BESPIN GLOBAL

Kubernetes 핵심운영(2)

베스핀아카데미 이성미(seongmi.lee@bespinglobal.com)

001. Kubernetes Pod

- Pods 운영
- Label & Annotation
- Node Label

002. Kubernetes Controller

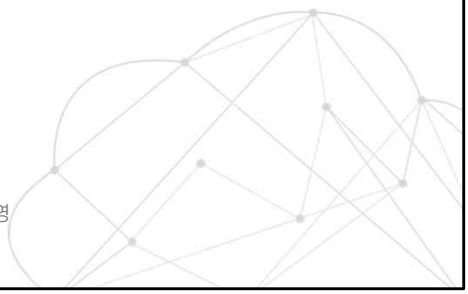
- ReplicationController, ReplicaSet, Deployment
- DaemonSet, StatefulSet
- Job & CronJob

003. Kubernetes Service

- CNI 이해
- Kubernetes Service
- Kube-proxy

004. Kubernetes Ingress

- Ingress의 이해
- Ingress를 이용한 웹 기반 서비스 운영



Pod 운영

Pod 란?

- 컨테이너를 표현하는 k8s API의 최소 단위
- 하나 또는 여러 개의 컨테이너의 그룹
  - 스토리지 및 네트워크를 공유
  - 해당 컨테이너를 구동하는 방식에 대한 명세를 가진다.

Pod : appjs
IP : 10.42.0.2

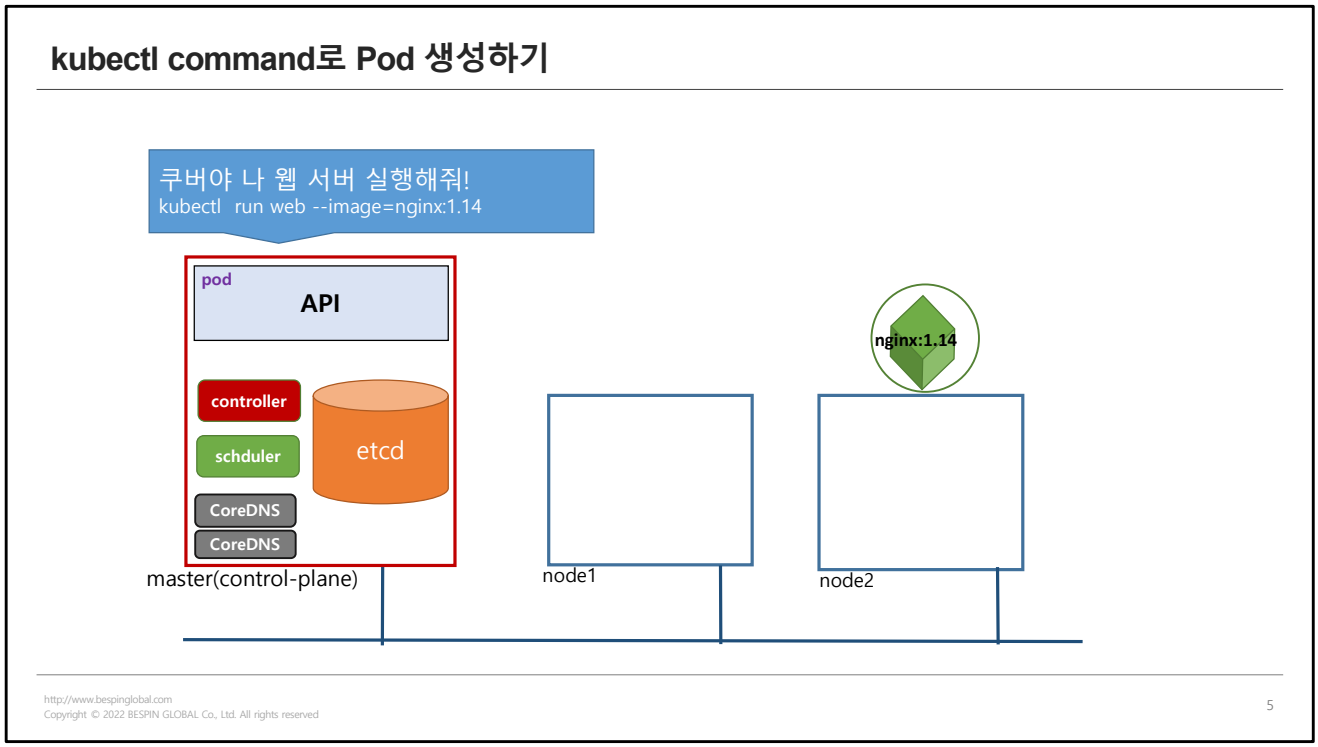
Pod : webserver
IP : 10.47.0.2

Pod : web-pod
IP : 10.36.0.2

Pod란?

The pod is a group of one or more containers and the smallest deployable unit in Kubernetes.

Each pod is isolated by the following Linux namespaces:
- Process ID(PID) namespace
- User namespace
- Mount namespace
- Interprocess Communication (IPC) namespace
- **Unix Time Sharing (UTS) namespace**
- **Network namespace**



kubectl command로 Pod 생성하기

포드에서 특정 이미지를 생성하고 실행

kubectl run NAME --image=image [--env="key=value"] [--port=port] [--dry-run=server|client] -- [COMMAND] [args...]

1. nginx 웹서버 컨테이너를 pod로 동작시키기

- pod name: web-pod
- image : nginx:1.14
- port : 80

 # kubectl run web --image=nginx:1.14 --port 80
pod/web created

2. 동작 중인 Pod 확인

 # kubectl get pods

```
NAME          READY   STATUS    RESTARTS   AGE
web           1/1     Running   0          7s
```

Pod 정보 자세히 보기

 # kubectl get pods -o wide

```
NAME          READY   STATUS    RESTARTS   AGE   IP           NODE
web           1/1     Running   0          36s   10.46.0.0    node2.example.com
```

3. 컨테이너로 동작하는 웹서버 접속

 # curl 10.46.0.0

```
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
...
</body>
</html>
```

4. Pod 삭제

 # kubectl delete pod web

Pod Template으로 Pod 실행

- 파드템플릿(PodTemplate)은 파드를 생성하기 위한 명세이다.
- Pod Template은 yaml 형식이나 json 형식을 가진다.
- Pod Template 을 사용하여 실제 파드가 동작된다.

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - image: nginx:1.14
    name: webserver
    ports:
    - containerPort: 80
```


Pod Template으로 Pod 실행

kubectl command로 Pod Templeate 만들기

dry-run : 실제 Pod를 실행하지 않고 실행 여부를 화면에 출력

```
# kubectl run webserver --image=nginx:1.14 --dry-run=client
```

-o yaml : 실제 Pod를 실행하지 않고 실행 여부를 확인하고 pod API를 yaml 파일 포맷으로 출력

```
# kubectl run webserver --image=nginx:1.14 --port=80 --dry-run=client -o yaml
```

실제 Pod를 실행하지 않고 실행 여부를 확인하고 pod API를 yaml 파일 포맷으로 출력한 결과를 yaml 파일에 저장

```
# kubectl run webserver --image=nginx:1.14 --port=80 --dry-run=client -o yaml > pod-nginx.yaml
```

```
# cat pod-nginx.yaml
  apiVersion: v1
  kind: Pod
  metadata:
    creationTimestamp: null
    labels:
      run: webserver
    name: webserver
  spec:
    containers:
    - image: nginx:1.14
      name: webserver
      ports:
      - containerPort: 80
      resources: {}
    dnsPolicy: ClusterFirst
    restartPolicy: Always
  status: {}
```

Pod 관리 Commands

- 동작중인 파드 정보 보기

  $ kubectl get pods

  $ kubectl describe pod webserver

- 동작중인 파드 수정

  $ kubectl edit pod webserver

- 동작중인 파드 삭제

  $ kubectl delete pod webserver

- 파드내 컨테이너 로그보기

  $ kubectl logs webserver

- 동작중인 파드로 연결

  $ kubectl exec -it webserver -- /bin/bash

Pod 관리 명령어

1. yaml 파일을 통해 Pod 실행하기

```
# kubectl apply -f  pod-nginx.yaml
pod/webserver created
```

동작중인 pod확인

```
# kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
webserver   1/1     Running   0          20s
web         1/1     Running   0          69m
```

pod 정보 자세히 보기

```
# kubectl get pods -o wide
NAME     READY   STATUS    RESTARTS   AGE   IP           NODE                NOMINATED NODE   READINESS GATES
webserver  1/1     Running   0          23s   10.36.0.0    node1.example.com   <none>           <none>
```

2. 애플리케이션 동작 확인하기

```
# curl 10.36.0.0
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
...
</body>
</html>
```

3. Pod 정보 자세히 보기

```
# kubectl describe pod webserver
Name:             webserver
Namespace:        default
Priority:         0
Node:             node1.example.com/10.0.0.11
Start Time:       Tue, 05 Oct 2021 17:31:32 +0900
Labels:           run=webserver
Annotations:      <none>
Status:           Running
```

```
IP:           10.36.0.0
IPs:
  IP:  10.36.0.0
Containers:
  nginx:
    Container ID:   docker://05c0fbc7df235b03e89771cecb3cf23620987a6853b4e33b569a8f7782cf01a0
    Image:          nginx:1.14
...
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  6s    default-scheduler  Successfully assigned default/webserver to node2.example.com
  Normal  Pulled     3s    kubelet            Container image "nginx:1.14" already present on machine
  Normal  Created    3s    kubelet            Created container webserver
  Normal  Started    3s    kubelet            Started container webserver
```

4. Pod 편집하기

```
# kubectl edit pod webserver
spec:
  containers:
  - image: nginx:1.15                    <- 1.14를 1.15로 컨테이너 수정후 <ESC>:wq
    imagePullPolicy: IfNotPresent
```

수정사항 적용되었는지 확인

```
# kubectl describe pod nginx
...
Events:
  Type    Reason                  Age                From               Message
  ----    ------                  ----               ----               -------
  Normal  Scheduled               101s               default-scheduler  Successfully assigned default/nginx to
node1.example.com
  Normal  Pulled                  100s               kubelet            Container image "nginx:1.14" already
present on machine
  Normal  Created                 9s (x2 over 100s)  kubelet            Created container webserver
  Normal  Killing                 9s                 kubelet            Container nginx definition changed, will
be restarted
  Normal  Pulled                  9s                 kubelet            Container image "nginx:1.15" already
present on machine
  Normal  Started                 8s (x2 over 100s)  kubelet            Started container webserver
```

5. log 정보보기

```
# kubectl logs webserver
10.32.0.0 - - [06/Oct/2021:06:56:35 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
10.32.0.0 - - [06/Oct/2021:06:57:55 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
```

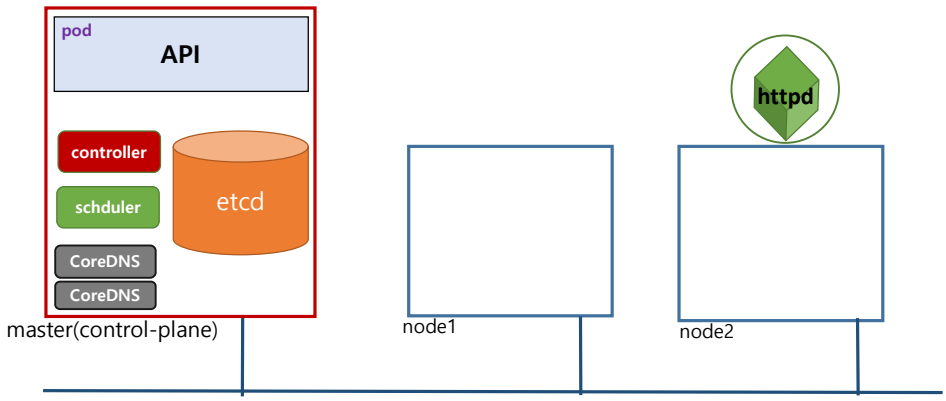
6. 컨테이너에 접속해서 웹페이지(index.html)수정하기

```
# kubectl exec -it webserver -- /bin/bash
root@webserver:/# cd /usr/share/nginx/html/
root@webserver:/usr/share/nginx/html# echo "CLASS" > index.html
root@webserver:/usr/share/nginx/html# exit
```

7. Pod 삭제

```
# kubectl delete pod  webserver
```

심화 | Pod 실행하기



Pod 생성하기

1. kubectl command를 이용해서 apache 웹서버 Pod를 실행하시오.
   - podname: apache
   - image: httpd
   - port: 80

2. 위에서 생성한 apache pod의 템플릿을 pod-apache.yaml로 생성하시오.

3. 동작되고 있는 apache POD를 삭제합니다.

4. 다음과 같은 Pod를 실행하는 pod template을 생성한후 실행하시오.
   - filename: pod-apache.yaml
   - podname: apache
   - image: httpd
   - port: 80
   - container name: httpd

5. 동작중인 apache pod의 IP ADDRESS는 몇인가?

   해당 IPAddress로 curl 명령을 실행해본다.

6. apache pod는 어느 노드에서 실행중인가?

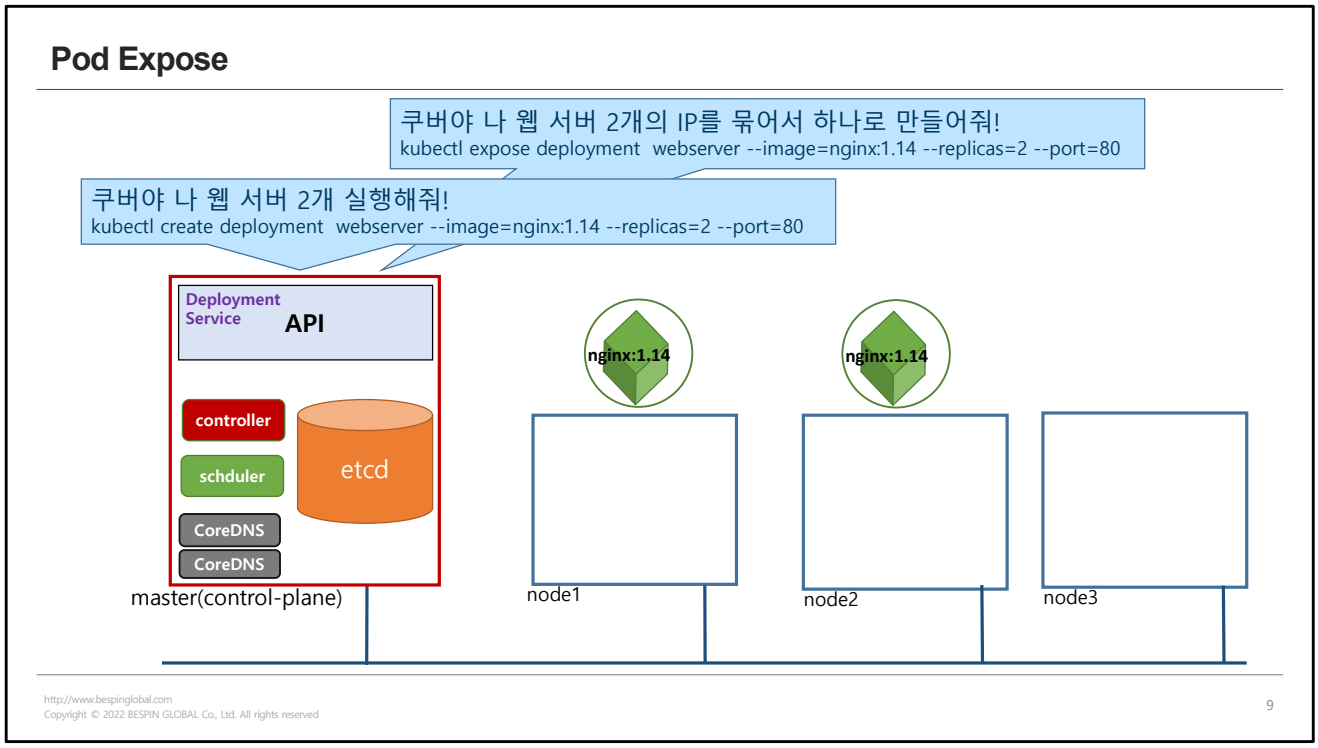
7. apache Pod의 로그를 확인해보자.

8. apache Pod의 web 컨테이너로 연결해서 /var/www/html/index.html 문서를 아래와 같이 수정한다.

```
echo "Learn-Do-Share" >  /usr/local/apache2/htdocs/index.html
```

```
# curl 10.36.0.1
Learn-Do-Share
```

9. 동작중인 Pod를 제거합시다.



Pod Expose

1. Deployment로 여러 개의 웹서버 실행
   동일한 애플리케이션을 여러 개 실행가능
   # kubectl create deployment webserver --image=nginx:1.14 --replicas=2 --port=80

   동작중인 Pod 확인
   # kubectl get deployments.apps

```
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
webserver   2/2     2            2           6m49s
```

   # kubectl get pods

```
NAME                                 READY   STATUS    RESTARTS   AGE
webserver-67db49c9d4-4c4gv           1/1     Running   0          6m45s
webserver-67db49c9d4-ljt7s           1/1     Running   0          6m45s
```

   Pod의 IP 확인
   # kubectl get pods -o wide

```
NAME                                 READY   STATUS    RESTARTS   AGE     IP           NODE
webserver-67db49c9d4-4c4gv           1/1     Running   0          116m    10.36.0.2    node1.example.com
webserver-67db49c9d4-ljt7s           1/1     Running   0          116m    10.46.0.1    node2.example.com
```

   문제: nginx 파드는 총몇개 입니까?

   문제: 여러 개의 Pod가 운영했을때 좋은점은 무엇입니까?

2. 단일 진입점 만들기 : expose
   # kubectl expose deployment webserver --port=80
   service/webserver exposed

   # kubectl get service

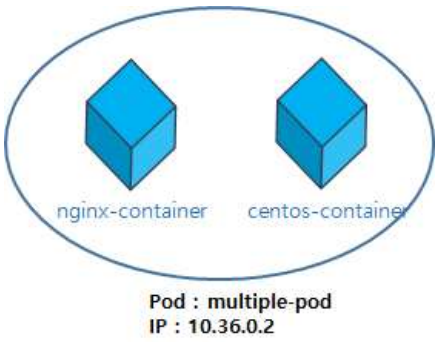
```
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   3d3h
webserver    ClusterIP   10.104.203.161   <none>        80/TCP    64s
```

3. 서비스 접속해보기

 # curl 10.104.203.161

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

multiple-container Pod




```
apiVersion: v1
kind: Pod
metadata:
  name: multiple-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:1.14
    ports:
    - containerPort: 80
  - name: centos-container
    image: centos:7
    command:
    - sleep
    - "10000"
```


multiple-container Pod

하나의 Pod에는 여러 개의 컨테이너가 포함될수 있다.
하나의 Pod에 여러 개의 컨테이너를 포함하는 경우 모든 컨테이너가 항상 단일 Node에서 실행된다.
보통 multiple-container Pod에 포함된 컨테이너들은 밀접하게 관련된 프로세스를 함께 실행하거나 동일한 IP 및 port를 사용한다.

실습: multiple-container Pod를 실행해보자.

1. multiple-container Pod 템플릿을 확인하자.

 # cat pod-multiple-container.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: multiple-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:1.14
    ports:
    - containerPort: 80
  - name: centos-container
    image: centos:7
    command:
    - sleep
    - "10000"
```

몇 개의 컨테이너를 실행하는가?

각 컨테이너는 어떤 이미지를 사용하는가?

두 컨테이너 간의 연관성은 무엇인가?

2. multiple-pod 컨테이너를 동작해보자.

 # kubectl apply -f pod-multiple-container.yaml

 # kubectl get pods

```
NAME                READY   STATUS    RESTARTS   AGE
multiple-pod        2/2     Running   0          55m
```

3. pod의 세부정보를 확인하자.

 # kubectl describe pod multiple-pod

```
Name:           multiple-pod
Namespace:      default
Priority:       0
Node:           node2.example.com/10.0.0.12
Start Time:     Tue, 05 Oct 2021 19:19:07 +0900
Labels:         <none>
Annotations:    <none>
Status:         Running
IP:             10.46.0.0
IPs:
  IP:  10.46.0.0
Containers:
  nginx-container:
    Container ID:   docker://b2bb7c0d25463066e03ea38976916869212d0450c9976554c3af09e0d9c60f44
    Image:          nginx:1.14
...
 centos-container:
    Container ID:  docker://ed38d18f93f94a67526ae834a40b5682e998f201840c337cd3b29584e785c268
    Image:         centos:7
...
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  57m   default-scheduler  Successfully assigned default/multiple-pod to node2.example.com
  Normal  Pulled     57m   kubelet            Container image "nginx:1.14" already present on machine
  Normal  Created    57m   kubelet            Created container nginx-container
  Normal  Started    57m   kubelet            Started container nginx-container
  Normal  Pulling    57m   kubelet            Pulling image "centos:7"
  Normal  Pulled     57m   kubelet            Successfully pulled image "centos:7" in 22.384794599s
  Normal  Created    57m   kubelet            Created container centos-container
  Normal  Started    57m   kubelet            Started container centos-container
```

multiple-pod의 컨테이너가 확인되었나?

Events 정보를 통해서 무엇을 확인할수 있나?

4. Pod를 삭제

kubectl delete pod multiple-pod

심화 | multiple-container pod

생각해보기

1. 어떤 container들을 하나의 Pod 넣으면 좋을까?

2. 어떤 container들을 하나에 Pod 넣으면 안될까?

Label & Annotation

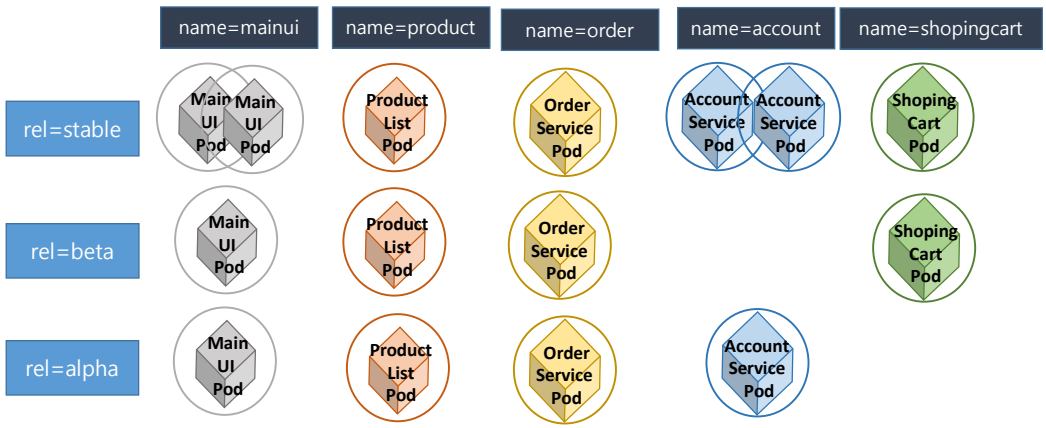
Label

- Node를 포함하여 pod, deployment 등 모든 리소스에 할당
- 리소스의 특성을 분류하고, Selector를 이용해서 선택
- Key-value 한쌍으로 적용

Pod의 Label

MSA 구조에서는 포드의 수는 빠르게 증가하고 이로 인해 시스템에 수백 개의 포드가 생길 수 있다.
이것들을 조직하는 메커니즘이 없다면 아래 그림처럼 이해할 수 없고 복잡한 구조가 될 것이다. 특정 기준에 따라 작은 그룹으로 구성할 필요가 있는데, 쿠버네티스는 label을 이용해 그룹을 설정한다.

Label과 Selector



Pod의 Label과 Selector

LABEL

- Label은 Pod와 같은 오브젝트에 첨부된 Key와 Value의 쌍이다.
- Label은 오브젝트의 특성을 식별하는 데 사용되어 사용자에게 중요하지만, 코어 시스템에 직접적인 의미는 없다.
- Label로 오브젝트의 하위 집합을 선택하고, 구성하는데 사용할 수 있다.
- Label은 오브젝트를 생성할 때에 붙이거나 생성 이후에 붙이거나 언제든지 수정이 가능하다.
- 오브젝트마다 Key와 Value로 Label을 정의할 수 있다. 오브젝트의 Key는 고유한 값이어야 한다.

```
metadata:
  labels:
    key1: value1
    key2: value2
```

Label & Selector

- Label

```
metadata:
  labels:
    rel: stable
    name: mainui
```

- Selector

```
selector:
  matchLabels:
    key: value
  matchExpressions:
    - {key: name, operator: In, values: [manui]}
    - {key: rel, operator: NotIn, values: [“beta”,”canary”]}
```


Label & Label Selector

Label

- 63자 이하여야 하고 (공백일 수도 있음),
- (공백이 아니라면)시작과 끝은 알파벳과 숫자([a-z0-9A-Z])이며, 알파벳과 숫자, 대시(-), 밑줄(_), 점(.)을 중간에 포함할 수 있다.

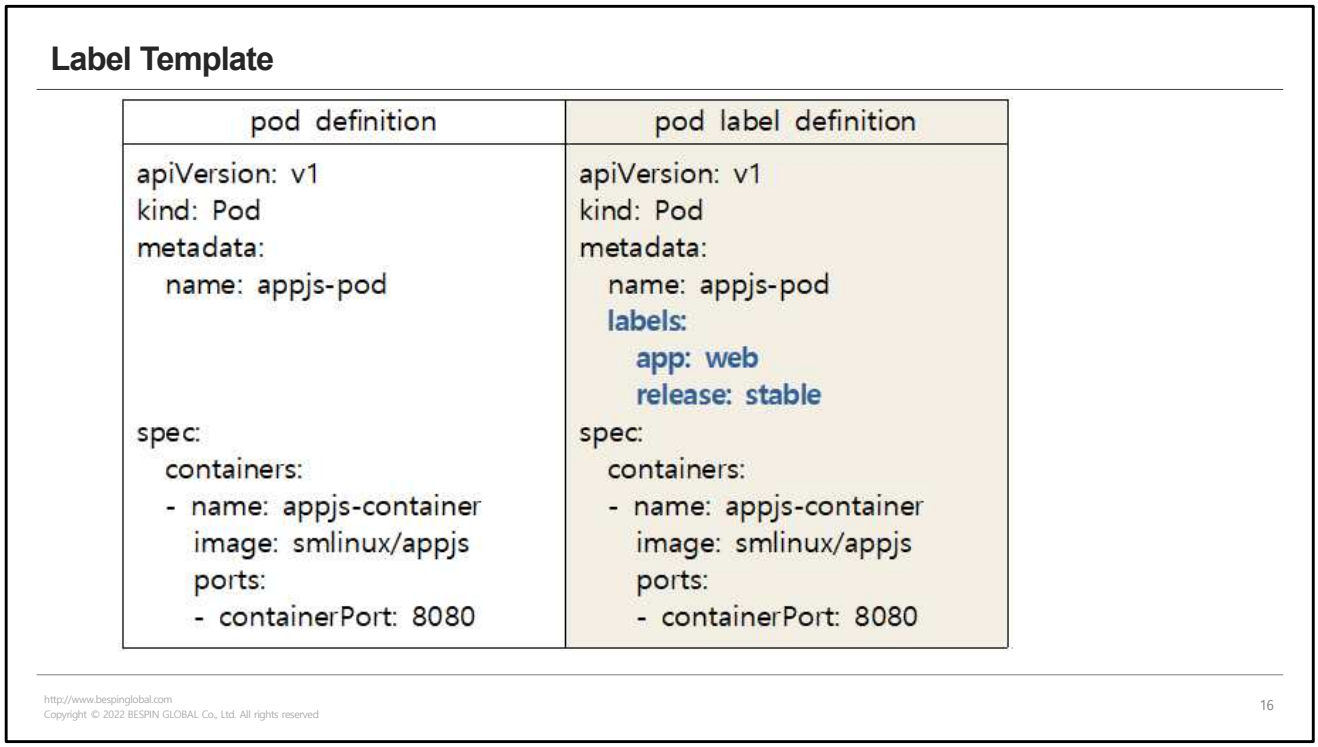
레이블 예시
다음 예시는 일반적으로 사용하는 레이블이며, 사용자는 자신만의 규칙(convention)에 따라 자유롭게 개발할 수 있다.
오브젝트에 붙여진 레이블 Key는 고유해야 한다는 것을 기억해야 한다.

- "release: stable, "release" : "canary"
- "environment" : "dev", "environment" : "qa", "environment" : "production"
- "tier" : "frontend", "tier" : "backend", "tier" : "cache"
- "partition" : "customerA", "partition" : "customerB"
- "track" : "daily", "track" : "weekly"

Label Selector
name과 UID와 다르게 label은 고유하지 않다. 일반적으로 많은 종류의 API에 같은 Label을 할당할것이다.
Label Selector를 통해 클라이언트와 사용자는 오브젝트를 식별할 수 있다. Label Selector는 쿠버네티스 코어 그룹의 기본이다.
집합성 기준 레이블 요건에 따라 값 집합을 키로 필터링할 수 있다. in,notin과 exists(키 식별자만 해당)의 3개의 연산자를 지원한다. 예를 들면,

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

key가 environment이고 값이 production 또는 qa인 모든 리소스를 선택
key가 tier이고 값이 frontend와 backend를 가지는 리소스를 제외한 모든 리소스와 키로 tier를 가지고 값을 공백으로 가지는 모든 리소스를 선택
레이블의 값에 상관없이 키가 partition을 포함하는 모든 리소스를 선택
레이블의 값에 상관없이 키가 partition을 포함하지 않는 모든 리소스를 선택



Label Template

Label은 모든 API 리소스에 할당할수 있다. 간단하게 Pod에 레이블을 할당해보자.

