

Contents

- **Workload**
 - **Pod**
 - **Pod lifecycle**
 - **Container probes**



Workload

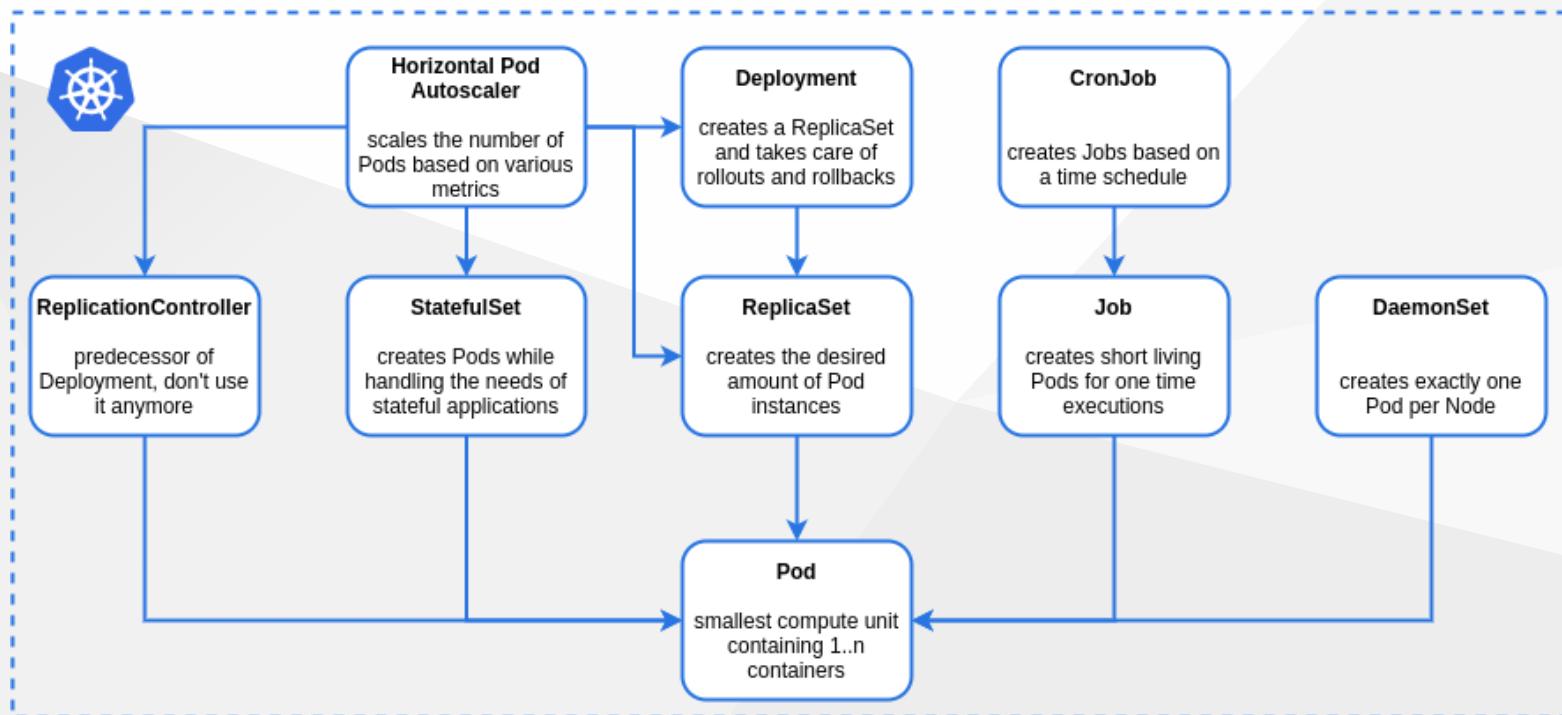
워크로드(Workload)는 Kubernetes에서 구동되는 애플리케이션을 말합니다.

애플리케이션을 컨테이너의 형태로 실행하기 위해서 Kubernetes에서는 Pod라는 Object를 이용합니다.

그리고, 이 Pod들의 집합을 관리하기 위해서 또 다른 Workload resource들을 사용합니다.

