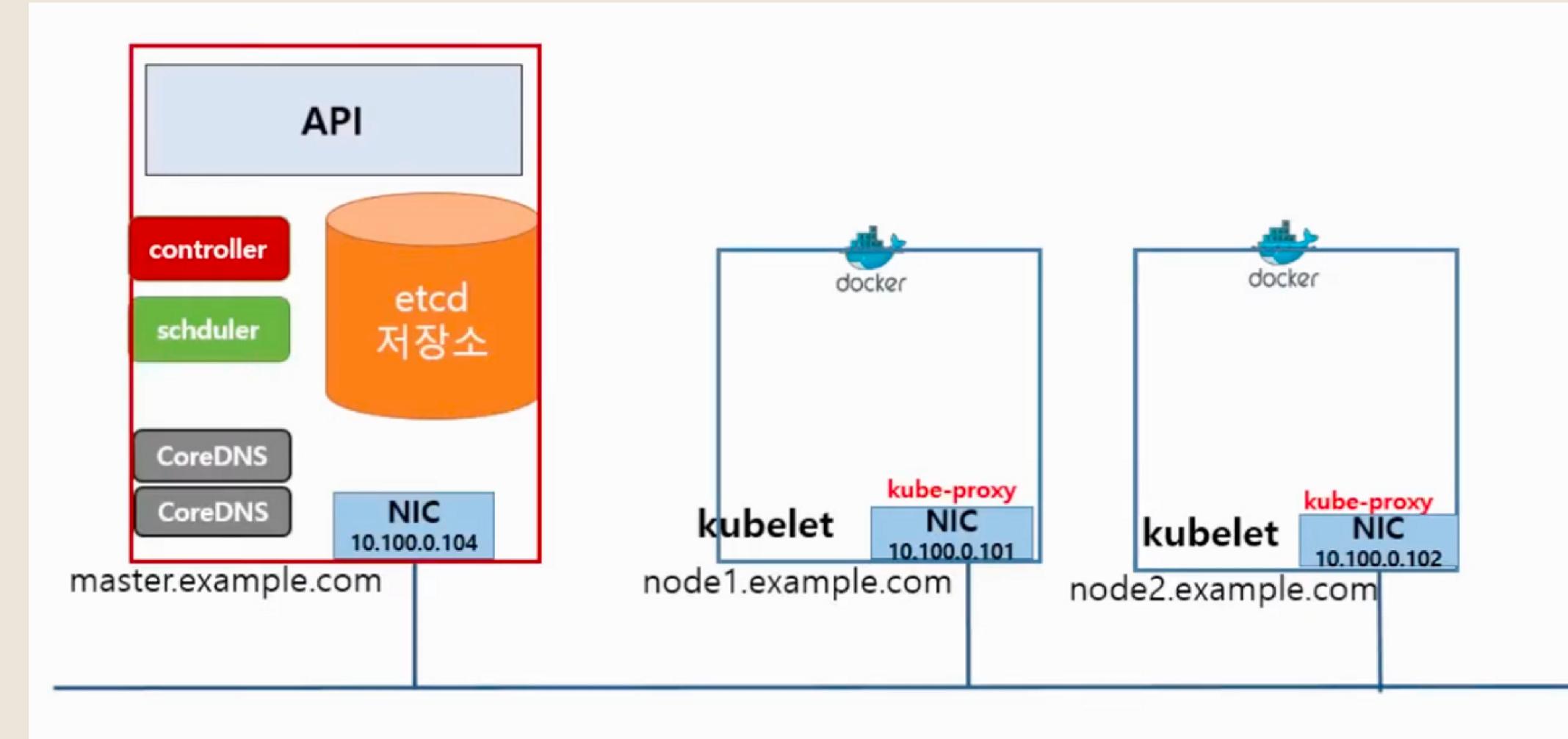
## 3) API는 etcd의 정보를 받아서 스케줄러에게 보냄

- 어떤 노드에 컨테이너를 실행하는 것이 가장 합당할지 물어봄
- 스케줄러: etcd 정보를 바탕으로 컨테이너 실행할 노드 정해서 응답
- API: 응답받은 정보로 해당 노드의 kubelet에 접속 - 요청 (kubectl 명령 실행 요청)

