

KubernetesPractice

Table of Contents:

- **Controller** (Master): API Server, Kube-Controller, Cloud-Controller, etcd (Database)
- **Node** (Worker): kubelet, kube-proxy, container manager
- **Pods** (Cluster)
- **Namespace** (Grouping)
- **Label**
- **Annotation** (Label with bigger size)
- **Replication Controller**
- **Replica Sets**
- **Daemon Sets**
- **Job**

1. What is Kubernetes?

Kubernetes is used for automation deployment and managing applications container-based.

2. How to install Kubernetes on Local?

- Use **Docker Desktop** > **Settings** > **Enable Kubernetes**

⚠️ Dont forget to **Reset Cluster** after installation, if Kubernetes still not working, please restart your **Docker Desktop**

3. Kubernetes Node

- Node could be VM or real PC
- Inside Node: **kubelet**, **kube-proxy**, and **container manager**
- Node is a worker node

To check all nodes:

```
kubectl get node
```

To check node detail:

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

4. Pod

- **Pod** is a smallest unit that we can deploy to Kubernetes Cluster
- Why not call it **Container**? because one **Pod** could have one more **Container**

- **Pod** is the application that we are running in Kubernetes Cluster
- **Pod** should running inside the **Node**
- A **Node** could running more than one **Pod**
- **Pod** could not run in multiple **Nodes** at once
- **Pod** could run more than one **Container** provider, e.g: Docker and Containerd

To check **Pod**:

```
kubectl get pod
```

To check pod detail:

```
kubectl describe pod <pod-name>
```

4.1. Create Pod

Configure the **yaml**:

```
apiVersion: v1
kind: Pod
metadata:
  name: <pod-name>
spec:
  containers:
    - name: <container-name>
      image: <image-name>
      ports:
        - containerPort: <port>
    - name: <container-name>
      image: <image-name>
      ports:
        - containerPort: <port>
```

Create the **Pod**:

```
kubectl create -f nginx.yaml
```

Check the pod:

```
kubectl get pod
kubectl get pod -o wide
kubectl describe pod <pod-name>
```

4.2. Pod Port-Forwarding

```
kubectl port-forward <pod-name> <port-access>:<pod-port>
kubectl port-forward nginx 8080:80
```

4.3. Delete Pod

To delete pod:

```
kubectl delete pod <pod-name>
kubectl delete pod <pod-name1> <pod-name2> <pod-name3>
```

To delete pod with label:

```
kubectl delete pod -l key=value
```

To delete all pods in a namespace:

```
kubectl delete pod --all --namespace <namespace>
```

5. Label

- **Label** is used to add additional information to **Pod**
- **Label** is not only for **Pod**
- **White Space** is not allowed in Label

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
    author: ashryramadhan
    environment: production
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "128Mi"
        cpu: "250m"
```

```
ports:
  - containerPort: 80
```

To show labels:

```
kubectl get pods --show-labels
```

To search by label:

```
kubectl get pods -l key
kubectl get pods -l key=value
kubectl get pods -l "!key"
kubectl get pods -l key!=value
kubectl get pods -l "key in (value1,value2)"
kubectl get pods -l "key notin (value1,value2)"
kubectl get pods -l "key1=value1,key2=value2"
```

6. Annotations

- **Annotations** works similar to **Label** but it can't be filtered
- It has been used for adding information in a big size
- It can save information up to 256kb

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
    author: ashryramadhan
    environment: production
  annotations:
    annotation-key1: annotation-value
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "128Mi"
          cpu: "250m"
      ports:
        - containerPort: 80
```

7. Namespace

- Namespace is like grouping for administration
- We need to separate our resources for multi-tenant, team, or environment
- Resources name could be same in other namespaces
- Pods with same name is allowed to run on the same machine but must in different namespace
- Namespace is not same as isolating our resources
- Even Pods with the same pod-name in different namespace, it still can communicate each other