이번 장에서는 이 Workload에 대해 알아보겠습니다.

Kubernetes Workload Resources Overview

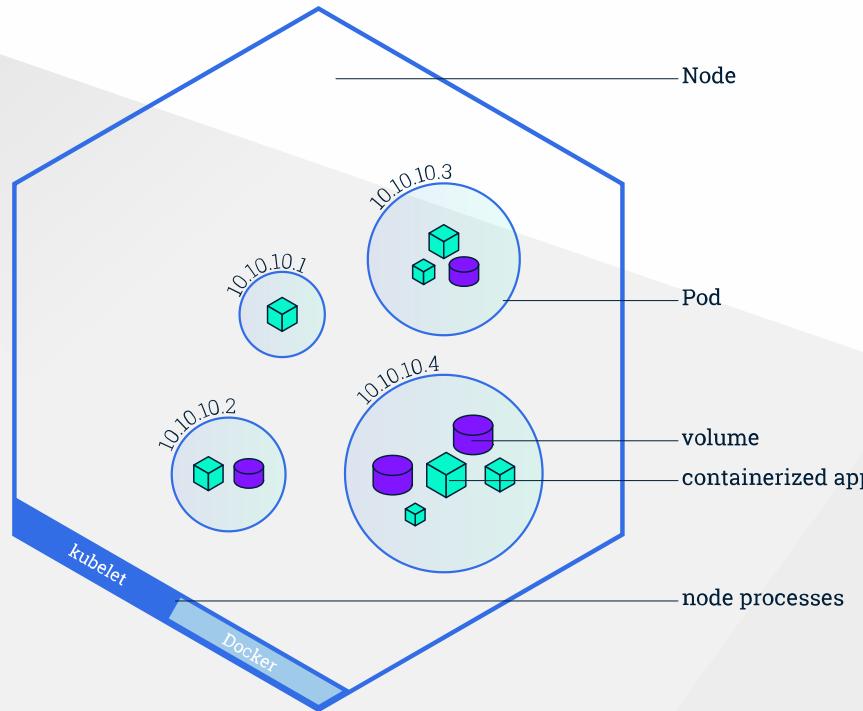


Pod

파드(Pod)는 Kubernetes에서 생성하고 관리할 수 있는 배포 가능한 **가장 작은 컴퓨팅 단위입니다.**
Pod는 하나 이상의 컨테이너 그룹으로 구성되며, **스토리지와 네트워크를 공유합니다.**

이 **Pod는 Node**에서 실행되는데, 이때 Node의 **Container runtime**을 이용하게 됩니다.

Docker는 대표적인 Kubernetes의 Container runtime이었지만, Kubernetes v1.200이후에는 deprecated 되었습니다. ([참조](#))
하지만, 앞서 배운 Docker환경에서 만들어진 컨테이너 이미지는 Kubernetes에서 문제없이 동작하니 걱정할 필요는 없습니다.



Pod

Pod을 구성하기 위한 Spec은 아래와 같이 작성할 수 있습니다.

내부에 포함될 Container의 image와 구성에 필요한 여러 정보(e.g. ports)를 포함하고 있습니다.

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

그리고, 아래와 같이 두 가지 유형의 Pod가 있습니다.

- **Pods that run a single container** : "one-container-per-Pod" 모델로, 가장 일반적인 유형
- **Pods that run multiple containers** : 밀접하게 결합되고 리소스를 공유하는 여러 개의 컨테이너로 구성

Pod는 결국 애플리케이션의 단일 인스턴스를 실행하기 위한 Kubernetes의 Object이며, 인스턴스를 확장(Pod의 개수를 증가)하기 위해서는 또 다른 **Workload resource**(아래)와 컨트롤러를 이용하게 됩니다. 이 부분은 뒤에 더 자세히 다루겠습니다.

