Configure Pods and Containers

Perform common configuration tasks for Pods and containers.

- 1: Assign Memory Resources to Containers and Pods
- 2: Assign CPU Resources to Containers and Pods
- 3: Configure GMSA for Windows Pods and containers
- 4: Configure RunAsUserName for Windows pods and containers
- 5: Configure Quality of Service for Pods
- 6: Assign Extended Resources to a Container
- 7: Configure a Pod to Use a Volume for Storage
- 8: Configure a Pod to Use a PersistentVolume for Storage
- 9: Configure a Pod to Use a Projected Volume for Storage
- 10: Configure a Security Context for a Pod or Container
- 11: Configure Service Accounts for Pods
- 12: Pull an Image from a Private Registry
- 13: Configure Liveness, Readiness and Startup Probes
- 14: Assign Pods to Nodes
- 15: Assign Pods to Nodes using Node Affinity
- 16: Configure Pod Initialization
- 17: Attach Handlers to Container Lifecycle Events
- 18: Configure a Pod to Use a ConfigMap
- 19: Share Process Namespace between Containers in a Pod
- 20: Create static Pods
- 21: Translate a Docker Compose File to Kubernetes Resources

1 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

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 --r
```

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:
  limits:
    memory: 200Mi
  requests:
    memory: 100Mi
...
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
```

```yaml
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-2-ctr
    image: polinux/stress
    resources:
      requests:
        memory: "50Mi"
      limits:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

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 -
```

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: docker://65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917
     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 66a3a20aa7980e61be4922780bf9d24d1a1d
... 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 199
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-3
```

```
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-3-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "1000Gi"
      requests:
        memory: "1000Gi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit-3.yaml -
```

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 pred
```

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

- Configure Quality of Service for 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

2 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: cpu-example
spec:
  containers:
  - name: cpu-demo-ctr
    image: vish/stress
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
    args:
    - -cpus
    - "2"
```

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 --name
```

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.

```yaml
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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
  namespace: cpu-example
spec:
  containers:
```

```
  - name: cpu-demo-ctr-2
    image: vish/stress
    resources:
      limits:
        cpu: "100"
      requests:
        cpu: "100"
    args:
    - -cpus
    - "2"
```

Create the Pod:

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

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 predi
```

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
- Configure Quality of Service for 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

3 - 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. As of v1.16, the Docker runtime supports GMSA for Windows workloads.

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`:

```yaml
apiVersion: windows.k8s.io/v1alpha1
kind: GMSACredentialSpec
metadata:
  name: gmsa-WebApp1  #This is an arbitrary name but it will be used as a reference
credspec:
  ActiveDirectoryConfig:
    GroupManagedServiceAccounts:
    - Name: WebApp1   #Username of the GMSA account
      Scope: CONTOSO  #NETBIOS Domain Name
    - Name: WebApp1   #Username of the GMSA account
      Scope: contoso.com #DNS Domain Name
  CmsPlugins:
  - ActiveDirectory
  DomainJoinConfig:
    DnsName: contoso.com  #DNS Domain Name
    DnsTreeName: contoso.com #DNS Domain Name Root
    Guid: 244818ae-87ac-4fcd-92ec-e79e5252348a  #GUID
    MachineAccountName: WebApp1 #Username of the GMSA account
    NetBiosName: CONTOSO  #NETBIOS Domain Name
    Sid: S-1-5-21-2126449477-2524075714-3094792973 #SID of GMSA
```

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/v1
kind: ClusterRole
metadata:
  name: webapp1-role
rules:
- apiGroups: ["windows.k8s.io"]
  resources: ["gmsacredentialspecs"]
  verbs: ["use"]
  resourceNames: ["gmsa-WebApp1"]
```

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: RoleBinding
metadata:
  name: allow-default-svc-account-read-on-gmsa-WebApp1
  namespace: default
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: ClusterRole
  name: webapp1-role
  apiGroup: rbac.authorization.k8s.io
```

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: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
```

```yaml
      labels:
        run: with-creds
    spec:
      securityContext:
        windowsOptions:
          gmsaCredentialSpecName: gmsa-webapp1
      containers:
      - image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
        imagePullPolicy: Always
        name: iis
      nodeSelector:
        kubernetes.io/os: windows
```

Individual containers in a Pod spec can also specify the desired GMSA credspec using a per-container `securityContext.windowsOptions.gmsaCredentialSpecName` field. For example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      containers:
      - image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
        imagePullPolicy: Always
        name: iis
        securityContext:
          windowsOptions:
            gmsaCredentialSpecName: gmsa-Webapp1
      nodeSelector:
        kubernetes.io/os: windows
```

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.

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

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\> nltest.exe /parentdomain
Getting parent domain failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
PS C:\>
```

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.

```
PS C:\> nltest.exe /query
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 `nltest.exe /sc_reset:domain.example` command.

