리소스 관리 와 오토 스케일링

리소스 제한

- 개요
  - ✓ 쿠버네티스에서는 컨테이너 단위로 리소스 제한 설정이 가능
  - ✓ 서비스에 맞는 성능을 내기 위해서도 리소스 제한 설정은 필요
  - ✓ 제한이 가능한 리소스는 CPU, 메모리, Ephemeral 스토리지이지만 Device Plugins를 사용하면 GPU 등의 다른 리소스에 대해서도 제한 설정을 할 수 있음

리소스 제한

- CPU/메모리 리소스 제한
  - ✓ CPU는 클럭 수로 지정하지 않고 1vCPU(가상 CPU)를 1,000m(millicores) 단위로 지정하는데 3GHz CPU라고 해도 1코어 정도를 지정할 경우 3,000m가 아닌 1,000m
  - ✓ 메모리는 1G = 1000M
  - ✓ 리소스 제한은 파드 정의 내부의 각 컨테이너 정의 부분에 포함되고 spec.containers[ ].resources의 requests.cpu/requests.memory 또는 limits.cpu/limits.memory를 지정하는 형태로 제한

리소스 제한

- CPU/메모리 리소스 제한
  - ✓ CPU/메모리 리소스 지정
    - 매니페스트 작성: sample-resource.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-resource
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
```

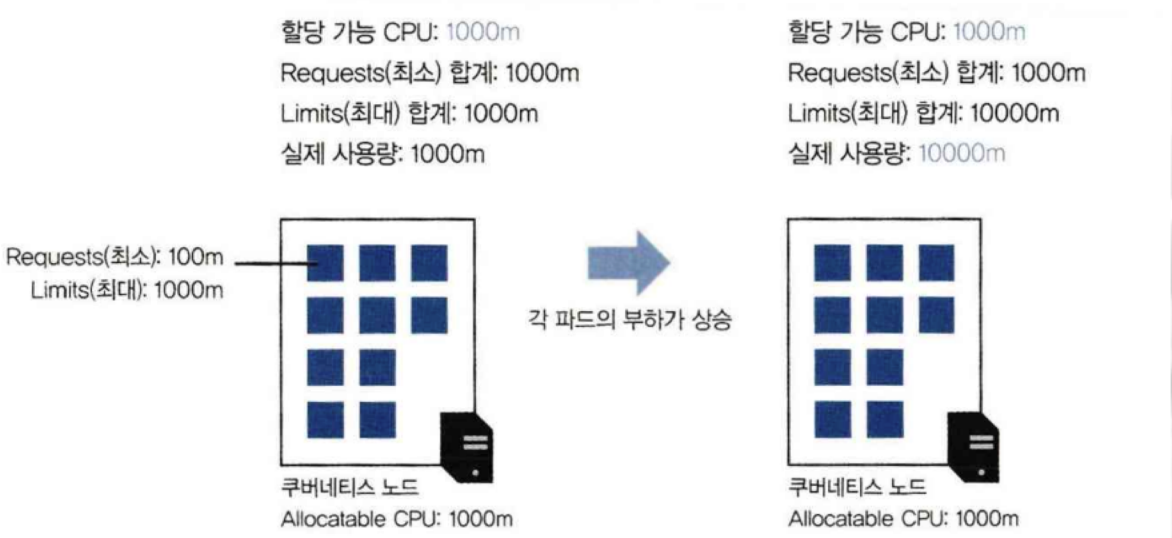
리소스 제한

- CPU/메모리 리소스 제한
  - ✓ CPU/메모리 리소스 지정
    - 매니페스트 작성: sample-resource.yaml

```
template:
  metadata:
    labels:
      app: sample-app
  spec:
    containers:
    - name: nginx-container
      image: nginx:1.17
      resources:
        requests:
          memory: "1024Mi"
          cpu: "500m"
        limits:
          memory: "2048Mi"
          cpu: "1000m"
```

리소스 제한

- ❖ CPU/메모리 리소스 제한
  - ✓ CPU/메모리 리소스 지정
    - ● 매니페스트 작성: sample-resource.yaml
      - ◆ requests는 사용하는 리소스 최솟값을 지정하는데 빈 노드에 requests로 지정한 양의 리소스가 존재하지 않으면 스케줄링되지 않음
      - ◆ limits는 사용할 리소스의 최댓값을 지정하는데 requests와 달리 노드에 limits로 지정한 리소스가 남아 있지 않아도 스케줄링되므로 requests 리소스 양 < limits 리소스 양이지만 requests와 limits의 차이가 아주 큰 컨테이너에서 부하가 상승하면 실제로 사용하려는 리소스 양과 차이가 있으므로 주의



리소스 제한

- CPU/메모리 리소스 제한
  - ✓ CPU/메모리 리소스 지정
    - 리소스 생성: kubectl apply -f sample-resource.yaml
    - 노드에 할당된 리소스 현황 확인: kubectl describe node worker1

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           requests      limits
  --------           --------      ------
  cpu                600m (60%)    1 (100%)
  memory             1074Mi (57%)  2Gi (110%)
  ephemeral-storage  0 (0%)        0 (0%)
  hugepages-1Gi      0 (0%)        0 (0%)
  hugepages-2Mi      0 (0%)        0 (0%)
  hugepages-32Mi     0 (0%)        0 (0%)
  hugepages-64Ki     0 (0%)        0 (0%)
```

리소스 제한

- CPU/메모리 리소스 제한
  - ✓ requests 만 설정한 경우
    - limits는 자동으로 설정되지 않고 호스트 측의 부하가 최대로 상승할 때 까지 리소스를 계속 소비하려고 하기 때문에 이런 파드가 많이 기동하는 노드에서 리소스 뺏기가 발생하고 메모리의 경우 OOM(Out of Memory)으로 인해 프로세스 정지로 이어지게 됨
    - 매니페스트 작성: sample-resource-only-requests.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-resource-only-requests
spec:
  containers:
  - name: nginx-container
    image: nginx:1.16
    resources:
      requests:
        memory: 256Mi
        cpu: 200m
```

리소스 제한

- CPU/메모리 리소스 제한
  - ✓ requests 만 설정한 경우
    - 리소스 생성: kubectl apply -f sample-resource-only-requests.yaml
    - 리소스 설정 확인: kubectl get pod sample-resource-only-requests -o json | jq ".spec.containers[].resources"

```
{
  "requests": {
    "cpu": "200m",
    "memory": "256Mi"
  }
}
```

리소스 제한

- CPU/메모리 리소스 제한
  - ✓ limits 만 설정한 경우
    - 반대로 limits만 설정한 경우에는 limits와 같은 값이 requests에 설정되게 되어 있음
    - 오버커밋되는 경우 limits뿐만 아니라 명시적으로 requests도 설정해야 함
    - 매니페스트: sample-resource-only-limits.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-resource-only-limits
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    resources:
      limits:
        memory: 256Mi
        cpu: 200m
```

리소스 제한

- CPU/메모리 리소스 제한
  - ✓ limits 만 설정한 경우
    - 리소스 생성: kubectl apply -f sample-resource-only-limits.yaml
    - 리소스 설정 확인: kubectl get pod sample-resource-only-limits -o json | jq ".spec.containers[].resources"

```
{
  "limits": {
    "cpu": "200m",
    "memory": "256Mi"
  },
  "requests": {
    "cpu": "200m",
    "memory": "256Mi"
  }
}
```

리소스 제한

- Ephemeral 스토리지 리소스 제어
  - ✓ 일반적으로 스토리지는 용량 문제나 영속성 관점에서 영구 볼륨을 사용하여 그곳에 데이터를 쓰는 것이 바람직하지만 컨테이너 재기동 시 삭제되어도 좋은 데이터의 경우에는 컨테이너 내부의 디스크 영역(간접적으로 노드 디스크 영역)을 사용할 수도 있음
  - ✓ 애플리케이션이 로그 데이터를 대량으로 출력하는 경우에도 예상치 못하게 쿠버네티스 노드의 디스크 영역을 많이 차지하는 경우가 있는데 이때 사용하는 것이 Ephemeral 스토리지 리소스 제한
  - ✓ 쿠버네티스 노드에서 기동 중인 시스템 구성 요소의 kubelet이 디스크 사용 현황을 정기적으로 모니터링하고 초과한 경우 파드는 축출(Evict)되게 되어 있음
  - ✓ Ephemeral 스토리지 용량으로 계산되는 것
    - 컨테이너가 출력하는 로그
    - emptyDir에 기록된 데이터(medium: 메모리가 아닌 것)
    - 컨테이너의 쓰기 가능한 레이어에 기록된 데이터
  - ✓ 컨테이너에서 쓰기로 소비되는 양은 emptyDir에 기록된 데이터와 컨테이너의 쓰기 가능한 레이어에 기록된 데이터의 합계로 쓰기 가능한 레이어의 데이터양은 간단히 말하면 영구 볼륨/hostPath/nfs 등이 마운트된 영역을 제외한 모든 영역에 대한 쓰기와 거의 동등
  - ✓ 컨테이너 내부에서 쓰기 외에 컨테이너가 출력하는 로그도 kubectl logs 명령어로 확인할 수 있게 내부에 저장되어 있는데 이 로그 데이터도 Ephemeral 스토리지에 의해 계산되기 때문에 주의
  - ✓ Ephemeral 스토리지도 CPU나 메모리와 마찬가지로 requests와 limits에 의해 제한을 설정할 수 있음

