

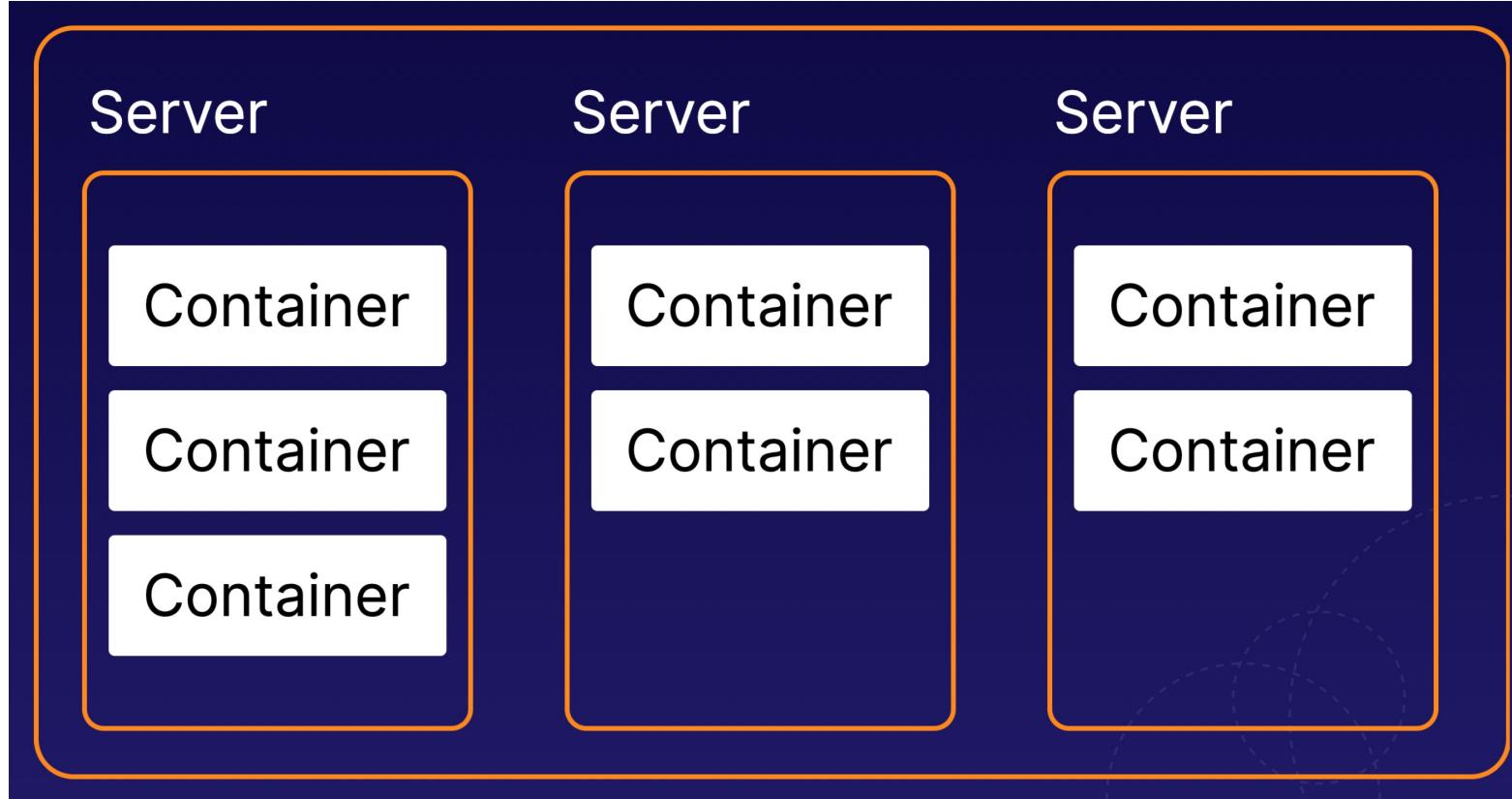


kubernetes

What Does “K8s” Mean?

K8s is simply short for Kubernetes. The 8 represents the 8 letters between K and S!

Introduction



K8s Cluster

K8s Features

Container Orchestration

The primary purpose of Kubernetes is to dynamically manage containers across multiple host systems.

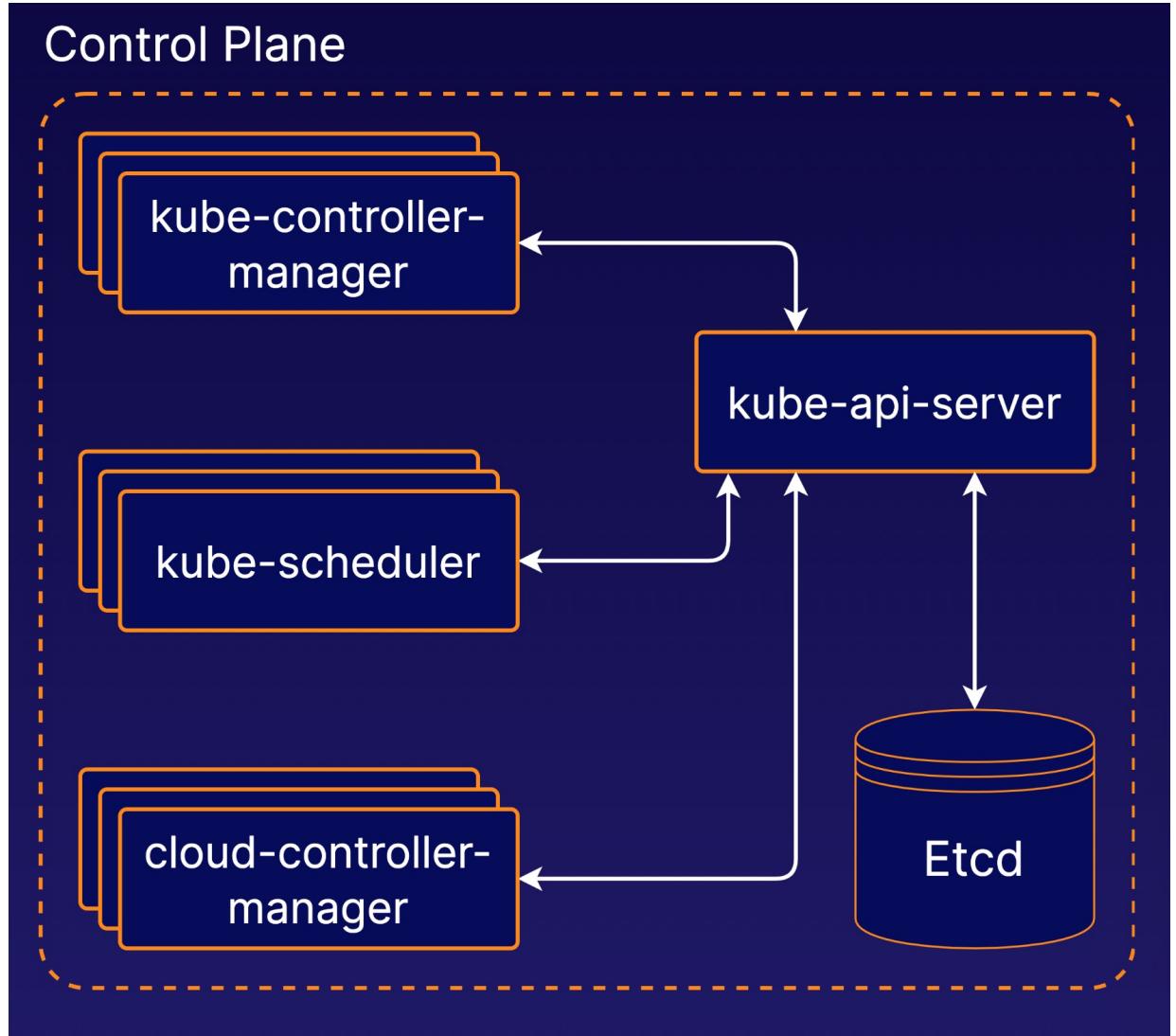
Application Reliability

Kubernetes makes it easier to build reliable, self-healing, and scalable applications.

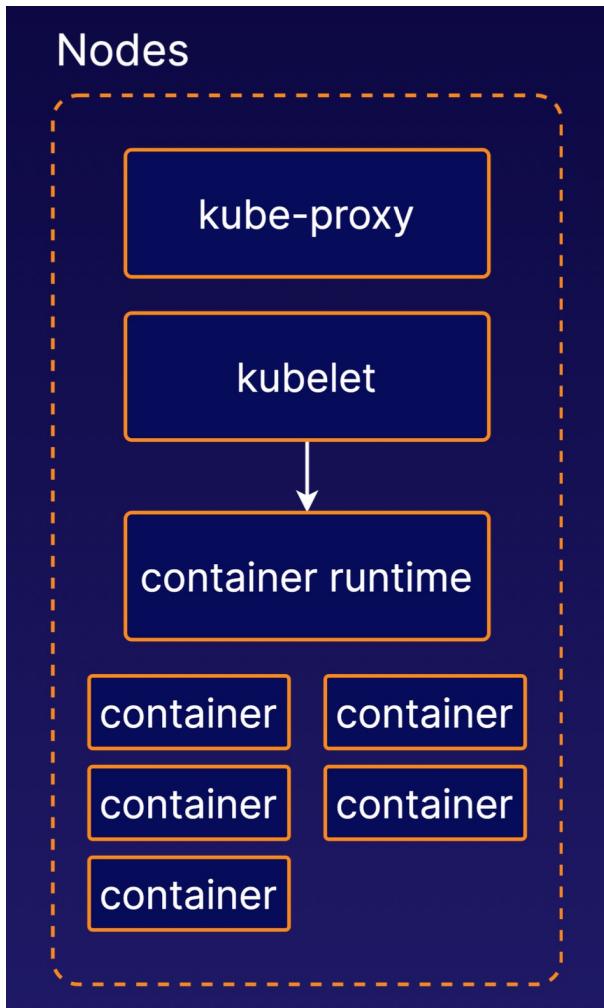
Automation

Kubernetes offers a variety of features to help automate the management of your container apps.