```
PS C:\> nltest /sc_reset:domain.example
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
PS C:\>
```

If the above command 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 netl
        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.

GMSA limitations

When using the ContainerD runtime for Windows accessing restricted network shares via the GMSA domain identity fails. The container will receive the identity of and calls from `nltest.exe /query` will work. It is recommended to use the Docker EE runtime if access to network shares is required. The Windows Server team is working on resolving the issue in the Windows Kernel and will release a patch to resolve this issue in the future. Look for updates on the Microsoft Windows Containers issue tracker.

4 - 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
  nodeSelector:
    kubernetes.io/os: windows
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
    securityContext:
        windowsOptions:
            runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

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

5 - 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 scheduling and evicting 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

QoS classes

When Kubernetes creates a Pod it assigns one of these QoS classes to the Pod:

- Guaranteed
- Burstable
- BestEffort

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, including init containers, in the Pod must have a memory limit and a memory request, and they must be the same.
- Every Container, including init containers, in the Pod must have a CPU limit and a CPU request, and they must be the same.

Here is the configuration file 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

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
```

```yaml
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod.yaml --namespace=qos-examp
```

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.

```yaml
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.

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.

Here is the configuration file 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-2
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-2.yaml --namespace=qos-exa
```

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.

```yaml
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: qos-demo-2-ctr
    resources:
      limits:
        memory: 200Mi
      requests:
        memory: 100Mi
  ...
status:
  qosClass: Burstable
```

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 the configuration file for a Pod that has one Container. The Container has no memory or CPU limits or requests:

pods/qos/qos-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-3
  namespace: qos-example
spec:
  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-exa
```

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
```

Delete your Pod:

```
kubectl delete pod qos-demo-3 --namespace=qos-example
```

Create a Pod that has two Containers

Here is the configuration file 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-4
  namespace: qos-example
spec:
  containers:

  - name: qos-demo-4-ctr-1
    image: nginx
    resources:
      requests:
        memory: "200Mi"

  - name: qos-demo-4-ctr-2
    image: redis
```

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-exa
```

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:

```yaml
spec:
  containers:
    ...
    name: qos-demo-4-ctr-1
    resources:
      requests:
        memory: 200Mi
    ...
    name: qos-demo-4-ctr-2
    resources: {}
    ...
status:
  qosClass: Burstable
```

Delete your Pod:

```
kubectl delete pod qos-demo-4 --namespace=qos-example
```

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

6 - Assign Extended Resources to a Container

FEATURE STATE: `Kubernetes v1.21 [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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo
spec:
  containers:
  - name: extended-resource-demo-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 3
      limits:
        example.com/dongle: 3
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo-2
spec:
  containers:
  - name: extended-resource-demo-2-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 2
      limits:
        example.com/dongle: 2
```

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 an
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

7 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

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 *
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.

8 - 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

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

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.

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     STOR
task-pv-volume   10Gi       RWO           Retain          Available             manu
```

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 least one Node.

Here is the configuration file for the PersistentVolumeClaim:

pods/storage/pv-claim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

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
task-pv-volume   10Gi       RWO           Retain          Bound     default/task-pv-
```

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   A
task-pv-claim   Bound     task-pv-volume   10Gi       RWO           manual         3
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

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, the PersistentVolumeClaim and the PersistentVolume:

```
kubectl delete pod task-pv-pod
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/index.html
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: PersistentVolume
metadata:
  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

9 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: user
      - secret:
          name: pass
```

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.

10 - 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 true always when the container is: 1) run as Privileged OR 2) 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.

For more information about security mechanisms in Linux, see Overview of Linux Kernel Security 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
```

```
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

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.

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

You will see that gid is 3000 which is same as `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 root(0) group and that have the required group permissions for root(0) group.

Exit your shell:

```
exit
```

Configure volume permission and ownership change policy for Pods

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

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 permission and 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"
```

This is an alpha feature. To use it, enable the feature gate `ConfigurableFSGroupPolicy` for the kube-api-server, the kube-controller-manager, and for the kubelet.