1. 기존의 nginx pod 파일을 "pod-nginx-label.yaml" 로 복사한후 label을 설정해서 실행해보자.

```
# cp pod-appjs.yaml pod-label.yaml

# vi pod-label.yaml
apiVersion: v1
kind: Pod
metadata:
  name: appjs-pod-1
  labels:
    app: web
    release: stable
spec:
  containers:
  - name: appjs-container
    image: smlinux/appjs
    ports:
    - containerPort: 8080
```

2. 생성한 pod-label.yaml 템플릿을 실행한다.

```
# kubectl apply -f pod-label.yaml
pod/appjs-pod-1 created
```

3. label을 설정한 pod와 설정하지 않은 pod를 구분해보자.

```
# kubectl get pods
NAME          READY              STATUS    RESTARTS   AGE
appjs-pod     1/1                Running   0          3m20s
appjs-pod-1   1/1                Running   0          12m

# kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE     LABELS
appjs-pod       1/1     Running     0          3m26s   <none>
appjs-pod-1     1/1     Running     0          12m     app=web,release=stable
```

Label 관리 및 실습

- Label 보기
  kubectl get pods --show-labels
  kubectl get pods -l <label_name>

- Label 관리 : kubectl label --help
  - Label 생성 및 변경
    kubectl label pod <name> key=value
    kubectl label pod <name> key=value --overwrite
  - Label 확인
    - kubectl label pod <name> --show-labels
  - Label 제거
    kubectl label pod <name> key-

Label 관리

1. label이 설정된 pod를 생성하고 실행해보자.

```
# cat pod-nginx label.yaml
apiVersion: v1
kind: Pod
metadata:
  name: examlabel
  labels:
    app: web
    type: frontend
    name: nginx
spec:
  containers:
    - name: nginx-container
      image: nginx:1.14
```

```
# kubectl create -f pod-label.yaml
```

2. You can see them by using the --show-labels switch:

```
# kubectl get pod
```

```
# kubectl get pod --show-labels
```

3. If you're only interested in certain labels, you can specify them with the -L switch.

```
# kubectl get pod -L app
```

4. Labels can also be added to and modified on existing pods:

```
# kubectl label pod redis-pod app=db
```

```
# kubectl label pod nginx-pod app=web-services type=frontend
```

```
# kubectl label po label-pod app=web-services --overwrite
```

```
# kubectl get pod --show-labels
```

```
# kubectl get pod -L app,type

# kubectl get pod --selector app=db
```

Annotation

- Label과 동일하게 **key-value**를 통해 리소스의 특성을 기록
- Kubernetes 에게 특정 정보 전달할 용도로 사용
- 예를 들어 Deployment의 rolling update 정보 기록

```
annotations:
  kubernetes.io/change-cause: version 1.15
```

- 관리를 위해 필요한 정보를 기록할 용도로 사용
- 릴리즈, 로깅, 모니터링에 필요한 정보들을 기록

```
annotations:
  builder: “seongmi Lee (seongmi.lee@bespinglobal.com)"
  buildDate: "20211002"
  imageRegistry: https://hub.docker.com/
```


Annotation

Annotaiton은 Label과는 달리 오브젝트를 식별하여 select 하지 않음.
Annotation도 Label과 같이 key/value로 구성한다.

```
metadata:
  annotations:
    key1: value1
    key2: value2
```

- 필드는 선언적 구성 계층에 의해 관리된다. 이러한 필드를 어노테이션으로 첨부하는 것은 클라이언트 또는 서버가 설정한 기본 값, 자동 생성된 필드, 그리고 오토사이징 또는 오토스케일링 시스템에 의해 설정된 필드와 구분된다.
- 빌드, 릴리스, 또는 타임 스탬프, 릴리스 ID, git 브랜치, PR 번호, 이미지 해시 및 레지스트리 주소와 같은 이미지 정보.
- 로깅, 모니터링, 분석 또는 감사 리포지터리에 대한 포인터.
- 디버깅 목적으로 사용될 수 있는 클라이언트 라이브러리 또는 도구 정보: 예를 들면, 이름, 버전, 그리고 빌드 정보.
- 다른 생태계 구성 요소의 관련 오브젝트 URL과 같은 사용자 또는 도구/시스템 출처 정보.
- 경량 롤아웃 도구 메타데이터. 예: 구성 또는 체크포인트
- 책임자의 전화번호 또는 호출기 번호, 또는 팀 웹 사이트 같은 해당 정보를 찾을 수 있는 디렉터리 진입점.
- 행동을 수정하거나 비표준 기능을 수행하기 위한 최종 사용자의 지시 사항.
- 어노테이션을 사용하는 대신, 이 유형의 정보를 외부 데이터베이스 또는 디렉터리에 저장할 수 있지만, 이는 배포, 관리, 인트로스펙션(introspection) 등을 위한 공유 클라이언트 라이브러리와 도구 생성을 훨씬 더 어렵게 만들 수 있다.

레이블과 Annotation 관련 정보 정리:
https://www.replex.io/blog/9-best-practices-and-examples-for-working-with-kubernetes-labels

Annotation 구성 실습

1. pod에 anonotation을 설정해서 확인해보자.

```
# cp pod-appjs.yaml  pod-annotation.yaml
# vi pod-annotation.yaml
apiVersion: v1
kind: Pod
metadata:
  name: appjs-pod-an
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: appjs-container
    image: smlinux/appjs
    ports:
    - containerPort: 8080

# kubectl apply -f pod-annotation.yaml
pod/appjs-pod-an created
```

2. 설정된 Annotation을 확인해보자.

```
# kubectl describe pod appjs-pod-an

# kubectl describe pod appjs-pod-an  | grep -i ann
Annotations:  imageregistry: https://hub.docker.com/
```

Node Label

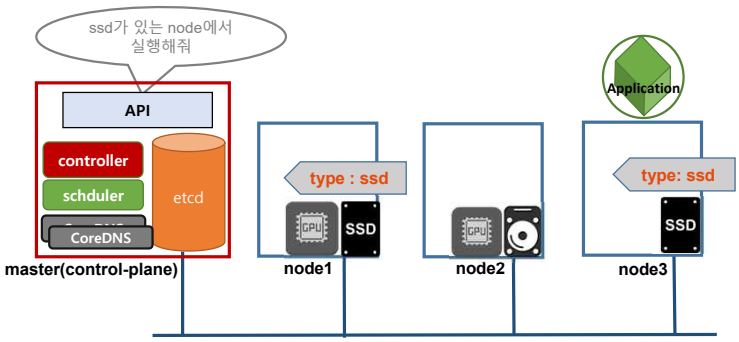
Node Label

- Worker node에 할당된 label을 이용해 node를 선택
- node Label 설정

  kubectl label nodes <노드 이름> <레이블 키>=<레이블 값>

  kubectl label nodes node1.example.com type=ssd

  kubectl get nodes -L type



Node Label 관리명령

- worker node를 구분할 용도로 레이블을 할당

  kubectl label nodes <노드이름> key=vaule

- node에 할당된 label 보기

  kubectl get nodes –show-labels

  kubectl get nodes -L <label_name>

- node label 관리
  - node label 생성 및 변경

    kubectl label nodes <name> key=value

    kubectl label nodes <name> key=value –overwrite

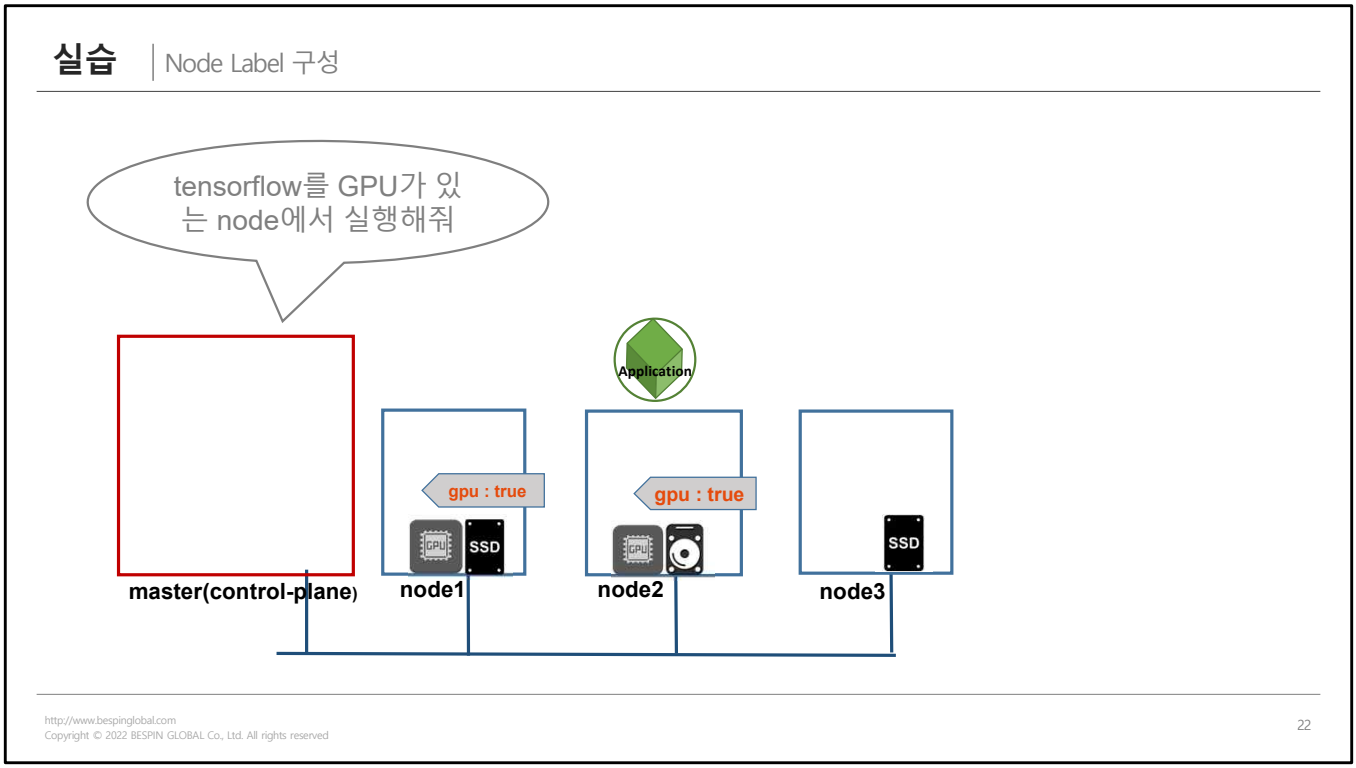
  - node label 제거

    kubectl label nodes <name> key-

- nodeSelector

  label이 할당된 node는 nodeSelector를 통해 선택된다.

```
apiVersion: v1
kind: Pod
metadata:
 name: testpod
spec:
 containers:
 - name: nginx
   image: nginx:1.14
 nodeSelector:
   type: ssd
```



Node에 LABEL 설정

TensorFlow는 머신러닝을 위한 오픈소스 소프트웨어 라이브러리이다. Tenserflow를 이용해 ML 연산이나 딥러닝을 구현해야 한다면 반드시 병렬처리를 하는 GPU가 있어야 한다.
간단하게 nodeSelector를 이용해서 gpu가 있는 node에서만 Tenserflow 컨테이너가 동작되도록 구현해보자.

1. GPU 가 있는 node에 node label을 설정한다.
```
# kubectl label nodes  node1.example.com gpu=true
# kubectl label nodes  node2.example.com gpu=true

# kubectl get nodes -L gpu
```

2. tenserflow pod가 gpu가 있는 node에서 실행되도록 nodeSelector를 구성한다.
```
# cat tensorflow.yaml
apiVersion: v1
kind: Pod
metadata:
  name: tensorflow-gpu
spec:
  containers:
  - name: tensorflow
    image: tensorflow/tensorflow:nightly-jupyter
    ports:
    - containerPort: 8888
      protocol: TCP
  nodeSelector:
    gpu: "true"
```

3. pod 실행결과 node2 또는 node3에서 실행되었는지 확인해보자.
```
# kubectl apply -f tensorflow.yaml

# kubectl get pod -o wide
```

4. 결과 확인 후 pod를 삭제한다.
```
# kubectl delete pod tensorflow-gpu
```

Pod Resources 제한 및 관리

Pod Resources 제한 및 관리 | Pod Resource 요청 및 제한

- Pod Resource 요청 및 제한
- Resource Requests
  - Pod를 실행하기 위한 최소 리소스 양을 지정
- Resource Limits
  - Pod가 사용할 수 있는 최대 리소스 양을 지정

Pod Resources 단위
Pod의 CPU 사용량은 CPU 단위로 표시된다. Kubernetes CPU는 AWS의 vCPU, GCP의 core, Azure의 vCore, Hyper-Threading을 지원하는 Bare Metal 프로세서의 하이퍼스레드와 동일하다. 즉 쿠버네티스 용어로 1CPU는 일반적인 CPU 단위와 같다.
대부분의 Pod는 CPU 전체가 필요하지 않기 때문에 요청과 상한은 millicpus (또는 millicores)로 표시된다. 메모리는 바이트나 mebibytes(MiB)로 측정된다.
1000m (milicores) = 1 core = 1 CPU = 1 AWS vCPU = 1 GCP Core.
100m (milicores) = 0.1 core = 0.1 CPU = 0.1 AWS vCPU = 0.1 GCP Core.

Resource Requests
리소스 요청에서는 Pod를 실행하기 위한 최소 리소스 양을 지정한다.
예를 들어 CPU 100m(100 millicpus)와 메모리 250Mi(250 MiB)로 리소스를 요청하면, Pod는 이보다 적은 리소스를 가진 노드에는 스케줄링 되지 않는다. 만약 사용할 수 있는 용량이 충분한 노드가 없으면, Pod는 여유 용량이 확보될 때까지 대기(pending) 상태로 기다린다.

Resource Limits
리소스 limits은 Pod가 사용할 수 있는 최대 리소스 양을 지정한다.
Container가 limit을 초과하여 CPU를 사용하려고 하면, 해당 Pod는 제한되고 성능저하가 발생한다.
Memory limit을 초과해서 사용되는 container는 종료(OOM Kill)되고 이후 다시 running된다. 실제로는 동일한 노드에서 컨테이너가 단순히 다시 실행되는 것이다.
네트워크 서버와 같은 애플리케이션은 수요가 증가함에 따라 더 많은 리소스를 사용할 수 있다. 리소스 상한(limit)을 지정해두면 이러한 파드가 클러스터 용량을 과다하게 점유하는 것을 막을 수 있다.

Pod Resources 요청

- Pod Resource Requests

Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod
spec:
  containers:
    - name: nginx-container
      image: nginx:1.14
      resources:
        requests:
          cpu: 100m
          memory: 250Mi
```


실습: Pod Resource Requests

1. worker node는 제한된 하드웨어를 포함하고 있다. 각 worker node의 리소스 정보 및 사용량을 확인하기 위해 다음과 같은 명령을 실행해 보자.

 # kubectl describe nodes <node 이름>

2. pod에서 resource request는 Pod가 해당 하드웨어 리소스를 요구하는 것이다.
요구에 맞는 리소를 가진 node가 없다면 Pod는 동작되지 않고 pending으로 머물게 된다.
아래 예제는 2core CPU를 요청하고 있다. 모든 worker node는 기본 메모리가 2core이고, 이미 리소스는 사용중이기 때문에 아래의 pod는 실행되지 않을 것이다.

 # cp pod-nginx.yaml pod-nginx-request.yaml

 # cat pod-nginx-request.yaml

```
...
  containers:
    - name: nginx-container
      image: nginx:1.14
      resources:
        requests:
          cpu: 2
```

 # kubectl create -f pod-nginx-request.yaml

3. pod 배치할 worker node가 없을 때 다음과같이 pending 상태로 머문다.

 # kubectl get pods

```
NAME                READY   STATUS    RESTARTS   AGE
nginx-pod-request   0/1     Pending   0          19s
```

4. 해당 Pod를 종료하고 yaml 파일을 수정하여 cpu request를 1core 로 변경하자.

 # kubectl delete pod nginx-pod-request
 # vi pod-nginx-request.yaml

```
..
      resources:
        requests:
          cpu: 1
```

 # kubectl create -f pod-nginx-request.yaml

 # kubectl get pods

```
# kubectl describe pod nginx-pod-request
    Requests:
      cpu:        1
```

Pod Resources 제한

- Pod Resource Limits

Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod
spec:
  containers:
    - name: nginx-container
      image: nginx:1.14
      resources:
        requests:
          cpu: 100m
          memory: 250Mi
        limits:
          cpu: 500m
          memory: 500Mi
```


실습: Pod Resource Limits

1. Resource Limit을 설정하지 않은 Pod는 동작되는 Node의 컴퓨팅 리소스를 모두 사용할 수 있다.

```
# cat pod-nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:1.14
      ports:
      - containerPort: 80
        protocol: TCP
```

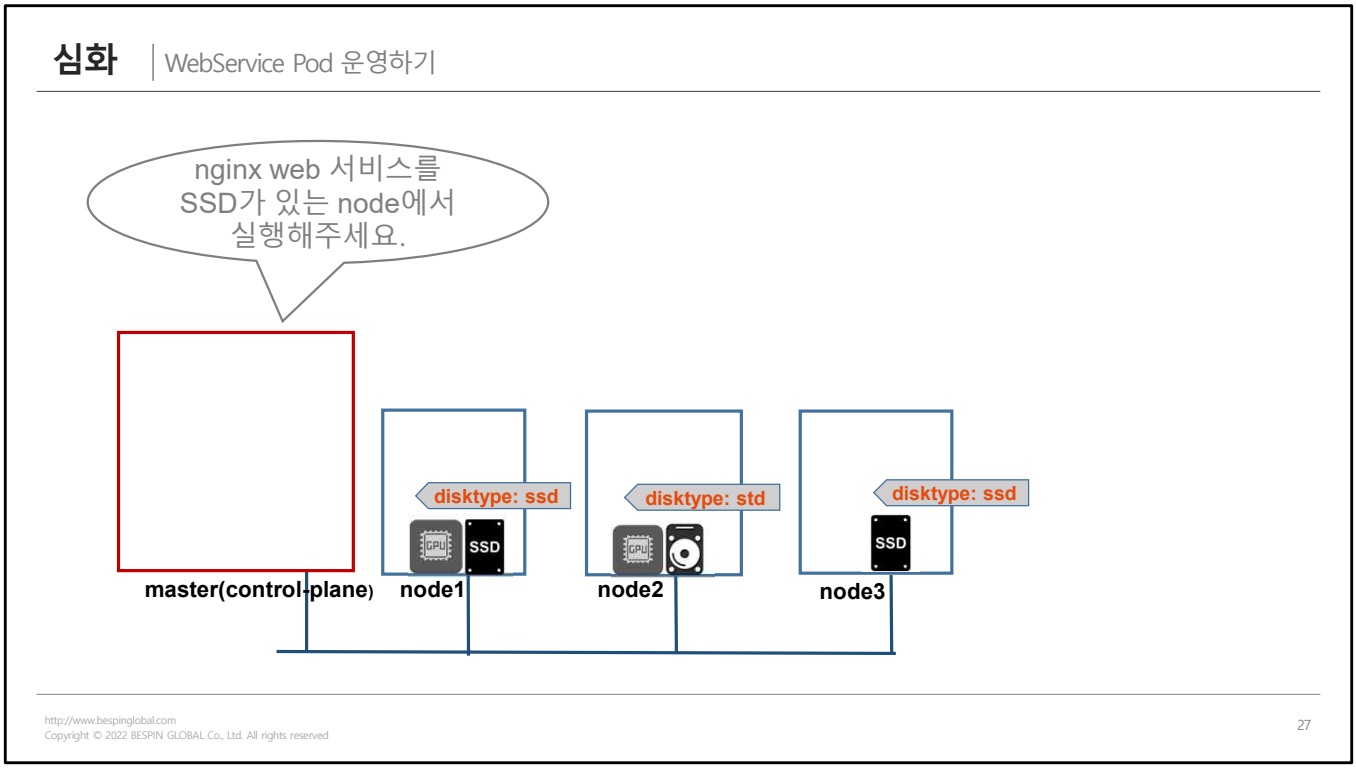
```
# kubectl create -f pod-nginx.yaml
# kubectl describe pod nginx-pod
```

2. pod-nginx.yaml을 아래와 같이 수정하여 다시 실행해보자.

```
# cp pod-nginx.yaml  limit-nginx.yaml
```

```
# vi limit-nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-limits
spec:
  containers:
  - name: nginx-container
    image: nginx:1.14
    resources:
      limits:
        cpu: 200m
        memory: 100Mi
    ports:
    - containerPort: 80
      protocol: TCP
```

```
# kubectl create -f limit-nginx.yaml
# kubectl get pods
# kubectl describe pod nginx-pod-limits
```



웹 서비스 Pod 동작 시키기

1. 위의 그림과 같이 Node에 label을 설정하시오.
   - node1.example.com : disktype: ssd
   - node2.example.com : disktype: std
   - node3.example.com : disktype: ssd

2. 다음 조건에 맞게 nginx web 서버를 실행하는 Pod template를 생성하고 실행하세요.

Pod name: webui

- Pod label:
  - app: web
  - tier: frontend
  - release: stable
- Node Selector
  - disktype: ssd
- Container name: web
- Image: nginx:1.14
- Resource Requests
  - CPU: 500m
  - Memory: 700Mi
- index.html : "Helping You Adopt Cloud"

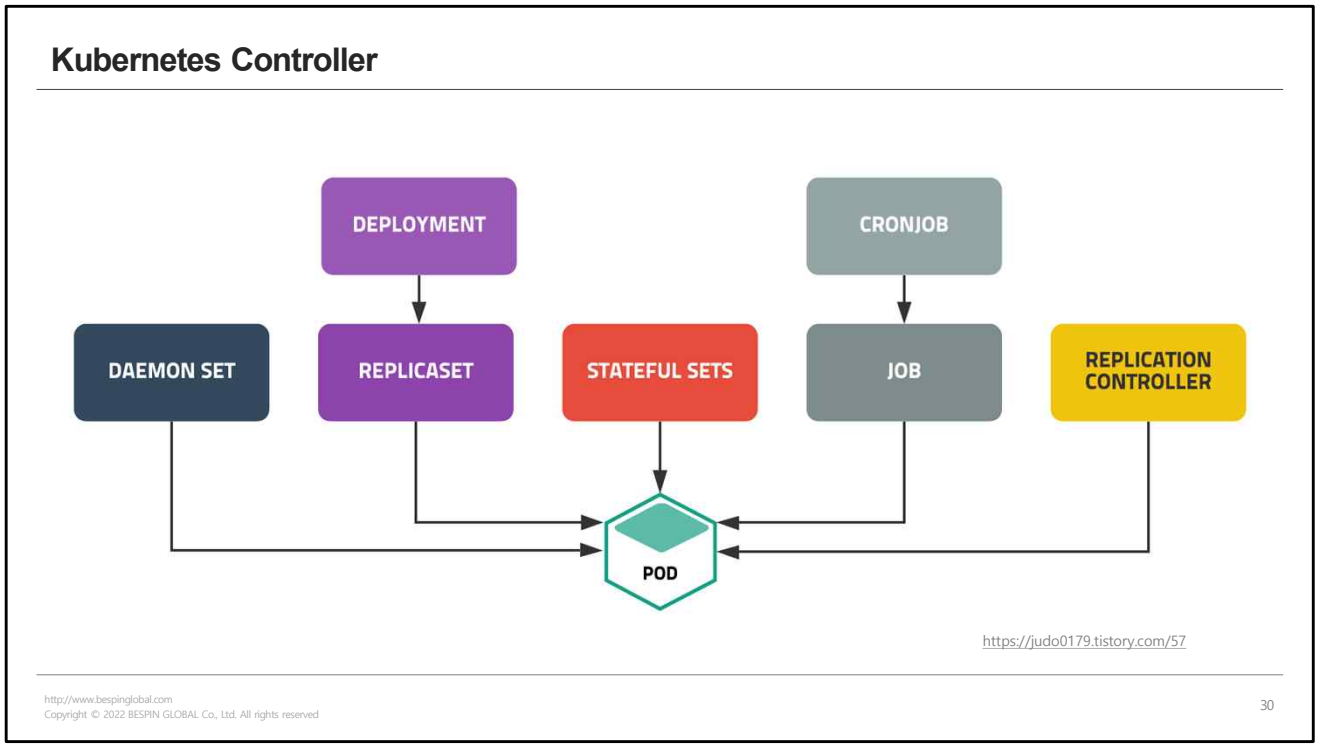
지금까지 배운 내용을 기준으로

- Application Pod
  - Pod와 컨테이너를 구분해서 설명할 수 있어야합니다.
  - kubectl command로 Pod를 생성할 수 있어야합니다.
  - Pod에 label을 붙일 수 있어야합니다.

- Pod Label & Selector
  - Node에 레이블을 붙이고 확인하고 삭제할 수 있어야합니다.
  - 원하는 node에 pod를 실행할 수 있어야합니다.

추가학습 : 아래 링크로 접속해서 label 작업을 진행 해보시오.
https://kubernetes.io/ko/docs/tasks/configure-pod-container/assign-pods-nodes/

Kubernetes Controller 동작방식



Kubernetes Controller

에어컨 온도기와 같은 control loop는 특정 온도를 설정하면 에어컨을 켜거나 끄기를 반복해서 해당온도를 맞춘다.
이런 control loop과 이처럼 control loop는 시스템 상태를 조절하는 종료되지 않는 loop이다.

Controller는 Pod를 관리하는 역할을 한다. control loop같은 역할을 수행하는데, 시스템이 아닌 Pod 실행하거나 종료시켜 원하는 서비스를 지원하는 것이다.
Kubernetes가 지원하는 컨트롤러는 목적에 따라 다양한 종류를 지원하고 있고, 애플리케이션 서비스에 따라 적절히 선택하여 사용할 수 있다.

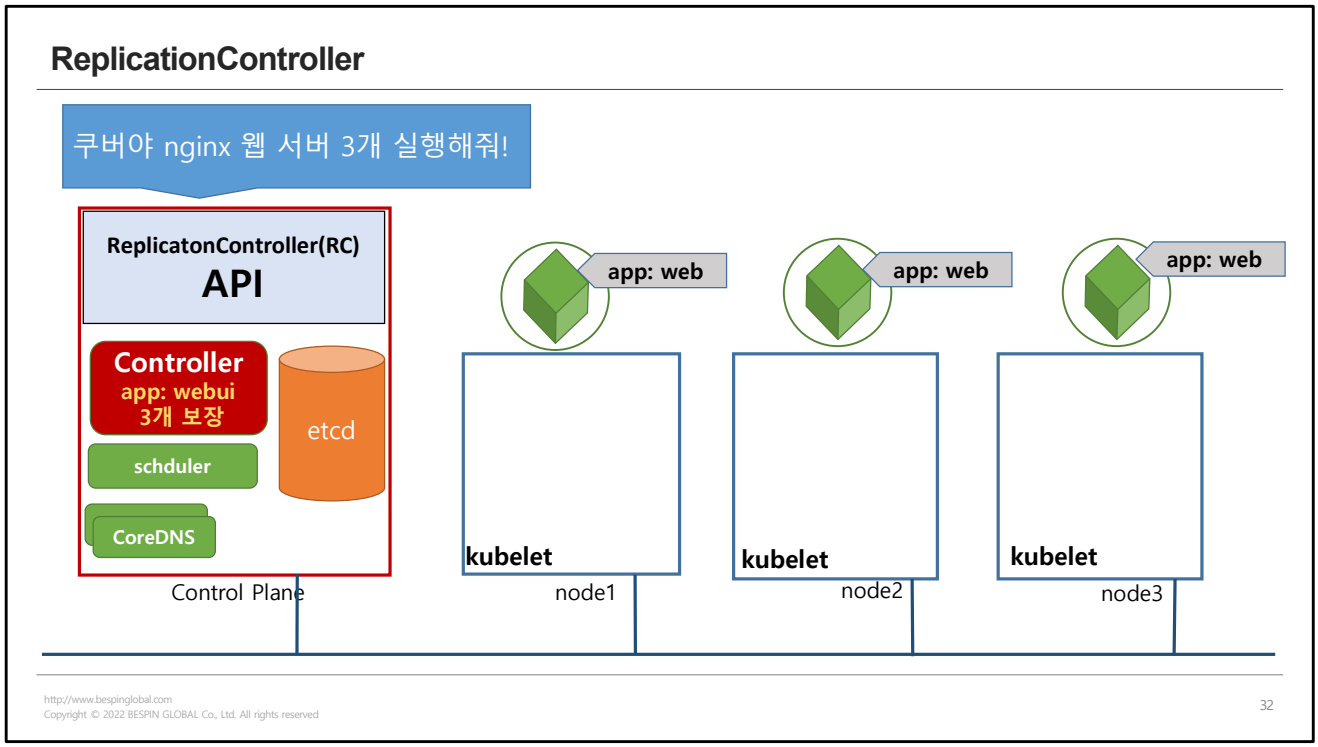
상태를 유지하지 않아도 되는 파드를 관리하는 경우
- 레플리케이션 컨트롤러(Relication Controller)
- 레플리카 세트(Replica Set)
- 디플로이먼트(Deployment)

클러스터 전체에 배포가 필요한 파드를 관리하는 경우
- 데몬 셋(Deamon Set)

상태관리가 필요한 파드를 관리하는 경우
- 스테이트풀세트(Stateful Set)

배치성 작업을 진행하는 파드를 관리하는 경우
- 잡(Job)
- 크론잡(Cronjob)

ReplicationController



Replication controller

ReplicationController는 Pod가 항상 실행되도록 유지하는 Kubernetes resource이다.
Node가 클러스터에서 사라지는 경우나 노드에서 Pod가 제거된 경우와 같이 어떤 이유로든 Pod가 사라지면 ReplicationController는 누락된 Pod를 감지하고 대체 Pod는 만든다.