## 4) Kubelet은 docker에게 docker 명령어로 kubectl 명령 실행 요청

- ex) nginx 실행 요청

# Kubernetes Architectures

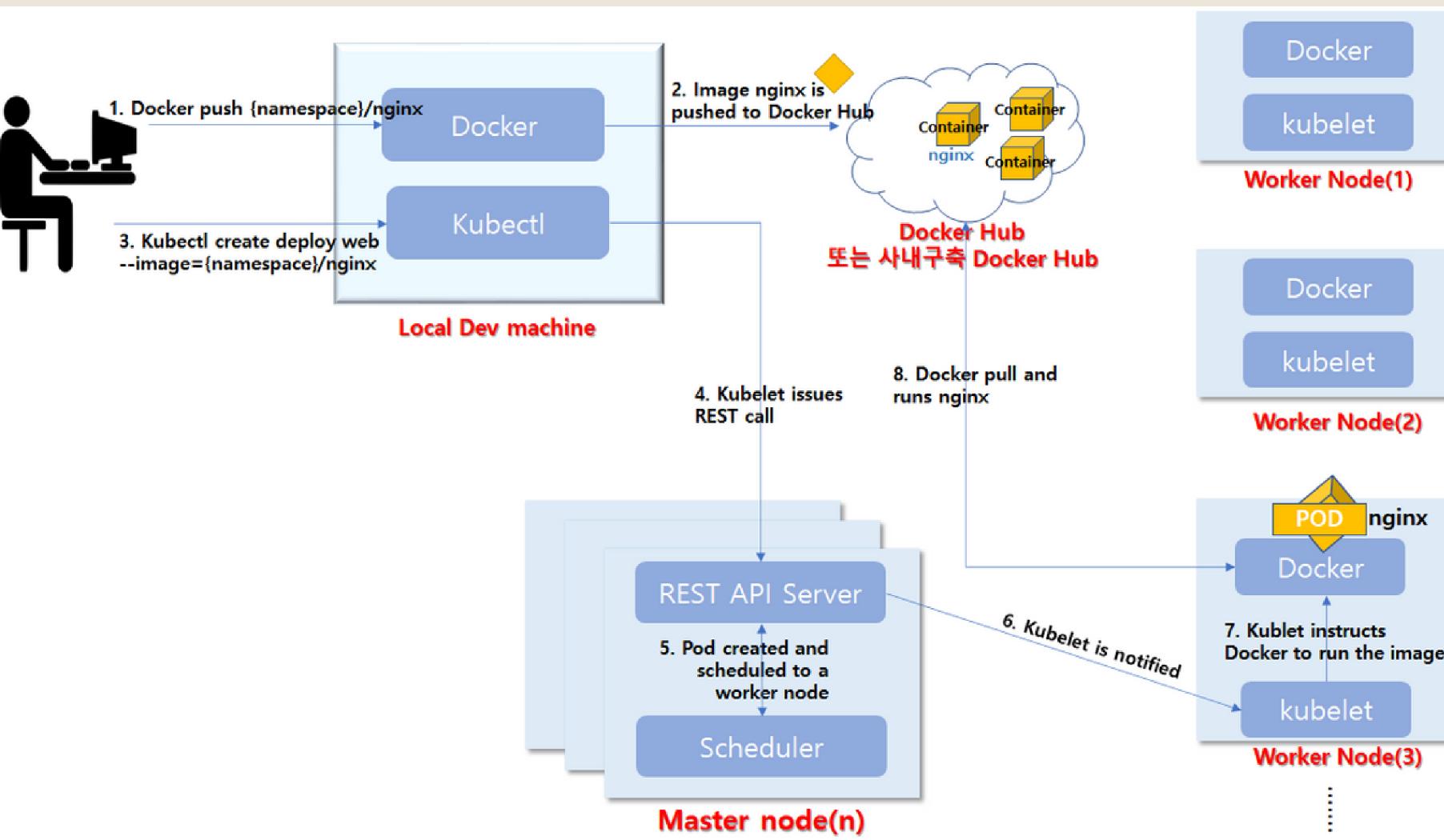


5) docker는 docker 플랫폼에서 hub에 있는 nginx 중 우리가 원하는 버전이 있는지 확인 후 받아서 실행

6) controller: 명령에 따라 컨테이너 개수 보장

- 의도한 상태에 가깝게 유지하는 역할
- 컨트롤러: 컨테이너 1개가 동작 중인지 계속 확인 → nginx 동작 중인 노드 다운
- 다른 노드에 nginx를 실행할 수 있도록 스케줄러를 통해 정보를 얻어 가능한 곳에 다시 nginx 동작

# Kubernetes Container Working Flow



- 1) 여러 컨테이너 빌드 (main UI 생성, 로그인, 상품 주문 등)
- 2) 도커 명령 → 도커 허브에 push
- 3) 쿠버네티스 명령 → 컨테이너 실행
  - yaml
  - CLI
  - `kubectl create deploy web \ --image=hub.example.com/nginx`
- 4) **kubectl** 명령 → master node에게 전달
- 5) **master node**의 API가 kubectl 명령어 요청 받음
- 6) **API**: 요청에 따라 컨테이너를 어느 노드에 배치할지 스케줄러에게 요청
- 7) **스케줄러**: 노드들의 상태를 보고 컨테이너 실행할 노드 선택하여 응답
- 8) **API**: 해당 노드의 kubelet에게 명령어 실행 요청 보냄
- 9) **해당 노드 kubelet**: 해당 요청을 docker 명령어로 바꿔서  
도커 데몬에게 실제 컨테이너 실행 요청
- 10) **도커 데몬**: 명령어에 정의되어있는 허브에 해당 컨테이너가 있는지 검색  
있으면 받아와서 노드에 컨테이너로 실행
- 11) **쿠버네티스**: 이렇게 동작하는 컨테이너를 pod라는 단위로 관리

# Node

- 컨테이너를 pod 내에 배치하고 노드에서 실행함으로써 워크로드 구동
- 클러스터에 따라 가상머신 또는 물리적인 머신일 수 있음
- 각 노드는 control-plane에 의해 관리되며, pod를 실행하는 데 필요한 서비스 제공

- **컴포넌트**

- kubelet : pod에서 컨테이너가 확실하게 동작하도록 관리
- 컨테이너 런타임 : 컨테이너 실행을 담당하는 소프트웨어
- kube-proxy : 각 노드에서 실행되는 네트워크 프록시