리소스 제한

- ❖ Ephemeral 스토리지 리소스 제어
  - ✓ Ephemeral 스토리지 제한 설정
    - ● 매니페스트: sample-ephemeral-storage.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-ephemeral-storage
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    resources:
      requests:
        ephemeral-storage: "1024Mi"
      limits:
        ephemeral-storage: "2048Mi"
```

리소스 제한

- ❖ Ephemeral 스토리지 리소스 제어
  - ✓ Ephemeral 스토리지 제한 설정
    - 최대 2,048Mi를 초과하는 데이터를 쓰면 파드는 자동으로 Evict
    - 기본적으로 설정에 따라 시스템용으로 여분 리소스가 확보되거나 빠듯하게 Evict되지 않게 여분의 리소스가 확보되기도 하지만 ephemeral-storage 사용 제한을 하지 않으면 예상치 못한 데이터 쓰기로 인해 쿠버네티스 노드의 디스크 용량을 초과하여 그 노드의 모든 파드에 영향을 줄 가능성도 있으므로 주의
    - 리소스 생성: kubectl apply -f sample-ephemeral-storage.yaml
    - 2048MB 데이터 쓰기: kubectl exec -it sample-ephemeral-storage -- dd if=/dev/zero of=/dummy bs=1M count=2048

      2048+0 records in

      2048+0 records out

      2147483648 bytes (2.1 GB, 2.0 GiB) copied, 1.68393 s, 1.3 GB/s
    - kubectl get pods

      NAME READY STATUS RESTARTS AGE

      sample-ephemeral-storage 0/1 Completed 0 78s

리소스 제한

- Ephemeral 스토리지 리소스 제어
  - ✓ 여러 컨테이너에서 emptyDir을 공유하는 파드
    - emptyDir은 파드에서 공유하고 사용할 수 있기 때문에 여러 파드에 설정된 limits의 합계를 초과한 경우 Evict
    - 파드에는 각 컨테이너의 /cache 디렉터리에 같은 emptyDir이 마운트되어 있는데 각각의 컨테이너에 2GiB의 Ephemeral 스토리지 제한(limits)이 설정되어 있음
    - 매니페스트: sample-ephemeral-storage-multi.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-ephemeral-storage-multi
```

리소스 제한

- ❖ Ephemeral 스토리지 리소스 제어
  - ✓ 여러 컨테이너에서 emptyDir을 공유하는 파드
    - ● 매니페스트: sample-ephemeral-storage-multi.yaml

```
spec:
  containers:
  - name: container-a
    image: amsy810/tools:v2.0
    resources:
      requests:
        ephemeral-storage: "1024Mi"
      limits:
        ephemeral-storage: "2048Mi"
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
```

리소스 제한

- Ephemeral 스토리지 리소스 제어
  - ✓ 여러 컨테이너에서 emptyDir을 공유하는 파드
    - 매니페스트: sample-ephemeral-storage-multi.yaml

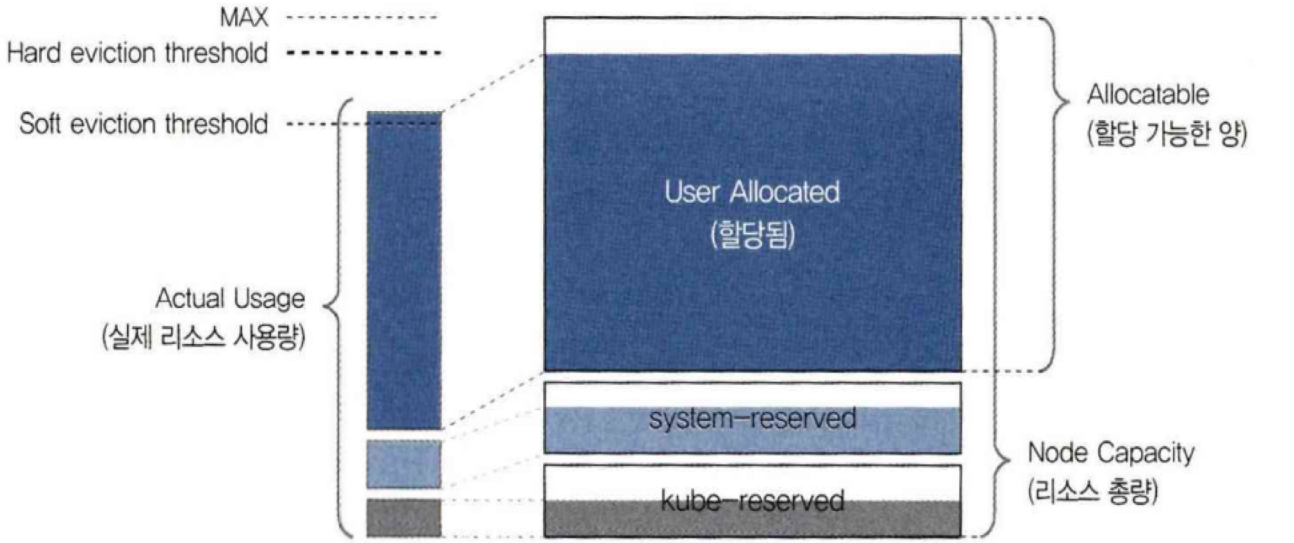
```
- name: container-b
  image: amsy810/tools:v2.0
  resources:
    requests:
      ephemeral-storage: "1024Mi"
    limits:
      ephemeral-storage: "2048Mi"
  volumeMounts:
  - mountPath: /cache
    name: cache-volume
volumes:
- name: cache-volume
  emptyDir: {}
```

리소스 제한

- ❖ Ephemeral 스토리지 리소스 제어
  - ✓ 여러 컨테이너에서 emptyDir을 공유하는 파드
    - 리소스 생성: kubectl apply -f sample-ephemeral-storage-multi.yaml
    - 컨테이너에 3GB 데이터 쓰기: kubectl exec -it sample-ephemeral-storage-multi -c container-a -- dd if=/dev/zero of=/dummy bs=1M count=3000
    - 파드 확인: kubectl get pods
    - 파드를 삭제하고 다시 생성한 후 emptyDir 영역에 같은 처리를 실행
    - 이번에는 첫번째 컨테이너의 emptyDir 영역에 3GB 데이터를 써도 두 개 파드의 Ephemeral 스토리지 제한(limits)인 4GiB에 도달하지 않아 파드는 Evict되지 않음
    - 4GiB(4,096MiB)이상의 데이터를 쓸 때 Evict

리소스 제한

- 시스템에 할당된 리소스 와 Eviction 매니저
  - ✓ CPU/메모리/Ephemeral 스토리지의 일반적인 리소스는 완전히 고갈되면 쿠버네티스 자체가 동작하지 않거나 그 노드 전체에 영향을 미칠 수 있기 때문에 각 노드에는 kube-reserved, system-reserved라는 두 가지 리소스가 시스템용으로 확보되어 있음
  - ✓ kube-reserved는 쿠버네티스 시스템 구성 요소나 컨테이너 런타임에 확보된 리소스
  - ✓ system-reserved는 OS에 깊이 관련된 데몬 등에 확보된 리소스
  - ✓ 실제 파드에 할당 가능한 리소스(Allocatable)는 쿠버네티스 노드에 존재하는 리소스 총량에서 kube-reserved, system-reserved를 제외한 양



리소스 제한

- 시스템에 할당된 리소스 와 Eviction 매니저
  - ✓ 노드의 리소스 양과 할당 가능한 양은 노드 리소스에서 확인 가능: kubectl get nodes -o custom-columns="NAME:.metadata.name, CPU Capacity:.status.capacity.cpu, CPU Allocatable:.status.allocatable.cpu, Mem Capacity:.status.capacity.memory, Mem Allocatable:.status.allocatable.memory"

NAME	CPU Capacity	CPU Allocatable	Mem Capacity	Mem Allocatable
master	2	2	2001348Ki	1898948Ki
worker1	1	1	2001564Ki	1899164Ki
worker2	1	1	2001564Ki	1899164Ki

  - ✓ 쿠버네티스 내부에서는 Eviction 매니저라는 구성 요소가 동작하며 시스템 전체가 과부하되지 않도록 관리
  - ✓ Eviction 매니저는 Allocatable, system-reserved, kube-reserved에서 실제로 사용되는 리소스 합계가 Eviction Threshold 초과하지 않는지 정기적으로 확인하고 초과한 경우 파드를 Evict하고 이를 통해 노드의 리소스를 100% 다 쓰지 않도록 되어 있음
  - ✓ 특히 메모리의 경우 100% 빠듯하게 시스템을 가동시키면 OOM이 많이 발생하므로 Eviction Threshold에 의해 OOM 발생을 일정하게 제어하고 있음
  - ✓ Eviction Threshold는 soft와 hard 두 가지가 있는데 soft 제한에 걸리면 SIGTERM 신호를 보내 파드를 정지하려고 하는데 이때 terminationGracePeriodSeconds는 파드에 설정되어 있는 값 또는 kubelet에 지정되어 있는 --eviction-soft-grace-period 옵션값 중 짧은 쪽이 선택되는데 그래서 파드에 긴 terminationGracePeriodSeconds 값을 설정했다 하더라도 클러스터 제공자가 최대로 지정한 시간이 더 짧으면 더 빨리 정지되는 경우가 있으므로 주의