Replication control manager

Pod의 개수를 관리
- current state 개수와 desired state 개수가 같을 때 까지 관리(실행 중인 Pod의 수를 항상 보장)
- pod template 을 통해 Pod가 부족하면 더 생성하고
- Pod가 많으면 종료시킨다.

ReplicationController Template

Pod-template	ReplicationController-template

apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
  app: web
spec:
  containers:
  - name: nginx-container
  image: nginx:1.14
	apiVersion: v1
	kind: ReplicationController
	metadata:
	  name: rc-nginx
	spec:
	  replicas: 3
	  selector:
	  app: web
	  template:
	  metadata:
	  name: nginx-pod
	  labels:
	  app: web
	  spec:
	  containers:
	  - name: nginx-container
	  image: nginx:1.14

ReplicationConroller Template

ReplicationController의 세 가지 요소

- label selector
- replica count
- pod template

ReplicationController Template

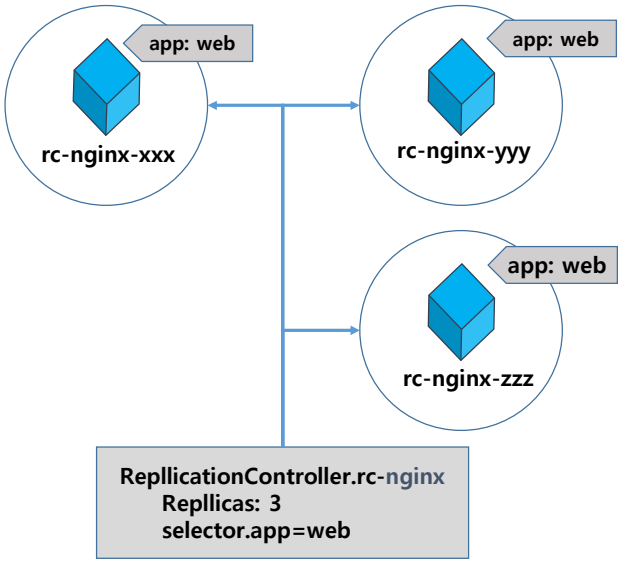
```
apiVersion: v1
kind: ReplicationController
metadata:
  name: <RC_이름>
spec:
  replicas: <배포갯수>
  selector:
    key: value
  template:
    <컨테이너 템플릿>
```

ReplicationController 운영

Controller를 통해 nginx Pod를 3개 동작

kubectl apply -f rc-nginx.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
 name: rc-nginx
spec:
 replicas: 3
 selector:
  app: web
 template:
  metadata:
   name: nginx-pod
   labels:
    app: web
  spec:
   containers:
   - name: nginx-container
    image: nginx:1.14
```



ReplicationController 운영

1. rc-nginx.yaml 파일을 확인해보자.

```
# cat rc-nginx.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: rc-nginx
spec:
  replicas: 3
  selector:
    app: web
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

실행되는 Pod수는 몇개인가?

실행되는 Pod 이름은?

실행되는 container 이미지는?

2. ReplicationController를 동작시키자.

```
# kubectl apply -f rc-nginx.yaml
replicationcontroller/rc-nginx created
```

3. 동작된 ReplicationController의 상태를 확인하자. 현재 요구하고 있는 Pod수, 실행중인 Pod수 등을 확인할 수 있다

```
# kubectl get replicationcontrollers
NAME       DESIRED   CURRENT   READY   AGE
rc-nginx   3         3         3       20s

# kubectl get rc
```

```
# kubectl describe rc rc-nginx
Name:         rc-nginx
Namespace:    default
Selector:     app=web
Labels:       app=web
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=web
  Containers:
   nginx-container:
    Image:        nginx:1.14
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  47s   replication-controller  Created pod: rc-nginx-kgj5z
  Normal  SuccessfulCreate  47s   replication-controller  Created pod: rc-nginx-l2jkp
  Normal  SuccessfulCreate  47s   replication-controller  Created pod: rc-nginx-nkpt7
```

4. Pod 하나를 삭제해보자. 컨트롤러는 원하는 수의 pod가 동작되고 있는지 모니터링하면서 Pod가 부족하면 추가로 생성하고, 많으면 Pod를 제거한다.

```
T2# watch kubectl get pods
# kubectl delete pod rc-nginx-XXX
```

현재 Pod 수는?

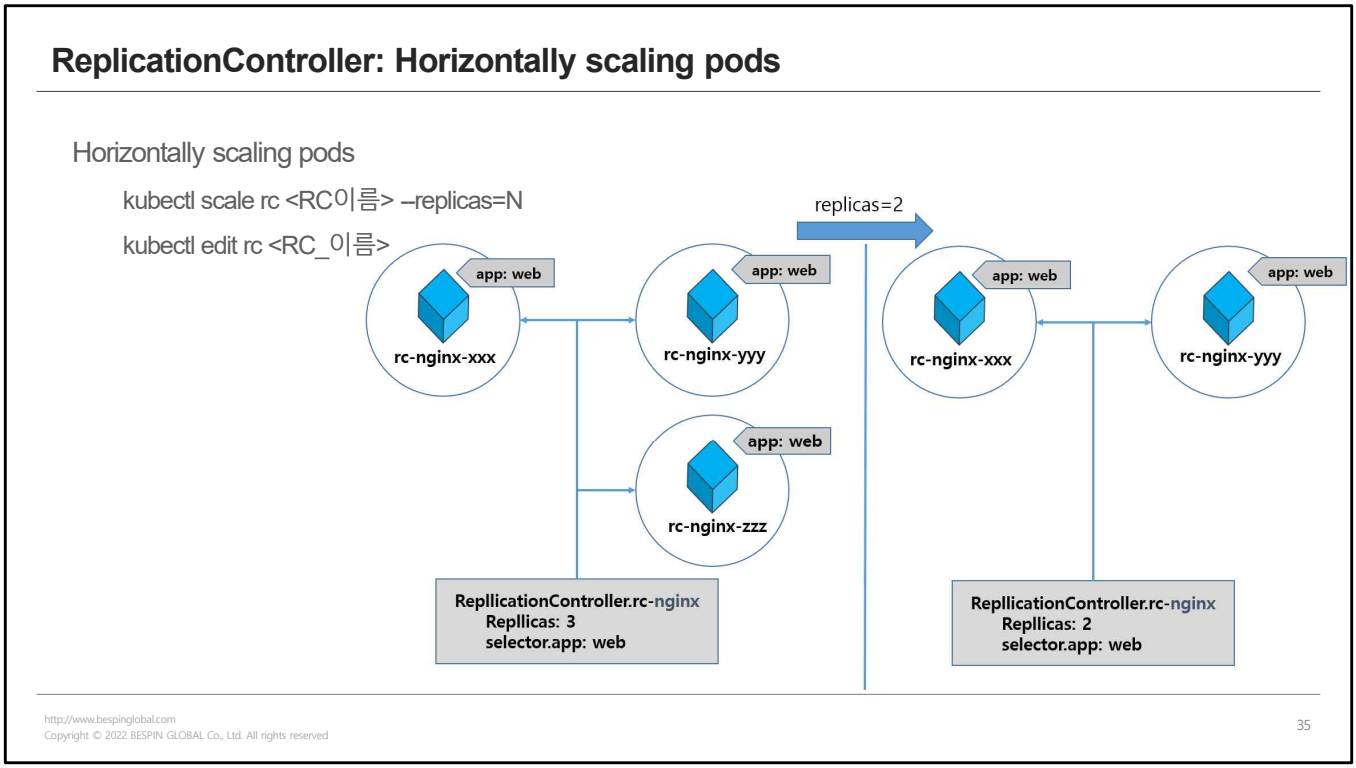
무엇이 달라졌나?

5. pod-other appjs 웹 애플리케이션을 동작하는 pod를 실행해보자.

```
# cat pod-other.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-other
  labels:
    app: web
spec:
  containers:
  - name: appjs-container
    image: smlinux/appjs
    ports:
    - containerPort: 8080

# kubectl create -f pod-other.yaml
pod/pod-other created

# kubectl get pods
```

pod-other는 정상으로 실행되었나?

실행되었다면/실행되지 않았다면 이유는 무엇일까?



ReplicationController: Horizontally scaling pods

1. 동작중인 rc-nginx의 replicas 수를 3에서 2로 줄여보자.

```
# kubectl scale rc  rc-nginx  --replicas=2
replicationcontroller/rc-nginx scaled

# kubectl get rc,pods
NAME                         DESIRED   CURRENT   READY   AGE
replicationcontroller/rc-nginx   2         2         2       16m

NAME              READY   STATUS    RESTARTS   AGE
pod/rc-nginx-kgj5z   1/1     Running   0          16m
pod/rc-nginx-l2jkp   1/1     Running   0          16m
```

현재 Pod수는 몇 개인가?

2. edit 명령을 통해 Pod수를 확장(scale-out)해보자.

```
# kubectl edit rc  rc-nginx
apiVersion: v1
kind: ReplicationController
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"ReplicationController","metadata":{"annotations":{},"name":"rc-
    nginx","namespace":"default"},"spec":{"replicas":3,"selector":{"app":"web"},"template":{"metadata":{"labels":{"app":"web"},"name":"nginx-
    pod"},"spec":{"containers":[{"image":"nginx:1.14","name":"nginx-container"}]}}}}
  creationTimestamp: "2021-10-06T13:59:06Z"
  generation: 2
  labels:
    app: web
  name: rc-nginx
  namespace: default
  resourceVersion: "210751"
  uid: 885b8e71-fb37-4c27-8ab1-e12328a1ccd0
spec:
  replicas: 5
  selector:
    app: web
  template:
<ESC>:wq
```

Pod수는 어떻게 되었나?

3. 파드를 삭제해보자. 동작하고 있는 Pod 전체를 삭제하자.
 # kubectl delete pod all

결과는 어떻게 되었나?

삭제되었다면/삭제되지 않았다면 이유는 무엇인가?

4. 동작중인 ReplicationController를 삭제하자.
 # kubectl delete rc rc-nginx
replicationcontroller "rc-nginx" deleted

심화 | ReplicationController 운영

ReplicationController 운영

1. 다음의 조건으로 ReplicationController를 사용하는 rc-lab.yaml 파일을 생성하고 동작시킵니다.
   labels(name: apache, app:main, rel:stable)를 가지는 httpd:2.2 버전의 Pod를 2개 운영합니다.
   - **rc name : rc-mainui**
   - **container : httpd:2.2**

   현재 디렉토리에 rc-lab.yaml 파일이 생성되어야 하고, 애플리케이션 동작은 파일을 이용해 실행합니다.

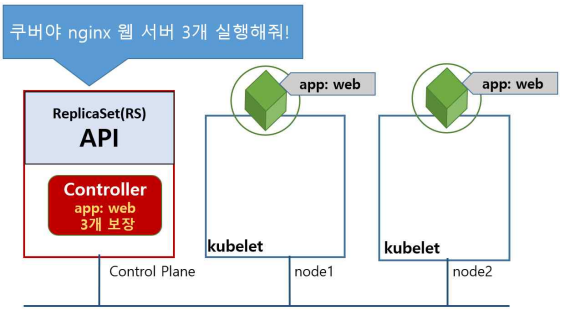
2. kubectl scale 명령으로 http:2.2 버전의 컨테이너를 3개로 확장하시오.
   CLI #

ReplicaSet

ReplicaSet

- ReplicationController 와 같은 역할을 하는 컨트롤러
- ReplicationController 보다 풍부한 selector

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]
```



- matchExpressions 연산자
  - In : key와 values를 지정하여 key, value가 일치하는 Pod만 연결
  - NotIn : key는 일치하고 value는 일치하지 않는 Pod에 연결
  - Exists : key에 맞는 label의 pod를 연결
  - DoesNotExist : key와 다른 label의 pod를 연결

ReplicaSet

ReplicaSet은 실행해야 하는 Pod Replicas를 보장한다.
ReplicaSet을 ReplicationController의 대체로 간주 할 수 있는데, ReplicaSet과 ReplicationController의 주요 차이점은 ReplicationController는 equality-based selector만 지원하는 반면 ReplicaSet은 set-based selector를 지원한다는 것이다.

ReplicaSet의 풍부한 label selectors 사용하기
ReplicationController에 비해 ReplicaSets의 주요 개선 사항은 보다 표현적인 label selector이다.

```
spec:
  replicas: 3
  selector:
    matchExpressions:
      - {key: ver, operator: In, value: ["1.14","1.15"]}
  template:
    ...
```

```
spec:
  replicas: 3
  selector:
    matchExpressions:
      - {key: ver, operator: Exists}
  template:
    ...
```

selector에 추가 할 수 있는 유효한 operators

- In : label의 value가 지정된 value 중 하나와 일치해야 한다.
- NotIn : label의 value가 지정된 value과 일치해서는 안 된다.
- Exists : Pod에 지정된 Key가 있는 label이 포함되어야 한다.
- DoesNotExist : Pod에 지정된 Key가 있는 label을 포함하면 안 된다.

여러 개의 표현식을 지정하는 경우 selector가 Pod와 일치하려면 모든 표현식이 true로 평가되어야한다.
또한 matchLabels와 matchExpressions를 모두 지정하는 경우 모든 label이 일치해야 하며 pod가 selector와 일치하려면 모든 표현식이 true로 평가되어야 한다.

ReplicaSet Template

ReplicationController-template	ReplicaSet-template

apiVersion: v1
kind: ReplicationController
metadata:
  name: rc-nginx
spec:
  replicas: 3
  selector:
  app: web
  version: 1.14
  template:
  metadata:
  name: nginx-pod
  labels:
  app: web
  spec:
  containers:
  - name: nginx-container
  image: nginx:1.14
	apiVersion: apps/v1
	kind: ReplicaSet
	metadata:
	  name: rs-nginx
	spec:
	  replicas: 3
	  selector:
	  matchLabels:
	  app: web
	  matchExpressions:
	  - {key: version, operator: In, values: ["1.14", "1.15"]}
	  template:
	  metadata:
	  name: nginx-pod
	  labels:
	  app: web
	  version: "1.14"
	  spec:
	  containers:
	  - name: nginx-container
	  image: nginx:1.14

ReplicaSet 구성

ReplicaSet definition을 보면 ReplicationController와 별반 다르지 않다. 유일한 차이점은 selector인데, Pod가 selector 속성 바로 아래에 있어야하는 레이블을 나열하는 대신 selector.matchLabels에서 지정한다.

실습 : 앞서 실행했던 ReplicationController 기반의 웹서버 nginx를 ReplicaSet 컨트롤러 오브젝트로 변경해서 운영해보자.
현재 웹서버를 통해 쇼핑몰을 운영한다 생각해보자.

1. 먼저 rc-nginx.yaml 파일을 rs-nginx.yaml 파일로 복사해서 ReplicaSet의 template에 맞춰서 수정하자.

```
# cp rc-nginx.yaml rs-nginx.yaml

# vi rs-nginx.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
    matchExpressions:
    - {key: version, operator: In, values: ["1.14", "1.15"]}
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
        version: "1.14"
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

2. 웹서비스를 동작시켜보자.

```
# kubectl apply -f rs-nginx.yaml
replicaset.apps/rs-nginx created

# kubectl get pod --show-labels
NAME             READY   STATUS    RESTARTS        AGE   LABELS
rs-nginx-2hg2v                     1/1     Running       0         63s   app=web,version=1.14
rs-nginx-dfkcd   1/1     Running       0         63s   app=web,version=1.14
rs-nginx-mkmtq                     1/1     Running       0         63s   app=web,version=1.14
```

3. 내일 오전에 MD추천 제품을 대거 Sale 하려고 한다. 고객 접속량이 급격히 높아질것을 예상하고 미리 Pod를 확장하려고한다. 스케일 아웃해보자.
위의 파드중 하나를 삭제하면 어떻게 되나?

```
# kubectl scale rs rs-nginx --replicas=5
replicaset.apps/rs-nginx scaled
```

Pod가 총 몇 개가 되었나?

Pod 하나를 삭제하면 무슨일이 발생하겠나?

4. rs-nginx ReplicaSet을 종료하자.

```
# kubectl delete rs rs-nginx
replicaset.apps "rs-nginx" deleted
```

심화 | ReplicaSet 운영

ReplicaSet 운영

1. 다음의 조건으로 ReplicaSet을 사용하는 rs-lab.yaml 파일을 생성하고 동작시킵니다.
   - labels(name: apache, app:main, rel:stable)를 가지는 httpd:2.2 버전의 Pod를 2개 운영합니다.
     - **rs name : rs-mainui**
     - **container : httpd:2.2**

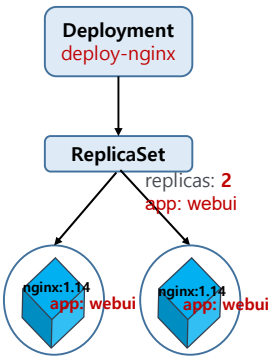
   - 현재 디렉토리에 rs-lab.yaml 파일이 생성되어야 하고, 애플리케이션 동작은 파일을 이용해 실행합니다.

2. 동작되는 http:2.2 버전의 컨테이너를 1개로 축소하는 명령을 적고 실행하세요.
   CLI #

Deployment

Deployment

- ReplicaSet을 컨트롤해서 Pod수를 조절
- Rolling Update & Rolling Back




```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      labels:
        app: webui
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```


Deployment

Deployments 는 ReplicaSet controller를 통해 Pod수를 유지하며, Rolling Update 또는 Rolling Back 하는 기능을 포함하고 있다.

실습
앞서 실행했던 rs-nginx.yaml 파일을 deploy-nginx.yaml 파일로 복사해서 deployment 오브젝트를 실행해보자.
현재 웹서버를 통해 쇼핑몰을 운영한다 생각해보자.

1. 먼저 rc-nginx.yaml 파일을 rs-nginx.yaml 파일로 복사해서 Deployments의 template에 맞춰서 수정하자.

```
# cp rs-nginx.yaml deploy-nginx.yaml

# vi deploy-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
    matchExpressions:
    - {key: version, operator: In, values: ["1.14", "1.15"]}
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
        version: "1.14"
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

2. 수정한 deploy-nginx.yaml을 실행한다.

```
# kubectl apply -f deploy-nginx.yaml
deployment.apps/deploy-nginx created
```

3. 동작 중인 Deployment를 확인해보자.

```
# kubectl get deploy,rs,pod
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
```

```
deployment.apps/deploy-nginx   3/3     3           3          28s

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/deploy-nginx-858b7d8fcd   3         3         3       28s

NAME                                READY   STATUS    RESTARTS   AGE
pod/deploy-nginx-858b7d8fcd-8kclx   1/1     Running   0          28s
pod/deploy-nginx-858b7d8fcd-8tfnm   1/1     Running   0          28s
pod/deploy-nginx-858b7d8fcd-qp5hj   1/1     Running   0          28s
```

위의 실행 결과를 보고 Deployment, ReplicaSet, Pod의 관계를 설명해보라.

4. replicas 5로 nginx를 확장해보자.

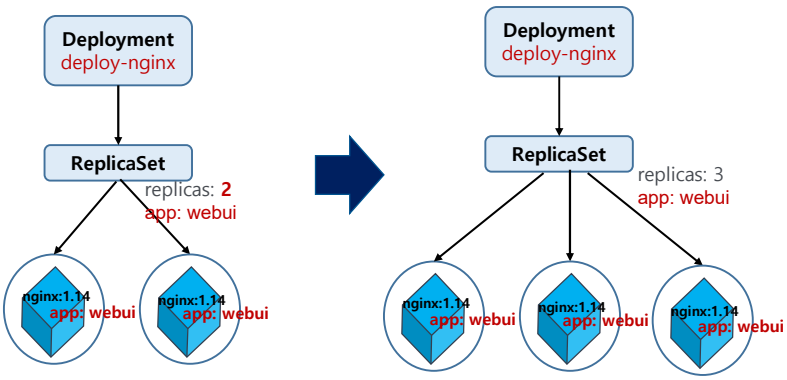
```
# kubectl scale deployment deploy-nginx --replicas=5
deployment.apps/deploy-nginx scaled

# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deploy-nginx-858b7d8fcd-8kclx    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-8tfnm    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-hbdd7                      1/1     Running   0          7s
deploy-nginx-858b7d8fcd-qp5hj    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-xmdm8                      1/1     Running   0          7s
```

5. 서비스를 종료시킵니다.

```
# kubectl delete deployments.apps deploy-nginx
deployment.apps "deploy-nginx" deleted
```

Deployment

- ReplicaSet 컨트롤해서 Pod수를 조절



Deployment

Deployments 는 ReplicaSet controller를 통해 Pod수를 유지하며, Rolling Update 또는 Rolling Back 하는 기능을 포함하고 있다.

실습
앞서 실행했던 rs-nginx.yaml 파일을 deploy-nginx.yaml 파일로 복사해서 deployment 오브젝트를 실행해보자.
현재 웹서버를 통해 쇼핑몰을 운영한다 생각해보자.

1. 먼저 rc-nginx.yaml 파일을 rs-nginx.yaml 파일로 복사해서 Deployments의 template에 맞춰서 수정하자.

```
# cp rs-nginx.yaml deploy-nginx.yaml

# vi deploy-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
    matchExpressions:
    - {key: version, operator: In, values: ["1.14", "1.15"]}
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
        version: "1.14"
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

2. 수정한 deploy-nginx.yaml을 실행한다.

```
# kubectl apply -f deploy-nginx.yaml
deployment.apps/deploy-nginx created
```

3. 동작 중인 Deployment를 확인해보자.

```
# kubectl get deploy,rs,pod
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
```

```
deployment.apps/deploy-nginx   3/3     3           3          28s

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/deploy-nginx-858b7d8fcd   3         3         3       28s

NAME                                READY   STATUS    RESTARTS   AGE
pod/deploy-nginx-858b7d8fcd-8kclx   1/1     Running   0          28s
pod/deploy-nginx-858b7d8fcd-8tfnm   1/1     Running   0          28s
pod/deploy-nginx-858b7d8fcd-qp5hj   1/1     Running   0          28s
```

위의 실행 결과를 보고 Deployment, ReplicaSet, Pod의 관계를 설명해보라.

4. replicas 5로 nginx를 확장해보자.

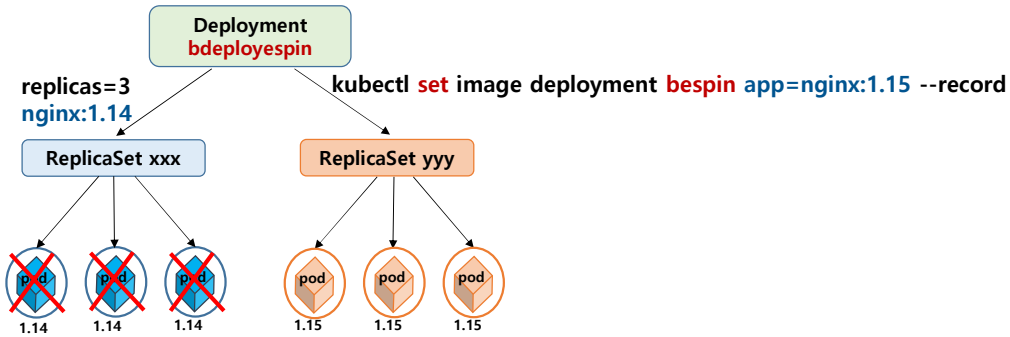
```
# kubectl scale deployment deploy-nginx --replicas=5
deployment.apps/deploy-nginx scaled

# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deploy-nginx-858b7d8fcd-8kclx    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-8tfnm    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-hbdd7                      1/1     Running   0          7s
deploy-nginx-858b7d8fcd-qp5hj    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-xmdm8                      1/1     Running   0          7s
```

5. 서비스를 종료시킵니다.

```
# kubectl delete deployments.apps deploy-nginx
deployment.apps "deploy-nginx" deleted
```

Deployment Rolling Update & Rolling Back(1)

- BackRolling Update
  kubectl set image deployment <deploy_name> <container_name>=<new_version_image>
- RollBack
  kubectl **rollout history** deployment <deploy_name>
  kubectl **rollout undo** deploy <deploy_name>



Rolling Update Example

1. nginx 웹서버를 rolling update하고 여러가지 deploy 관리 명령을 통해 업데이트 상태를 관리해보자
   # kubectl create -f deploy-nginx.yaml --record
   # kubectl get deployment,rs,pod,svc

2. nginx웹서버를 1.14 version에서 1.15, 1.16, 1.17버전으로 순차적으로 rollingupdate 진행해보자.

   1.14에서 1.15버전으로 rollingupdate 한다.
   # kubectl set image deploy deploy-nginx nginx-container=nginx:1.15 --record

   1.15버전에서 1.16버전으로 rollingupdate한다.
   # kubectl set image deploy deploy-nginx nginx-container=nginx:1.16 --record

   update되는 중간에 일시중지시키고 다시 재가동한다.
   # kubectl rollout pause deploy deploy-nginx
   # kubectl rollout resume deploy deploy-nginx

   1.16버전에서 1.17버전으로 rollingupdate한다.
   # kubectl set image deploy deploy-nginx nginx-container=nginx:1.17 --record

   update 되는 상태를 모니터링 해본다.
   # kubectl rollout status deploy deploy-nginx

3. rolling update를 통해 생성된 ReplicaSet을 확인해보라.
   # kubectl get rs

```
NAME                          DESIRED   CURRENT   READY   AGE
deploy-nginx-5c586f9bd        0         0         0       6m12s
deploy-nginx-5c8869fd97       0         0         0       6m29s
deploy-nginx-858b7d8fcd       0         0         0       7m6s
deploy-nginx-86b744b959       3         3         3       6m6s
```

4. application rollback

   현재의 rolling update 히스토리를 확인한다.
   # kubectl rollout history deployment deploy-nginx

```
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy-nginx.yaml --record=true
2         kubectl set image deploy deploy-nginx nginx-container=nginx:1.15 --record=true
```

```
3          kubectl set image deploy deploy-nginx nginx-container=nginx:1.16 --record=true
4          kubectl set image deploy deploy-nginx nginx-container=nginx:1.17 --record=true
```

revision 2에 해당하는 nginx:1.15버전이 동작하도록 rollback 한다.
 # kubectl rollout undo deployment deploy-nginx --to-revision 2

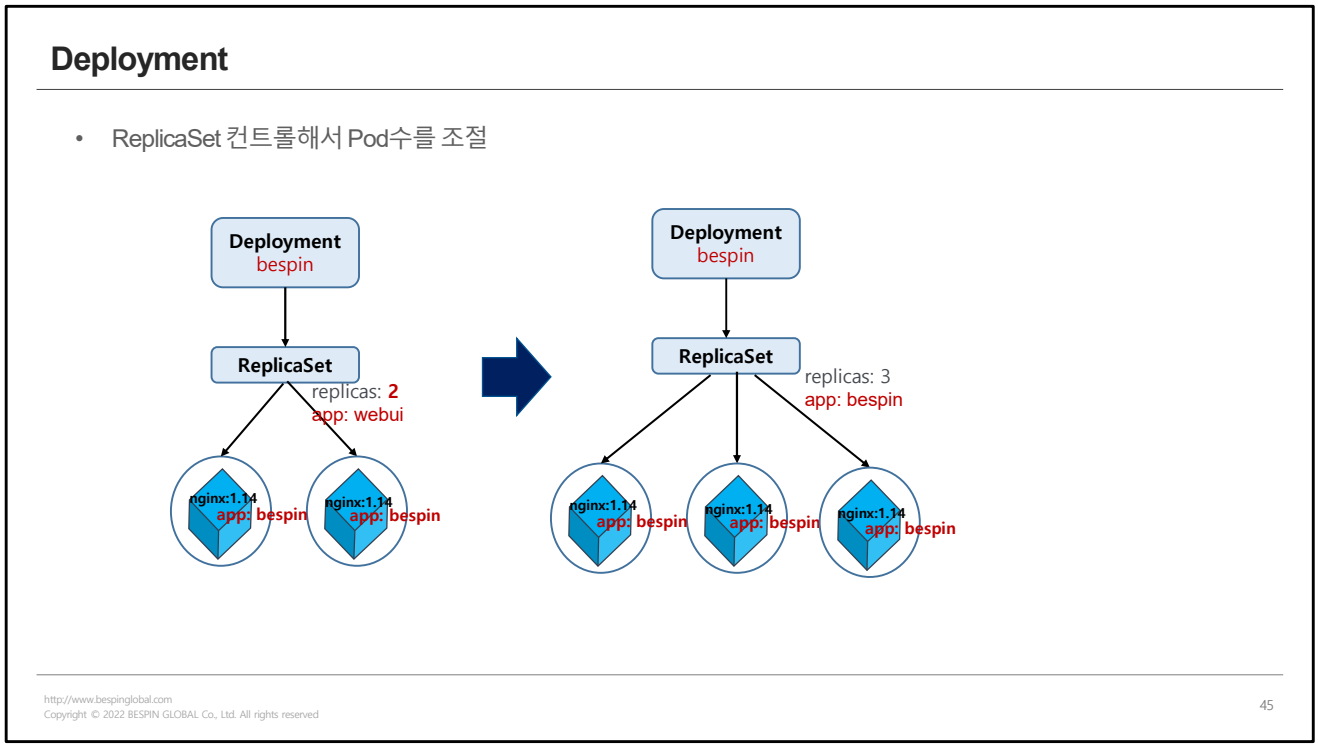
현재의 rolling update 히스토리를 확인하고 undo 명령을 통해 1.14버전으로 돌려본다.
 # kubectl rollout history deployment deploy-nginx

```
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy-nginx.yaml --record=true
3         kubectl set image deploy deploy-nginx nginx-container=nginx:1.16 --record=true
4         kubectl set image deploy deploy-nginx nginx-container=nginx:1.17 --record=true
5         kubectl set image deploy deploy-nginx nginx-container=nginx:1.15 --record=true
```

kubectl rollout undo 명령을 실행하면 어떤 버전이 실행될까?

5. 앞서 실행한 deploy-nginx.yaml을 삭제하자.