> **Note:** This field has no effect on ephemeral volume types such as `secret`, `configMap`, and `emptydir`.

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

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 ser
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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
  - name: sec-ctx-3
    image: gcr.io/google-samples/node-hello:1.0
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

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 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
```

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.

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
- Tuning Docker with the newest security enhancements
- Security Contexts design document
- Ownership Management design document
- Pod Security Policies
- AllowPrivilegeEscalation design document

11 - Configure Service Accounts for Pods

A service account provides an identity for processes that run in a Pod.

> **Note:** This document is a user introduction to Service Accounts and describes how service accounts behave in a cluster set up as recommended by the Kubernetes project. Your cluster administrator may have customized the behavior in your cluster, in which case this documentation may not apply.

When you (a human) access the cluster (for example, using `kubectl`), you are authenticated by the apiserver as a particular User Account (currently this is usually `admin`, unless your cluster administrator has customized your cluster). Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (for example, `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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

Use the Default Service Account to access the API server.

When you create a pod, if you do not specify a service account, it is automatically assigned the `default` service account in the same namespace. If you get the raw json or yaml for a pod you have created (for example, `kubectl get pods/<podname> -o yaml`), you can see the `spec.serviceAccountName` field has been automatically set.

You can access the API from inside a pod using automatically mounted service account credentials, as described in Accessing the Cluster. The API permissions of the service account depend on the authorization plugin and policy in use.

In version 1.6+, you can opt out of automounting API credentials for a service account by setting `automountServiceAccountToken: false` on the service account:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

In version 1.6+, you can also opt out of automounting API credentials for a particular pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
```

```
  automountServiceAccountToken: false
  ...
```

The pod spec takes precedence over the service account if both specify a `automountServiceAccountToken` value.

Use Multiple Service Accounts.

Every namespace has a default service account resource called `default`. You can list this and any other serviceAccount resources in the namespace with this command:

```
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: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

then you will see that a token has automatically been created and is referenced by the service account.

You may 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 service account you wish to use.

The service account has to exist at the time the pod is created, or it will be rejected.

You cannot update the service account of an already created pod.

You can clean up the service account from this example like this:

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

Manually create a service account API token.

Suppose we have an existing service account named "build-robot" as mentioned above, and we create a new secret manually.

```
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
```

Now you can confirm that the newly built secret is populated with an API token for the "build-robot" service account.

Any tokens for non-existent service accounts will be cleaned up by the token controller.

```
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-002dc52800

Type:   kubernetes.io/service-account-token

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

Note: The content of `token` is elided here.

Add ImagePullSecrets to a service account

Create an imagePullSecret

- Create an imagePullSecret, as described in [Specifying ImagePullSecrets on a Pod](#).

```
kubectl create secret docker-registry myregistrykey --docker-server=DUMMY_SERVE
        --docker-username=DUMMY_USERNAME --docker-password=DUMMY_DOCKER_PASSWOR
        --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": "myregistryk
```

You can instead use `kubectl edit`, or manually edit the YAML manifests as shown below:

```
kubectl get serviceaccounts default -o yaml > ./sa.yaml
```

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

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
```

Using your editor of choice (for example `vi`), open the `sa.yaml` file, delete line with key `resourceVersion`, add lines with `imagePullSecrets:` and save.

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

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
```

Finally replace the serviceaccount with the new updated `sa.yaml` file

```
kubectl replace serviceaccount default -f ./sa.yaml
```

Verify imagePullSecrets was added to pod spec

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

The output is:

```
myregistrykey
```

Service Account 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`
> - `--service-account-key-file`
> - `--service-account-signing-key-file`
> - `--api-audiences`

The kubelet can also project a service account 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 service account token. The service account token will also become invalid against the API when the Pod or the ServiceAccount is deleted.

