Configure a Pod to Use a Volume for Storage

This page shows how to configure a Pod to use a Volume for storage.

A Container's file system lives only as long as the Container does. So when a Container terminates and restarts, filesystem changes are lost. For more consistent storage that is independent of the Container, you can use a Volume. This is especially important for stateful applications, such as key-value stores (such as Redis) and databases.

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`.

Configure a volume for a Pod

In this exercise, you create a Pod that runs one Container. This Pod has a Volume of type emptyDir that lasts for the life of the Pod, even if the Container terminates and restarts. Here is the configuration file for the Pod:

`pods/storage/redis.yaml` Copy pods/storage/redis.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: redisspec:  containers:  - name: redis    image: redis    volumeMounts:    - name: redis-storage      mo
```

1. Create the Pod:

   ```
   kubectl apply -f https://k8s.io/examples/pods/storage/redis.yaml
   ```

2. Verify that the Pod's Container is running, and then watch for changes to the Pod:

   ```
   kubectl get pod redis --watch
   ```

   The output looks like this:

   ```
   NAME      READY     STATUS    RESTARTS   AGE
   redis     1/1       Running   0          13s
   ```

3. In another terminal, get a shell to the running Container:

   ```
   kubectl exec -it redis -- /bin/bash
   ```

4. In your shell, go to `/data/redis`, and then create a file:

   ```
   root@redis:/data# cd /data/redis/
   root@redis:/data/redis# echo Hello > test-file
   ```

5. In your shell, list the running processes:

   ```
   root@redis:/data/redis# apt-get update
   root@redis:/data/redis# apt-get install procps
   root@redis:/data/redis# ps aux
   ```

   The output is similar to this:

   ```
   USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
   redis        1  0.1  0.1  33308  3828 ?        Ssl  00:46   0:00 redis-server *:6379
   root        12  0.0  0.0  20228  3020 ?        Ss   00:47   0:00 /bin/bash
   root        15  0.0  0.0  17500  2072 ?        R+   00:48   0:00 ps aux
   ```

6. In your shell, kill the Redis process:

   ```
   root@redis:/data/redis# kill <pid>
   ```

   where `<pid>` is the Redis process ID (PID).

7. In your original terminal, watch for changes to the Redis Pod. Eventually, you will see something like this:

   ```
   NAME      READY     STATUS     RESTARTS   AGE
   redis     1/1       Running    0           13s
   redis     0/1       Completed  0          6m
   redis     1/1       Running    1          6m
   ```

At this point, the Container has terminated and restarted. This is because the Redis Pod has a restartPolicy of `Always`.

1. Get a shell into the restarted Container:

```
kubectl exec -it redis -- /bin/bash
```

2. In your shell, go to `/data/redis`, and verify that `test-file` is still there.

```
root@redis:/data/redis# cd /data/redis/
root@redis:/data/redis# ls
test-file
```

3. Delete the Pod that you created for this exercise:

```
kubectl delete pod redis
```

What's next

- See Volume.

- See Pod.

- In addition to the local disk storage provided by `emptyDir`, Kubernetes supports many different network-attached storage solutions, including PD on GCE and EBS on EC2, which are preferred for critical data and will handle details such as mounting and unmounting the devices on the nodes. See Volumes for more details.

Assign Pods to Nodes using Node Affinity

This page shows how to assign a Kubernetes Pod to a particular node using Node Affinity in a Kubernetes 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

Your Kubernetes server must be at or later than version v1.10.

To check the version, enter `kubectl version`.

Add a label to a node

1. List the nodes in your cluster, along with their labels:

```
kubectl get nodes --show-labels
```

The output is similar to this:

```
NAME      STATUS    ROLES    AGE     VERSION        LABELS
worker0   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker0
worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker1
worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker2
```

2. Choose one of your nodes, and add a label to it:

```
kubectl label nodes <your-node-name> disktype=ssd
```

where `<your-node-name>` is the name of your chosen node.

3. Verify that your chosen node has a `disktype=ssd` label:

```
kubectl get nodes --show-labels
```

The output is similar to this:

```
NAME      STATUS    ROLES    AGE     VERSION        LABELS
worker0   Ready     <none>   1d      v1.13.0        ...,disktype=ssd,kubernetes.io/hostname=worker0
worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker1
worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker2
```

In the preceding output, you can see that the `worker0` node has a `disktype=ssd` label.

Schedule a Pod using required node affinity

This manifest describes a Pod that has a `requiredDuringSchedulingIgnoredDuringExecution` node affinity,`disktype: ssd`. This means that the pod will get scheduled only on a node that has a `disktype=ssd` label.

`pods/pod-nginx-required-affinity.yaml` Copy pods/pod-nginx-required-affinity.yaml to clipboard

```
apiVersion: v1
```

```
kind: Podmetadata:  name: nginxspec:  affinity:    nodeAffinity:      requiredDuringSchedulingIgnoredDuringExecution:        nodeS
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx-required-affinity.yaml
```

2. Verify that the pod is running on your chosen node:

```
kubectl get pods --output=wide
```

The output is similar to this:

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

Schedule a Pod using preferred node affinity

This manifest describes a Pod that has a `preferredDuringSchedulingIgnoredDuringExecution` node affinity,`disktype: ssd`. This means that the pod will prefer a node that has a `disktype=ssd` label.

pods/pod-nginx-preferred-affinity.yaml Copy pods/pod-nginx-preferred-affinity.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginxspec:  affinity:    nodeAffinity:      preferredDuringSchedulingIgnoredDuringExecution:      - weig
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx-preferred-affinity.yaml
```

2. Verify that the pod is running on your chosen node:

```
kubectl get pods --output=wide
```

The output is similar to this:

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

What's next

Learn more about Node Affinity.

Configure Service Accounts for Pods

Kubernetes offers two distinct ways for clients that run within your cluster, or that otherwise have a relationship to your cluster's control plane to authenticate to the API server.

A *service account* provides an identity for processes that run in a Pod, and maps to a ServiceAccount object. When you authenticate to the API server, you identify yourself as a particular *user*. Kubernetes recognises the concept of a user, however, Kubernetes itself does **not** have a User API.

This task guide is about ServiceAccounts, which do exist in the Kubernetes API. The guide shows you some ways to configure ServiceAccounts for Pods.

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

Use the default service account to access the API server

When Pods contact the API server, Pods authenticate as a particular ServiceAccount (for example, `default`). There is always at least one ServiceAccount in each namespace.

Every Kubernetes namespace contains at least one ServiceAccount: the default ServiceAccount for that namespace, named `default`. If you do not specify a ServiceAccount when you create a Pod, Kubernetes automatically assigns the ServiceAccount named `default` in that namespace.

You can fetch the details for a Pod you have created. For example:

```
kubectl get pods/<podname> -o yaml
```

In the output, you see a field `spec.serviceAccountName`. Kubernetes automatically sets that value if you don't specify it when you create a Pod.

An application running inside a Pod can access the Kubernetes API using automatically mounted service account credentials. See accessing the Cluster to learn more.

When a Pod authenticates as a ServiceAccount, its level of access depends on the authorization plugin and policy in use.

The API credentials are automatically revoked when the Pod is deleted, even if finalizers are in place. In particular, the API credentials are revoked 60 seconds beyond the `.metadata.deletionTimestamp` set on the Pod (the deletion timestamp is typically the time that the **delete** request was accepted plus the Pod's termination grace period).

Opt out of API credential automounting

If you don't want the kubelet to automatically mount a ServiceAccount's API credentials, you can opt out of the default behavior. You can opt out of automounting API credentials on `/var/run/secrets/kubernetes.io/serviceaccount/token` for a service account by setting `automountServiceAccountToken: false` on the ServiceAccount:

For example:

```
apiVersion: v1
kind: ServiceAccountmetadata:  name: build-robotautomountServiceAccountToken: false...
```

You can also opt out of automounting API credentials for a particular Pod:

```
apiVersion: v1
kind: Podmetadata:  name: my-podspec:  serviceAccountName: build-robot  automountServiceAccountToken: false  ...
```

If both the ServiceAccount and the Pod's `.spec` specify a value for `automountServiceAccountToken`, the Pod spec takes precedence.

Use more than one ServiceAccount

Every namespace has at least one ServiceAccount: the default ServiceAccount resource, called `default`. You can list all ServiceAccount resources in your current namespace with:

```
kubectl get serviceaccounts
```

The output is similar to this:

```
NAME      SECRETS    AGE
default   1          1d
```

You can create additional ServiceAccount objects like this:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
```

The name of a ServiceAccount object must be a valid DNS subdomain name.

If you get a complete dump of the service account object, like this:

```
kubectl get serviceaccounts/build-robot -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ServiceAccountmetadata:  creationTimestamp: 2019-06-16T00:12:34Z  name: build-robot  namespace: default  resourceVersion: "2
```

You can use authorization plugins to set permissions on service accounts.

To use a non-default service account, set the `spec.serviceAccountName` field of a Pod to the name of the ServiceAccount you wish to use.

You can only set the `serviceAccountName` field when creating a Pod, or in a template for a new Pod. You cannot update the `.spec.serviceAccountName` field of a Pod that already exists.

Note:

The `.spec.serviceAccount` field is a deprecated alias for `.spec.serviceAccountName`. If you want to remove the fields from a workload resource, set both fields to empty explicitly on the pod template.

Cleanup

If you tried creating `build-robot` ServiceAccount from the example above, you can clean it up by running:

```
kubectl delete serviceaccount/build-robot
```

Manually create an API token for a ServiceAccount

Suppose you have an existing service account named "build-robot" as mentioned earlier.

You can get a time-limited API token for that ServiceAccount using `kubectl`:

```
kubectl create token build-robot
```

The output from that command is a token that you can use to authenticate as that ServiceAccount. You can request a specific token duration using the `--duration` command line argument to `kubectl create token` (the actual duration of the issued token might be shorter, or could even be longer).

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

Using `kubectl` v1.31 or later, it is possible to create a service account token that is directly bound to a Node:

```
kubectl create token build-robot --bound-object-kind Node --bound-object-name node-001 --bound-object-uid 123...456
```

The token will be valid until it expires or either the associated Node or service account are deleted.

Note:

Versions of Kubernetes before v1.22 automatically created long term credentials for accessing the Kubernetes API. This older mechanism was based on creating token Secrets that could then be mounted into running Pods. In more recent versions, including Kubernetes v1.34, API credentials are obtained directly by using the TokenRequest API, and are mounted into Pods using a projected volume. The tokens obtained using this method have bounded lifetimes, and are automatically invalidated when the Pod they are mounted into is deleted.

You can still manually create a service account token Secret; for example, if you need a token that never expires. However, using the TokenRequest subresource to obtain a token to access the API is recommended instead.

Manually create a long-lived API token for a ServiceAccount

If you want to obtain an API token for a ServiceAccount, you create a new Secret with a special annotation, `kubernetes.io/service-account.name`.

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
```

If you view the Secret using:

```
kubectl get secret/build-robot-secret -o yaml
```

you can see that the Secret now contains an API token for the "build-robot" ServiceAccount.

Because of the annotation you set, the control plane automatically generates a token for that ServiceAccounts, and stores them into the associated Secret. The control plane also cleans up tokens for deleted ServiceAccounts.

```
kubectl describe secrets/build-robot-secret
```

The output is similar to this:

```
Name:           build-robot-secret
Namespace:      default
Labels:         <none>
Annotations:    kubernetes.io/service-account.name: build-robot
                kubernetes.io/service-account.uid: da68f9c6-9d26-11e7-b84e-002dc52800da

Type:   kubernetes.io/service-account-token

Data
====
ca.crt:         1338 bytes
namespace:      7 bytes
token:          ...
```

Note:

The content of `token` is omitted here.

Take care not to display the contents of a `kubernetes.io/service-account-token` Secret somewhere that your terminal / computer screen could be seen by an onlooker.

When you delete a ServiceAccount that has an associated Secret, the Kubernetes control plane automatically cleans up the long-lived token from that Secret.

Note:

If you view the ServiceAccount using:

```
kubectl get serviceaccount build-robot -o yaml
```

You can't see the `build-robot-secret` Secret in the ServiceAccount API objects `.secrets` field because that field is only populated with auto-generated Secrets.

Add ImagePullSecrets to a service account

First, create an imagePullSecret. Next, verify it has been created. For example:

- Create an imagePullSecret, as described in Specifying ImagePullSecrets on a Pod.

  ```
  kubectl create secret docker-registry myregistrykey --docker-server=<registry name> \
          --docker-username=DUMMY_USERNAME --docker-password=DUMMY_DOCKER_PASSWORD \        --docker-email=DUMMY_DOCKER_EMAIL
  ```

- Verify it has been created.

  ```
  kubectl get secrets myregistrykey
  ```

  The output is similar to this:

  ```
  NAME             TYPE                              DATA    AGE
  myregistrykey    kubernetes.io/.dockerconfigjson   1       1d
  ```

Add image pull secret to service account

Next, modify the default service account for the namespace to use this Secret as an imagePullSecret.

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
```

You can achieve the same outcome by editing the object manually:

```
kubectl edit serviceaccount/default
```

The output of the `sa.yaml` file is similar to this:

Your selected text editor will open with a configuration looking something like this:

```
apiVersion: v1
kind: ServiceAccountmetadata:  creationTimestamp: 2021-07-07T22:02:39Z  name: default  namespace: default  resourceVersion: "24302
```

Using your editor, delete the line with key `resourceVersion`, add lines for `imagePullSecrets:` and save it. Leave the `uid` value set the same as you found it.

After you made those changes, the edited ServiceAccount looks something like this:

```
apiVersion: v1
kind: ServiceAccountmetadata:  creationTimestamp: 2021-07-07T22:02:39Z  name: default  namespace: default  uid: 052fb0f4-3d50-11e5
```

Verify that imagePullSecrets are set for new Pods

Now, when a new Pod is created in the current namespace and using the default ServiceAccount, the new Pod has its `spec.imagePullSecrets` field set automatically:

```
kubectl run nginx --image=<registry name>/nginx --restart=Never
kubectl get pod nginx -o=jsonpath='{.spec.imagePullSecrets[0].name}{"\n"}'
```

The output is:

```
myregistrykey
```

ServiceAccount token volume projection

FEATURE STATE: `Kubernetes v1.20 [stable]`

Note:

To enable and use token request projection, you must specify each of the following command line arguments to `kube-apiserver`:

`--service-account-issuer`
: defines the Identifier of the service account token issuer. You can specify the `--service-account-issuer` argument multiple times, this can be useful to enable a non-disruptive change of the issuer. When this flag is specified multiple times, the first is used to generate tokens and all are used to determine which issuers are accepted. You must be running Kubernetes v1.22 or later to be able to specify `--service-account-issuer` multiple times.

`--service-account-key-file`
: specifies the path to a file containing PEM-encoded X.509 private or public keys (RSA or ECDSA), used to verify ServiceAccount tokens. The specified file can contain multiple keys, and the flag can be specified multiple times with different files. If specified multiple times, tokens signed by any of the specified keys are considered valid by the Kubernetes API server.

`--service-account-signing-key-file`
: specifies the path to a file that contains the current private key of the service account token issuer. The issuer signs issued ID tokens with this private key.

`--api-audiences` (can be omitted)
: defines audiences for ServiceAccount tokens. The service account token authenticator validates that tokens used against the API are bound to at least one of these audiences. If `api-audiences` is specified multiple times, tokens for any of the specified audiences are considered valid by the Kubernetes API server. If you specify the `--service-account-issuer` command line argument but you don't set `--api-audiences`, the control plane defaults to a single element audience list that contains only the issuer URL.

The kubelet can also project a ServiceAccount token into a Pod. You can specify desired properties of the token, such as the audience and the validity duration. These properties are *not* configurable on the default ServiceAccount token. The token will also become invalid against the API when either the Pod or the ServiceAccount is deleted.

You can configure this behavior for the `spec` of a Pod using a projected volume type called `ServiceAccountToken`.

The token from this projected volume is a JSON Web Token (JWT). The JSON payload of this token follows a well defined schema - an example payload for a pod bound token:

```
{
  "aud": [  # matches the requested audiences, or the API server's default audiences when none are explicitly requested
    "https://kubernetes.default.svc"
```

```
  ],
  "exp": 1731613413,
  "iat": 1700077413,
  "iss": "https://kubernetes.default.svc",  # matches the first value passed to the --service-account-issuer flag
  "jti": "ea28ed49-2e11-4280-9ec5-bc3d1d84661a",
  "kubernetes.io": {
    "namespace": "kube-system",
    "node": {
      "name": "127.0.0.1",
      "uid": "58456cb0-dd00-45ed-b797-5578fdceaced"
    },
    "pod": {
      "name": "coredns-69cbfb9798-jv9gn",
      "uid": "778a530c-b3f4-47c0-9cd5-ab018fb64f33"
    },
    "serviceaccount": {
      "name": "coredns",
      "uid": "a087d5a0-e1dd-43ec-93ac-f13d89cd13af"
    },
    "warnafter": 1700081020
  },
  "nbf": 1700077413,
  "sub": "system:serviceaccount:kube-system:coredns"
}
```

Launch a Pod using service account token projection

To provide a Pod with a token with an audience of `vault` and a validity duration of two hours, you could define a Pod manifest that is similar to:

pods/pod-projected-svc-token.yaml Copy pods/pod-projected-svc-token.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginxspec:  containers:  - image: nginx    name: nginx    volumeMounts:    - mountPath: /var/run/secrets
```

Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/pod-projected-svc-token.yaml
```

The kubelet will: request and store the token on behalf of the Pod; make the token available to the Pod at a configurable file path; and refresh the token as it approaches expiration. The kubelet proactively requests rotation for the token if it is older than 80% of its total time-to-live (TTL), or if the token is older than 24 hours.

The application is responsible for reloading the token when it rotates. It's often good enough for the application to load the token on a schedule (for example: once every 5 minutes), without tracking the actual expiry time.

Service account issuer discovery

FEATURE STATE: `Kubernetes v1.21 [stable]`

If you have enabled token projection for ServiceAccounts in your cluster, then you can also make use of the discovery feature. Kubernetes provides a way for clients to federate as an *identity provider*, so that one or more external systems can act as a *relying party*.

Note:

The issuer URL must comply with the OIDC Discovery Spec. In practice, this means it must use the `https` scheme, and should serve an OpenID provider configuration at `{service-account-issuer}/.well-known/openid-configuration`.

If the URL does not comply, ServiceAccount issuer discovery endpoints are not registered or accessible.

When enabled, the Kubernetes API server publishes an OpenID Provider Configuration document via HTTP. The configuration document is published at `/.well-known/openid-configuration`. The OpenID Provider Configuration is sometimes referred to as the *discovery document*. The Kubernetes API server publishes the related JSON Web Key Set (JWKS), also via HTTP, at `/openid/v1/jwks`.

Note:

The responses served at `/.well-known/openid-configuration` and `/openid/v1/jwks` are designed to be OIDC compatible, but not strictly OIDC compliant. Those documents contain only the parameters necessary to perform validation of Kubernetes service account tokens.

Clusters that use RBAC include a default ClusterRole called `system:service-account-issuer-discovery`. A default ClusterRoleBinding assigns this role to the `system:serviceaccounts` group, which all ServiceAccounts implicitly belong to. This allows pods running on the cluster to access the service account discovery document via their mounted service account token. Administrators may, additionally, choose to bind the role to `system:authenticated` or `system:unauthenticated` depending on their security requirements and which external systems they intend to federate with.

The JWKS response contains public keys that a relying party can use to validate the Kubernetes service account tokens. Relying parties first query for the OpenID Provider Configuration, and use the `jwks_uri` field in the response to find the JWKS.

In many cases, Kubernetes API servers are not available on the public internet, but public endpoints that serve cached responses from the API server can be made available by users or by service providers. In these cases, it is possible to override the `jwks_uri` in the OpenID Provider Configuration so that it points to the public endpoint, rather than the API server's address, by passing the `--service-account-jwks-uri` flag to the API server. Like the issuer URL, the JWKS URI is required to use the `https` scheme.

What's next

See also:

- Read the Cluster Admin Guide to Service Accounts
- Read about Authorization in Kubernetes
- Read about Secrets
  - or learn to distribute credentials securely using Secrets
  - but also bear in mind that using Secrets for authenticating as a ServiceAccount is deprecated. The recommended alternative is ServiceAccount token volume projection.
- Read about projected volumes.
- For background on OIDC discovery, read the ServiceAccount signing key retrieval Kubernetes Enhancement Proposal
- Read the OIDC Discovery Spec

Configure Pod Initialization

This page shows how to use an Init Container to initialize a Pod before an application Container runs.

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`.

Create a Pod that has an Init Container

In this exercise you create a Pod that has one application Container and one Init Container. The init container runs to completion before the application container starts.

Here is the configuration file for the Pod:

`pods/init-containers.yaml` Copy pods/init-containers.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: init-demospec:  containers:  - name: nginx    image: nginx    ports:    - containerPort: 80    volumeMou
```

In the configuration file, you can see that the Pod has a Volume that the init container and the application container share.

The init container mounts the shared Volume at `/work-dir`, and the application container mounts the shared Volume at `/usr/share/nginx/html`. The init container runs the following command and then terminates:

```
wget -O /work-dir/index.html http://info.cern.ch
```

Notice that the init container writes the `index.html` file in the root directory of the nginx server.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/init-containers.yaml
```

Verify that the nginx container is running:

```
kubectl get pod init-demo
```

The output shows that the nginx container is running:

```
NAME        READY     STATUS    RESTARTS   AGE
init-demo   1/1       Running   0          1m
```

Get a shell into the nginx container running in the init-demo Pod:

```
kubectl exec -it init-demo -- /bin/bash
```

In your shell, send a GET request to the nginx server:

```
root@nginx:~# apt-get update
root@nginx:~# apt-get install curl
root@nginx:~# curl localhost
```

The output shows that nginx is serving the web page that was written by the init container:

```
<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>

<h1>http://info.cern.ch - home of the first website</h1>
  ...
  <li><a href="http://info.cern.ch/hypertext/WWW/TheProject.html">Browse the first website</a></li>
  ...
```

What's next

- Learn more about communicating between Containers running in the same Pod.
- Learn more about Init Containers.
- Learn more about Volumes.
- Learn more about Debugging Init Containers

Create a Windows HostProcess Pod

FEATURE STATE: `Kubernetes v1.26 [stable]`

Windows HostProcess containers enable you to run containerized workloads on a Windows host. These containers operate as normal processes but have access to the host network namespace, storage, and devices when given the appropriate user privileges. HostProcess containers can be used to deploy network plugins, storage configurations, device plugins, kube-proxy, and other components to Windows nodes without the need for dedicated proxies or the direct installation of host services.

Administrative tasks such as installation of security patches, event log collection, and more can be performed without requiring cluster operators to log onto each Windows node. HostProcess containers can run as any user that is available on the host or is in the domain of the host machine, allowing administrators to restrict resource access through user permissions. While neither filesystem or process isolation are supported, a new volume is created on the host upon starting the container to give it a clean and consolidated workspace. HostProcess containers can also be built on top of existing Windows base images and do not inherit the same compatibility requirements as Windows server containers, meaning that the version of the base images does not need to match that of the host. It is, however, recommended that you use the same base image version as your Windows Server container workloads to ensure you do not have any unused images taking up space on the node. HostProcess containers also support volume mounts within the container volume.

When should I use a Windows HostProcess container?

- When you need to perform tasks which require the networking namespace of the host. HostProcess containers have access to the host's network interfaces and IP addresses.
- You need access to resources on the host such as the filesystem, event logs, etc.
- Installation of specific device drivers or Windows services.
- Consolidation of administrative tasks and security policies. This reduces the degree of privileges needed by Windows nodes.

Before you begin

This task guide is specific to Kubernetes v1.34. If you are not running Kubernetes v1.34, check the documentation for that version of Kubernetes.

In Kubernetes 1.34, the HostProcess container feature is enabled by default. The kubelet will communicate with containerd directly by passing the hostprocess flag via CRI. You can use the latest version of containerd (v1.6+) to run HostProcess containers. How to install containerd.

Limitations

These limitations are relevant for Kubernetes v1.34:

- HostProcess containers require containerd 1.6 or higher container runtime and containerd 1.7 is recommended.
- HostProcess pods can only contain HostProcess containers. This is a current limitation of the Windows OS; non-privileged Windows containers cannot share a vNIC with the host IP namespace.
- HostProcess containers run as a process on the host and do not have any degree of isolation other than resource constraints imposed on the HostProcess user account. Neither filesystem or Hyper-V isolation are supported for HostProcess containers.
- Volume mounts are supported and are mounted under the container volume. See Volume Mounts
- A limited set of host user accounts are available for HostProcess containers by default. See Choosing a User Account.
- Resource limits (disk, memory, cpu count) are supported in the same fashion as processes on the host.
- Both Named pipe mounts and Unix domain sockets are **not** supported and should instead be accessed via their path on the host (e.g. \\.\pipe*)

HostProcess Pod configuration requirements

Enabling a Windows HostProcess pod requires setting the right configurations in the pod security configuration. Of the policies defined in the Pod Security Standards HostProcess pods are disallowed by the baseline and restricted policies. It is therefore recommended that HostProcess pods run in alignment with the privileged profile.

When running under the privileged policy, here are the configurations which need to be set to enable the creation of a HostProcess pod:

Control	Policy
`securityContext.windowsOptions.hostProcess`	Windows pods offer the ability to run HostProcess containers which enables privileged access to the Windows node. **Allowed Values** - `true`
`hostNetwork`	Pods container HostProcess containers must use the host's network namespace. **Allowed Values** - `true`
`securityContext.windowsOptions.runAsUserName`	Specification of which user the HostProcess container should run as is required for the pod spec.

Control	Policy
	Allowed Values • `NT AUTHORITY\SYSTEM` • `NT AUTHORITY\Local service` • `NT AUTHORITY\NetworkService` • Local usergroup names (see below)
`runAsNonRoot`	Because HostProcess containers have privileged access to the host, the `runAsNonRoot` field cannot be set to true. **Allowed Values** • Undefined/Nil • `false`

Example manifest (excerpt)