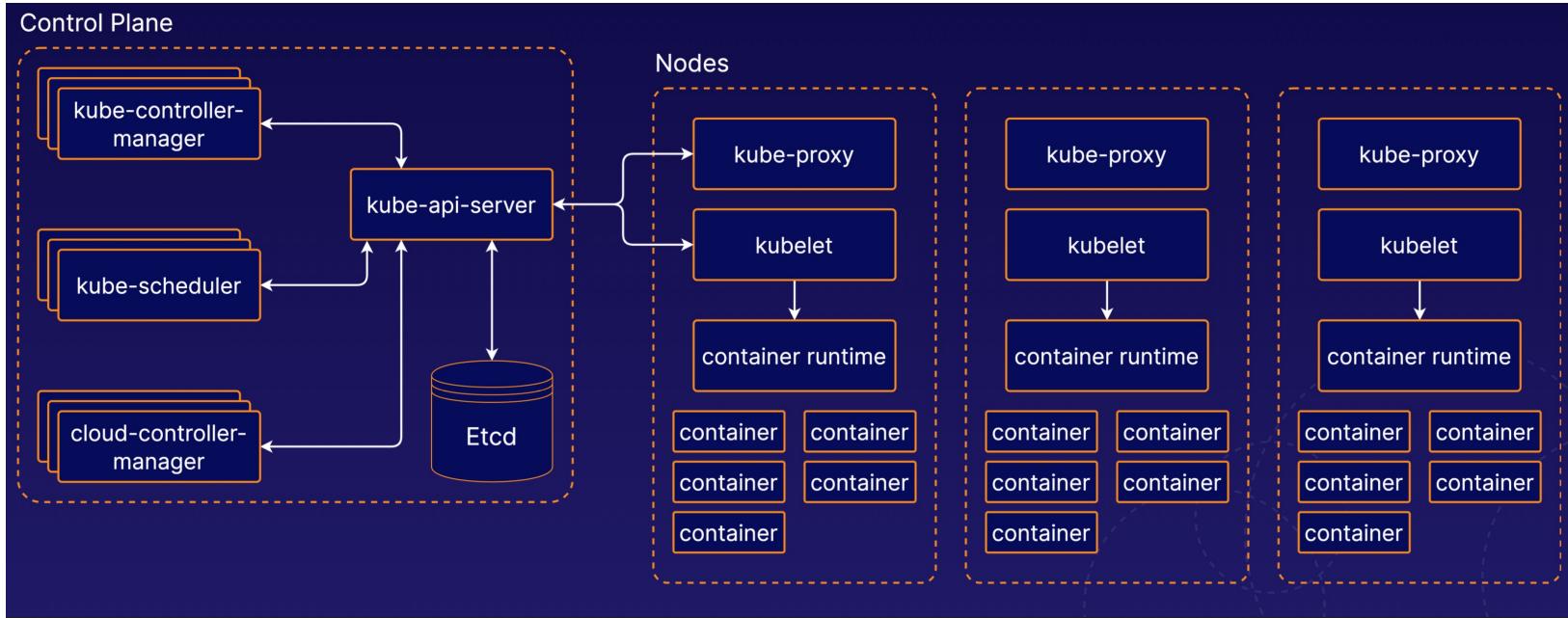
K8s Architecture



K8s Architecture



K8s Architecture



Workshop 1

```
> k3d cluster create mycluster \
--servers 1 \
--agents 3 \
--image rancher/k3s:latest
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-mycluster'
INFO[0000] Created image volume k3d-mycluster-images
INFO[0000] Starting new tools node...
INFO[0001] Creating node 'k3d-mycluster-server-0'
INFO[0001] Pulling image 'ghcr.io/k3d-io/k3d-tools:5.6.0'
INFO[0004] Starting Node 'k3d-mycluster-tools'
INFO[0004] Pulling image 'rancher/k3s:latest'
INFO[0011] Creating node 'k3d-mycluster-agent-0'
INFO[0011] Creating node 'k3d-mycluster-agent-1'
INFO[0011] Creating node 'k3d-mycluster-agent-2'
INFO[0011] Creating LoadBalancer 'k3d-mycluster-serverlb'
INFO[0013] Pulling image 'ghcr.io/k3d-io/k3d-proxy:5.6.0'
INFO[0018] Using the k3d-tools node to gather environment information
INFO[0018] Starting new tools node...
INFO[0018] Starting Node 'k3d-mycluster-tools'
INFO[0019] Starting cluster 'mycluster'
INFO[0019] Starting servers...
INFO[0019] Starting Node 'k3d-mycluster-server-0'
INFO[0023] Starting agents...
INFO[0023] Starting Node 'k3d-mycluster-agent-0'
INFO[0023] Starting Node 'k3d-mycluster-agent-1'
INFO[0023] Starting Node 'k3d-mycluster-agent-2'
INFO[0027] Starting helpers...
INFO[0027] Starting Node 'k3d-mycluster-serverlb'
INFO[0033] Injecting records for hostAliases (incl. host.k3d.internal) and for 6 network members into CoreDNS configmap...
INFO[0035] Cluster 'mycluster' created successfully!
INFO[0035] You can now use it like this:
kubectl cluster-info
> kubectl get no
NAME                  STATUS    ROLES          AGE      VERSION
k3d-mycluster-server-0  Ready     control-plane,master  9m54s   v1.25.16+k3s4
k3d-mycluster-agent-0   Ready     <none>        9m52s   v1.25.16+k3s4
k3d-mycluster-agent-2   Ready     <none>        9m50s   v1.25.16+k3s4
k3d-mycluster-agent-1   Ready     <none>        9m50s   v1.25.16+k3s4
```

YAML

```
name: Courses
list:
- name: Go for Beginner
  price: 600
- name: Redis Fundamental
  price: 300
- name: RxJS for Beginner
  price: 500
```

JSON

```
{
  "name": "Courses",
  "list": [
    {
      "name": "Go for Beginner",
      "price": 600
    },
    {
      "name": "Redis Fundamental",
      "price": 300
    },
    {
      "name": "RxJS for Beginner",
      "price": 500
    }
  ]
}
```

YAML 101

Reference: <https://yaml.org>

YAML 101

- Simple variables (name = “Kongsak Limpitikeat”)

```
# Simple variables (name = "Kongsak Limpitikeat")
name: Kongsak Limpitikeat
video_ideas: hundreds
free_time: 0
```

- List

```
# List (books = ["Cryptonomicon", "Snow Crash" ... ])
books:
  - Cryptonomicon
  - Snow Crash
  - The Design and Implementation of the FreeBSD Operating System
  - The Practice of Cloud System Administration
```

- Dictionary

```
# Dictionary (languages = {"python": "excellent", "ruby": "good",
... })
languages:
  python: excellent
  ruby: good
  clojure: bad
  assembly: wannabe
```

- Multi-line string: with newlines

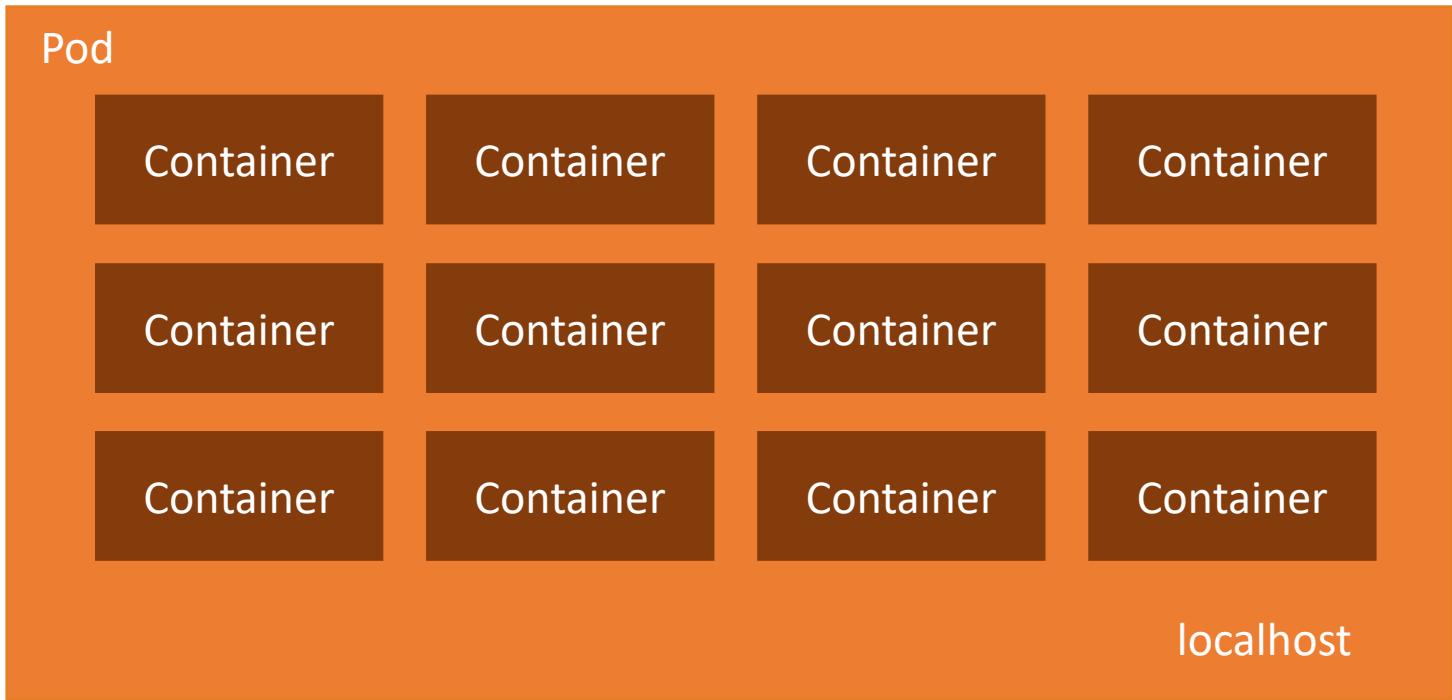
```
# Multi-line string: with newlines
hobbies: |
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
Output: "4 GECSEs\n3 A-Levels\nBsc in the Internet of
Things\n"
```

- Multi-line string: one line

```
# Multi-line string: with newlines
hobbies: >
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
Output: "4 GECSEs 3 A-Levels Bsc in the Internet of Things\n"
```

YAML 101

- Pods (po) : a group of one or more containers



POD

Workshop 2

Pod and Namespaces

```
00-namespace.yml ×  
1 apiVersion: v1  
2 kind: Namespace  
3 metadata:  
4   name: workshop-02  
5   labels:  
6     name: workshop-02  
7     module: workshop-02  
8
```

```
01-one-container-pod.yml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   name: one-container-pod  
5   namespace: workshop-02  
6 spec:  
7   containers:  
8     - name: curl001  
9       image: curlimages/curl  
10      command: ['sh', '-c', 'while true; do sleep 5; done']  
11
```

```
02-multi-container-pod.yml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   name: multi-container-pod  
5   namespace: workshop-02  
6 spec:  
7   containers:  
8     - name: curl002  
9       image: curlimages/curl  
10      command: ['sh', '-c', 'while true; do sleep 5; done']  
11     - name: curl003  
12       image: curlimages/curl  
13      command: ['sh', '-c', 'while true; do sleep 5; done']  
14     - name: curl004  
15       image: curlimages/curl  
16      command: ['sh', '-c', 'while true; do sleep 5; done']  
17
```

What Is kubectl ?

kubectl is a command line tool that allows you to interact with Kubernetes. kubectl uses the Kubernetes API to communicate with the cluster and carry out your commands.