```
# kubectl delete deploy deploy-nginx
```



Deployment

Deployments 는 ReplicaSet controller를 통해 Pod수를 유지하며, Rolling Update 또는 Rolling Back 하는 기능을 포함하고 있다.

실습
앞서 실행했던 rs-nginx.yaml 파일을 deploy-nginx.yaml 파일로 복사해서 deployment 오브젝트를 실행해보자.
현재 웹서버를 통해 쇼핑몰을 운영한다 생각해보자.

1. 먼저 rc-nginx.yaml 파일을 rs-nginx.yaml 파일로 복사해서 Deployments의 template에 맞춰서 수정하자.

```
# cp rs-nginx.yaml deploy-nginx.yaml

# vi deploy-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
    matchExpressions:
    - {key: version, operator: In, values: ["1.14", "1.15"]}
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
        version: "1.14"
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

2. 수정한 deploy-nginx.yaml을 실행한다.

```
# kubectl apply -f deploy-nginx.yaml
deployment.apps/deploy-nginx created
```

3. 동작 중인 Deployment를 확인해보자.

```
# kubectl get deploy,rs,pod
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
```

```
deployment.apps/deploy-nginx   3/3     3           3          28s

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/deploy-nginx-858b7d8fcd   3         3         3       28s

NAME                                READY   STATUS    RESTARTS   AGE
pod/deploy-nginx-858b7d8fcd-8kclx   1/1     Running   0          28s
pod/deploy-nginx-858b7d8fcd-8tfnm   1/1     Running   0          28s
pod/deploy-nginx-858b7d8fcd-qp5hj   1/1     Running   0          28s
```

위의 실행 결과를 보고 Deployment, ReplicaSet, Pod의 관계를 설명해보라.

4. replicas 5로 nginx를 확장해보자.

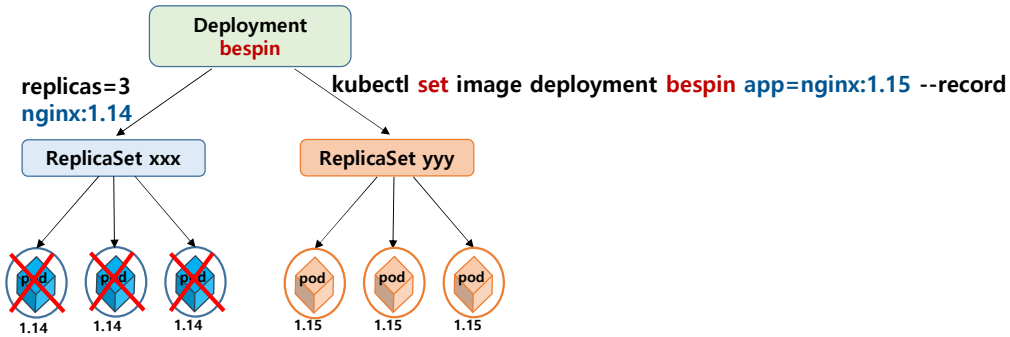
```
# kubectl scale deployment deploy-nginx --replicas=5
deployment.apps/deploy-nginx scaled

# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deploy-nginx-858b7d8fcd-8kclx    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-8tfnm    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-hbdd7                      1/1     Running   0          7s
deploy-nginx-858b7d8fcd-qp5hj    1/1     Running   0          17m
deploy-nginx-858b7d8fcd-xmdm8                      1/1     Running   0          7s
```

5. 서비스를 종료시킵니다.

```
# kubectl delete deployments.apps deploy-nginx
deployment.apps "deploy-nginx" deleted
```

Deployment Rolling Update & Rolling Back(1)

- BackRolling Update
  kubectl set image deployment <deploy_name> <container_name>=<new_version_image>
- RollBack
  kubectl **rollout history** deployment <deploy_name>
  kubectl **rollout undo** deploy <deploy_name>



Rolling Update Example

1. nginx 웹서버를 rolling update하고 여러가지 deploy 관리 명령을 통해 업데이트 상태를 관리해보자
   # kubectl create -f deploy-nginx.yaml --record
   # kubectl get deployment,rs,pod,svc

2. nginx웹서버를 1.14 version에서 1.15, 1.16, 1.17버전으로 순차적으로 rollingupdate 진행해보자.

   1.14에서 1.15버전으로 rollingupdate 한다.
   # kubectl set image deploy deploy-nginx nginx-container=nginx:1.15 --record

   1.15버전에서 1.16버전으로 rollingupdate한다.
   # kubectl set image deploy deploy-nginx nginx-container=nginx:1.16 --record

   update되는 중간에 일시중지시키고 다시 재가동한다.
   # kubectl rollout pause deploy deploy-nginx
   # kubectl rollout resume deploy deploy-nginx

   1.16버전에서 1.17버전으로 rollingupdate한다.
   # kubectl set image deploy deploy-nginx nginx-container=nginx:1.17 --record

   update 되는 상태를 모니터링 해본다.
   # kubectl rollout status deploy deploy-nginx

3. rolling update를 통해 생성된 ReplicaSet을 확인해보라.
   # kubectl get rs

```
NAME                          DESIRED   CURRENT   READY   AGE
deploy-nginx-5c586f9bd        0         0         0       6m12s
deploy-nginx-5c8869fd97       0         0         0       6m29s
deploy-nginx-858b7d8fcd       0         0         0       7m6s
deploy-nginx-86b744b959       3         3         3       6m6s
```

4. application rollback

   현재의 rolling update 히스토리를 확인한다.
   # kubectl rollout history deployment deploy-nginx

```
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy-nginx.yaml --record=true
2         kubectl set image deploy deploy-nginx nginx-container=nginx:1.15 --record=true
```

```
3          kubectl set image deploy deploy-nginx nginx-container=nginx:1.16 --record=true
4          kubectl set image deploy deploy-nginx nginx-container=nginx:1.17 --record=true
```

revision 2에 해당하는 nginx:1.15버전이 동작하도록 rollback 한다.
 # kubectl rollout undo deployment deploy-nginx --to-revision 2

현재의 rolling update 히스토리를 확인하고 undo 명령을 통해 1.14버전으로 돌려본다.
 # kubectl rollout history deployment deploy-nginx

```
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy-nginx.yaml --record=true
3         kubectl set image deploy deploy-nginx nginx-container=nginx:1.16 --record=true
4         kubectl set image deploy deploy-nginx nginx-container=nginx:1.17 --record=true
5         kubectl set image deploy deploy-nginx nginx-container=nginx:1.15 --record=true
```

kubectl rollout undo 명령을 실행하면 어떤 버전이 실행될까?

5. 앞서 실행한 deploy-nginx.yaml을 삭제하자.

```
# kubectl delete deploy deploy-nginx
```

Deployment Rolling Update & Rolling Back(2)

- Annotation
  - **key-value**를 통해 리소스 특성을 기록
  - Kubernetes 에게 특정 정보 전달할 용도로 사용
  - 예를 들어 Deployment의 rolling update 정보 기록

```
annotations:
  kubernetes.io/change-cause: version 1.15
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: deploy-nginx
 annotations:
  kubernetes.io/change-cause: version 1.14
spec:
 progressDeadlineSeconds: 600
 revisionHistoryLimit: 10
 strategy:
  rollingUpdate:
   maxSurge: 25%
   maxUnavailable: 25%
  type: RollingUpdate
 replicas: 3
 selector:
…
```


Anotation을 이용한 Rolling Update Example

1. annotations에 update version 정보를 저장한다.

```
# cp deploy-nginx.yaml deploy-nginx-update.yaml

# vi deploy-nginx-update.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx-update
  annotations:
    kubernetes.io/change-cause: version 1.14
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
        version: "1.14"
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14

# kubectl apply -f deploy-nginx-update.yaml
```

2. 히스토리 정보를 확인해서 CHANGE-CAUSE에 어떤 내용이 기록되었는지 확인해보라.

```
# kubectl rollout history deployment deploy-nginx-update
deployment.apps/deploy-nginx-update
REVISION  CHANGE-CAUSE
1         version 1.14
```

3. deploy-nginx-image 버전을 수정하고, annotation도 수정한 후에 재 적용한다.

```
# vi deploy-nginx-update.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx-update
```

```
  annotations:
    kubernetes.io/change-cause: version 1.15
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.15
```

```
# kubectl apply -f deploy-nginx.yaml
deployment.apps/deploy-nginx configured
```

업데이트 상황은 어떠한가? ReplicaSet이 2개로 확장되었나?

```
# kubectl get deployments,rs
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/deploy-nginx-update   3/3     3            3           3m33s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/deploy-nginx-update-5759bbff85   3         3         3       108s
replicaset.apps/deploy-nginx-update-74dbbbc58    0         0         0       3m33s
```

4. 업데이트 히스토리를 확인해보자.

```
# kubectl rollout history deployment deploy-nginx-update
deployment.apps/deploy-nginx-update
REVISION  CHANGE-CAUSE
1         version 1.14
2         version 1.15
```

5. 1.15버전을 1.16버전으로 rolling update시키고 히스토리를 확인해보자.

6. rollout undo 명령을 이용해서 1.14버전으로 rollback 해보자.

7. 동작중인 deployment를 삭제하라.

```
# kubectl delete deployments.apps deploy-nginx-update
deployment.apps "deploy-nginx-update" deleted
```

심화 | Deployment 운영

Deployment 운영

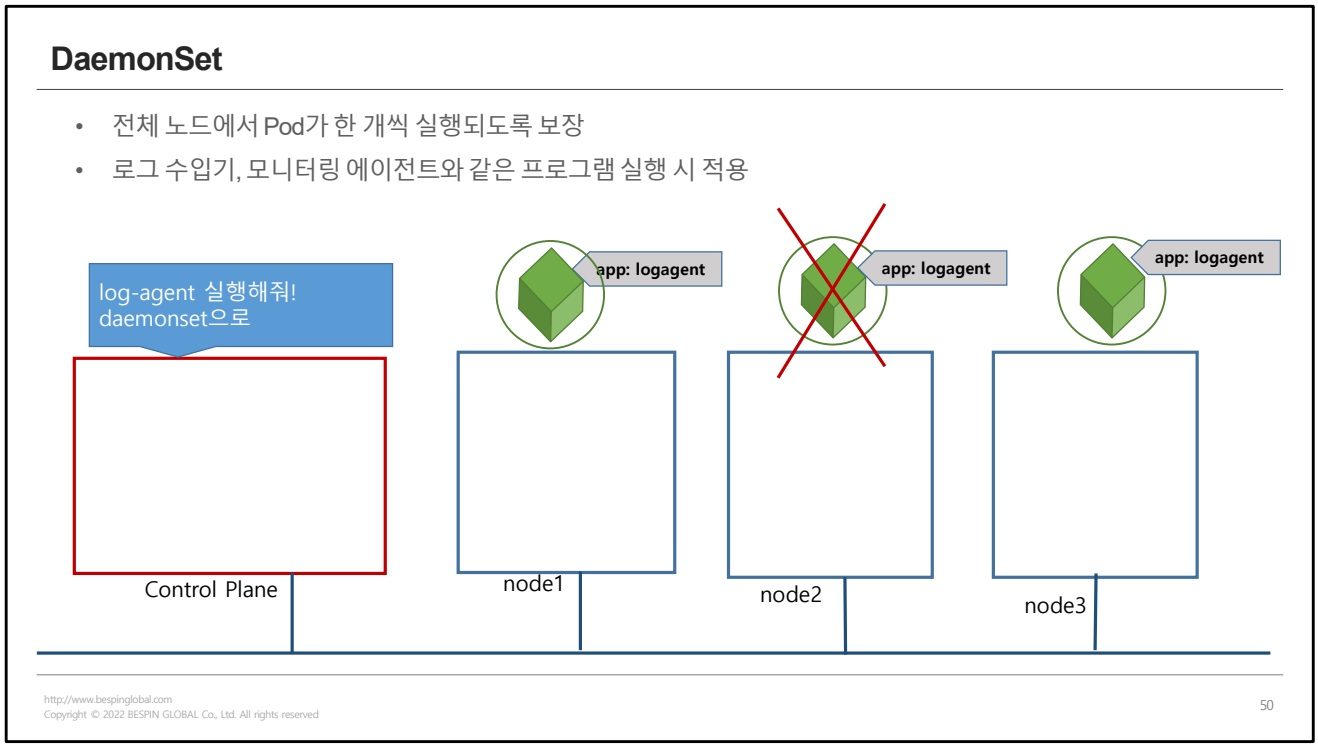
1. 다음의 조건으로 Deployment을 사용하는 dep-lab.yaml 파일을 생성하고 apply 명령으로 동작시킵니다.
   - labels(name: apache, app: main, rel: stable)를 가지는 httpd:2.2 버전의 Pod를 2개 히스토리를 기록하며 운영합니다.
   - annotations(kubernetes.io/change-cause: version 2.2) 를 추가로 설정합니다.
     - **deployment name : dep-mainui**
     - **container : httpd:2.2**

2. 동작 되는 dep-lab.yaml 의 이미지를 http:2.4 버전으로 rolling update 합니다.
   단, apply 명령을 통해 rolling update 진행합니다.

3. 현재의 dep-mainui 히스토리(history)를 확인하고 rollback 시킵니다.

4. 현재 동작중인 Pod의 httpd 이미지 버전은 어떻게 되는지 확인합니다.

DaemonSet



DaemonSet

ReplicatSet은 일반적으로 중복성을 목적으로 다수의 replicas를 이용한 서비스 생성과 관련이 있다. 하지만 클러스터에서 Pod 집합을 복제하는 것은 이 때문만은 아니다.
Pod 집합을 복제하는 또 다른 이유는 클러스터의 모든 Node에서 단일 Pod를 스케줄링하려는 것이다.
일반적으로 Pod를 모든 Node로 복제하는 이유는 각 Node에 일종의 agent나 daemon을 두려는 것이다.

DaemonSet은 kubernetes의 각 노드에서 Pod가 한 개씩 생성되도록 보장해주는 오브젝트이다. 보통 로그 수집기나 모니터링 에이전트처럼 모든 노드에서 동작하는 시스템 데몬을 배포할 때 DaemonSet을 사용한다.

시스템이 사용하는 daemonSet

kubernetes 사용하고 있는 DaemonSet 오브젝트들이 있다.

```
# kubectl get daemonsets.apps -n kube-system
NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
kube-proxy   3         3         3       3            3           kubernetes.io/os=linux   5d2h
weave-net    3         3         3       3            3           <none>                   5d2h
```

weave-net과 kube-proxy는 모든 노드에서 실행하며 Pod Network(CNI)과 kubernetes Network을 지원한다.

```
# kubectl get pods -n kube-system -o wide | grep -e proxy  -e weave
kube-proxy-8zkvr                    1/1     Running   3 (9h ago)    2d18h   10.0.0.12   node2.example.com    <none>           <none>
kube-proxy-b66bf                    1/1     Running   5 (9h ago)    5d2h    10.0.0.10   master.example.com   <none>           <none>
kube-proxy-c29vk                    1/1     Running   5 (9h ago)    5d2h    10.0.0.11   node1.example.com    <none>           <none>
weave-net-jtcgb                     2/2     Running   12 (9h ago)   5d2h    10.0.0.11   node1.example.com    <none>           <none>
weave-net-l888m                     2/2     Running   6 (9h ago)    2d18h   10.0.0.12   node2.example.com    <none>           <none>
weave-net-s99f4                     2/2     Running   12 (9h ago)   5d2h    10.0.0.10   master.example.com   <none>           <none>
```

DaemonSet Example

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemonset-nginx
spec:
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      name: nginx-pod
      labels:
        app: webui
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

```
$ kubectl create -f daemonset-exam.yaml
$ kubectl get daemonset
$ kubectl get pods

daemonset으로 동작중인 pod를 삭제하면?
$ kubectl delete pod daemonset-nginx-XXX

daemonset rolling update - 컨테이너 버전 수정
$ kubectl edit ds daemonset-nginx
containers:
     - image: nginx:1.15
저장하면 무슨 일이 일어나나?

Rolling Back
$ kubectl rollout undo daemonset daemonset-nginx

삭제
$ kubectl delete daemonsets.apps daemonset-nginx
```


DaemonSet Example

1. DaemonSet은 Pod가 전체 worker node 또는 특정 worker node에서 관리자의 개입 없이 실행 운영되도록 컨트롤 한다. 보통 백 그라운드 작업을 배포하는 데 사용하는데, ceph와 같은 스토리지 데몬, fluentd와 같은 로그 수집 데몬, collectd와 같은 노드 모니터링 데몬들이 대표적으로 DaemonSet으로 실행된다.

 # cat daemonset-fluentd.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      containers:
      - name: fluentd-container
        image: gcr.io/google-containers/fluentd-elasticsearch:1.19
```

위의 template에서 deployment와 다른점을 찾아보시오.

2. fluented 애플리케이션을 실행하고 node당 한개씩 실행되었는지 확인해보자.

 # kubectl apply -f daemonset-fluentd.yaml

```
daemonset.apps/fluentd created
```

 # kubectl get pods -o wide

```
NAME            READY   STATUS    RESTARTS   AGE    IP          NODE               NOMINATED NODE   READINESS GATES
fluentd-4blfk   1/1     Running   0          2m8s   10.36.0.1   node1.example.com   <none>           <none>
fluentd-5p9kr   1/1     Running   0          2m8s   10.46.0.0   node2.example.com   <none>           <none>
```

3. node1에서 실행중인 pod를 삭제하면 어떻게 되는가?
   # kubectl delete pod fluentd-XXX
   pod "fluentd-XXX" deleted

   Pod 삭제후 어떤 변화가 있는지 확인하세요.

4. 만약 새로운 Node가 추가된다면 어떤 상황이 되겠는가?

5. 동작중인 DaemonSet을 제거하자.
   # kubectl delete daemonsets.apps fluentd
   daemonset.apps "fluentd" deleted

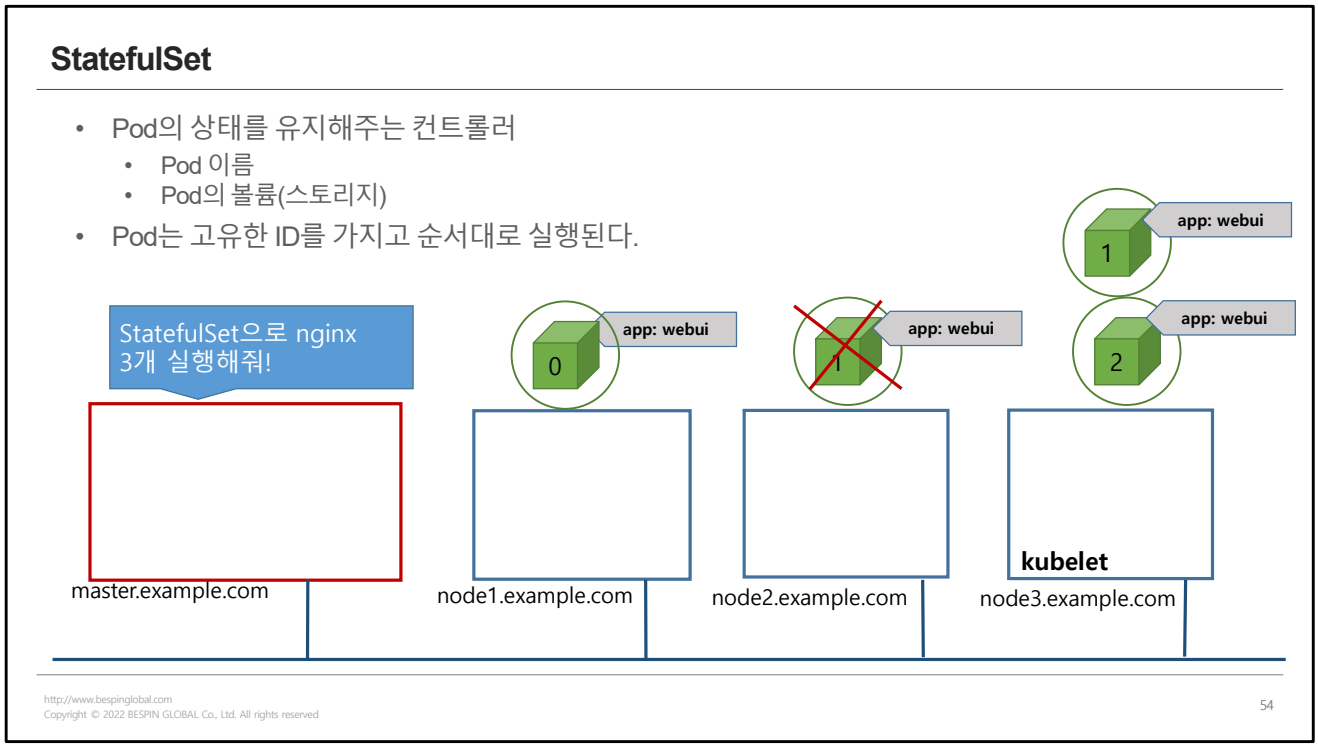
심화 | DaemonSet 운영

DaemonSet 운영

1. httpd:2.2 버전의 Pod를 worker node 당 1개씩 실행하도록 DaemonSet API를 생성합니다.
   - **filename: ds-lab.yaml**
   - **DaemonSet name : ds-mainui**
   - **Container name : web-container**
   - **Container Image : httpd:2.2**
   - **CPU Requests: 500m**
   - **NodeSelector: disktype=ssd**

2. 동작 되는 ds-lab.yaml 의 이미지를 http:2.4 버전으로 rolling update 합니다.

StatefulSet



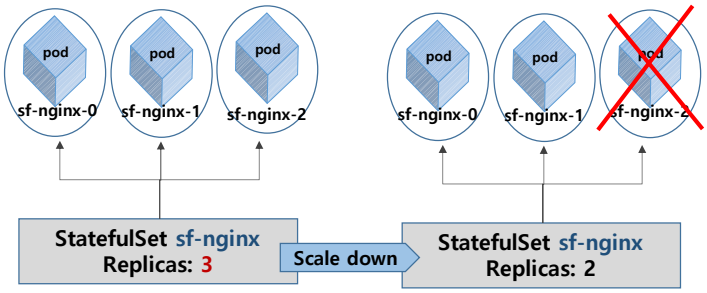
StatefulSet

Statefulsets는 Pod가 지속적인 상태를 유지 할 수 있도록 한다.
statefulsets를 통해 생성된 Pod는 고유한 ID를 가지며 지정된 순서대로 scale-out이나 scale-in을 수행할 수 있다.
Replicas 수를 변경할 경우 StatefulSets는 Pod을 제거하거나 추가한다. 이 과정에서 가장 높은 번호를 가지는 Pod부터 순서대로 종료시키는데, 낮은 번호를 갖는 Pod의 상태가 Read 및 Running 상태가 될 때까지 높은 번호를 갖는 Pod를 종료시키지 않는다.

Pod의 지속적인 상태를 유지
- Pod name(Host name)
- Storage Volume

StatefulSet Example

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sf-nginx
spec:
  replicas: 3
  serviceName: sf-nginx-service
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      name: nginx-pod
      labels:
        app: web
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```



55

StatefulSet Example

1. StatefulSet는 Pod의 상태를 보존하는 오브젝트이다. 간단한 예제를 통해서 확인해보자.

```
# cp rs-nginx.yaml statefulset-nginx.yaml
# vi statefulset-nginx.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sf-nginx
spec:
  replicas: 3
  serviceName: sf-nginx-service
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      name: nginx-pod
      labels:
        app: webui
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.14
```

serviceName : StatefulSet에서 사용할 servicename을 명시한다.
terminationGracePeriodSeconds : gracefull shutdown시간을 설정해서 실행중인 프로세스가 종료

2. nginx 웹서비스를 실행하고, 상태를 확인하자.

```
# kubectl apply -f statefulset-nginx.yaml
statefulset.apps/sf-nginx created

# kubectl get statefulsets,pod
NAME                    READY   AGE
statefulset.apps/sf-nginx   3/3     6m2s

NAME           READY   STATUS    RESTARTS   AGE
pod/sf-nginx-0   1/1     Running   0          6m2s
pod/sf-nginx-1   1/1     Running   0          6m
pod/sf-nginx-2   1/1     Running   0          5m58s
```

StatefulSet으로 생성된 Pod와 ReplicaSet으로 만들어지는 Pod가 다른점은 무엇인가?

3. Pod 하나를 삭제했을때 생성되는 Pod의 이름은 어떻게 되나?

```
# kubectl delete pod sf-nginx-1
pod "sf-nginx-1" deleted

# kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
sf-nginx-0  1/1    Running   0          17m
sf-nginx-1  1/1    Running   0          67s
sf-nginx-2  1/1    Running   0          17m
```

4. statefulset 또한 다른 컨트롤러 처럼 scale-out, scale-in을 지원한다.

```
# kubectl scale statefulset sf-nginx --replicas=2
statefulset.apps/sf-nginx scaled

# kubectl get pods
NAME         READY   STATUS    RESTARTS   AGE
sf-nginx-0   1/1     Running   0          18m
sf-nginx-1   1/1     Running   0          2m6s
```

문제: **--replicas=4**로 설정해서 scale-out하면 pod명은 어떻게 만들어질까?

5. StatefulSet을 통해 애플리케이션 rolling update 가능하다.
편집모드로 들어가서 nginx를 1.14에서 1.15로 변경하고 저장해보자. 어떤일이 일어나는가?

```
# kubectl edit statefulsets.apps sf-nginx
...
 spec:
      containers:
      - image: nginx:1.15
```

6. StatefulSet 서비스를 중지시키자.

```
# kubectl delete statefulsets.apps sf-nginx
statefulset.apps "sf-nginx" deleted
```

심화 | StatefulSet 운영

StatefulSet 운영

1. httpd:2.2 버전의 Pod를 2개 실행하는 StatefulSet을 생성합니다.
   - **filename: sf-lab.yaml**
   - **StatefulSet name : sf-mainui**
   - **SerivceName: sf-mainui-svc**
   - **Container name : web-container**
   - **Container Image : httpd:2.2**

2. 동작 되는 web-container Pod를 4개로 scale-out 합니다.

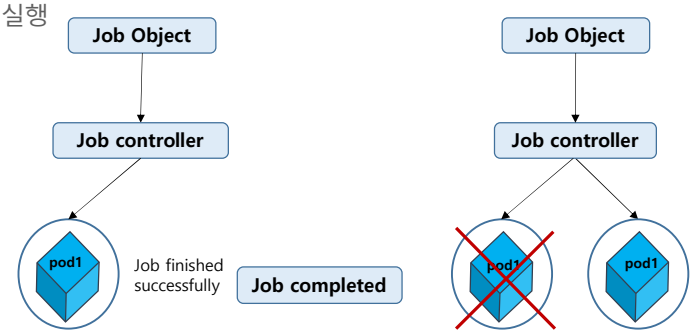
3. 확장된 Pod의 번호는 몇번까지 만들어졌나?

4. node1이 fail 되었다. node1에서 동작중인 파드는 어떻게 될까?

Job

Job

- Kubernetes는 Pod를 running 중인 상태로 유지
- Batch 처리하는 Pod는 작업이 완료되면 종료됨.
  - 작업이 완료되어도 Pod가 제거되지는 않음.
  - 작업이 완료되는 시점이 중요한 서비스에 유용
- Batch 처리에 적합한 컨트롤러로 Pod의 성공적인 완료를 보장
  - Pod 내의 컨테이너가 비정상 종료 시 다시 실행
  - Pod 내의 컨테이너가 정상 종료 시 완료



Job

Job의 주요 기능은 하나 이상의 pod를 만들어 실행하고, pod의 성공적인 종료 상태를 추적하는 것이다.
지정된 수의 포드가 성공적으로 완료되면 작업이 완료된 것으로 간주하고, 완료되지 않은 상태로 종료되면 Pod를 재시작한다.

완료 가능한 단일 task를 수행하는 pod 실행 : Job
ReplicationControllers, ReplicaSets, DaemonSets의 Pod들은 프로세스가 종료된 후에도 다시 시작된다. 그러나 Job 은 완료된 Pod가 성공적으로 종료 시 다시 시작하지 않는다.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: <Job 이름>
spec:
  template:
    metadata:
    ...
    spec:
      restartPolicy: OnFailure
      containers:
      ...
```

restartPolicy
- OnFailure : 비정상 종료 시 원래 실행 중이던 노드에서 컨테이너 재시작
- Never : 비정상 종료 시 재 시작을 막은 후 새로운 Pod를 실행

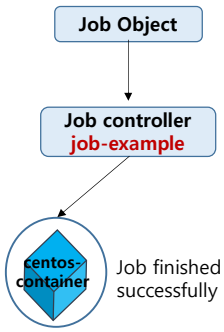
Parallel Jobs
- completions : Job 오브젝트로 실행할 Pod 수
- parallelism: job에서 여러 개의 pod를 동시에 실행

activeDeadlineSeconds
- 지정된 시간 안에 Job 애플리케이션을 동작시키고 종료하려면 activeDeadlineSeconds 필드에 시간을 설정한다.
- 아래 예와 같이 300초를 할당하면 300초 내에 pod가 실행되어 종료되어야하는데, 종료하지 못하면.
  name: <Job 이름>강제로 종료시키고 DeadlineExceeded를 기록한다

```
spec:
  completions: 5
  parallelism: 3
  activeDeadlineSeconds: 300
  template:
    metadata:
    ...
    spec:
      restartPolicy: OnFailure
```

Job Example

```
apiVersion: apps/v1
kind: Job
metadata:
  name: job-example
spec:
  template:
    spec:
      containers:
      - name: centos-container
        image: centos:7
        command: ["bash"]
        args:
        - "-c"
        - "echo 'Hello World'; sleep 50; echo 'Bye'"
      restartPolicy: Never
```



completions : 실행해야 할 job의 수가 몇 개인지 지정
parallelism : 병렬성. 동시 running되는 pod 수
activeDeadlineSeconds : 지정 시간 내에 Job을 완료

Job Example

1. 간단한 Job 기반의 배치처리 애플리케이션을 동작해보자.

```
# vi# cat job-example.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  template:
    spec:
      containers:
      - name: centos-container
        image: centos:7
        command: ["bash"]
        args:
        - "-c"
        - "echo 'Hello World'; sleep 50; echo 'Bye'"
      restartPolicy: Never
```

위의 yaml 파일을 보고 어떤 작업을 수행하는지 확인해보자.