- **`kubectl describe node <node name>`**

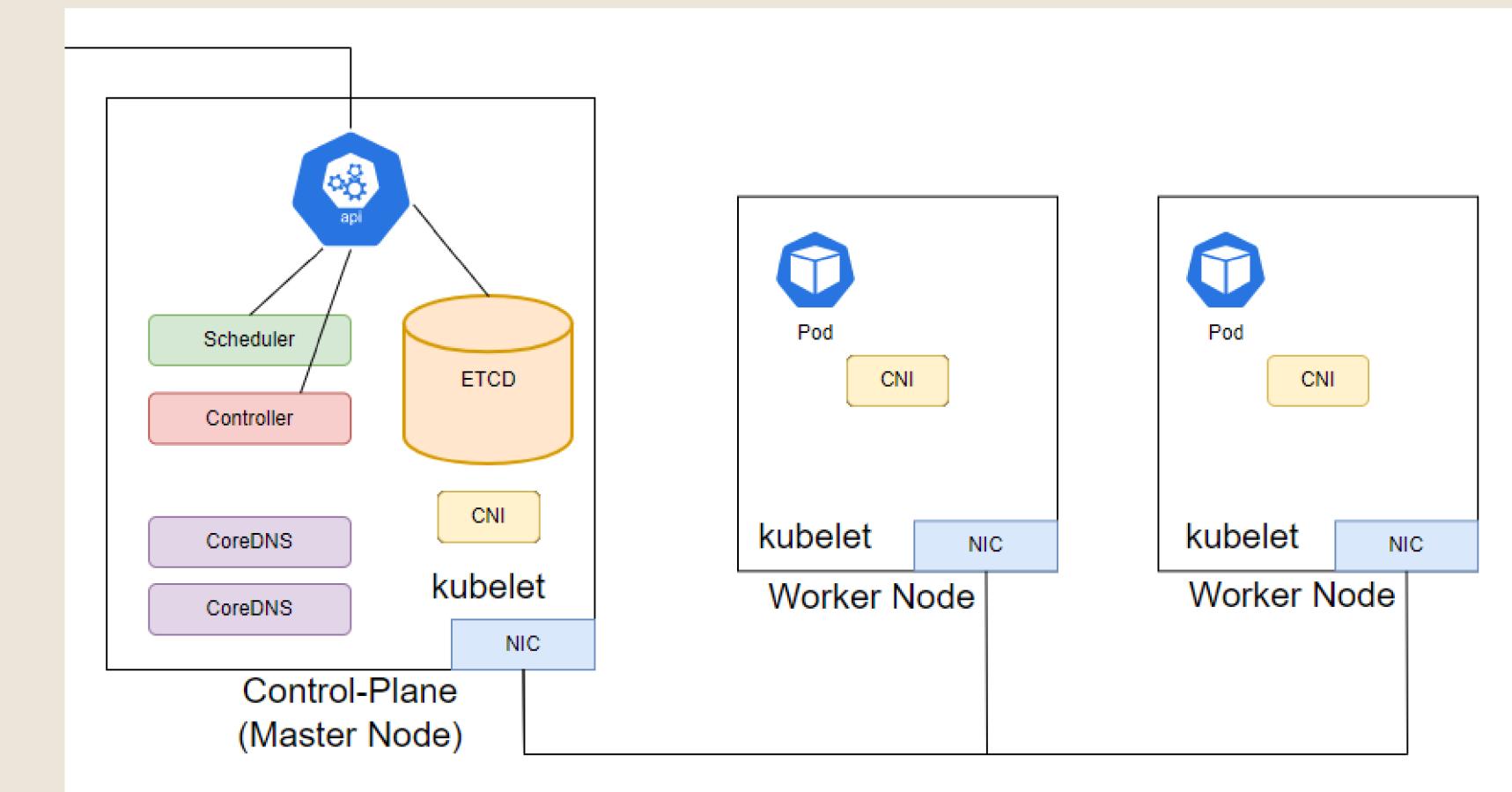
- 주소, 컨디션, 용량과 할당가능, 정보

- **하트비트**

- 클러스터가 개별 노드가 사용가능한지 판단할 수 있도록 도움
- 장애가 발견된 경우 조치할 수 있게 함
- kubelet이 `.status` 생성과 업데이트 & 관련된 lease의 업데이트 담당

- **컨트롤러**

- 등록 시점에 노드에 CIDR 블럭 할당
- 컨트롤러의 내부 노드 리스트를 최신 상태로 유지
- 노드의 동작 상태 모니터링 (.status 필드의 컨디션 업데이트)



# Lease

- 분산 시스템에서 공유 리소스를 잠그고, 노드간의 활동을 조정하는 메커니즘 제공
  - `coordination.k8s.io` API 그룹에 있는 `Lease` 오브젝트로 표현
  - 각 노드는 연관된 리스 오브젝트를 가짐
- 
- **하트비트**
    - 리스 API를 통해 kubelet 노드의 하트비트를 쿠버네티스 API 서버에 전달
    - 모든 kubelet 하트비트는 이 Lease 오브젝트에 대한 업데이트 요청
    - 이 요청은 해당 오브젝트의 `spec.renewTime` 필드 업데이트
    - control-plane: 이 필드의 타임스탬프를 통해 해당 노드의 가용성 확인
  - **리더 선출**
    - 특정 시간동안 컴포넌트의 인스턴스 하나만 실행되도록 보장
    - 컨트롤러, 스케줄러와 같은 control-plane 컴포넌트의 고가용성 설정에서 사용됨
    - 한 인스턴스만 활성 상태로 실행되고, 다른 인스턴스는 대기 상태여야 함
  - **API 서버 신원**
    - 클라이언트가 쿠버네티스 control-plane을 운영중인 kube-apiserver 인스턴스 수를 파악할 수 있는 메커니즘 제공
    - 더 이상 존재하지 않는 kube-apiserver의 만료된 임대는 정해진 시간 후에 새로운 kube-apiserver에 의해 가비지컬렉션 됨