To look namespaces:

```
kubectl get namespaces
kubectl get namespace
kubectl get ns
```

To check all **Pods** in particular **namespace**:

```
kubectl get pods --namespace <namespace>
```

7.1. Creating Namespace

To create **namespace**:

```
# namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: nama-namespace
```

To create it:

```
kubectl create -f namespace.yaml
```

7.2. Add Pods to Namespace

To create **Pods** inside a namespace:

```
kubectl create -f pods.yaml --namespace <namespace>
```

7.3. Delete Namespace

To delete namespace:

```
kubectl delete namespace <namespace>
```

8. Probe

✨ It is a standard for our apps to have endpoint such `http://localhost/health` if we are using container technology

In Kubernetes, a **probe** is a diagnostic mechanism used to determine the health and status of a container. Probes are used to check if a container is running as expected, and based on the results, Kubernetes can take actions like restarting the container or routing traffic away from it.

Liveness, Readiness, Startup Probe:

- kubelet use liveness probe to check the liveness of the containers
- Kubelet use readiness probe to check the Pod if it's ready with the traffic
- Kubelet use startup probe to check the container has started correctly
- Startup Pod is suited for Pod that needs to have a long startup, this will make sure the pod is not stopped by kubelet before it will run smoothly

To check probe:

```
kubectl get pod  
kubectl describe pod pod-name
```

Mecahnism of Code Checking:

- HTTP Get
- TCP Socket
- Command Exec

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: pod-name  
  labels:  
    label-key1: label-value1  
  annotations:  
    annotation-key1: annotation-value  
    annotation-key2: veri long annotation value, bla bla bla bla bla bla  
spec:  
  containers:  
    - name: container-name  
      image: image-name  
      ports:  
        - containerPort: 80  
      livenessProbe:  
        httpGet:
```

```
    path: /health
    port: 80
    initialDelaySeconds: 0
    periodSeconds: 10
    timeoutSeconds: 1
    successThreshold: 1
    failureThreshold: 3
  readinessProbe:
    httpGet:
      path: /ready
      port: 80
    initialDelaySeconds: 0
    periodSeconds: 10
    timeoutSeconds: 1
    successThreshold: 1
    failureThreshold: 3
  startupProbe:
    httpGet:
      path: /startup
      port: 80
    initialDelaySeconds: 0
    periodSeconds: 10
    timeoutSeconds: 1
    successThreshold: 1
    failureThreshold: 3
```

1. `initialDelaySeconds: 0`

- **Description:** This is the number of seconds after the container has started before the first probe is initiated.
- **Purpose:** It allows the application time to start before Kubernetes begins performing health checks.
- **Example:** If set to `0`, the probe starts immediately when the container starts. If it's set to `30`, the probe will wait 30 seconds after the container starts before performing the first check.

2. `periodSeconds: 10`

- **Description:** This is the interval (in seconds) between each probe.
- **Purpose:** It defines how frequently the health checks are performed.
- **Example:** If set to `10`, Kubernetes will perform a health check every 10 seconds.

3. `timeoutSeconds: 1`

- **Description:** This is the maximum number of seconds that Kubernetes waits for a probe to complete.
- **Purpose:** It sets a time limit for the probe response.
- **Example:** If set to `1`, Kubernetes will wait for up to 1 second for the probe to complete. If the probe doesn't respond within this time, it's considered a failure.

4. `successThreshold: 1`

- **Description:** This is the number of consecutive successful probes required for the container to be considered healthy after having failed.
- **Purpose:** It defines the number of successful checks needed to reset the failure counter.
- **Example:** If set to **1**, a single successful probe will mark the container as healthy. If it's set to **3**, three consecutive successful probes are needed.

5. **failureThreshold: 3**

- **Description:** This is the number of consecutive failed probes required for the container to be considered unhealthy.
- **Purpose:** It defines how tolerant the system is to probe failures before taking action (e.g., restarting the container).
- **Example:** If set to **3**, Kubernetes will consider the container unhealthy if it fails the probe three times in a row.

Example Usage

Here is a more detailed example of how these parameters might be used in a readiness probe for a pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: myapp-container
    image: myapp:latest
    readinessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 0
      periodSeconds: 10
      timeoutSeconds: 1
      successThreshold: 1
      failureThreshold: 3
```