```
spec:
  securityContext:
    windowsOptions:
      hostProcess: true
      runAsUserName: "NT AUTHORITY\\Local service"
  hostNetwork: true
  containers:
  - name: test
    image: image1:latest
    command:
      - ping
      - -t
      - 127.0.0.1
  nodeSelector:
    "kubernetes.io/os": windows
```

Volume mounts

HostProcess containers support the ability to mount volumes within the container volume space. Volume mount behavior differs depending on the version of containerd runtime used by on the node.

Containerd v1.6

Applications running inside the container can access volume mounts directly via relative or absolute paths. An environment variable `$CONTAINER_SANDBOX_MOUNT_POINT` is set upon container creation and provides the absolute host path to the container volume. Relative paths are based upon the `.spec.containers.volumeMounts.mountPath` configuration.

To access service account tokens (for example) the following path structures are supported within the container:

- `.\var\run\secrets\kubernetes.io\serviceaccount\`
- `$CONTAINER_SANDBOX_MOUNT_POINT\var\run\secrets\kubernetes.io\serviceaccount\`

Containerd v1.7 (and greater)

Applications running inside the container can access volume mounts directly via the volumeMount's specified `mountPath` (just like Linux and non-HostProcess Windows containers).

For backwards compatibility volumes can also be accessed via using the same relative paths configured by containerd v1.6.

As an example, to access service account tokens within the container you would use one of the following paths:

- `c:\var\run\secrets\kubernetes.io\serviceaccount`
- `/var/run/secrets/kubernetes.io/serviceaccount/`
- `$CONTAINER_SANDBOX_MOUNT_POINT\var\run\secrets\kubernetes.io\serviceaccount\`

Resource limits

Resource limits (disk, memory, cpu count) are applied to the job and are job wide. For example, with a limit of 10MB set, the memory allocated for any HostProcess job object will be capped at 10MB. This is the same behavior as other Windows container types. These limits would be specified the same way they are currently for whatever orchestrator or runtime is being used. The only difference is in the disk resource usage calculation used for resource tracking due to the difference in how HostProcess containers are bootstrapped.

Choosing a user account

System accounts

By default, HostProcess containers support the ability to run as one of three supported Windows service accounts:

- **LocalSystem**
- **LocalService**
- **NetworkService**

You should select an appropriate Windows service account for each HostProcess container, aiming to limit the degree of privileges so as to avoid accidental (or even malicious) damage to the host. The LocalSystem service account has the highest level of privilege of the three and should be used only if absolutely necessary. Where possible, use the LocalService service account as it is the least privileged of the three options.

Local accounts

If configured, HostProcess containers can also run as local user accounts which allows for node operators to give fine-grained access to workloads.

To run HostProcess containers as a local user; A local usergroup must first be created on the node and the name of that local usergroup must be specified in the `runAsUserName` field in the deployment. Prior to initializing the HostProcess container, a new **ephemeral** local user account to be created and joined to the specified usergroup, from which the container is run. This provides a number a benefits including eliminating the need to manage passwords for local user accounts. An initial HostProcess container running as a service account can be used to prepare the user groups for later HostProcess containers.

Note:

Running HostProcess containers as local user accounts requires containerd v1.7+

Example:

1. Create a local user group on the node (this can be done in another HostProcess container).

   ```
   net localgroup hpc-localgroup /add
   ```

2. Grant access to desired resources on the node to the local usergroup. This can be done with tools like icacls.

3. Set `runAsUserName` to the name of the local usergroup for the pod or individual containers.

   ```
   securityContext:
     windowsOptions:
       hostProcess: true
       runAsUserName: hpc-localgroup
   ```

4. Schedule the pod!

Base Image for HostProcess Containers

HostProcess containers can be built from any of the existing Windows Container base images.

Additionally a new base mage has been created just for HostProcess containers! For more information please check out the windows-host-process-containers-base-image github project.

Troubleshooting HostProcess containers

- HostProcess containers fail to start with `failed to create user process token: failed to logon user: Access is denied.: unknown`

  Ensure containerd is running as `LocalSystem` or `LocalService` service accounts. User accounts (even Administrator accounts) do not have permissions to create logon tokens for any of the supported user accounts.

Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller

This page describes the process of migrating from PodSecurityPolicies to the built-in PodSecurity admission controller. This can be done effectively using a combination of dry-run and `audit` and `warn` modes, although this becomes harder if mutating PSPs are used.

Before you begin

Your Kubernetes server must be at or later than version v1.22.

To check the version, enter `kubectl version`.

If you are currently running a version of Kubernetes other than 1.34, you may want to switch to viewing this page in the documentation for the version of Kubernetes that you are actually running.

This page assumes you are already familiar with the basic Pod Security Admission concepts.

Overall approach

There are multiple strategies you can take for migrating from PodSecurityPolicy to Pod Security Admission. The following steps are one possible migration path, with a goal of minimizing both the risks of a production outage and of a security gap.

0. Decide whether Pod Security Admission is the right fit for your use case.
1. Review namespace permissions
2. Simplify & standardize PodSecurityPolicies
3. Update namespaces
   1. Identify an appropriate Pod Security level
   2. Verify the Pod Security level

    3. Enforce the Pod Security level
    4. Bypass PodSecurityPolicy
4. Review namespace creation processes
5. Disable PodSecurityPolicy

0. Decide whether Pod Security Admission is right for you

Pod Security Admission was designed to meet the most common security needs out of the box, and to provide a standard set of security levels across clusters. However, it is less flexible than PodSecurityPolicy. Notably, the following features are supported by PodSecurityPolicy but not Pod Security Admission:

- **Setting default security constraints** - Pod Security Admission is a non-mutating admission controller, meaning it won't modify pods before validating them. If you were relying on this aspect of PSP, you will need to either modify your workloads to meet the Pod Security constraints, or use a Mutating Admission Webhook to make those changes. See Simplify & Standardize PodSecurityPolicies below for more detail.
- **Fine-grained control over policy definition** - Pod Security Admission only supports 3 standard levels. If you require more control over specific constraints, then you will need to use a Validating Admission Webhook to enforce those policies.
- **Sub-namespace policy granularity** - PodSecurityPolicy lets you bind different policies to different Service Accounts or users, even within a single namespace. This approach has many pitfalls and is not recommended, but if you require this feature anyway you will need to use a 3rd party webhook instead. The exception to this is if you only need to completely exempt specific users or RuntimeClasses, in which case Pod Security Admission does expose some static configuration for exemptions.

Even if Pod Security Admission does not meet all of your needs it was designed to be *complementary* to other policy enforcement mechanisms, and can provide a useful fallback running alongside other admission webhooks.

1. Review namespace permissions

Pod Security Admission is controlled by labels on namespaces. This means that anyone who can update (or patch or create) a namespace can also modify the Pod Security level for that namespace, which could be used to bypass a more restrictive policy. Before proceeding, ensure that only trusted, privileged users have these namespace permissions. It is not recommended to grant these powerful permissions to users that shouldn't have elevated permissions, but if you must you will need to use an admission webhook to place additional restrictions on setting Pod Security labels on Namespace objects.

2. Simplify & standardize PodSecurityPolicies

In this section, you will reduce mutating PodSecurityPolicies and remove options that are outside the scope of the Pod Security Standards. You should make the changes recommended here to an offline copy of the original PodSecurityPolicy being modified. The cloned PSP should have a different name that is alphabetically before the original (for example, prepend a `0` to it). Do not create the new policies in Kubernetes yet - that will be covered in the Rollout the updated policies section below.

2.a. Eliminate purely mutating fields

If a PodSecurityPolicy is mutating pods, then you could end up with pods that don't meet the Pod Security level requirements when you finally turn PodSecurityPolicy off. In order to avoid this, you should eliminate all PSP mutation prior to switching over. Unfortunately PSP does not cleanly separate mutating & validating fields, so this is not a straightforward migration.

You can start by eliminating the fields that are purely mutating, and don't have any bearing on the validating policy. These fields (also listed in the Mapping PodSecurityPolicies to Pod Security Standards reference) are:

- `.spec.defaultAllowPrivilegeEscalation`
- `.spec.runtimeClass.defaultRuntimeClassName`
- `.metadata.annotations['seccomp.security.alpha.kubernetes.io/defaultProfileName']`
- `.metadata.annotations['apparmor.security.beta.kubernetes.io/defaultProfileName']`
- `.spec.defaultAddCapabilities` - Although technically a mutating & validating field, these should be merged into `.spec.allowedCapabilities` which performs the same validation without mutation.

Caution:

Removing these could result in workloads missing required configuration, and cause problems. See Rollout the updated policies below for advice on how to roll these changes out safely.

2.b. Eliminate options not covered by the Pod Security Standards

There are several fields in PodSecurityPolicy that are not covered by the Pod Security Standards. If you must enforce these options, you will need to supplement Pod Security Admission with an admission webhook, which is outside the scope of this guide.

First, you can remove the purely validating fields that the Pod Security Standards do not cover. These fields (also listed in the Mapping PodSecurityPolicies to Pod Security Standards reference with "no opinion") are:

- `.spec.allowedHostPaths`
- `.spec.allowedFlexVolumes`
- `.spec.allowedCSIDrivers`
- `.spec.forbiddenSysctls`
- `.spec.runtimeClass`

You can also remove the following fields, that are related to POSIX / UNIX group controls.

Caution:

If any of these use the `MustRunAs` strategy they may be mutating! Removing these could result in workloads not setting the required groups, and cause problems. See Rollout the updated policies below for advice on how to roll these changes out safely.

- `.spec.runAsGroup`
- `.spec.supplementalGroups`
- `.spec.fsGroup`

The remaining mutating fields are required to properly support the Pod Security Standards, and will need to be handled on a case-by-case basis later:

- `.spec.requiredDropCapabilities` - Required to drop `ALL` for the Restricted profile.
- `.spec.seLinux` - (Only mutating with the `MustRunAs` rule) required to enforce the SELinux requirements of the Baseline & Restricted profiles.
- `.spec.runAsUser` - (Non-mutating with the `RunAsAny` rule) required to enforce `RunAsNonRoot` for the Restricted profile.
- `.spec.allowPrivilegeEscalation` - (Only mutating if set to `false`) required for the Restricted profile.

2.c. Rollout the updated PSPs

Next, you can rollout the updated policies to your cluster. You should proceed with caution, as removing the mutating options may result in workloads missing required configuration.

For each updated PodSecurityPolicy:

1. Identify pods running under the original PSP. This can be done using the `kubernetes.io/psp` annotation. For example, using kubectl:

   ```
   PSP_NAME="original" # Set the name of the PSP you're checking for
   kubectl get pods --all-namespaces -o jsonpath="{range .items[?(@.metadata.annotations.kubernetes\.io\/psp=='$PSP_NAME')]}{.me
   ```

2. Compare these running pods against the original pod spec to determine whether PodSecurityPolicy has modified the pod. For pods created by a workload resource you can compare the pod with the PodTemplate in the controller resource. If any changes are identified, the original Pod or PodTemplate should be updated with the desired configuration. The fields to review are:
   - `.metadata.annotations['container.apparmor.security.beta.kubernetes.io/*']` (replace * with each container name)
   - `.spec.runtimeClassName`
   - `.spec.securityContext.fsGroup`
   - `.spec.securityContext.seccompProfile`
   - `.spec.securityContext.seLinuxOptions`
   - `.spec.securityContext.supplementalGroups`
   - On containers, under `.spec.containers[*]` and `.spec.initContainers[*]`:
     - `.securityContext.allowPrivilegeEscalation`
     - `.securityContext.capabilities.add`
     - `.securityContext.capabilities.drop`
     - `.securityContext.readOnlyRootFilesystem`
     - `.securityContext.runAsGroup`
     - `.securityContext.runAsNonRoot`
     - `.securityContext.runAsUser`
     - `.securityContext.seccompProfile`
     - `.securityContext.seLinuxOptions`
3. Create the new PodSecurityPolicies. If any Roles or ClusterRoles are granting `use` on all PSPs this could cause the new PSPs to be used instead of their mutating counter-parts.
4. Update your authorization to grant access to the new PSPs. In RBAC this means updating any Roles or ClusterRoles that grant the `use` permission on the original PSP to also grant it to the updated PSP.
5. Verify: after some soak time, rerun the command from step 1 to see if any pods are still using the original PSPs. Note that pods need to be recreated after the new policies have been rolled out before they can be fully verified.
6. (optional) Once you have verified that the original PSPs are no longer in use, you can delete them.

3. Update Namespaces

The following steps will need to be performed on every namespace in the cluster. Commands referenced in these steps use the `$NAMESPACE` variable to refer to the namespace being updated.

3.a. Identify an appropriate Pod Security level

Start reviewing the Pod Security Standards and familiarizing yourself with the 3 different levels.

There are several ways to choose a Pod Security level for your namespace:

1. **By security requirements for the namespace** - If you are familiar with the expected access level for the namespace, you can choose an appropriate level based on those requirements, similar to how one might approach this on a new cluster.
2. **By existing PodSecurityPolicies** - Using the Mapping PodSecurityPolicies to Pod Security Standards reference you can map each PSP to a Pod Security Standard level. If your PSPs aren't based on the Pod Security Standards, you may need to decide between choosing a level that is at least as permissive as the PSP, and a level that is at least as restrictive. You can see which PSPs are in use for pods in a given namespace with this command:

   ```
   kubectl get pods -n $NAMESPACE -o jsonpath="{.items[*].metadata.annotations.kubernetes\.io\/psp}" | tr " " "\n" | sort -u
   ```

3. **By existing pods** - Using the strategies under Verify the Pod Security level, you can test out both the Baseline and Restricted levels to see whether they are sufficiently permissive for existing workloads, and chose the least-privileged valid level.

Caution:

Options 2 & 3 above are based on *existing* pods, and may miss workloads that aren't currently running, such as CronJobs, scale-to-zero workloads, or other workloads that haven't rolled out.

3.b. Verify the Pod Security level

Once you have selected a Pod Security level for the namespace (or if you're trying several), it's a good idea to test it out first (you can skip this step if using the Privileged level). Pod Security includes several tools to help test and safely roll out profiles.

First, you can dry-run the policy, which will evaluate pods currently running in the namespace against the applied policy, without making the new policy take effect:

```
# $LEVEL is the level to dry-run, either "baseline" or "restricted".
kubectl label --dry-run=server --overwrite ns $NAMESPACE pod-security.kubernetes.io/enforce=$LEVEL
```

This command will return a warning for any *existing* pods that are not valid under the proposed level.

The second option is better for catching workloads that are not currently running: audit mode. When running under audit-mode (as opposed to enforcing), pods that violate the policy level are recorded in the audit logs, which can be reviewed later after some soak time, but are not forbidden. Warning mode works similarly, but returns the warning to the user immediately. You can set the audit level on a namespace with this command:

```
kubectl label --overwrite ns $NAMESPACE pod-security.kubernetes.io/audit=$LEVEL
```

If either of these approaches yield unexpected violations, you will need to either update the violating workloads to meet the policy requirements, or relax the namespace Pod Security level.

3.c. Enforce the Pod Security level

When you are satisfied that the chosen level can safely be enforced on the namespace, you can update the namespace to enforce the desired level:

```
kubectl label --overwrite ns $NAMESPACE pod-security.kubernetes.io/enforce=$LEVEL
```

3.d. Bypass PodSecurityPolicy

Finally, you can effectively bypass PodSecurityPolicy at the namespace level by binding the fully privileged PSP to all service accounts in the namespace.

```
# The following cluster-scoped commands are only needed once.
kubectl apply -f privileged-psp.yaml
kubectl create clusterrole privileged-psp --verb use --resource podsecuritypolicies.policy --resource-name privileged

# Per-namespace disable
kubectl create -n $NAMESPACE rolebinding disable-psp --clusterrole privileged-psp --group system:serviceaccounts:$NAMESPACE
```

Since the privileged PSP is non-mutating, and the PSP admission controller always prefers non-mutating PSPs, this will ensure that pods in this namespace are no longer being modified or restricted by PodSecurityPolicy.

The advantage to disabling PodSecurityPolicy on a per-namespace basis like this is if a problem arises you can easily roll the change back by deleting the RoleBinding. Just make sure the pre-existing PodSecurityPolicies are still in place!

```
# Undo PodSecurityPolicy disablement.
kubectl delete -n $NAMESPACE rolebinding disable-psp
```

4. Review namespace creation processes

Now that existing namespaces have been updated to enforce Pod Security Admission, you should ensure that your processes and/or policies for creating new namespaces are updated to ensure that an appropriate Pod Security profile is applied to new namespaces.

You can also statically configure the Pod Security admission controller to set a default enforce, audit, and/or warn level for unlabeled namespaces. See Configure the Admission Controller for more information.

5. Disable PodSecurityPolicy

Finally, you're ready to disable PodSecurityPolicy. To do so, you will need to modify the admission configuration of the API server: How do I turn off an admission controller?.

To verify that the PodSecurityPolicy admission controller is no longer enabled, you can manually run a test by impersonating a user without access to any PodSecurityPolicies (see the PodSecurityPolicy example), or by verifying in the API server logs. At startup, the API server outputs log lines listing the loaded admission controller plugins:

```
I0218 00:59:44.903329      13 plugins.go:158] Loaded 16 mutating admission controller(s) successfully in the following order: Name
I0218 00:59:44.903350      13 plugins.go:161] Loaded 14 validating admission controller(s) successfully in the following order: Li
```

You should see `PodSecurity` (in the validating admission controllers), and neither list should contain `PodSecurityPolicy`.

Once you are certain the PSP admission controller is disabled (and after sufficient soak time to be confident you won't need to roll back), you are free to delete your PodSecurityPolicies and any associated Roles, ClusterRoles, RoleBindings and ClusterRoleBindings (just make sure they don't grant any other unrelated permissions).

Assign CPU Resources to Containers and Pods

This page shows how to assign a CPU *request* and a CPU *limit* to a container. Containers cannot use more CPU than the configured limit. Provided the system has CPU time free, a container is guaranteed to be allocated as much CPU as it requests.

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`.

Your cluster must have at least 1 CPU available for use to run the task examples.

A few of the steps on this page require you to run the metrics-server service in your cluster. If you have the metrics-server running, you can skip those steps.

If you are running Minikube, run the following command to enable metrics-server:

```
minikube addons enable metrics-server
```

To see whether metrics-server (or another provider of the resource metrics API, `metrics.k8s.io`) is running, type the following command:

```
kubectl get apiservices
```

If the resource metrics API is available, the output will include a reference to `metrics.k8s.io`.

```
NAME
v1beta1.metrics.k8s.io
```

Create a namespace

Create a Namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace cpu-example
```

Specify a CPU request and a CPU limit

To specify a CPU request for a container, include the `resources:requests` field in the Container resource manifest. To specify a CPU limit, include `resources:limits`.

In this exercise, you create a Pod that has one container. The container has a request of 0.5 CPU and a limit of 1 CPU. Here is the configuration file for the Pod:

`pods/resource/cpu-request-limit.yaml` Copy pods/resource/cpu-request-limit.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: cpu-demo  namespace: cpu-examplespec:  containers:  - name: cpu-demo-ctr    image: vish/stress    resour
```

The `args` section of the configuration file provides arguments for the container when it starts. The `-cpus "2"` argument tells the Container to attempt to use 2 CPUs.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-limit.yaml --namespace=cpu-example
```

Verify that the Pod is running:

```
kubectl get pod cpu-demo --namespace=cpu-example
```

View detailed information about the Pod:

```
kubectl get pod cpu-demo --output=yaml --namespace=cpu-example
```

The output shows that the one container in the Pod has a CPU request of 500 milliCPU and a CPU limit of 1 CPU.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 500m
```

Use `kubectl top` to fetch the metrics for the Pod:

```
kubectl top pod cpu-demo --namespace=cpu-example
```

This example output shows that the Pod is using 974 milliCPU, which is slightly less than the limit of 1 CPU specified in the Pod configuration.

```
NAME                        CPU(cores)   MEMORY(bytes)
cpu-demo                    974m         <something>
```

Recall that by setting `-cpu "2"`, you configured the Container to attempt to use 2 CPUs, but the Container is only being allowed to use about 1 CPU. The container's CPU use is being throttled, because the container is attempting to use more CPU resources than its limit.

Note:

Another possible explanation for the CPU use being below 1.0 is that the Node might not have enough CPU resources available. Recall that the prerequisites for this exercise require your cluster to have at least 1 CPU available for use. If your Container runs on a Node that has only 1 CPU, the Container cannot use more than 1 CPU regardless of the CPU limit specified for the Container.

CPU units

The CPU resource is measured in *CPU* units. One CPU, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading

Fractional values are allowed. A Container that requests 0.5 CPU is guaranteed half as much CPU as a Container that requests 1 CPU. You can use the suffix m to mean milli. For example 100m CPU, 100 milliCPU, and 0.1 CPU are all the same. Precision finer than 1m is not allowed.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

Delete your Pod:

```
kubectl delete pod cpu-demo --namespace=cpu-example
```

Specify a CPU request that is too big for your Nodes

CPU requests and limits are associated with Containers, but it is useful to think of a Pod as having a CPU request and limit. The CPU request for a Pod is the sum of the CPU requests for all the Containers in the Pod. Likewise, the CPU limit for a Pod is the sum of the CPU limits for all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough CPU resources available to satisfy the Pod CPU request.

In this exercise, you create a Pod that has a CPU request so big that it exceeds the capacity of any Node in your cluster. Here is the configuration file for a Pod that has one Container. The Container requests 100 CPU, which is likely to exceed the capacity of any Node in your cluster.

pods/resource/cpu-request-limit-2.yaml Copy pods/resource/cpu-request-limit-2.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: cpu-demo-2  namespace: cpu-examplespec:  containers:  - name: cpu-demo-ctr-2    image: vish/stress    re
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-limit-2.yaml --namespace=cpu-example
```

View the Pod status:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
```

The output shows that the Pod status is Pending. That is, the Pod has not been scheduled to run on any Node, and it will remain in the Pending state indefinitely:

```
NAME         READY     STATUS    RESTARTS   AGE
cpu-demo-2   0/1       Pending   0          7m
```

View detailed information about the Pod, including events:

```
kubectl describe pod cpu-demo-2 --namespace=cpu-example
```

The output shows that the Container cannot be scheduled because of insufficient CPU resources on the Nodes:

```
Events:
  Reason                        Message
  ------                        -------
  FailedScheduling      No nodes are available that match all of the following predicates:: Insufficient cpu (3).
```

Delete your Pod:

```
kubectl delete pod cpu-demo-2 --namespace=cpu-example
```

If you do not specify a CPU limit

If you do not specify a CPU limit for a Container, then one of these situations applies:

- The Container has no upper bound on the CPU resources it can use. The Container could use all of the CPU resources available on the Node where it is running.

- The Container is running in a namespace that has a default CPU limit, and the Container is automatically assigned the default limit. Cluster administrators can use a LimitRange to specify a default value for the CPU limit.

If you specify a CPU limit but do not specify a CPU request

If you specify a CPU limit for a Container but do not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit. Similarly, if a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit.

Motivation for CPU requests and limits

By configuring the CPU requests and limits of the Containers that run in your cluster, you can make efficient use of the CPU resources available on your cluster Nodes. By keeping a Pod CPU request low, you give the Pod a good chance of being scheduled. By having a CPU limit that is greater than the CPU request, you accomplish two things:

- The Pod can have bursts of activity where it makes use of CPU resources that happen to be available.
- The amount of CPU resources a Pod can use during a burst is limited to some reasonable amount.

Clean up

Delete your namespace:

```
kubectl delete namespace cpu-example
```

What's next

For app developers

- Assign Memory Resources to Containers and Pods
- Assign Pod-level CPU and memory resources
- Configure Quality of Service for Pods
- Resize CPU and Memory Resources assigned to Containers

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects
- Resize CPU and Memory Resources assigned to Containers

Share Process Namespace between Containers in a Pod

This page shows how to configure process namespace sharing for a pod. When process namespace sharing is enabled, processes in a container are visible to all other containers in the same pod.

You can use this feature to configure cooperating containers, such as a log handler sidecar container, or to troubleshoot container images that don't include debugging utilities like a shell.

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

Configure a Pod

Process namespace sharing is enabled using the `shareProcessNamespace` field of `.spec` for a Pod. For example:

`pods/share-process-namespace.yaml` Copy pods/share-process-namespace.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginxspec:  shareProcessNamespace: true  containers:  - name: nginx    image: nginx  - name: shell    ima
```

1. Create the pod `nginx` on your cluster:

   ```
   kubectl apply -f https://k8s.io/examples/pods/share-process-namespace.yaml
   ```

2. Attach to the `shell` container and run `ps`:

   ```
   kubectl exec -it nginx -c shell -- /bin/sh
   ```

   If you don't see a command prompt, try pressing enter. In the container shell:

   ```
   # run this inside the "shell" container
   ps ax
   ```

   The output is similar to this:

   ```
   PID   USER     TIME  COMMAND
       1 root      0:00 /pause
       8 root      0:00 nginx: master process nginx -g daemon off;
      14 101       0:00 nginx: worker process
      15 root      0:00 sh
      21 root      0:00 ps ax
   ```

You can signal processes in other containers. For example, send `SIGHUP` to `nginx` to restart the worker process. This requires the `SYS_PTRACE` capability.

```
# run this inside the "shell" container
kill -HUP 8   # change "8" to match the PID of the nginx leader process, if necessary
ps ax
```

The output is similar to this:

```
PID   USER     TIME  COMMAND
    1 root      0:00 /pause
    8 root      0:00 nginx: master process nginx -g daemon off;
   15 root      0:00 sh
   22 101       0:00 nginx: worker process
   23 root      0:00 ps ax
```

It's even possible to access the file system of another container using the `/proc/$pid/root` link.

```
# run this inside the "shell" container
# change "8" to the PID of the Nginx process, if necessary
head /proc/8/root/etc/nginx/nginx.conf
```

The output is similar to this:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
```

Understanding process namespace sharing

Pods share many resources so it makes sense they would also share a process namespace. Some containers may expect to be isolated from others, though, so it's important to understand the differences:

1. **The container process no longer has PID 1.** Some containers refuse to start without PID 1 (for example, containers using `systemd`) or run commands like `kill -HUP 1` to signal the container process. In pods with a shared process namespace, `kill -HUP 1` will signal the pod sandbox (`/pause` in the above example).

2. **Processes are visible to other containers in the pod.** This includes all information visible in `/proc`, such as passwords that were passed as arguments or environment variables. These are protected only by regular Unix permissions.

3. **Container filesystems are visible to other containers in the pod through the `/proc/$pid/root` link.** This makes debugging easier, but it also means that filesystem secrets are protected only by filesystem permissions.

Resize CPU and Memory Resources assigned to Containers

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

This page explains how to change the CPU and memory resource requests and limits assigned to a container *without recreating the Pod*.

Traditionally, changing a Pod's resource requirements necessitated deleting the existing Pod and creating a replacement, often managed by a workload controller. In-place Pod Resize allows changing the CPU/memory allocation of container(s) within a running Pod while potentially avoiding application disruption.

Key Concepts:

- **Desired Resources:** A container's `spec.containers[*].resources` represent the *desired* resources for the container, and are mutable for CPU and memory.
- **Actual Resources:** The `status.containerStatuses[*].resources` field reflects the resources *currently configured* for a running container. For containers that haven't started or were restarted, it reflects the resources allocated upon their next start.
- **Triggering a Resize:** You can request a resize by updating the desired `requests` and `limits` in the Pod's specification. This is typically done using `kubectl patch`, `kubectl apply`, or `kubectl edit` targeting the Pod's `resize` subresource. When the desired resources don't match the allocated resources, the Kubelet will attempt to resize the container.

- **Allocated Resources (Advanced):** The `status.containerStatuses[*].allocatedResources` field tracks resource values confirmed by the Kubelet, primarily used for internal scheduling logic. For most monitoring and validation purposes, focus on `status.containerStatuses[*].resources`.

If a node has pods with a pending or incomplete resize (see Pod Resize Status below), the scheduler uses the *maximum* of a container's desired requests, allocated requests, and actual requests from the status when making scheduling decisions.

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`.

The `InPlacePodVerticalScaling` feature gate must be enabled for your control plane and for all nodes in your cluster.

The `kubectl` client version must be at least v1.32 to use the `--subresource=resize` flag.

Pod resize status

The Kubelet updates the Pod's status conditions to indicate the state of a resize request:

- `type: PodResizePending`: The Kubelet cannot immediately grant the request. The `message` field provides an explanation of why.
  - `reason: Infeasible`: The requested resize is impossible on the current node (for example, requesting more resources than the node has).
  - `reason: Deferred`: The requested resize is currently not possible, but might become feasible later (for example if another pod is removed). The Kubelet will retry the resize.
- `type: PodResizeInProgress`: The Kubelet has accepted the resize and allocated resources, but the changes are still being applied. This is usually brief but might take longer depending on the resource type and runtime behavior. Any errors during actuation are reported in the `message` field (along with `reason: Error`).

How kubelet retries Deferred resizes

If the requested resize is *Deferred*, the kubelet will periodically re-attempt the resize, for example when another pod is removed or scaled down. If there are multiple deferred resizes, they are retried according to the following priority:

- Pods with a higher Priority (based on PriorityClass) will have their resize request retried first.
- If two pods have the same Priority, resize of guaranteed pods will be retried before the resize of burstable pods.
- If all else is the same, pods that have been in the Deferred state longer will be prioritized.

A higher priority resize being marked as pending will not block the remaining pending resizes from being attempted; all remaining pending resizes will still be retried even if a higher-priority resize gets deferred again.

Leveraging `observedGeneration` Fields

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

- The top-level `status.observedGeneration` field shows the `metadata.generation` corresponding to the latest pod specification that the kubelet has acknowledged. You can use this to determine the most recent resize request the kubelet has processed.
- In the `PodResizeInProgress` condition, the `conditions[].observedGeneration` field indicates the `metadata.generation` of the podSpec when the current in-progress resize was initiated.
- In the `PodResizePending` condition, the `conditions[].observedGeneration` field indicates the `metadata.generation` of the podSpec when the pending resize's allocation was last attempted.

Container resize policies

You can control whether a container should be restarted when resizing by setting `resizePolicy` in the container specification. This allows fine-grained control based on resource type (CPU or memory).