# Pod

- 쿠버네티스에서 생성하고 관리할 수 있는 배포 가능한 가장 작은 컴퓨팅 단위
- 컨테이너를 실행하기 위한 환경으로, 하나 이상의 컨테이너 그룹으로 구성
- 스토리지 및 네트워크 공유, 해당 컨테이너 구동 방식에 대한 명세
- 일반적으로 pod는 직접 생성하지는 않으며, 워크로드 리소스를 사용하여 생성

ex) nginx:1.14.2 이미지를 실행하는 컨테이너로 구성되는 pod의 예시

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

```
kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
```

# Pod

## Replication

- 각 pod는 특정 애플리케이션의 단일 인스턴스를 실행하기 위한 것
- 더 많은 인스턴스를 실행하여 리소스를 제공하기 위해 애플리케이션 수평적으로 확장하려면
  - 각 인스턴스에 하나씩, 여러 pod를 사용해야 함
  - 이 과정을 Replication이라고 함
- 복제된 pod는 일반적으로 워크로드 리소스와 컨트롤러에 의해 그룹으로 생성되고 관리됨

## PodTemplate

- 워크로드 리소스의 컨트롤러가 pod를 생성하기 위한 명세
- 파트 템플릿을 변경하는 경우, 대체 파드를 생성해야 함 (컨트롤러가)
- kubelet은 파트 템플릿과 업데이트에 대한 정보를 직접 관리 X
  - 상세 내용은 추상화됨
  - 시스템 시맨틱 단순화 & 코드 변경 없이 클러스터 동작 확장

ex) 컨테이너는 메시지를 출력한 다음 일시 중지

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # 여기서부터 파드 템플릿이다
    spec:
      containers:
        - name: hello
          image: busybox:1.28
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
          restartPolicy: OnFailure
    # 여기까지 파드 템플릿이다
```

# Workloads

- 쿠버네티스에서 구동되는 애플리케이션 - 일련의 Pod 집합 내에서 실행
- pod의 라이프사이클
  - 클러스터에서 실행
  - 동작 중인 노드에 심각한 오류가 발생하면 해당 노드의 모든 pod가 실패 = final
  - 사용자는 향후 노드의 복구와 상관없이 pod를 새로 생성해야 함
- pod를 직접 관리할 필요 없이, pod 집합을 관리하는 워크로드 리소스를 사용할 수 있음
  - 올바른 수의 올바른 pod 유형이 실행되고 있는지 확인하는 컨트롤러 구성
- built-in 워크로드 리소스
  - **Deployment 및 ReplicaSet** : 클러스터의 stateless 애플리케이션 워크로드 관리에 적합
  - **StatefulSet** : 어떻게든 상태를 추적하는 하나 이상의 pod를 동작하게 해줌.
  - **DemonSet** : 노드-로컬 기능을 제공하는 pod 정의 (네트워킹 지원 도구 / add-on 등)
  - **Job 및 CronJob** : 실행 완료 후 중단되는 작업 정의
  - 추가적인 제 3자의 워크로드 리소스도 추가할 수 있음

# Workloads

## ReplicaSet

- Replica pod 집합의 실행을 항상 안정적으로 유지하는 것
- 보통 명시된 동일 pod 개수에 대한 가용성을 보증하는데 사용
- 작동 방식
  - 정의하는 필드
    - selector : 획득 가능한 pod를 식별하는 방법
    - replicas: 유지해야 하는 pod 개수
    - podTemplate : 레플리카 수 유지를 위해 생성하는 신규 pod에 대한 데이터 명시
  - 필드에 지정된 설정을 충족하기 위해 pod 생성 및 삭제
  - 새로운 pod 생성할 경우, 명시된 podTemplate 사용
- 사용 시기
  - 지정된 수의 pod 레플리카가 항상 실행되도록 보장
  - 별도의 사용자 정의 업데이트 오케스트레이션이 필요한 경우 / 업데이트가 전혀 필요없는 경우가 아니라면
  - ReplicaSet을 직접 사용하기보다는 Deployment를 사용하는 것을 권장
    - ReplicaSet 오브젝트를 직접 조작할 필요 없음
    - 배포 작업 세분화하여 조작 가능

ex) controllers/frontend.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # 케이스에 따라 레플리카를 수정한다.
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

# Workloads

## Deployment

- ReplicaSet의 상위 개념
- pod와 ReplicaSet에 대한 선언적 업데이트 제공
- Deployment는 '의도하는 상태'를 설명
- Deployment 컨트롤러는 '현재 상태'에서 '의도하는 상태'로 비율을 조정하며 변경
- Use Case
  - ReplicaSet 롤아웃 할 Deployment 생성
  - pod의 새로운 상태 선언 (PodTemplateSpec 업데이트)
  - Deployment 이전 버전으로 롤백
  - Deployment 스케일 업
  - Deployment 롤아웃, 상태 관리
  - 이전 ReplicaSet 정리
- ex) 3개의 nginx pod를 불러오기 위한 ReplicaSet 생성  
`controllers/nginx-deployment.yaml`  