리소스 제한

- 시스템에 할당된 리소스 와 Eviction 매니저
  - ✓ hard 제한이 걸리면 바로 SIGKILL이 보내져 파드가 정지됨
  - ✓ kube-reserved, system-reserved나 Eviction Threshold 설정값은 사용하는 쿠버네티스 환경이나 노드 인스턴스 유형에 따라 다름
  - ✓ Eviction 매니저가 Evict하는 파드는 다음 우선순위에 따라 결정
    - requests에 할당된 양보다 초과하여 리소스를 소비하고 있는 것
    - PodPriority가 더 낮은 것
    - requests에 할당된 양보다 초과하여 소비하고 있는 리소스 양이 더 많은 것

리소스 제한

- GPU 등의 리소스 제한
  - ✓ Device Plugins 기능이 지원되어 GPU(NVIDIA/AMD)나 그 외 리소스도 CPU나 메모리와 마찬가지로 requests/limits 제한을 설정할 수 있음
  - ✓ FPGA/VPU(Vision Processing Unit - 머신 비전 액셀러레이터)/QAT(Quick AssistTechnology - 암호 관련 워크로드 액셀러레이터)/TPU(Tensor Processing Unit - 기계 학습용 프로세서) 등의 장치도 지원
  - ✓ 엔비디아 GPU 제한

```
resources:
  requests:
    nvidia.com/gpu: 2
  limits:
    nvidia.com/gpu: 2
```

    - 엔비디아의 GPU는 requests에 nvidia.com/gpu 지정해도 GPU를 점유하지 않기 때문에 프로그램에 따라서는 경합을 일으킬 수 있음
    - 특별한 상황이 아니라면 requests와 limits 값을 함께 설정하여 사용하는 것을 추천
    - limits만 지정한 경우 자동으로 같은 값이 requests에 설정되기 때문에 limits만 지정할 수도 있음

리소스 제한

- ❖ 오버커밋 과 리소스 부족
  - ✓ 오버커밋 테스트
    - 매니페스트: sample-resource.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-resource
spec:
  replicas: 2
  selector:
    matchLabels:
      app: sample-app
```

리소스 제한

- ❖ 오버커밋 과 리소스 부족
  - ✓ 오버커밋 테스트
    - ● 매니페스트: sample-resource.yaml

```
template:
  metadata:
    labels:
      app: sample-app
  spec:
    containers:
    - name: nginx-container
      image: nginx:1.17
      resources:
        requests:
          memory: "1024Mi"
          cpu: "500m"
        limits:
          memory: "2048Mi"
          cpu: "1000m"
```

리소스 제한

- ❖ 오버커밋 과 리소스 부족
  - ✓ 오버커밋 테스트
    - 리소스 배포: kubectl apply -f sample-resource.yaml
    - 리소스 확인: kubectl get pods

```
NAME                                READY   STATUS    RESTARTS   AGE
sample-resource-55978d875f-2n97h    1/1     Running   0          63s
sample-resource-55978d875f-4tx5x    1/1     Running   0          63s
```

    - 스케일 아웃: kubectl scale deployment --replicas 6 sample-resource
    - 리소스 확인: kubectl get pods

```
NAME                                READY   STATUS    RESTARTS   AGE
sample-resource-55978d875f-2n97h    1/1     Running   0          4m11s
sample-resource-55978d875f-4tx5x    1/1     Running   0          4m11s
sample-resource-55978d875f-hl9d7    0/1     Pending   0          6s
sample-resource-55978d875f-tqvd7    0/1     Pending   0          6s
sample-resource-55978d875f-xgfj5    0/1     Pending   0          6s
sample-resource-55978d875f-zf86x    0/1     Pending   0          6s
```

리소스 제한

- ❖ 오버커밋 과 리소스 부족
  - ✓ 오버커밋 테스트
    - 리소스 확인: kubectl get deployments

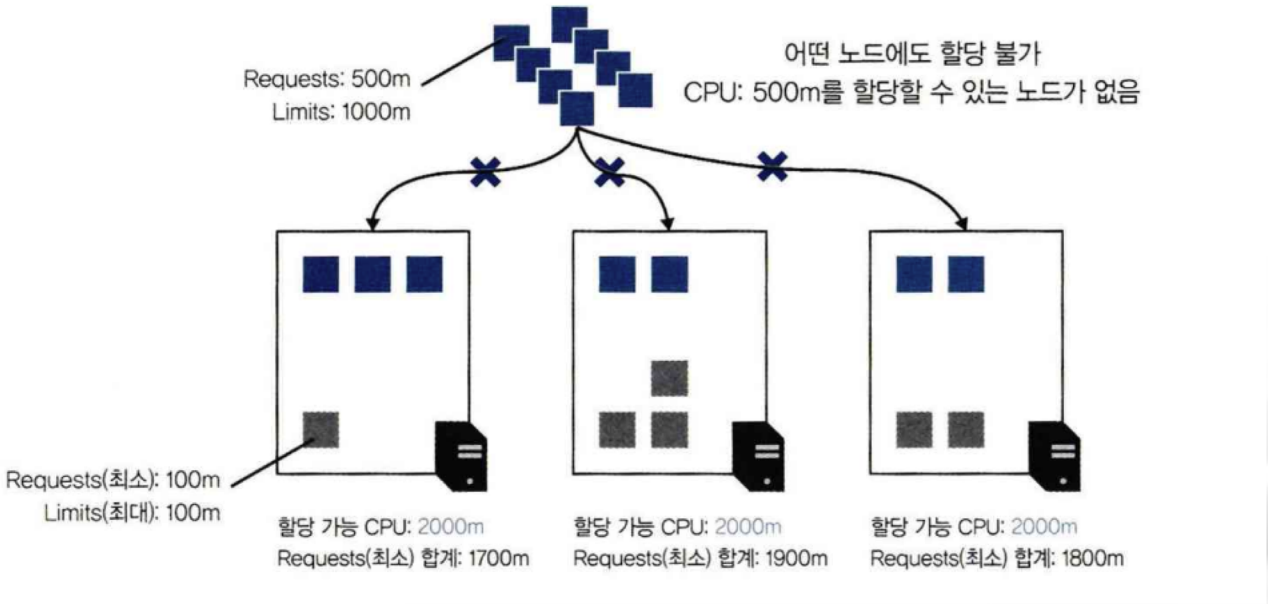
```
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
sample-resource   2/6     6            2           5m32s
```

    - 노드 확인: kubectl describe node worker1

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           requests      limits
  --------           --------      ------
  cpu                600m (60%)    1 (100%)
  memory             1074Mi (57%)  2Gi (110%)
  ephemeral-storage  0 (0%)        0 (0%)
  hugepages-1Gi      0 (0%)        0 (0%)
  hugepages-2Mi      0 (0%)        0 (0%)
  hugepages-32Mi     0 (0%)        0 (0%)
  hugepages-64Ki     0 (0%)        0 (0%)
```

리소스 제한

- ❖ 오버커밋 과 리소스 부족
  - ✓ 오버커밋 테스트
    - ● CPU 리소스를 할당할 수 없는 상태



리소스 제한

- ❖ 오버커밋 과 리소스 부족
  - ✓ 오버커밋 테스트
    - 생성된 파드는 CPU requests 값 500m와 CPU limits 값 1,000m처럼 requests 리소스 양 < limits 리소스 양으로 설정되어 있는데 이 상태에서 파드에 부하가 증가하면 노드 리소스 사용량이 100%를 초과하더라도 오버 커밋하여 실행