- **Deployment**
- **StatefulSet**
- **DaemonSet**

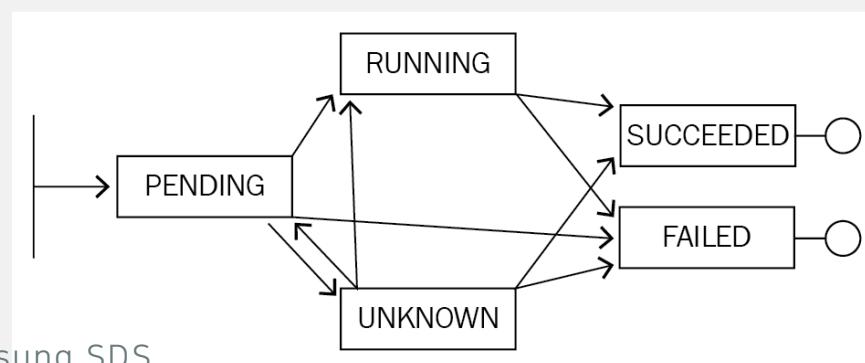
Pod lifecycle

파드(Pod)는 정의된 라이프사이클을 따릅니다. **Pending** 단계(Phase)에서 시작해서, 기본 컨테이너 중 적어도 하나 이상이 OK로 시작하면 **Running** 단계를 통과하고, 그런 다음 파드의 컨테이너가 어떤 상태로 종료되었는지에 따라 **Succeeded** 또는 **Failed** 단계로 이동합니다.

Pod phase

Pod의 Lifecycle에서의 단계(Phase)를 나타내는 고수준 요약

Value	Description
Pending	Pod가 Kuberneet cluster에서 승인되었지만, 컨테이너가 준비상태인 경우 (스케줄링이나 이미지 다운로드에 걸리는 시간을 포함함)
Running	Pod가 Node에 바인딩 되고 모든 컨테이너가 생성되어 실행 중
Succeeded	Pod의 모든 컨테이너가 성공적으로 종료
Failed	Pod의 모든 컨테이너가 종료되었고, 그 중 적어도 하나의 컨테이너가 실패로 종료
Unknown	Pod의 상태를 확인할 수 없는 단계로, 일반적으로 Node와의 통신오류로 인해 발생함



Pod conditions

Pod가 통과하거나 통과하지 못한 컨디션을 나타냄.

- **PodScheduled** : Pod가 Node에 스케줄되었다.
- **ContainersReady** : Pod의 모든 컨테이너가 준비되었다.
- **Initialized** : 모든 [초기화 컨테이너\(Init container\)](#) 가 성공적으로 완료(completed)되었다.
- **Ready** : Pod는 요청을 처리할 수 있으며 일치하는 모든 서비스의 로드 밸런싱 풀에 추가되어야 한다.

`kubectl describe pod` 명령어로 조회했을 때, 아래와 같이 각 Condition이 True/False로 표시됩니다.

Conditions:	
Type	Status
Initialized	True
Ready	True
ContainersReady	True
PodScheduled	True

Container states

Pod의 단계(Phase)뿐 아니라, Kubernetes는 Pod 내부 컨테이너의 상태(State)도 추적합니다. 컨테이너는 각 Node의 Container runtime에 의해 생성되며, 아래와 같은 상태(Status)를 가집니다.

Container states	Description
Waiting	컨테이너가 시작되기 전의 상태로, 이미지 pull이나 Secret의 적용과 같은 처리가 진행 중인 상태
Running	컨테이너가 문제없이 실행중인 상태
Terminated	컨테이너가 종료된 상태. (실패인 경우 포함.)

요약하자면 아래와 같습니다.

- Pod
 - **Phase** : Pending / Running / Succeeded / Failed / Unknown
 - **Condition** : PodScheduled / ContainersReady / Initialized / Ready
 - **Reason** : ContainersNotReady / PodCompleted
 - Containers:
 - Container #N
 - **State** : Waiting / Running / Terminated
 - **Reason** : ContainerCreating / CrashLoopBackOff / Error / Completed

Docker & Kubernetes - 08. Kubernetes workload(1)

일반적인 Pod가 생성되는 단계의 각 상태변화는 다음과 같습니다.