```
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
```

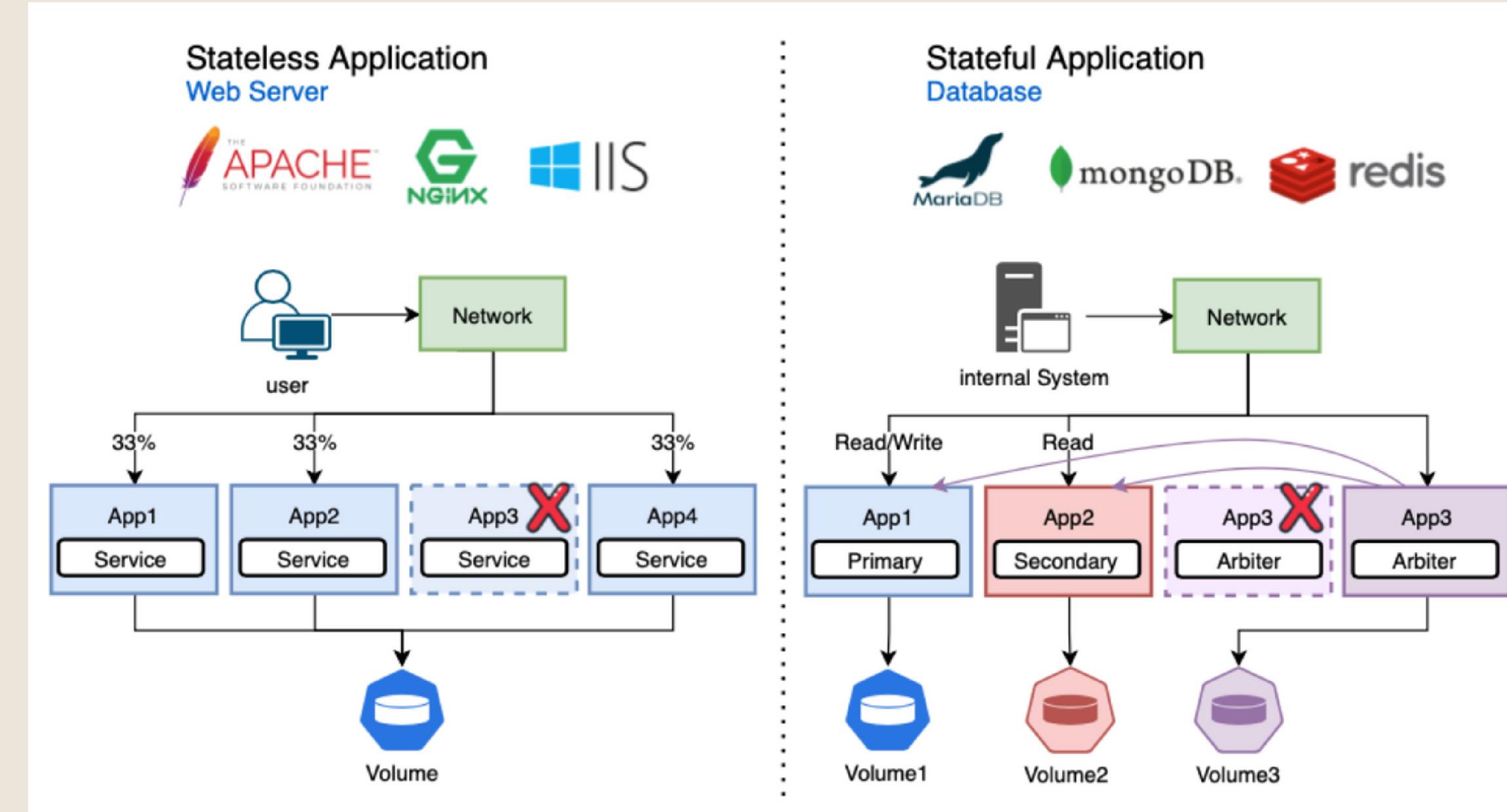
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

# Workloads

## StatefulSet

- 애플리케이션의 상태 관리
- pod 집합의 Deployment와 스케일링 관리, **pod들의 순서 및 고유성 보장**
- 동일한 컨테이너 스펙을 기반으로 둔 pod들 관리, 각 pod의 독자성 유지
- 다음 중 하나 이상이 필요한 애플리케이션에 유용
  - 안정된, 고유한 네트워크 식별자
  - 안정된, 지속성을 갖는 스토리지
  - 순차적인, 정상 배포와 스케일링
  - 순차적인, 자동 롤링 업데이트
- **pod마다 다른 스토리지를 사용해 각각 다른 상태를 유지하기 위해 사용**
- k8s의 ms 구조로 동작하는 애플리케이션은 대부분 stateless : deployment, replicaSet
  - 모두 같은 volume으로 같은 상태를 가질 수 밖에 없음
  - DB처럼 상태를 갖는(stateful) 애플리케이션을 k8s 실행하는 것은 매우 복잡
  - 완벽하진 않지만 이에 대한 해결책으로 statefulSet 오브젝트 제공
- 문제가 생긴 pod와 같은 이름, 같은 IP를 가진 pod로 교체

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # .spec.template.metadata.labels 와 일치해야 한다
  serviceName: "nginx"
  replicas: 3 # 기본값은 1
  minReadySeconds: 10 # 기본값은 0
  template:
    metadata:
      labels:
        app: nginx # .spec.selector.matchLabels 와 일치해야 한다
  spec:
    terminationGracePeriodSeconds: 10
    containers:
    - name: nginx
      image: registry.k8s.io/nginx-slim:0.8
      ports:
      - containerPort: 80
        name: web
      volumeMounts:
      - name: www
        mountPath: /usr/share/nginx/html
    volumeClaimTemplates:
    - metadata:
        name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```



# Workloads

## DemonSet

- 클러스터 전체에서 pod를 띄울 때 사용
- 모든 / 일부 노드가 pod의 사본을 실행하도록 함
- 노드가 클러스터에 추가되면 pod도 추가됨
- 노드가 클러스터에서 제거되면 해당 pod는 가비지로 수집됨
- 데몬셋을 삭제하면 데몬셋이 생성한 파드들이 정리됨
- 용도
  - 모든 노드에서 클러스터 스토리지 데몬 실행
  - 모든 노드에서 로그 수집 데몬 실행
  - 모든 노드에서 노드 모니터링 데몬 실행
- Deployment와 유사하게 pod 생성하고 관리
  - Deployment : 배포 작업 세분화
  - DemonSet : 특정 노드 또는 모든 노드에 실행되어야 할 특정 pod 관리
    - 새로 노드가 추가될 때 자동으로 그 노드에 데몬셋 pod 실행됨
- 데몬셋 pod와 통신
  - push, 노드IP와 포트, DNS, Service

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
        # 이 �oller레이션(toleration)은 데몬셋이 컨트롤 플레인 노드에서 실행될 수 있도록 만든다.
        # 컨트롤 플레인 노드가 이 파드를 실행해서는 안 되는 경우, 이 �oller레이션을 제거한다.
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          operator: Exists
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

# Workloads

## Job

- job에서 하나 이상의 pod를 생성하고, 지정된 수의 pod가 성공적으로 종료될 때까지 계속해서 pod의 실행 재시도
- pod가 성공적으로 완료되면, 성공적으로 완료된 job 추적
- 지정된 수의 성공 완료에 도달하면 job 완료됨
- job 삭제하면 job이 생성한 pod 정리됨, 일시중지하면 다시 재개될 때까지 활성 pod가 삭제됨
- 용도
  - 백업이나 특정 배치 파일들처럼 한 번 실행하고 종료되는 작업
  - 여러 pod를 병렬로 실행할 수 있음

ex) 파이의 2000자리까지 계산해서 출력

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl:5.34.0
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: Never
      backoffLimit: 4
```