2. job-example 서비스를 백업과 같은 애플리케이션이다 생각하고 동작시켜 봅시다.

```
# # kubectl apply -f job-example.yaml
job.batch/job-example created

# kubectl get jobs.batch
NAME          COMPLETIONS   DURATION   AGE
job-example   0/1           24s        24s

# kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
job-example--1-8kx7j   1/1     Running   0          36s

1분 후
# kubectl get pods
NAME                 READY   STATUS      RESTARTS   AGE
job-example--1-8kx7j   0/1     Completed   0          70s
```

Job을 통해 실행된 Pod는 작업이 완료되면 completed 상태가 된다.

위와 동일한 서비스를 ReplicaSet으로 동작시켜보자. 뭐가 다를까?

동작중인 Job을 종료시켜보자. 왜 pod를 종료시키면 안될까?
 # kubectl delete jobs job-example

3. 이번에는 job-example을 수정해서 completions로 Pod를 3개 실행하도록 운영해보자.

아래와 같이 수정하고 적용해보자.

```
# vi job-example.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  completions: 5
  template:
    spec:
      containers:
      - name: centos-container
        image: centos:7
        command: ["bash"]
        args:
        - "-c"
        - "echo 'Hello World'; sleep 50; echo 'Bye'"
      restartPolicy: Never
```

```
# kubectl apply -f job-example.yaml
job.batch/job-example created
```

Pod가 몇 개 실행되나?

```
# kubectl get job
NAME          COMPLETIONS   DURATION   AGE
job-example   2/5           119s       2m

# kubectl get pods
NAME                  READY   STATUS      RESTARTS   AGE
job-example--1-5vddc   0/1     Completed   0          2m5s
job-example--1-cwhzh   1/1     Running     0          22s
job-example--1-lgc6b   0/1     Completed   0          73s
```

4. 동작중인 Job을 종료시킨다.

```
# kubectl delete job job-example
```

심화 | 배치 Job 운영

배치 Job 운영

1. 앞서 동작되는 job-example을 다음의 조건으로 수정하자.

   - 배치 job 실행 조건
     completions: 5
     parallelism: 4
     activeDeadlineSeconds: 15

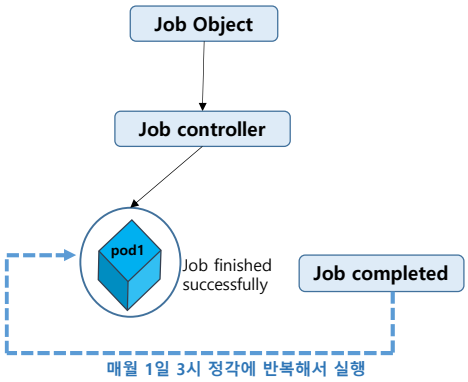
   - container image : centos
   - container command : bash -c echo 'Hello World'; sleep 5; echo 'Bye'"

2. Job이 동작되는 상황을 살펴보세요.

CronJob

CronJob

- job 오브젝트에 linux cronjob의 스케줄링 기능을 추가
- 다음과 같은 자동화 작업에 유리
  - Data Backup
  - Send email
  - Cleaning tasks
- CronJob Controller가 관리
- Cronjob Schedule: " 0 3 1 * *"
  - Minutes (from 0 to 59)
  - Hours (from 0 to 23)
  - Day of the month (from 1 to 31)
  - Month (from 1 to 12)
  - Day of the week (from 0 to 6)



CronJob

CronJobs은 Job을 시간 기준으로 주기적으로 반복해서 실행한다. Cronjob은 백업 실행이나 email 전송과 같은 주기적이고 반복적인 작업을 만드는 데 유용하다.

Schedule
cronjob schedule 은 리눅스나 유닉스 cronjob 명령의 포맷과 동일한 방식으로 적용한다. default time은 UTC이다.
Minutes (from 0 to 59)
Hours (from 0 to 23)
Day of the month (from 1 to 31)
Month (from 1 to 12)
Day of the week (from 0 to 6)

매일 아침 9시 정각에 실행
schedule: "0 9 * * *"

5분마다 한 번씩 반복해서 실행
schedule: "*/5 * * * *"

매주 토요일 새벽 5시에 반복해서 실행

매월 1일 저녁 11:30분에 반복해서 실행

cronjob 실패 처리
CronJob 컨트롤러가 cronjob이 등록된 일정을 실행할 수 없으면 실패로 간주한다. 이후 Crontab controller는 10초마다 한 번씩 검사하여 실패한 작업을 재 시도하는데, 실패횟수가 100번에 도달하면 해당 Job은 실패로 처리하고 더 이상 시도하지 않는다.

StartingDeadlineSeconds
지정된 시간에 cronjob이 실행되지 못했을 때 해당 cronjob 작업을 실행하지 않도록 한다.
아래 예는 600초 내에 등록된 cronjob의 job이 start 되지 못하면 해당 작업을 취소한다.
startingDeadlineSeconds에 값을 할당하면, 실패한 작업을 재시도할 때 startDeadlineSeconds로 할당된 시간동안은 100번의 실패횟수를 카운트하지 않고 재시도 한다.

```
spec:
  schedule: "*/1 * * * *"
  startingDeadlineSeconds: 600
  concurrencyPolicy: Forbid
  jobTemplate:
    ....
```

concurrencyPolicy
concurrencyPolicy를 Forbid로 설정 시 job 을 동시에 실행하지 않도록 한다.

Job Example

Job-template	CronJob-template

apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
 	
  template:
  spec:
  containers:
  - name: centos-container
  image: centos:7
  command: ["bash"]
  args:
  - "-c"
  - "echo 'Hello'; sleep 5; echo 'Bye'"
  restartPolicy: Never
	apiVersion: batch/v1
	kind: **CronJob**
	metadata:
	  name: hello
	spec:
	  schedule: "*/1 * * * *"
	  jobTemplate:
	  spec:
	  template:
	  spec:
	  containers:
	  - name: hello
	  image: busybox
	  command:
	  - /bin/sh
	  - -c
	  - date; echo Hello
	  restartPolicy: OnFailure

Job Example

1. 앞서 실행했던 job-example.yaml 파일을 이용해 cronjob.yaml 파일 생성하자.
   # cp job-example.yaml cronjob-example.yaml
   # vi cronjob-example.yaml
   apiVersion: batch/v1
   kind: CronJob
   metadata:
     name: cronjob-example
   spec:
     schedule: "*/1 * * * *"
     jobTemplate:
       spec:
         template:
           spec:
             containers:
             - name: hello
               image: busybox
               command:
               - /bin/sh
               - -c
               - date; echo Hello
             restartPolicy: OnFailure

   위의 job은 언제 반복해서 실행되나?

2. CronJob을 동작시키자
   # kubectl apply -f cronjob-example.yaml
   cronjob.batch/cronjob-example created

   job이 1분마다 한번씩 반복해서 실행되고 있나?

   보존되는 Pod는 총 몇개인가?

3. 실행한 cronjob을 종료시키자
   # kubectl delete cronjobs.batch cronjob-example

심화 | CronJob 운영

CronJob 운영

1. 앞서 동작되는 cronjob-example을 다음의 조건으로 수정하자.

   - Cronjob 실행 조건
     매일 새벽 5시 정각에 실행

   - 적용 spec
     schedule: "0 5 * * *"
     startingDeadlineSeconds: 300
     concurrencyPolicy: Forbid

지금까지 배운 내용을 기준으로

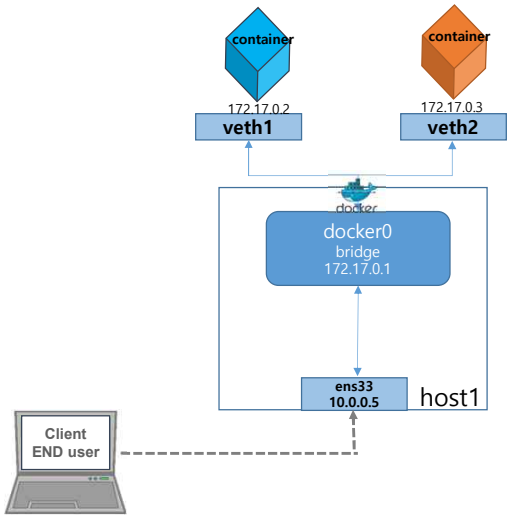
- 쿠버네티스 컨트롤러
  - 컨트롤러를 설명할수 있어야 합니다.
  - 다양한 컨트롤러의 특성을 설명할수 있어야 합니다.
  - DaemonSet이나 StatefulSet 타입의 애플리케이션 동작할 수 있어야 합니다.
  - Pod를 scale-out, scale-in 할수 있어야합니다.

- Deployment
  - rolling update를 하면서 업데이트 상태를 history에 기록되도록 운영할 수 있어야 합니다.
  - 서비스 rollback 할수 있어야 합니다.

- Job & CronJob
  - 애플리케이션이 주기적으로 동작하도록 구성할수 있어야 합니다.

CNI 이해

Docker Network



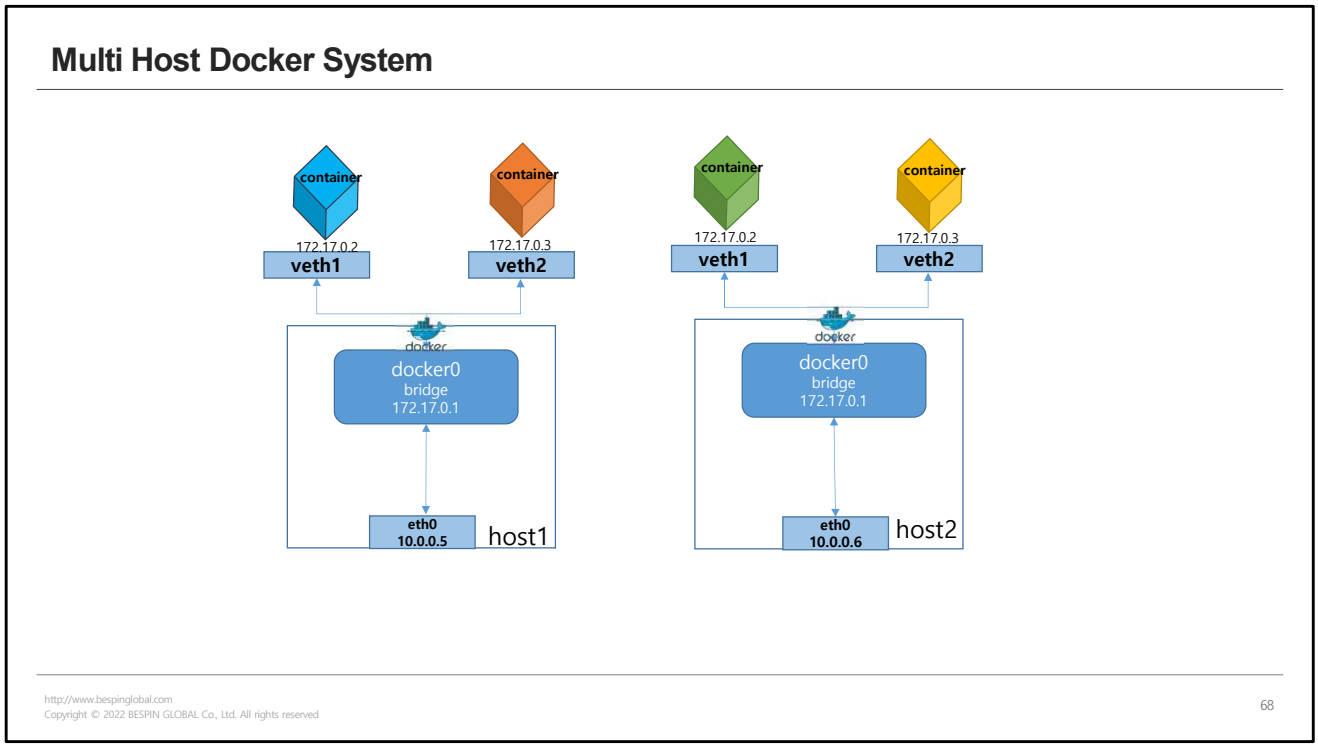
- 브릿지 네트워크는 모든 Docker 호스트에 존재
- docker0
  - Docker 호스트의 default network
  - 컨테이너를 NAT를 사용하여 호스트 외부 네트워크로 포워딩
  - 172.17.0.0/16을 사용하여 DHCP로 컨테이너에 IP address 제공
  - 172.17.0.1이 docker0에 할당
  - 컨테이너에 할당된 IP 확인: docker inspect

Docker Network

도커가 동작되는 시스템에서는 docker0 인테페이스를 볼수 있다.
도커 컨테이너는 docker0라는 인터페이스를 Bridge 타입으로 운영하면서 컨테이너에 IP Address(Endpoint)를 할당한다.
도커는 각 컨테이너에 외부와 네트워크를 제공하기 위해 컨테이너마다 가상 네트워크 인터페이스를 호스트에 생성하고, 이 인터페이스 이름을 veth로 시작한다.
veth는 사용자가 직접 생성할 필요는 없으며 컨테이너가 생성될 때 자동으로 생성한다.
그리고 docker0가 각 veth 인터페이스와 바인딩되어 호스트의 eth0 인터페이스와 이어주는 역할을 한다.

```
# ip addr
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:41:5f:4c:28 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:41ff:fe5f:4c28/64 scope link
       valid_lft forever preferred_lft forever
```

브리지 네트워크
docker0늘 브리지 네트워크이다.
docker0 브리지는 172.17.0.X IP 대역을 컨테이너에 순차적으로 할당한다. 이와 마찬가지로 **브리지 네트워크는 docker0이 아닌 사용자 정의 브리지를 새로 생성해서 각 컨테이너에 연결하는 네트워크 구조**이다. 이로써 컨테이너는 연결된 브리지를 통해 외부와 통신할 수 있게 된다.



Multi Host Docker System

도커 네트워크는 bridge기반으로 잘 만들어진 네트워크 이다. 도커 호스트내에서 컨테이너 간의 통신은 docker0가 할당해주는 IP Address를 이용한다.

그러면 도커 호스트가 두 대 이상인 경우를 생각해보자.

위의 그림에서 blue 컨테이너와 green 컨테이너는 둘 다 IP Address 172.17.0.2번을 가진다.

다음 질문에 답해 보자.

1. blue 컨테이너와 orange 컨테이너가 통신이 가능한가? (yes / no)

2. Green 컨테이너와 yellow 컨테이너가 통신이 가능한가? (yes / no)

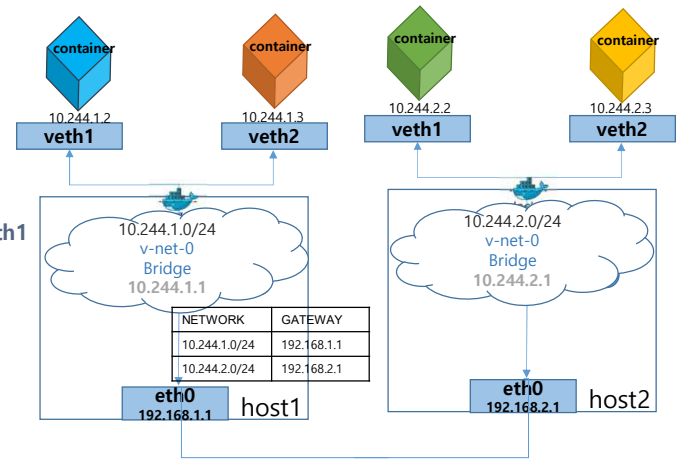
3. Blue 컨테이너와 green 컨테이너가 통신이 가능한가? (yes / no)

4. Blue 컨테이너와 green 컨테이너가 통신이 가능하지 않다면 왜 그런가?

5. 컨테이너와 green 컨테이너가 통신이 가능하도록 하기 위해서 무엇을 해야 할까?

Overlay Network

```
ip netns add v-net-0
ip link add v-net-0 type bridge
ip link set dev v-net-0 up

ip link add veth0 type veth peer name veth1
ip link set veth1 netns  v-net-0
ip link set veth0 up
ip addr add 10.244.1.1/24 dev veth0

ip netns exec v-net-0 ip link set veth1up
ip netns exec v-net-0 ip addr add 10.244.1.2/24 dev veth1

ip netns exec v-net-0 ip route add 10.244.1.0/24 via
192.168.1.1
iptables –t nat –A POSTROUTING –s 10.244.0.0/24 –j
MASQUERADE

ip route add 10.244.1.0 via 192.168.1.1
ip route add 10.244.2.0 via 192.168.2.1
```



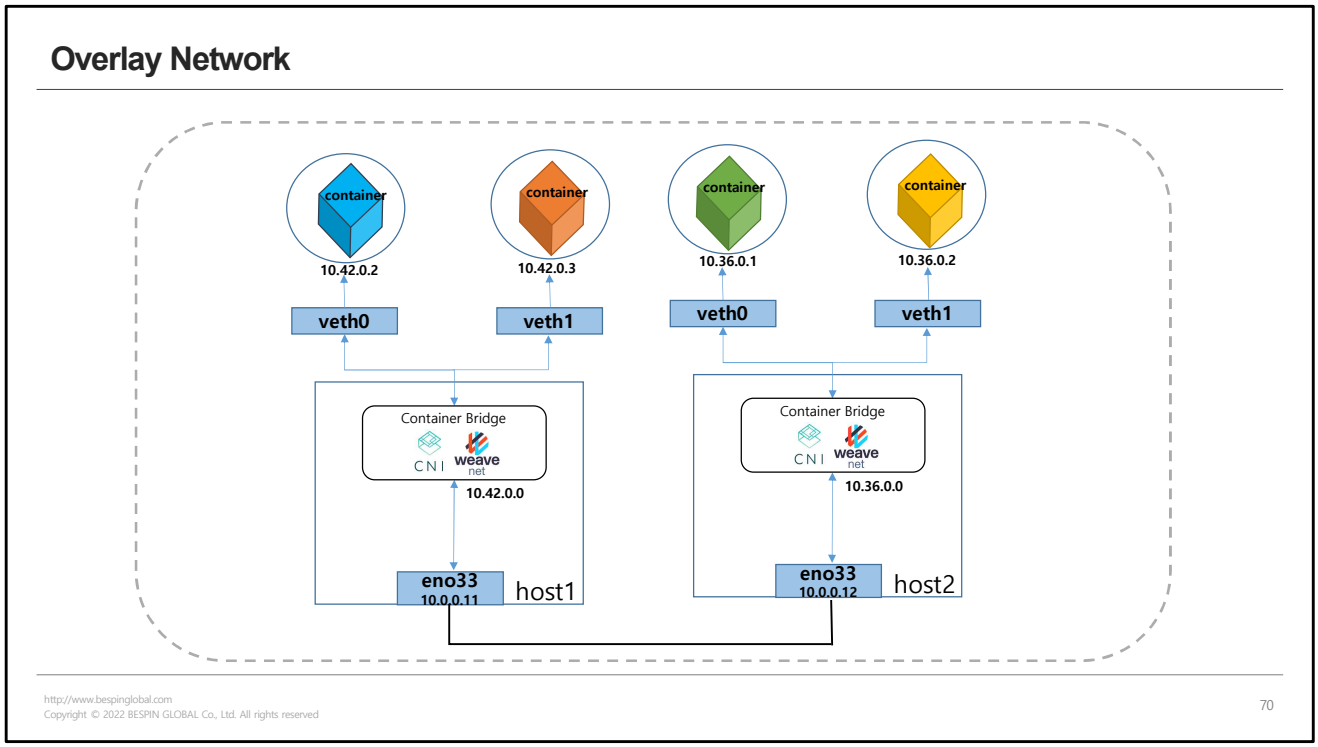
Overlay Network

위의 그림은 리눅스 커널이 가지고 있는 network namespace 기술을 이용해 멀티 호스트 컨테이너간 통신이 가능하도록 구현해 본 것이다.
완벽한 명령어 리스트는 아니지만 위의 코드를 가지고 blue 컨테이너가 green 컨테이너와 통신이 가능해진다.

1. blue 컨테이너와 green 컨테이너는 서로 다른 IP 주소를 가지고 있다.

2. blue 컨테이너에서 green 컨테이너로 통신을 요청할 때
   - 10.244.1.2 ----> 10.244.1.1 (v-net-0)
   - v-net-0 --> router를 통과하며 NAT주소 변환된다. 192.168.1.1
   - host2번의 eth0와 통신 되고 이후 host2의 v-net-0를 통과해서 10.244.2.2번으로 연결된다.

확인한 것처럼 중간에 v-net0, router, NAT 서비스를 기준으로 blue와 green 컨테이너는 통신할 수 있었다.



Overlay Network

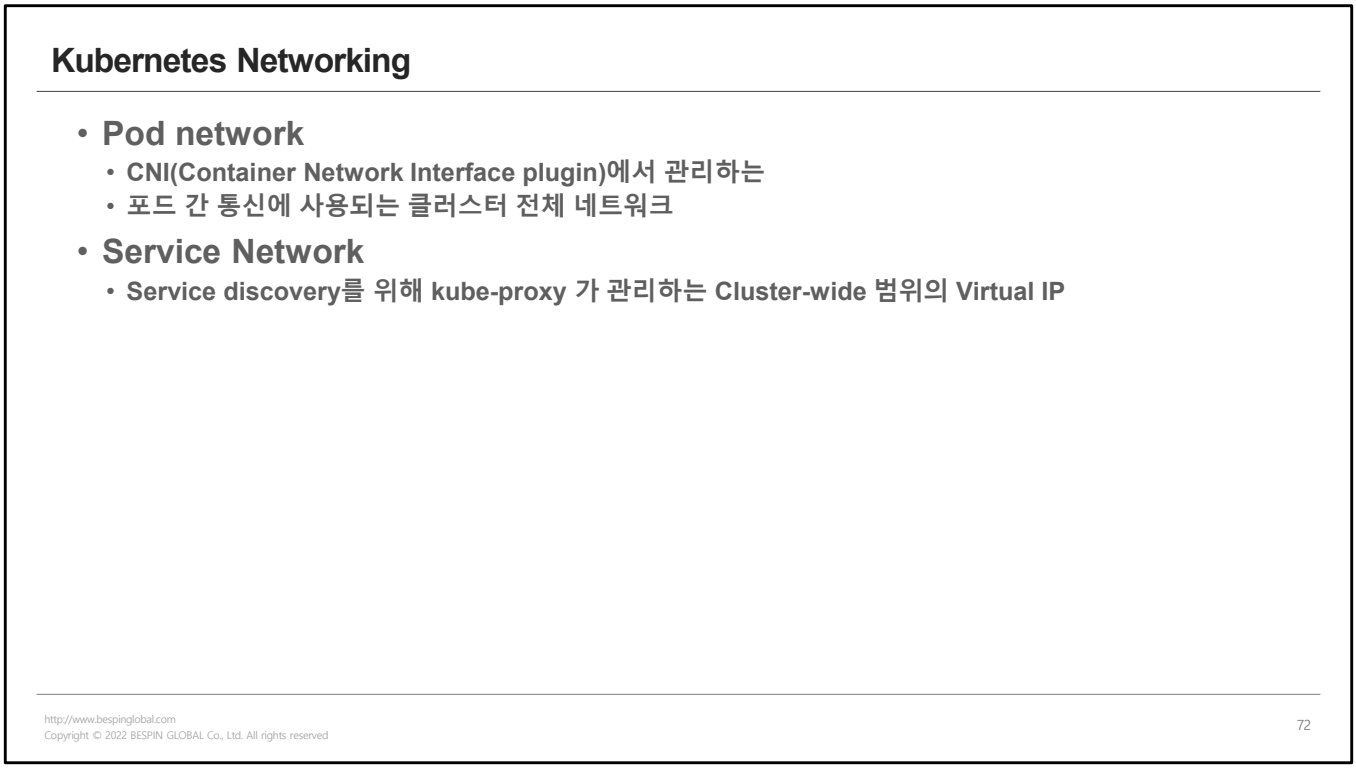
멀티 호스트 컨테이너 간 통신을 지원하기 위해 앞서 v-net-0를 확인해보았다.
구현해서 network namespace와 gateway를 구성해야 한다면 과연 몇 명이나 구성하고 운영할 수 있을까?
누군가 만들어서 여러분이 직접 이러한 v-net-0를 공해주면 좋겠다는 생각은?? 필자만 하는 것은 아닐 것이다.

많은 커뮤니티와 벤더들이 멀티 호스트에서 컨테이너간 통신이 가능하도록 Container Network Interface를 지원한다. VLAN, Overlay와 같은 세밀하게 따지면 다르지만 궁극적으로 앞서 살펴본 v-net-0와 같은 환경을 만들어주는 프로그램들이다.

- Flannel
- Weave
- Project Calico

쿠버네티스에 적용할 수 있는 CNI는 https://kubernetes.io/docs/concepts/cluster-administration/addons/ 에서 확인해볼 수 있다.

Kubernetes Service



Kubernetes Networking

Service Network
Pod는 임시적인 특성을 가지기 때문에 Pod와 통신을 하기 위해서는 Pod의 Endpoint IP Address를 단일 포인트로 관리할 수 있는 서비스가 필요하다.
이러한 서비스를 지원하는 것이 Kubernetes Service Network이다.

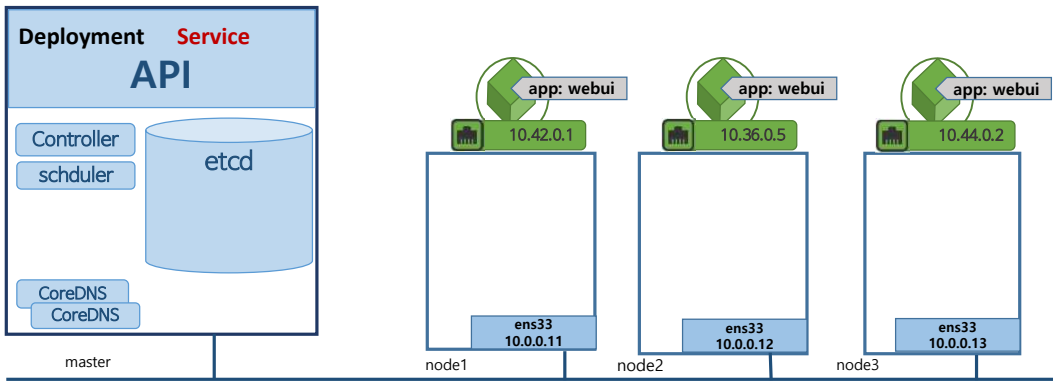
kubernetes Service Network은 다음의 특성을 가진다.
- proxy 서버 스스로 내구성이 있어야 하며 장애에 대응할 수 있어야 한다.
- 트래픽을 전달할 Pod의 Endpoint 리스트를 가지고 있어야 한다.
- Pod들이 정상적으로 동작되는지 확인할 수 있어야 한다.

Service는 kubernetes API 오브젝트로 Pod로 트래픽을 포워딩 해주는 Proxy 역할을 한다.

Pod Network과 동일하게 service network 또한 가상 IP 주소이다.
Pod 네트워크는 실질적으로 가상 이더넷 네트워크 인터페이스(veth)가 세팅되어져 ifconfig에서 조회할 수 있지만, service 네트워크는 ifconfig로 조회할 수 없다. 또한 routing 테이블에서도 service 네트워크에 대한 경로를 찾아볼 수 없다. 이러한 이유는 service네트워크의 구조와 동작 방식을 통해 확인할 수 있다.

Service Type

- ClusterIP, NodePort, LoadBalancer, ExternalName



Service Type

Kubernetes Service 동일한 서비스들을 제공하는 Pod그룹에 단일 진입점을 만들기 위해 생성하는 리소스이다.
각 서비스에는 서비스가 존재하는 동안 절대로 변경되지 않는 IP 주소와 port가 있고, 클라이언트는 해당 **IP:port**에 연결할 수 있고, 이 연결은 서비스를 제공하는 Pod중 하나로 라우팅 된다.
서비스 클라이언트는 서비스를 제공하는 Pod의 개별 위치와 무관해지므로 Pod는 언제든 클러스터 주변에서 움직일 수 있다.

서비스 관리

- 서비스를 지원하는 Pod는 하나 또는 그 이상일 수 있다.
- 서비스로의 연결은 LoadBalancer 될 수 있다.
- 서비스의 Pod들은 label selector를 통해 관리된다.