“You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs.”

Use `kubectl get` to list objects in the Kubernetes cluster.

- `-o` — Set output format.
- `--sort-by` — Sort output using a JSONPath expression.
- `--selector` — Filter results by label.

```
$ kubectl get <object type> <object name> -o  
<output> --sort-by <JSONPath> --selector  
<selector>
```

`kubectl get`

You can get detailed information about
Kubernetes objects using `kubectl describe`.

```
$ kubectl describe <object type> <object name>
```

`kubectl describe`

Use `kubectl create` to create objects.

Supply a YAML file with `-f` to create an object from a YAML descriptor stored in the file.

If you attempt to create an object that already exists, an error will occur.

```
$ kubectl create -f <file name>
```

`kubectl create`

`kubectl apply` is similar to `kubectl create`. However, if you use `kubectl apply` on an object that already exists, it will modify the existing object, if possible.

```
$ kubectl apply -f <file name>
```

`kubectl apply`

Use `kubectl delete` to delete objects from the cluster.

```
$ kubectl delete <object type> <object name>
```

`kubectl delete`

`kubectl exec` can be used to run commands inside containers. Keep in mind that, in order for a command to succeed, the necessary software must exist within the container to run it.

For pods with multiple containers, specify the container name with `-c`.

```
$ kubectl exec <pod name> -c <container name> --<command>
```

`kubectl exec`

Workshop 3

kubectl

```
 1  apiVersion: v1
 2  kind: Namespace
 3  metadata:
 4    name: workshop-03
 5    labels:
 6      name: workshop-03
 7      module: workshop-03
 8  ---
 9  apiVersion: v1
10  kind: Pod
11  metadata:
12    name: echo-pod
13    namespace: workshop-03
14    labels:
15      app: echo-pod-app
16  spec:
17    containers:
18      - name: echo
19        image: ealen/echo-server:latest
20      - name: netshoot
21        image: nicolaka/netshoot
22        command: ['bash', '-c', 'while true; do sleep 5; done']
23        securityContext:
24          privileged: true
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  key1: value1
  key2: value2
  key3:
    subkey:
      morekeys: data
      evenmore: some more data
  key4: |
    You can also do
    multi-line
    data.
```

You can store configuration data in Kubernetes using **ConfigMaps**. ConfigMaps store data in the form of a key-value map. ConfigMap data can be passed to your container applications.

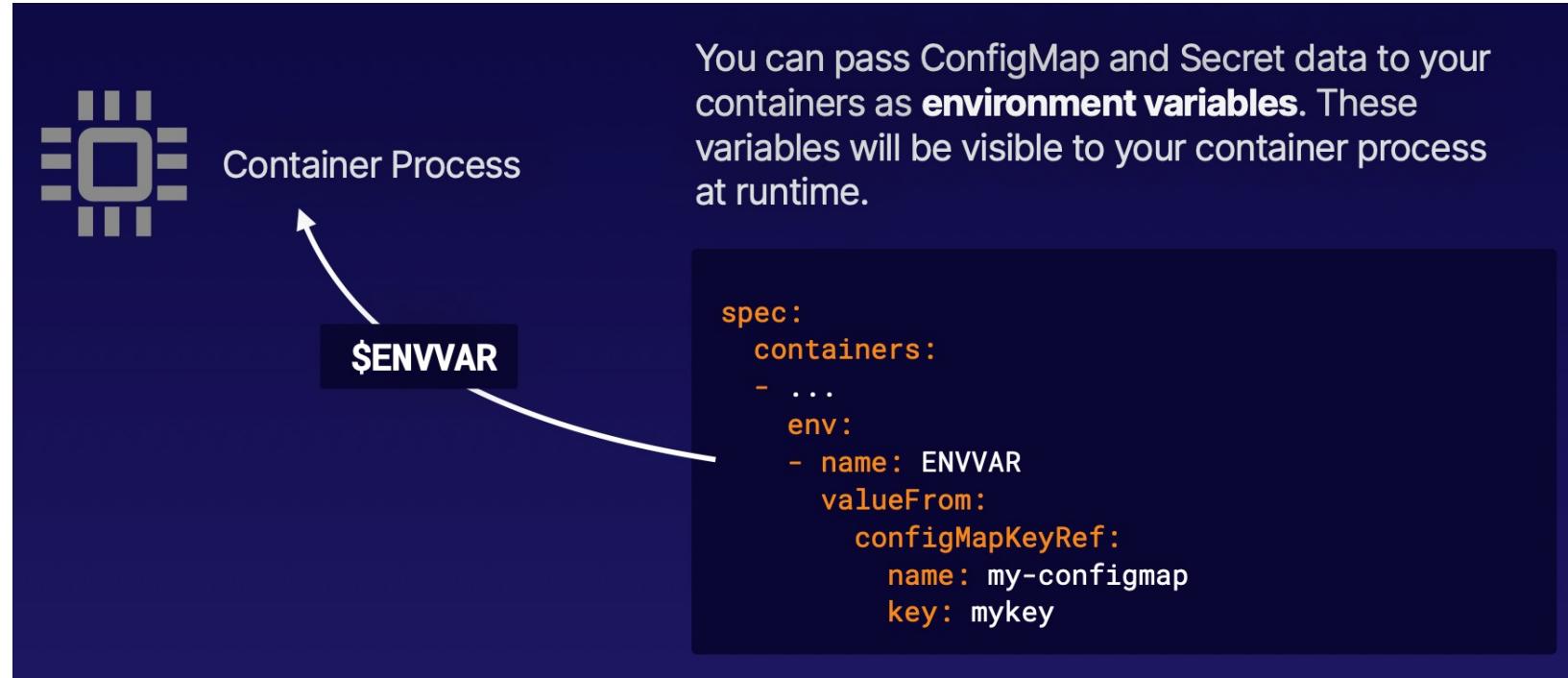
ConfigMaps

Secrets are similar to ConfigMaps but are designed to store sensitive data, such as passwords or API keys, more securely. They are created and used similarly to ConfigMaps.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: user
  password: mypass
```

Secrets

Environment Variables



Configuration Volumes

Configuration data from ConfigMaps and Secrets can also be passed to containers in the form of **mounted volumes**. This will cause the configuration data to appear in files available to the container file system.

Each top-level key in the configuration data will appear as a file containing all keys below that top-level key.

```
...
volumes:
- name: secret-vol
  secret:
    secretName: my-secret
```

Workshop 4

Managing Application Configuration

```
01-config-map.yml ×  
1 apiVersion: v1  
2 kind: ConfigMap  
3 metadata:  
4   name: my-configmap  
5   namespace: workshop-04  
6 data:  
7   key1: Hello, world!  
8   key2: |  
9     Test  
10    multiple lines  
11    more lines  
12   key3: >  
13     Test  
14     multiple lines  
15     one lines|
```

```
00-namespace.yml ×  
1 apiVersion: v1  
2 kind: Namespace  
3 metadata:  
4   name: workshop-04  
5   labels:  
6     name: workshop-04  
7     module: workshop-04
```

```
02-secret.yml ×  
1 apiVersion: v1  
2 kind: Secret  
3 metadata:  
4   name: my-secret  
5   namespace: workshop-04  
6 type: Opaque  
7 data:  
8   secretkey1: c2VjcmV0  
9   secretkey2: YW5vdGhlcnNlY3JldA==  
10
```

Workshop 4

Managing Application Configuration

```
03-environment-variable.yaml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   name: env-pod  
5   namespace: workshop-04  
6 spec:  
7   containers:  
8     - name: busybox  
9       image: busybox  
10      command: [ '/bin/sh', '-c', 'while true; do sleep 5; echo "configmap: $CONFIGMAPVAR secret: $SECRETVAR"; done' ]  
11    env:  
12      - name: CONFIGMAPVAR  
13        valueFrom:  
14          configMapKeyRef:  
15            name: my-configmap  
16            key: key1  
17      - name: SECRETVAR  
18        valueFrom:  
19          secretKeyRef:  
20            name: my-secret  
21            key: secretkey1
```

Workshop 4

Managing Application Configuration

```
 04-volume-pod.yaml ×  
1  apiVersion: v1  
2  kind: Pod  
3  metadata:  
4    name: volume-pod  
5    namespace: workshop-04  
6  spec:  
7    containers:  
8      - name: busybox  
9        image: busybox  
10       command: [ '/bin/sh', '-c', 'while true; do sleep 5; done']  
11       volumeMounts:  
12         - name: configmap-volume  
13           mountPath: /etc/config/configmap  
14         - name: secret-volume  
15           mountPath: /etc/config/secret  
16       volumes:  
17         - name: configmap-volume  
18           configMap:  
19             name: my-configmap  
20         - name: secret-volume  
21           secret:  
22             secretName: my-secret
```

Workshop 4

Managing Application Configuration

```
05-nginx-pod.yaml x
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    namespace: workshop-04
5    name: nginx-config
6  data:
7    nginx.conf: |
8      user  nginx;
9      worker_processes  1;
10
11      error_log  /var/log/nginx/error.log warn;
12      pid      /var/run/nginx.pid;
13
14      events {
15        worker_connections  1024;
16      }
17
18      http {
19        server {
20          listen      80;
21          listen  [::]:80;
22          server_name  localhost;
23          location / {
24            root   /usr/share/nginx/html;
25            index  index.html index.htm;
26          }
27          auth_basic "Secure Site";
28          auth_basic_user_file conf/.htpasswd;
29        }
30      }
31  ---
```

```
05-nginx-pod.yaml x
32  apiVersion: v1
33  kind: Pod
34  metadata:
35    namespace: workshop-04
36    name: nginx
37  spec:
38    containers:
39      - name: nginx
40        image: nginx
41        ports:
42          - containerPort: 80
43        volumeMounts:
44          - name: config-volume
45            mountPath: /etc/nginx
46          - name: htpasswd-volume
47            mountPath: /etc/nginx/conf
48    volumes:
49      - name: config-volume
50        configMap:
51          name: nginx-config
52      - name: htpasswd-volume
53        secret:
54          secretName: nginx-htpasswd
55  ---
56
57  apiVersion: v1
58  kind: Pod
59  metadata:
60    namespace: workshop-04
61    name: curl-pod
62  spec:
63    containers:
64      - name: curl-pod
65        image: curlimages/curl
66        command: [ 'sh', '-c', 'while true; do sleep 5; done' ]
```

Kubernetes Metrics Server

In order to view metrics about the resources pods and containers are using, we need an add-on to collect and provide that data. One such add-on is **Kubernetes Metrics Server**.

Reference: <https://github.com/kubernetes-sigs/metrics-server>

Metrics
Server

With `kubectl top`, you can view data about resource usage in your pods and nodes. `kubectl top` also supports flags like `--sort-by` and `--selector`.