# Service

## 쿠버네티스 네트워크 모델

- 클러스터의 모든 pod는 고유한 IP 주소 가짐
- pod 간 연결을 명시적으로 만들 필요 X, 컨테이너 포트를 호스트 포트에 매핑할 필요 거의 X
- 포트 할당, 네이밍, 서비스 디스커버리, 로드 밸런싱, 애플리케이션 구성, 마이그레이션 관점에서 pod를 VM / 물리 호스트처럼
- 쿠버네티스 IP 주소는 pod 범주 내에 존재하며, pod 내의 컨테이너들은 IP 주소, MAC 주소를 포함하는 네트워크 네임스페이스 공유
  - pod 내의 컨테이너들이 각자의 포트에 `localhost`로 접근할 수 있음을 의미

## 쿠버네티스 네트워킹 특징

- pod 내의 컨테이너는 loopback을 통한 네트워킹을 사용하여 통신
- 클러스터 네트워킹은 서로 다른 pod 간의 통신 제공
- 서비스 API를 사용하면 pod에서 실행 중인 애플리케이션을 클러스터 외부에서 접근할 수 있음
- 서비스를 사용하여 서비스를 클러스터 내부에서만 사용할 수 있도록 게시할 수 있음

# Service

## 서비스

- pod 집합에서 실행 중인 애플리케이션을 네트워크 서비스로 노출하는 추상화 방법
  - pod의 논리적 집합, 어떻게 접근할지에 대한 정책을 정의해놓은 것
  - 익숙하지 않은 서비스 디스커버리 메커니즘을 사용하기 위해 애플리케이션을 수정할 필요 없음
  - 원래 각 pod의 IP로는 외부에서 직접적인 접근이 불가능하지만, 서비스는 이를 가능하게 해줌
  - `label selector`를 통해 어떤 pod를 포함할지 정의할 수 있음
- 
- 용도
    - pod에 고유 IP 주소 부여
    - pod 집합에 대한 단일 DNS명 부여
    - 여러 pod 묶어서 로드밸런싱
  - 유형
    - ClusterIP (default) : pod들이 **클러스터 내부**의 다른 리소스들과 통신할 수 있도록하는 가상의 클러스터 전용 IP
    - NodePort: 외부에서 노드 IP의 특정 포트로 들어오는 요청 감지, **모든 노드**의 IP와 포트로 접근 가능하게 됨
    - LoadBalancer: 보통 클라우드 벤더에서 제공하는 설정 방식, 외부 IP를 가지고 있는 **로드밸런서** 할당
    - ExternalName: selector 대신 DNS명을 직접 명시하고자 할 때 사용

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

# Add-on Program

## Add-on

- 마스터노드, 워커노드 외에 클러스터 내부에서 필요한 기능들을 위해 실행되는 pod
- 쿠버네티스 클러스터에서 기능 구현 및 확장하는 역할
- pod는 deployment, replication controller 등에 의해 관리됨
- 외부 연동 라이브러리라고 생각하면 됨

## DNS Addon

- 클러스터 내에서 작동하는 DNS Server로, 필수적인 애드온!
- 쿠버네티스 서비스 객체에게 DNS 레코드를 제공하는 역할
- 쿠버네티스 내부에서 실행된 컨테이너들은 자동으로 DNS 서버에 등록되어 자동 탐색 가능
- DNS Server
  - 사람이 읽을 수 있는 도메인 이름을 IP 주소로 변환할 수 있게 해주는 인터넷의 핵심 구성 요소
  - pod 간 통신을 할 때 IP가 아닌 도메인을 설정해 두고 사용할 수 있음
  - 한 클러스터에서 사용하던 YAML 파일에서 pod 간 통신을 도메인으로 설정해 둔다면, 그대로 다른 클러스터로 가져가서 사용 가능
  - pod는 쉽게 생성하고 삭제할 수 있는 불안정한 자원 → IP로 서비스 호출하면 바뀐 IP 계속 추적해야 함