Pod phase	Pod condition	Container state	Description
Pending	PodScheduled : F ContainersReady : F Initialized : T Ready : F	-	Pod가 최초로 생성되었을 때
Pending	PodScheduled : T ContainersReady : F Initialized : T Ready : F	-	실행될 Node가 정해짐 (Kube-scheduler에 의해)
Pending	PodScheduled : T ContainersReady : F Initialized : T Ready : F	Waiting	Image를 Pull
Running	PodScheduled : T ContainersReady : T Initialized : T Ready : T	Running	컨테이너가 실행됨

Container probes

Kublet은 주기적으로 **Pod의 상태를 진단하게 되는데, 이때 사용되는것이 Probe입니다.**
아래와 같은 체크 메커니즘이 사용됩니다.

- **exec** : 컨테이너에서 지정된 명령어를 실행 (exits with 0 -> Successful)
- **httpget** : HTTP GET request (200이상 400미만 -> Successful)
- **tcpsocket** : 특정 포트에 대한 TCP 체크 수행 (Port가 Open됨 -> Successful)

Probe의 종류는 다음과 같은 것들이 있습니다.

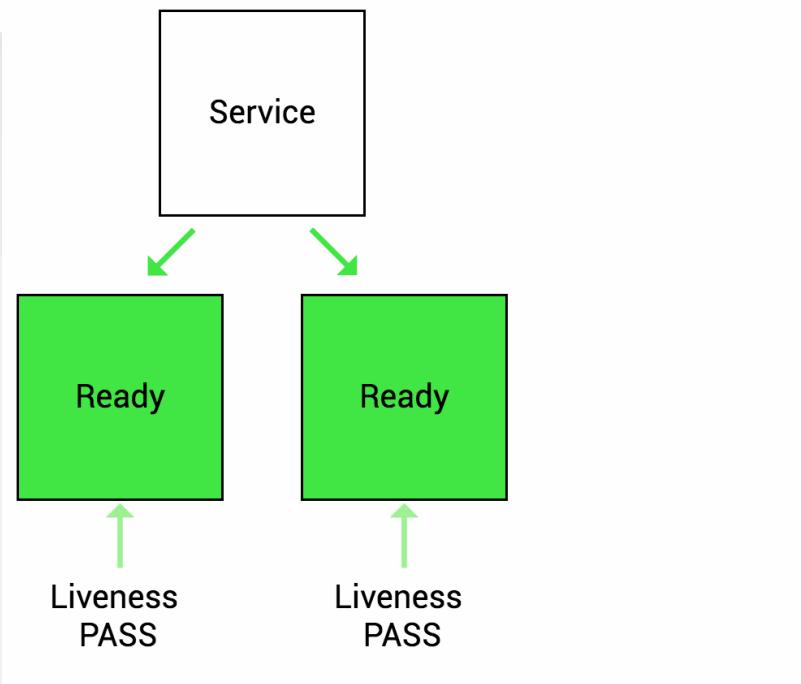
- **livenessProbe** : 컨테이너가 동작 중인지 여부를 나타냄. (실패한 경우 컨테이너를 재시작)
- **readinessProbe** : 컨테이너가 요청을 처리할 준비가 되었는지 여부를 나타냄.
- **startupProbe** : 컨테이너 내의 애플리케이션이 시작되었는지를 나타냄. 성공이후 다른 Probe가 활성화됨. (실패한 경우 컨테이너를 재시작)

Container probes - livenessProbe

livenessProbe는 애플리케이션의 동작상태를 체크합니다.

애플리케이션에 교착 상태(deadlock)가 발생하여 앱이 무기한 중단되고 요청(Request) 처리가 중단되는 시나리오를 상상해 보겠습니다. 프로세스는 계속 실행중이기 때문에 기본적으로 Kubernetes는 모든 것이 정상이라고 생각하고 계속해서 손상된 Pod에 요청을 보냅니다.

livenessProbe를 사용하면 이런경우 애플리케이션이 더 이상 Request를 정상적으로 처리하지 않음을 감지하고 문제가 되는 Pod의 컨테이너를 재시작합니다. (Pod의 Restart 카운트가 증가함.)