쿠버네티스 서비스의 4가지 타입

ClusterIP(default)
- Pod 그룹의 단일 진입점(Virtual IP) 생성

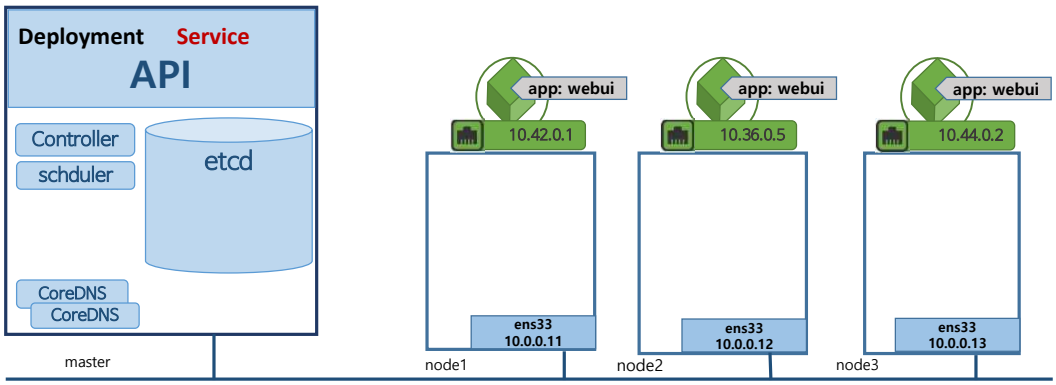
NodePort
- ClusterIP 가 생성된 후
- 모든 Worker Node에 외부에서 접속가능 한 포트가 예약

LoadBalancer
- 클라우드 인프라스트럭처(AWS, Azure, GCP 등)나 오픈스택 클라우드에 적용
- LoadBalancer를 자동으로 프로 비전하는 기능 지원

ExternalName
- 클러스터 안에서 외부에 접속 시 사용할 도메인을 등록해서 사용
- 클러스터 도메인이 실제 외부 도메인으로 치환되어 동작

ClusterIP

- selector의 label이 동일한 pod 들을 group으로 묶어
- 단일 진입점 (Virtual_IP)을 생성
- 클러스터 내부에서만 사용가능
- IP 할당 생략 시 10.96.0.0/12(10.96.0.0 ~10.111.255.255) 범위내에서 random한 IP가 할당됨



Cluster IP

서비스를 지원하는 Pod는 하나 또는 그 이상일 수 있다. 서비스로의 연결은 backend의 모든 Pod들 사이에 loadblancing 될 수 있다.
서비스 그룹에 들어가는 Pod는 label selector를 통해 검색된다.

ClusterIP

- 동일한 서비스를 제공하는 Pod 그룹에 단일 진입점을 제공
- Default Type으로 --type 옵션 생략시 적용된다.
- selector를 지정하여 Pod group을 구성한다.
- 각 Pod의 endpoint의 묶음
- CLI :

  kubectl expose deployment webserver **--type=ClusterIP --port=8080**

ClusterIP example

ClusterIP Service

```
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  clusterIP: 10.100.100.100
  selector:
    app: webui
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

$ kubectl create -f deployment-webui.yaml
$ kubectl get pod --show-labels

```
NAME                     READY   STATUS    RESTARTS   AGE   LABELS
webui-6d4c4cc4b8-66bff   1/1     Running   0          10s   app=webui,pod-template-hash=6d4c4cc4b8
webui-6d4c4cc4b8-rdv4q   1/1     Running   0          10s   app=webui,pod-template-hash=6d4c4cc4b8
webui-6d4c4cc4b8-vszsr   1/1     Running   0          10s   app=webui,pod-template-hash=6d4c4cc4b8
```

$ kubectl create -f clusterip-nginx.yaml
$ kubectl get svc

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
clusterip-service   ClusterIP   10.100.100.100   <none>        80/TCP    5s
kubernetes          ClusterIP   10.100.0.1       <none>        443/TCP   4h27m
```

$ kubectl describe svc clusterip-service

$ curl 10.100.100.100

문제: ubuntu 시스템에서 curl 10.100.100.100을 실행하면?
문제: pod 1개가 삭제되면 무슨일이 생기나?

$ kubectl delete svc clusterip-service

ClusterIP Example

smlinux/appjs를 실행하는 rc-appjs.yaml을 실행하고 app=appjs 라벨을 가지는 Pod들을 하나의 서비스로 묶어보자.

1. 먼저 appjs Pod를 실행한다. yamls 디렉토리에서 rs-appjs.yaml을 실행하여 appjs 포드를 3개 실행한다.

```
# cat rs-appjs.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-appjs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: appjs
  template:
    metadata:
      labels:
        app: appjs
    spec:
      containers:
      - name: appjs-container
        image: smlinux/appjs
        ports:
        - containerPort: 8080
```

```
# kubectl apply -f rs-appjs.yaml
# kubectl get pod,replicaset
```

2. ClusterIP를 생성하여 위에서 만들어진 3개의 Pod의 Endpoint를 묶어보자.

```
# cat svc-appjs.yaml
apiVersion: v1
kind: Service
metadata:
  name: appjs-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: appjs
```

```
# kubectl apply -f svc-appjs.yaml
```

3. cluster IP를 확인해본다.

 # kubectl get pod --show-labels

```
NAME              READY   STATUS    RESTARTS   AGE   LABELS
rs-appjs-plqwm    1/1     Running   0          25s   app=appjs
rs-appjs-q7mbw    1/1     Running   0          25s   app=appjs
rs-appjs-w6sbb    1/1     Running   0          25s   app=appjs
```

 # kubectl get svc -o wide

```
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE   SELECTOR
appjs-service   ClusterIP   10.108.183.25   <none>        80/TCP    90s   app=appjs
```

4. curl <cluster IP>:8080 명령을 여러 번 반복 실행해본다. 어떤 컨테이너의 웹페이지가 표시되는가?

 # curl 10.108.183.25

Container Hostname: rs-appjs-w6sbb

```
# for n in $(seq 10)
do
  curl 10.108.183.25
  sleep 1
done
```

10번의 출력 결과를 통해 무엇을 확인하였나?

ClusterIP Address는 LB인가?

5. Pod 전체를 삭제하면 ReplicaSet controller는 새로운 pod를 생성할 것이다. 이때 Pod들의 IP Address 가 변경되는데, Service에는 어떻게 영향을 주는지 알아보자.

 # kubectl delete pod --all

새로운 Pod들이 생성되었다.

 # kubectl get pods -o wide

service 오브젝트의 세부 항목을 살펴보자.

 # kubectl describe svc appjs-service

6. Cluster-IP로 Access해보자. Pod를 삭제하기 전과 비교하여 달라진 것이 있는가?

7. scale-out을 통해 appjs POD의 수가 늘어나거나 줄어들면 어떻게 될까?

 # kubectl scale rs rs-appjs --replicas=2

replicaset.apps/rs-appjs scaled

 # kubectl describe svc appjs-service

```
Name:              appjs-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=appjs
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.108.183.25
IPs:               10.108.183.25
Port:              <unset>  80/TCP
TargetPort:        8080/TCP
Endpoints:         10.36.0.6:8080,10.46.0.5:8080
Session Affinity:  None
Events:            <none>
```

무엇을 확인하였나?

8. 지금까지으 실습을 통해 확인한 ClusterIP 에 대해 자신의 생각을 간단히 정리해보자.

Session Affinity

- 특정 클라이언트에서 생성된 모든 요청을 매번 같은 Pod로 연결
- session affinity type
  - None
  - ClientIP

ClusterIP Service	ClusterIP Service(w sessionAffinity)

apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  sessionAffinity: None
  selector:
  app: webui
  ports:
  - protocol: TCP
  port: 80
  targetPort: 80
	apiVersion: v1
	kind: Service
	metadata:
	  name: clusterip-service
	spec:
	  type: ClusterIP
	  sessionAffinity: ClientIP
	  selector:
	  app: webui
	  ports:
	  - protocol: TCP
	  port: 80
	  targetPort: 80

SessionAffinity

앞서살펴본 Cluster 애플리케이션 서비스를 운영하다 보면 ClusterIP를 통해 접속할 때마다 Pod가 바뀌는 것을 원하지 않을때가 있다. 다시 말해 첫번째 접속했던 Pod를 계속 사용하고 싶은것이다.
이럴때 사용하는 기능이 SessionAffinity이다.

SessionAffinity는 클라이언트에서 생성된 모든 요청이 매번 같은 Pod로 redirect 되도록 할 때 sessionAffinity값을 기본 값인 None 대신 ClientIP로 설정한다.

```
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  sessionAffinity: ClientIP
  selector:
    app: webui
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

클라이언트 IP 의 세션이 유지되는 시간은 sessionAffinity의 timeoutSeconds에 설정되어 있고, 기본 값은 10800(for 3 hours)초로 구성되어 있다.
timeout변경 시 다음의 정보를 추가한다.

```
..
spec:
  sessionAffinity: ClientIP
  sessionAffinityConfig:
    clientIP:
      timeoutSeconds: 10
...
```

실습 : 간단한 실습을 통해 SessionAffinity를 확인해보자.

1. 동작중인 애플리케이션의 label을 확인하자.

```
# kubectl get pod --show-labels
NAME              READY   STATUS    RESTARTS   AGE   LABELS
rs-appjs-q7mbw    1/1     Running   0          11m   app=appjs
rs-appjs-w6sbb    1/1     Running   0          11m   app=appjs
```

2. svc-appjs 파일을 svc-affinity 파일로 복사해서 사용하자.
   appjs service에 sessionAffinity 기능을 추가하자

```
# cp svc-appjs.yaml svc-affinity.yaml

# vi svc-affinity.yaml
apiVersion: v1
kind: Service
metadata:
  name: appjs-service-affinity
spec:
  sessionAffinity: ClientIP
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: appjs
```

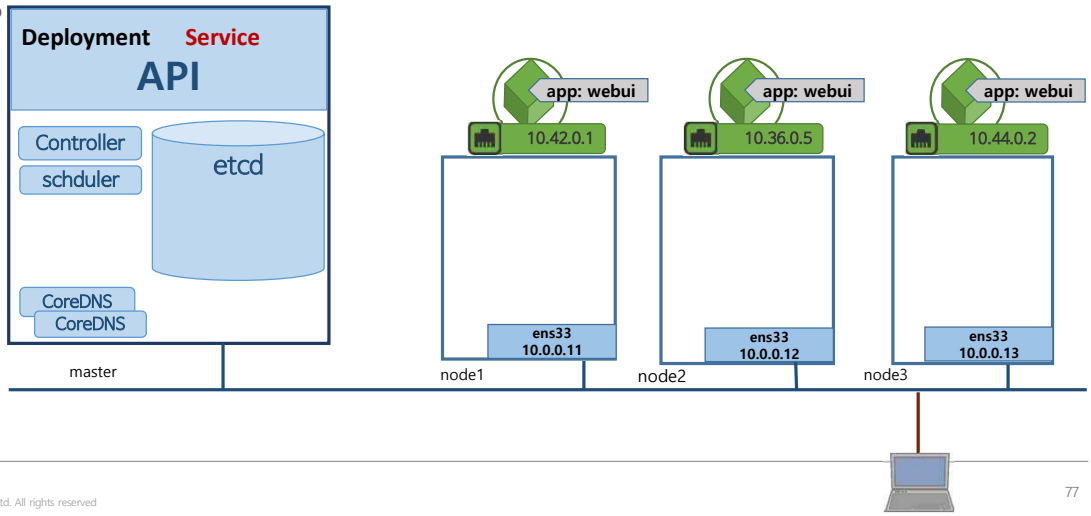
3. 서비스 동작 후 접속 TEST 할 때 어떤 결과가 나오는지 확인해보자.

```
# kubectl apply -f svc-affinity.yaml
service/appjs-service-affinity created

# kubectl get svc
NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
appjs-service          ClusterIP   10.108.183.25    <none>        80/TCP    18m
appjs-service-affinity ClusterIP   10.111.231.174   <none>        80/TCP    6s
kubernetes             ClusterIP   10.96.0.1        <none>        443/TCP   5d17h

# for n in $(seq 10); do    curl 10.111.231.174;   sleep 1; done
Container Hostname: rs-appjs-q7mbw
Container Hostname: rs-appjs-q7mbw
Container Hostname: rs-appjs-q7mbw
...
```

결과를 설명해 보자.

외부에 서비스 노출

- 외부 클라이언트에 서비스 노출
  - NodePort
  - LoadBalancer
  - ExternalIP



외부에 서비스 노출하기

cluster-IP는 여러 개의 Pod들의 단일진입점을 제공해주기는 하지만, 쿠버네티스 내부에서만 연결할 수 있다.
Kubernetes는 쿠버네티스 외부에서 Service API 오브젝트에 Access 가능하게 하는 방법을 제공한다.

NodePort
- NodePort 서비스는 각 클러스터 노드는 노드 자체의 이름을 통해 포트를 열고 포트에서 발생한 트래픽을 서비스로 redirect한다.

LoadBalancer
- kubernetes가 실행중인 클라우드 인프라스트럭처에 프로비전 된 LoadBalancer를 통해 서비스 액세스 가능하다.
- LoadBalancer는 발생한 트래픽을 모든 노드에서 노드 포트로 리디렉트 한다.
- 클라이언트는 LoadBalancer IP를 통해 서비스에 접속한다.

ExternalIP
- 하나 이상의 클러스터 노드로 라우팅되는 외부IP가 있는 경우 Kubernetes 서비스를 External IP로 노출 시킬 수 있다.

ExternalIP

- 하나 이상의 클러스터 노드로 라우팅 되는 외부IP가 있는 경우 Kubernetes 서비스를 External IP로 노출 시킬 수 있다

Service-template	externalIP-template

apiVersion: v1
kind: Service
metadata:
  name: appjs-service
spec:
  ports:
  - port: 80
  targetPort: 8080
  selector:
  app: appjs
	apiVersion: v1
	kind: Service
	metadata:
	  name: appjs-service
	spec:
	  ports:
	  - port: 80
	  targetPort: 8080
	  selector:
	  app: appjs
	  externalIPs:
	  - node_IP_Address

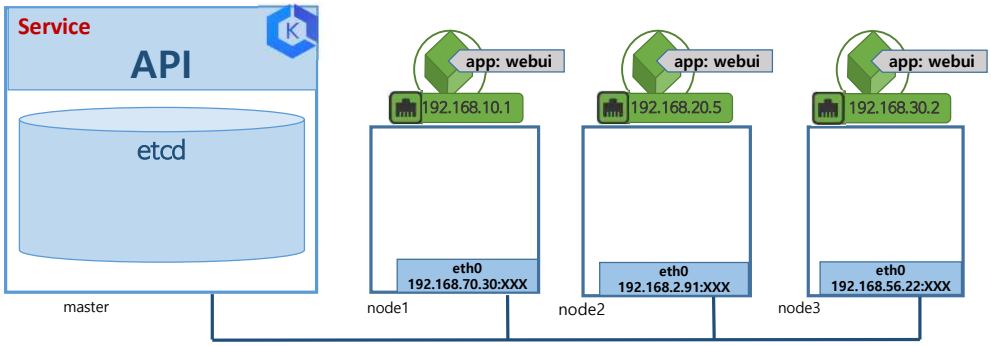
ExternalIP

ExternalIP하나 이상의 클러스터 노드로 라우팅되는 외부IP가 있는 경우 Kubernetes 서비스를 External IP로 노출 시킬 수 있다.

```
apiVersion: v1
kind: Service
metadata:
  name: appjs-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: appjs
  externalIPs:
  - 10.0.0.11
```

NodePort

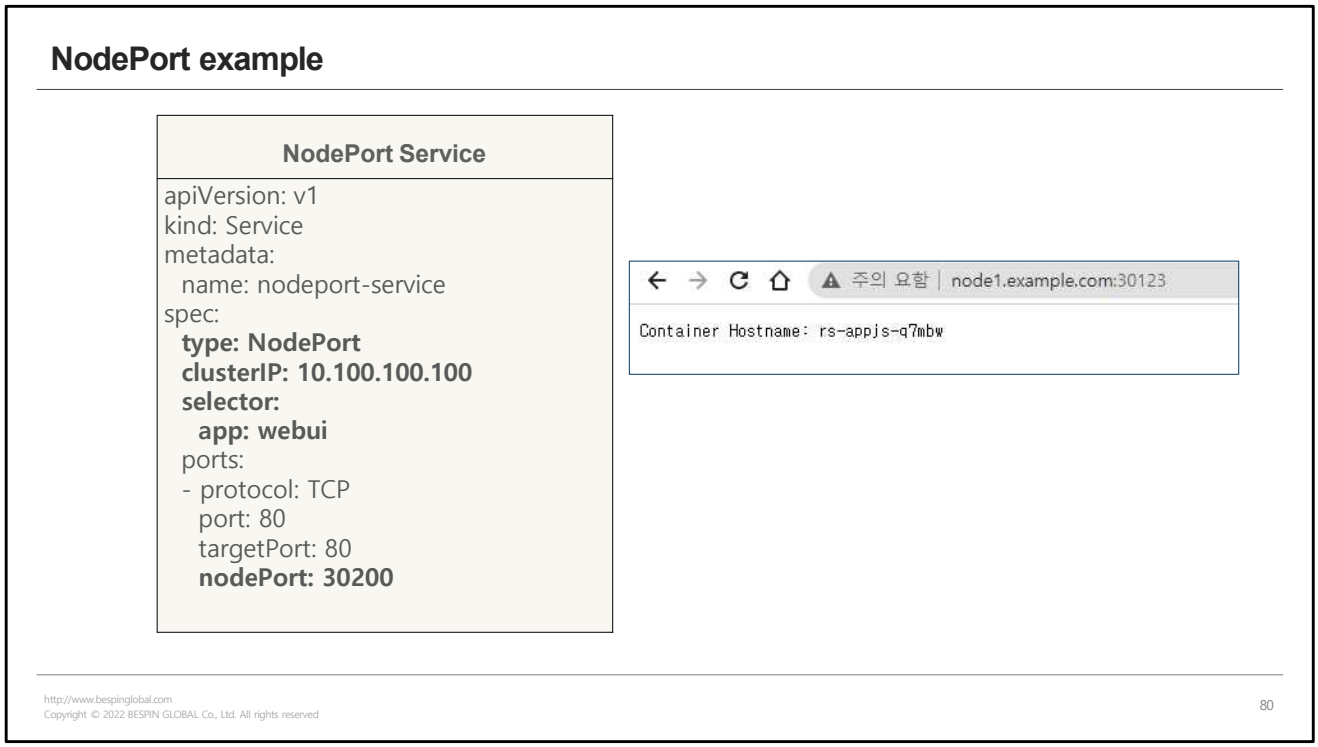
- 모든 노드를 대상으로 외부 접속 가능한 포트를 예약
- Default NodePort 범위: 30000-32767
- ClusterIP 를 생성하고 추가로 NodePort를 예약



NodePort

외부 클라이언트에 Pod set을 노출하는 첫 번째 방법은 Service를 생성하고 그 타입을 NodePort로 지정하는 것이다.
NodePort서비스를 생성해 kubernetes가 모든 노드를 대상으로 포트를 예약한다.
NodePort는 30000-32767 범위로, default로 random하게 할당된다.

```
apiVersion: v1
kind: Service
metadata:
  name: appjs-service
spec:
  type: NodePort
  sessionAffinity: None
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30123
  selector:
    app: appjs
```



NodePort

실습: 실습을 통해 NodePort를 열어보자.

1. **svc-appjs.yaml** 파일을 **svc-node.yaml**로 복사한 후 다음과 같이 **type과 nodeport number를 고정**으로 할당하자.

```
# cp svc-appjs.yaml  svc-node.yaml
# cat svc-node.yaml
apiVersion: v1
kind: Service
metadata:
  name: appjs-service-nodeport
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30123
  selector:
    app: appjs

# kubectl apply -f svc-node.yaml
```

2. 동작 중인 서비스 목록을 확인해보자.

```
# kubectl get services
NAME                        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
appjs-service-nodeport      NodePort    10.102.83.10    <none>        80:30123/TCP     15s
```

ClusterIP와 NodePort의 차이점을 확인해보자.

- Nodeport에는 PORT(S) 필드에 80:30123이 추가로 포함되어 있다. 앞의 80은 clusterIP의 port이고, 뒤의 **30123은 node IP의 port 번호**이다.
- 모든 node의 kubeproxy는 iptables의 rule을 추가해서 **NODE_IP:30123** 포트로 연결되는 모든 커넥션을 pod로 전달하고 있다.

3. External 환경에서 NodePort로 접속을 시도해보자.

ClusterIP로 연결시도

```
# curl  10.102.83.10
```

worker node의 NodePort로 연결시도

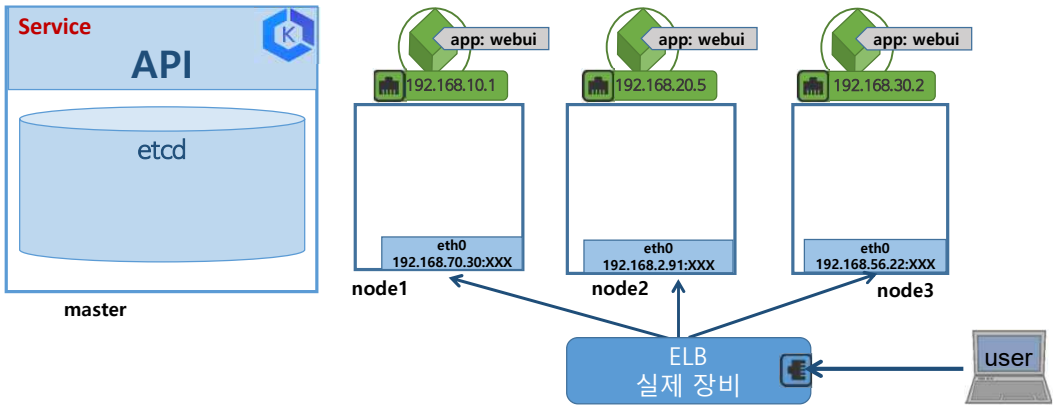
```
# curl 10.0.0.11:30123
Container Hostname: rs-appjs-w6sbb
```

웹 브라우저를 통해서도 연결되는지 확인해보라.

http://node1.example.com:30123

LoadBalancer

- Public 클라우드(AWS, Azure, GCP 등)에서 운영가능
- LoadBalancer를 자동으로 구성 요청
- NodePort를 예약 후 해당 nodeport로 외부 접근을 허용



LoadBalancer

kubernetes는 클라우드 인프라스트럭처(AWS, Azure, GCP 등)로 부터 LoadBalancer를 자동으로 프로비전하는 기능을 지원한다. 이때는 service type를 NodePort에서 LoadBalancer로 설정한다.
kubernetes가 LoadBalancer 서비스를 지원하지 않는 환경에서 실행하고 있다면 로드 밸런서는 프로비전 되지 않는다. NodePort와 같이 동작한다.

LoadBalancer example

LoadBalancer Service

```
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer-service
spec:
  type: LoadBalancer
  selector:
    app: webui
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

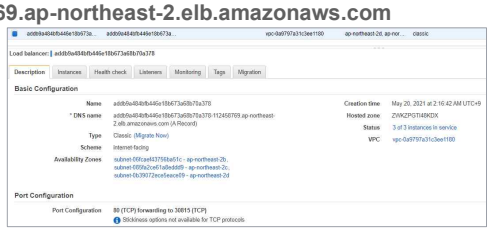
$ cp clusterip-nginx.yaml loadbalancer-nginx.yaml
$ vi loadbalancer-nginx.yaml
<yaml 파일을 loadbalancer 타입으로 변경>

$ kubectl create –f loadbalancer-nginx.yaml
$ kubectl get svc

```
NAME                  TYPE           CLUSTER-IP      EXTERNAL-IP                                                              PORT(S)        AGE
kubernetes            ClusterIP      10.100.0.1      <none>                                                                   443/TCP        10h
loadbalancer-service  LoadBalancer   10.100.139.254  addb9a484bfb446e18b673a68b70a378-112458769.ap-northeast-2.elb.amazonaws.com  80:30815/TCP   65s
```

연결 확인, CLI와 웹 브라우저 통해 확인
$ curl addb9a484bfb..769.ap-northeast-2.elb.amazonaws.com

생성된 ELB 확인
<EC2> - <Load Balancers>



$ kubectl delete svc loadbalancer-service

LoadBalancer

GCP(Google Cloud Platform)에서 LoadBalancer
LoadBalancer 유형의 서비스를 만들면 Google Cloud 컨트롤러가 작동하여 네트워크 부하 분산기를 구성한다. 컨트롤러가 네트워크 부하 분산기를 구성하고 안정적인 IP 주소를 생성할 때까지 1분 정도 기다린 후 서비스를 확인하면 다음과 같은 결과를 볼 수 있다.

```
# kubectl get svc loadbalance-svc
NAME                    CLUSTER-IP                EXTERNAL-IP                PORT(S)
                AGE
kubia-loadbalancer      10.111.241.153    130.211.53.173    80:32143/TCP 1m

# curl http://130.211.53.173
```

참조링크 : https://cloud.google.com/kubernetes-engine/docs/how-to/exposing-apps?hl=ko

AWS annotation을 이용한 LoadBalancer 설정
기본적으로 type: LoadBalancer의 서비스를 배포 시 외부(public) Classic Load Balancer가 생성된다.
만약 CLB 대신 Network Load Balancer를 배포하려면 annotations 을 아래와 같이 추가해야 한다.

```
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb
spec:
  type: LoadBalancer
  selector:
    app: webui
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

참조링크: https://aws.amazon.com/ko/premiumsupport/knowledge-center/eks-kubernetes-services-cluster/

AKS Loadbalancer 설정

Azure Load Balancer는 인 바운드 및 아웃 바운드 시나리오를 모두 지원하는 OSI(개방형 시스템 연결) 모델의 L4에 있습니다. 부하 분산 장치의 프런트 엔드에 도착하는 인바운드 흐름을 백 엔드 풀 인스턴스에 분산합니다.
AKS와 통합된 **공용** 부하 분산 장치는 두 가지 용도로 사용됩니다.

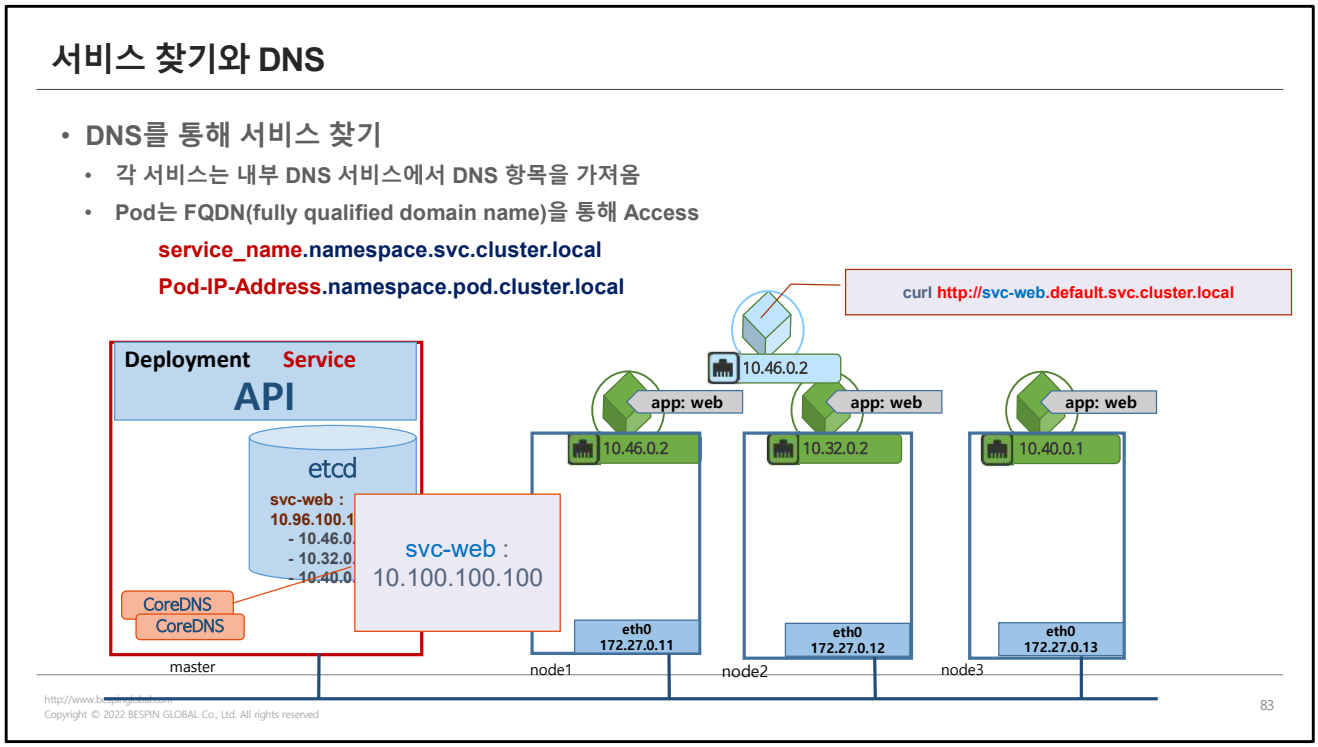
AKS 가상 네트워크 내의 클러스터 노드에 아웃 바운드 연결을 제공합니다. 노드 프라이빗 IP 주소를 해당 아웃 바운드 풀의 일부인 공용 IP로 변환하여 이 목표를 달성합니다.

Kubernetes 서비스 형식 LoadBalancer를 통해 애플리케이션에 대한 액세스를 제공합니다. 이를 통해 손쉽게 애플리케이션 크기를 조정하고 고가용 서비스를 만들 수 있습니다.

내부(또는 개인) 부하 분산 장치는 개인 IP만 프런트 엔드로 허용되는 경우에 사용됩니다. 내부 부하 분산 장치는 트래픽 부하를 가상 네트워크 내에 분산하는 데 사용됩니다. 하이브리드 시나리오의 온-프레미스 네트워크에서 부하 분산 장치 프런트 엔드에 액세스할 수도 있습니다.