# Add-on Program

## 네트워킹 Addon (CNI) : Container Network Interface

- 컨테이너 간의 네트워킹을 제어할 수 있는 플러그인을 만들기 위한 표준
- 클러스터는 가상 네트워크가 구성되어 있는데, 기본적으로는 Kube-proxy가 네트워크 관리
- 보다 효율적인 네트워크 환경 구성을 위해 다양한 네트워크 관련 Addon 제공됨

## 대시보드 Addon

- 클러스터를 위한 웹 기반 UI
- 일반적으로 클러스터에 명령을 내릴 때 사용하는 kubectl을 가시화하여 대시보드 제공
- 관리의 편의성 향상

## 컨테이너 자원 모니터링 Addon

- 컨테이너의 CPU, 메모리와 같은 리소스 데이터를 저장하고 볼 수 있는 방법 제공
- 워커노드의 kubelet에 포함된 cAdvisor 도 이것의 한 종류

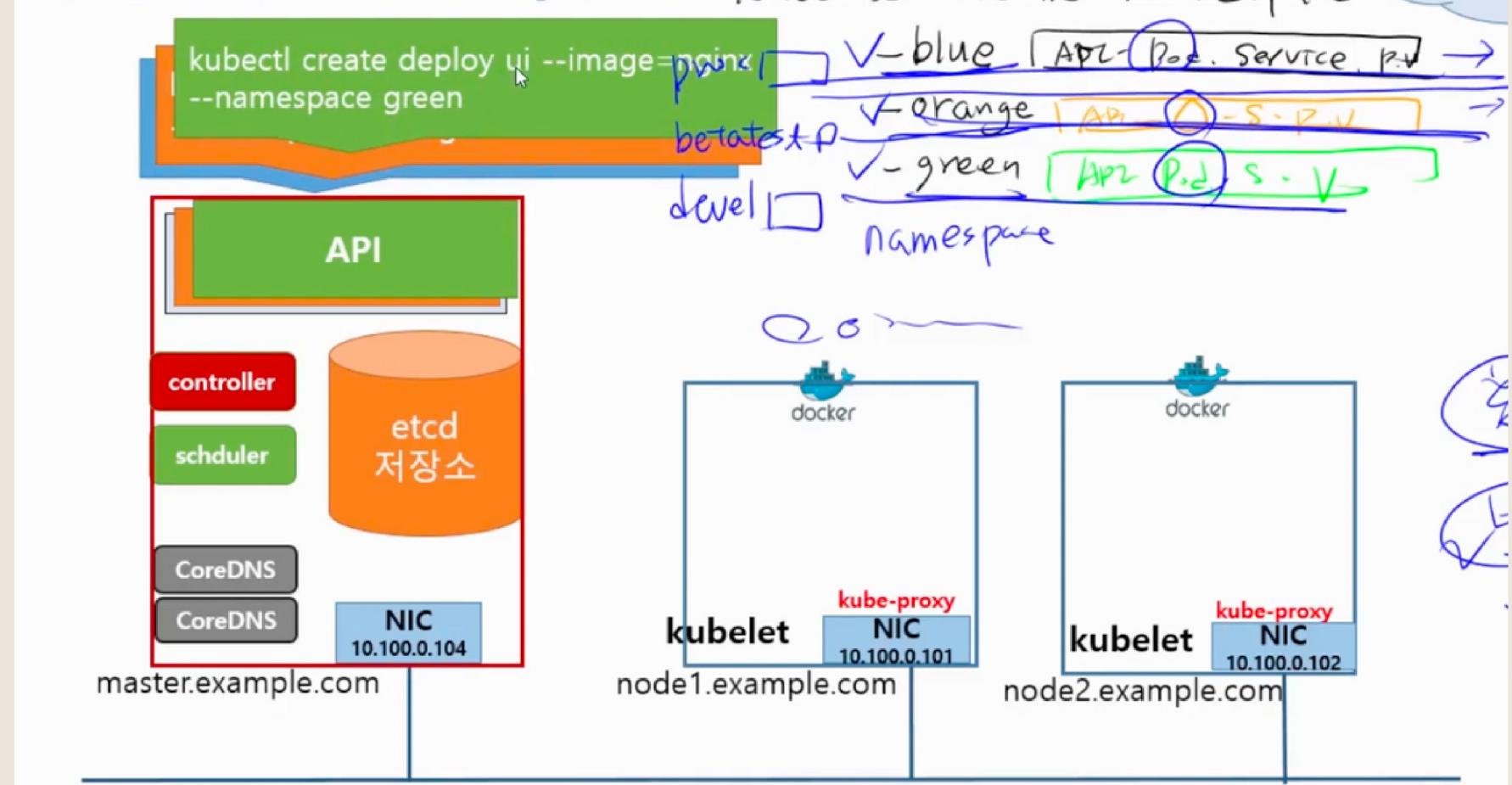
## 클러스터 로깅 Addon

- 클러스터 내부에서 생성되는 모든 로그를 중앙화하여 관리할 수 있음
- 클라우드 서비스가 아닌 직접 쿠버네티스를 설치한 경우에 사용 (클라우드 - 클라우드 벤더에서 로깅 서비스 제공)