```
resizePolicy:
- resourceName: cpu
  restartPolicy: NotRequired
- resourceName: memory
  restartPolicy: RestartContainer
```

- `NotRequired`: (Default) Apply the resource change to the running container without restarting it.
- `RestartContainer`: Restart the container to apply the new resource values. This is often necessary for memory changes because many applications and runtimes cannot adjust their memory allocation dynamically.

If `resizePolicy[*].restartPolicy` is not specified for a resource, it defaults to `NotRequired`.

Note:

If a Pod's overall `restartPolicy` is `Never`, then any container `resizePolicy` must be `NotRequired` for all resources. You cannot configure a resize policy that would require a restart in such Pods.

Example Scenario:

Consider a container configured with `restartPolicy: NotRequired` for CPU and `restartPolicy: RestartContainer` for memory.

- If only CPU resources are changed, the container is resized in-place.
- If only memory resources are changed, the container is restarted.
- If *both* CPU and memory resources are changed simultaneously, the container is restarted (due to the memory policy).

Limitations

For Kubernetes 1.34, resizing pod resources in-place has the following limitations:

- **Resource Types:** Only CPU and memory resources can be resized.
- **Memory Decrease:** If the memory resize restart policy is `NotRequired` (or unspecified), the kubelet will make a best-effort attempt to prevent oom-kills when decreasing memory limits, but doesn't provide any guarantees. Before decreasing container memory limits, if memory usage exceeds the requested limit, the resize will be skipped and the status will remain in an "In Progress" state. This is considered best-effort because it is still subject to a race condition where memory usage may spike right after the check is performed.
- **QoS Class:** The Pod's original Quality of Service (QoS) class (Guaranteed, Burstable, or BestEffort) is determined at creation and **cannot** be changed by a resize. The resized resource values must still adhere to the rules of the original QoS class:
    - *Guaranteed*: Requests must continue to equal limits for both CPU and memory after resizing.
    - *Burstable*: Requests and limits cannot become equal for *both* CPU and memory simultaneously (as this would change it to Guaranteed).
    - *BestEffort*: Resource requirements (`requests` or `limits`) cannot be added (as this would change it to Burstable or Guaranteed).
- **Container Types:** Non-restartable init containers and ephemeral containers cannot be resized. Sidecar containers can be resized.
- **Resource Removal:** Resource requests and limits cannot be entirely removed once set; they can only be changed to different values.
- **Operating System:** Windows pods do not support in-place resize.
- **Node Policies:** Pods managed by static CPU or Memory manager policies cannot be resized in-place.
- **Swap:** Pods utilizing swap memory cannot resize memory requests unless the `resizePolicy` for memory is `RestartContainer`.

These restrictions might be relaxed in future Kubernetes versions.

Example 1: Resizing CPU without restart

First, create a Pod designed for in-place CPU resize and restart-required memory resize.

`pods/resource/pod-resize.yaml` Copy pods/resource/pod-resize.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: resize-demospec:  containers:  - name: pause    image: registry.k8s.io/pause:3.8    resizePolicy:    - r
```

Create the pod:

```
kubectl create -f pod-resize.yaml
```

This pod starts in the Guaranteed QoS class. Verify its initial state:

```
# Wait a moment for the pod to be running
kubectl get pod resize-demo --output=yaml
```

Observe the `spec.containers[0].resources` and `status.containerStatuses[0].resources`. They should match the manifest (700m CPU, 200Mi memory). Note the `status.containerStatuses[0].restartCount` (should be 0).

Now, increase the CPU request and limit to `800m`. You use `kubectl patch` with the `--subresource resize` command line argument.

```
kubectl patch pod resize-demo --subresource resize --patch \
  '{"spec":{"containers":[{"name":"pause", "resources":{"requests":{"cpu":"800m"}, "limits":{"cpu":"800m"}}}]}}'# Alternative meth
```

Note:

The `--subresource resize` command line argument requires `kubectl` client version v1.32.0 or later. Older versions will report an `invalid subresource` error.

Check the pod status again after patching:

```
kubectl get pod resize-demo --output=yaml --namespace=qos-example
```

You should see:

- `spec.containers[0].resources` now shows `cpu: 800m`.
- `status.containerStatuses[0].resources` also shows `cpu: 800m`, indicating the resize was successful on the node.
- `status.containerStatuses[0].restartCount` remains `0`, because the CPU `resizePolicy` was `NotRequired`.

Example 2: Resizing memory with restart

Now, resize the memory for the *same* pod by increasing it to `300Mi`. Since the memory `resizePolicy` is `RestartContainer`, the container is expected to restart.

```
kubectl patch pod resize-demo --subresource resize --patch \
  '{"spec":{"containers":[{"name":"pause", "resources":{"requests":{"memory":"300Mi"}, "limits":{"memory":"300Mi"}}}]}}'
```

Check the pod status shortly after patching:

```
kubectl get pod resize-demo --output=yaml
```

You should now observe:

- `spec.containers[0].resources` shows `memory: 300Mi`.
- `status.containerStatuses[0].resources` also shows `memory: 300Mi`.
- `status.containerStatuses[0].restartCount` has increased to `1` (or more, if restarts happened previously), indicating the container was restarted to apply the memory change.

Troubleshooting: Infeasible resize request

Next, try requesting an unreasonable amount of CPU, such as 1000 full cores (written as `"1000"` instead of `"1000m"` for millicores), which likely exceeds node capacity.

```
# Attempt to patch with an excessively large CPU request
kubectl patch pod resize-demo --subresource resize --patch \
  '{"spec":{"containers":[{"name":"pause", "resources":{"requests":{"cpu":"1000"}, "limits":{"cpu":"1000"}}}]}}'
```

Query the Pod's details:

```
kubectl get pod resize-demo --output=yaml
```

You'll see changes indicating the problem:

- The `spec.containers[0].resources` reflects the *desired* state (`cpu: "1000"`).
- A condition with `type: PodResizePending` and `reason: Infeasible` was added to the Pod.
- The condition's `message` will explain why (`Node didn't have enough capacity: cpu, requested: 800000, capacity: ...`)
- Crucially, `status.containerStatuses[0].resources` will *still show the previous values* (`cpu: 800m`, `memory: 300Mi`), because the infeasible resize was not applied by the Kubelet.
- The `restartCount` will not have changed due to this failed attempt.

To fix this, you would need to patch the pod again with feasible resource values.

Clean up

Delete the pod:

```
kubectl delete pod resize-demo
```

What's next

For application developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Assign Pod-level CPU and memory resources

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace

Configure a Pod to Use a PersistentVolume for Storage

This page shows you how to configure a Pod to use a PersistentVolumeClaim for storage. Here is a summary of the process:

1. You, as cluster administrator, create a PersistentVolume backed by physical storage. You do not associate the volume with any Pod.
2. You, now taking the role of a developer / cluster user, create a PersistentVolumeClaim that is automatically bound to a suitable PersistentVolume.
3. You create a Pod that uses the above PersistentVolumeClaim for storage.

Before you begin

- You need to have a Kubernetes cluster that has only one Node, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a single-node cluster, you can create one by using Minikube.

- Familiarize yourself with the material in Persistent Volumes.

Create an index.html file on your Node

Open a shell to the single Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering `minikube ssh`.

In your shell on that Node, create a `/mnt/data` directory:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo mkdir /mnt/data
```

In the `/mnt/data` directory, create an `index.html` file:

```
# This again assumes that your Node uses "sudo" to run commands
# as the superuser
sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/index.html"
```

Note:

If your Node uses a tool for superuser access other than `sudo`, you can usually make this work if you replace `sudo` with the name of the other tool.

Test that the `index.html` file exists:

```
cat /mnt/data/index.html
```

The output should be:

```
Hello from Kubernetes storage
```

You can now close the shell to your Node.

Create a PersistentVolume

In this exercise, you create a *hostPath* PersistentVolume. Kubernetes supports hostPath for development and testing on a single-node cluster. A hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, you would not use hostPath. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume. Cluster administrators can also use StorageClasses to set up dynamic provisioning.

Here is the configuration file for the hostPath PersistentVolume:

`pods/storage/pv-volume.yaml` Copy pods/storage/pv-volume.yaml to clipboard

```
apiVersion: v1
kind: PersistentVolumemetadata:  name: task-pv-volume  labels:    type: localspec:  storageClassName: manual  capacity:    storage
```

The configuration file specifies that the volume is at `/mnt/data` on the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of `ReadWriteOnce`, which means the volume can be mounted as read-write by a single Node. It defines the StorageClass name `manual` for the PersistentVolume, which will be used to bind PersistentVolumeClaim requests to this PersistentVolume.

Note:

This example uses the `ReadWriteOnce` access mode, for simplicity. For production use, the Kubernetes project recommends using the `ReadWriteOncePod` access mode instead.

Create the PersistentVolume:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml
```

View information about the PersistentVolume:

```
kubectl get pv task-pv-volume
```

The output shows that the PersistentVolume has a `STATUS` of `Available`. This means it has not yet been bound to a PersistentVolumeClaim.

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS      CLAIM     STORAGECLASS   REASON    AGE
task-pv-volume   10Gi       RWO           Retain          Available             manual                   4s
```

Create a PersistentVolumeClaim

The next step is to create a PersistentVolumeClaim. Pods use PersistentVolumeClaims to request physical storage. In this exercise, you create a PersistentVolumeClaim that requests a volume of at least three gibibytes that can provide read-write access for at most one Node at a time.

Here is the configuration file for the PersistentVolumeClaim:

`pods/storage/pv-claim.yaml` Copy pods/storage/pv-claim.yaml to clipboard

```
apiVersion: v1
kind: PersistentVolumeClaimmetadata:  name: task-pv-claimspec:  storageClassName: manual  accessModes:    - ReadWriteOnce  resourc
```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-claim.yaml
```

After you create the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim's requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Look again at the PersistentVolume:

```
kubectl get pv task-pv-volume
```

Now the output shows a `STATUS` of `Bound`.

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS   CLAIM                   STORAGECLASS   REASON    AGE
task-pv-volume   10Gi       RWO           Retain          Bound    default/task-pv-claim   manual                   2m
```

Look at the PersistentVolumeClaim:

```
kubectl get pvc task-pv-claim
```

The output shows that the PersistentVolumeClaim is bound to your PersistentVolume, `task-pv-volume`.

```
NAME            STATUS    VOLUME           CAPACITY   ACCESSMODES   STORAGECLASS   AGE
task-pv-claim   Bound     task-pv-volume   10Gi       RWO           manual         30s
```

Create a Pod

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod:

pods/storage/pv-pod.yaml Copy pods/storage/pv-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: task-pv-podspec:  volumes:    - name: task-pv-storage      persistentVolumeClaim:        claimName: task-
```

Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
```

Verify that the container in the Pod is running:

```
kubectl get pod task-pv-pod
```

Get a shell to the container running in your Pod:

```
kubectl exec -it task-pv-pod -- /bin/bash
```

In your shell, verify that nginx is serving the `index.html` file from the hostPath volume:

```
# Be sure to run these 3 commands inside the root shell that comes from
# running "kubectl exec" in the previous step
apt update
apt install curl
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file on the hostPath volume:

```
Hello from Kubernetes storage
```

If you see that message, you have successfully configured a Pod to use storage from a PersistentVolumeClaim.

Clean up

Delete the Pod:

```
kubectl delete pod task-pv-pod
```

Mounting the same PersistentVolume in two places

You have understood how to create a PersistentVolume & PersistentVolumeClaim, and how to mount the volume to a single location in a container. Let's explore how you can mount the same PersistentVolume at two different locations in a container. Below is an example:

pods/storage/pv-duplicate.yaml Copy pods/storage/pv-duplicate.yaml to clipboard

```
apiVersion: v1kind: Podmetadata:  name: testspec:  containers:    - name: test      image: nginx      volumeMounts:        # a mou
```

Here:

- `subPath`: This field allows specific files or directories from the mounted PersistentVolume to be exposed at different locations within the container. In this example:
    - `subPath: html` mounts the html directory.
    - `subPath: nginx.conf` mounts a specific file, nginx.conf.

Since the first subPath is `html`, an `html` directory has to be created within `/mnt/data/` on the node.

The second subPath `nginx.conf` means that a file within the `/mnt/data/` directory will be used. No other directory needs to be created.

Two volume mounts will be made on your nginx container:

- `/usr/share/nginx/html` for the static website
- `/etc/nginx/nginx.conf` for the default config

Move the index.html file on your Node to a new folder

The `index.html` file mentioned here refers to the one created in the "Create an index.html file on your Node" section.

Open a shell to the single Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering `minikube ssh`.

Create a `/mnt/data/html` directory:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo mkdir /mnt/data/html
```

Move index.html into the directory:

```
# Move index.html from its current location to the html sub-directory
sudo mv /mnt/data/index.html html
```

Create a new nginx.conf file

pods/storage/nginx.conf Copy pods/storage/nginx.conf to clipboard

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;



events {
worker_connections  1024;
}



http {
include       /etc/nginx/mime.types;
default_type  application/octet-stream;



log_format  main  &#39;$remote_addr - $remote_user [$time_local] &#34;$request&#34; &#39;
                  &#39;$status $body_bytes_sent &#34;$http_referer&#34; &#39;
                  &#39;&#34;$http_user_agent&#34; &#34;$http_x_forwarded_for&#34;&#39;;

access_log  /var/log/nginx/access.log  main;

sendfile        on;
#tcp_nopush     on;

keepalive_timeout  60;

#gzip  on;

include /etc/nginx/conf.d/*.conf;

}
```

This is a modified version of the default `nginx.conf` file. Here, the default `keepalive_timeout` has been modified to `60`

Create the nginx.conf file:

```
cat <<EOF > /mnt/data/nginx.conf
user  nginx;
worker_processes  auto;
error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '\$remote_addr - \$remote_user [\$time_local] "\$request" '
                      '\$status \$body_bytes_sent "\$http_referer" '
                      '"\$http_user_agent" "\$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  60;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
}
EOF
```

Create a Pod

Here we will create a pod that uses the existing persistentVolume and persistentVolumeClaim. However, the pod mounts only a specific file, `nginx.conf`, and directory, `html`, to the container.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-duplicate.yaml
```

Verify that the container in the Pod is running:

```
kubectl get pod test
```

Get a shell to the container running in your Pod:

```
kubectl exec -it test -- /bin/bash
```

In your shell, verify that nginx is serving the `index.html` file from the hostPath volume:

```
# Be sure to run these 3 commands inside the root shell that comes from
# running "kubectl exec" in the previous step
apt update
apt install curl
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file on the hostPath volume:

```
Hello from Kubernetes storage
```

In your shell, also verify that nginx is serving the `nginx.conf` file from the hostPath volume:

```
# Be sure to run these commands inside the root shell that comes from
# running "kubectl exec" in the previous step
cat /etc/nginx/nginx.conf | grep keepalive_timeout
```

The output shows the modified text that you wrote to the `nginx.conf` file on the hostPath volume:

```
keepalive_timeout  60;
```

If you see these messages, you have successfully configured a Pod to use a specific file and directory in a storage from a PersistentVolumeClaim.

Clean up

Delete the Pod:

```
kubectl delete pod test
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

If you don't already have a shell open to the Node in your cluster, open a new shell the same way that you did earlier.

In the shell on your Node, remove the file and directory that you created:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo rm /mnt/data/html/index.html
sudo rm /mnt/data/nginx.conf
sudo rmdir /mnt/data
```

You can now close the shell to your Node.

Access control

Storage configured with a group ID (GID) allows writing only by Pods using the same GID. Mismatched or missing GIDs cause permission denied errors. To reduce the need for coordination with users, an administrator can annotate a PersistentVolume with a GID. Then the GID is automatically added to any Pod that uses the PersistentVolume.

Use the `pv.beta.kubernetes.io/gid` annotation as follows:

```
apiVersion: v1
kind: PersistentVolumemetadata:  name: pv1  annotations:    pv.beta.kubernetes.io/gid: "1234"
```

When a Pod consumes a PersistentVolume that has a GID annotation, the annotated GID is applied to all containers in the Pod in the same way that GIDs specified in the Pod's security context are. Every GID, whether it originates from a PersistentVolume annotation or the Pod's specification, is applied to the first process run in each container.

Note:

When a Pod consumes a PersistentVolume, the GIDs associated with the PersistentVolume are not present on the Pod resource itself.

What's next

- Learn more about PersistentVolumes.
- Read the Persistent Storage design document.

Reference

- PersistentVolume
- PersistentVolumeSpec
- PersistentVolumeClaim
- PersistentVolumeClaimSpec

Configure a Security Context for a Pod or Container

A security context defines privilege and access control settings for a Pod or Container. Security context settings include, but are not limited to:

- Discretionary Access Control: Permission to access an object, like a file, is based on user ID (UID) and group ID (GID).

- Security Enhanced Linux (SELinux): Objects are assigned security labels.

- Running as privileged or unprivileged.

- Linux Capabilities: Give a process some privileges, but not all the privileges of the root user.

- AppArmor: Use program profiles to restrict the capabilities of individual programs.

- Seccomp: Filter a process's system calls.

- `allowPrivilegeEscalation`: Controls whether a process can gain more privileges than its parent process. This bool directly controls whether the `no_new_privs` flag gets set on the container process. `allowPrivilegeEscalation` is always true when the container:

  - is run as privileged, or
  - has `CAP_SYS_ADMIN`

- `readOnlyRootFilesystem`: Mounts the container's root filesystem as read-only.

The above bullets are not a complete set of security context settings -- please see SecurityContext for a comprehensive list.

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`.

Set the security context for a Pod

To specify security settings for a Pod, include the `securityContext` field in the Pod specification. The `securityContext` field is a PodSecurityContext object. The security settings that you specify for a Pod apply to all Containers in the Pod. Here is a configuration file for a Pod that has a `securityContext` and an `emptyDir` volume:

`pods/security/security-context.yaml` Copy pods/security/security-context.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: security-context-demospec:  securityContext:    runAsUser: 1000    runAsGroup: 3000    fsGroup: 2000
```

In the configuration file, the `runAsUser` field specifies that for any Containers in the Pod, all processes run with user ID 1000. The `runAsGroup` field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is omitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when `runAsGroup` is specified. Since `fsGroup` field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume `/data/demo` and any files created in that volume will be Group ID 2000. Additionally, when the `supplementalGroups` field is specified, all processes of the container are also part of the specified groups. If this field is omitted, it means empty.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

In your shell, list the running processes:

```
ps
```

The output shows that the processes are running as user 1000, which is the value of `runAsUser`:

```
PID   USER     TIME  COMMAND
    1 1000      0:00 sleep 1h
    6 1000      0:00 sh
...
```

In your shell, navigate to `/data`, and list the one directory:

```
cd /data
ls -l
```

The output shows that the `/data/demo` directory has group ID 2000, which is the value of `fsGroup`.

```
drwxrwsrwx 2 root 2000 4096 Jun  6 20:08 demo
```

In your shell, navigate to `/data/demo`, and create a file:

```
cd demo
echo hello > testfile
```

List the file in the `/data/demo` directory:

```
ls -l
```

The output shows that `testfile` has group ID 2000, which is the value of `fsGroup`.

```
-rw-r--r-- 1 1000 2000 6 Jun  6 20:08 testfile
```

Run the following command:

```
id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=2000,3000,4000
```

From the output, you can see that `gid` is 3000 which is same as the `runAsGroup` field. If the `runAsGroup` was omitted, the `gid` would remain as 0 (root) and the process will be able to interact with files that are owned by the root(0) group and groups that have the required group permissions for the root (0) group. You can also see that `groups` contains the group IDs which are specified by `fsGroup` and `supplementalGroups`, in addition to `gid`.

Exit your shell:

```
exit
```

Implicit group memberships defined in `/etc/group` in the container image

By default, kubernetes merges group information from the Pod with information defined in `/etc/group` in the container image.

pods/security/security-context-5.yaml Copy pods/security/security-context-5.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: security-context-demospec:  securityContext:    runAsUser: 1000    runAsGroup: 3000    supplementalGroup
```

This Pod security context contains `runAsUser`, `runAsGroup` and `supplementalGroups`. However, you can see that the actual supplementary groups attached to the container process will include group IDs which come from `/etc/group` in the container image.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-5.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

Check the process identity:

```
id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=3000,4000,50000
```

You can see that `groups` includes group ID `50000`. This is because the user (`uid=1000`), which is defined in the image, belongs to the group (`gid=50000`), which is defined in `/etc/group` inside the container image.

Check the `/etc/group` in the container image:

```
cat /etc/group
```

You can see that uid `1000` belongs to group `50000`.

```
...
user-defined-in-image:x:1000:
group-defined-in-image:x:50000:user-defined-in-image
```

Exit your shell:

```
exit
```

Note:

Implicitly merged supplementary groups may cause security problems particularly when accessing the volumes (see kubernetes/kubernetes#112879 for details). If you want to avoid this. Please see the below section.

Configure fine-grained SupplementalGroups control for a Pod

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

This feature can be enabled by setting the `SupplementalGroupsPolicy` feature gate for kubelet and kube-apiserver, and setting the `.spec.securityContext.supplementalGroupsPolicy` field for a pod.

The `supplementalGroupsPolicy` field defines the policy for calculating the supplementary groups for the container processes in a pod. There are two valid values for this field:

- `Merge`: The group membership defined in `/etc/group` for the container's primary user will be merged. This is the default policy if not specified.
- `Strict`: Only group IDs in `fsGroup`, `supplementalGroups`, or `runAsGroup` fields are attached as the supplementary groups of the container processes. This means no group membership from `/etc/group` for the container's primary user will be merged.

When the feature is enabled, it also exposes the process identity attached to the first container process in `.status.containerStatuses[].user.linux` field. It would be useful for detecting if implicit group ID's are attached.

`pods/security/security-context-6.yaml` Copy pods/security/security-context-6.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: security-context-demospec:  securityContext:    runAsUser: 1000    runAsGroup: 3000    supplementalGroup
```

This pod manifest defines `supplementalGroupsPolicy=Strict`. You can see that no group memberships defined in `/etc/group` are merged to the supplementary groups for container processes.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-6.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Check the process identity:

```
kubectl exec -it security-context-demo -- id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=3000,4000
```

See the Pod's status:

```
kubectl get pod security-context-demo -o yaml
```

You can see that the `status.containerStatuses[].user.linux` field exposes the process identity attached to the first container process.

```
...
status:
  containerStatuses:
  - name: sec-ctx-demo
    user:
      linux:
        gid: 3000
        supplementalGroups:
        - 3000
        - 4000
        uid: 1000
...
```

Note:

Please note that the values in the `status.containerStatuses[].user.linux` field is *the first attached* process identity to the first container process in the container. If the container has sufficient privilege to make system calls related to process identity (e.g. `setuid(2)`, `setgid(2)` or `setgroups(2)`, etc.), the container process can change its identity. Thus, the *actual* process identity will be dynamic.

Implementations

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.

The following container runtimes are known to support fine-grained SupplementalGroups control.

CRI-level:

- containerd, since v2.0
- CRI-O, since v1.31

You can see if the feature is supported in the Node status.

```
apiVersion: v1
kind: Node...status:  features:    supplementalGroupsPolicy: true
```

Note:

At this alpha release(from v1.31 to v1.32), when a pod with `SupplementalGroupsPolicy=Strict` are scheduled to a node that does NOT support this feature(i.e. `.status.features.supplementalGroupsPolicy=false`), the pod's supplemental groups policy falls back to the `Merge` policy *silently*.

However, since the beta release (v1.33), to enforce the policy more strictly, **such pod creation will be rejected by kubelet because the node cannot ensure the specified policy**. When your pod is rejected, you will see warning events with `reason=SupplementalGroupsPolicyNotSupported` like below:

```
apiVersion: v1
kind: Event...type: Warningreason: SupplementalGroupsPolicyNotSupportedmessage: "SupplementalGroupsPolicy=Strict is not supported
```

Configure volume permission and ownership change policy for Pods

FEATURE STATE: `Kubernetes v1.23 [stable]`

By default, Kubernetes recursively changes ownership and permissions for the contents of each volume to match the `fsGroup` specified in a Pod's `securityContext` when that volume is mounted. For large volumes, checking and changing ownership and permissions can take a lot of time, slowing Pod startup. You can use the `fsGroupChangePolicy` field inside a `securityContext` to control the way that Kubernetes checks and manages ownership and permissions for a volume.

fsGroupChangePolicy - `fsGroupChangePolicy` defines behavior for changing ownership and permission of the volume before being exposed inside a Pod. This field only applies to volume types that support `fsGroup` controlled ownership and permissions. This field has two possible values:

- *OnRootMismatch*: Only change permissions and ownership if the permission and the ownership of root directory does not match with expected permissions of the volume. This could help shorten the time it takes to change ownership and permission of a volume.
- *Always*: Always change permission and ownership of the volume when volume is mounted.

For example:

```
securityContext:
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
  fsGroupChangePolicy: "OnRootMismatch"
```

Note:

This field has no effect on ephemeral volume types such as `secret`, `configMap`, and `emptyDir`.

Delegating volume permission and ownership change to CSI driver

FEATURE STATE: `Kubernetes v1.26 [stable]`

If you deploy a Container Storage Interface (CSI) driver which supports the `VOLUME_MOUNT_GROUP NodeServiceCapability`, the process of setting file ownership and permissions based on the `fsGroup` specified in the `securityContext` will be performed by the CSI driver instead of Kubernetes. In this case, since Kubernetes doesn't perform any ownership and permission change, `fsGroupChangePolicy` does not take effect, and as specified by CSI, the driver is expected to mount the volume with the provided `fsGroup`, resulting in a volume that is readable/writable by the `fsGroup`.

Set the security context for a Container

To specify security settings for a Container, include the `securityContext` field in the Container manifest. The `securityContext` field is a SecurityContext object. Security settings that you specify for a Container apply only to the individual Container, and they override settings made at the Pod level when there is overlap. Container settings do not affect the Pod's Volumes.

Here is the configuration file for a Pod that has one Container. Both the Pod and the Container have a `securityContext` field:

`pods/security/security-context-2.yaml` Copy pods/security/security-context-2.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: security-context-demo-2spec:  securityContext:    runAsUser: 1000  containers:  - name: sec-ctx-demo-2
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-2.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-2
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-2 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows that the processes are running as user 2000. This is the value of `runAsUser` specified for the Container. It overrides the value 1000 that is specified for the Pod.

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
2000         1  0.0  0.0   4336   764 ?        Ss   20:36   0:00 /bin/sh -c node server.js
2000         8  0.1  0.5 772124 22604 ?        Sl   20:36   0:00 node server.js
...
```

Exit your shell:

```
exit
```

Set capabilities for a Container

With [Linux capabilities](), you can grant certain privileges to a process without granting all the privileges of the root user. To add or remove Linux capabilities for a Container, include the `capabilities` field in the `securityContext` section of the Container manifest.

First, see what happens when you don't include a `capabilities` field. Here is configuration file that does not add or remove any Container capabilities:

`pods/security/security-context-3.yaml` Copy pods/security/security-context-3.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: security-context-demo-3spec:  containers:  - name: sec-ctx-3    image: gcr.io/google-samples/hello-app:2
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-3.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-3
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-3 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows the process IDs (PIDs) for the Container:

```
USER  PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
root    1  0.0  0.0   4336   796 ?     Ss   18:17   0:00 /bin/sh -c node server.js
root    5  0.1  0.5 772124 22700 ?     Sl   18:17   0:00 node server.js
```

In your shell, view the status for process 1:

```
cd /proc/1
cat status
```

The output shows the capabilities bitmap for the process:

```
...
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
...
```

Make a note of the capabilities bitmap, and then exit your shell:

```
exit
```

Next, run a Container that is the same as the preceding container, except that it has additional capabilities set.

Here is the configuration file for a Pod that runs one Container. The configuration adds the `CAP_NET_ADMIN` and `CAP_SYS_TIME` capabilities:

`pods/security/security-context-4.yaml` Copy pods/security/security-context-4.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: security-context-demo-4spec:  containers:  - name: sec-ctx-4    image: gcr.io/google-samples/hello-app:2
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-4.yaml
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-4 -- sh
```

In your shell, view the capabilities for process 1:

```
cd /proc/1
cat status
```

The output shows capabilities bitmap for the process:

```
...
CapPrm: 00000000aa0435fb
CapEff: 00000000aa0435fb
...
```

Compare the capabilities of the two Containers:

```
00000000a80425fb
00000000aa0435fb
```

In the capability bitmap of the first container, bits 12 and 25 are clear. In the second container, bits 12 and 25 are set. Bit 12 is `CAP_NET_ADMIN`, and bit 25 is `CAP_SYS_TIME`. See capability.h for definitions of the capability constants.

Note:

Linux capability constants have the form `CAP_XXX`. But when you list capabilities in your container manifest, you must omit the `CAP_` portion of the constant. For example, to add `CAP_SYS_TIME`, include `SYS_TIME` in your list of capabilities.

Set the Seccomp Profile for a Container

To set the Seccomp profile for a Container, include the `seccompProfile` field in the `securityContext` section of your Pod or Container manifest. The `seccompProfile` field is a SeccompProfile object consisting of `type` and `localhostProfile`. Valid options for `type` include `RuntimeDefault`, `Unconfined`, and `Localhost`. `localhostProfile` must only be set if `type: Localhost`. It indicates the path of the pre-configured profile on the node, relative to the kubelet's configured Seccomp profile location (configured with the `--root-dir` flag).

Here is an example that sets the Seccomp profile to the node's container runtime default profile:

```
...
securityContext:   seccompProfile:    type: RuntimeDefault
```

Here is an example that sets the Seccomp profile to a pre-configured file at `<kubelet-root-dir>/seccomp/my-profiles/profile-allow.json`:

```
...
securityContext:   seccompProfile:    type: Localhost    localhostProfile: my-profiles/profile-allow.json
```

Set the AppArmor Profile for a Container

To set the AppArmor profile for a Container, include the `appArmorProfile` field in the `securityContext` section of your Container. The `appArmorProfile` field is a AppArmorProfile object consisting of `type` and `localhostProfile`. Valid options for `type` include `RuntimeDefault`(default), `Unconfined`, and `Localhost`. `localhostProfile` must only be set if `type` is `Localhost`. It indicates the name of the pre-configured profile on the node. The profile needs to be loaded onto all nodes suitable for the Pod, since you don't know where the pod will be scheduled. Approaches for setting up custom profiles are discussed in Setting up nodes with profiles.

Note: If `containers[*].securityContext.appArmorProfile.type` is explicitly set to `RuntimeDefault`, then the Pod will not be admitted if AppArmor is not enabled on the Node. However if `containers[*].securityContext.appArmorProfile.type` is not specified, then the default (which is also `RuntimeDefault`) will only be applied if the node has AppArmor enabled. If the node has AppArmor disabled the Pod will be admitted but the Container will not be restricted by the `RuntimeDefault` profile.

Here is an example that sets the AppArmor profile to the node's container runtime default profile:

```
...
containers:- name: container-1  securityContext:    appArmorProfile:      type: RuntimeDefault
```

Here is an example that sets the AppArmor profile to a pre-configured profile named `k8s-apparmor-example-deny-write`:

```
...
containers:- name: container-1  securityContext:    appArmorProfile:      type: Localhost      localhostProfile: k8s-apparmor-exam
```

For more details please see, Restrict a Container's Access to Resources with AppArmor.

Assign SELinux labels to a Container

To assign SELinux labels to a Container, include the `seLinuxOptions` field in the `securityContext` section of your Pod or Container manifest. The `seLinuxOptions` field is an SELinuxOptions object. Here's an example that applies an SELinux level:

```
...
securityContext:   seLinuxOptions:    level: "s0:c123,c456"
```

Note:

To assign SELinux labels, the SELinux security module must be loaded on the host operating system. On Windows and Linux worker nodes without SELinux support, this field and any SELinux feature gates described below have no effect.

Efficient SELinux volume relabeling

FEATURE STATE: `Kubernetes v1.28 [beta]` (enabled by default: true)

Note:

Kubernetes v1.27 introduced an early limited form of this behavior that was only applicable to volumes (and PersistentVolumeClaims) using the `ReadWriteOncePod` access mode.

Kubernetes v1.33 promotes `SELinuxChangePolicy` and `SELinuxMount` feature gates as beta to widen that performance improvement to other kinds of PersistentVolumeClaims, as explained in detail below. While in beta, `SELinuxMount` is still disabled by default.

With `SELinuxMount` feature gate disabled (the default in Kubernetes 1.33 and any previous release), the container runtime recursively assigns SELinux label to all files on all Pod volumes by default. To speed up this process, Kubernetes can change the SELinux label of a volume instantly by using a mount option `-o context=<label>`.

To benefit from this speedup, all these conditions must be met:

- The feature gate `SELinuxMountReadWriteOncePod` must be enabled.
- Pod must use PersistentVolumeClaim with applicable `accessModes` and feature gates:
  - Either the volume has `accessModes: ["ReadWriteOncePod"]`, and feature gate `SELinuxMountReadWriteOncePod` is enabled.
  - Or the volume can use any other access modes and all feature gates `SELinuxMountReadWriteOncePod`, `SELinuxChangePolicy` and `SELinuxMount` must be enabled and the Pod has `spec.securityContext.seLinuxChangePolicy` either nil (default) or `MountOption`.
- Pod (or all its Containers that use the PersistentVolumeClaim) must have `seLinuxOptions` set.
- The corresponding PersistentVolume must be either:
  - A volume that uses the legacy in-tree `iscsi`, `rbd` or `fc` volume type.
  - Or a volume that uses a CSI driver. The CSI driver must announce that it supports mounting with `-o context` by setting `spec.seLinuxMount: true` in its CSIDriver instance.

When any of these conditions is not met, SELinux relabelling happens another way: the container runtime recursively changes the SELinux label for all inodes (files and directories) in the volume. Calling out explicitly, this applies to Kubernetes ephemeral volumes like `secret`, `configMap` and `projected`, and all volumes whose CSIDriver instance does not explicitly announce mounting with `-o context`.

When this speedup is used, all Pods that use the same applicable volume concurrently on the same node **must have the same SELinux label**. A Pod with a different SELinux label will fail to start and will be `ContainerCreating` until all Pods with other SELinux labels that use the volume are deleted.

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: true)
For Pods that want to opt-out from relabeling using mount options, they can set `spec.securityContext.seLinuxChangePolicy` to `Recursive`. This is required when multiple pods share a single volume on the same node, but they run with different SELinux labels that allows simultaneous access to the volume. For example, a privileged pod running with label `spc_t` and an unprivileged pod running with the default label `container_file_t`. With unset `spec.securityContext.seLinuxChangePolicy` (or with the default value `MountOption`), only one of such pods is able to run on a node, the other one gets ContainerCreating with error `conflicting SELinux labels of volume <name of the volume>: <label of the running pod> and <label of the pod that can't start>`.

SELinuxWarningController

To make it easier to identify Pods that are affected by the change in SELinux volume relabeling, a new controller called `SELinuxWarningController` has been introduced in kube-controller-manager. It is disabled by default and can be enabled by either setting the `--controllers=*,selinux-warning-controller` command line flag, or by setting `genericControllerManagerConfiguration.controllers` field in KubeControllerManagerConfiguration. This controller requires `SELinuxChangePolicy` feature gate to be enabled.

When enabled, the controller observes running Pods and when it detects that two Pods use the same volume with different SELinux labels:

1. It emits an event to both of the Pods. `kubectl describe pod <pod-name>` the shows `SELinuxLabel "<label on the pod>" conflicts with pod <the other pod name> that uses the same volume as this pod with SELinuxLabel "<the other pod label>". If both pods land on the same node, only one of them may access the volume.`
2. Raise `selinux_warning_controller_selinux_volume_conflict` metric. The metric has both pod names + namespaces as labels to identify the affected pods easily.

A cluster admin can use this information to identify pods affected by the planning change and proactively opt-out Pods from the optimization (i.e. set `spec.securityContext.seLinuxChangePolicy: Recursive`).

Warning:

We strongly recommend clusters that use SELinux to enable this controller and make sure that `selinux_warning_controller_selinux_volume_conflict` metric does not report any conflicts before enabling `SELinuxMount` feature gate or upgrading to a version where `SELinuxMount` is enabled by default.

Feature gates

The following feature gates control the behavior of SELinux volume relabeling:

- `SELinuxMountReadWriteOncePod`: enables the optimization for volumes with `accessModes: ["ReadWriteOncePod"]`. This is a very safe feature gate to enable, as it cannot happen that two pods can share one single volume with this access mode. This feature gate is enabled by default sine v1.28.
- `SELinuxChangePolicy`: enables `spec.securityContext.seLinuxChangePolicy` field in Pod and related SELinuxWarningController in kube-controller-manager. This feature can be used before enabling `SELinuxMount` to check Pods running on a cluster, and to pro-actively opt-out Pods from the optimization. This feature gate requires `SELinuxMountReadWriteOncePod` enabled. It is beta and enabled by default in 1.33.
- `SELinuxMount` enables the optimization for all eligible volumes. Since it can break existing workloads, we recommend enabling `SELinuxChangePolicy` feature gate + SELinuxWarningController first to check the impact of the change. This feature gate requires `SELinuxMountReadWriteOncePod` and `SELinuxChangePolicy` enabled. It is beta, but disabled by default in 1.33.

Managing access to the `/proc` filesystem

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

For runtimes that follow the OCI runtime specification, containers default to running in a mode where there are multiple paths that are both masked and read-only. The result of this is the container has these paths present inside the container's mount namespace, and they can function similarly to if the container was an isolated host, but the container process cannot write to them. The list of masked and read-only paths are as follows:

- Masked Paths:
    - `/proc/asound`
    - `/proc/acpi`
    - `/proc/kcore`
    - `/proc/keys`
    - `/proc/latency_stats`
    - `/proc/timer_list`
    - `/proc/timer_stats`
    - `/proc/sched_debug`
    - `/proc/scsi`
    - `/sys/firmware`
    - `/sys/devices/virtual/powercap`
- Read-Only Paths:
    - `/proc/bus`
    - `/proc/fs`
    - `/proc/irq`
    - `/proc/sys`
    - `/proc/sysrq-trigger`

For some Pods, you might want to bypass that default masking of paths. The most common context for wanting this is if you are trying to run containers within a Kubernetes container (within a pod).

The `securityContext` field `procMount` allows a user to request a container's `/proc` be `Unmasked`, or be mounted as read-write by the container process. This also applies to `/sys/firmware` which is not in `/proc`.

```
...
securityContext:  procMount: Unmasked
```

Note:

Setting `procMount` to Unmasked requires the `spec.hostUsers` value in the pod spec to be `false`. In other words: a container that wishes to have an Unmasked `/proc` or unmasked `/sys` must also be in a user namespace. Kubernetes v1.12 to v1.29 did not enforce that requirement.

Discussion

The security context for a Pod applies to the Pod's Containers and also to the Pod's Volumes when applicable. Specifically `fsGroup` and `seLinuxOptions` are applied to Volumes as follows:

- `fsGroup`: Volumes that support ownership management are modified to be owned and writable by the GID specified in `fsGroup`. See the Ownership Management design document for more details.

- `seLinuxOptions`: Volumes that support SELinux labeling are relabeled to be accessible by the label specified under `seLinuxOptions`. Usually you only need to set the `level` section. This sets the Multi-Category Security (MCS) label given to all Containers in the Pod as well as the Volumes.

Warning:

After you specify an MCS label for a Pod, all Pods with the same label can access the Volume. If you need inter-Pod protection, you must assign a unique MCS label to each Pod.

Clean up

Delete the Pod:

```
kubectl delete pod security-context-demo
kubectl delete pod security-context-demo-2
kubectl delete pod security-context-demo-3
kubectl delete pod security-context-demo-4
```

What's next

- PodSecurityContext
- SecurityContext
- CRI Plugin Config Guide
- Security Contexts design document
- Ownership Management design document
- PodSecurity Admission
- AllowPrivilegeEscalation design document
- For more information about security mechanisms in Linux, see Overview of Linux Kernel Security Features (Note: Some information is out of date)
- Read about User Namespaces for Linux pods.
- Masked Paths in the OCI Runtime Specification

Enforce Pod Security Standards by Configuring the Built-in Admission Controller

Kubernetes provides a built-in admission controller to enforce the Pod Security Standards. You can configure this admission controller to set cluster-wide defaults and exemptions.

Before you begin

Following an alpha release in Kubernetes v1.22, Pod Security Admission became available by default in Kubernetes v1.23, as a beta. From version 1.25 onwards, Pod Security Admission is generally available.

To check the version, enter `kubectl version`.

If you are not running Kubernetes 1.34, you can switch to viewing this page in the documentation for the Kubernetes version that you are running.

Configure the Admission Controller

Note:

`pod-security.admission.config.k8s.io/v1` configuration requires v1.25+. For v1.23 and v1.24, use v1beta1. For v1.22, use v1alpha1.

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:- name: PodSecurity  configuration:    apiVersion: pod-security.admission.config.k8s.io/v1 # s
```

Note:

The above manifest needs to be specified via the `--admission-control-config-file` to kube-apiserver.

Translate a Docker Compose File to Kubernetes Resources

What's Kompose? It's a conversion tool for all things compose (namely Docker Compose) to container orchestrators (Kubernetes or OpenShift).

More information can be found on the Kompose website at https://kompose.io/.

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`.

Install Kompose

We have multiple ways to install Kompose. Our preferred method is downloading the binary from the latest GitHub release.

- GitHub download
- Build from source
- Homebrew (macOS)

Kompose is released via GitHub on a three-week cycle, you can see all current releases on the GitHub release page.

```
# Linux
curl -L https://github.com/kubernetes/kompose/releases/download/v1.34.0/kompose-linux-amd64 -o kompose

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/v1.34.0/kompose-darwin-amd64 -o kompose

# Windows
curl -L https://github.com/kubernetes/kompose/releases/download/v1.34.0/kompose-windows-amd64.exe -o kompose.exe

chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

Alternatively, you can download the tarball.

Installing using `go get` pulls from the master branch with the latest development changes.

```
go get -u github.com/kubernetes/kompose
```

On macOS you can install the latest release via Homebrew:

```
brew install kompose
```

Use Kompose

In a few steps, we'll take you from Docker Compose to Kubernetes. All you need is an existing `docker-compose.yml` file.

1. Go to the directory containing your `docker-compose.yml` file. If you don't have one, test using this one.

   ```
   services:   redis-leader:    container_name: redis-leader    image: redis    ports:      - "6379"  redis-replica:    container_
   ```

2. To convert the `docker-compose.yml` file to files that you can use with `kubectl`, run `kompose convert` and then `kubectl apply -f <output file>`.

   ```
   kompose convert
   ```

   The output is similar to:

   ```
   INFO Kubernetes file "redis-leader-service.yaml" created
   INFO Kubernetes file "redis-replica-service.yaml" created
   INFO Kubernetes file "web-tcp-service.yaml" created
   INFO Kubernetes file "redis-leader-deployment.yaml" created
   INFO Kubernetes file "redis-replica-deployment.yaml" created
   INFO Kubernetes file "web-deployment.yaml" created
   ```

   ```
    kubectl apply -f web-tcp-service.yaml,redis-leader-service.yaml,redis-replica-service.yaml,web-deployment.yaml,redis-leader-
   ```

   The output is similar to:

   ```
   deployment.apps/redis-leader created
   deployment.apps/redis-replica created
   deployment.apps/web created
   service/redis-leader created
   service/redis-replica created
   service/web-tcp created
   ```

   Your deployments are running in Kubernetes.

3. Access your application.

   If you're already using `minikube` for your development process:

   ```
   minikube service web-tcp
   ```

   Otherwise, let's look up what IP your service is using!

   ```
   kubectl describe svc web-tcp
   ```

   ```
    Name:                     web-tcp
    Namespace:                default
    Labels:                   io.kompose.service=web-tcp
    Annotations:              kompose.cmd: kompose convert
                              kompose.service.type: LoadBalancer
                              kompose.version: 1.33.0 (3ce457399)
    Selector:                 io.kompose.service=web
    Type:                     LoadBalancer
    IP Family Policy:         SingleStack
    IP Families:              IPv4
    IP:                       10.102.30.3
    IPs:                      10.102.30.3
    Port:                     8080  8080/TCP
    TargetPort:               8080/TCP
    NodePort:                 8080  31624/TCP
    Endpoints:                10.244.0.5:8080
    Session Affinity:         None
    External Traffic Policy:  Cluster
    Events:                   <none>
   ```

   If you're using a cloud provider, your IP will be listed next to `LoadBalancer Ingress`.

   ```
   curl http://192.0.2.89
   ```

4. Clean-up.

   After you are finished testing out the example application deployment, simply run the following command in your shell to delete the resources used.

   ```
   kubectl delete -f web-tcp-service.yaml,redis-leader-service.yaml,redis-replica-service.yaml,web-deployment.yaml,redis-leader-
   ```

User Guide

- CLI
  - [kompose convert]
- Documentation
  - Alternative Conversions
  - Labels
  - Restart
  - Docker Compose Versions

Kompose has support for two providers: OpenShift and Kubernetes. You can choose a targeted provider using global option `--provider`. If no provider is specified, Kubernetes is set by default.

`kompose convert`

Kompose supports conversion of V1, V2, and V3 Docker Compose files into Kubernetes and OpenShift objects.

Kubernetes `kompose convert` example

```
kompose --file docker-voting.yml convert
```

```
WARN Unsupported key networks - ignoring
WARN Unsupported key build - ignoring
INFO Kubernetes file "worker-svc.yaml" created
INFO Kubernetes file "db-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "result-svc.yaml" created
INFO Kubernetes file "vote-svc.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
INFO Kubernetes file "result-deployment.yaml" created
INFO Kubernetes file "vote-deployment.yaml" created
INFO Kubernetes file "worker-deployment.yaml" created
INFO Kubernetes file "db-deployment.yaml" created
```

```
ls
```

```
db-deployment.yaml  docker-compose.yml         docker-gitlab.yml  redis-deployment.yaml  result-deployment.yaml  vote-deployment.ya
db-svc.yaml         docker-voting.yml          redis-svc.yaml     result-svc.yaml        vote-svc.yaml           worker-svc.yaml
```

You can also provide multiple docker-compose files at the same time:

```
kompose -f docker-compose.yml -f docker-guestbook.yml convert
```

```
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "mlbparks-service.yaml" created
INFO Kubernetes file "mongodb-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "mlbparks-deployment.yaml" created
INFO Kubernetes file "mongodb-deployment.yaml" created
INFO Kubernetes file "mongodb-claim0-persistentvolumeclaim.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

```
ls
```

```
mlbparks-deployment.yaml  mongodb-service.yaml                       redis-slave-service.jsonmlbparks-service.yaml
frontend-deployment.yaml  mongodb-claim0-persistentvolumeclaim.yaml  redis-master-service.yaml
frontend-service.yaml     mongodb-deployment.yaml                    redis-slave-deployment.yaml
redis-master-deployment.yaml
```

When multiple docker-compose files are provided the configuration is merged. Any configuration that is common will be overridden by subsequent file.

OpenShift `kompose convert` example

```
kompose --provider openshift --file docker-voting.yml convert
```

```
WARN [worker] Service cannot be created because of missing port.
INFO OpenShift file "vote-service.yaml" created
INFO OpenShift file "db-service.yaml" created
INFO OpenShift file "redis-service.yaml" created
INFO OpenShift file "result-service.yaml" created
INFO OpenShift file "vote-deploymentconfig.yaml" created
INFO OpenShift file "vote-imagestream.yaml" created
INFO OpenShift file "worker-deploymentconfig.yaml" created
INFO OpenShift file "worker-imagestream.yaml" created
INFO OpenShift file "db-deploymentconfig.yaml" created
INFO OpenShift file "db-imagestream.yaml" created
INFO OpenShift file "redis-deploymentconfig.yaml" created
INFO OpenShift file "redis-imagestream.yaml" created
INFO OpenShift file "result-deploymentconfig.yaml" created
INFO OpenShift file "result-imagestream.yaml" created
```

It also supports creating buildconfig for build directive in a service. By default, it uses the remote repo for the current git branch as the source repo, and the current branch as the source branch for the build. You can specify a different source repo and branch using `--build-repo` and `--build-branch` options respectively.

```
kompose --provider openshift --file buildconfig/docker-compose.yml convert
```

```
WARN [foo] Service cannot be created because of missing port.
INFO OpenShift Buildconfig using git@github.com:rtnpro/kompose.git::master as source.
INFO OpenShift file "foo-deploymentconfig.yaml" created
INFO OpenShift file "foo-imagestream.yaml" created
INFO OpenShift file "foo-buildconfig.yaml" created
```

Note:

If you are manually pushing the OpenShift artifacts using `oc create -f`, you need to ensure that you push the imagestream artifact before the buildconfig artifact, to workaround this OpenShift issue: https://github.com/openshift/origin/issues/4518 .

Alternative Conversions

The default `kompose` transformation will generate Kubernetes Deployments and Services, in yaml format. You have alternative option to generate json with `-j`. Also, you can alternatively generate Replication Controllers objects, Daemon Sets, or Helm charts.

```
kompose convert -j
INFO Kubernetes file "redis-svc.json" created
INFO Kubernetes file "web-svc.json" created
INFO Kubernetes file "redis-deployment.json" created
INFO Kubernetes file "web-deployment.json" created
```

The `*-deployment.json` files contain the Deployment objects.

```
kompose convert --replication-controller
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-replicationcontroller.yaml" created
INFO Kubernetes file "web-replicationcontroller.yaml" created
```

The `*-replicationcontroller.yaml` files contain the Replication Controller objects. If you want to specify replicas (default is 1), use `--replicas` flag: `kompose convert --replication-controller --replicas 3`.

```
kompose convert --daemon-set
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-daemonset.yaml" created
INFO Kubernetes file "web-daemonset.yaml" created
```

The `*-daemonset.yaml` files contain the DaemonSet objects.

If you want to generate a Chart to be used with Helm run:

```
kompose convert -c

INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-deployment.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
chart created in "./docker-compose/"

tree docker-compose/

docker-compose
├── Chart.yaml
├── README.md
└── templates
    ├── redis-deployment.yaml
    ├── redis-svc.yaml
    ├── web-deployment.yaml
    └── web-svc.yaml
```

The chart structure is aimed at providing a skeleton for building your Helm charts.

Labels

`kompose` supports Kompose-specific labels within the `docker-compose.yml` file in order to explicitly define a service's behavior upon conversion.

- `kompose.service.type` defines the type of service to be created.

  For example:

  ```
  version: "2"
  services:   nginx:     image: nginx     dockerfile: foobar     build: ./foobar     cap_add:       - ALL     container_name: foobar
  ```

- `kompose.service.expose` defines if the service needs to be made accessible from outside the cluster or not. If the value is set to "true", the provider sets the endpoint automatically, and for any other value, the value is set as the hostname. If multiple ports are defined in a service, the first one is chosen to be the exposed.

  - For the Kubernetes provider, an ingress resource is created and it is assumed that an ingress controller has already been configured.
  - For the OpenShift provider, a route is created.

  For example:

  ```
  version: "2"
  services:   web:     image: tuna/docker-counter23     ports:     - "5000:5000"     links:     - redis     labels:       kompose.servi
  ```

The currently supported options are:

Key	Value
kompose.service.type	nodeport / clusterip / loadbalancer
kompose.service.expose	true / hostname

Note:

The `kompose.service.type` label should be defined with `ports` only, otherwise `kompose` will fail.

Restart

If you want to create normal pods without controllers you can use `restart` construct of docker-compose to define that. Follow table below to see what happens on the `restart` value.

`docker-compose restart`	object created	Pod `restartPolicy`
`""`	controller object	`Always`
`always`	controller object	`Always`
`on-failure`	Pod	`OnFailure`
`no`	Pod	`Never`

Note:

The controller object could be `deployment` or `replicationcontroller`.

For example, the `pival` service will become pod down here. This container calculated value of `pi`.

```
version: '2'
```

```
services:   pival:     image: perl     command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]     restart: "on-failure"
```

Warning about Deployment Configurations

If the Docker Compose file has a volume specified for a service, the Deployment (Kubernetes) or DeploymentConfig (OpenShift) strategy is changed to "Recreate" instead of "RollingUpdate" (default). This is done to avoid multiple instances of a service from accessing a volume at the same time.

If the Docker Compose file has service name with _ in it (for example, `web_service`), then it will be replaced by `-` and the service name will be renamed accordingly (for example, `web-service`). Kompose does this because "Kubernetes" doesn't allow _ in object name.

Please note that changing service name might break some `docker-compose` files.

Docker Compose Versions

Kompose supports Docker Compose versions: 1, 2 and 3. We have limited support on versions 2.1 and 3.2 due to their experimental nature.

A full list on compatibility between all three versions is listed in our conversion document including a list of all incompatible Docker Compose keys.

Use an Image Volume With a Pod

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: false)

This page shows how to configure a pod using image volumes. This allows you to mount content from OCI registries inside containers.

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 v1.31.

To check the version, enter `kubectl version`.

- The container runtime needs to support the image volumes feature
- You need to exec commands in the host
- You need to be able to exec into pods
- You need to enable the `ImageVolume` feature gate

Run a Pod that uses an image volume

An image volume for a pod is enabled by setting the `volumes.[*].image` field of `.spec` to a valid reference and consuming it in the `volumeMounts` of the container. For example:

`pods/image-volumes.yaml` Copy pods/image-volumes.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: image-volumespec:  containers:  - name: shell    command: ["sleep", "infinity"]    image: debian    volu
```

1. Create the pod on your cluster:

   ```
   kubectl apply -f https://k8s.io/examples/pods/image-volumes.yaml
   ```

2. Attach to the container:

   ```
   kubectl attach -it image-volume bash
   ```

3. Check the content of a file in the volume:

   ```
   cat /volume/dir/file
   ```

   The output is similar to:

   ```
   1
   ```

   You can also check another file in a different path:

   ```
   cat /volume/file
   ```

   The output is similar to:

   ```
   2
   ```

Use `subPath` (or `subPathExpr`)

It is possible to utilize `subPath` or `subPathExpr` from Kubernetes v1.33 when using the image volume feature.

`pods/image-volumes-subpath.yaml` Copy pods/image-volumes-subpath.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: image-volumespec:  containers:  - name: shell    command: ["sleep", "infinity"]    image: debian    volu
```

1. Create the pod on your cluster:

   ```
   kubectl apply -f https://k8s.io/examples/pods/image-volumes-subpath.yaml
   ```

2. Attach to the container:

   ```
   kubectl attach -it image-volume bash
   ```

3. Check the content of the file from the `dir` sub path in the volume:

   ```
   cat /volume/file
   ```

   The output is similar to:

   ```
   1
   ```

Further reading

- image volumes

Configure a Pod to Use a Projected Volume for Storage

This page shows how to use a `projected` Volume to mount several existing volume sources into the same directory. Currently, `secret`, `configMap`, `downwardAPI`, and `serviceAccountToken` volumes can be projected.

Note:

`serviceAccountToken` is not a volume type.

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`.

Configure a projected volume for a pod

In this exercise, you create username and password Secrets from local files. You then create a Pod that runs one container, using a `projected` Volume to mount the Secrets into the same shared directory.

Here is the configuration file for the Pod:

`pods/storage/projected.yaml` Copy pods/storage/projected.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: test-projected-volumespec:  containers:  - name: test-projected-volume    image: busybox:1.28    args:
```

1. Create the Secrets:

```
# Create files containing the username and password:
echo -n "admin" > ./username.txt
echo -n "1f2d1e2e67df" > ./password.txt

# Package these files into secrets:
kubectl create secret generic user --from-file=./username.txt
kubectl create secret generic pass --from-file=./password.txt
```

2. Create the Pod:

   ```
   kubectl apply -f https://k8s.io/examples/pods/storage/projected.yaml
   ```

3. Verify that the Pod's container is running, and then watch for changes to the Pod:

   ```
   kubectl get --watch pod test-projected-volume
   ```

   The output looks like this:

   ```
   NAME                    READY     STATUS    RESTARTS   AGE
   test-projected-volume   1/1       Running   0          14s
   ```

4. In another terminal, get a shell to the running container:

   ```
   kubectl exec -it test-projected-volume -- /bin/sh
   ```

5. In your shell, verify that the `projected-volume` directory contains your projected sources:

   ```
   ls /projected-volume/
   ```

Clean up

Delete the Pod and the Secrets:

```
kubectl delete pod test-projected-volume
kubectl delete secret user pass
```

What's next

- Learn more about `projected` volumes.
- Read the all-in-one volume design document.

Assign Pods to Nodes

This page shows how to assign a Kubernetes Pod to a particular node in a Kubernetes 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`.

Add a label to a node

1. List the nodes in your cluster, along with their labels:

   ```
   kubectl get nodes --show-labels
   ```

   The output is similar to this:

   ```
   NAME      STATUS    ROLES    AGE     VERSION        LABELS
   worker0   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker0
   worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker1
   worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker2
   ```

2. Choose one of your nodes, and add a label to it:

   ```
   kubectl label nodes <your-node-name> disktype=ssd
   ```

   where `<your-node-name>` is the name of your chosen node.

3. Verify that your chosen node has a `disktype=ssd` label:

   ```
   kubectl get nodes --show-labels
   ```

   The output is similar to this:

```
NAME      STATUS    ROLES    AGE     VERSION        LABELS
worker0   Ready     <none>   1d      v1.13.0        ...,disktype=ssd,kubernetes.io/hostname=worker0
worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker1
worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=worker2
```

In the preceding output, you can see that the `worker0` node has a `disktype=ssd` label.

Create a pod that gets scheduled to your chosen node

This pod configuration file describes a pod that has a node selector, `disktype: ssd`. This means that the pod will get scheduled on a node that has a `disktype=ssd` label.

pods/pod-nginx.yaml Copy pods/pod-nginx.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginx  labels:    env: testspec:  containers:  - name: nginx    image: nginx    imagePullPolicy: IfNotPr
```

1. Use the configuration file to create a pod that will get scheduled on your chosen node:

   ```
   kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml
   ```

2. Verify that the pod is running on your chosen node:

   ```
   kubectl get pods --output=wide
   ```

   The output is similar to this:

   ```
   NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
   nginx    1/1       Running   0          13s    10.200.0.4   worker0
   ```

Create a pod that gets scheduled to specific node

You can also schedule a pod to one specific node via setting `nodeName`.

pods/pod-nginx-specific-node.yaml Copy pods/pod-nginx-specific-node.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginxspec:  nodeName: foo-node # schedule pod to specific node  containers:  - name: nginx    image: ngi
```

Use the configuration file to create a pod that will get scheduled on `foo-node` only.

What's next

- Learn more about labels and selectors.
- Learn more about nodes.

Allocate Devices to Workloads with DRA

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

This page shows you how to allocate devices to your Pods by using *dynamic resource allocation (DRA)*. These instructions are for workload operators. Before reading this page, familiarize yourself with how DRA works and with DRA terminology like ResourceClaims and ResourceClaimTemplates. For more information, see Dynamic Resource Allocation (DRA).

About device allocation with DRA

As a workload operator, you can *claim* devices for your workloads by creating ResourceClaims or ResourceClaimTemplates. When you deploy your workload, Kubernetes and the device drivers find available devices, allocate them to your Pods, and place the Pods on nodes that can access those devices.

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 version v1.34.

To check the version, enter `kubectl version`.

- Ensure that your cluster admin has set up DRA, attached devices, and installed drivers. For more information, see Set Up DRA in a Cluster.

Identify devices to claim

Your cluster administrator or the device drivers create *DeviceClasses* that define categories of devices. You can claim devices by using Common Expression Language to filter for specific device properties.

Get a list of DeviceClasses in the cluster:

```
kubectl get deviceclasses
```

The output is similar to the following:

```
NAME                  AGE
driver.example.com    16m
```

If you get a permission error, you might not have access to get DeviceClasses. Check with your cluster administrator or with the driver provider for available device properties.

Claim resources

You can request resources from a DeviceClass by using ResourceClaims. To create a ResourceClaim, do one of the following:

- Manually create a ResourceClaim if you want multiple Pods to share access to the same devices, or if you want a claim to exist beyond the lifetime of a Pod.
- Use a ResourceClaimTemplate to let Kubernetes generate and manage per-Pod ResourceClaims. Create a ResourceClaimTemplate if you want every Pod to have access to separate devices that have similar configurations. For example, you might want simultaneous access to devices for Pods in a Job that uses parallel execution.

If you directly reference a specific ResourceClaim in a Pod, that ResourceClaim must already exist in the cluster. If a referenced ResourceClaim doesn't exist, the Pod remains in a pending state until the ResourceClaim is created. You can reference an auto-generated ResourceClaim in a Pod, but this isn't recommended because auto-generated ResourceClaims are bound to the lifetime of the Pod that triggered the generation.

To create a workload that claims resources, select one of the following options:

- ResourceClaimTemplate
- ResourceClaim

Review the following example manifest:

`dra/resourceclaimtemplate.yaml` Copy dra/resourceclaimtemplate.yaml to clipboard

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimTemplatemetadata:  name: example-resource-claim-templatespec:  spec:    devices:      requests:      - name: gp
```

This manifest creates a ResourceClaimTemplate that requests devices in the `example-device-class` DeviceClass that match both of the following parameters:

- Devices that have a `driver.example.com/type` attribute with a value of `gpu`.
- Devices that have `64Gi` of capacity.

To create the ResourceClaimTemplate, run the following command:

```
kubectl apply -f https://k8s.io/examples/dra/resourceclaimtemplate.yaml
```

Review the following example manifest:

`dra/resourceclaim.yaml` Copy dra/resourceclaim.yaml to clipboard

```
apiVersion: resource.k8s.io/v1
kind: ResourceClaimmetadata:  name: example-resource-claimspec:  devices:    requests:    - name: single-gpu-claim      exactly:
```

This manifest creates ResourceClaim that requests devices in the `example-device-class` DeviceClass that match both of the following parameters:

- Devices that have a `driver.example.com/type` attribute with a value of `gpu`.
- Devices that have `64Gi` of capacity.

To create the ResourceClaim, run the following command:

```
kubectl apply -f https://k8s.io/examples/dra/resourceclaim.yaml
```

Request devices in workloads using DRA

To request device allocation, specify a ResourceClaim or a ResourceClaimTemplate in the `resourceClaims` field of the Pod specification. Then, request a specific claim by name in the `resources.claims` field of a container in that Pod. You can specify multiple entries in the `resourceClaims` field and use specific claims in different containers.

1. Review the following example Job:

   `dra/dra-example-job.yaml` Copy dra/dra-example-job.yaml to clipboard

   ```
   apiVersion: batch/v1
   kind: Jobmetadata:  name: example-dra-jobspec:  completions: 10  parallelism: 2  template:    spec:      restartPolicy: Never
   ```

   Each Pod in this Job has the following properties:

   - Makes a ResourceClaimTemplate named `separate-gpu-claim` and a ResourceClaim named `shared-gpu-claim` available to containers.
   - Runs the following containers:
     - `container0` requests the devices from the `separate-gpu-claim` ResourceClaimTemplate.

        - `container1` and `container2` share access to the devices from the `shared-gpu-claim` ResourceClaim.

2. Create the Job:

   ```
   kubectl apply -f https://k8s.io/examples/dra/dra-example-job.yaml
   ```

Try the following troubleshooting steps:

1. When the workload does not start as expected, drill down from Job to Pods to ResourceClaims and check the objects at each level with `kubectl describe` to see whether there are any status fields or events which might explain why the workload is not starting.
2. When creating a Pod fails with `must specify one of: resourceClaimName, resourceClaimTemplateName`, check that all entries in `pod.spec.resourceClaims` have exactly one of those fields set. If they do, then it is possible that the cluster has a mutating Pod webhook installed which was built against APIs from Kubernetes < 1.32. Work with your cluster administrator to check this.

Clean up

To delete the Kubernetes objects that you created in this task, follow these steps:

1. Delete the example Job:

   ```
   kubectl delete -f https://k8s.io/examples/dra/dra-example-job.yaml
   ```

2. To delete your resource claims, run one of the following commands:

   - Delete the ResourceClaimTemplate:

     ```
     kubectl delete -f https://k8s.io/examples/dra/resourceclaimtemplate.yaml
     ```

   - Delete the ResourceClaim:

     ```
     kubectl delete -f https://k8s.io/examples/dra/resourceclaim.yaml
     ```

What's next

- Learn more about DRA

Pull an Image from a Private Registry

This page shows how to create a Pod that uses a Secret to pull an image from a private container image registry or repository. There are many private registries in use. This task uses Docker Hub as an example registry.

□ This item links to a third party project or product that is not part of Kubernetes itself. More information

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 do this exercise, you need the `docker` command line tool, and a Docker ID for which you know the password.

- If you are using a different private container registry, you need the command line tool for that registry and any login information for the registry.

Log in to Docker Hub

On your laptop, you must authenticate with a registry in order to pull a private image.

Use the `docker` tool to log in to Docker Hub. See the *log in* section of Docker ID accounts for more information.

```
docker login
```

When prompted, enter your Docker ID, and then the credential you want to use (access token, or the password for your Docker ID).

The login process creates or updates a `config.json` file that holds an authorization token. Review how Kubernetes interprets this file.

View the `config.json` file:

```
cat ~/.docker/config.json
```

The output contains a section similar to this:

```
{
    "auths": {
        "https://index.docker.io/v1/": {
            "auth": "c3R...zE2"
```

```
            }
        }
}
```

Note:

If you use a Docker credentials store, you won't see that `auth` entry but a `credsStore` entry with the name of the store as value. In that case, you can create a secret directly. See [Create a Secret by providing credentials on the command line](#).

Create a Secret based on existing credentials

A Kubernetes cluster uses the Secret of `kubernetes.io/dockerconfigjson` type to authenticate with a container registry to pull a private image.

If you already ran `docker login`, you can copy that credential into Kubernetes:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \    --type=kubernetes.io/dockerconfigjson
```

If you need more control (for example, to set a namespace or a label on the new secret) then you can customise the Secret before storing it. Be sure to:

- set the name of the data item to `.dockerconfigjson`
- base64 encode the Docker configuration file and then paste that string, unbroken as the value for field `data[".dockerconfigjson"]`
- set `type` to `kubernetes.io/dockerconfigjson`

Example:

```
apiVersion: v1
kind: Secretmetadata:  name: myregistrykey  namespace: awesomeappsdata:  .dockerconfigjson: UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFhYWI
```

If you get the error message `error: no objects passed to create`, it may mean the base64 encoded string is invalid. If you get an error message like `Secret "myregistrykey" is invalid: data[.dockerconfigjson]: invalid value ...`, it means the base64 encoded string in the data was successfully decoded, but could not be parsed as a `.docker/config.json` file.

Create a Secret by providing credentials on the command line

Create this Secret, naming it `regcred`:

```
kubectl create secret docker-registry regcred --docker-server=<your-registry-server> --docker-username=<your-name> --docker-passwo
```

where:

- `<your-registry-server>` is your Private Docker Registry FQDN. Use `https://index.docker.io/v1/` for DockerHub.
- `<your-name>` is your Docker username.
- `<your-pword>` is your Docker password.
- `<your-email>` is your Docker email.

You have successfully set your Docker credentials in the cluster as a Secret called `regcred`.

Note:

Typing secrets on the command line may store them in your shell history unprotected, and those secrets might also be visible to other users on your PC during the time that `kubectl` is running.

Inspecting the Secret `regcred`

To understand the contents of the `regcred` Secret you created, start by viewing the Secret in YAML format:

```
kubectl get secret regcred --output=yaml
```

The output is similar to this:

```
apiVersion: v1
kind: Secretmetadata:  ...  name: regcred  ...data:  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=type: kubernetes.i
```

The value of the `.dockerconfigjson` field is a base64 representation of your Docker credentials.

To understand what is in the `.dockerconfigjson` field, convert the secret data to a readable format:

```
kubectl get secret regcred --output="jsonpath={.data.\.dockerconfigjson}" | base64 --decode
```

The output is similar to this:

```
{"auths":{"your.private.registry.example.com":{"username":"janedoe","password":"xxxxxxxxxxx","email":"jdoe@example.com","auth":"c3I
```

To understand what is in the `auth` field, convert the base64-encoded data to a readable format:

```
echo "c3R...zE2" | base64 --decode
```

The output, username and password concatenated with a `:`, is similar to this:

```
janedoe:xxxxxxxxxxx
```

Notice that the Secret data contains the authorization token similar to your local `~/.docker/config.json` file.

You have successfully set your Docker credentials as a Secret called `regcred` in the cluster.

Create a Pod that uses your Secret

Here is a manifest for an example Pod that needs access to your Docker credentials in `regcred`:

[`pods/private-reg-pod.yaml`]Copy pods/private-reg-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: private-regspec:  containers:  - name: private-reg-container    image: <your-private-image>  imagePullSe
```

Download the above file onto your computer:

```
curl -L -o my-private-reg-pod.yaml https://k8s.io/examples/pods/private-reg-pod.yaml
```

In file `my-private-reg-pod.yaml`, replace `<your-private-image>` with the path to an image in a private registry such as:

```
your.private.registry.example.com/janedoe/jdoe-private:v1
```

To pull the image from the private registry, Kubernetes needs credentials. The `imagePullSecrets` field in the configuration file specifies that Kubernetes should get the credentials from a Secret named `regcred`.

Create a Pod that uses your Secret, and verify that the Pod is running:

```
kubectl apply -f my-private-reg-pod.yaml
kubectl get pod private-reg
```

Note:

To use image pull secrets for a Pod (or a Deployment, or other object that has a pod template that you are using), you need to make sure that the appropriate Secret does exist in the right namespace. The namespace to use is the same namespace where you defined the Pod.

Also, in case the Pod fails to start with the status `ImagePullBackOff`, view the Pod events:

```
kubectl describe pod private-reg
```

If you then see an event with the reason set to `FailedToRetrieveImagePullSecret`, Kubernetes can't find a Secret with name (`regcred`, in this example).

Make sure that the Secret you have specified exists, and that its name is spelled properly.

```
Events:
  ...  Reason                           ...  Message
       ------                                -------
  ...  FailedToRetrieveImagePullSecret  ...  Unable to retrieve some image pull secrets (<regcred>); attempting to pull the image
```

Using images from multiple registries

A pod can have multiple containers, each container image can be from a different registry. You can use multiple `imagePullSecrets` with one pod, and each can contain multiple credentials.

The image pull will be attempted using each credential that matches the registry. If no credentials match the registry, the image pull will be attempted without authorization or using custom runtime specific configuration.

What's next

- Learn more about Secrets
    - or read the API reference for Secret
- Learn more about using a private registry.
- Learn more about adding image pull secrets to a service account.
- See kubectl create secret docker-registry.
- See the `imagePullSecrets` field within the container definitions of a Pod

Create static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. Unlike Pods that are managed by the control plane (for example, a Deployment); instead, the kubelet watches each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node.

The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there. The Pod names will be suffixed with the node hostname with a leading hyphen.

Note:

If you are running clustered Kubernetes and are using static Pods to run a Pod on every node, you should probably be using a DaemonSet instead.

Note:

The `spec` of a static Pod cannot refer to other API objects (e.g., ServiceAccount, ConfigMap, Secret, etc).

Note:

Static pods do not support ephemeral containers.

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`.

This page assumes you're using CRI-O to run Pods, and that your nodes are running the Fedora operating system. Instructions for other distributions or Kubernetes installations may vary.

Create a static pod

You can configure a static Pod with either a file system hosted configuration file or a web hosted configuration file.

Filesystem-hosted static Pod manifest

Manifests are standard Pod definitions in JSON or YAML format in a specific directory. Use the `staticPodPath: <the directory>` field in the kubelet configuration file, which periodically scans the directory and creates/deletes static Pods as YAML/JSON files appear/disappear there. Note that the kubelet will ignore files starting with dots when scanning the specified directory.

For example, this is how to start a simple web server as a static Pod:

1. Choose a node where you want to run the static Pod. In this example, it's `my-node1`.

   ```
   ssh my-node1
   ```

2. Choose a directory, say `/etc/kubernetes/manifests` and place a web server Pod definition there, for example `/etc/kubernetes/manifests/static-web.yaml`:

   ```
   # Run this command on the node where kubelet is running
   mkdir -p /etc/kubernetes/manifests/
   cat <<EOF >/etc/kubernetes/manifests/static-web.yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: static-web
     labels:
       role: myrole
   spec:
     containers:
       - name: web
         image: nginx
         ports:
           - name: web
             containerPort: 80
             protocol: TCP
   EOF
   ```

3. Configure the kubelet on that node to set a `staticPodPath` value in the kubelet configuration file.
   See Set Kubelet Parameters Via A Configuration File for more information.

   An alternative and deprecated method is to configure the kubelet on that node to look for static Pod manifests locally, using a command line argument. To use the deprecated approach, start the kubelet with the
   `--pod-manifest-path=/etc/kubernetes/manifests/` argument.

4. Restart the kubelet. On Fedora, you would run:

   ```
   # Run this command on the node where the kubelet is running
   systemctl restart kubelet
   ```

Web-hosted static pod manifest

Kubelet periodically downloads a file specified by `--manifest-url=<URL>` argument and interprets it as a JSON/YAML file that contains Pod definitions. Similar to how filesystem-hosted manifests work, the kubelet refetches the manifest on a schedule. If there are changes to the list of static Pods, the kubelet applies them.

To use this approach:

1. Create a YAML file and store it on a web server so that you can pass the URL of that file to the kubelet.

   ```
   apiVersion: v1
   kind: Podmetadata:  name: static-web  labels:    role: myrolespec:  containers:    - name: web      image: nginx      ports:
   ```

2. Configure the kubelet on your selected node to use this web manifest by running it with `--manifest-url=<manifest-url>`. On Fedora, edit `/etc/kubernetes/kubelet` to include this line:

   ```
   KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.local --manifest-url=<manifest-url>"
   ```

3. Restart the kubelet. On Fedora, you would run:

   ```
   # Run this command on the node where the kubelet is running
   systemctl restart kubelet
   ```

Observe static pod behavior

When the kubelet starts, it automatically starts all defined static Pods. As you have defined a static Pod and restarted the kubelet, the new static Pod should already be running.

You can view running containers (including static Pods) by running (on the node):

```
# Run this command on the node where the kubelet is running
crictl ps
```

The output might be something like:

```
CONTAINER       IMAGE                                 CREATED           STATE      NAME    ATTEMPT    POD ID
129fd7d382018   docker.io/library/nginx@sha256:...    11 minutes ago    Running    web     0          34533c6729106
```

Note:

`crictl` outputs the image URI and SHA-256 checksum. `NAME` will look more like: `docker.io/library/nginx@sha256:0d17b565c37bcbd895e9d92315a05c1c3c9a29f762b011a10c54a66cd53c9b31`.

You can see the mirror Pod on the API server:

```
kubectl get pods
```

```
NAME                   READY    STATUS     RESTARTS         AGE
static-web-my-node1    1/1      Running    0                2m
```

Note:

Make sure the kubelet has permission to create the mirror Pod in the API server. If not, the creation request is rejected by the API server.

Labels from the static Pod are propagated into the mirror Pod. You can use those labels as normal via selectors, etc.

If you try to use `kubectl` to delete the mirror Pod from the API server, the kubelet *doesn't* remove the static Pod:

```
kubectl delete pod static-web-my-node1
```

```
pod "static-web-my-node1" deleted
```

You can see that the Pod is still running:

```
kubectl get pods
```

```
NAME                   READY    STATUS     RESTARTS    AGE
static-web-my-node1    1/1      Running    0           4s
```

Back on your node where the kubelet is running, you can try to stop the container manually. You'll see that, after a time, the kubelet will notice and will restart the Pod automatically:

```
# Run these commands on the node where the kubelet is running
crictl stop 129fd7d382018 # replace with the ID of your container
sleep 20
crictl ps
```

```
CONTAINER       IMAGE                                 CREATED           STATE      NAME    ATTEMPT    POD ID
89db4553e1eeb   docker.io/library/nginx@sha256:...    19 seconds ago    Running    web     1          34533c6729106
```

Once you identify the right container, you can get the logs for that container with `crictl`:

```
# Run these commands on the node where the container is running
crictl logs <container_id>
```

```
10.240.0.48 - - [16/Nov/2022:12:45:49 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
10.240.0.48 - - [16/Nov/2022:12:45:50 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
10.240.0.48 - - [16/Nove/2022:12:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

To find more about how to debug using `crictl`, please visit *Debugging Kubernetes nodes with crictl*.

Dynamic addition and removal of static pods

The running kubelet periodically scans the configured directory (`/etc/kubernetes/manifests` in our example) for changes and adds/removes Pods as files appear/disappear in this directory.

```
# This assumes you are using filesystem-hosted static Pod configuration
# Run these commands on the node where the container is running
#
mv /etc/kubernetes/manifests/static-web.yaml /tmp
```

```
sleep 20
crictl ps
# You see that no nginx container is running
mv /tmp/static-web.yaml  /etc/kubernetes/manifests/
sleep 20
crictl ps
```

```
CONTAINER       IMAGE                                 CREATED           STATE      NAME    ATTEMPT    POD ID
f427638871c35   docker.io/library/nginx@sha256:...    19 seconds ago    Running    web     1          34533c6729106
```

What's next

- Generate static Pod manifests for control plane components
- Generate static Pod manifest for local etcd
- Debugging Kubernetes nodes with `crictl`
- Learn more about `crictl`
- Map `docker` CLI commands to `crictl`
- Set up etcd instances as static pods managed by a kubelet

Assign Extended Resources to a Container

FEATURE STATE: `Kubernetes v1.34 [stable]`

This page shows how to assign extended resources to a Container.

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`.

Before you do this exercise, do the exercise in Advertise Extended Resources for a Node. That will configure one of your Nodes to advertise a dongle resource.

Assign an extended resource to a Pod

To request an extended resource, include the `resources:requests` field in your Container manifest. Extended resources are fully qualified with any domain outside of `*.kubernetes.io/`. Valid extended resource names have the form `example.com/foo` where `example.com` is replaced with your organization's domain and `foo` is a descriptive resource name.

Here is the configuration file for a Pod that has one Container:

`pods/resource/extended-resource-pod.yaml` Copy pods/resource/extended-resource-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:   name: extended-resource-demospec:   containers:   - name: extended-resource-demo-ctr    image: nginx    resource
```

In the configuration file, you can see that the Container requests 3 dongles.

Create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/extended-resource-pod.yaml
```

Verify that the Pod is running:

```
kubectl get pod extended-resource-demo
```

Describe the Pod:

```
kubectl describe pod extended-resource-demo
```

The output shows dongle requests:

```
Limits:
  example.com/dongle: 3
Requests:   example.com/dongle: 3
```

Attempt to create a second Pod

Here is the configuration file for a Pod that has one Container. The Container requests two dongles.

`pods/resource/extended-resource-pod-2.yaml` Copy pods/resource/extended-resource-pod-2.yaml to clipboard

```
apiVersion: v1
```

```
kind: Podmetadata:  name: extended-resource-demo-2spec:  containers:  - name: extended-resource-demo-2-ctr    image: nginx    reso
```

Kubernetes will not be able to satisfy the request for two dongles, because the first Pod used three of the four available dongles.

Attempt to create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/extended-resource-pod-2.yaml
```

Describe the Pod

```
kubectl describe pod extended-resource-demo-2
```

The output shows that the Pod cannot be scheduled, because there is no Node that has 2 dongles available:

```
Conditions:
  Type    Status
  PodScheduled  False
...
Events:
  ...
  ... Warning   FailedScheduling  pod (extended-resource-demo-2) failed to fit in any node
fit failure summary on nodes : Insufficient example.com/dongle (1)
```

View the Pod status:

```
kubectl get pod extended-resource-demo-2
```

The output shows that the Pod was created, but not scheduled to run on a Node. It has a status of Pending:

```
NAME                       READY     STATUS    RESTARTS   AGE
extended-resource-demo-2   0/1       Pending   0          6m
```

Clean up

Delete the Pods that you created for this exercise:

```
kubectl delete pod extended-resource-demo
kubectl delete pod extended-resource-demo-2
```

What's next

For application developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Advertise Extended Resources for a Node

Configure GMSA for Windows Pods and containers

FEATURE STATE: `Kubernetes v1.18 [stable]`

This page shows how to configure Group Managed Service Accounts (GMSA) for Pods and containers that will run on Windows nodes. Group Managed Service Accounts are a specific type of Active Directory account that provides automatic password management, simplified service principal name (SPN) management, and the ability to delegate the management to other administrators across multiple servers.

In Kubernetes, GMSA credential specs are configured at a Kubernetes cluster-wide scope as Custom Resources. Windows Pods, as well as individual containers within a Pod, can be configured to use a GMSA for domain based functions (e.g. Kerberos authentication) when interacting with other Windows services.

Before you begin

You need to have a Kubernetes cluster and the `kubectl` command-line tool must be configured to communicate with your cluster. The cluster is expected to have Windows worker nodes. This section covers a set of initial steps required once for each cluster:

Install the GMSACredentialSpec CRD

A CustomResourceDefinition(CRD) for GMSA credential spec resources needs to be configured on the cluster to define the custom resource type `GMSACredentialSpec`. Download the GMSA CRD YAML and save it as gmsa-crd.yaml. Next, install the CRD with `kubectl apply -f gmsa-crd.yaml`

Install webhooks to validate GMSA users

Two webhooks need to be configured on the Kubernetes cluster to populate and validate GMSA credential spec references at the Pod or container level:

1. A mutating webhook that expands references to GMSAs (by name from a Pod specification) into the full credential spec in JSON form within the Pod spec.
2. A validating webhook ensures all references to GMSAs are authorized to be used by the Pod service account.

Installing the above webhooks and associated objects require the steps below:

1. Create a certificate key pair (that will be used to allow the webhook container to communicate to the cluster)
2. Install a secret with the certificate from above.
3. Create a deployment for the core webhook logic.
4. Create the validating and mutating webhook configurations referring to the deployment.

A script can be used to deploy and configure the GMSA webhooks and associated objects mentioned above. The script can be run with a `--dry-run=server` option to allow you to review the changes that would be made to your cluster.

The YAML template used by the script may also be used to deploy the webhooks and associated objects manually (with appropriate substitutions for the parameters)

Configure GMSAs and Windows nodes in Active Directory

Before Pods in Kubernetes can be configured to use GMSAs, the desired GMSAs need to be provisioned in Active Directory as described in the Windows GMSA documentation. Windows worker nodes (that are part of the Kubernetes cluster) need to be configured in Active Directory to access the secret credentials associated with the desired GMSA as described in the Windows GMSA documentation.

Create GMSA credential spec resources

With the GMSACredentialSpec CRD installed (as described earlier), custom resources containing GMSA credential specs can be configured. The GMSA credential spec does not contain secret or sensitive data. It is information that a container runtime can use to describe the desired GMSA of a container to Windows. GMSA credential specs can be generated in YAML format with a utility PowerShell script.

Following are the steps for generating a GMSA credential spec YAML manually in JSON format and then converting it:

1. Import the CredentialSpec module: `ipmo CredentialSpec.psm1`
2. Create a credential spec in JSON format using `New-CredentialSpec`. To create a GMSA credential spec named WebApp1, invoke `New-CredentialSpec -Name WebApp1 -AccountName WebApp1 -Domain $(Get-ADDomain -Current LocalComputer)`
3. Use `Get-CredentialSpec` to show the path of the JSON file.
4. Convert the credspec file from JSON to YAML format and apply the necessary header fields `apiVersion`, `kind`, `metadata` and `credspec` to make it a GMSACredentialSpec custom resource that can be configured in Kubernetes.

The following YAML configuration describes a GMSA credential spec named `gmsa-WebApp1`:

```
apiVersion: windows.k8s.io/v1
kind: GMSACredentialSpecmetadata:  name: gmsa-WebApp1  # This is an arbitrary name but it will be used as a referencecredspec:  Ac
```

The above credential spec resource may be saved as `gmsa-Webapp1-credspec.yaml` and applied to the cluster using: `kubectl apply -f gmsa-Webapp1-credspec.yml`

Configure cluster role to enable RBAC on specific GMSA credential specs

A cluster role needs to be defined for each GMSA credential spec resource. This authorizes the `use` verb on a specific GMSA resource by a subject which is typically a service account. The following example shows a cluster role that authorizes usage of the `gmsa-WebApp1` credential spec from above. Save the file as gmsa-webapp1-role.yaml and apply using `kubectl apply -f gmsa-webapp1-role.yaml`

```
# Create the Role to read the credspec
apiVersion: rbac.authorization.k8s.io/v1kind: ClusterRolemetadata:  name: webapp1-rolerules:- apiGroups: ["windows.k8s.io"]  resou
```

Assign role to service accounts to use specific GMSA credspecs

A service account (that Pods will be configured with) needs to be bound to the cluster role create above. This authorizes the service account to use the desired GMSA credential spec resource. The following shows the default service account being bound to a cluster role `webapp1-role` to use `gmsa-WebApp1` credential spec resource created above.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBindingmetadata:  name: allow-default-svc-account-read-on-gmsa-WebApp1  namespace: defaultsubjects:- kind: ServiceAccoun
```

Configure GMSA credential spec reference in Pod spec

The Pod spec field `securityContext.windowsOptions.gmsaCredentialSpecName` is used to specify references to desired GMSA credential spec custom resources in Pod specs. This configures all containers in the Pod spec to use the specified GMSA. A sample Pod spec with the annotation populated to refer to `gmsa-WebApp1`:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  labels:    run: with-creds  name: with-creds  namespace: defaultspec:  replicas: 1  selector:    matchLa
```

Individual containers in a Pod spec can also specify the desired GMSA credspec using a per-container `securityContext.windowsOptions.gmsaCredentialSpecName` field. For example:

```
apiVersion: apps/v1
kind: Deploymentmetadata:  labels:    run: with-creds  name: with-creds  namespace: defaultspec:  replicas: 1  selector:    matchLa
```

As Pod specs with GMSA fields populated (as described above) are applied in a cluster, the following sequence of events take place:

1. The mutating webhook resolves and expands all references to GMSA credential spec resources to the contents of the GMSA credential spec.
2. The validating webhook ensures the service account associated with the Pod is authorized for the `use` verb on the specified GMSA credential spec.
3. The container runtime configures each Windows container with the specified GMSA credential spec so that the container can assume the identity of the GMSA in Active Directory and access services in the domain using that identity.

Authenticating to network shares using hostname or FQDN

If you are experiencing issues connecting to SMB shares from Pods using hostname or FQDN, but are able to access the shares via their IPv4 address then make sure the following registry key is set on the Windows nodes.

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\hns\State" /v EnableCompartmentNamespace /t REG_DWORD /d 1
```

Running Pods will then need to be recreated to pick up the behavior changes. More information on how this registry key is used can be found here

Troubleshooting

If you are having difficulties getting GMSA to work in your environment, there are a few troubleshooting steps you can take.

First, make sure the credspec has been passed to the Pod. To do this you will need to `exec` into one of your Pods and check the output of the `nltest.exe /parentdomain` command.

In the example below the Pod did not get the credspec correctly:

```
kubectl exec -it iis-auth-7776966999-n5nzr powershell.exe
```

`nltest.exe /parentdomain` results in the following error:

```
Getting parent domain failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
```

If your Pod did get the credspec correctly, then next check communication with the domain. First, from inside of your Pod, quickly do an nslookup to find the root of your domain.

This will tell us 3 things:

1. The Pod can reach the DC
2. The DC can reach the Pod
3. DNS is working correctly.

If the DNS and communication test passes, next you will need to check if the Pod has established secure channel communication with the domain. To do this, again, `exec` into your Pod and run the `nltest.exe /query` command.

```
nltest.exe /query
```

Results in the following output:

```
I_NetLogonControl failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
```

This tells us that for some reason, the Pod was unable to logon to the domain using the account specified in the credspec. You can try to repair the secure channel by running the following:

```
nltest /sc_reset:domain.example
```

If the command is successful you will see and output similar to this:

```
Flags: 30 HAS_IP  HAS_TIMESERV
Trusted DC Name \\dc10.domain.example
Trusted DC Connection Status Status = 0 0x0 NERR_Success
The command completed successfully
```

If the above corrects the error, you can automate the step by adding the following lifecycle hook to your Pod spec. If it did not correct the error, you will need to examine your credspec again and confirm that it is correct and complete.

```
        image: registry.domain.example/iis-auth:1809v1
        lifecycle:
          postStart:
            exec:
              command: ["powershell.exe","-command","do { Restart-Service -Name netlogon } while ( $($Result = (nltest.exe /query)
        imagePullPolicy: IfNotPresent
```

If you add the `lifecycle` section show above to your Pod spec, the Pod will execute the commands listed to restart the `netlogon` service until the `nltest.exe /query` command exits without error.

Configure RunAsUserName for Windows pods and containers

FEATURE STATE: `Kubernetes v1.18 [stable]`

This page shows how to use the `runAsUserName` setting for Pods and containers that will run on Windows nodes. This is roughly equivalent of the Linux-specific `runAsUser` setting, allowing you to run applications in a container as a different username than the default.

Before you begin

You need to have a Kubernetes cluster and the kubectl command-line tool must be configured to communicate with your cluster. The cluster is expected to have Windows worker nodes where pods with containers running Windows workloads will get scheduled.

Set the Username for a Pod

To specify the username with which to execute the Pod's container processes, include the `securityContext` field (PodSecurityContext) in the Pod specification, and within it, the `windowsOptions` (WindowsSecurityContextOptions) field containing the `runAsUserName` field.

The Windows security context options that you specify for a Pod apply to all Containers and init Containers in the Pod.

Here is a configuration file for a Windows Pod that has the `runAsUserName` field set:

`windows/run-as-username-pod.yaml` Copy windows/run-as-username-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: run-as-username-pod-demospec:  securityContext:    windowsOptions:      runAsUserName: "ContainerUser"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-pod.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod run-as-username-pod-demo
```

Get a shell to the running Container:

```
kubectl exec -it run-as-username-pod-demo -- powershell
```

Check that the shell is running user the correct username:

```
echo $env:USERNAME
```

The output should be:

```
ContainerUser
```

Set the Username for a Container

To specify the username with which to execute a Container's processes, include the `securityContext` field (SecurityContext) in the Container manifest, and within it, the `windowsOptions` (WindowsSecurityContextOptions) field containing the `runAsUserName` field.

The Windows security context options that you specify for a Container apply only to that individual Container, and they override the settings made at the Pod level.

Here is the configuration file for a Pod that has one Container, and the `runAsUserName` field is set at the Pod level and the Container level:

`windows/run-as-username-container.yaml` Copy windows/run-as-username-container.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: run-as-username-container-demospec:  securityContext:    windowsOptions:      runAsUserName: "ContainerU
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-container.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod run-as-username-container-demo
```

Get a shell to the running Container:

```
kubectl exec -it run-as-username-container-demo -- powershell
```

Check that the shell is running user the correct username (the one set at the Container level):

```
echo $env:USERNAME
```

The output should be:

```
ContainerAdministrator
```

Windows Username limitations

In order to use this feature, the value set in the `runAsUserName` field must be a valid username. It must have the following format: `DOMAIN\USER`, where `DOMAIN\` is optional. Windows user names are case insensitive. Additionally, there are some restrictions regarding the `DOMAIN` and `USER`:

- The `runAsUserName` field cannot be empty, and it cannot contain control characters (ASCII values: `0x00-0x1F`, `0x7F`)
- The `DOMAIN` must be either a NetBios name, or a DNS name, each with their own restrictions:
  - NetBios names: maximum 15 characters, cannot start with `.` (dot), and cannot contain the following characters: `\ / : * ? " < > |`
  - DNS names: maximum 255 characters, contains only alphanumeric characters, dots, and dashes, and it cannot start or end with a `.` (dot) or `-` (dash).

- The `USER` must have at most 20 characters, it cannot contain *only* dots or spaces, and it cannot contain the following characters: `" / \ [ ] : ; | = , + * ? < > @`.

Examples of acceptable values for the `runAsUserName` field: `ContainerAdministrator`, `ContainerUser`, `NT AUTHORITY\NETWORK SERVICE`, `NT AUTHORITY\LOCAL SERVICE`.

For more information about these limtations, check here and here.

What's next

- Guide for scheduling Windows containers in Kubernetes
- Managing Workload Identity with Group Managed Service Accounts (GMSA)
- Configure GMSA for Windows pods and containers

Assign Pod-level CPU and memory resources

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

This page shows how to specify CPU and memory resources for a Pod at pod-level in addition to container-level resource specifications. A Kubernetes node allocates resources to a pod based on the pod's resource requests. These requests can be defined at the pod level or individually for containers within the pod. When both are present, the pod-level requests take precedence.

Similarly, a pod's resource usage is restricted by limits, which can also be set at the pod-level or individually for containers within the pod. Again, pod-level limits are prioritized when both are present. This allows for flexible resource management, enabling you to control resource allocation at both the pod and container levels.

In order to specify the resources at pod-level, it is required to enable `PodLevelResources` feature gate.

For Pod Level Resources:

- Priority: When both pod-level and container-level resources are specified, pod-level resources take precedence.
- QoS: Pod-level resources take precedence in influencing the QoS class of the pod.
- OOM Score: The OOM score adjustment calculation considers both pod-level and container-level resources.
- Compatibility: Pod-level resources are designed to be compatible with existing features.

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 1.34.

To check the version, enter `kubectl version`.

The `PodLevelResources` feature gate must be enabled for your control plane and for all nodes in your cluster.

Limitations

For Kubernetes 1.34, resizing pod-level resources has the following limitations:

- **Resource Types:** Only CPU, memory and hugepages resources can be specified at pod-level.
- **Operating System:** Pod-level resources are not supported for Windows pods.
- **Resource Managers:** The Topology Manager, Memory Manager and CPU Manager do not align pods and containers based on pod-level resources as these resource managers don't currently support pod-level resources.
- **In-Place Resize:** In-place resize of pod-level resources is not supported. Modifying the pod-level resource limits or requests on a pod result in a field.Forbidden error. The error message explicitly states, "pods with pod-level resources cannot be resized."

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace pod-resources-example
```

Create a pod with memory requests and limits at pod-level

To specify memory requests for a Pod at pod-level, include the `resources.requests.memory` field in the Pod spec manifest. To specify a memory limit, include `resources.limits.memory`.

In this exercise, you create a Pod that has one Container. The Pod has a memory request of 100 MiB and a memory limit of 200 MiB. Here's the configuration file for the Pod:

pods/resource/pod-level-memory-request-limit.yaml Copy pods/resource/pod-level-memory-request-limit.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: memory-demo  namespace: pod-resources-examplespec:  resources:    requests:      memory: "100Mi"    limi
```

The `args` section in the manifest provides arguments for the container when it starts. The `"--vm-bytes", "150M"` arguments tell the Container to attempt to allocate 150 MiB of memory.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/pod-level-memory-request-limit.yaml --namespace=pod-resources-example
```

Verify that the Pod is running:

```
kubectl get pod memory-demo --namespace=pod-resources-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo --output=yaml --namespace=pod-resources-example
```

The output shows that the Pod has a memory request of 100 MiB and a memory limit of 200 MiB.

```
...
spec:   containers:     ...  resources:    requests:      memory: 100Mi    limits:      memory: 200Mi...
```

Run `kubectl top` to fetch the metrics for the pod:

```
kubectl top pod memory-demo --namespace=pod-resources-example
```

The output shows that the Pod is using about 162,900,000 bytes of memory, which is about 150 MiB. This is greater than the Pod's 100 MiB request, but within the Pod's 200 MiB limit.

```
NAME                        CPU(cores)   MEMORY(bytes)
memory-demo                 <something>  162856960
```

Create a pod with CPU requests and limits at pod-level

To specify a CPU request for a Pod, include the `resources.requests.cpu` field in the Pod spec manifest. To specify a CPU limit, include `resources.limits.cpu`.

In this exercise, you create a Pod that has one container. The Pod has a request of 0.5 CPU and a limit of 1 CPU. Here is the configuration file for the Pod:

pods/resource/pod-level-cpu-request-limit.yaml Copy pods/resource/pod-level-cpu-request-limit.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: cpu-demo  namespace: pod-resources-examplespec:  resources:    limits:      cpu: "1"    requests:      c
```

The `args` section of the configuration file provides arguments for the container when it starts. The `-cpus "2"` argument tells the Container to attempt to use 2 CPUs.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/pod-level-cpu-request-limit.yaml --namespace=pod-resources-example
```

Verify that the Pod is running:

```
kubectl get pod cpu-demo --namespace=pod-resources-example
```

View detailed information about the Pod:

```
kubectl get pod cpu-demo --output=yaml --namespace=pod-resources-example
```

The output shows that the Pod has a CPU request of 500 milliCPU and a CPU limit of 1 CPU.

```
spec:
  containers:
  ...
  resources:
    limits:
      cpu: "1"
    requests:
      cpu: 500m
```

Use `kubectl top` to fetch the metrics for the Pod:

```
kubectl top pod cpu-demo --namespace=pod-resources-example
```

This example output shows that the Pod is using 974 milliCPU, which is slightly less than the limit of 1 CPU specified in the Pod configuration.

```
NAME                        CPU(cores)   MEMORY(bytes)
cpu-demo                    974m         <something>
```

Recall that by setting `-cpu "2"`, you configured the Container to attempt to use 2 CPUs, but the Container is only being allowed to use about 1 CPU. The container's CPU use is being throttled, because the container is attempting to use more CPU resources than the Pod CPU limit.

Create a pod with resource requests and limits at both pod-level and container-level

To assign CPU and memory resources to a Pod, you can specify them at both the pod level and the container level. Include the `resources` field in the Pod spec to define resources for the entire Pod. Additionally, include the `resources` field within container's specification in the Pod's manifest to set container-specific resource requirements.

In this exercise, you'll create a Pod with two containers to explore the interaction of pod-level and container-level resource specifications. The Pod itself will have defined CPU requests and limits, while only one of the containers will have its own explicit resource requests and limits. The other container will inherit the resource constraints from the pod-level settings. Here's the configuration file for the Pod:

pods/resource/pod-level-resources.yaml Copy pods/resource/pod-level-resources.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: pod-resources-demo  namespace: pod-resources-examplespec:  resources:    limits:      cpu: "1"      memo
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/pod-level-resources.yaml --namespace=pod-resources-example
```

Verify that the Pod Container is running:

```
kubectl get pod-resources-demo --namespace=pod-resources-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo --output=yaml --namespace=pod-resources-example
```

The output shows that one container in the Pod has a memory request of 50 MiB and a CPU request of 0.5 cores, with a memory limit of 100 MiB and a CPU limit of 0.5 cores. The Pod itself has a memory request of 100 MiB and a CPU request of 1 core, and a memory limit of 200 MiB and a CPU limit of 1 core.

```
...
containers:  name: pod-resources-demo-ctr-1  resources:    requests:      cpu: 500m      memory: 50Mi    limits:      cp
```

Since pod-level requests and limits are specified, the request guarantees for both containers in the pod will be equal 1 core or CPU and 100Mi of memory. Additionally, both containers together won't be able to use more resources than specified in the pod-level limits, ensuring they cannot exceed a combined total of 200 MiB of memory and 1 core of CPU.

Clean up

Delete your namespace:

```
kubectl delete namespace pod-resources-example
```

What's next

For application developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace

Managing Secrets

Managing confidential settings data using Secrets.

Managing Secrets using kubectl

Creating Secret objects using kubectl command line.

Managing Secrets using Configuration File

Creating Secret objects using resource configuration file.

Managing Secrets using Kustomize

Creating Secret objects using kustomization.yaml file.

Configure Quality of Service for Pods

This page shows how to configure Pods so that they will be assigned particular Quality of Service (QoS) classes. Kubernetes uses QoS classes to make decisions about evicting Pods when Node resources are exceeded.

When Kubernetes creates a Pod it assigns one of these QoS classes to the Pod:

- Guaranteed
- Burstable
- BestEffort

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 also need to be able to create and delete namespaces.

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace qos-example
```

Create a Pod that gets assigned a QoS class of Guaranteed

For a Pod to be given a QoS class of `Guaranteed`:

- Every Container in the Pod must have a memory limit and a memory request.
- For every Container in the Pod, the memory limit must equal the memory request.
- Every Container in the Pod must have a CPU limit and a CPU request.
- For every Container in the Pod, the CPU limit must equal the CPU request.

These restrictions apply to init containers and app containers equally. Ephemeral containers cannot define resources so these restrictions do not apply.

Here is a manifest for a Pod that has one Container. The Container has a memory limit and a memory request, both equal to 200 MiB. The Container has a CPU limit and a CPU request, both equal to 700 milliCPU:

`pods/qos/qos-pod.yaml` Copy pods/qos/qos-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: qos-demo  namespace: qos-examplespec:  containers:  - name: qos-demo-ctr    image: nginx    resources:
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod.yaml --namespace=qos-example
```

View detailed information about the Pod:

```
kubectl get pod qos-demo --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of `Guaranteed`. The output also verifies that the Pod Container has a memory request that matches its memory limit, and it has a CPU request that matches its CPU limit.

```
spec:
  containers:
    ...
    resources:
      limits:
        cpu: 700m
        memory: 200Mi
      requests:
        cpu: 700m
        memory: 200Mi
    ...
status:  qosClass: Guaranteed
```

Note:

If a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit. Similarly, if a Container specifies its own CPU limit, but does not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit.

Clean up

Delete your Pod:

```
kubectl delete pod qos-demo --namespace=qos-example
```

Create a Pod that gets assigned a QoS class of Burstable

A Pod is given a QoS class of `Burstable` if:

- The Pod does not meet the criteria for QoS class `Guaranteed`.
- At least one Container in the Pod has a memory or CPU request or limit.

Here is a manifest for a Pod that has one Container. The Container has a memory limit of 200 MiB and a memory request of 100 MiB.

pods/qos/qos-pod-2.yaml Copy pods/qos/qos-pod-2.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: qos-demo-2  namespace: qos-examplespec:  containers:  - name: qos-demo-2-ctr    image: nginx    resource
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-2.yaml --namespace=qos-example
```

View detailed information about the Pod:

```
kubectl get pod qos-demo-2 --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of `Burstable`:

```
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: qos-demo-2-ctr
    resources:
      limits:
        memory: 200Mi
      requests:
        memory: 100Mi
  ...
status:  qosClass: Burstable
```

Clean up

Delete your Pod:

```
kubectl delete pod qos-demo-2 --namespace=qos-example
```

Create a Pod that gets assigned a QoS class of BestEffort

For a Pod to be given a QoS class of `BestEffort`, the Containers in the Pod must not have any memory or CPU limits or requests.

Here is a manifest for a Pod that has one Container. The Container has no memory or CPU limits or requests:

pods/qos/qos-pod-3.yaml Copy pods/qos/qos-pod-3.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: qos-demo-3  namespace: qos-examplespec:  containers:  - name: qos-demo-3-ctr    image: nginx
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-3.yaml --namespace=qos-example
```

View detailed information about the Pod:

```
kubectl get pod qos-demo-3 --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of `BestEffort`:

```
spec:
  containers:
    ...
    resources: {}
  ...
status:  qosClass: BestEffort
```

Clean up

Delete your Pod:

```
kubectl delete pod qos-demo-3 --namespace=qos-example
```

Create a Pod that has two Containers

Here is a manifest for a Pod that has two Containers. One container specifies a memory request of 200 MiB. The other Container does not specify any requests or limits.

pods/qos/qos-pod-4.yaml Copy pods/qos/qos-pod-4.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: qos-demo-4  namespace: qos-examplespec:  containers:  - name: qos-demo-4-ctr-1    image: nginx    resour
```

Notice that this Pod meets the criteria for QoS class `Burstable`. That is, it does not meet the criteria for QoS class `Guaranteed`, and one of its Containers has a memory request.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-4.yaml --namespace=qos-example
```

View detailed information about the Pod:

```
kubectl get pod qos-demo-4 --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of `Burstable`:

```
spec:
  containers:
    ...
    name: qos-demo-4-ctr-1
    resources:
      requests:
        memory: 200Mi
    ...
    name: qos-demo-4-ctr-2
    resources: {}
    ...
status:  qosClass: Burstable
```

Retrieve the QoS class for a Pod

Rather than see all the fields, you can view just the field you need:

```
kubectl --namespace=qos-example get pod qos-demo-4 -o jsonpath='{ .status.qosClass}{"\n"}'
```

```
Burstable
```

Clean up

Delete your namespace:

```
kubectl delete namespace qos-example
```

What's next

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects
- Control Topology Management policies on a node

Enforce Pod Security Standards with Namespace Labels

Namespaces can be labeled to enforce the Pod Security Standards. The three policies privileged, baseline and restricted broadly cover the security spectrum and are implemented by the Pod Security admission controller.

Before you begin

Pod Security Admission was available by default in Kubernetes v1.23, as a beta. From version 1.25 onwards, Pod Security Admission is generally available.

To check the version, enter `kubectl version`.

Requiring the `baseline` Pod Security Standard with namespace labels

This manifest defines a Namespace `my-baseline-namespace` that:

- *Blocks* any pods that don't satisfy the `baseline` policy requirements.
- Generates a user-facing warning and adds an audit annotation to any created pod that does not meet the `restricted` policy requirements.
- Pins the versions of the `baseline` and `restricted` policies to v1.34.

```
apiVersion: v1
kind: Namespacemetadata:  name: my-baseline-namespace  labels:    pod-security.kubernetes.io/enforce: baseline    pod-security.kub
```

Add labels to existing namespaces with `kubectl label`

Note:

When an `enforce` policy (or version) label is added or changed, the admission plugin will test each pod in the namespace against the new policy. Violations are returned to the user as warnings.

It is helpful to apply the `--dry-run` flag when initially evaluating security profile changes for namespaces. The Pod Security Standard checks will still be run in *dry run* mode, giving you information about how the new policy would treat existing pods, without actually updating a policy.

```
kubectl label --dry-run=server --overwrite ns --all \
    pod-security.kubernetes.io/enforce=baseline
```

Applying to all namespaces

If you're just getting started with the Pod Security Standards, a suitable first step would be to configure all namespaces with audit annotations for a stricter level such as `baseline`:

```
kubectl label --overwrite ns --all \
  pod-security.kubernetes.io/audit=baseline \  pod-security.kubernetes.io/warn=baseline
```

Note that this is not setting an enforce level, so that namespaces that haven't been explicitly evaluated can be distinguished. You can list namespaces without an explicitly set enforce level using this command:

```
kubectl get namespaces --selector='!pod-security.kubernetes.io/enforce'
```

Applying to a single namespace

You can update a specific namespace as well. This command adds the `enforce=restricted` policy to `my-existing-namespace`, pinning the restricted policy version to v1.34.

```
kubectl label --overwrite ns my-existing-namespace \
  pod-security.kubernetes.io/enforce=restricted \  pod-security.kubernetes.io/enforce-version=v1.34
```

Set Up DRA in a Cluster

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

This page shows you how to configure *dynamic resource allocation (DRA)* in a Kubernetes cluster by enabling API groups and configuring classes of devices. These instructions are for cluster administrators.

About DRA

A Kubernetes feature that lets you request and share resources among Pods. These resources are often attached devices like hardware accelerators.

With DRA, device drivers and cluster admins define device *classes* that are available to *claim* in workloads. Kubernetes allocates matching devices to specific claims and places the corresponding Pods on nodes that can access the allocated devices.

Ensure that you're familiar with how DRA works and with DRA terminology like DeviceClasses, ResourceClaims, and ResourceClaimTemplates. For details, see Dynamic Resource Allocation (DRA).

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 version v1.34.

To check the version, enter `kubectl version`.

- Directly or indirectly attach devices to your cluster. To avoid potential issues with drivers, wait until you set up the DRA feature for your cluster before you install drivers.

Optional: enable legacy DRA API groups

DRA graduated to stable in Kubernetes 1.34 and is enabled by default. Some older DRA drivers or workloads might still need the v1beta1 API from Kubernetes 1.30 or v1beta2 from Kubernetes 1.32. If and only if support for those is desired, then enable the following API groups:

```
* `resource.k8s.io/v1beta1`
* `resource.k8s.io/v1beta2`
```

For more information, see Enabling or disabling API groups.

Verify that DRA is enabled

To verify that the cluster is configured correctly, try to list DeviceClasses:

```
kubectl get deviceclasses
```

If the component configuration was correct, the output is similar to the following:

```
No resources found
```

If DRA isn't correctly configured, the output of the preceding command is similar to the following:

```
error: the server doesn't have a resource type "deviceclasses"
```

Try the following troubleshooting steps:

1. Reconfigure and restart the `kube-apiserver` component.

2. If the complete `.spec.resourceClaims` field gets removed from Pods, or if Pods get scheduled without considering the ResourceClaims, then verify that the `DynamicResourceAllocation` feature gate is not turned off for kube-apiserver, kube-controller-manager, kube-scheduler or the kubelet.

Install device drivers

After you enable DRA for your cluster, you can install the drivers for your attached devices. For instructions, check the documentation of your device owner or the project that maintains the device drivers. The drivers that you install must be compatible with DRA.

To verify that your installed drivers are working as expected, list ResourceSlices in your cluster:

```
kubectl get resourceslices
```

The output is similar to the following:

```
NAME                                                 NODE              DRIVER              POOL                              AG
cluster-1-device-pool-1-driver.example.com-lqx8x     cluster-1-node-1  driver.example.com  cluster-1-device-pool-1-r1gc      7s
cluster-1-device-pool-2-driver.example.com-29t7b     cluster-1-node-2  driver.example.com  cluster-1-device-pool-2-446z      8s
```

Try the following troubleshooting steps:

1. Check the health of the DRA driver and look for error messages about publishing ResourceSlices in its log output. The vendor of the driver may have further instructions about installation and troubleshooting.

Create DeviceClasses

You can define categories of devices that your application operators can claim in workloads by creating DeviceClasses. Some device driver providers might also instruct you to create DeviceClasses during driver installation.

The ResourceSlices that your driver publishes contain information about the devices that the driver manages, such as capacity, metadata, and attributes. You can use Common Expression Language to filter for properties in your DeviceClasses, which can make finding devices easier for your workload operators.

1. To find the device properties that you can select in DeviceClasses by using CEL expressions, get the specification of a ResourceSlice:

   ```
   kubectl get resourceslice <resourceslice-name> -o yaml
   ```

   The output is similar to the following:

   ```
   apiVersion: resource.k8s.io/v1
   kind: ResourceSlice# lines omitted for clarityspec:  devices:  - attributes:      type:         string: gpu    capacity:
   ```

   You can also check the driver provider's documentation for available properties and values.

2. Review the following example DeviceClass manifest, which selects any device that's managed by the `driver.example.com` device driver:

   dra/deviceclass.yaml Copy dra/deviceclass.yaml to clipboard

   ```
   apiVersion: resource.k8s.io/v1
   kind: DeviceClassmetadata:  name: example-device-classspec:  selectors:  - cel:      expression: |-        device.driver == "
   ```

3. Create the DeviceClass in your cluster:

   ```
   kubectl apply -f https://k8s.io/examples/dra/deviceclass.yaml
   ```

Clean up

To delete the DeviceClass that you created in this task, run the following command:

```
kubectl delete -f https://k8s.io/examples/dra/deviceclass.yaml
```

What's next

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

Configure Liveness, Readiness and Startup Probes

This page shows how to configure liveness, readiness and startup probes for containers.

For more information about probes, see Liveness, Readiness and Startup Probes

The kubelet uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

A common pattern for liveness probes is to use the same low-cost HTTP endpoint as for readiness probes, but with a higher failureThreshold. This ensures that the pod is observed as not-ready for some period of time before it is hard killed.

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. One use of this signal is to control which Pods are used as backends for Services. A Pod is considered ready when its `Ready` condition is true. When a Pod is not ready, it is removed from Service load balancers. A Pod's `Ready` condition is false when its Node's `Ready` condition is not true, when one of the Pod's `readinessGates` is false, or when at least one of its containers is not ready.

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, liveness and readiness probes do not start until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

Caution:

Liveness probes can be a powerful way to recover from application failures, but they should be used with caution. Liveness probes must be configured carefully to ensure that they truly indicate unrecoverable application failure, for example a deadlock.

Note:

Incorrect implementation of liveness probes can lead to cascading failures. This results in restarting of container under high load; failed client requests as your application became less scalable; and increased workload on remaining pods due to some failed pods. Understand the difference between readiness and liveness probes and when to apply them for your app.

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

Define a liveness command

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides liveness probes to detect and remedy such situations.

In this exercise, you create a Pod that runs a container based on the `registry.k8s.io/busybox:1.27.2` image. Here is the configuration file for the Pod:

`pods/probe/exec-liveness.yaml` Copy pods/probe/exec-liveness.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  labels:    test: liveness  name: liveness-execspec:  containers:  - name: liveness    image: registry.k8s.io/b
```

In the configuration file, you can see that the Pod has a single `Container`. The `periodSeconds` field specifies that the kubelet should perform a liveness probe every 5 seconds. The `initialDelaySeconds` field tells the kubelet that it should wait 5 seconds before performing the first probe. To perform a probe, the kubelet executes the command `cat /tmp/healthy` in the target container. If the command succeeds, it returns 0, and the kubelet considers the container to be alive and healthy. If the command returns a non-zero value, the kubelet kills the container and restarts it.

When the container starts, it executes this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600"
```

For the first 30 seconds of the container's life, there is a `/tmp/healthy` file. So during the first 30 seconds, the command `cat /tmp/healthy` returns a success code. After 30 seconds, `cat /tmp/healthy` returns a failure code.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/exec-liveness.yaml
```

Within 30 seconds, view the Pod events:

```
kubectl describe pod liveness-exec
```

The output indicates that no liveness probes have failed yet:

```
Type    Reason     Age   From               Message
----    ------     ----  ----               -------
Normal  Scheduled  11s   default-scheduler  Successfully assigned default/liveness-exec to node01
Normal  Pulling    9s    kubelet, node01    Pulling image "registry.k8s.io/busybox:1.27.2"
Normal  Pulled     7s    kubelet, node01    Successfully pulled image "registry.k8s.io/busybox:1.27.2"
Normal  Created    7s    kubelet, node01    Created container liveness
Normal  Started    7s    kubelet, node01    Started container liveness
```

After 35 seconds, view the Pod events again:

```
kubectl describe pod liveness-exec
```

At the bottom of the output, there are messages indicating that the liveness probes have failed, and the failed containers have been killed and recreated.

```
Type     Reason     Age                From               Message
----     ------     ----               ----               -------
Normal   Scheduled  57s                default-scheduler  Successfully assigned default/liveness-exec to node01
Normal   Pulling    55s                kubelet, node01    Pulling image "registry.k8s.io/busybox:1.27.2"
Normal   Pulled     53s                kubelet, node01    Successfully pulled image "registry.k8s.io/busybox:1.27.2"
Normal   Created    53s                kubelet, node01    Created container liveness
Normal   Started    53s                kubelet, node01    Started container liveness
Warning  Unhealthy  10s (x3 over 20s)  kubelet, node01    Liveness probe failed: cat: can't open '/tmp/healthy': No such file or d
Normal   Killing    10s                kubelet, node01    Container liveness failed liveness probe, will be restarted
```

Wait another 30 seconds, and verify that the container has been restarted:

```
kubectl get pod liveness-exec
```

The output shows that `RESTARTS` has been incremented. Note that the `RESTARTS` counter increments as soon as a failed container comes back to the running state:

```
NAME            READY     STATUS    RESTARTS   AGE
liveness-exec   1/1       Running   1          1m
```

Define a liveness HTTP request

Another kind of liveness probe uses an HTTP GET request. Here is the configuration file for a Pod that runs a container based on the `registry.k8s.io/e2e-test-images/agnhost` image.

pods/probe/http-liveness.yaml Copy pods/probe/http-liveness.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  labels:    test: liveness  name: liveness-httpspec:  containers:  - name: liveness    image: registry.k8s.io/e
```

In the configuration file, you can see that the Pod has a single container. The `periodSeconds` field specifies that the kubelet should perform a liveness probe every 3 seconds. The `initialDelaySeconds` field tells the kubelet that it should wait 3 seconds before performing the first probe. To perform a probe, the kubelet sends an HTTP GET request to the server that is running in the container and listening on port 8080. If the handler for the server's `/healthz` path returns a success code, the kubelet considers the container to be alive and healthy. If the handler returns a failure code, the kubelet kills the container and restarts it.

Any code greater than or equal to 200 and less than 400 indicates success. Any other code indicates failure.

You can see the source code for the server in server.go.

For the first 10 seconds that the container is alive, the `/healthz` handler returns a status of 200. After that, the handler returns a status of 500.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

The kubelet starts performing health checks 3 seconds after the container starts. So the first couple of health checks will succeed. But after 10 seconds, the health checks will fail, and the kubelet will kill and restart the container.

To try the HTTP liveness check, create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/http-liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the container has been restarted:

```
kubectl describe pod liveness-http
```

In releases after v1.13, local HTTP proxy environment variable settings do not affect the HTTP liveness probe.

Define a TCP liveness probe

A third type of liveness probe uses a TCP socket. With this configuration, the kubelet will attempt to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

pods/probe/tcp-liveness-readiness.yaml Copy pods/probe/tcp-liveness-readiness.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: goproxy  labels:    app: goproxyspec:  containers:  - name: goproxy    image: registry.k8s.io/goproxy:0.
```

As you can see, configuration for a TCP check is quite similar to an HTTP check. This example uses both readiness and liveness probes. The kubelet will run the first liveness probe 15 seconds after the container starts. This will attempt to connect to the `goproxy` container on port 8080. If the liveness probe fails, the container will be restarted. The kubelet will continue to run this check every 10 seconds.

In addition to the liveness probe, this configuration includes a readiness probe. The kubelet will run the first readiness probe 15 seconds after the container starts. Similar to the liveness probe, this will attempt to connect to the `goproxy` container on port 8080. If the probe succeeds, the Pod will be marked as ready and will receive traffic from services. If the readiness probe fails, the pod will be marked unready and will not receive traffic from any services.

To try the TCP liveness check, create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/tcp-liveness-readiness.yaml
```

After 15 seconds, view Pod events to verify that liveness probes:

```
kubectl describe pod goproxy
```

Define a gRPC liveness probe

FEATURE STATE: `Kubernetes v1.27 [stable]`

If your application implements the gRPC Health Checking Protocol, this example shows how to configure Kubernetes to use it for application liveness checks. Similarly you can configure readiness and startup probes.

Here is an example manifest:

pods/probe/grpc-liveness.yaml Copy pods/probe/grpc-liveness.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: etcd-with-grpcspec:  containers:  - name: etcd    image: registry.k8s.io/etcd:3.5.1-0    command: [ "/us
```

To use a gRPC probe, `port` must be configured. If you want to distinguish probes of different types and probes for different features you can use the `service` field. You can set `service` to the value `liveness` and make your gRPC Health Checking endpoint respond to this request differently than when you set `service` set to `readiness`. This lets you use the same endpoint for different kinds of container health check rather than listening on two different ports. If you want to specify your own custom service name and also specify a probe type, the Kubernetes project recommends that you use a name that concatenates those. For example: `myservice-liveness` (using `-` as a separator).

Note:

Unlike HTTP or TCP probes, you cannot specify the health check port by name, and you cannot configure a custom hostname.

Configuration problems (for example: incorrect port or service, unimplemented health checking protocol) are considered a probe failure, similar to HTTP and TCP probes.

To try the gRPC liveness check, create a Pod using the command below. In the example below, the etcd pod is configured to use gRPC liveness probe.

```
kubectl apply -f https://k8s.io/examples/pods/probe/grpc-liveness.yaml
```

After 15 seconds, view Pod events to verify that the liveness check has not failed:

```
kubectl describe pod etcd-with-grpc
```

When using a gRPC probe, there are some technical details to be aware of:

- The probes run against the pod IP address or its hostname. Be sure to configure your gRPC endpoint to listen on the Pod's IP address.
- The probes do not support any authentication parameters (like `-tls`).
- There are no error codes for built-in probes. All errors are considered as probe failures.
- If `ExecProbeTimeout` feature gate is set to `false`, grpc-health-probe does **not** respect the `timeoutSeconds` setting (which defaults to 1s), while built-in probe would fail on timeout.

Use a named port

You can use a named `port` for HTTP and TCP probes. gRPC probes do not support named ports.

For example:

```
ports:
- name: liveness-port  containerPort: 8080livenessProbe:  httpGet:    path: /healthz    port: liveness-port
```

Protect slow starting containers with startup probes

Sometimes, you have to deal with applications that require additional startup time on their first initialization. In such cases, it can be tricky to set up liveness probe parameters without compromising the fast response to deadlocks that motivated such a probe. The solution is to set up a startup probe with the same command, HTTP or TCP check, with a `failureThreshold * periodSeconds` long enough to cover the worst case startup time.

So, the previous example would become:

```
ports:
- name: liveness-port  containerPort: 8080livenessProbe:  httpGet:    path: /healthz    port: liveness-port  failureThreshold: 1
```

Thanks to the startup probe, the application will have a maximum of 5 minutes (30 * 10 = 300s) to finish its startup. Once the startup probe has succeeded once, the liveness probe takes over to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's `restartPolicy`.

Define readiness probes

Sometimes, applications are temporarily unable to serve traffic. For example, an application might need to load large data or configuration files during startup, or depend on external services after startup. In such cases, you don't want to kill the application, but you don't want to send it requests either. Kubernetes provides readiness probes to detect and mitigate these situations. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

Note:

Readiness probes runs on the container during its whole lifecycle.

Caution:

The readiness and liveness probes do not depend on each other to succeed. If you want to wait before executing a readiness probe, you should use `initialDelaySeconds` or a `startupProbe`.

Readiness probes are configured similarly to liveness probes. The only difference is that you use the `readinessProbe` field instead of the `livenessProbe` field.

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

Configuration for HTTP and TCP readiness probes also remains identical to liveness probes.

Readiness and liveness probes can be used in parallel for the same container. Using both can ensure that traffic does not reach a container that is not ready for it, and that containers are restarted when they fail.

Configure Probes

Probes have a number of fields that you can use to more precisely control the behavior of startup, liveness and readiness checks:

- `initialDelaySeconds`: Number of seconds after the container has started before startup, liveness or readiness probes are initiated. If a startup probe is defined, liveness and readiness probe delays do not begin until the startup probe has succeeded. In some older Kubernetes versions, the initialDelaySeconds might be ignored if periodSeconds was set to a value higher than initialDelaySeconds. However, in current versions, initialDelaySeconds is always honored and the probe will not start until after this initial delay. Defaults to 0 seconds. Minimum value is 0.
- `periodSeconds`: How often (in seconds) to perform the probe. Default to 10 seconds. The minimum value is 1. While a container is not Ready, the `ReadinessProbe` may be executed at times other than the configured `periodSeconds` interval. This is to make the Pod ready faster.
- `timeoutSeconds`: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
- `successThreshold`: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness and startup Probes. Minimum value is 1.
- `failureThreshold`: After a probe fails `failureThreshold` times in a row, Kubernetes considers that the overall check has failed: the container is *not* ready/healthy/live. Defaults to 3. Minimum value is 1. For the case of a startup or liveness probe, if at least `failureThreshold` probes have failed, Kubernetes treats the container as unhealthy and triggers a restart for that specific container. The kubelet honors the setting of `terminationGracePeriodSeconds` for that container. For a failed readiness probe, the kubelet continues running the container that failed checks, and also continues to run more probes; because the check failed, the kubelet sets the `Ready` condition on the Pod to `false`.
- `terminationGracePeriodSeconds`: configure a grace period for the kubelet to wait between triggering a shut down of the failed container, and then forcing the container runtime to stop that container. The default is to inherit the Pod-level value for `terminationGracePeriodSeconds` (30 seconds if not specified), and the minimum value is 1. See probe-level `terminationGracePeriodSeconds` for more detail.

Caution:

Incorrect implementation of readiness probes may result in an ever growing number of processes in the container, and resource starvation if this is left unchecked.

HTTP probes

HTTP probes have additional fields that can be set on `httpGet`:

- `host`: Host name to connect to, defaults to the pod IP. You probably want to set "Host" in `httpHeaders` instead.
- `scheme`: Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to "HTTP".
- `path`: Path to access on the HTTP server. Defaults to "/".
- `httpHeaders`: Custom headers to set in the request. HTTP allows repeated headers.
- `port`: Name or number of the port to access on the container. Number must be in the range 1 to 65535.

For an HTTP probe, the kubelet sends an HTTP request to the specified port and path to perform the check. The kubelet sends the probe to the Pod's IP address, unless the address is overridden by the optional `host` field in `httpGet`. If `scheme` field is set to `HTTPS`, the kubelet sends an HTTPS request skipping the certificate verification. In most scenarios, you do not want to set the `host` field. Here's one scenario where you would set it. Suppose the container listens on 127.0.0.1 and the Pod's `hostNetwork` field is true. Then `host`, under `httpGet`, should be set to 127.0.0.1. If your pod relies on virtual hosts, which is probably the more common case, you should not use `host`, but rather set the `Host` header in `httpHeaders`.

For an HTTP probe, the kubelet sends two request headers in addition to the mandatory `Host` header:

- `User-Agent`: The default value is `kube-probe/1.34`, where `1.34` is the version of the kubelet.
- `Accept`: The default value is `*/*`.

You can override the default headers by defining `httpHeaders` for the probe. For example:

```
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: application/json

startupProbe:  httpGet:    httpHeaders:      - name: User-Agent        value: MyUserAgent
```

You can also remove these two headers by defining them with an empty value.

```
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: ""

startupProbe:  httpGet:    httpHeaders:      - name: User-Agent        value: ""
```

Note:

When the kubelet probes a Pod using HTTP, it only follows redirects if the redirect is to the same host. If the kubelet receives 11 or more redirects during probing, the probe is considered successful and a related Event is created:

```
Events:
  Type     Reason        Age                     From               Message
  ----     ------        ----                    ----               -------
  Normal   Scheduled     29m                     default-scheduler  Successfully assigned default/httpbin-7b8bc9cb85-bjzwn to daoc
  Normal   Pulling       29m                     kubelet            Pulling image "docker.io/kennethreitz/httpbin"
  Normal   Pulled        24m                     kubelet            Successfully pulled image "docker.io/kennethreitz/httpbin" in
  Normal   Created       24m                     kubelet            Created container httpbin
  Normal   Started       24m                     kubelet            Started container httpbin
 Warning  ProbeWarning  4m11s (x1197 over 24m)  kubelet            Readiness probe warning: Probe terminated redirects
```

If the kubelet receives a redirect where the hostname is different from the request, the outcome of the probe is treated as successful and kubelet creates an event to report the redirect failure.

Caution:

When processing an **httpGet** probe, the kubelet stops reading the response body after 10KiB. The probe's success is determined solely by the response status code, which is found in the response headers.

If you probe an endpoint that returns a response body larger than **10KiB**, the kubelet will still mark the probe as successful based on the status code, but it will close the connection after reaching the 10KiB limit. This abrupt closure can cause **connection reset by peer** or **broken pipe errors** to appear in your application's logs, which can be difficult to distinguish from legitimate network issues.

For reliable `httpGet` probes, it is strongly recommended to use dedicated health check endpoints that return a minimal response body. If you must use an existing endpoint with a large payload, consider using an `exec` probe to perform a HEAD request instead.

TCP probes

For a TCP probe, the kubelet makes the probe connection at the node, not in the Pod, which means that you can not use a service name in the `host` parameter since the kubelet is unable to resolve it.

Probe-level `terminationGracePeriodSeconds`

FEATURE STATE: `Kubernetes v1.28 [stable]`

In 1.25 and above, users can specify a probe-level `terminationGracePeriodSeconds` as part of the probe specification. When both a pod- and probe-level `terminationGracePeriodSeconds` are set, the kubelet will use the probe-level value.

When setting the `terminationGracePeriodSeconds`, please note the following:

- The kubelet always honors the probe-level `terminationGracePeriodSeconds` field if it is present on a Pod.

- If you have existing Pods where the `terminationGracePeriodSeconds` field is set and you no longer wish to use per-probe termination grace periods, you must delete those existing Pods.

For example:

```
spec:
  terminationGracePeriodSeconds: 3600  # pod-level
  containers:
  - name: test
```

```
    image: ...

    ports:
    - name: liveness-port
      containerPort: 8080

    livenessProbe:
      httpGet:
        path: /healthz
        port: liveness-port
      failureThreshold: 1
      periodSeconds: 60
      # Override pod-level terminationGracePeriodSeconds #
      terminationGracePeriodSeconds: 60
```

Probe-level `terminationGracePeriodSeconds` cannot be set for readiness probes. It will be rejected by the API server.

What's next

- Learn more about Container Probes.

You can also read the API references for:

- Pod, and specifically:
  - container(s)
  - probe(s)

Use a User Namespace With a Pod

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

This page shows how to configure a user namespace for pods. This allows you to isolate the user running inside the container from the one in the host.

A process running as root in a container can run as a different (non-root) user in the host; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

You can use this feature to reduce the damage a compromised container can do to the host or other pods in the same node. There are several security vulnerabilities rated either **HIGH** or **CRITICAL** that were not exploitable when user namespaces is active. It is expected user namespace will mitigate some future vulnerabilities too.

Without using a user namespace a container running as root, in the case of a container breakout, has root privileges on the node. And if some capability were granted to the container, the capabilities are valid on the host too. None of this is true when user namespaces are used.

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 v1.25.

To check the version, enter `kubectl version`.

□ This item links to a third party project or product that is not part of Kubernetes itself. More information

- The node OS needs to be Linux
- You need to exec commands in the host
- You need to be able to exec into pods
- You need to enable the `UserNamespacesSupport` feature gate

Note:

The feature gate to enable user namespaces was previously named `UserNamespacesStatelessPodsSupport`, when only stateless pods were supported. Only Kubernetes v1.25 through to v1.27 recognise `UserNamespacesStatelessPodsSupport`.

The cluster that you're using **must** include at least one node that meets the requirements for using user namespaces with Pods.

If you have a mixture of nodes and only some of the nodes provide user namespace support for Pods, you also need to ensure that the user namespace Pods are scheduled to suitable nodes.

Run a Pod that uses a user namespace

A user namespace for a pod is enabled setting the `hostUsers` field of `.spec` to `false`. For example:

`pods/user-namespaces-stateless.yaml` Copy pods/user-namespaces-stateless.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: usernsspec:  hostUsers: false  containers:  - name: shell    command: ["sleep", "infinity"]    image: de
```

1. Create the pod on your cluster:

   ```
   kubectl apply -f https://k8s.io/examples/pods/user-namespaces-stateless.yaml
   ```

2. Exec into the pod and run `readlink /proc/self/ns/user`:

   ```
   kubectl exec -ti userns -- bash
   ```

Run this command:

```
readlink /proc/self/ns/user
```

The output is similar to:

```
user:[4026531837]
```

Also run:

```
cat /proc/self/uid_map
```

The output is similar to:

```
0  833617920      65536
```

Then, open a shell in the host and run the same commands.

The `readlink` command shows the user namespace the process is running in. It should be different when it is run on the host and inside the container.

The last number of the `uid_map` file inside the container must be 65536, on the host it must be a bigger number.

If you are running the kubelet inside a user namespace, you need to compare the output from running the command in the pod to the output of running in the host:

```
readlink /proc/$pid/ns/user
```

replacing `$pid` with the kubelet PID.

Managing Secrets using Kustomize

Creating Secret objects using kustomization.yaml file.

`kubectl` supports using the Kustomize object management tool to manage Secrets and ConfigMaps. You create a *resource generator* using Kustomize, which generates a Secret that you can apply to the API server using `kubectl`.

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

Create a Secret

You can generate a Secret by defining a `secretGenerator` in a `kustomization.yaml` file that references other existing files, `.env` files, or literal values. For example, the following instructions create a kustomization file for the username `admin` and the password `1f2d1e2e67df`.

Note:

The `stringData` field for a Secret does not work well with server-side apply.

Create the kustomization file

- Literals
- Files
- .env files

```
secretGenerator:- name: database-creds  literals:  - username=admin  - password=1f2d1e2e67df
```

1. Store the credentials in files. The filenames are the keys of the secret:

   ```
   echo -n 'admin' > ./username.txt
   echo -n '1f2d1e2e67df' > ./password.txt
   ```

The `-n` flag ensures that there's no newline character at the end of your files.

2. Create the `kustomization.yaml` file:

   ```
   secretGenerator:
   - name: database-creds  files:  - username.txt  - password.txt
   ```

You can also define the secretGenerator in the `kustomization.yaml` file by providing `.env` files. For example, the following `kustomization.yaml` file pulls in data from an `.env.secret` file:

```
secretGenerator:
- name: db-user-pass  envs:  - .env.secret
```

In all cases, you don't need to encode the values in base64. The name of the YAML file **must** be `kustomization.yaml` or `kustomization.yml`.

Apply the kustomization file

To create the Secret, apply the directory that contains the kustomization file:

```
kubectl apply -k <directory-path>
```

The output is similar to:

```
secret/database-creds-5hdh7hhgfk created
```

When a Secret is generated, the Secret name is created by hashing the Secret data and appending the hash value to the name. This ensures that a new Secret is generated each time the data is modified.

To verify that the Secret was created and to decode the Secret data,

```
kubectl get -k <directory-path> -o jsonpath='{.data}'
```

The output is similar to:

```
{ "password": "MWYyZDFlMmU2N2Rm", "username": "YWRtaW4=" }
```

```
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

The output is similar to:

```
1f2d1e2e67df
```

For more information, refer to Managing Secrets using kubectl and Declarative Management of Kubernetes Objects Using Kustomize.

Edit a Secret

1. In your `kustomization.yaml` file, modify the data, such as the `password`.
2. Apply the directory that contains the kustomization file:

   ```
   kubectl apply -k <directory-path>
   ```

   The output is similar to:

   ```
   secret/db-user-pass-6f24b56cc8 created
   ```

The edited Secret is created as a new `Secret` object, instead of updating the existing `Secret` object. You might need to update references to the Secret in your Pods.

Clean up

To delete a Secret, use `kubectl`:

```
kubectl delete secret db-user-pass
```

What's next

- Read more about the Secret concept
- Learn how to manage Secrets using kubectl
- Learn how to manage Secrets using config file

Assign Devices to Pods and Containers

Assign infrastructure resources to your Kubernetes workloads.

Set Up DRA in a Cluster

Allocate Devices to Workloads with DRA

Managing Secrets using Configuration File

Creating Secret objects using resource configuration file.

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

Create the Secret

You can define the `Secret` object in a manifest first, in JSON or YAML format, and then create that object. The Secret resource contains two maps: `data` and `stringData`. The `data` field is used to store arbitrary data, encoded using base64. The `stringData` field is provided for convenience, and it allows you to provide the same data as unencoded strings. The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`.

The following example stores two strings in a Secret using the `data` field.

1. Convert the strings to base64:

   ```
   echo -n 'admin' | base64
   echo -n '1f2d1e2e67df' | base64
   ```

   Note:

   The serialized JSON and YAML values of Secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/macOS, users should avoid using the `-b` option to split long lines. Conversely, Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if the `-w` option is not available.

   The output is similar to:

   ```
   YWRtaW4=
   MWYyZDFlMmU2N2Rm
   ```

2. Create the manifest:

   ```
   apiVersion: v1
   kind: Secretmetadata:  name: mysecrettype: Opaquedata:  username: YWRtaW4=  password: MWYyZDFlMmU2N2Rm
   ```

   Note that the name of a Secret object must be a valid DNS subdomain name.

3. Create the Secret using `kubectl apply`:

   ```
   kubectl apply -f ./secret.yaml
   ```

   The output is similar to:

   ```
   secret/mysecret created
   ```

To verify that the Secret was created and to decode the Secret data, refer to Managing Secrets using kubectl.

Specify unencoded data when creating a Secret

For certain scenarios, you may wish to use the `stringData` field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

For example, if your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "<user>"password: "<password>"
```

You could store this in a Secret using the following definition:

```
apiVersion: v1
kind: Secretmetadata:  name: mysecrettype: OpaquestringData:  config.yaml: |    apiUrl: "https://my.api.com/api/v1"    username: <
```

Note:

The `stringData` field for a Secret does not work well with server-side apply.

When you retrieve the Secret data, the command returns the encoded values, and not the plaintext values you provided in `stringData`.

For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
data:  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6IHt7dXNlcm5hbWV9fQpwYXNzd29yZDoge3twYXNzd29yZH19ki
```

Specify both `data` and `stringData`

If you specify a field in both `data` and `stringData`, the value from `stringData` is used.

For example, if you define the following Secret:

```
apiVersion: v1
kind: Secretmetadata:  name: mysecrettype: Opaquedata:  username: YWRtaW4=stringData:  username: administrator
```

Note:

The `stringData` field for a Secret does not work well with server-side apply.

The `Secret` object is created as follows:

```
apiVersion: v1
data:  username: YWRtaW5pc3RyYXRvcg==kind: Secretmetadata:  creationTimestamp: 2018-11-15T20:46:46Z  name: mysecret  namespace: de
```

`YWRtaW5pc3RyYXRvcg==` decodes to `administrator`.

Edit a Secret

To edit the data in the Secret you created using a manifest, modify the `data` or `stringData` field in your manifest and apply the file to your cluster. You can edit an existing `Secret` object unless it is immutable.

For example, if you want to change the password from the previous example to `birdsarentreal`, do the following:

1. Encode the new password string:

   ```
   echo -n 'birdsarentreal' | base64
   ```

   The output is similar to:

   ```
   YmlyZHNhcmVudHJlYWw=
   ```

2. Update the `data` field with your new password string:

   ```
   apiVersion: v1
   kind: Secretmetadata:  name: mysecrettype: Opaquedata:  username: YWRtaW4=  password: YmlyZHNhcmVudHJlYWw=
   ```

3. Apply the manifest to your cluster:

   ```
   kubectl apply -f ./secret.yaml
   ```

   The output is similar to:

   ```
   secret/mysecret configured
   ```

Kubernetes updates the existing `Secret` object. In detail, the `kubectl` tool notices that there is an existing `Secret` object with the same name. `kubectl` fetches the existing object, plans changes to it, and submits the changed `Secret` object to your cluster control plane.

If you specified `kubectl apply --server-side` instead, `kubectl` uses Server Side Apply instead.

Clean up

To delete the Secret you have created:

```
kubectl delete secret mysecret
```

What's next

- Read more about the Secret concept
- Learn how to manage Secrets using kubectl
- Learn how to manage Secrets using kustomize

Configure a Pod to Use a ConfigMap

Many applications rely on configuration which is used during either application initialization or runtime. Most times, there is a requirement to adjust values assigned to configuration parameters. ConfigMaps are a Kubernetes mechanism that let you inject configuration data into application pods.

The ConfigMap concept allow you to decouple configuration artifacts from image content to keep containerized applications portable. For example, you can download and run the same container image to spin up containers for the purposes of local development, system test, or running a live end-user workload.

This page provides a series of usage examples demonstrating how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

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 `wget` tool installed. If you have a different tool such as `curl`, and you do not have `wget`, you will need to adapt the step that downloads example data.

Create a ConfigMap

You can use either `kubectl create configmap` or a ConfigMap generator in `kustomization.yaml` to create a ConfigMap.

Create a ConfigMap using `kubectl create configmap`

Use the `kubectl create configmap` command to create ConfigMaps from directories, files, or literal values:

```
kubectl create configmap <map-name> <data-source>
```

where <map-name> is the name you want to assign to the ConfigMap and <data-source> is the directory, file, or literal value to draw the data from. The name of a ConfigMap object must be a valid DNS subdomain name.

When you are creating a ConfigMap based on a file, the key in the <data-source> defaults to the basename of the file, and the value defaults to the file content.

You can use `kubectl describe` or `kubectl get` to retrieve information about a ConfigMap.

Create a ConfigMap from a directory

You can use `kubectl create configmap` to create a ConfigMap from multiple files in the same directory. When you are creating a ConfigMap based on a directory, kubectl identifies files whose filename is a valid key in the directory and packages each of those files into the new ConfigMap. Any directory entries except regular files are ignored (for example: subdirectories, symlinks, devices, pipes, and more).

Note:

Each filename being used for ConfigMap creation must consist of only acceptable characters, which are: letters (`A` to `Z` and `a` to `z`), digits (`0` to `9`), '-', '_', or '.'. If you use `kubectl create configmap` with a directory where any of the file names contains an unacceptable character, the `kubectl` command may fail.

The `kubectl` command does not print an error when it encounters an invalid filename.

Create the local directory:

```
mkdir -p configure-pod-container/configmap/
```

Now, download the sample configuration and create the ConfigMap:

```
# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game.properties -O configure-pod-container/configmap/game.properties
wget https://kubernetes.io/examples/configmap/ui.properties -O configure-pod-container/configmap/ui.properties

# Create the ConfigMap
kubectl create configmap game-config --from-file=configure-pod-container/configmap/
```

The above command packages each file, in this case, `game.properties` and `ui.properties` in the `configure-pod-container/configmap/` directory into the game-config ConfigMap. You can display details of the ConfigMap using the following command:

```
kubectl describe configmaps game-config
```

The output is similar to this:

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

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
----
color.good=purple
color.bad=yellow
```

```
allow.textmode=true
how.nice.to.look=fairlyNice
```

The `game.properties` and `ui.properties` files in the `configure-pod-container/configmap/` directory are represented in the `data` section of the ConfigMap.

```
kubectl get configmaps game-config -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ConfigMapmetadata:  creationTimestamp: 2022-02-18T18:52:05Z  name: game-config  namespace: default  resourceVersion: "516"
```

Create ConfigMaps from files

You can use `kubectl create configmap` to create a ConfigMap from an individual file, or from multiple files.

For example,

```
kubectl create configmap game-config-2 --from-file=configure-pod-container/configmap/game.properties
```

would produce the following ConfigMap:

```
kubectl describe configmaps game-config-2
```

where the output is similar to this:

```
Name:         game-config-2
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

You can pass in the `--from-file` argument multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap game-config-2 --from-file=configure-pod-container/configmap/game.properties --from-file=configure-pod-con
```

You can display details of the `game-config-2` ConfigMap using the following command:

```
kubectl describe configmaps game-config-2
```

The output is similar to this:

```
Name:         game-config-2
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

Use the option `--from-env-file` to create a ConfigMap from an env-file, for example:

```
# Env-files contain a list of environment variables.
# These syntax rules apply:
#   Each line in an env file has to be in VAR=VAL format.
#   Lines beginning with # (i.e. comments) are ignored.
#   Blank lines are ignored.
#   There is no special handling of quotation marks (i.e. they will be part of the ConfigMap value)).

# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game-env-file.properties -O configure-pod-container/configmap/game-env-file.properti
wget https://kubernetes.io/examples/configmap/ui-env-file.properties -O configure-pod-container/configmap/ui-env-file.properties

# The env-file `game-env-file.properties` looks like below
cat configure-pod-container/configmap/game-env-file.properties
enemies=aliens
```

```
lives=3
allowed="true"

# This comment and the empty line above it are ignored
```

```
kubectl create configmap game-config-env-file \
       --from-env-file=configure-pod-container/configmap/game-env-file.properties
```

would produce a ConfigMap. View the ConfigMap:

```
kubectl get configmap game-config-env-file -o yaml
```

the output is similar to:

```
apiVersion: v1
kind: ConfigMapmetadata:  creationTimestamp: 2019-12-27T18:36:28Z  name: game-config-env-file  namespace: default  resourceVersion
```

Starting with Kubernetes v1.23, `kubectl` supports the `--from-env-file` argument to be specified multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap config-multi-env-files \
        --from-env-file=configure-pod-container/configmap/game-env-file.properties \        --from-env-file=configure-pod-containe
```

would produce the following ConfigMap:

```
kubectl get configmap config-multi-env-files -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMapmetadata:  creationTimestamp: 2019-12-27T18:38:34Z  name: config-multi-env-files  namespace: default  resourceVersi
```

Define the key to use when creating a ConfigMap from a file

You can define a key other than the file name to use in the `data` section of your ConfigMap when using the `--from-file` argument:

```
kubectl create configmap game-config-3 --from-file=<my-key-name>=<path-to-file>
```

where `<my-key-name>` is the key you want to use in the ConfigMap and `<path-to-file>` is the location of the data source file you want the key to represent.

For example:

```
kubectl create configmap game-config-3 --from-file=game-special-key=configure-pod-container/configmap/game.properties
```

would produce the following ConfigMap:

```
kubectl get configmaps game-config-3 -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMapmetadata:  creationTimestamp: 2022-02-18T18:54:22Z  name: game-config-3  namespace: default  resourceVersion: "530"
```

Create ConfigMaps from literal values

You can use `kubectl create configmap` with the `--from-literal` argument to define a literal value from the command line:

```
kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm
```

You can pass in multiple key-value pairs. Each pair provided on the command line is represented as a separate entry in the `data` section of the ConfigMap.

```
kubectl get configmaps special-config -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ConfigMapmetadata:  creationTimestamp: 2022-02-18T19:14:38Z  name: special-config  namespace: default  resourceVersion: "651
```

Create a ConfigMap from generator

You can also create a ConfigMap from generators and then apply it to create the object in the cluster's API server. You should specify the generators in a `kustomization.yaml` file within a directory.

Generate ConfigMaps from files

For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-4
  options:
    labels:
      game-config: config-4
  files:
  - configure-pod-container/configmap/game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object:

```
kubectl apply -k .
```

```
configmap/game-config-4-m9dm2f92bt created
```

You can check that the ConfigMap was created like this:

```
kubectl get configmap
```

```
NAME                       DATA   AGE
game-config-4-m9dm2f92bt   1      37s
```

and also:

```
kubectl describe configmaps/game-config-4-m9dm2f92bt
```

```
Name:         game-config-4-m9dm2f92bt
Namespace:    default
Labels:       game-config=config-4
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","data":{"game.properties":"enemies=aliens\nlives=3\nenemies.cheat=true\nenemies.cheat.level=noG

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
Events:  <none>
```

Notice that the generated ConfigMap name has a suffix appended by hashing the contents. This ensures that a new ConfigMap is generated each time the content is modified.

Define the key to use when generating a ConfigMap from a file

You can define a key other than the file name to use in the ConfigMap generator. For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties` with the key `game-special-key`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-5
  options:
    labels:
      game-config: config-5
  files:
  - game-special-key=configure-pod-container/configmap/game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
```

```
configmap/game-config-5-m67dt67794 created
```

Generate ConfigMaps from literals

This example shows you how to create a `ConfigMap` from two literal key/value pairs: `special.type=charm` and `special.how=very`, using Kustomize and kubectl. To achieve this, you can specify the `ConfigMap` generator. Create (or replace) `kustomization.yaml` so that it has the following contents:

```
---
# kustomization.yaml contents for creating a ConfigMap from literalsconfigMapGenerator:- name: special-config-2  literals:  - spec
```

Apply the kustomization directory to create the ConfigMap object:

```
kubectl apply -k .
```

```
configmap/special-config-2-c92b5mmcf2 created
```

Interim cleanup

Before proceeding, clean up some of the ConfigMaps you made:

```
kubectl delete configmap special-config
kubectl delete configmap env-config
kubectl delete configmap -l 'game-config in (config-4,config-5)'
```

Now that you have learned to define ConfigMaps, you can move on to the next section, and learn how to use these objects with Pods.

Define container environment variables using ConfigMap data

Define a container environment variable with data from a single ConfigMap

1. Define an environment variable as a key-value pair in a ConfigMap:

   ```
   kubectl create configmap special-config --from-literal=special.how=very
   ```

2. Assign the `special.how` value defined in the ConfigMap to the `SPECIAL_LEVEL_KEY` environment variable in the Pod specification.

   [pods/pod-single-configmap-env-variable.yaml]Copy pods/pod-single-configmap-env-variable.yaml to clipboard

   ```
   apiVersion: v1
   kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.27.
   ```

   Create the Pod:

   ```
   kubectl create -f https://kubernetes.io/examples/pods/pod-single-configmap-env-variable.yaml
   ```

   Now, the Pod's output includes environment variable `SPECIAL_LEVEL_KEY=very`.

Define container environment variables with data from multiple ConfigMaps

As with the previous example, create the ConfigMaps first. Here is the manifest you will use:

[configmap/configmaps.yaml]Copy configmap/configmaps.yaml to clipboard

```
apiVersion: v1
kind: ConfigMapmetadata:  name: special-config  namespace: defaultdata:  special.how: very---apiVersion: v1kind: ConfigMapmetadata
```

- Create the ConfigMap:

  ```
  kubectl create -f https://kubernetes.io/examples/configmap/configmaps.yaml
  ```

- Define the environment variables in the Pod specification.

  [pods/pod-multiple-configmap-env-variable.yaml]Copy pods/pod-multiple-configmap-env-variable.yaml to clipboard

  ```
  apiVersion: v1
  kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.27.
  ```

  Create the Pod:

  ```
  kubectl create -f https://kubernetes.io/examples/pods/pod-multiple-configmap-env-variable.yaml
  ```

  Now, the Pod's output includes environment variables `SPECIAL_LEVEL_KEY=very` and `LOG_LEVEL=INFO`.

  Once you're happy to move on, delete that Pod and ConfigMap:

  ```
  kubectl delete pod dapi-test-pod --now
  kubectl delete configmap special-config
  kubectl delete configmap env-config
  ```

Configure all key-value pairs in a ConfigMap as container environment variables

- Create a ConfigMap containing multiple key-value pairs.

  [configmap/configmap-multikeys.yaml]Copy configmap/configmap-multikeys.yaml to clipboard

  ```
  apiVersion: v1
  kind: ConfigMapmetadata:  name: special-config  namespace: defaultdata:  SPECIAL_LEVEL: very  SPECIAL_TYPE: charm
  ```

  Create the ConfigMap:

  ```
  kubectl create -f https://kubernetes.io/examples/configmap/configmap-multikeys.yaml
  ```

- Use `envFrom` to define all of the ConfigMap's data as container environment variables. The key from the ConfigMap becomes the environment variable name in the Pod.

  [pods/pod-configmap-envFrom.yaml]Copy pods/pod-configmap-envFrom.yaml to clipboard

  ```
  apiVersion: v1
  kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.27.
  ```

  Create the Pod:

  ```
  kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-envFrom.yaml
  ```

  Now, the Pod's output includes environment variables `SPECIAL_LEVEL=very` and `SPECIAL_TYPE=charm`.

  Once you're happy to move on, delete that Pod:

  ```
  kubectl delete pod dapi-test-pod --now
  ```

Use ConfigMap-defined environment variables in Pod commands

You can use ConfigMap-defined environment variables in the `command` and `args` of a container using the `$(VAR_NAME)` Kubernetes substitution syntax.

For example, the following Pod manifest:

pods/pod-configmap-env-var-valueFrom.yaml Copy pods/pod-configmap-env-var-valueFrom.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.27.2
```

Create that Pod, by running:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-env-var-valueFrom.yaml
```

That pod produces the following output from the `test-container` container:

```
kubectl logs dapi-test-pod
```

```
very charm
```

Once you're happy to move on, delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

Add ConfigMap data to a Volume

As explained in Create ConfigMaps from files, when you create a ConfigMap using `--from-file`, the filename becomes a key stored in the `data` section of the ConfigMap. The file contents become the key's value.

The examples in this section refer to a ConfigMap named `special-config`:

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

```
apiVersion: v1
kind: ConfigMapmetadata:  name: special-config  namespace: defaultdata:  SPECIAL_LEVEL: very  SPECIAL_TYPE: charm
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/configmap-multikeys.yaml
```

Populate a Volume with data stored in a ConfigMap

Add the ConfigMap name under the `volumes` section of the Pod specification. This adds the ConfigMap data to the directory specified as `volumeMounts.mountPath` (in this case, `/etc/config`). The `command` section lists directory files with names that match the keys in ConfigMap.

pods/pod-configmap-volume.yaml Copy pods/pod-configmap-volume.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.27.2
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume.yaml
```

When the pod runs, the command `ls /etc/config/` produces the output below:

```
SPECIAL_LEVEL
SPECIAL_TYPE
```

Text data is exposed as files using the UTF-8 character encoding. To use some other character encoding, use `binaryData` (see ConfigMap object for more details).

Note:

If there are any files in the `/etc/config` directory of that container image, the volume mount will make those files from the image inaccessible.

Once you're happy to move on, delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

Add ConfigMap data to a specific path in the Volume

Use the `path` field to specify the desired file path for specific ConfigMap items. In this case, the `SPECIAL_LEVEL` item will be mounted in the `config-volume` volume at `/etc/config/keys`.

pods/pod-configmap-volume-specific-key.yaml Copy pods/pod-configmap-volume-specific-key.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.27.2
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume-specific-key.yaml
```

When the pod runs, the command `cat /etc/config/keys` produces the output below:

```
very
```

Caution:

Like before, all previous files in the `/etc/config/` directory will be deleted.

Delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

Project keys to specific paths and file permissions

You can project keys to specific paths. Refer to the corresponding section in the Secrets guide for the syntax.
You can set POSIX permissions for keys. Refer to the corresponding section in the Secrets guide for the syntax.

Optional references

A ConfigMap reference may be marked *optional*. If the ConfigMap is non-existent, the mounted volume will be empty. If the ConfigMap exists, but the referenced key is non-existent, the path will be absent beneath the mount point. See Optional ConfigMaps for more details.

Mounted ConfigMaps are updated automatically

When a mounted ConfigMap is updated, the projected content is eventually updated too. This applies in the case where an optionally referenced ConfigMap comes into existence after a pod has started.

Kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, it uses its local TTL-based cache for getting the current value of the ConfigMap. As a result, 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 (1 minute by default) + TTL of ConfigMaps cache (1 minute by default) in kubelet. You can trigger an immediate refresh by updating one of the pod's annotations.

Note:

A container using a ConfigMap as a subPath volume will not receive ConfigMap updates.

Understanding ConfigMaps and Pods

The ConfigMap API resource stores configuration data as key-value pairs. The data can be consumed in pods or provide the configurations for system components such as controllers. ConfigMap is similar to Secrets, but provides a means of working with strings that don't contain sensitive information. Users and system components alike can store configuration data in ConfigMap.

Note:

ConfigMaps should reference properties files, not replace them. Think of the ConfigMap as representing something similar to the Linux `/etc` directory and its contents. For example, if you create a Kubernetes Volume from a ConfigMap, each data item in the ConfigMap is represented by an individual file in the volume.

The ConfigMap's `data` field contains the configuration data. As shown in the example below, this can be simple (like individual properties defined using `--from-literal`) or complex (like configuration files or JSON blobs defined using `--from-file`).

```
apiVersion: v1
kind: ConfigMapmetadata:  creationTimestamp: 2016-02-18T19:14:38Z  name: example-config  namespace: defaultdata:  # example of a s
```

When `kubectl` creates a ConfigMap from inputs that are not ASCII or UTF-8, the tool puts these into the `binaryData` field of the ConfigMap, and not in `data`. Both text and binary data sources can be combined in one ConfigMap.

If you want to view the `binaryData` keys (and their values) in a ConfigMap, you can run `kubectl get configmap -o jsonpath='{.binaryData}' <name>`.

Pods can load data from a ConfigMap that uses either `data` or `binaryData`.

Optional ConfigMaps

You can mark a reference to a ConfigMap as *optional* in a Pod specification. If the ConfigMap doesn't exist, the configuration for which it provides data in the Pod (for example: environment variable, mounted volume) will be empty. If the ConfigMap exists, but the referenced key is non-existent the data is also empty.

For example, the following Pod specification marks an environment variable from a ConfigMap as optional:

```
apiVersion: v1
kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: gcr.io/google_containers/busybox
```

If you run this pod, and there is no ConfigMap named `a-config`, the output is empty. If you run this pod, and there is a ConfigMap named `a-config` but that ConfigMap doesn't have a key named `akey`, the output is also empty. If you do set a value for `akey` in the `a-config` ConfigMap, this pod prints that value and then terminates.

You can also mark the volumes and files provided by a ConfigMap as optional. Kubernetes always creates the mount paths for the volume, even if the referenced ConfigMap or key doesn't exist. For example, the following Pod specification marks a volume that references a ConfigMap as optional:

```
apiVersion: v1
kind: Podmetadata:  name: dapi-test-podspec:  containers:    - name: test-container      image: gcr.io/google_containers/busybox
```

Restrictions

- You must create the `ConfigMap` object before you reference it in a Pod specification. Alternatively, mark the ConfigMap reference as `optional` in the Pod spec (see Optional ConfigMaps). If you reference a ConfigMap that doesn't exist and you don't mark the reference as `optional`, the Pod won't start. Similarly, references to keys that don't exist in the ConfigMap will also prevent the Pod from starting, unless you mark the key references as `optional`.
- If you use `envFrom` to define environment variables from ConfigMaps, keys that are considered invalid will be skipped. The pod will be allowed to start, but the invalid names will be recorded in the event log (`InvalidVariableNames`). The log message lists each skipped key. For example:

  ```
  kubectl get events
  ```

  The output is similar to this:

  ```
  LASTSEEN FIRSTSEEN COUNT NAME           KIND  SUBOBJECT  TYPE      REASON                             SOURCE                MES
  0s       0s        1     dapi-test-pod Pod              Warning   InvalidEnvironmentVariableNames   {kubelet, 127.0.0.1}  Key
  ```
- ConfigMaps reside in a specific Namespace. Pods can only refer to ConfigMaps that are in the same namespace as the Pod.
- You can't use ConfigMaps for static pods, because the kubelet does not support this.

Cleaning up

Delete the ConfigMaps and Pods that you made:

```
kubectl delete configmaps/game-config configmaps/game-config-2 configmaps/game-config-3 \
               configmaps/game-config-env-filekubectl delete pod dapi-test-pod --now# You might already have removed the next setk
```

Remove the `kustomization.yaml` file that you used to generate the ConfigMap:

```
rm kustomization.yaml
```

If you created a directory `configure-pod-container` and no longer need it, you should remove that too, or move it into the trash can / deleted files location.

```
rm -r configure-pod-container
```

What's next

- Follow a real world example of Configuring Redis using a ConfigMap.
- Follow an example of Updating configuration via a ConfigMap.

Attach Handlers to Container Lifecycle Events

This page shows how to attach handlers to Container lifecycle events. Kubernetes supports the postStart and preStop events. Kubernetes sends the postStart event immediately after a Container is started, and it sends the preStop event immediately before the Container is terminated. A Container may specify one handler per event.

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`.

Define postStart and preStop handlers

In this exercise, you create a Pod that has one Container. The Container has handlers for the postStart and preStop events.

Here is the configuration file for the Pod:

`pods/lifecycle-events.yaml` Copy pods/lifecycle-events.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: lifecycle-demospec:  containers:  - name: lifecycle-demo-container    image: nginx    lifecycle:      po
```

In the configuration file, you can see that the postStart command writes a `message` file to the Container's `/usr/share` directory. The preStop command shuts down nginx gracefully. This is helpful if the Container is being terminated because of a failure.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/lifecycle-events.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pod lifecycle-demo
```

Get a shell into the Container running in your Pod:

```
kubectl exec -it lifecycle-demo -- /bin/bash
```

In your shell, verify that the `postStart` handler created the `message` file:

```
root@lifecycle-demo:/# cat /usr/share/message
```

The output shows the text written by the postStart handler:

```
Hello from the postStart handler
```

Discussion

Kubernetes sends the postStart event immediately after the Container is created. There is no guarantee, however, that the postStart handler is called before the Container's entrypoint is called. The postStart handler runs asynchronously relative to the Container's code, but Kubernetes' management of the container blocks until the postStart handler completes. The Container's status is not set to RUNNING until the postStart handler completes.

Kubernetes sends the preStop event immediately before the Container is terminated. Kubernetes' management of the Container blocks until the preStop handler completes, unless the Pod's grace period expires. For more details, see Pod Lifecycle.

Note:

Kubernetes only sends the preStop event when a Pod or a container in the Pod is *terminated*. This means that the preStop hook is not invoked when the Pod is *completed*. About this limitation, please see Container hooks for the detail.

What's next

- Learn more about Container lifecycle hooks.
- Learn more about the lifecycle of a Pod.

Reference

- Lifecycle
- Container
- See `terminationGracePeriodSeconds` in PodSpec

Configure Pods and Containers

Perform common configuration tasks for Pods and containers.

Assign Memory Resources to Containers and Pods

Assign CPU Resources to Containers and Pods

Assign Devices to Pods and Containers

Assign infrastructure resources to your Kubernetes workloads.

Assign Pod-level CPU and memory resources

Configure GMSA for Windows Pods and containers

Resize CPU and Memory Resources assigned to Containers

Configure RunAsUserName for Windows pods and containers

Create a Windows HostProcess Pod

Configure Quality of Service for Pods

Assign Extended Resources to a Container

Configure a Pod to Use a Volume for Storage

Configure a Pod to Use a PersistentVolume for Storage

Configure a Pod to Use a Projected Volume for Storage

Configure a Security Context for a Pod or Container

Configure Service Accounts for Pods

Pull an Image from a Private Registry

Configure Liveness, Readiness and Startup Probes

Assign Memory Resources to Containers and Pods

This page shows how to assign a memory *request* and a memory *limit* to a Container. A Container is guaranteed to have as much memory as it requests, but is not allowed to use more memory than its limit.

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`.

Each node in your cluster must have at least 300 MiB of memory.

A few of the steps on this page require you to run the metrics-server service in your cluster. If you have the metrics-server running, you can skip those steps.

If you are running Minikube, run the following command to enable the metrics-server:

```
minikube addons enable metrics-server
```

To see whether the metrics-server is running, or another provider of the resource metrics API (`metrics.k8s.io`), run the following command:

```
kubectl get apiservices
```

If the resource metrics API is available, the output includes a reference to `metrics.k8s.io`.

```
NAME
v1beta1.metrics.k8s.io
```

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace mem-example
```

Specify a memory request and a memory limit

To specify a memory request for a Container, include the `resources:requests` field in the Container's resource manifest. To specify a memory limit, include `resources:limits`.

In this exercise, you create a Pod that has one Container. The Container has a memory request of 100 MiB and a memory limit of 200 MiB. Here's the configuration file for the Pod:

`pods/resource/memory-request-limit.yaml` Copy pods/resource/memory-request-limit.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: memory-demo  namespace: mem-examplespec:  containers:  - name: memory-demo-ctr    image: polinux/stress
```

The `args` section in the configuration file provides arguments for the Container when it starts. The `"--vm-bytes", "150M"` arguments tell the Container to attempt to allocate 150 MiB of memory.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit.yaml --namespace=mem-example
```

Verify that the Pod Container is running:

```
kubectl get pod memory-demo --namespace=mem-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo --output=yaml --namespace=mem-example
```

The output shows that the one Container in the Pod has a memory request of 100 MiB and a memory limit of 200 MiB.

```
...
resources:  requests:    memory: 100Mi  limits:    memory: 200Mi...
```

Run `kubectl top` to fetch the metrics for the pod:

```
kubectl top pod memory-demo --namespace=mem-example
```

The output shows that the Pod is using about 162,900,000 bytes of memory, which is about 150 MiB. This is greater than the Pod's 100 MiB request, but within the Pod's 200 MiB limit.

```
NAME                        CPU(cores)   MEMORY(bytes)
memory-demo                 <something>  162856960
```

Delete your Pod:

```
kubectl delete pod memory-demo --namespace=mem-example
```

Exceed a Container's memory limit

A Container can exceed its memory request if the Node has memory available. But a Container is not allowed to use more than its memory limit. If a Container allocates more memory than its limit, the Container becomes a candidate for termination. If the Container continues to consume memory beyond its limit, the Container is terminated. If a terminated Container can be restarted, the kubelet restarts it, as with any other type of runtime failure.

In this exercise, you create a Pod that attempts to allocate more memory than its limit. Here is the configuration file for a Pod that has one Container with a memory request of 50 MiB and a memory limit of 100 MiB:

pods/resource/memory-request-limit-2.yaml Copy pods/resource/memory-request-limit-2.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: memory-demo-2  namespace: mem-examplespec:  containers:  - name: memory-demo-2-ctr    image: polinux/str
```

In the `args` section of the configuration file, you can see that the Container will attempt to allocate 250 MiB of memory, which is well above the 100 MiB limit.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit-2.yaml --namespace=mem-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

At this point, the Container might be running or killed. Repeat the preceding command until the Container is killed:

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          24s
```

Get a more detailed view of the Container status:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-example
```

The output shows that the Container was killed because it is out of memory (OOM):

```
lastState:
   terminated:
     containerID: 65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917c23b10f
     exitCode: 137
     finishedAt: 2017-06-20T20:52:19Z
     reason: OOMKilled
     startedAt: null
```

The Container in this exercise can be restarted, so the kubelet restarts it. Repeat this command several times to see that the Container is repeatedly killed and restarted:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container is killed, restarted, killed again, restarted again, and so on:

```
kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          37s
```

```
kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-2   1/1       Running   2          40s
```

View detailed information about the Pod history:

```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container starts and fails repeatedly:

```
... Normal  Created   Created container with id 66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511
... Warning BackOff   Back-off restarting failed container
```

View detailed information about your cluster's Nodes:

```
kubectl describe nodes
```

The output includes a record of the Container being killed because of an out-of-memory condition:

```
Warning OOMKilling Memory cgroup out of memory: Kill process 4481 (stress) score 1994 or sacrifice child
```

Delete your Pod:

```
kubectl delete pod memory-demo-2 --namespace=mem-example
```

Specify a memory request that is too big for your Nodes

Memory requests and limits are associated with Containers, but it is useful to think of a Pod as having a memory request and limit. The memory request for the Pod is the sum of the memory requests for all the Containers in the Pod. Likewise, the memory limit for the Pod is the sum of the limits of all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough available memory to satisfy the Pod's memory request.

In this exercise, you create a Pod that has a memory request so big that it exceeds the capacity of any Node in your cluster. Here is the configuration file for a Pod that has one Container with a request for 1000 GiB of memory, which likely exceeds the capacity of any Node in your cluster.

pods/resource/memory-request-limit-3.yaml Copy pods/resource/memory-request-limit-3.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: memory-demo-3  namespace: mem-examplespec:  containers:  - name: memory-demo-3-ctr    image: polinux/str
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit-3.yaml --namespace=mem-example
```

View the Pod status:

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

The output shows that the Pod status is PENDING. That is, the Pod is not scheduled to run on any Node, and it will remain in the PENDING state indefinitely:

```
kubectl get pod memory-demo-3 --namespace=mem-example
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-3   0/1       Pending   0          25s
```

View detailed information about the Pod, including events:

```
kubectl describe pod memory-demo-3 --namespace=mem-example
```

The output shows that the Container cannot be scheduled because of insufficient memory on the Nodes:

```
Events:
  ...  Reason            Message
       ------            -------
  ...  FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient memory (3).
```

Memory units

The memory resource is measured in bytes. You can express memory as a plain integer or a fixed-point integer with one of these suffixes: E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent approximately the same value:

```
128974848, 129e6, 129M, 123Mi
```

Delete your Pod:

```
kubectl delete pod memory-demo-3 --namespace=mem-example
```

If you do not specify a memory limit

If you do not specify a memory limit for a Container, one of the following situations applies:

- The Container has no upper bound on the amount of memory it uses. The Container could use all of the memory available on the Node where it is running which in turn could invoke the OOM Killer. Further, in case of an OOM Kill, a container with no resource limits will have a greater chance of being killed.

- The Container is running in a namespace that has a default memory limit, and the Container is automatically assigned the default limit. Cluster administrators can use a LimitRange to specify a default value for the memory limit.

Motivation for memory requests and limits

By configuring memory requests and limits for the Containers that run in your cluster, you can make efficient use of the memory resources available on your cluster's Nodes. By keeping a Pod's memory request low, you give the Pod a good chance of being scheduled. By having a memory limit that is greater than the memory request, you accomplish two things:

- The Pod can have bursts of activity where it makes use of memory that happens to be available.
- The amount of memory a Pod can use during a burst is limited to some reasonable amount.

Clean up

Delete your namespace. This deletes all the Pods that you created for this task:

```
kubectl delete namespace mem-example
```

What's next

For app developers

- Assign CPU Resources to Containers and Pods

- Assign Pod-level CPU and memory resources

- Configure Quality of Service for Pods

- Resize CPU and Memory Resources assigned to Containers

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace

- Configure Default CPU Requests and Limits for a Namespace

- Configure Minimum and Maximum Memory Constraints for a Namespace

- Configure Minimum and Maximum CPU Constraints for a Namespace

- Configure Memory and CPU Quotas for a Namespace

- Configure a Pod Quota for a Namespace

- Configure Quotas for API Objects

- Resize CPU and Memory Resources assigned to Containers

Managing Secrets using kubectl

Creating Secret objects using kubectl command line.

This page shows you how to create, edit, manage, and delete Kubernetes Secrets using the `kubectl` command-line tool.

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

Create a Secret

A `Secret` object stores sensitive data such as credentials used by Pods to access services. For example, you might need a Secret to store the username and password needed to access a database.

You can create the Secret by passing the raw data in the command, or by storing the credentials in files that you pass in the command. The following commands create a Secret that stores the username `admin` and the password `S!B\*d$zDsb=`.

Use raw data

Run the following command:

```
kubectl create secret generic db-user-pass \
    --from-literal=username=admin \    --from-literal=password='S!B\*d$zDsb='
```

You must use single quotes `''` to escape special characters such as `$`, `\`, `*`, `=`, and `!` in your strings. If you don't, your shell will interpret these characters.

Note:

The `stringData` field for a Secret does not work well with server-side apply.

Use source files

1. Store the credentials in files:

   ```
   echo -n 'admin' > ./username.txt
   echo -n 'S!B\*d$zDsb=' > ./password.txt
   ```

   The `-n` flag ensures that the generated files do not have an extra newline character at the end of the text. This is important because when `kubectl` reads a file and encodes the content into a base64 string, the extra newline character gets encoded too. You do not need to escape special characters in strings that you include in a file.

2. Pass the file paths in the `kubectl` command:

   ```
   kubectl create secret generic db-user-pass \
       --from-file=./username.txt \    --from-file=./password.txt
   ```

   The default key name is the file name. You can optionally set the key name using `--from-file=[key=]source`. For example:

   ```
   kubectl create secret generic db-user-pass \
       --from-file=username=./username.txt \    --from-file=password=./password.txt
   ```

With either method, the output is similar to:

```
secret/db-user-pass created
```

Verify the Secret

Check that the Secret was created:

```
kubectl get secrets
```

The output is similar to:

```
NAME              TYPE       DATA      AGE
db-user-pass      Opaque     2         51s
```

View the details of the Secret:

```
kubectl describe secret db-user-pass
```

The output is similar to:

```
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password:    12 bytes
username:    5 bytes
```

The commands `kubectl get` and `kubectl describe` avoid showing the contents of a `Secret` by default. This is to protect the `Secret` from being exposed accidentally, or from being stored in a terminal log.

Decode the Secret

1. View the contents of the Secret you created:

   ```
   kubectl get secret db-user-pass -o jsonpath='{.data}'
   ```

   The output is similar to:

   ```
   { "password": "UyFCXCpkJHpEc2I9", "username": "YWRtaW4=" }
   ```

2. Decode the `password` data:

   ```
   echo 'UyFCXCpkJHpEc2I9' | base64 --decode
   ```

The output is similar to:

```
S!B\*d$zDsb=
```

Caution:

This is an example for documentation purposes. In practice, this method could cause the command with the encoded data to be stored in your shell history. Anyone with access to your computer could find the command and decode the secret. A better approach is to combine the view and decode commands.

```
kubectl get secret db-user-pass -o jsonpath='{.data.password}' | base64 --decode
```

Edit a Secret

You can edit an existing `Secret` object unless it is immutable. To edit a Secret, run the following command:

```
kubectl edit secrets <secret-name>
```

This opens your default editor and allows you to update the base64 encoded Secret values in the `data` field, such as in the following example:

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file, it will be# reopened with the relevant failure.
```

Clean up

To delete a Secret, run the following command:

```
kubectl delete secret db-user-pass
```

What's next

- Read more about the Secret concept
- Learn how to manage Secrets using config file
- Learn how to manage Secrets using kustomize