This behavior is configured on a PodSpec using a ProjectedVolume type called ServiceAccountToken. To provide a pod with a token with an audience of "vault" and a validity duration of two hours, you would configure the following in your PodSpec:

pods/pod-projected-svc-token.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: vault-token
  serviceAccountName: build-robot
  volumes:
  - name: vault-token
    projected:
      sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 7200
          audience: vault
```

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 rotates the token if it is older than 80% of its total TTL, or if the token is older than 24 hours.

The application is responsible for reloading the token when it rotates. Periodic reloading (e.g. once every 5 minutes) is sufficient for most use cases.

Service Account Issuer Discovery

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

The Service Account Issuer Discovery feature is enabled when the Service Account Token Projection feature is enabled, as described above.

> **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, the `ServiceAccountIssuerDiscovery` endpoints will not be registered, even if the feature is enabled.

The Service Account Issuer Discovery feature enables federation of Kubernetes service account tokens issued by a cluster (the *identity provider*) with external systems (*relying parties*).

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

Clusters include a default RBAC ClusterRole called `system:service-account-issuer-discovery`. No role bindings are provided by default. Administrators may, for example, choose whether to bind the role to `system:authenticated` or `system:unauthenticated` depending on their security requirements and which external systems they intend to federate with.

> **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.

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 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:

- Cluster Admin Guide to Service Accounts
- Service Account Signing Key Retrieval KEP
- OIDC Discovery Spec

12 - 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 Docker registry or repository.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - Katacoda
  - Play with Kubernetes

  To check the version, enter `kubectl version`.
- To do this exercise, you need a Docker ID and password.

Log in to Docker

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

```
docker login
```

When prompted, enter your Docker username and password.

The login process creates or updates a `config.json` file that holds an authorization token.

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.

Create a Secret based on existing Docker credentials

A Kubernetes cluster uses the Secret of `docker-registry` 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 file and paste that string, unbroken as the value for field `data[".dockerconfigjson"]`
- set `type` to `kubernetes.io/dockerconfigjson`

Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
  namespace: awesomeapps
data:
  .dockerconfigjson: UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFhYWFhYWFhYWFhYWFhYWFhYWFhYWF
type: kubernetes.io/dockerconfigjson
```

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>
```

where:

- `<your-registry-server>` is your Private Docker Registry FQDN. Use `https://index.docker.io/v2/` 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:

```yaml
apiVersion: v1
kind: Secret
metadata:
  ...
  name: regcred
  ...
data:
  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=
type: kubernetes.io/dockerconfigjson
```

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 -
```

The output is similar to this:

```json
{"auths":{"your.private.registry.example.com":{"username":"janedoe","password":"xxxx
```

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 configuration file for a Pod that needs access to your Docker credentials in `regcred`:

pods/private-reg-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
```

```yaml
  - name: private-reg-container
    image: <your-private-image>
  imagePullSecrets:
  - name: regcred
```

Download the above file:

```
wget -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
```

What's next

- Learn more about Secrets.
- 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 Secret.
- See the `imagePullSecrets` field of PodSpec.

13 - Configure Liveness, Readiness and Startup Probes

This page shows how to configure liveness, readiness and startup probes for containers.

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.

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, it disables liveness and readiness checks 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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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 `k8s.gcr.io/busybox` image. Here is the configuration file for the Pod:

pods/probe/exec-liveness.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
```

```
        initialDelaySeconds: 5
        periodSeconds: 5
```

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 -rf /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:

```
FirstSeen    LastSeen    Count   From            SubobjectPath           Type
--------- --------    -----   ----            -------------           --------    --
24s       24s     1   {default-scheduler }                    Normal      Scheduled
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal      Pu
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal      Pu
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal      Cr
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal      St
```

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 containers have been killed and recreated.

```
FirstSeen LastSeen    Count   From            SubobjectPath           Type        Re
--------- --------    -----   ----            -------------           --------    --
37s       37s      1  {default-scheduler }                  Normal      Scheduled
36s       36s      1  {kubelet worker0}   spec.containers{liveness}   Normal      Pu
36s       36s      1  {kubelet worker0}   spec.containers{liveness}   Normal      Pu
36s       36s      1  {kubelet worker0}   spec.containers{liveness}   Normal      Cr
36s       36s      1  {kubelet worker0}   spec.containers{liveness}   Normal      St
2s        2s       1  {kubelet worker0}   spec.containers{liveness}   Warning     Un
```

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:

```
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 `k8s.gcr.io/liveness` image.

pods/probe/http-liveness.yaml

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

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 prior to v1.13 (including v1.13), if the environment variable `http_proxy` (or `HTTP_PROXY`) is set on the node where a Pod is running, the HTTP liveness probe uses that proxy. 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
```

```
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

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 send the first readiness probe 5 seconds after the container starts. This will attempt to connect to the `goproxy` container on port 8080. If the probe succeeds, the Pod will be marked as ready. The kubelet will continue to run this check every 10 seconds.

In addition to the readiness probe, this configuration includes a liveness probe. The kubelet will run the first liveness probe 15 seconds after the container starts. Similar to the readiness probe, this will attempt to connect to the `goproxy` container on port 8080. If the liveness probe fails, the container will be restarted.

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
```