Explanation of the Example

- **initialDelaySeconds: 0:** The readiness probe starts immediately after the container starts.
- **periodSeconds: 10:** The readiness probe runs every 10 seconds.
- **timeoutSeconds: 1:** Each probe has a 1-second timeout period. If the **/health** endpoint does not respond within 1 second, the probe fails.
- **successThreshold: 1:** The container is considered ready after a single successful probe.
- **failureThreshold: 3:** The container is considered not ready after three consecutive probe failures.

Why These Parameters Are Important

- **Reliability:** By configuring these parameters appropriately, you can ensure that your application is given enough time to start and that Kubernetes is checking its health at reasonable intervals.
- **Performance:** Probes should be frequent enough to detect issues promptly but not so frequent that they create unnecessary load on the system.
- **Fault Tolerance:** The failure and success thresholds allow Kubernetes to distinguish between transient issues and real problems, ensuring that containers are only restarted when necessary.

Healthy (Liveness Probe)

Healthy means that the application is running correctly and is not stuck or in a state that requires restarting. The liveness probe checks the ongoing health of the application.

Example: A simple health check might verify that the application process is running and can respond to requests.

Code Example:

```
@app.route('/health', methods=['GET'])
def health():
    # Perform checks to determine if the application is healthy
    # For example, check if the database connection is alive
    try:
        # Simulate a check, e.g., database connection
        db_connection = True # Replace with actual health check logic
        if db_connection:
            return jsonify(status='healthy'), 200
        else:
            return jsonify(status='unhealthy'), 500
    except Exception as e:
        return jsonify(status='unhealthy', error=str(e)), 500
```

Ready (Readiness Probe)

Ready means that the application is ready to handle requests. The readiness probe checks whether the application is fully initialized and ready to serve traffic. If the readiness probe fails, the application will not receive any traffic from the service.

Example: A readiness check might verify that all necessary components (like external services or databases) are available and the application is fully initialized.

Code Example:

```
@app.route('/ready', methods=['GET'])
def ready():
    # Perform checks to determine if the application is ready to serve traffic
    try:
        # Simulate a check, e.g., database availability, service connectivity
        service_ready = True # Replace with actual readiness check logic
        if service_ready:
```

```
        return jsonify(status='ready'), 200
    else:
        return jsonify(status='not ready'), 500
except Exception as e:
    return jsonify(status='not ready', error=str(e)), 500
```

Startup (Startup Probe)

Startup means that the application has completed its startup routine and is ready to be checked by liveness and readiness probes. The startup probe checks that the application has started up correctly.

Example: A startup check might verify that all initial setup tasks (like schema migrations, configuration loading, etc.) are completed.

Code Example:

```
@app.route('/startup', methods=['GET'])
def startup():
    # Perform checks to determine if the application has started up correctly
    try:
        # Simulate a startup check, e.g., ensure all initializations are complete
        startup_complete = True # Replace with actual startup check logic
        if startup_complete:
            return jsonify(status='started up'), 200
        else:
            return jsonify(status='not started up'), 500
    except Exception as e:
        return jsonify(status='not started up', error=str(e)), 500
```

In summary, these parameters provide fine-grained control over how Kubernetes monitors and manages the health of your containers, helping to maintain the stability and reliability of your applications.

9. Replication Controller


- **Replication Controller** will make sure our Pods will be always running
- If one of our Pod is stopped or dissapeared, then **Replication Controller** automatically will run the Pod
- **Replication Controller** is purposed to manage more than one Pod
- **Replication Controller** will make sure the number of Running Pods is correct, if the number of running Pods is not match, then **Replication Controller** will add or subtract more Pods.