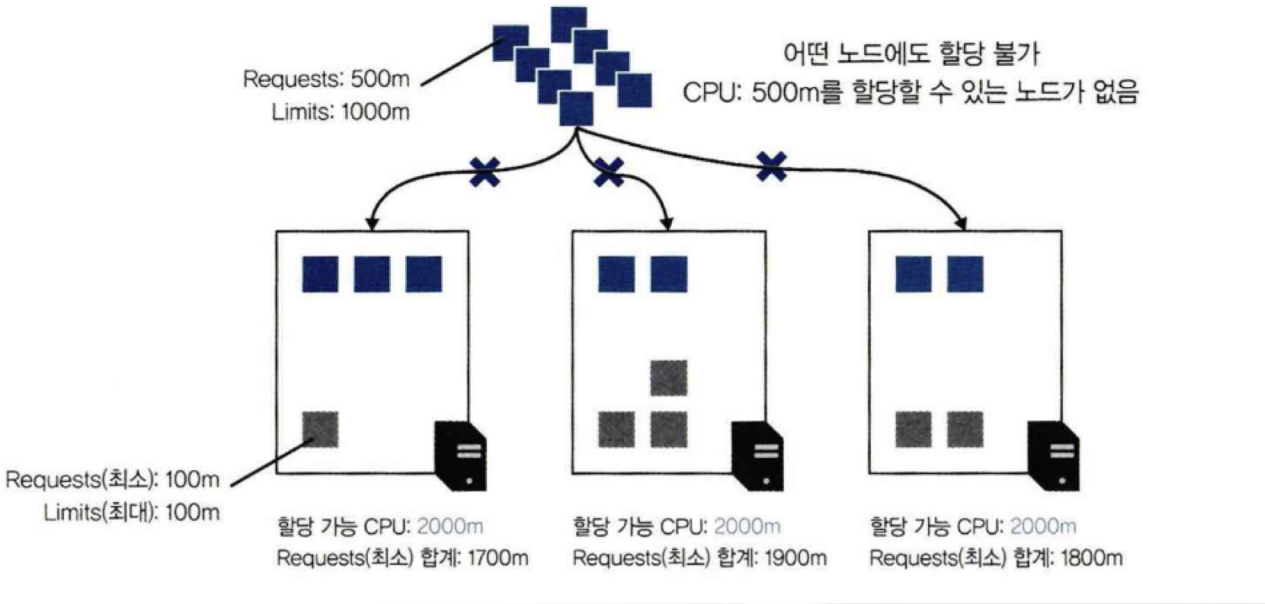
리소스 제한

- 여러 컨테이너 사용 시 리소스 할당
  - ✓ 파드에는 여러 컨테이너와 여러 초기화 컨테이너가 포함되어 있을 수 있는데 스케줄링될 때는 파드 단위로 수행되어 필요한 리소스는 이러한 여러 컨테이너를 기반으로 계산이 이루어짐
  - ✓ 필요한 리소스 양은 모든 컨테이너의 리소스 합계 또는 모든 초기화 컨테이너의 리소스 최댓값 중 큰 쪽을 그 파드의 리소스 값으로 사용

Cluster Autoscaler & 리소스 부족

❖ 개요

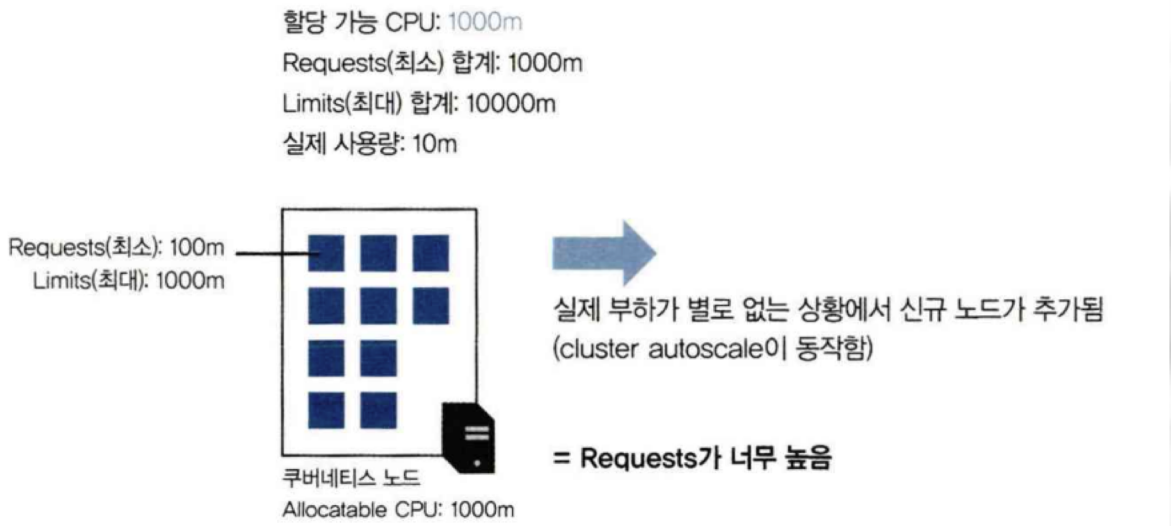
- ✓ 쿠버네티스에는 Cluster Autoscaler가 구현된 환경이 많음
- ✓ Cluster Autoscaler는 쿠버네티스 클러스터 자체의 오토 스케일링을 의미하며 수요에 따라 쿠버네티스 노드를 자동으로 추가하는 기능



Cluster Autoscaler & 리소스 부족

❖ 개요

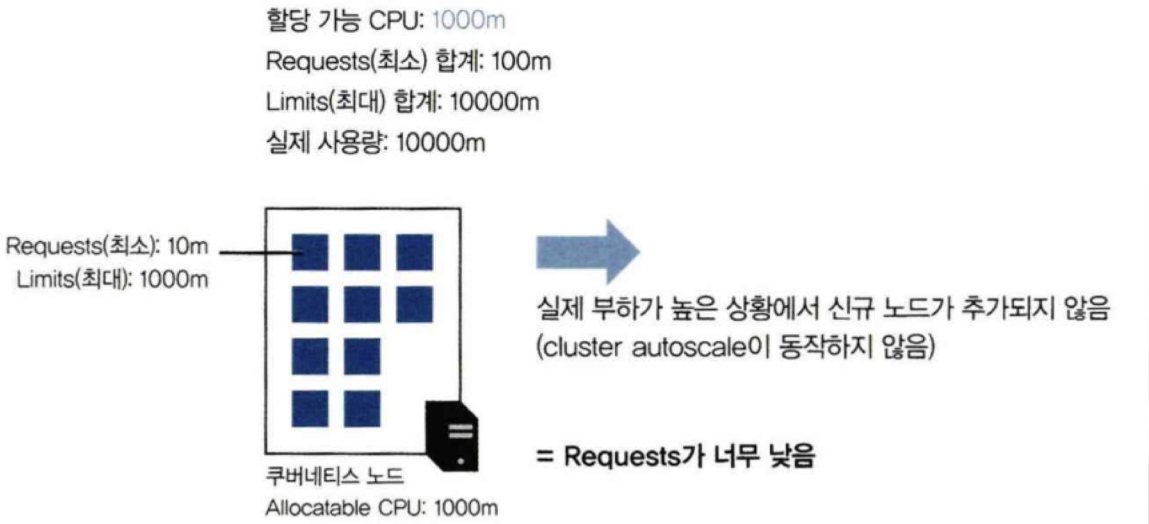
- ✓ Cluster Autoscaler는 클러스터 전체나 각 노드의 부하 평균이 높아졌을 때 확장하는 것처럼 보이지만 실제 동작은 조금 다름
- ✓ Pending 상태의 파드가 생기는 타이밍에 처음으로 Cluster Autoscaler가 동작하게 되는데 requests와 limits를 적절하게 설정하지 않은 상태에서는 실제 노드의 부하 평균이 낮은 상황에서도 스케일 아웃이 되거나 부하 평균이 높은 상황임에도 스케일 아웃이 되지 않는 상황이 발생
- ✓ 기본적으로 리소스에 의한 스케줄링은 requests(최소)를 기준으로 이루어지는데 requests를 초과하여 할당한 경우에는 최소 리소스 요청만으로 리소스가 꽉 차 버려서 신규 노드를 추가해야만 하는데 이때 실제 컨테이너 프로세스가 사용하는 리소스 사용량은 고려되지 않음



Cluster Autoscaler & 리소스 부족

❖ 개요

✓ 반대로 requests를 낮게 설정한 상태에서 limits 차이가 크게 나는 상황에서는 각 컨테이너는 limits로 할당된 리소스를 최대로 사용하는데 실제 리소스 사용량이 높아졌더라도 requests 합계로 보면 아직 스케줄링이 가능하기 때문에 클러스터가 스케일 아웃하지 않는 상황이 발생



Cluster Autoscaler & 리소스 부족

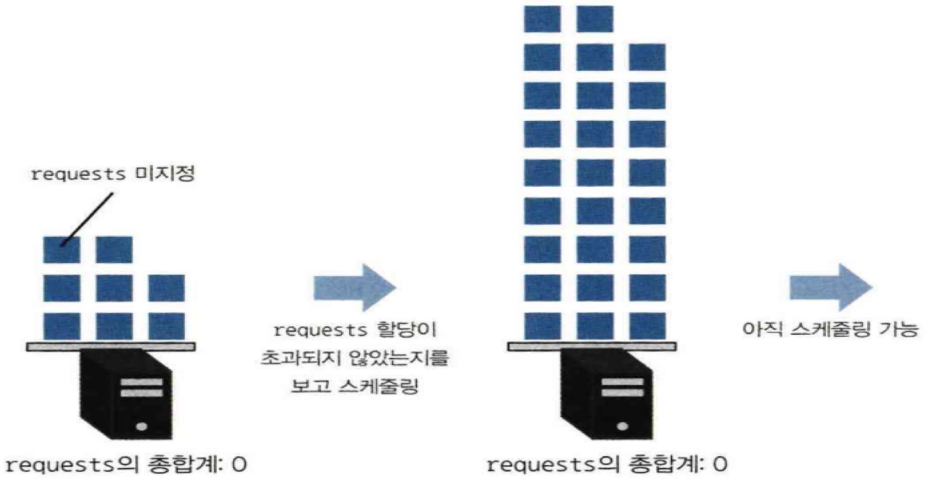
- 개요
  - ✓ 기본 정책으로 requests와 limits에 너무 큰 차이를 주지 않을 것, requests를 너무 크게 설정하지 않을 것 이 두 가지를 명심
  - ✓ 실제 값을 정할 때 requests와 limits를 낮게 설정하고 성능 테스트를 하면서 올려가는 것이 좋음
  - ✓ 메모리의 경우 낮게 설정하면 부하 테스트에서 OOM으로 프로세스가 kill되기(파드가 정지됨) 때문에 OOM이 발생하지 않을 정도의 리소스 할당을 하나의 지표로 생각