Use a named port

You can use a named [ContainerPort](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#containerport-v1-core) for HTTP or TCP liveness checks:

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
```

Protect slow starting containers with startup probes

Sometimes, you have to deal with legacy applications that might require an 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 trick is to set up a startup probe with the same command, HTTP or TCP check, with a `failureThreshold * periodSeconds` long enough to cover the worse case startup time.

So, the previous example would become:

```yaml
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

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:** Liveness probes *do not* wait for readiness probes to succeed. If you want to wait before executing a liveness 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.

```yaml
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 liveness and readiness checks:

- `initialDelaySeconds` : Number of seconds after the container has started before liveness or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.
- `periodSeconds` : How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
- `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` : When a probe fails, Kubernetes will try `failureThreshold` times before giving up. Giving up in case of liveness probe means restarting the container. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

> **Note:**
> Before Kubernetes 1.20, the field `timeoutSeconds` was not respected for exec probes: probes continued running indefinitely, even past their configured deadline, until a result was returned.
>
> This defect was corrected in Kubernetes v1.20. You may have been relying on the previous behavior, even without realizing it, as the default timeout is 1 second. As a cluster administrator, you can disable the feature gate `ExecProbeTimeout` (set it to `false`) on each kubelet to restore the behavior from older versions, then remove that override once all the exec probes in the cluster have a `timeoutSeconds` value set.
> If you have pods that are impacted from the default 1 second timeout, you should update their probe timeout so that you're ready for the eventual removal of that feature gate.
>
> With the fix of the defect, for exec probes, on Kubernetes `1.20+` with the `dockershim` container runtime, the process inside the container may keep running even after probe returned failure because of the timeout.

> **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 path and port 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` , and `Accept` . The default values for these headers are `kube-probe/1.21` (where `1.21` is the version of the kubelet), and `*/*` respectively.

You can override the default headers by defining `.httpHeaders` for the probe; for example

```yaml
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.

```yaml
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: ""

startupProbe:
  httpGet:
    httpHeaders:
      - name: User-Agent
        value: ""
```

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 v1.21 [alpha]`

Prior to release 1.21, the pod-level `terminationGracePeriodSeconds` was used for terminating a container that failed its liveness or startup probe. This coupling was unintended and may have resulted in failed containers taking an unusually long time to restart when a pod-level `terminationGracePeriodSeconds` was set.

In 1.21, when the feature flag `ProbeTerminationGracePeriod` is enabled, users can specify a probe-level `terminationGracePeriodSeconds` as part of the probe specification. When the feature flag is enabled, and both a pod- and probe-level `terminationGracePeriodSeconds` are set, the kubelet will use the probe-level value.

For example,

```yaml
spec:
  terminationGracePeriodSeconds: 3600  # pod-level
  containers:
  - name: test
    image: ...

    ports:
    - name: liveness-port
      containerPort: 8080
      hostPort: 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
- Container
- Probe

14 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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=
worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
```

2. Chose 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
worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeName: foo-node # schedule pod to specific node
  containers:
  - name: nginx
```

```
    image: nginx
    imagePullPolicy: IfNotPresent
```

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.

15 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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=
   worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
   worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
   ```

2. Chose 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
   worker1   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
   worker2   Ready     <none>   1d      v1.13.0        ...,kubernetes.io/hostname=
   ```

   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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
```

```yaml
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

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.

16 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: workdir
      mountPath: /usr/share/nginx/html
  # These containers are run during pod initialization
  initContainers:
  - name: install
    image: busybox
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://info.cern.ch
    volumeMounts:
    - name: workdir
      mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
  - name: workdir
    emptyDir: {}
```

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 w
  ...
```

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

17 - 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/s
      preStop:
        exec:
          command: ["/bin/sh","-c","nginx -s quit; while killall -0 nginx; do sleep
```

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 is *terminated*. This means that the preStop hook is not invoked when the Pod is *completed*. This limitation is tracked in issue #55087.

What's next

- Learn more about Container lifecycle hooks.
- Learn more about the lifecycle of a Pod.