```
$ kubectl top pod --sort-by <JSONPATH> --  
selector <selector>
```

`kubectl top`

Resource Requests

Resource requests allow you to define an amount of resources (such as CPU or memory) you expect a container to use. The Kubernetes scheduler will use resource requests to avoid scheduling pods on nodes that do not have enough available resources.

Tip: Containers are allowed to use more (or less) than the requested resources. Resource requests only affect scheduling.

Memory is measured in bytes. CPU is measured in CPU units, which are 1/1000 of one CPU.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: busybox
    image: busybox
    resources:
      requests:
        cpu: "250m"
        memory: "128Mi"
```

Resource Limits

Resource limits provide a way for you to limit the amount of resources your containers can use. The container runtime is responsible for enforcing these limits, and different container runtimes do this differently.

Tip: Some runtimes will enforce these limits by terminating container processes that attempt to use more than the allowed amount of resources.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: busybox
      image: busybox
      resources:
        limits:
          cpu: "250m"
          memory: "128Mi"
```

Reference: <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#resource-requests-and-limits-of-pod-and-container>

Container Health

K8s provides a number of features that allow you to build robust solutions, such as the ability to automatically restart unhealthy containers. To make the most of these features, K8s needs to be able to accurately determine the status of your applications. This means actively monitoring **container health**.

Liveness Probes

Liveness probes allow you to automatically determine whether or not a container application is in a healthy state.

By default, K8s will only consider a container to be "down" if the container process stops.

Liveness probes allow you to customize this detection mechanism and make it more sophisticated.

Startup Probes

Startup probes are very similar to liveness probes. However, while liveness probes run constantly on a schedule, startup probes run at container startup and stop running once they succeed.

They are used to determine when the application has successfully started up. Startup probes are especially useful for legacy applications that can have long startup times.

Readiness Probes

Readiness probes are used to determine when a container is ready to accept requests. When you have a service backed by multiple container endpoints, user traffic will not be sent to a particular pod until its containers have all passed the readiness checks defined by their readiness probes.

Use readiness probes to prevent user traffic from being sent to pods that are still in the process of starting up.

Pod Life Cycle: <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/>

Container Probes: <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes>

Workshop 5

Managing Container Resources

The image shows a code editor with four tabs open, each containing a YAML configuration file for Kubernetes:

- 00-namespace.yml**: A Namespace resource named "workshop-05" with a label "app: metrics-test".
- 01-metrics-test.yml**: A Pod resource named "metrics-test" in the "workshop-05" namespace, containing a single container "busybox" running a sleep loop command.
- 02-big-request-pod.yml**: A Pod resource named "big-request-pod" in the "workshop-05" namespace, containing a single container "busybox" running a sleep loop command, with a note to add requests for CPU and memory.
- 03-resource-pod.yml**: A Pod resource named "resource-pod" in the "workshop-05" namespace, containing a single container "busybox" running a sleep loop command, with defined requests and limits for both CPU and memory.

Workshop 5

Managing Container Resources

```
04-liveness-pod.yaml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-05  
5   name: liveness-pod  
6 spec:  
7   containers:  
8     - name: busybox  
9       image: busybox  
10      command: ['sh', '-c', 'while true; do sleep 5; done']  
11      livenessProbe:  
12        exec:  
13          command: ['echo', 'Hello, World!']  
14        initialDelaySeconds: 5  
15        periodSeconds: 5
```

```
05-liveness-pod-http.yaml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-05  
5   name: liveness-pod-http  
6 spec:  
7   containers:  
8     - name: nginx  
9       image: nginx:latest  
10      livenessProbe:  
11        httpGet:  
12          path: /  
13          port: 80  
14        initialDelaySeconds: 5  
15        periodSeconds: 5
```

```
06-startup-pod.yaml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-05  
5   name: startup-pod  
6 spec:  
7   containers:  
8     - name: nginx  
9       image: nginx:latest  
10      startupProbe:  
11        httpGet:  
12          path: /  
13          port: 80  
14        failureThreshold: 30  
15        periodSeconds: 10
```

```
07-readiness-pod.yaml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-05  
5   name: readiness-pod  
6 spec:  
7   containers:  
8     - name: nginx  
9       image: nginx:latest  
10      readinessProbe:  
11        httpGet:  
12          path: /  
13          port: 80  
14        initialDelaySeconds: 5  
15        periodSeconds: 5
```

Self-Healing Pods with Restart Policies

Restart Policies

K8s can automatically restart containers when they fail. **Restart policies** allow you to customize this behavior by defining when you want a pod's containers to be automatically restarted.

Restart policies are an important component of self-healing applications, which are automatically repaired when a problem arises.

There are three possible values for a pod's restart policy in K8s: **Always**, **OnFailure**, and **Never**.

Always

Always is the default restart policy in K8s. With this policy, containers will always be restarted if they stop, even if they completed successfully. Use this policy for applications that should always be running.

OnFailure

The **OnFailure** restart policy will restart containers only if the container process exits with an error code or the container is determined to be unhealthy by a liveness probe. Use this policy for applications that need to run successfully and then stop.

Never

The **Never** restart policy will cause the pod's containers to never be restarted, even if the container exits or a liveness probe fails. Use this for applications that should run once and never be automatically restarted.

Restart Policy: <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy>

Workshop 6

Managing Container Resources

```
01-always-pod.yml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-06  
5   name: always-pod  
6 spec:  
7   restartPolicy: Always  
8   containers:  
9     - name: busybox  
10    image: busybox  
11    command: [ 'sh', '-c', 'sleep 10' ]
```

```
02-onfailure-pod.yml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-06  
5   name: onfailure-pod  
6 spec:  
7   restartPolicy: OnFailure  
8   containers:  
9     - name: busybox  
10    image: busybox  
11    command: [ 'sh', '-c', 'sleep 10' ]
```

```
03-onfailure-pod-with-err.yml ×  
1 apiVersion: v1  
2 kind: Pod  
3 metadata:  
4   namespace: workshop-06  
5   name: onfailure-pod-with-err  
6 spec:  
7   restartPolicy: OnFailure  
8   containers:  
9     - name: busybox  
10    image: busybox  
11    command: [ 'sh', '-c', 'sleep 10; this is a bad command that will fail' ]
```

Workshop 6

Managing Container Resources

```
04-never-pod.yaml ×  
1  apiVersion: v1  
2  kind: Pod  
3  metadata:  
4    namespace: workshop-06  
5    name: never-pod  
6  spec:  
7    restartPolicy: Never  
8    containers:  
9      - name: busybox  
10        image: busybox  
11        command: [ 'sh', '-c', 'sleep 10; this is a bad command that will fail' ]
```

Example Use case

You have an application that is hard-coded to write log output to a file on disk.

You add a secondary container to the Pod (sometimes called a **sidecar**) that reads the log file from a shared volume and prints it to the console so the log output will appear in the container log.

A Kubernetes Pod can have one or more containers. A Pod with more than one container is a **multi-container Pod**.

In a multi-container Pod, the containers share resources such as network and storage. They can interact with one another, working together to provide functionality.

Best practice: Keep containers in separate Pods unless they need to share resources.

Cross-Container Interaction

Containers sharing the same Pod can interact with one another using shared resources.



Network

Containers share the same networking namespace and can communicate with one another on any port, even if that port is not exposed to the cluster.

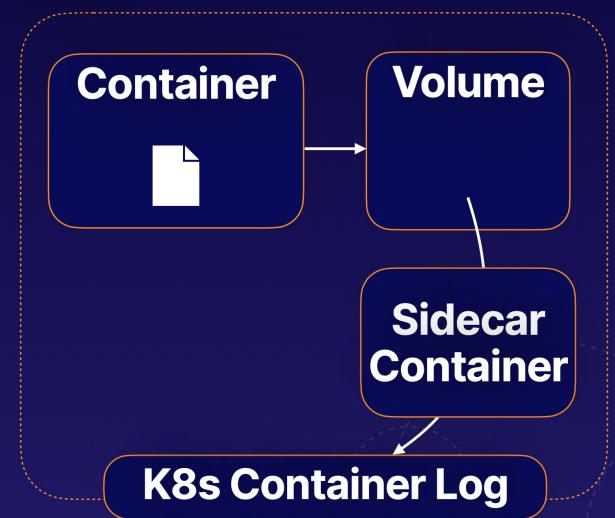


Storage

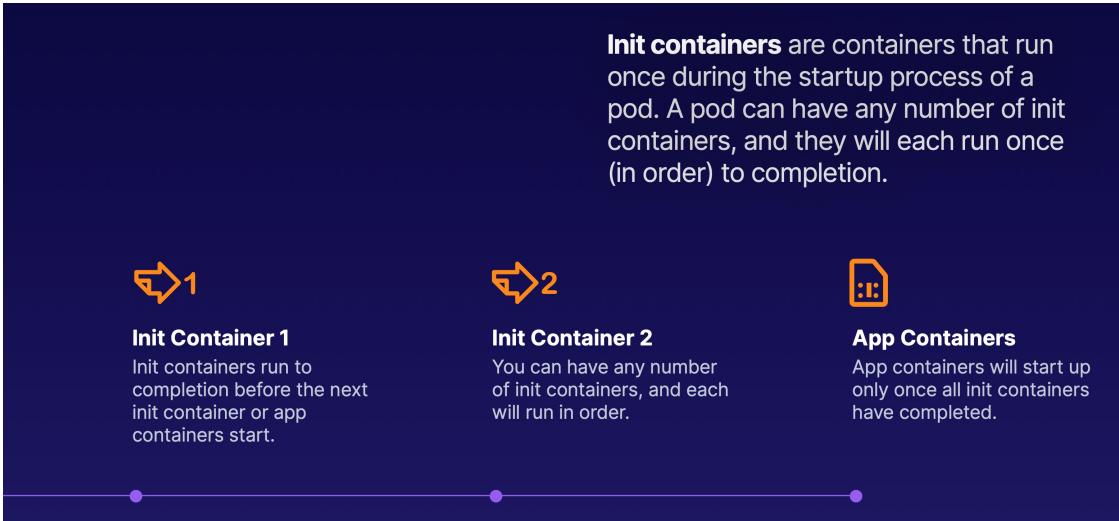
Containers can use volumes to share data in a Pod.

Multi-Container Pod

Pod



Init containers



You can use init containers to perform a variety of startup tasks. They can contain and use software and setup scripts that are not needed by your main containers.

They are often useful in keeping your main containers lighter and more secure by offloading startup tasks to a separate container.

Use Cases for Init Containers

Some sample use cases for init containers:

- Cause a pod to wait for another K8s resource to be created before finishing startup.
- Perform sensitive startup steps securely outside of app containers.
- Populate data into a shared volume at startup.
- Communicate with another service at startup.

Workshop 7

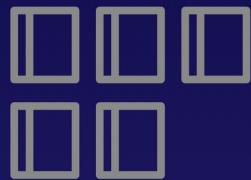
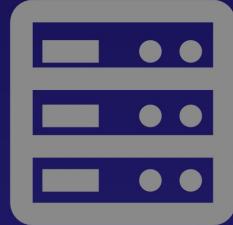
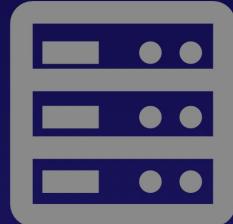
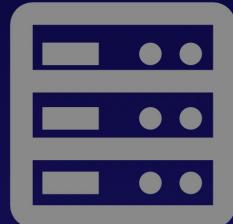
Multi-Container and Init Containers

```
01-sidecar-pod.yaml ×  
1  apiVersion: v1  
2  kind: Pod  
3  metadata:  
4    namespace: workshop-07  
5    name: sidecar-pod  
6  spec:  
7    containers:  
8      - name: busybox  
9        image: busybox  
10       command: ['sh', '-c', 'while true; do now=$(date);echo logs time $now >> /logs/output.log; sleep 5; done']  
11       volumeMounts:  
12         - name: sharedvol  
13           mountPath: /logs  
14      - name: sidecar  
15        image: busybox  
16        command: ['sh', '-c', 'tail -f /logs/output.log']  
17        volumeMounts:  
18          - name: sharedvol  
19            mountPath: /logs  
20    volumes:  
21      - name: sharedvol  
22        emptyDir: {}
```

Workshop 7

Multi-Container and Init Containers

```
02-initial-container.yml x
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    namespace: workshop-07
5    name: shipping-backend
6    labels:
7      app: shipping-svc
8  spec:
9    containers:
10      - name: nginx
11        image: nginx:latest
12    ...
13
14  apiVersion: v1
15  kind: Service
16  metadata:
17    namespace: workshop-07
18    name: shipping-svc
19  spec:
20    selector:
21      app: shipping-svc
22    ports:
23      - protocol: TCP
24        port: 80
25        targetPort: 80
26    ...
27
28  apiVersion: v1
29  kind: Pod
30  metadata:
31    namespace: workshop-07
32    name: shipping-web
33  spec:
34    containers:
35      - name: nginx
36        image: nginx:latest
37    initContainers:
38      - name: shipping-svc-check
39        image: curlimages/curl
40        command: ['sh', '-c', 'until curl -s -f -o /dev/null "http://shipping-svc"; do echo waiting for shipping-svc; sleep 5; done']
```



Scheduling

The process of assigning Pods to Nodes so kubelets can run them

Scheduler

Control plane component that handles scheduling

What Is Scheduling?

Kubernetes Scheduler: <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>

Assigning Pods to Nodes: <https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>

Scheduling Process and Node Selector

Scheduling Process

The Kubernetes scheduler selects a suitable Node for each Pod. It takes into account things like:

- Resource requests vs. available node resources
- Various configurations that affect scheduling using node labels

nodeSelector

You can configure a **nodeSelector** for your Pods to limit which Node(s) the Pod can be scheduled on.

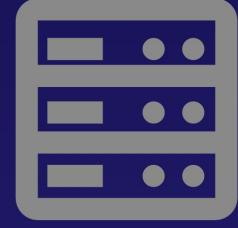
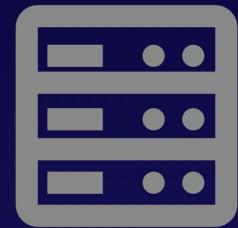
Node selectors use node labels to filter suitable nodes.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
    nodeSelector:
      mylabel: myvalue
```

nodeName

You can bypass scheduling and assign a Pod to a specific Node by name using **nodeName**.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
    nodeName: k8s-worker1
```



DaemonSet

Automatically runs a copy of a Pod on each node.

DaemonSets will run a copy of the Pod on new nodes as they are added to the cluster.

DaemonSet:

<https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>

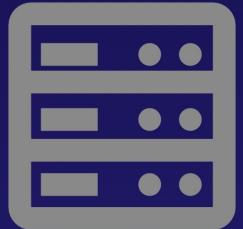
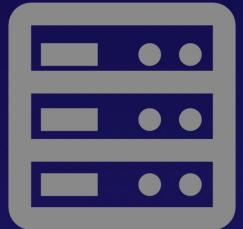
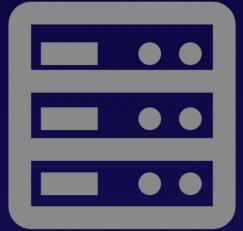
DaemonSet

Workshop 8

Scheduling and DaemonSets

The image shows a code editor with three tabs open, each displaying a YAML configuration file for Kubernetes:

- 01-nodeselector-pod.yaml**: A Pod specification for a frontend application. It uses a node selector to run on nodes with the "worker-type: frontend" label. The containers run an Nginx instance.
- 02-nodename-pod.yaml**: A Pod specification for a nodename application. It uses a node name selector to run on a specific node named "k3d-mycluster-agent-0". The containers run an Nginx instance.
- 03-daemonset-pod.yaml**: A DaemonSet specification named "my-daemonset". It runs an Nginx container on every node in the "workshop-08" namespace. The containers run an Nginx instance.



Deployment
(with 3 replicas)



Deployment

A K8s object that defines a **desired state** for a ReplicaSet (a set of replica Pods). The Deployment Controller seeks to maintain the desired state by creating, deleting, and replacing Pods with new configurations.

Deployment

Deployments:

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

A Deployment's Desired State Includes...

1

replicas

The number of replica Pods the Deployment will seek to maintain

2

selector

A label selector used to identify the replica Pods managed by the Deployment

3

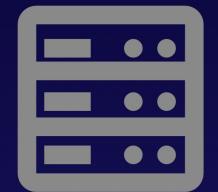
template

A template Pod definition used to create replica Pods

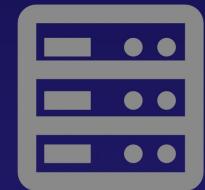
Deployment Desired State

Scaling refers to dedicating more (or fewer) resources to an application in order to meet changing needs.

K8s Deployments are very useful in **horizontal scaling**, which involves changing the number of containers running an application.



Deployment
Replicas: 3



Deployment Scaling

The Deployment's **replicas** setting determines how many replicas are desired in its desired state. If the **replicas** number is changed, replica Pods will be created or deleted to satisfy the new number.

Scaling with Deployments

How to Scale

Edit YAML

You can scale a deployment simply by changing the number of **replicas** in the YAML descriptor with **kubectl apply** or **kubectl edit**.

```
...  
spec:  
  replicas: 5  
...
```

kubectl scale

Or use the special **kubectl scale** command.

```
$ kubectl scale \  
  deployment.v1.apps/my-deployment \  
  --replicas=5
```

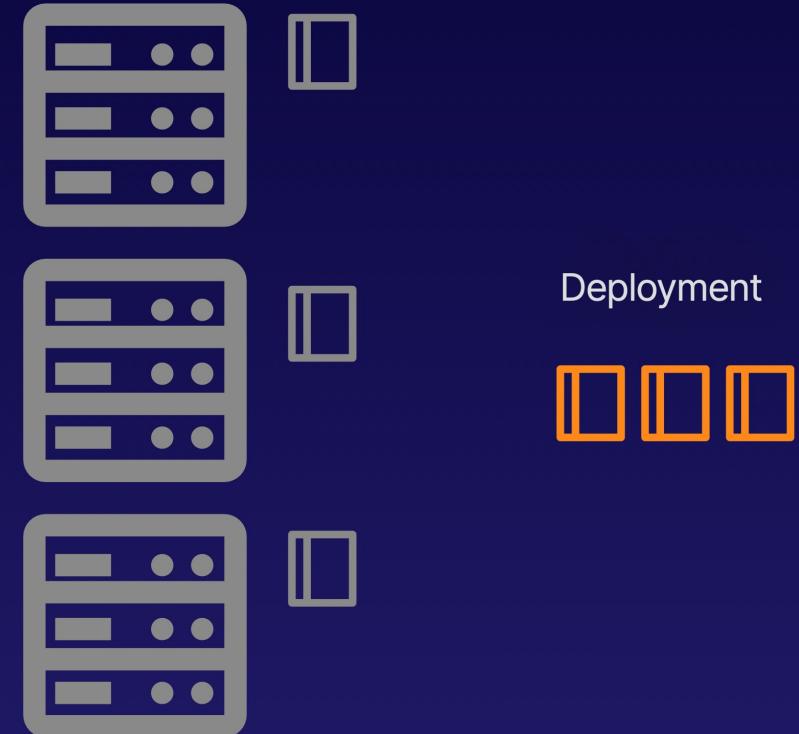
Scaling a Deployment:

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment>

Rolling Update

Rolling Update

Rolling updates allow you to make changes to a Deployment's Pods at a controlled rate, gradually replacing old Pods with new Pods. This allows you to update your Pods without incurring downtime.





Rollback

If an update to a deployment causes a problem, you can **roll back** the deployment to a previous working state.

Rollback

Workshop 9

Deployment

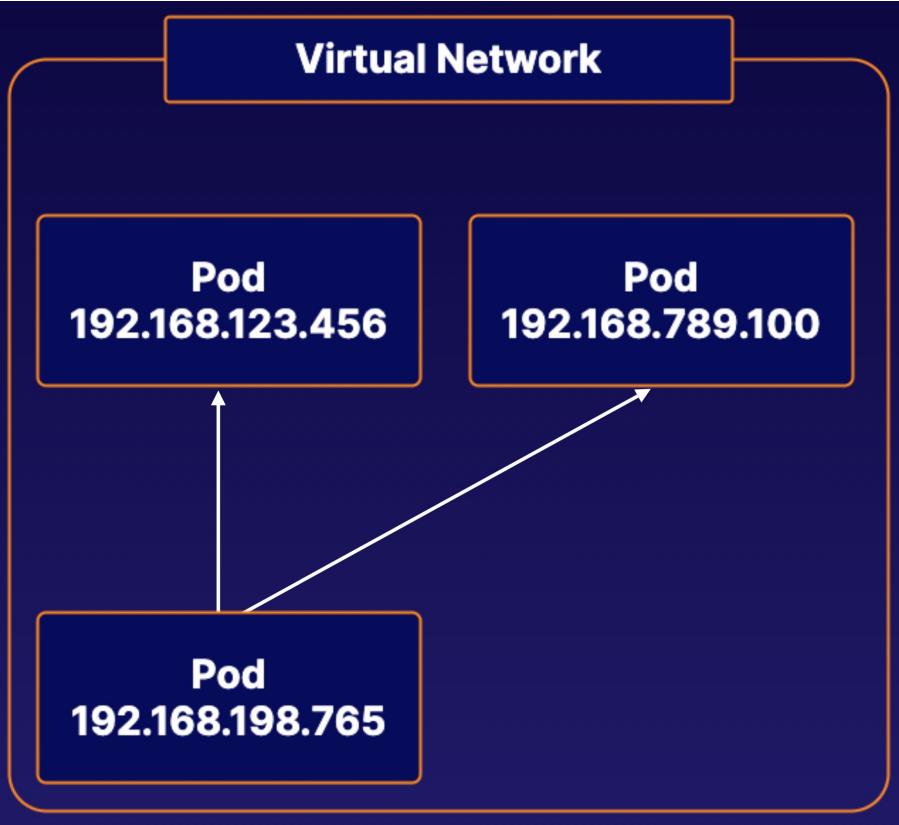
```
01-my-deployment.yaml ×  
1  apiVersion: apps/v1  
2  kind: Deployment  
3  metadata:  
4    namespace: workshop-09  
5    name: my-deployment  
6  spec:  
7    replicas: 6  
8    selector:  
9    →:      matchLabels:  
10       | app: my-deployment  
11    template:  
12      metadata:  
13      →:        labels:  
14         | app: my-deployment  
15    spec:  
16      containers:  
17        - name: nginx  
18        image: nginx:latest  
19        ports:  
20          - containerPort: 80  
21
```

Network Model

The Kubernetes network model defines how Pods communicate with each other, regardless of which Node they are running on.

Each Pod has its own **unique IP address** within the cluster.

Any Pod can reach any other Pod using that Pod's IP address. This creates a **virtual network** that allows Pods to easily communicate with each other, regardless of which node they are on.



The K8s virtual network uses a **DNS** (Domain Name System) to allow Pods to locate other Pods and Services using domain names instead of IP addresses.

This DNS runs as a Service within the cluster. You can usually find it in the **kube-system** namespace.

Kubeadm clusters use **CoreDNS**.

Pod Domain Names

All Pods in our kubeadm cluster are automatically given a domain name of the following form:

`pod-ip-address.namespace-name.pod.cluster.local`

A Pod in the **default** namespace with the IP address 192.168.10.100 would have a domain name like this:

`192-168-10-100.default.pod.cluster.local`

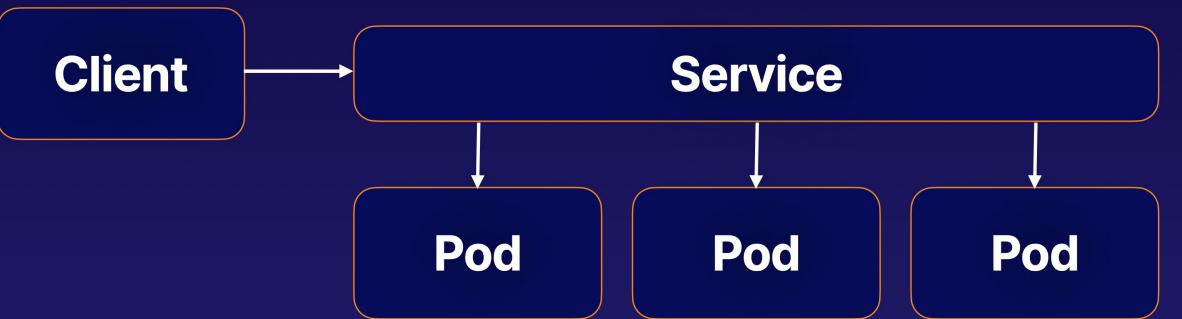
DNS in K8s

DNS for Services and Pods: <https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/>

Kubernetes **Services** provide a way to expose an application running as a set of Pods.

They provide an abstract way for clients to access applications without needing to be aware of the application's Pods.

Clients make requests to a Service, which **routes** traffic to its Pods in a load-balanced fashion.



Service

Endpoints are the backend entities to which Services route traffic. For a Service that routes traffic to multiple Pods, each Pod will have an endpoint associated with the Service.

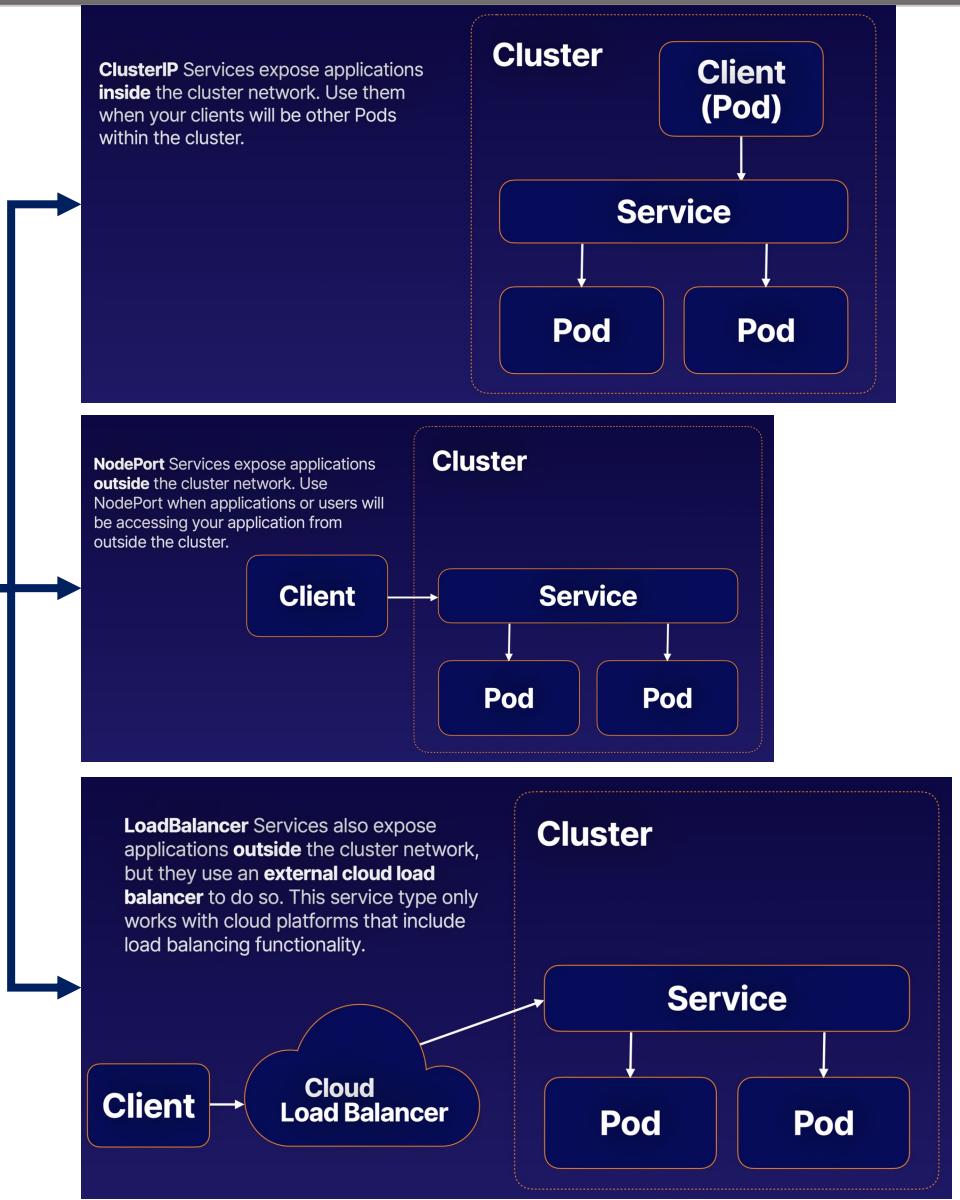
Tip: One way to determine which Pod(s) a Service is routing traffic to is to look at that service's Endpoints.

Endpoint

Service Types

Each Service has a type. The **Service type** determines how and where the Service will expose your application. There are four service types:

- ClusterIP
- NodePort
- LoadBalancer
- ExternalName



The Kubernetes DNS (Domain Name System) assigns **DNS names** to Services, allowing applications within the cluster to easily locate them.

A service's fully qualified domain name has the following format:

`service-name.namespace-name.svc.cluster-domain.example`

Service DNS and Namespaces

A Service's fully qualified domain name can be used to reach the service from within **any Namespace** in the cluster.

However, Pods within the same Namespace can also simply use the service name.

`my-service`

Discovering Services With DNS

An **Ingress** is a Kubernetes object that manages external access to Services in the cluster.

An Ingress is capable of providing more functionality than a simple NodePort Service, such as SSL termination, advanced load balancing, or name-based virtual hosting.

Ingress



External Clients

Ingress

Service



Routing to a Service

Ingresses define a set of **routing rules**. A routing rule's properties determine to which requests it applies.

Each rule has a set of **paths**, each with a **backend**. Requests matching a path will be routed to its associated backend.

In this example, a request to *http://<some-endpoint>/somepath* would be routed to port 80 on the **my-service** Service.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
    paths:
    - path: /somepath
      pathType: Prefix
      backend:
        service:
          name: my-service
          port:
            number: 80
```