```
apiVersion: v1
kind: Service
metadata:
  name: public-svc
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: public-app
```

참조링크: https://docs.microsoft.com/ko-kr/azure/aks/load-balancer-standard#use-the-public-standard-load-balancer



서비스 찾기와 DNS

서비스를 지원하는 Pod는 더 생성되기도 하고, 사라지기도 한다. 클라이언트 Pod가 서비스의 IP와 port를 어떻게 알 수 있을까? Kubernetes는 클라이언트 Pod가 서비스의 IP와 포트 번호를 알아낼 수 있는 방법을 제공한다.

환경변수를 통해 서비스 찾기

```
kubectl exec <podname> env
```

- 환경변수의 제한
  - Pod가 실행중인 상태에서 변경된 서비스의 상태 값은 적용되지 않음.
  - 서비스를 먼저 만들고 Pod가 추가되면 환경변수로 할당됨.
  - 간단히 동작중인 Pod를 삭제하고, 새로 생성된 Pod로 환경변수를 확인해보자.

DNS를 통해 서비스 찾기

```
service_name.namespace.svc.cluster.local
```

- kubernetes DNS는 클러스터에서 실행하는 모든 Pod가 사용할 수 있도록 구성된다.
  - Pod에서 수행하는 DNS query는 kubernetes 자체 DNS에서 처리.
  - 각 서비스는 내부 DNS 서비스에서 DNS 항목을 가져옴
  - Pod는 FQDN(fully qualified domain name)을 통해 Access.

coreDNS

- CoreDNS는 Kubernetes 클러스터 DNS 역할을 하는 유연하고 확장 가능한 DNS 서버이다.
- Kubernetes와 마찬가지로 CoreDNS 프로젝트는 CNCF에서 호스팅한다.
- kube-dns Service
    - CLUSTER-IP : 10.96.0.10
    - coreDNS Pod로 동작

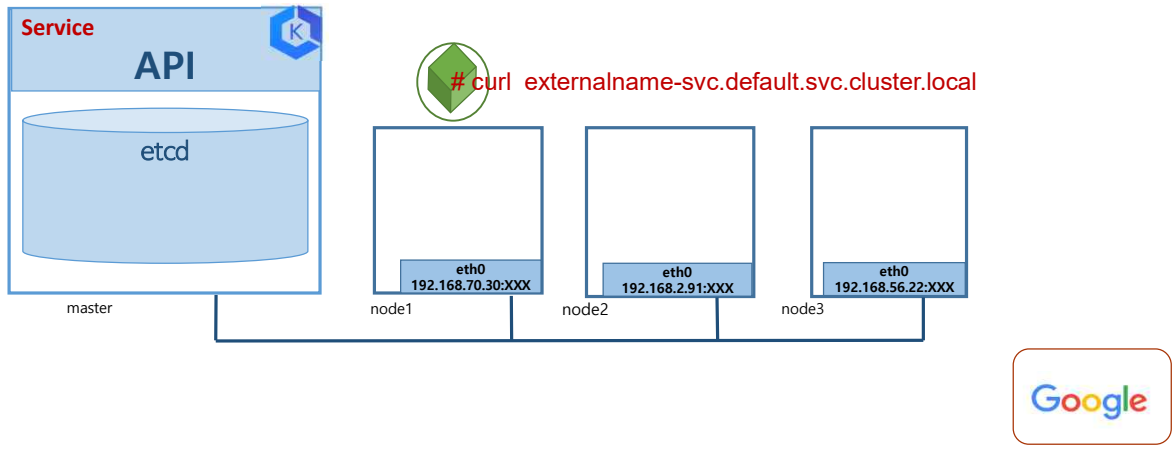
$ kubectl run frontend -it --image=centos:7

[root@frontend /]# cat /etc/resolv.conf
nameserver X.X.X.X
search default.svc.cluster.local svc.cluster.local cluster.local ap-northeast-2.compute.internal
options ndots:5

[root@frontend /]# **curl appjs-service.default.svc.cluster.local**

ExternalName

- 클러스터 내부에서 External(외부)의 도메인을 설정



http://www.bespinglobal.com

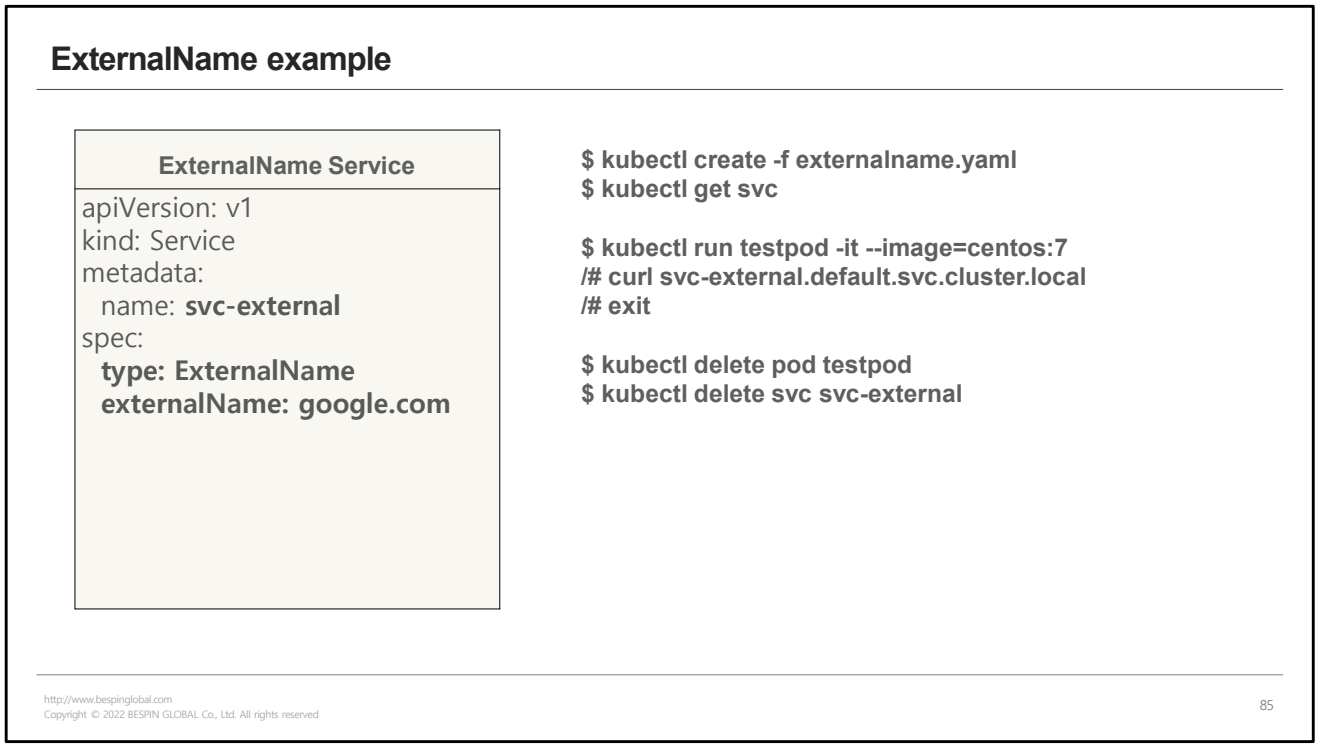
84

ExternalName

ExternalName은 외부 서비스를 쿠버네티스 내부에서 호출하고자할때 사용할 수 있다.

쿠버네티스 클러스터내의 Pod들은 클러스터 IP를 가지고 있기 때문에 클러스터 IP 대역 밖의 서비스를 호출하고자 하면, NAT 설정등 복잡한 설정이 필요하다.

특히 AWS 나 GCP와 같은 클라우드 환경을 사용할 경우 데이타 베이스나, 또는 클라우드에서 제공되는 매지니드 서비스 (RDS, CloudSQL)등을 사용하고자할 경우에는 쿠버네티스 클러스터 밖이기 때문에, 호출이 어려운 경우가 있는데, 이를 쉽게 해결할 수 있는 방법이 ExternalName 타입이다.



ExternalName Example

아래와 같이 서비스를 ExternalName 타입으로 설정하고, 주소를 DNS로 goole.com으로 설정해주면 이 svc-external.default.svc.cluster.local는 들어오는 모든 요청을 www.google.com으로 포워딩 해준다.
일종의 프록시와 같은 역할이라 생각하면 쉽다.

POD: svc-external.default.svc.cluster.local ----- External Name으로 포워딩 --> www.google.com

```
apiVersion: v1
kind: Service
metadata:
  name: svc-external
spec:
  type: ExternalName
  externalName: google.com
```

실습:

1. ExternalName 서비스를 구동하자.

```
# cat svc-externalname.yaml
apiVersion: v1
kind: Service
metadata:
  name: svc-external
spec:
  type: ExternalName
  externalName: google.com

# kubectl create -f svc-externalname.yaml
# kubectl get svc
```

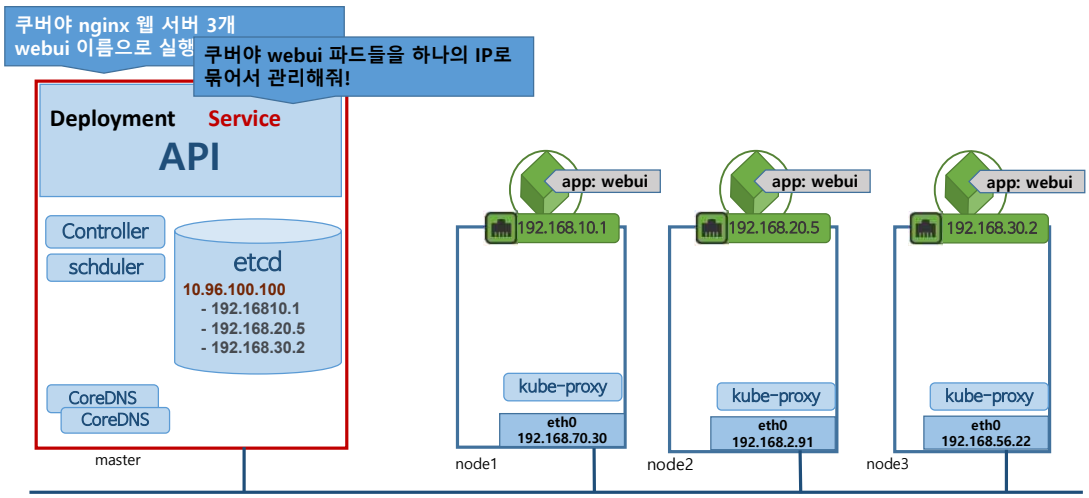
2. 간단히 TEST POD를 생성하여 ExternalName 접속 여부를 확인해본다.

```
# kubectl run testpod -it --image=centos:7
/# curl svc-external.default.svc.cluster.local
/# exit

# kubectl delete pod testpod

# kubectl delete svc svc-external
```

Kube-proxy

kube-proxy

- kube-proxy default mode는 iptables



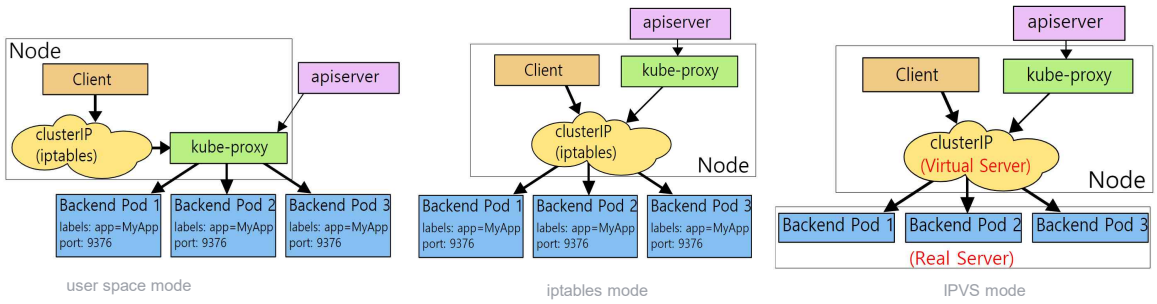
kube-proxy

Kubernetes 네트워크 프록시(kube-proxy)는 각 노드에서 실행되며 CNO(Cluster Network Operator)에 의해 관리된다.
kube-proxy는 서비스와 관련된 끝점에 대한 연결을 전달하기 위한 네트워크 규칙을 유지 관리한다.

iptables 규칙 동기화 정보
동기화 기간은 Kubernetes 네트워크 프록시(kube-proxy)가 노드에서 iptables 규칙을 동기화하는 빈도를 결정한다.

kube-proxy 동작 방식

- 3가지 mode: User space mode, iptables mode, IPVS mode



https://kubernetes.io/ko/docs/concepts/services-networking/service/

kube-proxy 동작방식

k8s에서 제공하는 kube-proxy 모드는 3가지가 존재하는데 베타 버전(현재 v1.9이 베타)을 제외한 나머지 모드는 아래와 같다.

- User space mode
- iptables mode
- IPVS mode

User space mode

쿠버네티스에서 각 Service는 Node에서 랜덤하게 Port를 선택하여 Open한다. 그리고 Opend된 Port(Proxy Port를 의미) 클라이언트의 요청을 해당 Pod로 redirect 하기 위해 사용된다.
Kubernetes API에 의해 Service가 생성되면 Node에 Proxy Port가 생성된다.

- 클라이언트 요청이 들어온다.
- iptables는 클라이언트가 요청한 ClusterIP와 Port를 확인하여 Proxy Port로 트래픽을 라우팅한다.
- Service의 SessionAffinity (내부 속성)에 따라 관련 Pod 중 하나를 선택하여 트래픽을 전달한다.

이와 같은 순서로 실행되며 특히 kube-proxy에 의해 전달받은 Pod는 Round-Robin에 의해 한 쪽에 몰리지 않고 균형있게 실행 할 수 있도록 해준다.

iptables mode

Default mode.
iptables Mode는 Userspace Mode와는 다르게 Iptables에 의해 Pod로 트래픽이 전달된다.
여기서 kube-proxy는 k8s의 apiserver에 의해 Service에 특정 이벤트 (추가, 삭제, 변경 등)이 생겼을 경우 관련된 IP와 Port를 Iptables에 업데이트한다. 때문에 Userspace Mode와 같이 Iptables -> Proxy와 같은 과정을 거치지 않는다. 그래서 장점으로 Iptables가 kernelspace에서 작업을 하기 때문에 성능면에서 Userspace 보다 좋다.

iptables Mode는 Pod가 응답하지 않을 경우 재시도를 하지 않는다.

- k8s API에 의해 Service가 생성되면 Node에 kube-proxy에 의해 Iptables가 갱신된다.
- 클라이언트 Pod의 Packet의 Target이 Service의 IP와 Port로 설정된다.
- Kernel에서 요청 Packet이 Iptables의 Rules 중에 맞는 것이 있는지 확인한다.
- iptables에서 맞는 것이 있다면 Backend Pod 중에 아무거나(Random) 선택한다.
- 지정된 Pod로 Packet를 전달하기전 Packet의 Target IP와 Port를 지정된 Pod의 것으로 수정한다.
- Packet을 전달된다.

IPVS mode

ipvs(IP Virtual Server) 모드는 리눅스 커널에 있는 L4 로드밸런싱 기술로 Netfilter에 포함되어 있다. 그래서 IPVS 커널모듈이 설치되어 있어야 한다. ipvs는 커널스페이스에서 작동하고 데이터 구조를 해시테이블로 저장해서 가지고 있기 때문에 iptables보다 빠르고 성능이 좋다.
ipvs는 rr(round-robin), lc(least connection), dh(destination hashing), sh(source hashing), sed(shortest expected delay), nq(never queue)등의 다양한 로드밸런싱 알고리즘을 제공한다.

실습 | kube-proxy 동작 원리 이해

- appjs 애플리케이션 배포
- NodePort 타입의 서비스 운영
- kube-proxy 확인
  - iptables 구성보기
  - pod수의 확장시 iptables 변화 확인

서비스 운영 실습

1. 먼저 appjs Pod를 실행한다. yamls 디렉토리에서 **rs-appjs.yaml**을 실행하여 appjs 포드를 3개 실행한다.
$ cat rs-appjs.yaml
$ kubectl apply -f rs-appjs.yaml
$ kubectl get pods

2. NodePort 타입의 서비스 동작
$ cat svc-node.yaml
$ kubectl apply -f svc-node.yaml

service 오브젝트의 세부 항목을 살펴보자.
$ kubectl describe svc appjs-service-nodeport

3. cluster IP와 NodePort를 확인해본다.
$ kubectl get pod -o wide
$ kubectl get service

```
NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
appjs-service-nodeport  NodePort    10.102.83.10    <none>        80:30123/TCP   64m
```

4. 노드에서 curl <cluster IP>:80 명령을 여러 번 반복 실행해본다. 어떤 컨테이너의 웹페이지가 표시되나?
$ curl 10.X.X.X

5. NodePort 접속 확인
$ curl 10.0.0.11:30123

6. Node에서 iptables 룰 확인
node# iptables -t nat -S | grep 80
node# iptables -t nat -S | grep KUBE-SVC-XXX

node# iptables -t nat -S | grep 30123

지금까지 배운 내용을 기준으로

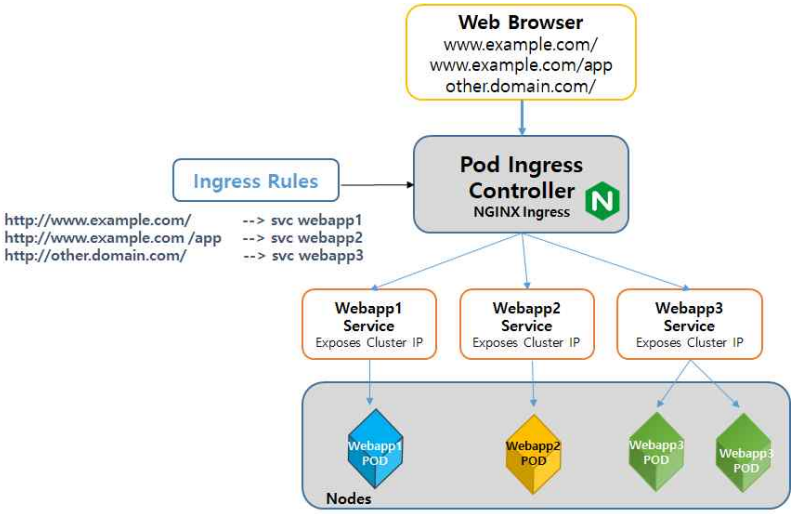
- Docker Network
  - Docker Network의 동작 원리를 이해할 수 있어야합니다.
  - docker0의 역할을 설명할 수 있어야합니다.
  - CNI에 대해 이해 해야합니다.

- Kubernetes Service
  - ClusterIP, NodePort, LoadBalancer 타입에 맞춰 서비스 구현이 가능해야합니다.
  - coreDNS를 이해하고 애플리케이션에 어떻게 적용할지 알아야합니다.
- Kube-Proxy
  - Kube-Proxy의 역할을 설명할수 있어야합니다.
  - Kube-Proxy의 iptables mode를 정확히 알고 있어야합니다.

Ingress의 이해

Service Ingress | Ingress의 이해

- HTTP나 HTTPS를 통해 클러스터 내부의 서비스를 외부로 노출시키는 API
- 기능
  - Service에 외부 URL을 제공
  - 트래픽을 로드밸런싱
  - SSL/TLS를 terminate
  - virtual hosting을 지정



Ingress란

쿠버네티스의 Ingress는 외부에서 쿠버네티스 클러스터 내부로 들어오는 네트워크 요청 즉, Ingress 트래픽을 어떻게 처리할지 정의한다. 쉽게 말하자면, Ingress는 외부에서 쿠버네티스에서 실행 중인 Deployment와 Service에 접근하기 위한, 일종의 관문(Gateway) 같은 역할을 담당한다.

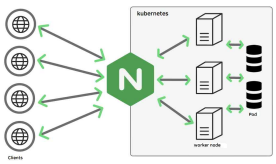
Ingress를 사용하지 않았다고 가정했을 때, 외부 요청을 처리할 수 있는 선택지는 NodePort, ExternalIP 등이 있다. 그러나 이러한 방법들은 일반적으로 Layer4(TCP,UDP)에서의 요청을 처리하며, 네트워크 요청에 대한 세부적인 처리 로직을 구현하기는 아무래도 한계가 있다.

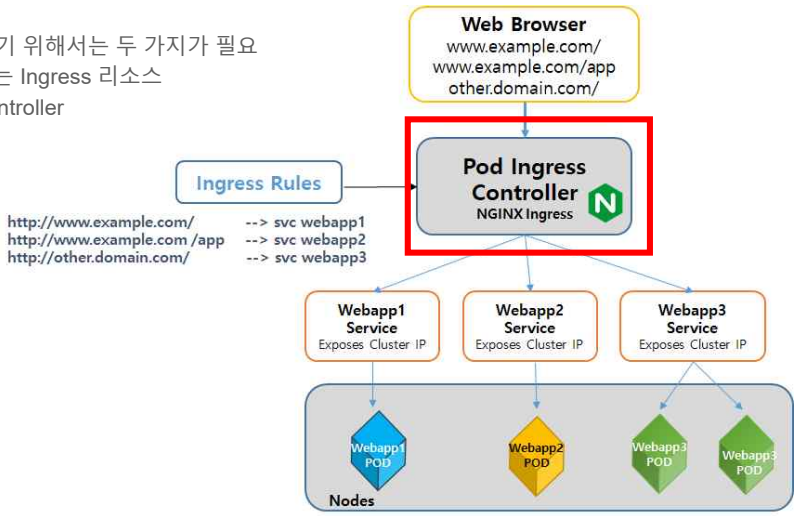
쿠버네티스의 Ingress는 Layer 7에서의 요청을 처리할 수 있다.
외부로부터 들어오는 요청에 대한 로드 밸런싱, TLS/SSL 인증서 처리, 특정 HTTP 경로의 라우팅 등을 Ingress를 통해 자세하게 정의할 수 있다. 물론, 이러한 기능들은 위에서 언급한 NodePort 등의 방법으로도 절대로 불가능한 것은 아니지만, 이러한 세부적인 로직을 모든 애플리케이션 개발 레벨에서 각각 구현하게 되면 서비스 운영 측면에서 추가적인 복잡성이 발생한다. 그 대신에, 외부 요청을 어떻게 처리할 것인지를 정의하는 집합인 Ingress를 정의한 뒤, 이를 Ingress Controller라고 부르는 특별한 웹 서버에 적용함으로써 추상화된 단계에서 서비스 처리 로직을 정의할 수 있다.

또한 Ingress Controller 종류 및 사용 중인 클라우드 공급자에 따라 다양한 기능을 부가적으로 사용할 수도 있으니, 서비스를 외부로 노출시켜 제공해야 한다면 Ingress를 사용하는 것이 바람직하다. 뒤에서 다시 설명하겠지만, Ingress 요청을 처리하기 위한 Service는 일반적으로 클라우드 플랫폼에서 제공되는 Load Balancer 타입의 Service를 사용한다. Private Cloud에서 운영하고 있는 서버에 Ingress를 직접 구축하게 된다면, Service의 Type으로서 NodePort또는 ExternalIP, MetalLB 등을 대신 사용할 수 있다.

Ingress Controller

- INGRESS Controller
  - 쿠버네티스에서 Ingress를 사용하기 위해서는 두 가지가 필요
  - YAML 파일에서 Ingress로 정의되는 Ingress 리소스
  - Ingress 규칙이 적용될 Ingress Controller
- 종류
  - ingress-gce
  - AWSLoad Balancer
  - ingress-nginx





Ingress와 Ingress Controller

쿠버네티스에서 Ingress를 사용하기 위해서는 두 가지가 필요하다. 첫 번째는 YAML 파일에서 Ingress로 정의되는 Ingress 오브젝트이고, 두 번째는 Ingress 규칙이 적용될 Ingress Controller 이다.

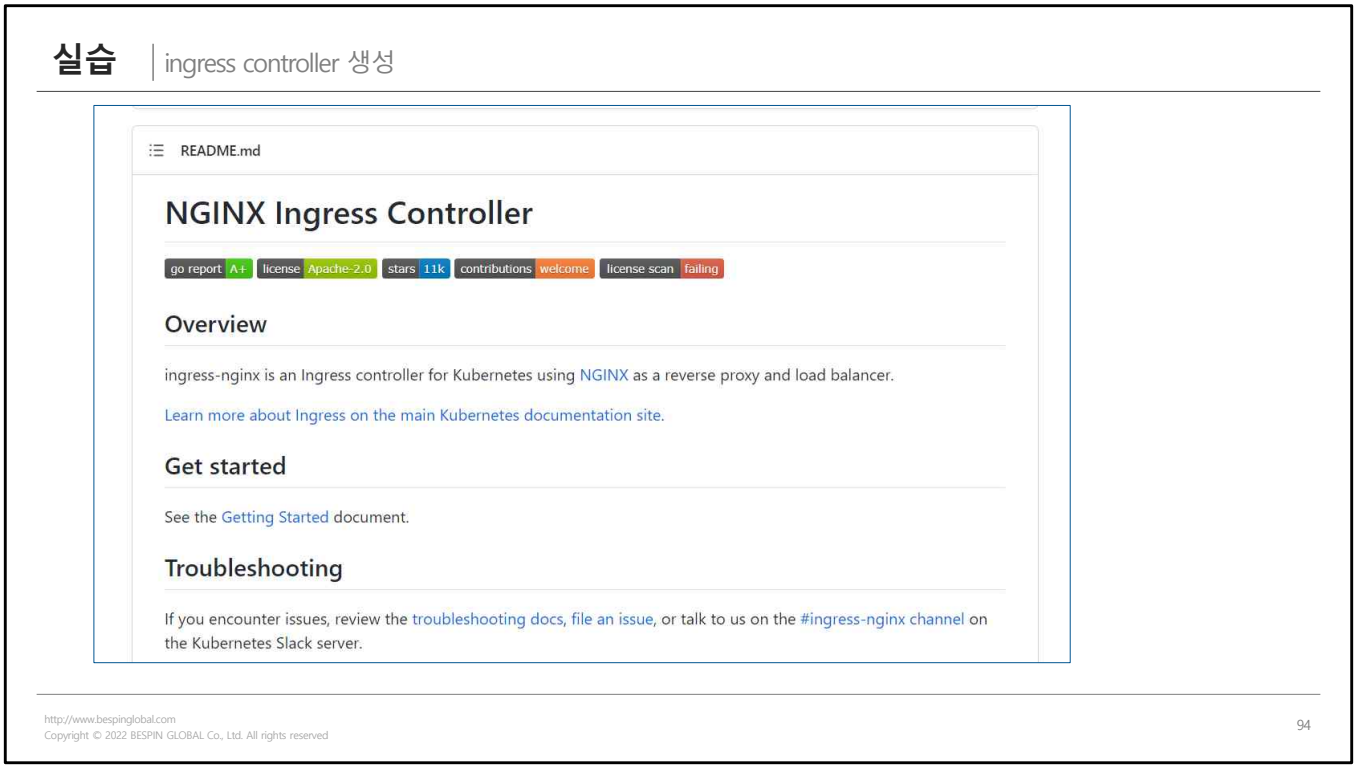
Ingress Controller를 직접 운영할지, 클라우드 플랫폼에 위임할지에 따라서 조금 고민해 볼 필요가 있다. 직접 Ingress Controller를 구동하려면 Nginx 웹 서버 기반의 Nginx Ingress Controller를 사용할 수 있겠고, 클라우드 플랫폼에 위임하려면 GKE (Google Kubernetes Engine) 의 기능을 사용할 수도 있다.

만약 AWS에서 EKS 또는 EC2 기반의 Kubespray를 사용하고 있다면, Nginx Ingress Controller를 직접 생성해 사용하되, 외부에서 Nginx에 접근하기 위한 쿠버네티스 Service를 Load Balancer 타입으로 설정해 로드 밸런서를 생성하는 방법을 생각해 볼 수 있다.

현재 Kubernetes에서 공식적으로 제공하는 Ingress Controller 는 다음의 플랫폼에 배포되어 있다.

- ingress-gce : Google Compute Engine 용. GCE를 이용하면 자동으로 사용됨
- ingress-nginx : 직접 설치해서 사용할 수 있는 ingress controller
- AWSLoad Balancer : AWS 클러스터에 프로비저닝 된 LoadBalancer controller

참고: https://kubernetes.github.io/ingress-nginx/



Ingress Controller 구성

Ingress-nginx는 NGINX를 reverse proxy와 load balancer로 사용하는 Kubernetes 용 Ingress 컨트롤러다.
ingress-nginx 인그레스 컨트롤러는 https://github.com/kubernetes/ingress-nginx 를 통해 설치 방법을 확인할 수 있다.

1. 설치 매뉴얼에 나와 있듯이 먼저 ingress controller를 제공하는 template 파일을 다운로드 받아서 실행한다.