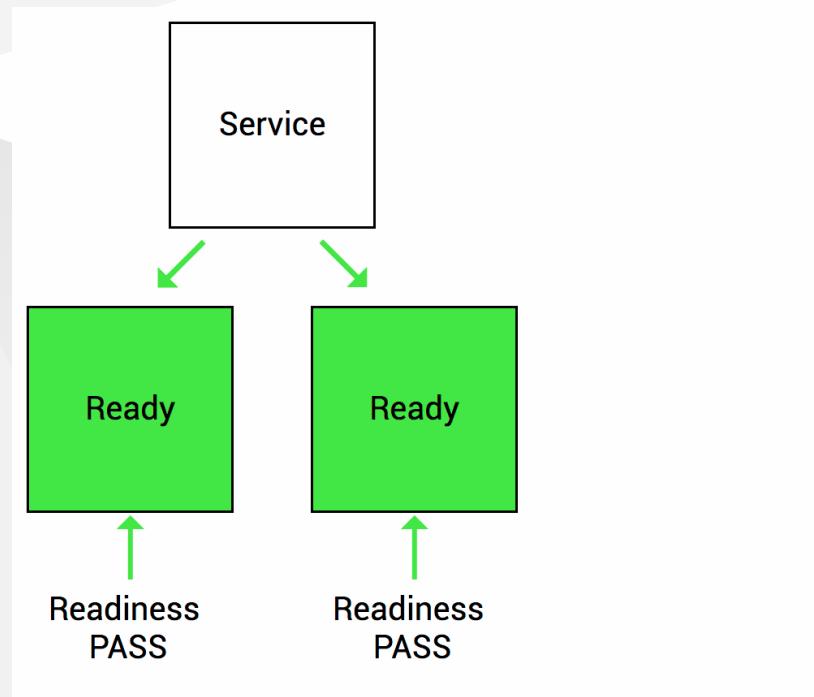


Container probes - readinessProbe

readinessProbe는 애플리케이션이 요청을 처리할 준비가 되었는지를 체크합니다.

예를 들어 애플리케이션이 시작되고 정상적으로 서비스되기까지 얼마정도의 시간이 걸린다고 가정해보겠습니다. 이러한 상황(준비가 완료되지 않은 상황)에서는 트래픽이 이 컨테이너로 전달되면 문제가 될 수 있습니다.

readinessProbe를 사용하면 애플리케이션이 완전히 시작될 때까지 기다렸다가 트래픽을 보낼 수 있습니다.



Container probes

몇 가지 Container probe의 사용 예시를 보겠습니다.

- Define a liveness HTTP request

httpGet 유형의 **livenessProbe** 예제입니다.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/liveness
      args:
        - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
          - name: Custom-Header
            value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

livenessProbe 는 **initialDelaySeconds**(3초) 후부터 **periodSeconds**(3초) 간격으로 /healthz 를 httpGet 요청을 보냄.