To check replicationcontrollers:

```
kubectl get replicationcontrollers
kubectl get replocationcontroller
kubectl get rc
```

9.1. Create Replication Controller

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: mynginx
  template:
    metadata:
      name: nginx # pod-name
      labels:
        app: mynginx
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            limits:
              memory: "128Mi"
              cpu: "250m"
          ports:
            - containerPort: 80
```

 Note: if you try to delete one of the pods, the **Replication Controller** will create it again in another node. You can try with `kubectl delete pod <pod-name>` command, then check again using `kubectl get pods`

9.2. Delete Replication Controller

To delete RC:

```
kubectl delete rc <rc-name>
```

To delete RC without deleting the Pods within:

```
kubectl delete rc <rc-name> --cascade=false
```

10. Replica Set

- **Replica Set** is a new version **Replication Controller**
- **Replica Set** has label selector feature which more expressive rather than **Replication Controller**

10.1. Create Replica Set

To check **Replication Set**:

```
kubectl get rs
```

Template:

```
apiVersion: apps/v1
kind: ReplicaSet # -> this also
...
spec:
  replicas: 3
  selector:
    matchLabels: # -> this the new feature
      label-key1: label-value1
  template:
    metadata:
      name: pod-name
      labels:
        label-key1: label-value1
    ...
```

10.2. Delete Replica Set

To delete Replica Set:

```
kubectl delete rs <rs-name>
```

10.3. Label Selector Match Expression Replica Set

- **matchLabels**, key=value pair must exact
- **matchExpression**, In, NotIn, Exist, NotExist

```
apiVersion: apps/v1
kind: ReplicaSet
...
spec:
  replicas: 3
  selector:
    matchLabels:
      label-key1: label-value1
    matchExpressions: # -> matchExpression
      - key: label-key # Key
        operator: In # Operator
        values:
          - label-value1
          - label-value2
```

```
template:
  metadata:
    name: pod-name
    labels:
      label-key1: label-value1
  ...
```

11. Daemon Set

- When we use **Replica Set**, **Pod** will be running in random node, it's set by kubernetes.
- If we want to run our **Pods** in every node, and **1 Pod** should only allowed to run in **1 Node**, we can use Daemon Set
- By default, Daemon Set will run our **Pods** on every nodes in our Kubernetes Cluster.

Use case of **1 Pod 1 Node**:

- Application for monitoring Node
- Application for get logs in Node
- etc.

To check daemon set:

```
kubectl get daemonsets
kubectl get ds
```

To delete daemon set:

```
kubectl delete daemonsets <daemonsets-name>
kubectl delete ds <daemonsets-name>
```

To check details daemonsets:

```
kubectl describe daemonsets <daemonsets-name>
kubectl describe ds <daemonsets-name>
```

Template:

```
apiVersion: apps/v1
kind: DaemonSet # -> this new
metadata:
  name: daemon-set-name
  labels:
    label-key1: label-value1
  annotations:
```

```
  annotation-key1: annotation-value1
spec:
  selector:
    matchLabels:
      label-key1: label-value1
    matchExpressions:
      - key: label-key1
        operator: In
        values:
          - label-value1
  template:
    metadata:
      name: pod-name
      labels:
        label-key1: label-value1
    spec:
      containers:
        - name: container-name
          image: image-name
          ports:
            - containerPort: 80
          readinessProbe:
            httpGet:
              path: /health
              port: 80
            initialDelaySeconds: 0
            periodSeconds: 10
            failureThreshold: 3
            successThreshold: 1
            timeoutSeconds: 1
```

12. Job

- **Job** is a **Pod** that will only run once then stop
- **Job** will be terminated after the task is done

Use Case:

- Backup and Restore Database
- Import and Export Data
- Process Batch
- etc.

To create Job:

```
kubectl create -f job.yaml
```

To check jobs:

```
kubectl get jobs
```

To delete Job:

```
kubectl delete job <job-name>
```

Template: