Running Kubelet in Standalone Mode

This tutorial shows you how to run a standalone kubelet instance.

You may have different motivations for running a standalone kubelet. This tutorial is aimed at introducing you to Kubernetes, even if you don't have much experience with it. You can follow this tutorial and learn about node setup, basic (static) Pods, and how Kubernetes manages containers.

Once you have followed this tutorial, you could try using a cluster that has a control plane to manage pods and nodes, and other types of objects. For example, Hello, minikube.

You can also run the kubelet in standalone mode to suit production use cases, such as to run the control plane for a highly available, resiliently deployed cluster. This tutorial does not cover the details you need for running a resilient control plane.

Objectives

- Install `cri-o`, and `kubelet` on a Linux system and run them as `systemd` services.
- Launch a Pod running `nginx` that listens to requests on TCP port 80 on the Pod's IP address.
- Learn how the different components of the solution interact among themselves.

Caution:

The kubelet configuration used for this tutorial is insecure by design and should *not* be used in a production environment.

Before you begin

- Admin (`root`) access to a Linux system that uses `systemd` and `iptables` (or nftables with `iptables` emulation).
- Access to the Internet to download the components needed for the tutorial, such as:
  - A container runtime that implements the Kubernetes (CRI).
  - Network plugins (these are often known as Container Networking Interface (CNI))
  - Required CLI tools: `curl`, `tar`, `jq`.

Prepare the system

Swap configuration

By default, kubelet fails to start if swap memory is detected on a node. This means that swap should either be disabled or tolerated by kubelet.

Note:

If you configure the kubelet to tolerate swap, the kubelet still configures Pods (and the containers in those Pods) not to use swap space. To find out how Pods can actually use the available swap, you can read more about swap memory management on Linux nodes.

If you have swap memory enabled, either disable it or add `failSwapOn: false` to the kubelet configuration file.

To check if swap is enabled:

```
sudo swapon --show
```

If there is no output from the command, then swap memory is already disabled.

To disable swap temporarily:

```
sudo swapoff -a
```

To make this change persistent across reboots:

Make sure swap is disabled in either `/etc/fstab` or `systemd.swap`, depending on how it was configured on your system.

Enable IPv4 packet forwarding

To check if IPv4 packet forwarding is enabled:

```
cat /proc/sys/net/ipv4/ip_forward
```

If the output is `1`, it is already enabled. If the output is `0`, then follow next steps.

To enable IPv4 packet forwarding, create a configuration file that sets the `net.ipv4.ip_forward` parameter to `1`:

```
sudo tee /etc/sysctl.d/k8s.conf <<EOF
net.ipv4.ip_forward = 1
EOF
```

Apply the changes to the system:

```
sudo sysctl --system
```

The output is similar to:

```
...
* Applying /etc/sysctl.d/k8s.conf ...
net.ipv4.ip_forward = 1
* Applying /etc/sysctl.conf ...
```

Download, install, and configure the components

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Install a container runtime

Download the latest available versions of the required packages (recommended).

This tutorial suggests installing the CRI-O container runtime (external link).

There are several ways to install the CRI-O container runtime, depending on your particular Linux distribution. Although CRI-O recommends using either `deb` or `rpm` packages, this tutorial uses the *static binary bundle* script of the CRI-O Packaging project, both to streamline the overall process, and to remain distribution agnostic.

The script installs and configures additional required software, such as `cni-plugins`, for container networking, and `crun` and `runc`, for running containers.

The script will automatically detect your system's processor architecture (`amd64` or `arm64`) and select and install the latest versions of the software packages.

Set up CRI-O

Visit the releases page (external link).

Download the static binary bundle script:

```
curl https://raw.githubusercontent.com/cri-o/packaging/main/get > crio-install
```

Run the installer script:

```
sudo bash crio-install
```

Enable and start the `crio` service:

```
sudo systemctl daemon-reload
sudo systemctl enable --now crio.service
```

Quick test:

```
sudo systemctl is-active crio.service
```

The output is similar to:

```
active
```

Detailed service check:

```
sudo journalctl -f -u crio.service
```

Install network plugins

The `cri-o` installer installs and configures the `cni-plugins` package. You can verify the installation running the following command:

```
/opt/cni/bin/bridge --version
```

The output is similar to:

```
CNI bridge plugin v1.5.1
CNI protocol versions supported: 0.1.0, 0.2.0, 0.3.0, 0.3.1, 0.4.0, 1.0.0
```

To check the default configuration:

```
cat /etc/cni/net.d/11-crio-ipv4-bridge.conflist
```

The output is similar to:

```
{
  "cniVersion": "1.0.0",
  "name": "crio",
  "plugins": [
    {
      "type": "bridge",
      "bridge": "cni0",
      "isGateway": true,
      "ipMasq": true,
      "hairpinMode": true,
      "ipam": {
        "type": "host-local",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ],
        "ranges": [
            [{ "subnet": "10.85.0.0/16" }]
        ]
      }
    }
  ]
}
```

Note:

Make sure that the default `subnet` range (`10.85.0.0/16`) does not overlap with any of your active networks. If there is an overlap, you can edit the file and change it accordingly. Restart the service after the change.

Download and set up the kubelet

Download the latest stable release of the kubelet.

- x86-64
- ARM64

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubelet"
```

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/arm64/kubelet"
```

Configure:

```
sudo mkdir -p /etc/kubernetes/manifests
```

```
sudo tee /etc/kubernetes/kubelet.yaml <<EOF
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  webhook:
    enabled: false # Do NOT use in production clusters!
authorization:
  mode: AlwaysAllow # Do NOT use in production clusters!
enableServer: false
logging:
  format: text
address: 127.0.0.1 # Restrict access to localhost
readOnlyPort: 10255 # Do NOT use in production clusters!
staticPodPath: /etc/kubernetes/manifests
containerRuntimeEndpoint: unix:///var/run/crio/crio.sock
EOF
```

Note:

Because you are not setting up a production cluster, you are using plain HTTP (`readOnlyPort: 10255`) for unauthenticated queries to the kubelet's API.

The *authentication webhook* is disabled and *authorization mode* is set to `AlwaysAllow` for the purpose of this tutorial. You can learn more about authorization modes and webhook authentication to properly configure kubelet in standalone mode in your environment.

See Ports and Protocols to understand which ports Kubernetes components use.

Install:

```
chmod +x kubelet
sudo cp kubelet /usr/bin/
```

Create a `systemd` service unit file:

```
sudo tee /etc/systemd/system/kubelet.service <<EOF
[Unit]
Description=Kubelet

[Service]
ExecStart=/usr/bin/kubelet \
 --config=/etc/kubernetes/kubelet.yaml
Restart=always

[Install]
WantedBy=multi-user.target
EOF
```

The command line argument `--kubeconfig` has been intentionally omitted in the service configuration file. This argument sets the path to a kubeconfig file that specifies how to connect to the API server, enabling API server mode. Omitting it, enables standalone mode.

Enable and start the `kubelet` service:

```
sudo systemctl daemon-reload
sudo systemctl enable --now kubelet.service
```

Quick test:

```
sudo systemctl is-active kubelet.service
```

The output is similar to:

```
active
```

Detailed service check:

```
sudo journalctl -u kubelet.service
```

Check the kubelet's API `/healthz` endpoint:

```
curl http://localhost:10255/healthz?verbose
```

The output is similar to:

```
[+]ping ok
[+]log ok
[+]syncloop ok
healthz check passed
```

Query the kubelet's API `/pods` endpoint:

```
curl http://localhost:10255/pods | jq '.'
```

The output is similar to:

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {},
  "items": null
}
```

Run a Pod in the kubelet

In standalone mode, you can run Pods using Pod manifests. The manifests can either be on the local filesystem, or fetched via HTTP from a configuration source.

Create a manifest for a Pod:

```
cat <<EOF > static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
EOF
```

Copy the `static-web.yaml` manifest file to the `/etc/kubernetes/manifests` directory.

```
sudo cp static-web.yaml /etc/kubernetes/manifests/
```

Find out information about the kubelet and the Pod

The Pod networking plugin creates a network bridge (`cni0`) and a pair of `veth` interfaces for each Pod (one of the pair is inside the newly made Pod, and the other is at the host level).

Query the kubelet's API endpoint at `http://localhost:10255/pods`:

```
curl http://localhost:10255/pods | jq '.'
```

To obtain the IP address of the `static-web` Pod:

```
curl http://localhost:10255/pods | jq '.items[].status.podIP'
```

The output is similar to:

```
"10.85.0.4"
```

Connect to the `nginx` server Pod on `http://<IP>:<Port>` (port 80 is the default), in this case:

```
curl http://10.85.0.4
```

The output is similar to:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Where to look for more details

If you need to diagnose a problem getting this tutorial to work, you can look within the following directories for monitoring and troubleshooting:

```
/var/lib/cni
/var/lib/containers
/var/lib/kubelet

/var/log/containers
/var/log/pods
```

Clean up

kubelet

```
sudo systemctl disable --now kubelet.service
sudo systemctl daemon-reload
sudo rm /etc/systemd/system/kubelet.service
sudo rm /usr/bin/kubelet
sudo rm -rf /etc/kubernetes
sudo rm -rf /var/lib/kubelet
sudo rm -rf /var/log/containers
sudo rm -rf /var/log/pods
```

Container Runtime

```
sudo systemctl disable --now crio.service
sudo systemctl daemon-reload
sudo rm -rf /usr/local/bin
sudo rm -rf /usr/local/lib
sudo rm -rf /usr/local/share
sudo rm -rf /usr/libexec/crio
sudo rm -rf /etc/crio
sudo rm -rf /etc/containers
```

Network Plugins

```
sudo rm -rf /opt/cni
sudo rm -rf /etc/cni
sudo rm -rf /var/lib/cni
```

Conclusion

This page covered the basic aspects of deploying a kubelet in standalone mode. You are now ready to deploy Pods and test additional functionality.

Notice that in standalone mode the kubelet does *not* support fetching Pod configurations from the control plane (because there is no control plane connection).

You also cannot use a ConfigMap or a Secret to configure the containers in a static Pod.

What's next

- Follow Hello, minikube to learn about running Kubernetes *with* a control plane. The minikube tool helps you set up a practice cluster on your own computer.
- Learn more about Network Plugins
- Learn more about Container Runtimes
- Learn more about kubelet
- Learn more about static Pods

Adopting Sidecar Containers

This section is relevant for people adopting a new built-in sidecar containers feature for their workloads.

Sidecar container is not a new concept as posted in the blog post. Kubernetes allows running multiple containers in a Pod to implement this concept. However, running a sidecar container as a regular container has a lot of limitations being fixed with the new built-in sidecar containers support.

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

Objectives

- Understand the need for sidecar containers
- Be able to troubleshoot issues with the sidecar containers
- Understand options to universally "inject" sidecar containers to any workload

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud

- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.29.

To check the version, enter `kubectl version`.

Sidecar containers overview

Sidecar containers are secondary containers that run along with the main application container within the same Pod. These containers are used to enhance or to extend the functionality of the primary *app container* by providing additional services, or functionalities such as logging, monitoring, security, or data synchronization, without directly altering the primary application code. You can read more in the Sidecar containers concept page.

The concept of sidecar containers is not new and there are multiple implementations of this concept. As well as sidecar containers that you, the person defining the Pod, want to run, you can also find that some addons modify Pods - before the Pods start running - so that there are extra sidecar containers. The mechanisms to *inject* those extra sidecars are often mutating webhooks. For example, a service mesh addon might inject a sidecar that configures mutual TLS and encryption in transit between different Pods.

While the concept of sidecar containers is not new, the native implementation of this feature in Kubernetes, however, is new. And as with every new feature, adopting this feature may present certain challenges.

This tutorial explores challenges and solutions that can be experienced by end users as well as by authors of sidecar containers.

Benefits of a built-in sidecar container

Using Kubernetes' native support for sidecar containers provides several benefits:

1. You can configure a native sidecar container to start ahead of init containers.
2. The built-in sidecar containers can be authored to guarantee that they are terminated last. Sidecar containers are terminated with a `SIGTERM` signal once all the regular containers are completed and terminated. If the sidecar container isn’t gracefully shut down, a `SIGKILL` signal will be used to terminate it.
3. With Jobs, when Pod's `restartPolicy: OnFailure` or `restartPolicy: Never`, native sidecar containers do not block Pod completion. With legacy sidecar containers, special care is needed to handle this situation.
4. Also, with Jobs, built-in sidecar containers would keep being restarted once they are done, even if regular containers would not with Pod's `restartPolicy: Never`.

See differences from init containers to learn more about it.

Adopting built-in sidecar containers

The `SidecarContainers` feature gate is in beta state starting from Kubernetes version 1.29 and is enabled by default. Some clusters may have this feature disabled or have software installed that is incompatible with the feature.

When this happens, the Pod may be rejected or the sidecar containers may block Pod startup, rendering the Pod useless. This condition is easy to detect as the Pod simply gets stuck on initialization. However, it is often unclear what caused the problem.

Here are the considerations and troubleshooting steps that one can take while adopting sidecar containers for their workload.

Ensure the feature gate is enabled

As a very first step, make sure that both API server and Nodes are at Kubernetes version v1.29 or later. The feature will break on clusters where Nodes are running earlier versions where it is not enabled.

Note

The feature can be enabled on nodes with the version 1.28. The behavior of built-in sidecar container termination was different in version 1.28, and it is not recommended to adjust the behavior of a sidecar to that behavior. However, if the only concern is the startup order, the above statement can be changed to Nodes running version 1.28 with the feature gate enabled.

You should ensure that the feature gate is enabled for the API server(s) within the control plane **and** for all nodes.

One of the ways to check the feature gate enablement is to run a command like this:

- For API Server:

  ```
  kubectl get --raw /metrics | grep kubernetes_feature_enabled | grep SidecarContainers
  ```

- For the individual node:

  ```
  kubectl get --raw /api/v1/nodes/<node-name>/proxy/metrics | grep kubernetes_feature_enabled | grep SidecarContainers
  ```

If you see something like this:

```
kubernetes_feature_enabled{name="SidecarContainers",stage="BETA"} 1
```

it means that the feature is enabled.

Check for 3rd party tooling and mutating webhooks

If you experience issues when validating the feature, it may be an indication that one of the 3rd party tools or mutating webhooks are broken.

When the `SidecarContainers` feature gate is enabled, Pods gain a new field in their API. Some tools or mutating webhooks might have been built with an earlier version of Kubernetes API.

If tools pass unknown fields as-is using various patching strategies to mutate a Pod object, this will not be a problem. However, there are tools that will strip out unknown fields; if you have those, they must be recompiled with the v1.28+ version of Kubernetes API client code.

The way to check this is to use the `kubectl describe pod` command with your Pod that has passed through mutating admission. If any tools stripped out the new field (`restartPolicy:Always`), you will not see it in the command output.

If you hit an issue like this, please advise the author of the tools or the webhooks use one of the patching strategies for modifying objects instead of a full object update.

Note

Mutating webhook may update Pods based on some conditions. Thus, sidecar containers may work for some Pods and fail for others.

Automatic injection of sidecars

If you are using software that injects sidecars automatically, there are a few possible strategies you may follow to ensure that native sidecar containers can be used. All strategies are generally options you may choose to decide whether the Pod the sidecar will be injected to will land on a Node supporting the feature or not.

As an example, you can follow this conversation in Istio community. The discussion explores the options listed below.

1. Mark Pods that land to nodes supporting sidecars. You can use node labels and node affinity to mark nodes supporting sidecar containers and Pods landing on those nodes.
2. Check Nodes compatibility on injection. During sidecar injection, you may use the following strategies to check node compatibility:
    - query node version and assume the feature gate is enabled on the version 1.29+
    - query node prometheus metrics and check feature enablement status
    - assume the nodes are running with a supported version skew from the API server
    - there may be other custom ways to detect nodes compatibility.
3. Develop a universal sidecar injector. The idea of a universal sidecar injector is to inject a sidecar container as a regular container as well as a native sidecar container. And have a runtime logic to decide which one will work. The universal sidecar injector is wasteful, as it will account for requests twice, but may be considered as a workable solution for special cases.
    - One way would be on start of a native sidecar container detect the node version and exit immediately if the version does not support the sidecar feature.
    - Consider a runtime feature detection design:
        - Define an empty dir so containers can communicate with each other
        - Inject an init container, let's call it `NativeSidecar` with `restartPolicy=Always`.
        - `NativeSidecar` must write a file to an empty directory indicating the first run and exit immediately with exit code `0`.
        - `NativeSidecar` on restart (when native sidecars are supported) checks that file already exists in the empty dir and changes it - indicating that the built-in sidecar containers are supported and running.
        - Inject regular container, let's call it `OldWaySidecar`.
        - `OldWaySidecar` on start checks the presence of a file in an empty dir.
        - If the file indicates that the `NativeSidecar` is NOT running, it assumes that the sidecar feature is not supported and works assuming it is the sidecar.
        - If the file indicates that the `NativeSidecar` is running, it either does nothing and sleeps forever (in the case when Pod’s `restartPolicy=Always`) or exits immediately with exit code `0` (in the case when Pod’s `restartPolicy!=Always`).

What's next

- Learn more about sidecar containers.

Configuring Redis using a ConfigMap

This page provides a real world example of how to configure Redis using a ConfigMap and builds upon the Configure a Pod to Use a ConfigMap task.

Objectives

- Create a ConfigMap with Redis configuration values
- Create a Redis Pod that mounts and uses the created ConfigMap
- Verify that the configuration was correctly applied.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

- The example shown on this page works with `kubectl` 1.14 and above.

- Understand Configure a Pod to Use a ConfigMap.

Real World Example: Configuring Redis using a ConfigMap

Follow the steps below to configure a Redis cache using data stored in a ConfigMap.

First create a ConfigMap with an empty configuration block:

```
cat <<EOF >./example-redis-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-redis-config
data:
  redis-config: ""
EOF
```

Apply the ConfigMap created above, along with a Redis pod manifest:

```
kubectl apply -f example-redis-config.yaml
kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/pods/config/redis-pod.yaml
```

Examine the contents of the Redis pod manifest and note the following:

- A volume named `config` is created by `spec.volumes[1]`
- The `key` and `path` under `spec.volumes[1].configMap.items[0]` exposes the `redis-config` key from the `example-redis-config` ConfigMap as a file named `redis.conf` on the `config` volume.
- The `config` volume is then mounted at `/redis-master` by `spec.containers[0].volumeMounts[1]`.

This has the net effect of exposing the data in `data.redis-config` from the `example-redis-config` ConfigMap above as `/redis-master/redis.conf` inside the Pod.

`pods/config/redis-pod.yaml` Copy pods/config/redis-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: redisspec:  containers:  - name: redis    image: redis:8.0.2    command:      - redis-server      - "/re
```

Examine the created objects:

```
kubectl get pod/redis configmap/example-redis-config
```

You should see the following output:

```
NAME        READY   STATUS    RESTARTS   AGE
pod/redis   1/1     Running   0          8s

NAME                             DATA   AGE
configmap/example-redis-config   1      14s
```

Recall that we left `redis-config` key in the `example-redis-config` ConfigMap blank:

```
kubectl describe configmap/example-redis-config
```

You should see an empty `redis-config` key:

```
Name:         example-redis-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
redis-config:
```

Use `kubectl exec` to enter the pod and run the `redis-cli` tool to check the current configuration:

```
kubectl exec -it pod/redis -- redis-cli
```

Check `maxmemory`:

```
127.0.0.1:6379> CONFIG GET maxmemory
```

It should show the default value of 0:

```
1) "maxmemory"
2) "0"
```

Similarly, check `maxmemory-policy`:

```
127.0.0.1:6379> CONFIG GET maxmemory-policy
```

Which should also yield its default value of `noeviction`:

```
1) "maxmemory-policy"
2) "noeviction"
```

Now let's add some configuration values to the `example-redis-config` ConfigMap:

`pods/config/example-redis-config.yaml` Copy pods/config/example-redis-config.yaml to clipboard

```
apiVersion: v1
kind: ConfigMapmetadata:  name: example-redis-configdata:  redis-config: |    maxmemory 2mb    maxmemory-policy allkeys-lru
```

Apply the updated ConfigMap:

```
kubectl apply -f example-redis-config.yaml
```

Confirm that the ConfigMap was updated:

```
kubectl describe configmap/example-redis-config
```

You should see the configuration values we just added:

```
Name:         example-redis-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
redis-config:
----
maxmemory 2mb
maxmemory-policy allkeys-lru
```

Check the Redis Pod again using `redis-cli` via `kubectl exec` to see if the configuration was applied:

```
kubectl exec -it pod/redis -- redis-cli
```

Check `maxmemory`:

```
127.0.0.1:6379> CONFIG GET maxmemory
```

It remains at the default value of 0:

```
1) "maxmemory"
2) "0"
```

Similarly, `maxmemory-policy` remains at the `noeviction` default setting:

```
127.0.0.1:6379> CONFIG GET maxmemory-policy
```

Returns:

```
1) "maxmemory-policy"
2) "noeviction"
```

The configuration values have not changed because the Pod needs to be restarted to grab updated values from associated ConfigMaps. Let's delete and recreate the Pod:

```
kubectl delete pod redis
kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/pods/config/redis-pod.yaml
```

Now re-check the configuration values one last time:

```
kubectl exec -it pod/redis -- redis-cli
```

Check `maxmemory`:

```
127.0.0.1:6379> CONFIG GET maxmemory
```

It should now return the updated value of 2097152:

```
1) "maxmemory"
2) "2097152"
```

Similarly, `maxmemory-policy` has also been updated:

```
127.0.0.1:6379> CONFIG GET maxmemory-policy
```

It now reflects the desired value of `allkeys-lru`:

```
1) "maxmemory-policy"
2) "allkeys-lru"
```

Clean up your work by deleting the created resources:

```
kubectl delete pod/redis configmap/example-redis-config
```

What's next

- Learn more about ConfigMaps.
- Follow an example of Updating configuration via a ConfigMap.

Updating Configuration via a ConfigMap

This page provides a step-by-step example of updating configuration within a Pod via a ConfigMap and builds upon the [Configure a Pod to Use a ConfigMap](#) task.
At the end of this tutorial, you will understand how to change the configuration for a running application.
This tutorial uses the `alpine` and `nginx` images as examples.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

You need to have the curl command-line tool for making HTTP requests from the terminal or command prompt. If you do not have `curl` available, you can install it. Check the documentation for your local operating system.

Objectives

- Update configuration via a ConfigMap mounted as a Volume
- Update environment variables of a Pod via a ConfigMap
- Update configuration via a ConfigMap in a multi-container Pod
- Update configuration via a ConfigMap in a Pod possessing a Sidecar Container

Update configuration via a ConfigMap mounted as a Volume

Use the `kubectl create configmap` command to create a ConfigMap from literal values:

```
kubectl create configmap sport --from-literal=sport=football
```

Below is an example of a Deployment manifest with the ConfigMap `sport` mounted as a volume into the Pod's only container.

deployments/deployment-with-configmap-as-volume.yaml Copy deployments/deployment-with-configmap-as-volume.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: configmap-volume  labels:    app.kubernetes.io/name: configmap-volumespec:  replicas: 3  selector
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/deployments/deployment-with-configmap-as-volume.yaml
```

Check the pods for this Deployment to ensure they are ready (matching by selector):

```
kubectl get pods --selector=app.kubernetes.io/name=configmap-volume
```

You should see an output similar to:

```
NAME                                READY   STATUS    RESTARTS   AGE
configmap-volume-6b976dfdcf-qxvbm   1/1     Running   0          72s
configmap-volume-6b976dfdcf-skpvm   1/1     Running   0          72s
configmap-volume-6b976dfdcf-tbc6r   1/1     Running   0          72s
```

On each node where one of these Pods is running, the kubelet fetches the data for that ConfigMap and translates it to files in a local volume. The kubelet then mounts that volume into the container, as specified in the Pod template. The code running in that container loads the information from the file and uses it to print a report to stdout. You can check this report by viewing the logs for one of the Pods in that Deployment:

```
# Pick one Pod that belongs to the Deployment, and view its logs
kubectl logs deployments/configmap-volume
```

You should see an output similar to:

```
Found 3 pods, using pod/configmap-volume-76d9c5678f-x5rgj
Thu Jan  4 14:06:46 UTC 2024 My preferred sport is football
Thu Jan  4 14:06:56 UTC 2024 My preferred sport is football
Thu Jan  4 14:07:06 UTC 2024 My preferred sport is football
Thu Jan  4 14:07:16 UTC 2024 My preferred sport is football
Thu Jan  4 14:07:26 UTC 2024 My preferred sport is football
```

Edit the ConfigMap:

```
kubectl edit configmap sport
```

In the editor that appears, change the value of key `sport` from `football` to `cricket`. Save your changes. The kubectl tool updates the ConfigMap accordingly (if you see an error, try again).

Here's an example of how that manifest could look after you edit it:

```
apiVersion: v1
data:  sport: cricketkind: ConfigMap# You can leave the existing metadata as they are.# The values you'll see won't exactly match
```

You should see the following output:

```
configmap/sport edited
```

Tail (follow the latest entries in) the logs of one of the pods that belongs to this Deployment:

```
kubectl logs deployments/configmap-volume --follow
```

After few seconds, you should see the log output change as follows:

```
Thu Jan  4 14:11:36 UTC 2024 My preferred sport is football
Thu Jan  4 14:11:46 UTC 2024 My preferred sport is football
Thu Jan  4 14:11:56 UTC 2024 My preferred sport is football
Thu Jan  4 14:12:06 UTC 2024 My preferred sport is cricket
Thu Jan  4 14:12:16 UTC 2024 My preferred sport is cricket
```

When you have a ConfigMap that is mapped into a running Pod using either a `configMap` volume or a `projected` volume, and you update that ConfigMap, the running Pod sees the update almost immediately.
However, your application only sees the change if it is written to either poll for changes, or watch for file updates.
An application that loads its configuration once at startup will not notice a change.

Note:

The total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the Pod can be as long as kubelet sync period.
Also check [Mounted ConfigMaps are updated automatically](#).

Update environment variables of a Pod via a ConfigMap

Use the `kubectl create configmap` command to create a ConfigMap from [literal values](#):

```
kubectl create configmap fruits --from-literal=fruits=apples
```

Below is an example of a Deployment manifest with an environment variable configured via the ConfigMap `fruits`.

[`deployments/deployment-with-configmap-as-envvar.yaml`](#) Copy deployments/deployment-with-configmap-as-envvar.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: configmap-env-var  labels:    app.kubernetes.io/name: configmap-env-varspec:  replicas: 3  select
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/deployments/deployment-with-configmap-as-envvar.yaml
```

Check the pods for this Deployment to ensure they are ready (matching by [selector](#)):

```
kubectl get pods --selector=app.kubernetes.io/name=configmap-env-var
```

You should see an output similar to:

```
NAME                                 READY   STATUS    RESTARTS   AGE
configmap-env-var-59cfc64f7d-74d7z   1/1     Running   0          46s
configmap-env-var-59cfc64f7d-c4wmj   1/1     Running   0          46s
configmap-env-var-59cfc64f7d-dpr98   1/1     Running   0          46s
```

The key-value pair in the ConfigMap is configured as an environment variable in the container of the Pod. Check this by viewing the logs of one Pod that belongs to the Deployment.

```
kubectl logs deployment/configmap-env-var
```

You should see an output similar to:

```
Found 3 pods, using pod/configmap-env-var-7c994f7769-l74nq
Thu Jan  4 16:07:06 UTC 2024 The basket is full of apples
Thu Jan  4 16:07:16 UTC 2024 The basket is full of apples
Thu Jan  4 16:07:26 UTC 2024 The basket is full of apples
```

Edit the ConfigMap:

```
kubectl edit configmap fruits
```

In the editor that appears, change the value of key `fruits` from `apples` to `mangoes`. Save your changes. The kubectl tool updates the ConfigMap accordingly (if you see an error, try again).

Here's an example of how that manifest could look after you edit it:

```
apiVersion: v1
data:  fruits: mangoeskind: ConfigMap# You can leave the existing metadata as they are.# The values you'll see won't exactly match
```

You should see the following output:

```
configmap/fruits edited
```

Tail the logs of the Deployment and observe the output for few seconds:

```
# As the text explains, the output does NOT change
kubectl logs deployments/configmap-env-var --follow
```

Notice that the output remains **unchanged**, even though you edited the ConfigMap:

```
Thu Jan  4 16:12:56 UTC 2024 The basket is full of apples
Thu Jan  4 16:13:06 UTC 2024 The basket is full of apples
Thu Jan  4 16:13:16 UTC 2024 The basket is full of apples
Thu Jan  4 16:13:26 UTC 2024 The basket is full of apples
```

Note:

Although the value of the key inside the ConfigMap has changed, the environment variable in the Pod still shows the earlier value. This is because environment variables for a process running inside a Pod are **not** updated when the source data changes; if you wanted to force an update, you would need to have Kubernetes replace your existing Pods. The new Pods would then run with the updated information.

You can trigger that replacement. Perform a rollout for the Deployment, using `kubectl rollout`:

```
# Trigger the rollout
kubectl rollout restart deployment configmap-env-var

# Wait for the rollout to complete
kubectl rollout status deployment configmap-env-var --watch=true
```

Next, check the Deployment:

```
kubectl get deployment configmap-env-var
```

You should see an output similar to:

```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
configmap-env-var   3/3     3            3           12m
```

Check the Pods:

```
kubectl get pods --selector=app.kubernetes.io/name=configmap-env-var
```

The rollout causes Kubernetes to make a new ReplicaSet for the Deployment; that means the existing Pods eventually terminate, and new ones are created. After few seconds, you should see an output similar to:

```
NAME                                 READY   STATUS        RESTARTS   AGE
configmap-env-var-6d94d89bf5-2ph2l   1/1     Running       0          13s
configmap-env-var-6d94d89bf5-74twx   1/1     Running       0          8s
configmap-env-var-6d94d89bf5-d5vx8   1/1     Running       0          11s
```

Note:

Please wait for the older Pods to fully terminate before proceeding with the next steps.

View the logs for a Pod in this Deployment:

```
# Pick one Pod that belongs to the Deployment, and view its logs
kubectl logs deployment/configmap-env-var
```

You should see an output similar to the below:

```
Found 3 pods, using pod/configmap-env-var-6d9ff89fb6-bzcf6
Thu Jan  4 16:30:35 UTC 2024 The basket is full of mangoes
Thu Jan  4 16:30:45 UTC 2024 The basket is full of mangoes
Thu Jan  4 16:30:55 UTC 2024 The basket is full of mangoes
```

This demonstrates the scenario of updating environment variables in a Pod that are derived from a ConfigMap. Changes to the ConfigMap values are applied to the Pod during the subsequent rollout. If Pods get created for another reason, such as scaling up the Deployment, then the new Pods also use the latest configuration values; if you don't trigger a rollout, then you might find that your app is running with a mix of old and new environment variable values.

Update configuration via a ConfigMap in a multi-container Pod

Use the `kubectl create configmap` command to create a ConfigMap from literal values:

```
kubectl create configmap color --from-literal=color=red
```

Below is an example manifest for a Deployment that manages a set of Pods, each with two containers. The two containers share an `emptyDir` volume that they use to communicate. The first container runs a web server (`nginx`). The mount path for the shared volume in the web server container is `/usr/share/nginx/html`. The second helper container is based on `alpine`, and for this container the `emptyDir` volume is mounted at `/pod-data`. The helper container writes a file in HTML that has its content based on a ConfigMap. The web server container serves the HTML via HTTP.

`deployments/deployment-with-configmap-two-containers.yaml` Copy deployments/deployment-with-configmap-two-containers.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: configmap-two-containers  labels:    app.kubernetes.io/name: configmap-two-containersspec:  repli
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/deployments/deployment-with-configmap-two-containers.yaml
```

Check the pods for this Deployment to ensure they are ready (matching by selector):

```
kubectl get pods --selector=app.kubernetes.io/name=configmap-two-containers
```

You should see an output similar to:

```
NAME                                        READY   STATUS    RESTARTS   AGE
configmap-two-containers-565fb6d4f4-2xhxf   2/2     Running   0          20s
```

```
configmap-two-containers-565fb6d4f4-g5v4j   2/2     Running   0          20s
configmap-two-containers-565fb6d4f4-mzsmf   2/2     Running   0          20s
```

Expose the Deployment (the `kubectl` tool creates a Service for you):

```
kubectl expose deployment configmap-two-containers --name=configmap-service --port=8080 --target-port=80
```

Use `kubectl` to forward the port:

```
# this stays running in the background
kubectl port-forward service/configmap-service 8080:8080 &
```

Access the service.

```
curl http://localhost:8080
```

You should see an output similar to:

```
Fri Jan  5 08:08:22 UTC 2024 My preferred color is red
```

Edit the ConfigMap:

```
kubectl edit configmap color
```

In the editor that appears, change the value of key `color` from `red` to `blue`. Save your changes. The kubectl tool updates the ConfigMap accordingly (if you see an error, try again).

Here's an example of how that manifest could look after you edit it:

```
apiVersion: v1
data:  color: bluekind: ConfigMap# You can leave the existing metadata as they are.# The values you'll see won't exactly match the
```

Loop over the service URL for few seconds.

```
# Cancel this when you're happy with it (Ctrl-C)
while true; do curl --connect-timeout 7.5 http://localhost:8080; sleep 10; done
```

You should see the output change as follows:

```
Fri Jan  5 08:14:00 UTC 2024 My preferred color is red
Fri Jan  5 08:14:02 UTC 2024 My preferred color is red
Fri Jan  5 08:14:20 UTC 2024 My preferred color is red
Fri Jan  5 08:14:22 UTC 2024 My preferred color is red
Fri Jan  5 08:14:32 UTC 2024 My preferred color is blue
Fri Jan  5 08:14:43 UTC 2024 My preferred color is blue
Fri Jan  5 08:15:00 UTC 2024 My preferred color is blue
```

Update configuration via a ConfigMap in a Pod possessing a sidecar container

The above scenario can be replicated by using a Sidecar Container as a helper container to write the HTML file.
As a Sidecar Container is conceptually an Init Container, it is guaranteed to start before the main web server container.
This ensures that the HTML file is always available when the web server is ready to serve it.

If you are continuing from the previous scenario, you can reuse the ConfigMap named `color` for this scenario.
If you are executing this scenario independently, use the `kubectl create configmap` command to create a ConfigMap from literal values:

```
kubectl create configmap color --from-literal=color=blue
```

Below is an example manifest for a Deployment that manages a set of Pods, each with a main container and a sidecar container. The two containers share an `emptyDir` volume that they use to communicate. The main container runs a web server (NGINX). The mount path for the shared volume in the web server container is `/usr/share/nginx/html`. The second container is a Sidecar Container based on Alpine Linux which acts as a helper container. For this container the `emptyDir` volume is mounted at `/pod-data`. The Sidecar Container writes a file in HTML that has its content based on a ConfigMap. The web server container serves the HTML via HTTP.

`deployments/deployment-with-configmap-and-sidecar-container.yaml`
Copy deployments/deployment-with-configmap-and-sidecar-container.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: configmap-sidecar-container  labels:    app.kubernetes.io/name: configmap-sidecar-containerspec:
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/deployments/deployment-with-configmap-and-sidecar-container.yaml
```

Check the pods for this Deployment to ensure they are ready (matching by selector):

```
kubectl get pods --selector=app.kubernetes.io/name=configmap-sidecar-container
```

You should see an output similar to:

```
NAME                                           READY   STATUS    RESTARTS   AGE
configmap-sidecar-container-5fb59f558b-87rp7   2/2     Running   0          94s
configmap-sidecar-container-5fb59f558b-ccs7s   2/2     Running   0          94s
configmap-sidecar-container-5fb59f558b-wnmgk   2/2     Running   0          94s
```

Expose the Deployment (the `kubectl` tool creates a Service for you):

```
kubectl expose deployment configmap-sidecar-container --name=configmap-sidecar-service --port=8081 --target-port=80
```

Use `kubectl` to forward the port:

```
# this stays running in the background
kubectl port-forward service/configmap-sidecar-service 8081:8081 &
```

Access the service.

```
curl http://localhost:8081
```

You should see an output similar to:

```
Sat Feb 17 13:09:05 UTC 2024 My preferred color is blue
```

Edit the ConfigMap:

```
kubectl edit configmap color
```

In the editor that appears, change the value of key `color` from `blue` to `green`. Save your changes. The kubectl tool updates the ConfigMap accordingly (if you see an error, try again).

Here's an example of how that manifest could look after you edit it:

```
apiVersion: v1
data:  color: greenkind: ConfigMap# You can leave the existing metadata as they are.# The values you'll see won't exactly match th
```

Loop over the service URL for few seconds.

```
# Cancel this when you're happy with it (Ctrl-C)
while true; do curl --connect-timeout 7.5 http://localhost:8081; sleep 10; done
```

You should see the output change as follows:

```
Sat Feb 17 13:12:35 UTC 2024 My preferred color is blue
Sat Feb 17 13:12:45 UTC 2024 My preferred color is blue
Sat Feb 17 13:12:55 UTC 2024 My preferred color is blue
Sat Feb 17 13:13:05 UTC 2024 My preferred color is blue
Sat Feb 17 13:13:15 UTC 2024 My preferred color is green
Sat Feb 17 13:13:25 UTC 2024 My preferred color is green
Sat Feb 17 13:13:35 UTC 2024 My preferred color is green
```

Update configuration via an immutable ConfigMap that is mounted as a volume

Note:

Immutable ConfigMaps are especially used for configuration that is constant and is **not** expected to change over time. Marking a ConfigMap as immutable allows a performance improvement where the kubelet does not watch for changes.

If you do need to make a change, you should plan to either:

- change the name of the ConfigMap, and switch to running Pods that reference the new name
- replace all the nodes in your cluster that have previously run a Pod that used the old value
- restart the kubelet on any node where the kubelet previously loaded the old ConfigMap

An example manifest for an Immutable ConfigMap is shown below.

configmap/immutable-configmap.yaml Copy configmap/immutable-configmap.yaml to clipboard

```
apiVersion: v1
data:  company_name: "ACME, Inc." # existing fictional company namekind: ConfigMapimmutable: truemetadata:  name: company-name-201
```

Create the Immutable ConfigMap:

```
kubectl apply -f https://k8s.io/examples/configmap/immutable-configmap.yaml
```

Below is an example of a Deployment manifest with the Immutable ConfigMap `company-name-20150801` mounted as a volume into the Pod's only container.

deployments/deployment-with-immutable-configmap-as-volume.yaml
Copy deployments/deployment-with-immutable-configmap-as-volume.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: immutable-configmap-volume  labels:    app.kubernetes.io/name: immutable-configmap-volumespec:  r
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/deployments/deployment-with-immutable-configmap-as-volume.yaml
```

Check the pods for this Deployment to ensure they are ready (matching by selector):

```
kubectl get pods --selector=app.kubernetes.io/name=immutable-configmap-volume
```

You should see an output similar to:

```
NAME                                          READY   STATUS    RESTARTS   AGE
immutable-configmap-volume-78b6fbff95-5gsfh   1/1     Running   0          62s
immutable-configmap-volume-78b6fbff95-7vcj4   1/1     Running   0          62s
immutable-configmap-volume-78b6fbff95-vdslm   1/1     Running   0          62s
```

The Pod's container refers to the data defined in the ConfigMap and uses it to print a report to stdout. You can check this report by viewing the logs for one of the Pods in that Deployment:

```
# Pick one Pod that belongs to the Deployment, and view its logs
kubectl logs deployments/immutable-configmap-volume
```

You should see an output similar to:

```
Found 3 pods, using pod/immutable-configmap-volume-78b6fbff95-5gsfh
Wed Mar 20 03:52:34 UTC 2024 The name of the company is ACME, Inc.
Wed Mar 20 03:52:44 UTC 2024 The name of the company is ACME, Inc.
Wed Mar 20 03:52:54 UTC 2024 The name of the company is ACME, Inc.
```

Note:

Once a ConfigMap is marked as immutable, it is not possible to revert this change nor to mutate the contents of the data or the binaryData field.
In order to modify the behavior of the Pods that use this configuration, you will create a new immutable ConfigMap and edit the Deployment to define a slightly different pod template, referencing the new ConfigMap.

Create a new immutable ConfigMap by using the manifest shown below:

configmap/new-immutable-configmap.yaml Copy configmap/new-immutable-configmap.yaml to clipboard

```
apiVersion: v1
data:  company_name: "Fiktivesunternehmen GmbH" # new fictional company namekind: ConfigMapimmutable: truemetadata:  name: company
```

```
kubectl apply -f https://k8s.io/examples/configmap/new-immutable-configmap.yaml
```

You should see an output similar to:

```
configmap/company-name-20240312 created
```

Check the newly created ConfigMap:

```
kubectl get configmap
```

You should see an output displaying both the old and new ConfigMaps:

```
NAME                    DATA   AGE
company-name-20150801   1      22m
company-name-20240312   1      24s
```

Modify the Deployment to reference the new ConfigMap.

Edit the Deployment:

```
kubectl edit deployment immutable-configmap-volume
```

In the editor that appears, update the existing volume definition to use the new ConfigMap.

```
volumes:
- configMap:    defaultMode: 420    name: company-name-20240312 # Update this field  name: config-volume
```

You should see the following output:

```
deployment.apps/immutable-configmap-volume edited
```

This will trigger a rollout. Wait for all the previous Pods to terminate and the new Pods to be in a ready state.

Monitor the status of the Pods:

```
kubectl get pods --selector=app.kubernetes.io/name=immutable-configmap-volume
```

```
NAME                                          READY   STATUS        RESTARTS   AGE
immutable-configmap-volume-5fdb88fcc8-29v8n   1/1     Running       0          13s
immutable-configmap-volume-5fdb88fcc8-52ddd   1/1     Running       0          14s
immutable-configmap-volume-5fdb88fcc8-n5jx4   1/1     Running       0          15s
immutable-configmap-volume-78b6fbff95-5gsfh   1/1     Terminating   0          32m
immutable-configmap-volume-78b6fbff95-7vcj4   1/1     Terminating   0          32m
immutable-configmap-volume-78b6fbff95-vdslm   1/1     Terminating   0          32m
```

You should eventually see an output similar to:

```
NAME                                          READY   STATUS    RESTARTS   AGE
immutable-configmap-volume-5fdb88fcc8-29v8n   1/1     Running   0          43s
immutable-configmap-volume-5fdb88fcc8-52ddd   1/1     Running   0          44s
immutable-configmap-volume-5fdb88fcc8-n5jx4   1/1     Running   0          45s
```

View the logs for a Pod in this Deployment:

```
# Pick one Pod that belongs to the Deployment, and view its logs
kubectl logs deployment/immutable-configmap-volume
```

You should see an output similar to the below:

```
Found 3 pods, using pod/immutable-configmap-volume-5fdb88fcc8-n5jx4
Wed Mar 20 04:24:17 UTC 2024 The name of the company is Fiktivesunternehmen GmbH
Wed Mar 20 04:24:27 UTC 2024 The name of the company is Fiktivesunternehmen GmbH
Wed Mar 20 04:24:37 UTC 2024 The name of the company is Fiktivesunternehmen GmbH
```

Once all the deployments have migrated to use the new immutable ConfigMap, it is advised to delete the old one.

```
kubectl delete configmap company-name-20150801
```

Summary

Changes to a ConfigMap mounted as a Volume on a Pod are available seamlessly after the subsequent kubelet sync.

Changes to a ConfigMap that configures environment variables for a Pod are available after the subsequent rollout for the Pod.

Once a ConfigMap is marked as immutable, it is not possible to revert this change (you cannot make an immutable ConfigMap mutable), and you also cannot make any change to the contents of the `data` or the `binaryData` field. You can delete and recreate the ConfigMap, or you can make a new different ConfigMap. When you delete a ConfigMap, running containers and their Pods maintain a mount point to any volume that referenced that existing ConfigMap.

Cleaning up

Terminate the `kubectl port-forward` commands in case they are running.

Delete the resources created during the tutorial:

```
kubectl delete deployment configmap-volume configmap-env-var configmap-two-containers configmap-sidecar-container immutable-config
kubectl delete service configmap-service configmap-sidecar-service
kubectl delete configmap sport fruits color company-name-20240312

kubectl delete configmap company-name-20150801 # In case it was not handled during the task execution
```

Namespaces Walkthrough

Kubernetes namespaces help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

This example demonstrates how to use Kubernetes namespaces to subdivide your cluster.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

Prerequisites

This example assumes the following:

1. You have an existing Kubernetes cluster.
2. You have a basic understanding of Kubernetes Pods, Services, and Deployments.

Understand the default namespace

By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster.

Assuming you have a fresh cluster, you can inspect the available namespaces by doing the following:

```
kubectl get namespaces
```

```
NAME      STATUS    AGE
default   Active    13m
```

Create new namespaces

For this exercise, we will create two additional Kubernetes namespaces to hold our content.

Let's imagine a scenario where an organization is using a shared Kubernetes cluster for development and production use cases.

The development team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.

The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: `development` and `production`.

Let's create two new namespaces to hold our work.

Use the file `namespace-dev.yaml` which describes a `development` namespace:

`admin/namespace-dev.yaml` Copy admin/namespace-dev.yaml to clipboard

```
apiVersion: v1
kind: Namespacemetadata:  name: development  labels:    name: development
```

Create the `development` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.yaml
```

Save the following contents into file `namespace-prod.yaml` which describes a `production` namespace:

`admin/namespace-prod.yaml` Copy admin/namespace-prod.yaml to clipboard

```
apiVersion: v1
kind: Namespacemetadata:  name: production  labels:    name: production
```

And then let's create the `production` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.yaml
```

To be sure things are right, let's list all of the namespaces in our cluster.

```
kubectl get namespaces --show-labels
```

```
NAME          STATUS    AGE       LABELS
default       Active    32m       <none>
development   Active    29s       name=development
production    Active    23s       name=production
```

Create pods in each namespace

A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster.

Users interacting with one namespace do not see the content in another namespace.

To demonstrate this, let's spin up a simple Deployment and Pods in the `development` namespace.

We first check what is the current context:

```
kubectl config view
```

```
apiVersion: v1
clusters:- cluster:    certificate-authority-data: REDACTED    server: https://130.211.122.180  name: lithe-cocoa-92103_kubernetes
```

```
kubectl config current-context
```

```
lithe-cocoa-92103_kubernetes
```

The next step is to define a context for the kubectl client to work in each namespace. The value of "cluster" and "user" fields are copied from the current context.

```
kubectl config set-context dev --namespace=development \
  --cluster=lithe-cocoa-92103_kubernetes \  --user=lithe-cocoa-92103_kuberneteskubectl config set-context prod --namespace=product
```

By default, the above commands add two contexts that are saved into file `.kube/config`. You can now view the contexts and alternate against the two new request contexts depending on which namespace you wish to work against.

To view the new contexts:

```
kubectl config view
```

```
apiVersion: v1
clusters:- cluster:    certificate-authority-data: REDACTED    server: https://130.211.122.180  name: lithe-cocoa-92103_kubernetes
```

Let's switch to operate in the `development` namespace.

```
kubectl config use-context dev
```

You can verify your current context by doing the following:

```
kubectl config current-context
```

```
dev
```

At this point, all requests we make to the Kubernetes cluster from the command line are scoped to the `development` namespace.

Let's create some contents.

admin/snowflake-deployment.yaml Copy admin/snowflake-deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  labels:    app: snowflake  name: snowflakespec:  replicas: 2  selector:    matchLabels:      app: snowf
```

Apply the manifest to create a Deployment

```
kubectl apply -f https://k8s.io/examples/admin/snowflake-deployment.yaml
```

We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
snowflake    2/2     2            2           2m
```

```
kubectl get pods -l app=snowflake
```

```
NAME                         READY     STATUS    RESTARTS   AGE
snowflake-3968820950-9dgr8   1/1       Running   0          2m
snowflake-3968820950-vgc4n   1/1       Running   0          2m
```

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the `production` namespace.

Let's switch to the `production` namespace and show how resources in one namespace are hidden from the other.

```
kubectl config use-context prod
```

The `production` namespace should be empty, and the following commands should return nothing.

```
kubectl get deployment
kubectl get pods
```

Production likes to run cattle, so let's create some cattle pods.

```
kubectl create deployment cattle --image=registry.k8s.io/serve_hostname --replicas=5
```

```
kubectl get deployment
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
cattle       5/5     5            5           10s
```

```
kubectl get pods -l app=cattle
```

```
NAME                      READY     STATUS    RESTARTS   AGE
cattle-2263376956-41xy6   1/1       Running   0          34s
cattle-2263376956-kw466   1/1       Running   0          34s
cattle-2263376956-n4v97   1/1       Running   0          34s
cattle-2263376956-p5p3i   1/1       Running   0          34s
cattle-2263376956-sxpth   1/1       Running   0          34s
```

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

Configuring swap memory on Kubernetes nodes

This page provides an example of how to provision and configure swap memory on a Kubernetes node using kubeadm.

Objectives

- Provision swap memory on a Kubernetes node using kubeadm.
- Learn to configure both encrypted and unencrypted swap.
- Learn to enable swap on boot.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.33.

To check the version, enter `kubectl version`.

You need at least one worker node in your cluster which needs to run a Linux operating system. It is required for this demo that the kubeadm tool be installed, following the steps outlined in the kubeadm installation guide.

On each worker node where you will configure swap use, you need:

- `fallocate`
- `mkswap`
- `swapon`
- For encrypted swap space (recommended), you also need:
- `cryptsetup`

Install a swap-enabled cluster with kubeadm

Create a swap file and turn swap on

If swap is not enabled, there's a need to provision swap on the node. The following sections demonstrate creating 4GiB of swap, both in the encrypted and unencrypted case.

- Setting up encrypted swap
- Setting up unencrypted swap

An encrypted swap file can be set up as follows. Bear in mind that this example uses the `cryptsetup` binary (which is available on most Linux distributions).

```
# Allocate storage and restrict access
fallocate --length 4GiB /swapfile
chmod 600 /swapfile

# Create an encrypted device backed by the allocated storage
cryptsetup --type plain --cipher aes-xts-plain64 --key-size 256 -d /dev/urandom open /swapfile cryptswap

# Format the swap space
mkswap /dev/mapper/cryptswap

# Activate the swap space for paging
swapon /dev/mapper/cryptswap
```

An unencrypted swap file can be set up as follows.

```
# Allocate storage and restrict access
fallocate --length 4GiB /swapfile
chmod 600 /swapfile

# Format the swap space
mkswap /swapfile

# Activate the swap space for paging
swapon /swapfile
```

Verify that swap is enabled

Swap can be verified to be enabled with both `swapon -s` command or the `free` command.

Using `swapon -s`:

```
Filename       Type            Size            Used            Priority
/dev/dm-0      partition       4194300         0               -2
```

Using `free -h`:

```
               total        used        free      shared  buff/cache   available
Mem:           3.8Gi       1.3Gi       249Mi        25Mi       2.5Gi       2.5Gi
Swap:          4.0Gi          0B       4.0Gi
```

Enable swap on boot

After setting up swap, to start the swap file at boot time, you typically either set up a systemd unit to activate (encrypted) swap, or you add a line similar to `/swapfile swap swap defaults 0 0` into `/etc/fstab`.

Using systemd for swap activation allows the system to delay kubelet start until swap is available, if that is something you want to ensure. In a similar way, using systemd allows your server to leave swap active until kubelet (and, typically, your container runtime) have shut down.

Set up kubelet configuration

After enabling swap on the node, kubelet needs to be configured to use it. You need to select a swap behavior for this node. You'll configure *LimitedSwap* behavior for this tutorial.

Find and edit the kubelet configuration file, and:

- set `failSwapOn` to false
- set `memorySwap.swapBehavior` to LimitedSwap

```
# this fragment goes into the kubelet's configuration file
failSwapOn: false
memorySwap:
    swapBehavior: LimitedSwap
```

In order for these configurations to take effect, kubelet needs to be restarted. Typically you do that by running:

```
systemctl restart kubelet.service
```

You should find that the kubelet is now healthy, and that you can run Pods that use swap memory as needed.

Install Drivers and Allocate Devices with DRA

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

This tutorial shows you how to install Dynamic Resource Allocation (DRA) drivers in your cluster and how to use them in conjunction with the DRA APIs to allocate devices to Pods. This page is intended for cluster administrators.

Dynamic Resource Allocation (DRA) lets a cluster manage availability and allocation of hardware resources to satisfy Pod-based claims for hardware requirements and preferences. To support this, a mixture of Kubernetes built-in components (like the Kubernetes scheduler, kubelet, and kube-controller-manager) and third-party drivers from device owners (called DRA drivers) share the responsibility to advertise, allocate, prepare, mount, healthcheck, unprepare, and cleanup resources throughout the Pod lifecycle. These components share information via a series of DRA specific APIs in the `resource.k8s.io` API group including DeviceClasses, ResourceSlices, ResourceClaims, as well as new fields in the Pod spec itself.

Objectives

- Deploy an example DRA driver
- Deploy a Pod requesting a hardware claim using DRA APIs
- Delete a Pod that has a claim

Before you begin

Your cluster should support RBAC. You can try this tutorial with a cluster using a different authorization mechanism, but in that case you will have to adapt the steps around defining roles and permissions.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

This tutorial has been tested with Linux nodes, though it may also work with other types of nodes.

Your Kubernetes server must be version v1.34.

To check the version, enter `kubectl version`.

If your cluster is not currently running Kubernetes 1.34 then please check the documentation for the version of Kubernetes that you plan to use.

Explore the initial cluster state

You can spend some time to observe the initial state of a cluster with DRA enabled, especially if you have not used these APIs extensively before. If you set up a new cluster for this tutorial, with no driver installed and no Pod claims yet to satisfy, the output of these commands won't show any resources.

1. Get a list of DeviceClasses:

   ```
   kubectl get deviceclasses
   ```

   The output is similar to this:

   ```
   No resources found
   ```

2. Get a list of ResourceSlices:

   ```
   kubectl get resourceslices
   ```

   The output is similar to this:

   ```
   No resources found
   ```

3. Get a list of ResourceClaims and ResourceClaimTemplates

   ```
   kubectl get resourceclaims -A
   kubectl get resourceclaimtemplates -A
   ```

   The output is similar to this:

   ```
   No resources found
   No resources found
   ```

At this point, you have confirmed that DRA is enabled and configured properly in the cluster, and that no DRA drivers have advertised any resources to the DRA APIs yet.

Install an example DRA driver

DRA drivers are third-party applications that run on each node of your cluster to interface with the hardware of that node and Kubernetes' built-in DRA components. The installation procedure depends on the driver you choose, but is likely deployed as a DaemonSet to all or a selection of the nodes (using selectors or similar mechanisms) in your cluster.

Check your driver's documentation for specific installation instructions, which might include a Helm chart, a set of manifests, or other deployment tooling.

This tutorial uses an example driver which can be found in the kubernetes-sigs/dra-example-driver repository to demonstrate driver installation. This example driver advertises simulated GPUs to Kubernetes for your Pods to interact with.

Prepare your cluster for driver installation

To simplify cleanup, create a namespace named dra-tutorial:

1. Create the namespace:

   ```
   kubectl create namespace dra-tutorial
   ```

In a production environment, you would likely be using a previously released or qualified image from the driver vendor or your own organization, and your nodes would need to have access to the image registry where the driver image is hosted. In this tutorial, you will use a publicly released image of the dra-example-driver to simulate access to a DRA driver image.

1. Confirm your nodes have access to the image by running the following from within one of your cluster's nodes:

   ```
   docker pull registry.k8s.io/dra-example-driver/dra-example-driver:v0.2.0
   ```

Deploy the DRA driver components

For this tutorial, you will install the critical example resource driver components individually with `kubectl`.

1. Create the DeviceClass representing the device types this DRA driver supports:

   dra/driver-install/deviceclass.yaml Copy dra/driver-install/deviceclass.yaml to clipboard

   ```
   apiVersion: resource.k8s.io/v1
   kind: DeviceClassmetadata:  name: gpu.example.comspec:  selectors:  - cel:      expression: "device.driver == 'gpu.example.c
   ```

   ```
   kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/deviceclass.yaml
   ```

2. Create the ServiceAccount, ClusterRole and ClusterRoleBinding that will be used by the driver to gain permissions to interact with the Kubernetes API on this cluster:

   1. Create the Service Account:

      dra/driver-install/serviceaccount.yaml Copy dra/driver-install/serviceaccount.yaml to clipboard

      ```
      apiVersion: v1
      kind: ServiceAccountmetadata:  name: dra-example-driver-service-account  namespace: dra-tutorial  labels:    app.kuberne
      ```

      ```
      kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/serviceaccount.yaml
      ```

   2. Create the ClusterRole:

      dra/driver-install/clusterrole.yaml Copy dra/driver-install/clusterrole.yaml to clipboard

      ```
      apiVersion: rbac.authorization.k8s.io/v1
      kind: ClusterRolemetadata:  name: dra-example-driver-rolerules:- apiGroups: ["resource.k8s.io"]  resources: ["resourcecl
      ```

      ```
      kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/clusterrole.yaml
      ```

   3. Create the ClusterRoleBinding:

      dra/driver-install/clusterrolebinding.yaml Copy dra/driver-install/clusterrolebinding.yaml to clipboard

      ```
      apiVersion: rbac.authorization.k8s.io/v1
      kind: ClusterRoleBindingmetadata:  name: dra-example-driver-role-bindingsubjects:- kind: ServiceAccount  name: dra-examp
      ```

      ```
      kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/clusterrolebinding.yaml
      ```

3. Create a PriorityClass for the DRA driver. The PriorityClass prevents preemption of th DRA driver component, which is responsible for important lifecycle operations for Pods with claims. Learn more about pod priority and preemption here.

   dra/driver-install/priorityclass.yaml Copy dra/driver-install/priorityclass.yaml to clipboard

   ```
   apiVersion: scheduling.k8s.io/v1
   kind: PriorityClassmetadata:  name: dra-driver-high-priorityvalue: 1000000globalDefault: falsedescription: "This priority cla
   ```

   ```
   kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/priorityclass.yaml
   ```

4. Deploy the actual DRA driver as a DaemonSet configured to run the example driver binary with the permissions provisioned above. The DaemonSet has the permissions that you granted to the ServiceAccount in the previous steps.

   dra/driver-install/daemonset.yaml Copy dra/driver-install/daemonset.yaml to clipboard

   ```
   apiVersion: apps/v1
   kind: DaemonSetmetadata:  name: dra-example-driver-kubeletplugin  namespace: dra-tutorial  labels:    app.kubernetes.io/name:
   ```

```
kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/daemonset.yaml
```

The DaemonSet is configured with the volume mounts necessary to interact with the underlying Container Device Interface (CDI) directory, and to expose its socket to `kubelet` via the `kubelet/plugins` directory.

Verify the DRA driver installation

1. Get a list of the Pods of the DRA driver DaemonSet across all worker nodes:

   ```
   kubectl get pod -l app.kubernetes.io/name=dra-example-driver -n dra-tutorial
   ```

   The output is similar to this:

   ```
   NAME                                     READY   STATUS    RESTARTS   AGE
   dra-example-driver-kubeletplugin-4sk2x   1/1     Running   0          13s
   dra-example-driver-kubeletplugin-cttr2   1/1     Running   0          13s
   ```

2. The initial responsibility of each node's local DRA driver is to update the cluster with what devices are available to Pods on that node, by publishing its metadata to the ResourceSlices API. You can check that API to see that each node with a driver is advertising the device class it represents.

   Check for available ResourceSlices:

   ```
   kubectl get resourceslices
   ```

   The output is similar to this:

   ```
   NAME                                 NODE           DRIVER            POOL           AGE
   kind-worker-gpu.example.com-k69gd    kind-worker    gpu.example.com   kind-worker    19s
   kind-worker2-gpu.example.com-qdgpn   kind-worker2   gpu.example.com   kind-worker2   19s
   ```

At this point, you have successfully installed the example DRA driver, and confirmed its initial configuration. You're now ready to use DRA to schedule Pods.

Claim resources and deploy a Pod

To request resources using DRA, you create ResourceClaims or ResourceClaimTemplates that define the resources that your Pods need. In the example driver, a memory capacity attribute is exposed for mock GPU devices. This section shows you how to use [Common Expression Language](#) to express your requirements in a ResourceClaim, select that ResourceClaim in a Pod specification, and observe the resource allocation.

This tutorial showcases only one basic example of a DRA ResourceClaim. Read [Dynamic Resource Allocation](#) to learn more about ResourceClaims.

Create the ResourceClaim

In this section, you create a ResourceClaim and reference it in a Pod. Whatever the claim, the `deviceClassName` is a required field, narrowing down the scope of the request to a specific device class. The request itself can include a [Common Expression Language](#) expression that references attributes that may be advertised by the driver managing that device class.

In this example, you will create a request for any GPU advertising over 10Gi memory capacity. The attribute exposing capacity from the example driver takes the form `device.capacity['gpu.example.com'].memory`. Note also that the name of the claim is set to `some-gpu`.

[dra/driver-install/example/resourceclaim.yaml](#) Copy dra/driver-install/example/resourceclaim.yaml to clipboard

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimmetadata: name: some-gpu namespace: dra-tutorialspec:   devices:     requests:     - name: some-gpu       exact
```

```
kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/example/resourceclaim.yaml
```

Create the Pod that references that ResourceClaim

Below is the Pod manifest referencing the ResourceClaim you just made, `some-gpu`, in the `spec.resourceClaims.resourceClaimName` field. The local name for that claim, `gpu`, is then used in the `spec.containers.resources.claims.name` field to allocate the claim to the Pod's underlying container.

[dra/driver-install/example/pod.yaml](#) Copy dra/driver-install/example/pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: pod0  namespace: dra-tutorial  labels:    app: podspec:  containers:  - name: ctr0    image: ubuntu:24.0
```

```
kubectl apply --server-side -f http://k8s.io/examples/dra/driver-install/example/pod.yaml
```

1. Confirm the pod has deployed:

   ```
   kubectl get pod pod0 -n dra-tutorial
   ```

   The output is similar to this:

   ```
   NAME   READY   STATUS    RESTARTS   AGE
   pod0   1/1     Running   0          9s
   ```

Explore the DRA state

After you create the Pod, the cluster tries to schedule that Pod to a node where Kubernetes can satisfy the ResourceClaim. In this tutorial, the DRA driver is deployed on all nodes, and is advertising mock GPUs on all nodes, all of which have enough capacity advertised to satisfy the Pod's claim, so Kubernetes can schedule this Pod on any node and can allocate any of the mock GPUs on that node.

When Kubernetes allocates a mock GPU to a Pod, the example driver adds environment variables in each container it is allocated to in order to indicate which GPUs *would* have been injected into them by a real resource driver and how they would have been configured, so you can check those environment variables to see how the Pods have been handled by the system.

1. Check the Pod logs, which report the name of the mock GPU that was allocated:

   ```
   kubectl logs pod0 -c ctr0 -n dra-tutorial | grep -E "GPU_DEVICE_[0-9]+=" | grep -v "RESOURCE_CLAIM"
   ```

   The output is similar to this:

   ```
   declare -x GPU_DEVICE_0="gpu-0"
   ```

2. Check the state of the ResourceClaim object:

   ```
   kubectl get resourceclaims -n dra-tutorial
   ```

   The output is similar to this:

   ```
   NAME       STATE                AGE
   some-gpu   allocated,reserved   34s
   ```

   In this output, the `STATE` column shows that the ResourceClaim is allocated and reserved.

3. Check the details of the `some-gpu` ResourceClaim. The `status` stanza of the ResourceClaim has information about the allocated device and the Pod it has been reserved for:

   ```
   kubectl get resourceclaim some-gpu -n dra-tutorial -o yaml
   ```

   The output is similar to this:

   ```
   apiVersion: resource.k8s.io/v1
   kind: ResourceClaim  metadata:       creationTimestamp: "2025-08-20T18:17:31Z"       finalizers:       - resource.ku
   ```

4. To check how the driver handled device allocation, get the logs for the driver DaemonSet Pods:

   ```
   kubectl logs -l app.kubernetes.io/name=dra-example-driver -n dra-tutorial
   ```

   The output is similar to this:

   ```
   I0820 18:17:44.131324       1 driver.go:106] PrepareResourceClaims is called: number of claims: 1
   I0820 18:17:44.135056       1 driver.go:133] Returning newly prepared devices for claim 'd3e48dbf-40da-47c3-a7b9-f7d54d1051c3
   ```

You have now successfully deployed a Pod that claims devices using DRA, verified that the Pod was scheduled to an appropriate node, and saw that the associated DRA APIs kinds were updated with the allocation status.

Delete a Pod that has a claim

When a Pod with a claim is deleted, the DRA driver deallocates the resource so it can be available for future scheduling. To validate this behavior, delete the Pod that you created in the previous steps and watch the corresponding changes to the ResourceClaim and driver.

1. Delete the `pod0` Pod:

   ```
   kubectl delete pod pod0 -n dra-tutorial
   ```

   The output is similar to this:

   ```
   pod "pod0" deleted
   ```

Observe the DRA state

When the Pod is deleted, the driver deallocates the device from the ResourceClaim and updates the ResourceClaim resource in the Kubernetes API. The ResourceClaim has a `pending` state until it's referenced in a new Pod.

1. Check the state of the `some-gpu` ResourceClaim:

   ```
   kubectl get resourceclaims -n dra-tutorial
   ```

   The output is similar to this:

   ```
   NAME       STATE     AGE
   some-gpu   pending   76s
   ```

2. Verify that the driver has processed unpreparing the device for this claim by checking the driver logs:

   ```
   kubectl logs -l app.kubernetes.io/name=dra-example-driver -n dra-tutorial
   ```

   The output is similar to this:

   ```
   I0820 18:22:15.629376       1 driver.go:138] UnprepareResourceClaims is called: number of claims: 1
   ```

You have now deleted a Pod that had a claim, and observed that the driver took action to unprepare the underlying hardware resource and update the DRA APIs to reflect that the resource is available again for future scheduling.

Cleaning up

To clean up the resources that you created in this tutorial, follow these steps:

```
kubectl delete namespace dra-tutorial
kubectl delete deviceclass gpu.example.com
kubectl delete clusterrole dra-example-driver-role
kubectl delete clusterrolebinding dra-example-driver-role-binding
kubectl delete priorityclass dra-driver-high-priority
```

What's next

- Learn more about DRA
- Allocate Devices to Workloads with DRA

Configuration

Updating Configuration via a ConfigMap

Configuring Redis using a ConfigMap

Adopting Sidecar Containers

Cluster Management

Running Kubelet in Standalone Mode

Configuring swap memory on Kubernetes nodes

Install Drivers and Allocate Devices with DRA

Namespaces Walkthrough