Container probes

- Define a TCP liveness probe

tcpSocket 유형의 readinessProbe와 livenessProbe 예제입니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
    - name: goproxy
      image: k8s.gcr.io/goproxy:0.1
      ports:
        - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 20
```

컨테이너의 8080번 포트의 상태를 이용하여 준비상태(readinessProbe)와 동작상태(livenessProbe)를 검사

Container probes

Configure Probes

정확한 Probe 설정을 위해서 다음 필드들을 사용할 수 있습니다.

- `initialDelaySeconds`: Container가 시작된 후 probe가 수행되기 전까지의 Delay (Default : 0 , Minimum : 0)
- `periodSeconds`: probe의 수행빈도 (Default : 10 , Minimum : 1)
- `timeoutSeconds`: probe 수행응답을 기다리는 timeout 시간을 설정 (Default : 1 , Minimum : 1)
- `successThreshold`: probe가 실패한 후 성공으로 간주되기 위한 최소 연속 성공 횟수 (Default : 1 , Minimum : 1)
- `failureThreshold`: probe가 실패로 판단하기 위한 실패 횟수 (Default : 3 , Minimum : 1)

HTTP probes

HTTP probe는 추가적으로 아래와 같은 필드를 더 설정할 수 있습니다.

- `host`: 연결하려는 Host Name (Default : pod IP)
- `scheme`: HTTP or HTTPS (Default : HTTP)
- `path`: HTTP server에 접근하려는 경로 (Default : /)
- `httpHeaders`: Custom header 설정 값
- `port`: container에 접근하려는 Port

Resource 할당

Kubernetes에서 Pod는 스케줄링할때 Container가 실제로 수행 할 수 있는 충분한 컴퓨팅 리소스를 갖는 것은 매우 중요합니다. 만약 어떤 애플리케이션이 수행되기 위한 리소스보다 적은 리소스로 Node에 할당되게 되면, 메모리나 CPU 문제로 애플리케이션이 중단될 수 있기 때문입니다. 또한 애플리케이션은 때때로 필요한 것보다 더 많은 리소스가 필요 할 수도 있습니다.

Resource requests and limits

Resource requests and limit은 Kubernetes가 CPU 및 메모리와 같은 리소스를 제어하는 매커니즘입니다.

Container를 명세할때 아래와 같이 각각의 Container에서 필요한 CPU와 RAM을 CPU는 Milicore 단위(1,000 Milicore = 1Core)로 Memory는 Mbyte 단위로 리소스의 requests과 limits(요청과 제한)이라는 측면에서 정의할 수 있습니다. Container가 리소스를 Request하면 Kubernetes는 리소스를 제공할 수 있는 노드에서만 스케줄링이 되고, Limit는 Container가 특정 리소스를 초과하지 않도록 제한합니다.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

cpu

CPU 요청과 관련하여 고려해야 할 사항 중 한가지는 Node의 CPU core 수보다 큰 값을 입력하면 Pod가 스케줄링 되지 않는다는 것입니다.

특별한 경우가 아니라면 CPU 요청은 1이하로 하고 Replica에 의해 오토스케일링되는 것을 염두해서 Pod를 설계해야합니다. 이렇게 설정하는 것이 시스템을 보다 유연하고 신뢰성있게 구성할 수 있습니다.

Memory

Memory도 CPU와 마찬가지로 Node의 메모리보다 더 큰 요청을 셋팅하게되면 Pod는 스케줄링되지 않습니다. 또한 메모리는 CPU와 달리 Kubernetes에서 메모리 사용량을 조절할 수 없으므로 Container가 메모리 제한을 초과하게 되면 애플리케이션(Container)가 종료되는데, Pod가 Deployment, StatefulSet, DaemonSet 또는 다른 유형의 컨트롤러에 의해 관리되는 경우, 기존 Pod는 종료되고 새로운 Pod가 시작됩니다.

Namespace settings

클러스터는 Namespace로 구성될 수 있는데, 만약 그 중 한 Namespace가 과도한 요청이 있을 경우 전체 클러스터에 영향을 미칠 수 있습니다. 따라서 이러한 경우를 제한하기 위해서는 **Namespace 레벨에서 ResourceQuotas와 LimitRanges**를 설정하는 것이 중요합니다.

ResourceQuotas

Namespace를 생성한 후, ResourceQuota를 통해 Namespace의 CPU 및 메모리를 제한 할 수 있습니다.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo
spec:
  hard:
    requests.cpu: 500m
    requests.memory: 100Mib
    limits.cpu: 700m
    limits.memory: 500Mib
```

LimitRange

ResourceQuotas는 Namespace 전체영역에 대한 리소스의 제한을 정의하는반면, LimitRange는 개별 컨테이너 단위의 리소스에 대한 제약입니다. 즉, 사용자들이 개별 컨테이너에 대한 리소스를 정의 할때 해당되는 범위를 제한하는 개념입니다. 이렇게 함으로써, 사용자들은 초소형 또는 초대형 컨테이너를 생성 할 수 없게 됩니다.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default: # this section defines default limits
    cpu: 500m
  defaultRequest: # this section defines default requests
    cpu: 500m
  max: # max and min define the limit range
    cpu: "1"
  min:
    cpu: 100m
```

▶ Hands-on : 08_Kubernetes_Workload(1)

Summary

- Workload
 - Pod
 - Pod lifecycle
 - Pod phase
 - Container state
 - Container probe
 - livenessProbe
 - readinessProbe
 - startupProbe
 - Resource 할당
 - Namespace settings