Routing to a Service

If a Service uses a **named port**, an Ingress can also use the port's name to choose to which port it will route.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: web
      protocol: TCP
      port: 80
      targetPort: 8080
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
    - http:
        paths:
          - path: /somepath
            pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              name: web
```

Workshop 10

Service and Ingress

```
01-dnstest.yaml x
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    namespace: workshop-10
5    name: netshoot
6  spec:
7    containers:
8      - name: netshoot
9        image: nicolaka/netshoot
10       command: ['sh', '-c', 'while true; do sleep 5; done']
11       securityContext:
12         privileged: true
13   ---
14  apiVersion: v1
15  kind: Pod
16  metadata:
17    namespace: workshop-10
18    name: nginx-dnstest
19  spec:
20    containers:
21      - name: nginx
22        image: nginx:latest
23        ports:
24          - containerPort: 80
```

Workshop 10

Service and Ingress

```
❶ 02-frontend-app.yml ×
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    namespace: workshop-10
5    name: frontend-app
6  spec:
7    replicas: 3
8    selector:
9    →:      matchLabels:
10       app: frontend-app
11    template:
12      metadata:
13      →:        labels:
14         app: frontend-app
15    spec:
16      containers:
17        - name: nginx
18          image: nginx:latest
19        ports:
20        →:          - containerPort: 80
21
```

```
❷ 03-svc-clusterip.yml ×
1  apiVersion: v1
2  kind: Service
3  metadata:
4    namespace: workshop-10
5    name: svc-clusterip
6  spec:
7    type: ClusterIP
8  →:    selector:
9     app: frontend-app
10    ports:
11      - protocol: TCP
12        port: 80
13        targetPort: 80
```

Workshop 10

Service and Ingress

```
04-svc-nodeport.yml ×  
1 apiVersion: v1  
2 kind: Service  
3 metadata:  
4   namespace: workshop-10  
5   name: svc-nodeport  
6 spec:  
7   type: NodePort  
8 →  
9     selector:  
10       app: frontend-app  
11     ports:  
12       - protocol: TCP  
13         port: 80  
14         targetPort: 80  
15         nodePort: 30080|
```

```
05-svc-external.yml ×  
1 apiVersion: v1  
2 kind: Service  
3 metadata:  
4   namespace: workshop-10  
5   name: chula-hospital  
6 spec:  
7   type: ExternalName  
8     externalName: chulalongkornhospital.go.th|
```

Workshop 10

Service and Ingress

```
06-my-ingress.yaml ×  
1  apiVersion: networking.k8s.io/v1  
2  kind: Ingress  
3  metadata:  
4    namespace: workshop-10  
5    name: my-ingress  
6    annotations:  
7      ingress.kubernetes.io/ssl-redirect: "false"  
8  spec:  
9    rules:  
10      - http:  
11        paths:  
12          - path: /  
13            pathType: Prefix  
14            backend:  
15              service:  
16                name: svc-clusterip  
17                port:  
18                  number: 80
```

Container File Systems

The container file system is **ephemeral**. Files on the container's file system exist only as long as the container exists.

If a container is deleted or re-created in K8s, data stored on the container file system is lost.

Pod

Container



Many applications need a more persistent method of data storage.

Volumes allow you to store data outside the container file system while allowing the container to access the data at runtime.

This can allow data to persist beyond the life of the container.

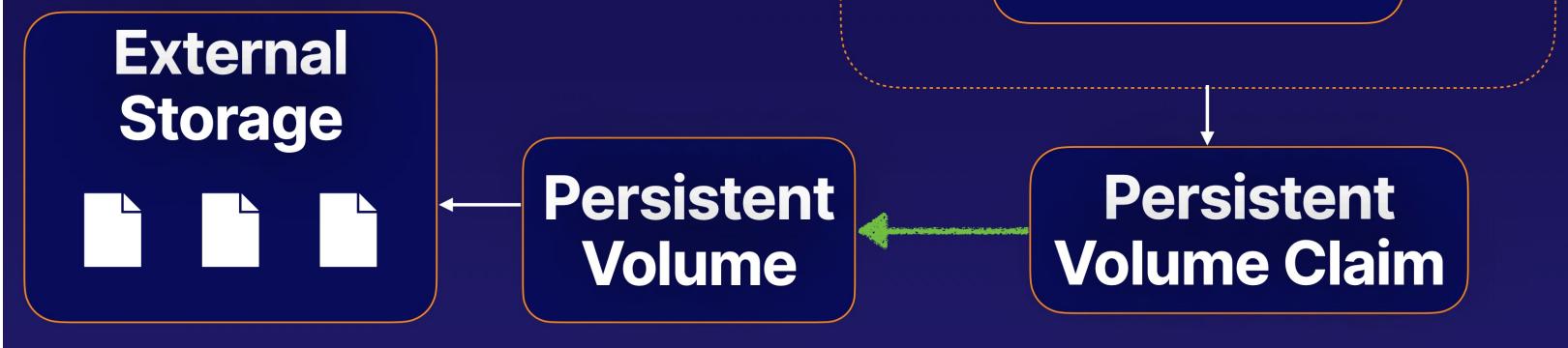


Volumes

Persistent Volumes

Volumes offer a simple way to provide external storage to containers within the Pod/container spec.

Persistent Volumes are a slightly more advanced form of Volume. They allow you to treat storage as an abstract resource and consume it using your Pods.



Volume Types

Both Volumes and Persistent Volumes each have a **volume type**. The volume type determines how the storage is actually handled.

Various volume types support storage methods such as:

- NFS
- Cloud storage mechanisms (AWS, Azure, GCP)
- ConfigMaps and Secrets
- A simple directory on the K8s node

Volumes and volumeMounts

Regular Volumes can be set up relatively easily within a Pod/container specification.

volumes: In the Pod spec, these specify the storage volumes available to the Pod. They specify the volume type and other data that determines where and how the data is actually stored.

volumeMounts: In the container spec, these reference the volumes in the Pod spec and provide a `mountPath` (the location on the file system where the container process will access the volume data).

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
    - name: busybox
      image: busybox
        volumeMounts:
          - name: my-volume
            mountPath: /output
  volumes:
    - name: my-volume
      hostPath:
        path: /data
```

Sharing Volumes Between Containers

You can use `volumeMounts` to mount the same volume to multiple containers within the same Pod.

This is a powerful way to have multiple containers interact with one another. For example, you could create a secondary sidecar container that processes or transforms output from another container.

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
    - name: busybox1
      image: busybox
      volumeMounts:
        - name: my-volume
          mountPath: /output
    - name: busybox2
      image: busybox
      volumeMounts:
        - name: my-volume
          mountPath: /input
  volumes:
    - name: my-volume
      emptyDir: {}
```

Common Volume Types

There are many volume types, but there are two you may want to be especially aware of.

hostPath: Stores data in a specified directory on the K8s node.

emptyDir: Stores data in a dynamically created location on the node. This directory exists only as long as the Pod exists on the node. The directory and the data are deleted when the Pod is removed. This volume type is very useful for simply sharing data between containers in the same Pod.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-pv
spec:
  storageClassName: localdisk
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /var/output
```

PersistentVolumes are K8s objects that allow you to treat storage as an abstract resource to be consumed by Pods, much like K8s treats compute resources such as memory and CPU.

A PersistentVolume uses a set of attributes to describe the underlying storage resource (such as a disk or cloud storage location) which will be used to store data.

PersistantVolumes

Storage Classes

Storage Classes allow K8s administrators to specify the types of storage services they offer on their platform.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: localdisk
provisioner: kubernetes.io/no-provisioner
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/no-provisioner
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/no-provisioner
```

For example, an administrator could create a StorageClass called **slow** to describe low-performance but inexpensive storage resources, and another called **fast** for high-performance but more costly resources.

This would allow users to choose storage resources that fit the needs of their applications.

Storage Classes

A PersistentVolume's **persistentVolumeReclaimPolicy** determines how the storage resources can be reused when the PersistentVolume's associated PersistentVolumeClaims are deleted.

- **Retain:** Keeps all data. This requires an administrator to manually clean up the data and prepare the storage resource for reuse.
- **Delete:** Deletes the underlying storage resource automatically (only works for cloud storage resources).
- **Recycle:** Automatically deletes all data in the underlying storage resource, allowing the PersistentVolume to be reused.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-pv
spec:
  storageClassName: localdisk
  persistentVolumeReclaimPolicy: Recycle
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /etc/output
```

reclaimPolicies

A **PersistentVolumeClaim** represents a user's request for storage resources. It defines a set of attributes similar to those of a PersistentVolume (StorageClass, etc.).

When a PersistentVolumeClaim is created, it will look for a PersistentVolume that is able to meet the requested criteria. If it finds one, it will automatically be **bound** to the PersistentVolume.