```
# yum install wget -y

# wget https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.3/deploy/static/provider/baremetal/deploy.yaml

--2021-10-07 23:40:32--  https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.3/deploy/static/provider/baremetal/deploy.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.133, 185.199.111.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.109.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 18950 (19K) [text/plain]
Saving to: 'deploy.yaml'

100%[=====================================================================================================
   ===============>] 18,950      --.-K/s   in 0.001s

2021-10-07 23:40:33 (19.3 MB/s) - 'deploy.yaml' saved [18950/18950]
```

2. 다운로드 받은 deploy.yaml 파일을 보면 Ingress-nginx를 동작하는데 필요로 하는 인증, configmap, secret과 함께 deploy, nodePort 기반의 서비스를 동작시킨다.
모든 Ingress-nginx 리소스들은 Ingress-nginx라는 namespace에 생성된다.
아래 예제는 ingress-controller의 NodePort를 설정하였다.

```
# vi deploy.yaml
...
apiVersion: v1
kind: Service
metadata:
  annotations:
  labels:
    helm.sh/chart: ingress-nginx-4.0.4
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/version: 1.0.3
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/component: controller
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  type: NodePort
  ports:
    - name: http
      port: 80
```

```
    protocol: TCP
    targetPort: http
    appProtocol: http
    nodePort: 30100
  - name: https
    port: 443
    protocol: TCP
    targetPort: https
    appProtocol: https
    nodePort: 30200
..
```

3. 다운받은 파일을 실행함으로 controller를 동작시킨다.

```
# kubectl apply -f deploy.yaml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
serviceaccount/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
```

4. 생성된 서비스와 Pod 정보를 확인해보자.

```
# kubectl get pod -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create--1-vw9nd     0/1     Completed   0          2m14s
ingress-nginx-admission-patch--1-x7s47      0/1     Completed   1          2m14s
ingress-nginx-controller-6c68f5b657-rtmq7   1/1     Running     0          2m15s
```

5. 설치된 ingress controller의 버전을 확인해보자

```
# kubectl exec -n ingress-nginx  ingress-nginx-controller-6c68f5b657-rtmq7 -- /nginx-ingress-controller --version

-------------------------------------------------------------------------------
NGINX Ingress controller
  Release:       v1.0.3
  Build:         6e125826ad3968709392f2056023d4d7474ed4f5
  Repository:    https://github.com/kubernetes/ingress-nginx
  nginx version: nginx/1.19.9
-------------------------------------------------------------------------------
```

6. Ingress Controller가 실행하고 있는 서비스 포트정보를 확인해본다.

```
# kubectl get svc -n ingress-nginx
NAME                                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller             NodePort    10.103.207.208   <none>        80:30100/TCP,443:30200/TCP   2m55s
ingress-nginx-controller-admission   ClusterIP   10.107.174.150   <none>        443/TCP                      2m55s
```

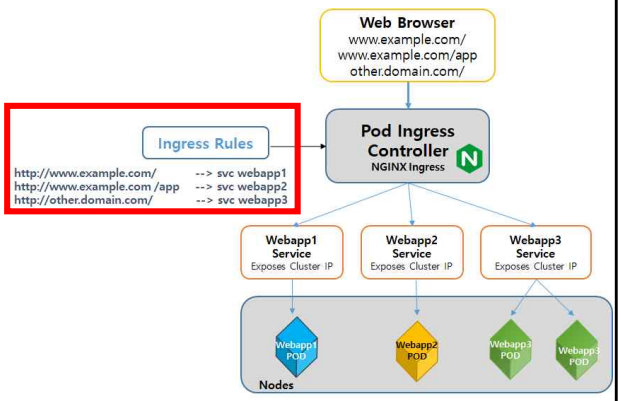
Ingress를 이용한 웹 기반 서비스 운영

Ingress를 이용한 웹 기반 서비스 운영

- Ingress 리소스

ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: marvel-heroes-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: webapp1-service
            port:
              number: 80
```



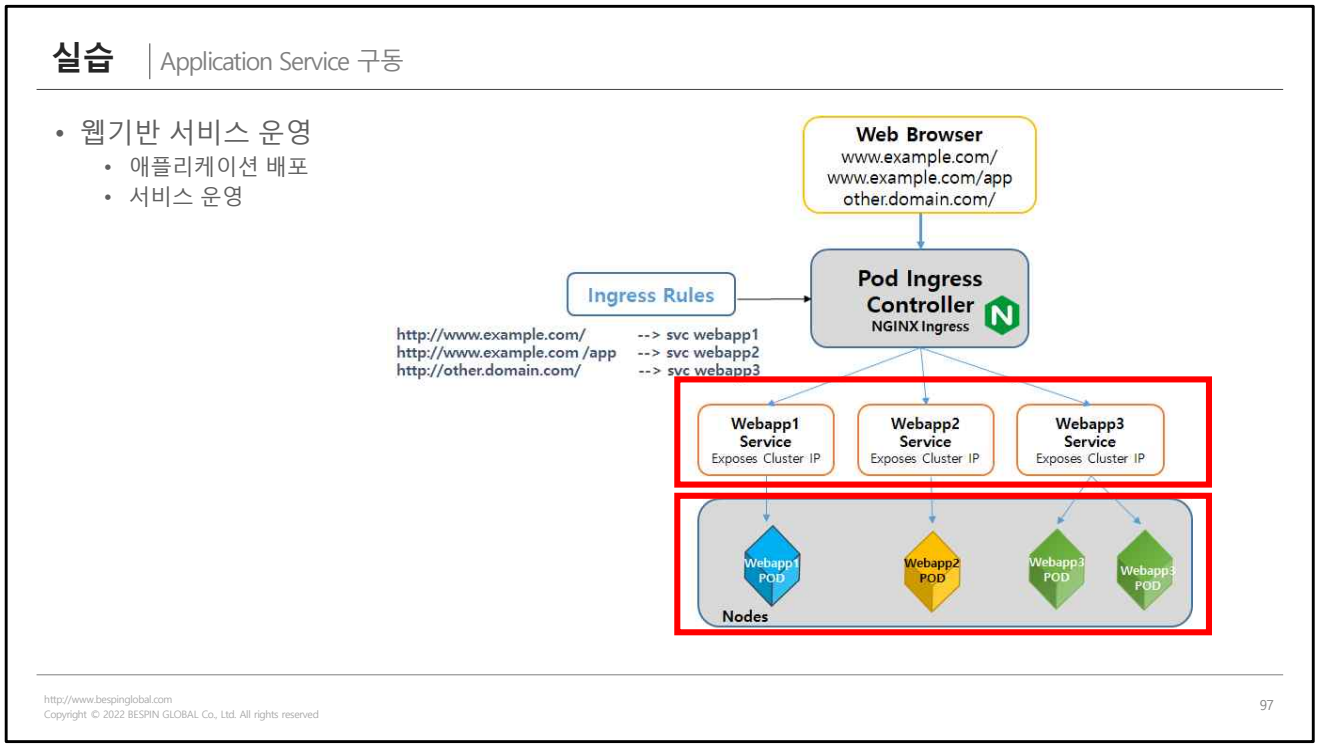
Ingress resource

Ingress 사양에는 LoadBalancer 또는 proxy 서버를 구성하는데 필요한 모든 정보가 있다. 가장 중요한 것은, 들어오는 요청과 일치하는 규칙 목록을 포함하는 것이다. ingress 리소스는 HTTP 트래픽을 지시하는 규칙만 지원한다.

Ingress Annotation

Ingress는 annotation을 통해 Ingress 구성 정보를 컨트롤 할 수 있다.

- nginx.ingress.kubernetes.io/rewrite-target : 지정된 경로로 Redirect 한다. "/"로 설정하면 하위경로와 무관하게 /로 redirect한다.
- nginx.ingress.kubernetes.io/force-ssl-redirect : http로 접속했을 때 강제로 https로 Redirect 한다.
- nginx.ingress.kubernetes.io/app-root : 기본적으로 설정되는 http 요청 경로의 root 경로를 재설정한다.



애플리케이션 배포

Switch Namespace

1. Ingress 운영을 쉽게 진행하기 위해 namespace를 치환한다. default namespace를 ingress-nginx로 변경하여 운영해보자.
   kubeconfig 파일을 사용하고 있는 병합된 구성 정보를 확인한다.

```
# kubectl config current-context
kubernetes-admin@kubernetes
```

2. kubeconfig 파일에 ingress-nginx 네임스페이스를 영구적으로 저장한다. ingress-nginx를 실행을 위한 context를 추가한다.

```
# kubectl config set-context ingress-nginx  --namespace=ingress-nginx  --cluster=kubernetes --user=kubernetes-admin
```

3. 앞서 생성한 ingress context가 잘 저장 되었는지 확인한다.

```
# kubectl config view
contexts:
- context:
    cluster: kubernetes
    namespace: ingress-nginx
    user: kubernetes-admin
  name: ingress-nginx
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
```

4. namespace를 default 대신 ingress-nginx로 변경한다.

```
# kubectl config use-context ingress-nginx
Switched to context "ingress-nginx".

# kubectl config current-context
ingress-nginx
```

5. namespace가 변경되었는지 확인한다. 구성된 ingress-nginx 네임스페이스에서 동작중인 pod, service를 확인해보자.

```
# kubectl get pods
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create--1-vw9nd     0/1     Completed   0          25m
ingress-nginx-admission-patch--1-x7s47      0/1     Completed   1          25m
ingress-nginx-controller-6c68f5b657-rtmq7   1/1     Running     0          25m


# kubectl get svc
```

```
NAME                                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller             NodePort    10.103.207.208   <none>        80:30100/TCP,443:30200/TCP   26m
ingress-nginx-controller-admission   ClusterIP   10.107.174.150   <none>        443/TCP                      26m
```

6. ingress 컨트롤러의 동작 상태를 확인한다. 위의 예제의 service 결과 중 node port로 접속 해보자. 아래와 같은 결과가 나온다면 해당 포트를 통해 서비스 대기하고 있는 것이다.

```
# curl -v node1.example.com:30100
* About to connect() to node1.example.com port 30100 (#0)
*   Trying 10.0.0.11...
* Connected to node1.example.com (10.0.0.11) port 30100 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: node1.example.com:30100
> Accept: */*
>
...
```

Appliation 배포 및 서비스 운영

배포하는 애플리케이션은 nginx base의 웹페이지 컨테이너인 marvel:latest와 thor:latest이다.
두 개의 웹페이지를 서비스하는 ingress-application.yaml의 내용은 아래와 같다.

1. 애플리케이션 운영을 위해 marvel-heroes 네임스페이스를 별도로 운영하며 marvel과 thor 서비스는 80 포트를 통해 제공된다.

```
$ cat ingress-application.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: ingress-nginx
  name: marvel
spec:
  replicas: 1
  selector:
    matchLabels:
      name: marvel
  template:
    metadata:
      labels:
        name: marvel
    spec:
      containers:
      - image: smlinux/marvel
        name: marvel-container
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  namespace: ingress-nginx
  name: marvel-service
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    name: marvel
---
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: ingress-nginx
  name: thor
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      name: thor
  template:
    metadata:
      labels:
        name: thor
    spec:
      containers:
      - image: smlinux/thor
        name: thor-continaer
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  namespace: ingress-nginx
  name: thor-service
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    name: thor
```

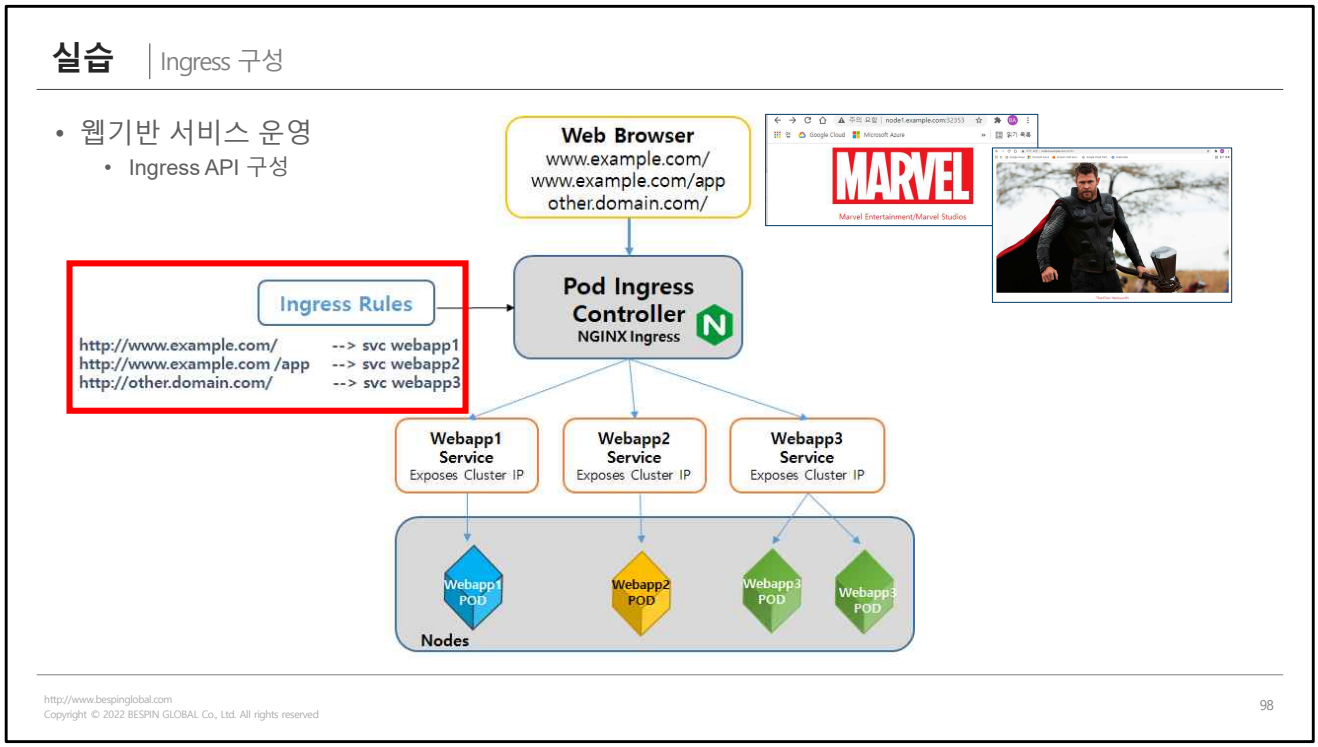
$ kubectl create -f ingress-application.yaml

```
deployment.apps/marvel created
service/marvel-service created
deployment.apps/thor created
service/thor-service created
```

2. 동작되는 서비스를 확인해본다.

 # kubectl get deployment,svc

```
NAME                                       READY   UP-TO-DATE   AVAILABLE
AGE
deployment.apps/ingress-nginx-controller   1/1     1            1           30m
deployment.apps/marvel                     1/1     1            1           37s
deployment.apps/thor                       1/1     1            1           37s

NAME                      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/marvel-service    ClusterIP   10.96.72.213    <none>        80/TCP
37s
service/thor-service      ClusterIP   10.108.229.19   <none>        80/TCP
37s
```



Ingress 구성

1. WEB URL 별로 서로 다른 서비스를 구현하는 ingress를 운영해보자.
ingress2.yaml 파일을 이용해 node2.example.com/ 로 접속했을 때 marvel 서비스가 표시되고, node2.example.com/thor 로 접속했을 때 thor 서비스가 동작하도록 ingress를 구성하자.

```
$ cat ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: ingress-nginx
  name: marvel-heroes-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  defaultBackend:
    service:
      name: nginx
      port:
        number: 80
rules:
  - host: node1.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: marvel-service
            port:
              number: 80
  - host: node2.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: thor-service
            port:
              number: 80
```

$ **kubectl apply -f ingress.yaml -f ingress-backend-nginx.yaml**

2. ingress 구성 상태를 자세히 확인해보자.

```
$ kubectl get ingress
NAME                 CLASS    HOSTS                                ADDRESS   PORTS   AGE
marvel-heroes-ingress   <none>   node1.example.com,node2.example.com             80      104s

$ kubectl describe ingress marvel-heroes-ingress
Name:             marvel-heroes-ingress
Namespace:        ingress-nginx
Address:          10.0.0.12
Default backend:  nginx:80 (10.44.0.1:80)
Rules:
  Host                  Path  Backends
  ----                  ----  --------
  node1.example.com   /       marvel-service:80 (10.40.0.2:80)
  node2.example.com   /       thor-service:80 (10.44.0.2:80)
Annotations:          kubernetes.io/ingress.class: nginx
Events:
  Type    Reason  Age                  From                      Message
  ----    ------  ----                 ----                      -------
  Normal  Sync    8m54s (x2 over 9m)   nginx-ingress-controller  Scheduled for sync
```

3. 웹 브라우저를 실행한 후 node1.example.com 과 node2.example.com 에 접속했을 때 서로 다른 서비스가 동작하는지 확인해보자.

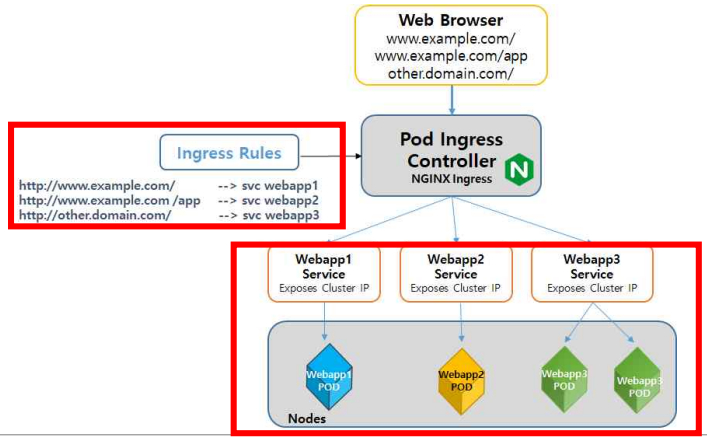
```
$ kubectl get svc
NAME                                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller             NodePort       10.103.207.208   <none>
80:30100/TCP,443:30200/TCP   48m
ingress-nginx-controller-admission   ClusterIP      10.107.174.150   <none>        443/TCP
48m
```

4. 웹 브라우저로 접속하여 서로 다른 웹페이지가 표시되는지 확인해본다.

```
$ curl http://node1.example.com:30100/
<html>
<head>
  <title>marvel heroes</title>
</head>
<body>
  <center>
  <img src="images/marvel_logo.png"><br>
  <p style="color:red;">Marvel Entertainment/Marvel Studios</p>
  </center>
</body>
</html>


curl http://node2.example.com:30100/
```

심화 | Ingress 구성

- 인증서 기반의 Ingress 서비스 운영
- 여러 개의 Pod가 동작되는 ingress를 운영할 때 클라이언트의 Session을 유지해줄 수 있다.
- 앞서 사용했던, appjs pod를 여러 개 실행한 후 ingress를 구성해보자.
  1) **Application service 구성**
  2) Session 유지 기능 추가
  3) 인증서 기능 추가 설정
  4) Ingress 삭제



세션 유지 및 인증서구성

1. Application service 구성

 # cat appjs.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  namespace: ingress-nginx
  name: appjs-rc
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: appjs
    spec:
      containers:
      - image: smlinux/appjs
        name: testcon
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  namespace: ingress-nginx
  name: appjs-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: appjs
```

 # kubectl apply -f appjs.yaml

 # kubectl get rc

```
NAME          DESIRED   CURRENT   READY   AGE
appjs-service   3         3         3       31s
```

```
# kubectl get svc
NAME                                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
appjs-service                        ClusterIP   10.108.4.168     <none>        80/TCP
43s
ingress-nginx-controller             NodePort    10.105.164.238   <none>
80:30100/TCP,443:30200/TCP   10m
ingress-nginx-controller-admission   ClusterIP   10.99.30.176     <none>        443/TCP
10m
```

2. ingress2.yaml 파일을 생성하여 ingress 서비스가 잘 되는지 확인해본다.

```
# kubectl delete ingress marvel-heroes-ingress
# vi  ingress2.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: marvel-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: marvel-service
            port:
              number: 80
  - http:
      paths:
      - path: /app
        pathType: Prefix
        backend:
          service:
            name: appjs-service
            port:
              number: 80

# kubectl apply -f  ingress2.yaml

# kubectl get ingress marvel-ingress

# kubectl describe ingress marvel-ingress
Name:             marvel-ingress
Namespace:        ingress-nginx
Address:
Default backend:  nginx:80 (10.44.0.1:80)
Rules:
  Host        Path  Backends
  ----        ----  --------
  *           /     marvel-service:80 (10.40.0.2:80)
  *           app   appjs-service:80 (10.40.0.3:8080,10.40.0.4:8080,10.44.0.3:8080)
Annotations:  kubernetes.io/ingress.class: nginx
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  Sync    16s   nginx-ingress-controller  Scheduled for sync
```

3. 이제 브라우저에서 ingress를 통해 서비스에 연결해 보라.
http://node1.example.com:30100/
http://nodeX.example.com:30100/app

Service Ingress | LAB

- 인증서 기반의 Ingress 서비스 운영
- 여러 개의 Pod가 동작되는 ingress를 운영할 때 클라이언트의 Session을 유지해줄 수 있다.
- 앞서 사용했던, appjs pod를 여러 개 실행한 후 ingress를 구성해보자.
  1) Ingress service 구성
  2) **Session 유지 기능 추가**
  3) 인증서 기능 추가 설정
  4) Ingress 삭제

세션 유지 및 인증서구성

1. 앞에서 실행한 ingress2.yaml을 수정해서 session이 유지되도록 운영하시오.

$ kubectl delete ingress marvel-ingress

$ vi ingress2.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: marvel-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "btc-cookie"
    nginx.ingress.kubernetes.io/session-cookie-hash: "sha1"
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: marvel-service
            port:
              number: 80
  - http:
      paths:
      - path: /app
        pathType: Prefix
        backend:
          service:
            name: appjs-service
```

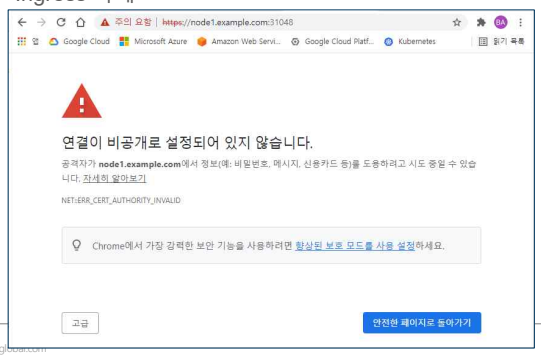
 # kubectl apply -f ingress2.yaml

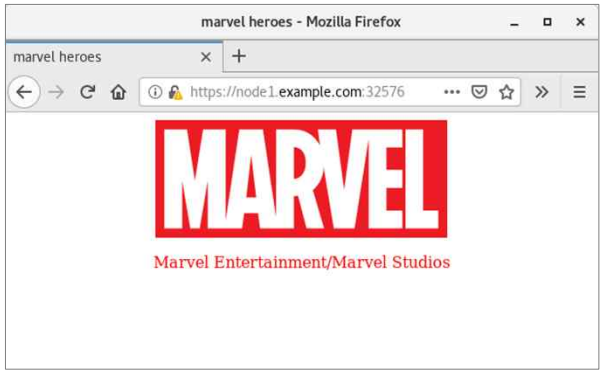
5. 웹 브라우저를 이용해 http://nodeX.example.com:30100/app 로 접속하여 세션이 유지되는지 확인해보자.

http://nodeX.example.com:30100/app

Service Ingress | LAB

- 인증서 기반의 Ingress 서비스 운영
- 여러 개의 Pod가 동작되는 ingress를 운영할 때 클라이언트의 Session을 유지해줄 수 있다.
- 앞서 사용했던, appjs pod를 여러 개 실행한 후 ingress를 구성해보자.
  1) Ingress service 구성
  2) **Session 유지 기능 추가**
  3) **인증서 기능 추가 설정**
  4) Ingress 삭제





세션 유지 및 인증서구성

인증서 기능 추가 설정

1. Ingress 컨트롤러가 TLS를 사용하려면, Ingress에 certificate 와 a private key를 연계해야 한다.
   $ openssl genrsa -out tls.key 2048
   $ openssl req -new -x509 -key tls.key -out tls.crt -days 365 -subj /CN=node2.example.com

2. 생성한 인증서를 secret에 기록한다.
   $ kubectl create secret tls tls-secret --cert=tls.crt --key=tls.key

3. Ingress 리소스를 수정해서 HTTPS 요청을 처리할 수 있도록 앞에서 만든 인증서를 등록한다..
   $ kubectl delete ingress marvel-ingress

```
$ vim ingress2.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  namespace: ingress-nginx
  name: marvel-ingress
 annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "btc-cookie"
    nginx.ingress.kubernetes.io/session-cookie-hash: "sha1"
spec:
  tls:
  - hosts:
    secretName: tls-secret

 defaultBackend:
    service:
      name: nginx
      port:
        number: 80

 rules:
 - http:
```

```
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: marvel-service
            port:
              number: 80
  - http:
      paths:
      - path: /app
        pathType: Prefix
        backend:
          service:
            name: appjs-service
            port:
              number: 80
```

$ kubectl apply -f ingress2.yaml
$ kubectl get ingress
$ kubectl get svc ingress-nginx-controller

4. HTTPS를 사용해서 Ingress를 통해 포드의 웹 서비스에 액세스할 수 있는지 확인한다.

https://node2.example.com:30200/ https://node2.example.com:30200/app

연결에 성공하면 아래와 같은 인증서를 확인하는 페이지가 표시된다. [Advanced] 버튼을 누르고 [Accept the Risk and continue]를 클릭해서 인증서를 저장해야 한다.

Service Ingress | LAB

- 인증서 기반의 Ingress 서비스 운영
- 여러 개의 Pod가 동작되는 ingress를 운영할 때 클라이언트의 Session을 유지해줄 수 있다.
- 앞서 사용했던, appjs pod를 여러 개 실행한 후 ingress를 구성해보자.
  1) Ingress service 구성
  2) Session 유지 기능 추가
  3) 인증서 기능 추가 설정
  4) **Ingress 삭제**

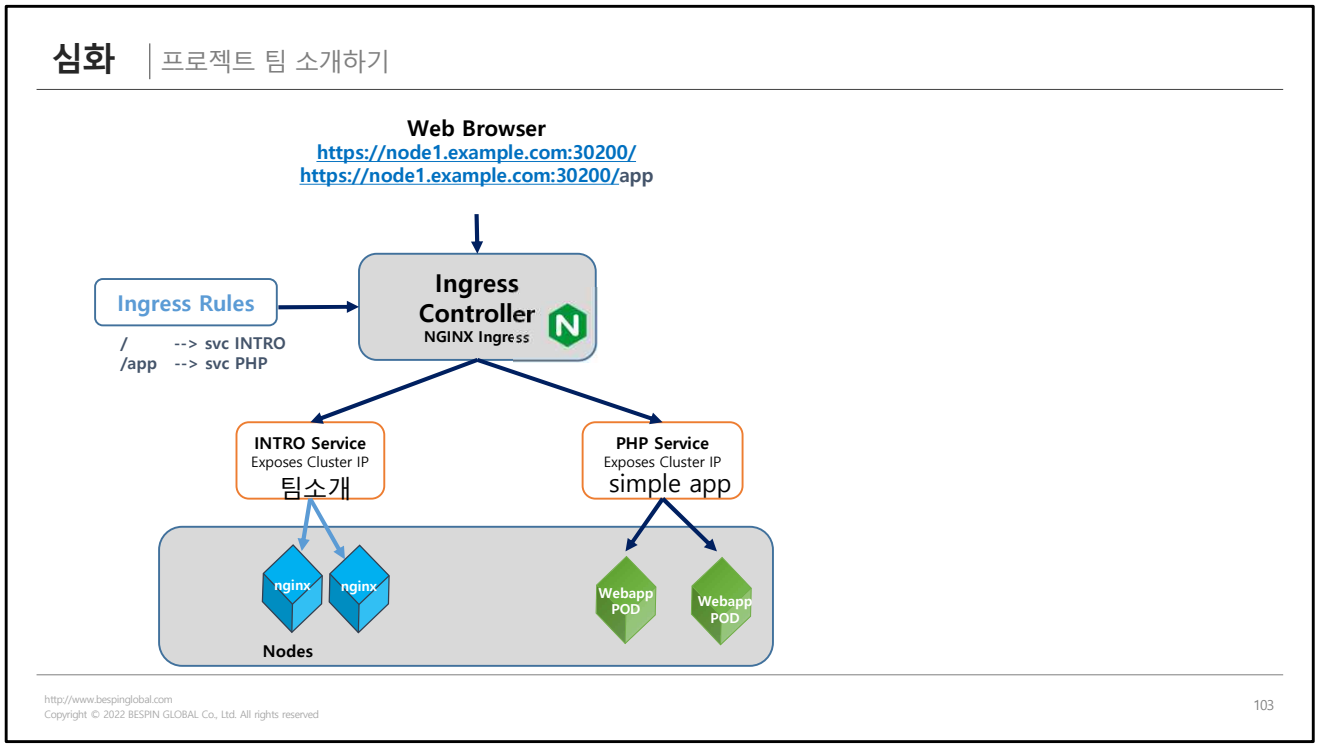
세션 유지 및 인증서구성

Ingress 삭제

1. 실습이 완료되면 ingress-nginx 네임스페이스를 삭제하여 모든 API resource를 제거하자.
   $ kubectl delete ns ingress-nginx

2. 원래의 default namespace를 config 정보를 수정하여 작업하자.
   $ kubectl config use-context kubernetes-admin@kubernetes

   $ kubectl config current-context



Ingress 를 이용해 팀소개 페이지 운영하기

Ingress 를 이용해서 두개의 서비스에 접근할 수 있도록 application 서비스를 구현하시오.

지금까지 배운 내용을 기준으로

- 인그레스
  - 인그레스를 이해하고 설명할 수 있습니다.
  - 인그레스 컨트롤러를 구축할 수 있습니다.
  - 인그래스 기반의 Application을 운영할 수 있습니다.