LimitRange를 사용한 리소스 제한

- 개요
  - ✓ LimitRange를 사용하면 파드 등에 대해 CPU나 메모리 리소스의 최솟값과 최댓값, 기본값 등을 설정할 수 있음
  - ✓ LimitRange가 네임스페이스에 제한을 주려면 네임스페이스마다 설정이 필요
  - ✓ LimitRange는 신규로 파드를 생성할 때 사용되므로 기존 파드에는 영향을 주지 않음
  - ✓ 제한 가능한 항목
    - default: 기본 limits
    - defaultRequest: 기본 requests
    - max: 최대 리소스
    - min: 최소 리소스
    - maxLimitRequestRatio: limits/requests 비율
  - ✓ LimitRange를 설정할 수 있는 리소스와 설정 항목
    - Container: default/defaultRequest/max/min/maxLimitRequestRatio
    - Pod: max/min/maxLimitRequestRatio
    - PersistentVolumeClaim: max/min

LimitRange를 사용한 리소스 제한

- 기본으로 생성되는 LimitRange
  - ✓ 파드나 컨테이너에 대해 리소스 제한을 전혀 하지 않을 경우 실제 노드 부하와 관계없이 계속 파드를 스케줄링
  - ✓ 스케줄러는 requests에서 지정한 리소스를 확보할 수 있을지에 따라 스케줄링 여부를 판단하기 때문에 모든 파드에 requests를 지정하지 않았을 경우 스케줄러는 계속 그 노드에 스케줄링하게 됨
  - ✓ 리소스 소비가 과도하게 많아지면 메모리는 OOM Killer에 의해 프로세스가 정지되지만 CPU는 전체 동작이 느려져 최악의 경우 운영체제가 동작하지 않게 됨
  - ✓ requests 미지정으로 인한 과도한 스케쥴링



노드 CPU 코어 수	4 (4000m)
할당 가능한 requests 총합계	3000m

LimitRange를 사용한 리소스 제한

- Container에 대한 LimitRange
  - ✓ 컨테이너 리소스를 제한하는 경우 type: Container의 LimitRange로 설정
  - ✓ 개별적으로 컨테이너에 대해 리소스 제한을 정의하지 않고 기본 설정을 정의할 수도 있으며 컨테이너에 허용하는 최대/최소 리소스 제한을 설정할 수도 있음
  - ✓ maxLimitRequestRatio를 설정하면 requests와 limits의 차이에 따른 과도한 오버커밋도 피할 수 있음

LimitRange를 사용한 리소스 제한

- Container에 대한 LimitRange
  - ✓ 컨테이너에 리소스 제한을 하는 LimitRange 매니페스트: sample-limitrange-container.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: sample-limitrange-container
  namespace: default
```

LimitRange를 사용한 리소스 제한

- Container에 대한 LimitRange
  - ✓ 컨테이너에 리소스 제한을 하는 LimitRange 매니페스트: sample-limitrange-container.yaml

```
spec:
  limits:
  - type: Container
    default:
      memory: 512Mi
      cpu: 500m
    defaultRequest:
      memory: 256Mi
      cpu: 250m
    max:
      memory: 1024Mi
      cpu: 1000m
    min:
      memory: 128Mi
      cpu: 125m
    maxLimitRequestRatio:
      memory: 2
      cpu: 2
```

  - ✓ 리소스 생성: kubectl apply -f sample-limitrange-container.yaml

LimitRange를 사용한 리소스 제한

- Container에 대한 LimitRange
  - ✓ 최저 CPU requests보다 낮은 파드를 생성하는 매니페스트: sample-pod-overrequest.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod-overrequest
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    resources:
      requests:
        cpu: 100m
      limits:
        cpu: 100m
```

  - ✓ 리소스 생성: kubectl apply -f sample-pod-overrequest.yaml

```
Error from server (Forbidden): error when creating "sample-pod-overrequest.yaml":
pods "sample-pod-overrequest" is forbidden: minimum cpu usage per Container is
125m, but request is 100m
```

LimitRange를 사용한 리소스 제한

- Container에 대한 LimitRange
  - ✓ requests 와 limits의 차이가 네 배인 파드를 생성하는 매니페스트: sample-pod-overratio.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod-overratio
spec:
  containers:
  - name: nginx-container
    image: nginx:1.17
    resources:
      requests:
        cpu: 125m
      limits:
        cpu: 500m
```

  - ✓ 리소스 생성: kubectl apply -f sample-pod-overratio.yaml

```
Error from server (Forbidden): error when creating "sample-pod-overratio.yaml": pods "sample-pod-overratio" is forbidden: cpu max limit to request ratio per Container is 2, but provided ratio is 4.000000
```

LimitRange를 사용한 리소스 제한

- Pod에 대한 LimitRange
  - ✓ Pod에 대해 리소스를 제한하는 매니페스트: sample-limitrange-pod.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: sample-limitrange-pod
  namespace: default
spec:
  limits:
  - type: Pod
    max:
      memory: 2048Mi
      cpu: 2000m
    min:
      memory: 128Mi
      cpu: 125m
    maxLimitRequestRatio:
      memory: 1.5
      cpu: 1.5
```

  - ✓ 리소스 생성: kubectl apply -f sample-limitrange-pod.yaml

LimitRange를 사용한 리소스 제한

- PersistentVolumeClaim에 대한 LimitRange
  - ✓ PersistentVolumeClaim에 대해 리소스를 제한하는 매니페스트: sample-limitrange-pvc.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: sample-limitrange-pvc
  namespace: default
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 20Gi
    min:
      storage: 3Gi
```

  - ✓ 리소스 생성: kubectl apply -f sample-limitrange-pvc.yaml

LimitRange를 사용한 리소스 제한

- PersistentVolumeClaim에 대한 LimitRange
  - ✓ 25Gi의 PersistentVolumeClaim을 사용하는 매니페스트: sample-pvc-fail.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sample-pvc-fail
spec:
  resources:
    requests:
      storage: 25Gi
  accessModes:
  - ReadWriteOnce
```

  - ✓ 리소스 생성: kubectl apply -f sample-pvc-fail.yaml

```
Error from server (Forbidden): error when creating "sample-pvc-fail.yaml":
persistentvolumeclaims "sample-pvc-fail" is forbidden: maximum storage usage per
PersistentVolumeClaim is 20Gi, but request is 25Gi
```

리소스 쿼터

- 개요
  - ✓ 리소스 쿼터(ResoiirceQuota)를 사용하여 각 네임스페이스마다(가상 쿠버네티스 클러스터) 사용 가능한 리소스 제한 가능
  - ✓ 리소스 쿼터는 생성이나 변경으로 그 시점에 제한이 걸린 상태가 되어도 이미 생성된 리소스에는 영향을 주지 않기 때문에 주의
  - ✓ 리소스 쿼터는 크게 생성 가능한 리소스 수 제한 과 리소스 사용량 제한으로 나눔

리소스 쿼터

- 생성 가능한 리소스 제한
  - ✓ configmap을 10개로 제한하는 리소스 쿼터
    - 리소스 쿼터 매니페스트: sample-resourcequota.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: sample-resourcequota
  namespace: default
spec:
  hard:
    count/configmaps: 10
```

    - 리소스 쿼터 생성: kubectl apply -f sample-resourcequota.yaml
    - 현재 사용 현황 과 쿼터 확인: kubectl describe resourcequota sample-resourcequota

```
Name:              sample-resourcequota
Namespace:         default
Resource           Used  Hard
--------           ----  ----
count/configmaps   1     10


count/configmaps: 10
```

리소스 쿼터

- ❖ 생성 가능한 리소스 제한
  - ✓ configmap을 10개로 제한하는 리소스 쿼터
    - configmap 11개 생성: for i in `seq 1 11`; do kubectl create configmap conf-$i --from-literal=key1=val1; done

```
configmap/conf-1 created
configmap/conf-2 created
configmap/conf-3 created
configmap/conf-4 created
configmap/conf-5 created
configmap/conf-6 created
configmap/conf-7 created
configmap/conf-8 created
configmap/conf-9 created
error: failed to create configmap: configmaps "conf-10" is forbidden: exceeded quota: sample-resourcequota, requested: count/configmaps=1, used: count/configmaps=10, limited: count/configmaps=10
error: failed to create configmap: configmaps "conf-11" is forbidden: exceeded quota: sample-resourcequota, requested: count/configmaps=1, used: count/configmaps=10, limited: count/configmaps=10
```

리소스 쿼터

- 생성 가능한 리소스 제한
  - ✓ 새로운 문법으로 리소스 수 제한
    - 매니페스트: sample-resourcequota-count-new.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: sample-resourcequota-count-new
  namespace: default
spec:
  hard:
    count/persistentvolumeclaims: 10
    count/services: 10
    count/secrets: 10
    count/configmaps: 10
    count/replicationcontrollers: 10
    count/deployments.apps: 10
    count/replicasets.apps: 10
    count/statefulsets.apps: 10
    count/jobs.batch: 10
    count/cronjobs.batch: 10
    count/deployments.extensions: 10
```