# Namespace

- k8s API 중 하나
- 단일 클러스터 내에서의 리소스 그룹 격리 메커니즘을 제공
- 클러스터 한 개를 **여러 개의 논리적인 단위로 나눠서 사용**
- 클러스터 한 개를 여러 팀이나 사용자가 함께 공유 / 용도에 따라 실행해야 하는 앱을 구분할 때 사용
- 여러 파트너사를 운영하는 통합 솔루션 / 여러 버전을 가지고 개발에서 운영까지 하는 과정 등에 필요
- 실제 API는 하나인데, API가 여러 개인 것처럼 사용할 수 있음
  
- ex) product, beta test, develop 버전
- ex) 롯데 이커머스 - 홈쇼핑, 백화점, 면세점

## 쿠버네티스 namespace



## 장점

- 위의 경우, 홈쇼핑에서 동작중인 pod, service, volume 등만 따로 모아서 보고 관리할 수 있음
- 수많은 pod 중 특정 namespace 것만 따로 보아서 한눈에 파악, 관리하기 좋음

# Default Namespace

## **default**

- 처음에 포함되어 있는 네임스페이스로, 네임스페이스를 생성하지 않고도 새 클러스터를 사용할 수 있음

## **kube-node-lease**

- 각 노드와 연관된 리스 오브젝트를 가짐
- 노드 리스: kubelet이 하트비트를 보내서 control-plane이 노드의 장애를 탐지할 수 있게 함

## **kube-public**

- 모든 클라이언트가 읽기 권한으로 접근할 수 있음
- 주로 전체 클러스터 중 공개적으로 드러나서 읽을 수 있는 리소스를 위해 예약되어 있음

## **kube-system**

- 쿠버네티스 시스템에서 생성한 오브젝트를 위한 네임스페이스

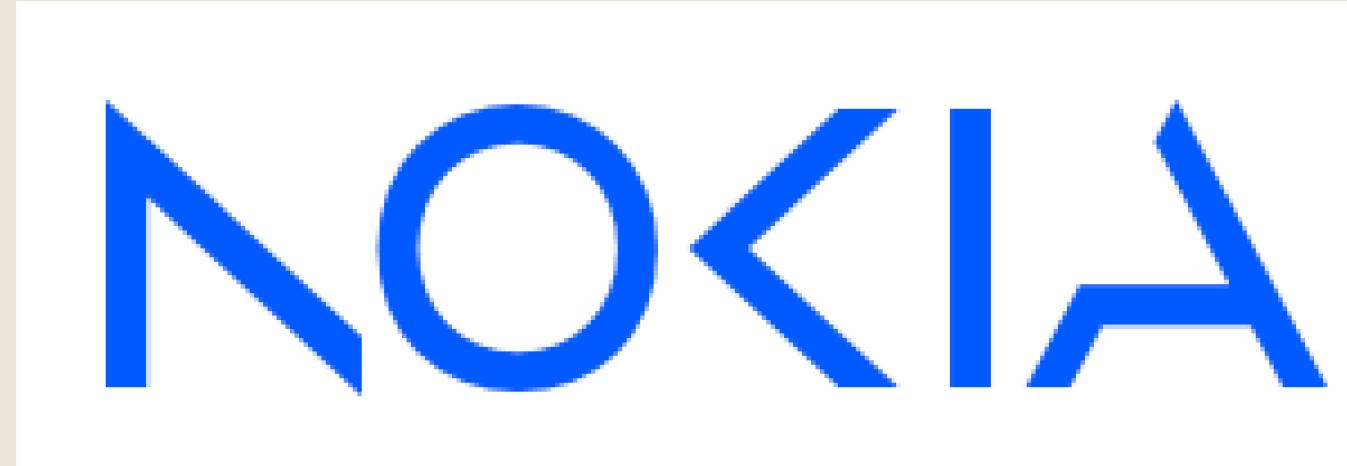
# Case Study



## 인공지능 연구소 OpenAI

- 자체 데이터 센터에서 실험을 실행하고, 쉽게 확장할 수 있는 딥러닝을 위한 인프라 필요
- 주로 배치 스케줄링 시스템으로 사용
- 클러스터를 동적으로 확장하기 위해 오토스케일러에 의존
- 낮은 대기 시간과 빠른 반복을 통해 유휴 노드의 비용 크게 줄일 수 있음
- 일관된 API 제공 -> 연구 실험을 클러스터 간 쉽게 이동할 수 있음
- 자체 데이터 센터 사용 -> 비용 절감, 클라우드에서 HW에 대한 엑세스 제공
- 분산 훈련 시스템 연구 : 1-2주만에 수 백개의 GPU로 확장하여 실행

# Case Study



## 통신 네트워크 솔루션 회사 NOKIA

- 통신 네트워크를 종단간 구축하는 것
- 여러 통신 사업자에게 소프트웨어 제공, 소프트웨어를 인프라에 넣어야 함
  - 각 사업자는 다른 인프라 보유
  - 제품을 변경하지 않고 모든 다른 인프라에서 동일한 제품 실행하고자 함
- k8s의 라벨 기반 스케줄링 -> 아키텍처 확장, 안정적
- 5G 진출 가능
- 인프라와 애플리케이션 계층 분리 -> 시스템 의존성이 적으며, 애플리케이션 계층에서 기능 구현이 더 쉬워짐
- 서로 다른 환경에서 동일한 테스트 여러번 진행할 필요X -> 제품 출시 시간 절약