**Persistent
Volume**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: localdisk
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

PersistentVolumeClaims

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod
spec:
  containers:
    - name: busybox
      image: busybox
      volumeMounts:
        - name: pv-storage
          mountPath: /output
  volumes:
    - name: pv-storage
      persistentVolumeClaim:
        claimName: my-pvc
```

PersistentVolumeClaims can be **mounted** to a Pod's containers just like any other volume.

If the PersistentVolumeClaim is bound to a PersistentVolume, the containers will use the underlying PersistentVolume storage.

Using a PersistentVolumeClaims in a Pod

Workshop 11

K8s Storage

```
01-volume-pod.yaml ×  
1  apiVersion: v1  
2  kind: Pod  
3  metadata:  
4    namespace: workshop-11  
5    name: volume-pod  
6  spec:  
7    restartPolicy: Never  
8    containers:  
9      - name: busybox  
10        image: busybox  
11        command: [sh, -c, 'while true; do now=$(date); echo "[now]: process success!" >> /output/volume-pod.log; sleep 5; done']  
12        volumeMounts:  
13          - name: my-volume  
14            mountPath: /output  
15    volumes:  
16      - name: my-volume  
17        hostPath:  
18          path: /var/data
```

Workshop 11

K8s Storage

```
02-share-data-pod.yaml ×
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    namespace: workshop-11
5    name: shared-data-pod
6  spec:
7    containers:
8      - name: busybox1
9        image: busybox
10       command: ['sh', '-c', 'while true; do now=$(date); echo "[${now}]: process success!" >> /output/output.log; sleep 2; done']
11       volumeMounts:
12         - name: shared-vol
13           mountPath: /output
14      - name: busybox2
15        image: busybox
16        command: ['sh', '-c', 'while true; do clear;cat /input/output.log; sleep 10; done']
17        volumeMounts:
18          - name: shared-vol
19            mountPath: /input
20    volumes:
21      - name: shared-vol
22        emptyDir: {}
```

Workshop 11

K8s Storage

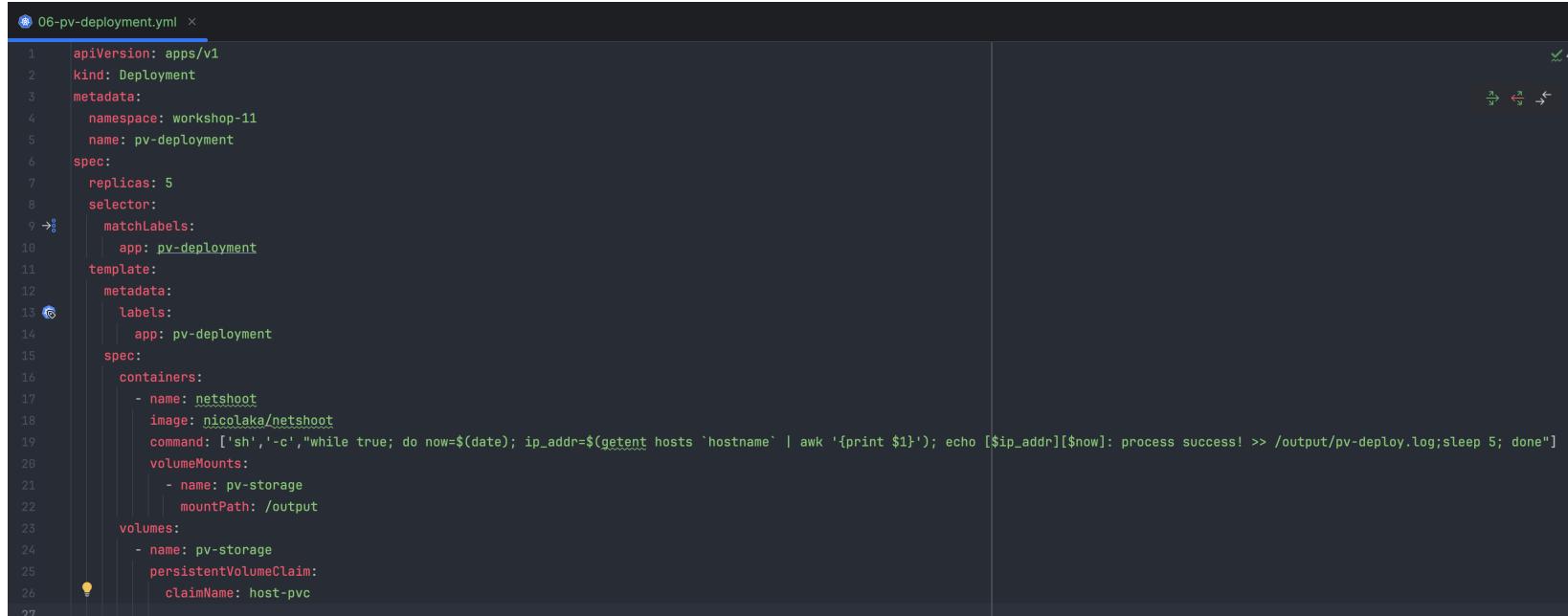
```
03-localdisk-sc.yaml ×
1 apiVersion: storage.k8s.io/v1
2 kind: StorageClass
3 metadata:
4   name: localdisk
5   provisioner: kubernetes.io/no-provisioner
6 allowVolumeExpansion: true

04-host-pv.yaml ×
1 apiVersion: v1
2 kind: PersistentVolume
3 metadata:
4   name: host-pv
5   labels:
6     type: node
7 spec:
8   storageClassName: localdisk
9   persistentVolumeReclaimPolicy: Retain
10  capacity:
11    storage: 1Gi
12    accessModes:
13      - ReadWriteMany
14    hostPath:
15      path: /var/data
16
```

```
05-host-pvc.yaml ×
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   namespace: workshop-11
5   name: host-pvc
6 spec:
7   storageClassName: localdisk
8   accessModes:
9     - ReadWriteMany
10  resources:
11    requests:
12      storage: 100Mi
13      volumeName: host-pv
```

Workshop 11

K8s Storage



```
06-pv-deployment.yaml
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   namespace: workshop-11
5   name: pv-deployment
6 spec:
7   replicas: 5
8   selector:
9     matchLabels:
10    app: pv-deployment
11 template:
12   metadata:
13     labels:
14       app: pv-deployment
15 spec:
16   containers:
17     - name: netshoot
18       image: nicolaka/netshoot
19       command: ['sh','-c','while true; do now=$(date); ip_addr=$(getent hosts `hostname` | awk '{print $1}'); echo [$ip_addr][$now]: process success! >> /output/pv-deploy.log; sleep 5; done']
20       volumeMounts:
21         - name: pv-storage
22           mountPath: /output
23 volumes:
24   - name: pv-storage
25     persistentVolumeClaim:
26       claimName: host-pvc
27
```

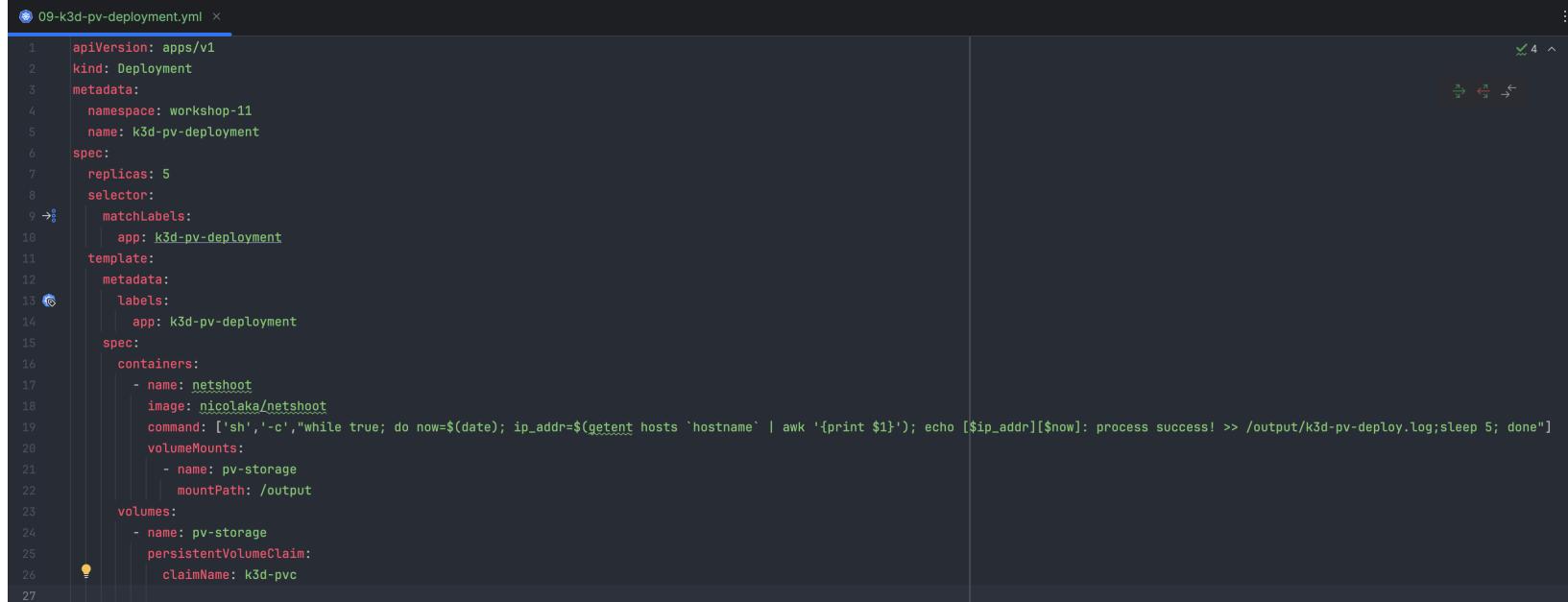
Workshop 11

K8s Storage

```
07-k3d-pv.yml ×          08-k3d-pvc.yml ×  
1  apiVersion: v1           1  apiVersion: v1  
2  kind: PersistentVolume  2  kind: PersistentVolumeClaim  
3  metadata:                metadata:  
4    name: k3d-pv            namespace: workshop-11  
5    labels:                  name: k3d-pvc  
6      type: cluster        spec:  
7  spec:                      storageClassName: manual  
8    storageClassName: manual  
9    capacity:                 accessModes:  
10   storage: 5Gi             - ReadWriteMany  
11   accessModes:              resources:  
12     - ReadWriteMany        requests:  
13   hostPath:                   storage: 1Gi  
14     path: "/tmp/k3dvol"      volumeName: k3d-pv|
```

Workshop 11

K8s Storage



```
09-k3d-pv-deployment.yaml
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   namespace: workshop-11
5   name: k3d-pv-deployment
6 spec:
7   replicas: 5
8   selector:
9     matchLabels:
10    app: k3d-pv-deployment
11 template:
12   metadata:
13     labels:
14       app: k3d-pv-deployment
15 spec:
16   containers:
17     - name: netshoot
18       image: nicolaka/netshoot
19       command: [sh,-c,"while true; do now=$(date); ip_addr=$(getent hosts `hostname` | awk '{print $1}'); echo [$ip_addr][$now]: process success! >> /output/k3d-pv-deploy.log; sleep 5; done"]
20     volumeMounts:
21       - name: pv-storage
22         mountPath: /output
23   volumes:
24     - name: pv-storage
25       persistentVolumeClaim:
26         claimName: k3d-pvc
27
```

thankyou.TM