리소스 쿼터

- 생성 가능한 리소스 제한
  - ✓ 두 가지 방식을 혼합한 리소스 수 제한
    - 매니페스트: sample-resourcequota-count-old.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: sample-resourcequota-count-old
  namespace: default
spec:
  hard:
    #이전 방식
    sample-storageclass.storageclass.storage.k8s.io/persistentvolumeclaims: 10
    services.loadbalancers: 10
    services.nodeports: 10

    # 새로운 방식
    pods: 10
    persistentvolumeclaims: 10
    replicationcontrollers: 10
    secrets: 10
    configmaps: 10
    services: 10
    resourcequotas: 10
```

리소스 쿼터

- 리소스 사용량 제한
  - ✓ 리소스 사용량도 제한할 수 있음
  - ✓ CPU나 메모리에 대해 컨테이너에 할당 가능한 리소스 양을 제한할 수 있음
  - ✓ 스토리지는 limits가 존재하지 않고 requests만 지정 가능하며 스토리지 클래스마다 제한을 둘 수 있음
  - ✓ 매니페스트: sample-resourcequota-usable.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: sample-resourcequota-usable
  namespace: default
spec:
  hard:
    # requests로 사용량 제한
    requests.memory: 2Gi
    requests.storage: 5Gi
    sample-storageclass.storageclass.storage.k8s.io/requests.storage: 5Gi
    requests.ephemeral-storage: 5Gi
    requests.nvidia.com/gpu: 2
    # limits로 사용량 제한
    limits.cpu: 4
    limits.ephemeral-storage: 10Gi
    limits.nvidia.com/gpu: 4
```

Horizontal Pod Autoscaler

❖ 개요

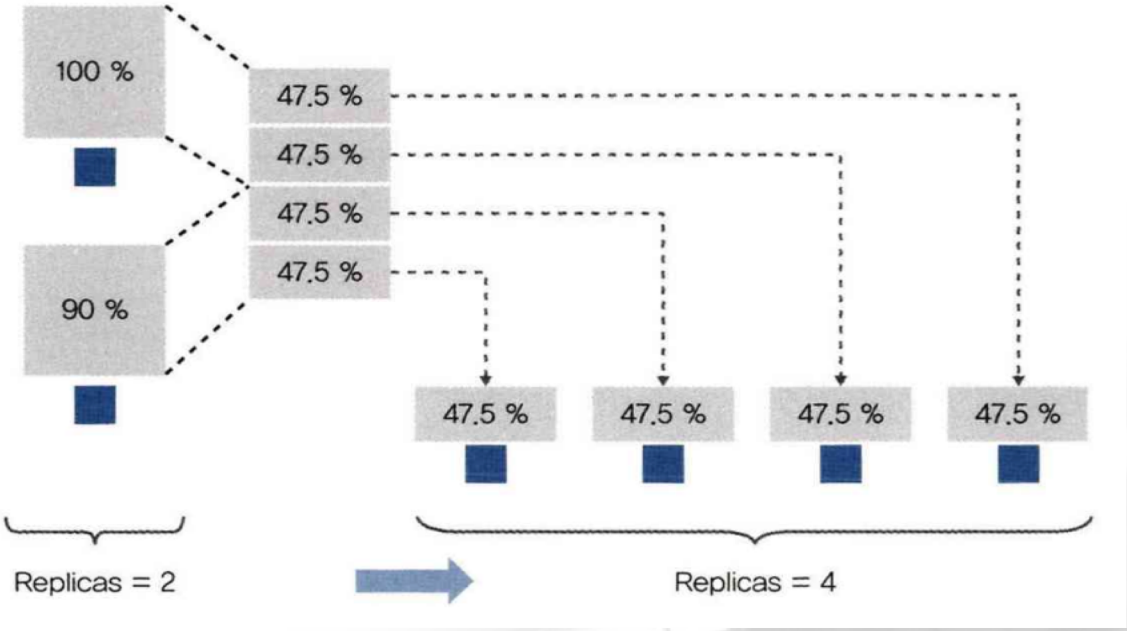
- ✓ HorizontalPodAutoscaler(HPA)는 디플로이먼트/레플리카셋/레플리케이션 컨트롤러의 레플리카 수를 CPU 부하 등에 따라 자동으로 스케일하는 리소스
- ✓ 부하가 높아지면 스케일 아웃하고 부하가 낮아지면 스케일 인 됨
- ✓ 파드에 Resource requests가 설정되어 있지 않은 경우에는 동작하지 않음

Horizontal Pod Autoscaler
파드(디플로이먼트의 레플리카 수)의 오토 스케일링

Horizontal Pod Autoscaler

❖ 개요

- ✓ HorizontalPodAutoscaler는 30초에 한 번 꼴로 오토 스케일링 여부를 확인
- ✓ 필요한 레플리카 수를 계산 수식(ceil 함수는 소수점 첫째 자리에서 올림하여 정수 값을 리턴하는 함수)
  - 필요한 레플리카 수 = ceil(sum(파드의 현재 CPU 사용률) / targetAverageUtilization)



  - 필요한 레플리카 수 = ceil(sum(파드의 현재 CPU 사용률) / targetAverageUtilization)

    = ceil((100% + 90%) / 50%) = ceil (190 / 50) = ceil (3.8) = 4

Horizontal Pod Autoscaler

- 개요
  - ✓ CPU 사용률은 metrics-server(구 Heapster)에서 가져온 각 파드의 1분간 평균값을 사용
  - ✓ 최대 3분에 1회 스케일 아웃을 실행하고 최대 5분에 1회 스케일 인을 실행함으로써 기동 시에만 CPU의 부하가 상승하는 메트릭 노이즈 영향을 최소화할 수 있음
  - ✓ 일반적으로 스케일 아웃은 빨리해야 하지만 스케일 인은 그렇게까지 중요하지 않아 이런 차이가 있고 스케일링의 실행 조건은 다음과 같은 조건식으로 결정되는데 이는 파드 수가 매우 많은 경우 등은 어느 정도 허용되고 세밀한 스케일링이 실행되지 않도록 고려된 조건식
    - 스케일 아웃 조건식: avg(파드의 현재 CPU 사용률) / targetAverageltilization > 1.1
    - 스케일 인 조건식: avg(파드의 현재 CPU 사용률) / targetAverageltilization < 0.9
  - ✓ HorizontalPodAutoscaler에서는 스케일 조건과 최저 레플리카 수/최고 레플리카 수를 지정

Horizontal Pod Autoscaler

- 사용 방법
  - ✓ kubectl 명령으로 사용: kubectl autoscale deployment 디플로이먼트 --cpu-percent=비율 --min=최소개수 --max=최대개수
  - ✓ yaml 파일

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: sample-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: 디플로이먼트
  minReplicas: 최소개수
  maxReplicas: 최대개수
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 비율
```

Horizontal Pod Autoscaler

- ❖ HPA 실습
  - ✓ Metric Server 설치
    - 매니페스트 이용: kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
    - 로컬 클러스터 사용 시 수행
      - ◆ 편집기 수행: kubectl edit deployment metrics-server -n kube-system
      - ◆ spec.template.spec.containers.args 섹션을 찾아 아래 두 줄을 추가
        - --kubelet-insecure-tls: 인증서 검증 건너뛰기
        - --kubelet-preferred-address-types=InternalIP: 로컬 노드 통신 설정(생략 가능하기도 함)
    - 확인
      - ◆ Pod 상태 확인: kubectl get pods -n kube-system | grep metrics-server
      - ◆ 노드 리소스 사용량 확인: kubectl top nodes
      - ◆ 개별 Pod 리소스 사용량 확인: kubectl top pods
    - Helm으로 설치
      - ◆ helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
      - ◆ helm upgrade --install metrics-server metrics-server/metrics-server --namespace kube-system --set args={--kubelet-insecure-tls}

Horizontal Pod Autoscaler

- ❖ HPA 실습
  - ✓ Deployment 와 Service 생성
    - ● 매니페스트 파일: php-apache.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
```

Horizontal Pod Autoscaler

- ❖ HPA 실습
  - ✓ Deployment 와 Service 생성
    - ● 매니페스트 파일: php-apache.yaml

```
template:
  metadata:
    labels:
      run: php-apache
  spec:
    containers:
    - name: php-apache
      image: registry.k8s.io/hpa-example
      ports:
      - containerPort: 80
      resources:
        limits:
          cpu: 500m
        requests:
          cpu: 200m # HPA가 계산하는 기준점
```

Horizontal Pod Autoscaler

- ❖ HPA 실습
  - ✓ Deployment 와 Service 생성
    - 매니페스트 파일: php-apache.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

    - 리소스 배포: kubectl apply -f php-apache.yaml
    - 확인
      - ◆ kubectl get pods
      - ◆ kubectl get services

Horizontal Pod Autoscaler

- HPA 실습
  - ✓ HorizontalPodAutoscaler 생성
    - HorizontalPodAutoscaler 생성: kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
    - HPA를 실시간으로 확인: kubectl get horizontalpodautoscalers --watch

```
NAME         REFERENCE                     TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache         cpu: 0%/10%     1         10        1          15s
php-apache   Deployment/php-apache         cpu: 13%/10%    1         10        1          2m6s
php-apache   Deployment/php-apache         cpu: 12%/10%    1         10        2          2m34s
php-apache   Deployment/php-apache         cpu: <unknown>/10%   1         10        2          2m43s
```

Horizontal Pod Autoscaler

- HPA 실습
  - ✓ 테스트
    - 트래픽 전송: kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
    - Apache Bench를 설치한 경우에 트래픽 전송: ab -n 100000 -c 500 http://127.0.0.1:8080/

Horizontal Pod Autoscaler

- HorizontalPodAutoscaler에서 사용 가능한 메트릭의 종류
  - ✓ Resource: CPU/Memory
  - ✓ Object: Kubernetes Object Metric(ex: Ingress hits/s)
  - ✓ Pods: Pod Metric(ex: Pod Connection 수)
  - ✓ External: kubernetes 외부의 Metric(ex: LB의 QPS, Cloud Pub/Sub에 쌓여 있는 메시지 수)

Horizontal Pod Autoscaler

- Horizontal Pod Autoscaler 스케일링 동작 설정
  - ✓ autoscaling/v2beta2의 HorizontalPodAutoscaler에서 spec.behavior 필드가 추가되어 오토스케일링 빈도나 증감 가능한 레플리카 수 같은 동작을 리소스 단위로 설정할 수 있게 됨
  - ✓ autoscaling/v2beta1에서는 쿠버네티스 클러스터 관리자가 시스템 구성 요소에 대해 통일되게 설정해야 해서 응용프로그램마다 조건을 변경할 수 없음
  - ✓ v2beta1 과의 차이점
    - 다중 지표 지원: CPU와 Memory를 동시에 감시할 수 있는데 이 경우 둘 중 하나만 임계치를 넘어도 스케일 아웃(Pod 증가)이 발생
    - target.type
      - Utilization: 백분율(%) 기준 (예: 50%)
      - AverageValue: 절대 수치 기준 (예: 500Mi, 100m)
    - Behavior(선택 사항): v2beta2부터는 스케일링이 발생하는 속도(예: 한 번에 최대 몇 개의 파드를 늘릴지, 줄어들 때 대기 시간은 얼마인지)를 spec.behavior 섹션에서 세밀하게 제어할 수 있음

Horizontal Pod Autoscaler

- Horizontal Pod Autoscaler 스케일링 동작 설정
  - ✓ 매니페스트: hpa-v2.yaml

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache-v2
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
```

Horizontal Pod Autoscaler

❖ Horizontal Pod Autoscaler 스케일링 동작 설정

✓ 매니페스트: hpa-v2.yaml

```
metrics:
# 1. CPU 기반 스케일링 설정
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
# 2. Memory 기반 스케일링 설정 (v2beta2의 강점)
- type: Resource
  resource:
    name: memory
    target:
      type: AverageValue
      averageValue: 500Mi
```

Horizontal Pod Autoscaler

- ❖ Horizontal Pod Autoscaler 스케일링 동작 설정
  - ✓ 매니페스트: hpa-v2.yaml

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
    selectPolicy: Max
```

Horizontal Pod Autoscaler

- ❖ Horizontal Pod Autoscaler 스케일링 동작 설정
  - ✓ 내용
    - spec.behavior에는 스케일 인(ScaleDown)과 스케일 아웃(ScaleUp) 각각에 대해 여러 정책을 정의
    - 정책에는 일정 기간 증감 가능한 레플리카 수를 제한하는 정의를 기술
    - 증감 가능한 레플리카 수 제한에는 현재 레플리카 수에 대한 백분율(type: Percent) 또는 정숫값(type: Pod)을 지정할 수 있음
    - 스케일 아웃 정책으로 15초 동안 100% 증가 와 15초 동안 네 개 파드 추가라는 두 개의 정책이 설정되어 있음
    - type: Percent에 200%가 지정된 경우는 15초 동안 최대 세 배(100%+200%)의 레플리카 수가 되는 것을 의미
    - selectPolicy에는 여러 개의 지정된 정책 중 어떤 값을 선택할지 지정하는데. 예를 들어 코드에서는 스케일 아웃 정책으로 15초 동안 100% 증가 와 15초 동안 네 개 파드 추가라는 두 개의 정책이 정의되어 있지만 그 중 최댓값(selectPolicy: Max)이 선택 됨
    - 레플리카 수가 두 개 파드 상태에서 12개 파드로 증가해야 하는 경우 첫 스케일 아웃 시점에서 100% 추가(두 개 파드) 또는 네 개 파드 추가 중 최댓값인 네 개 파드가 추가되고 총 여섯 개의 파드가 되고 나서 15초 후 100% 추가(여섯 개 파드) 또는 네 개 파드 추가 중 최댓값인 여섯 개 파드가 추가되고 총 12개의 파드가 되며 반대로 selectPolicy: Min은 여러 개의 정책 중 최솟값을 선택
    - 특수한 값으로 Disabled가 선택된 경우 스케일 아웃 또는 스케일 인이 비활성화되는데 spec.behavior.scaleDown.selectPolicy만을 Disabled로 지정한 경우 스케일 아웃만 실행되고 스케일 인은 실행되지 않게 할 수 있음

Horizontal Pod Autoscaler

- Horizontal Pod Autoscaler 스케일링 동작 설정
  - ✓ 내용
    - selectPolicy 선택 항목
      - Min: 지정한 기간 동안 최소 변화량으로 제한을 설정
      - Max: 지정한 기간 동안 최대 변화량으로 제한을 설정
      - Disabled: 스케일링을 비활성화
    - stabilizationWindowSeconds는 레플리카 수가 빈번하게 증감하는 것을 방지하고 트래픽 스파이크로 레플리카 수가 급감하거나 급증하는 것을 피하기 위한 설정
    - stabilizationWindowSeconds에서 지정된 기간 동안 추천 레플리카 수를 바탕으로 레플리카 수를 결정
    - 스케일 인의 경우에는 지정한 기간 동안 최댓값이 선택되고 스케일 아웃인 경우에는 지정한 기간 동안 최솟값을 선택하는데 이 기능은 주로 레플리카 수가 갑자기 감소하는 것을 막기 위해 사용하는 것이 좋음
    - stabilizationWindowSeconds가 0인 경우 바로 추천 레플리카 수로 변경하기 때문에 스케일 아웃인 경우에는 stabilizationWindowSeconds=0으로 지정해 두는 것이 좋음

Vertical Pod Autoscaler

- 개요
  - ✓ 파드에 할당하는 CPU/메모리의 requests(할당)는 실제로 성능을 측정하여 적절한 값을 설정해야 하므로 서비스 환경에 배포하지 않으면 조절이 어려울 수 있는데 이 requests 값은 적절한 범위로 조절해 두지 않으면 실제로 사용되는 리소스 양과 할당된 리소스 양에 차이가 생겨 리소스 부족으로 인해 동작이 불안정해지거나 반대로 리소스 과잉으로 인해 낭비가 생기기도 하는데 이 문제를 해결하는 것이 VerticalPodAutoscaler(VPA)
  - ✓ VerticalPodAutoscaler는 컨테이너에 할당하는 CPU/메모리 리소스의 할당을 자동으로 스케일해주는 기능
  - ✓ HorizontalPodAutoscaler가 파드를 스케일 아웃했다면 VerticalPodAutoscaler는 파드를 스케일 업

Vertical Pod Autoscaler

❖ HPA 와 VPA 차이

구분	HPA (Horizontal)	VPA (Vertical)
확장 방식	Pod의 개수를 늘림 (가로 확장)	Pod의 사양을 높임 (세로 확장)
주요 목적	트래픽 증가 대응	적절한 리소스 할당량 최적화
기본 포함	O (쿠버네티스 기본 기능)	X (별도 설치 필요)

Vertical Pod Autoscaler

- 설치
  - ✓ HPA는 쿠버네티스 기본 내장 기능이지만 VPA(Vertical Pod Autoscaler)는 쿠버네티스 설치 시 기본으로 포함되지 않는 별도의 애드온(Add-on)
  - ✓ GitHub에서 공식 VPA 저장소를 클론하거나 다운로드: git clone https://github.com/kubernetes/autoscaler.git
  - ✓ 디렉토리 이동: cd autoscaler/vertical-pod-autoscaler
  - ✓ 설치 스크립트 실행하는데 이 과정에서 필요한 CRD와 권한(RBAC), Admission Controller 등이 자동으로 생성: ./hack/vpa-up.sh
  - ✓ 설치가 완료되면 아래 명령어로 VerticalPodAutoscaler 리소스를 인식하는지 확인: kubectl get crd | grep verticalpodautoscalers

Vertical Pod Autoscaler

- 실습
  - ✓ 스케일 업을 위한 디플로이먼트: vpa-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vpa-demo-app
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: vpa-demo-app
```

Vertical Pod Autoscaler

- 실습
  - ✓ 스케일 업을 위한 디플로이먼트: vpa-deployment.yaml

```
template:
  metadata:
    labels:
      app: vpa-demo-app
  spec:
    containers:
    - name: app-container
      image: nginx:1.21
      ports:
      - containerPort: 80
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 200m
          memory: 256Mi
      env:
      - name: DEMO_ENV
        value: "vpa-test"
```

Vertical Pod Autoscaler

- 실습
  - ✓ 스케일 업을 위한 디플로이먼트: vpa-deployment.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: vpa-demo-service
  namespace: default
spec:
  selector:
    app: vpa-demo-app
  ports:
  - port: 80
    targetPort: 80
  type: ClusterIP
```

Vertical Pod Autoscaler

- 실습
  - ✓ VerticalPodAutoscaler 매니페스트: vpa.yaml

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-demo-app-vpa
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: vpa-demo-app

  # VPA 업데이트 정책 설정
  updatePolicy:
    updateMode: "Auto"   # Off, Initial, Recreation, Auto 중 선택
```

Vertical Pod Autoscaler

- 실습
  - ✓ VerticalPodAutoscaler 매니페스트: vpa.yaml

```
# 리소스 정책 설정 (선택사항)
resourcePolicy:
  containerPolicies:
  - containerName: app-container
    # 최소 리소스 제한
    minAllowed:
      cpu: 50m
      memory: 64Mi
    # 최대 리소스 제한
    maxAllowed:
      cpu: 1000m
      memory: 1Gi
    # 제어할 리소스 타입
    controlledResources: ["cpu", "memory"]
    # 스케일링 모드
    mode: Auto
```

Vertical Pod Autoscaler

- ❖ 실습
  - ✓ VerticalPodAutoscaler 매니페스트: sample-vpa.yaml
    - VerticalPodAutoscaler에서는 컨테이너 이름(containerName)을 지정하고 각각의 컨테이너에 오토 스케일링 활성화를 지정하면 할당 가능한 리소스 제약(minAllowed/maxAllowed)을 설정 가능
    - no-vpa-container 컨테이너와 지정하지 않은 나머지 모든 컨테이너 [*] 각각에 지정할 수 있고 vpa-*와 같은 정규 표현식은 사용할 수 없음
    - 각각의 컨테이너에는 VPA 대상 여부 설정 mode(Off/Auto) 또는 할당 가능한 리소스의 최솟값 minAllowed, 최댓값 maxAllowed를 지정 가능
    - requests를 변경하려면 파드 재기동이 필요하고 requests 변경으로 성능에 악영향을 미칠 수도 있기 때문에 추천 값을 계산만 하고 참고 값으로만 확인할 수 있게 할지 아니면 자동으로 스케일 업까지 할지 선택할 수 있도록 되어 있음
    - VerticalPodAutoscaler는 requests의 추천 값을 계산한 후의 동작을 선택 가능
    - VerticalPodAutoscaler에서 사용 가능한 updateMode
      - ◆ Off: requests의 추천 값을 계산만 하고 실제 변경은 없음
      - ◆ Initial: 파드가 재생성된 시점에만 추천 값을 requests로 변경
      - ◆ Recreate: 추천 값이 변경될 때 파드가 재생성되고 requests를 변경
      - ◆ InPlace: 추천 값이 변경될 때 파드가 기동한 상태로 requests를 변경
      - ◆ Auto: 추천 값이 변경될 때 InPlace 또는 Recreate로 requests를 변경

Vertical Pod Autoscaler

- 실습
  - ✓ 리소스 생성
    - VPA CRD가 설치되어 있는지 확인: kubectl get crd verticalpodautoscalers.autoscaling.k8s.io
    - kubectl apply -f vpa-deployment.yaml
    - kubectl apply -f vpa.yaml
  - ✓ 각 파드의 리소스 할당량 확인: kubectl get pods -o custom-columns=" NAME: .metadata.name, CPU_REQ:.spec.containers[*].resources.requests.cpu, MEM_REQ:.spec.containers[*].resources.requests.memory"

```
NAME      CPU_REQ     MEM_REQ
<none>    50m         250Mi
<none>    50m         250Mi
```

Vertical Pod Autoscaler

- 실습
  - ✓ 각 컨테이너 별 리소스 할당량 확인: kubectl get pods -o custom-columns="POD_NAME:.metadata.name,CONTAINER_NAMES:.spec.containers[*].name,CPU_REQ:.spec.containers[*].resources.requests.cpu"

    vpa-demo-app-658d9df694-hhhsv app-container 50m

    vpa-demo-app-658d9df694-n2rrm app-container 50m
  - ✓ VPA 상태 확인: kubectl get vp

    NAME MODE CPU MEM PROVIDED AGE

    vpa-demo-app-vpa Auto 50m 250Mi True 18m

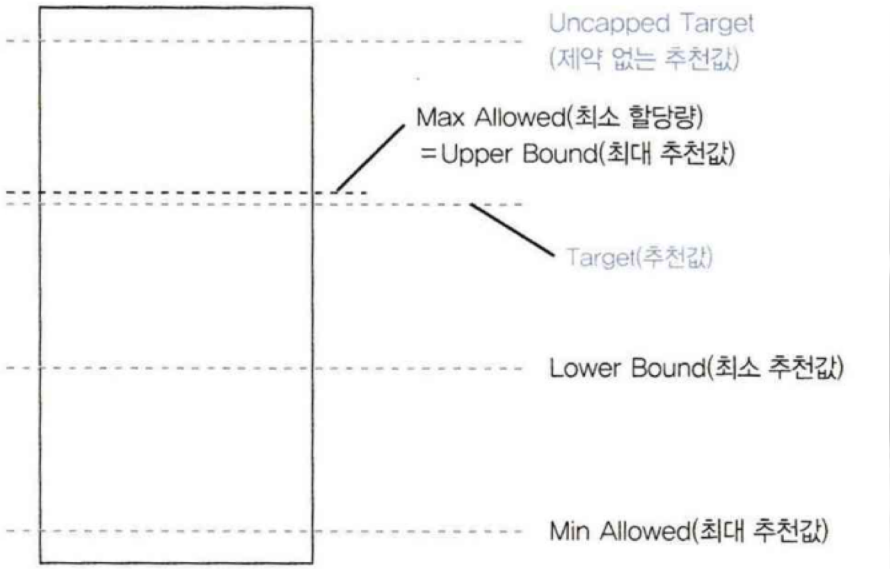
Vertical Pod Autoscaler

- ❖ 실습
  - ✓ VPA 상태 확인: kubectl describe vpa vpa-demo-app-vpa

```
Recommendation:
  Container Recommendations:
    Container Name:  app-container
    Lower Bound:
      Cpu:      50m
      Memory:   250Mi
    Target:
      Cpu:      50m
      Memory:   250Mi
    Uncapped Target:
      Cpu:      25m
      Memory:   250Mi
    Upper Bound:
      Cpu:      789m
      Memory:   825259213
```

Vertical Pod Autoscaler

- ❖ 실습
  - ✓ VPA 상태 확인: kubectl describe vpa vpa-demo-app-vpa
    - Lower Bound는 그 컨테이너에 최소한으로 필요한 리소스 양으로 이 양을 밑도는 경우 성능에 큰 영향을 미칠 수 있지만 반대로 Upper Bound는 그 컨테이너가 사용하는 리소스의 최대량으로 이 양을 초과하는 경우 리소스가 사용되지 않고 낭비될 수 있음
    - Target은 그 컨테이너가 가장 효율적으로 리소스를 사용할 수 있는 추천값으로 컨테이너의 requests를 변경할 때 이 값이 사용되는데 유사한 항목으로 Uncapped Target이 있지만 이것은 할당 가능한 리소스 제약(minAllowed/maxAllowed)을 고려하지 않고 실제 리소스 사용량 만을 기준으로 계산된 추천값이므로 참조값으로만 표시하고 requests 변경에는 사용되지 않음



Vertical Pod Autoscaler

- 실습
  - ✓ 모니터링용 부하 생성 파드: vpa-test.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: load-generator
  namespace: default
spec:
  containers:
  - name: load-generator
    image: busybox:1.35
    command: ["/bin/sh"]
    args:
    - -c
    - |
      while true; do
        wget -q -O- http://vpa-demo-service/
        sleep 1
      done
  restartPolicy: Always
```

Vertical Pod Autoscaler

- 실습
  - ✓ VPA 상태 확인: kubectl describe vpa sample-vpa

NAME	CPU_REQ	MEM_REQ
<none>	300m,100m	300Mi,100Mi
<none>	300m,100m	300Mi,100Mi
<none>	300m,100m	300Mi,100Mi
<none>	300m,100m	300Mi,100Mi

정리

- 개요
  - ✓ 파드에 리소스 할당은 CPU와 메모리 설정이 일반적이지만 GPU나 그 외의 외부 장치도 할당할 수 있도록 만들어짐
  - ✓ 할당하는 리소스의 양에는 최소 리소스(requests)와 최고 리소스(limits)가 있는데 두 가지 설정값은 예상치 못한 노드의 과부하를 피하기 위해 가능하면 설정값의 차이가 크지 않도록 설정해야 하는 점에 주의
  - ✓ requests와 limits 할당에 따라 파드에 자동으로 QoS Class가 설정됨
  - ✓ 리소스 관리의 기능
    - LimitRange: 리소스 할당의 최대, 최소, 최대와 최소가 허용하는 차이의 백분율, 기본값 설정
    - ResourceQuota: 리소스 할당 쿼터
  - ✓ 오토 스케일링 방식은 세 가지 중 선택
    - Cluster Autoscaler: 파드를 기동할 수 있는 노드가 없을 경우 신규 노드 추가
    - HorizontalPodAutoscaler: 파드의 레플리카 수를 부하에 따라 자동으로 증감
    - VerticalPodAutoscaler: 파드에 할당된 리소스(requests/limits)를 부하에 따라 자동으로 증감