Reference

- Lifecycle
- Container
- See `terminationGracePeriodSeconds` in PodSpec

18 - Configure a Pod to Use a ConfigMap

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

Create a ConfigMap

You can use either `kubectl create configmap` or a ConfigMap generator in `kustomization.yaml` to create a ConfigMap. Note that `kubectl` starts to support `kustomization.yaml` since 1.14.

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 ConfigMaps from directories

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 basename 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 (e.g. subdirectories, symlinks, devices, pipes, etc).

For example:

```
# Create the local directory
mkdir -p configure-pod-container/configmap/

# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game.properties -O configure-pod-con
wget https://kubernetes.io/examples/configmap/ui.properties -O configure-pod-conta
```

```
# 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: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
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
```

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
```

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
```

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
```

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>`.

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 C

# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game-env-file.properties -O configure-

# 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 the following ConfigMap:

```
kubectl get configmap game-config-env-file -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
```

```yaml
metadata:
  creationTimestamp: 2017-12-27T18:36:28Z
  name: game-config-env-file
  namespace: default
  resourceVersion: "809965"
  uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
```

Caution: When passing `--from-env-file` multiple times to create a ConfigMap from multiple data sources, only the last env-file is used.

The behavior of passing `--from-env-file` multiple times is demonstrated by:

```shell
# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/ui-env-file.properties -O configure-pc

# Create the configmap
kubectl create configmap config-multi-env-files \
        --from-env-file=configure-pod-container/configmap/game-env-file.properties \
        --from-env-file=configure-pod-container/configmap/ui-env-file.properties
```

would produce the following ConfigMap:

```shell
kubectl get configmap config-multi-env-files -o yaml
```

where the output is similar to this:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:38:34Z
  name: config-multi-env-files
  namespace: default
  resourceVersion: "810136"
  uid: 252c4572-eb35-11e7-887b-42010a8002b8
data:
  color: purple
  how: fairlyNice
  textmode: "true"
```

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:

```shell
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-
```

would produce the following ConfigMap:

```
kubectl get configmaps game-config-3 -o yaml
```

where the output is similar to this:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
```

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-liter
```

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:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  uid: dadce046-d673-11e5-8cd0-68f728db1985
data:
  special.how: very
  special.type: charm
```

Create a ConfigMap from generator

`kubectl` supports `kustomization.yaml` since 1.14. You can also create a ConfigMap from generators and then apply it to create the object on the Apiserver. The generators should be specified in a `kustomization.yaml` inside 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
  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


kubectl describe configmaps/game-config-4-m9dm2f92bt
Name:         game-config-4-m9dm2f92bt
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","data":{"game.properties":"enemies=aliens\nlives=

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>
```

Note 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
  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

To generate a ConfigMap from literals `special.type=charm` and `special.how=very`, you can specify the ConfigMap generator in `kustomization.yaml` as

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: special-config-2
  literals:
  - special.how=very
  - special.type=charm
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/special-config-2-c92b5mmcf2 created
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
```

```yaml
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        # Define the environment variable
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              # The ConfigMap containing the value you want to assign to SPECIAL_LE
              name: special-config
              # Specify the key associated with the value
              key: special.how
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-single-configmap-env-varia
```

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.

configmap/configmaps.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

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

```yaml
apiVersion: v1
```

```yaml
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: env-config
              key: log_level
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-multiple-configmap-env-var
```

Now, the Pod's output includes environment variables `SPECIAL_LEVEL_KEY=very` and `LOG_LEVEL=INFO`.

Configure all key-value pairs in a ConfigMap as container environment variables

> **Note:** This functionality is available in Kubernetes v1.6 and later.

- Create a ConfigMap containing multiple key-value pairs.

configmap/configmap-multikeys.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
      - configMapRef:
          name: special-config
  restartPolicy: Never
```

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`.

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 specification

pods/pod-configmap-env-var-valueFrom.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/echo", "$(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_LEVEL
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_TYPE
  restartPolicy: Never
```

created by running

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-env-var-valueFro
```

produces the following output in the `test-container` container:

```
very charm
```

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, shown below.

[configmap/configmap-multikeys.yaml]()

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  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]()

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
      - name: config-volume
```

```yaml
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want
        # to add to the container
        name: special-config
  restartPolicy: Never
```

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
```

Caution: If there are some files in the `/etc/config/` directory, they will be deleted.

Note: Text data is exposed as files using the UTF-8 character encoding. To use some other character encoding, use binaryData.

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh","-c","cat /etc/config/keys" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: SPECIAL_LEVEL
          path: keys
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume-specific-
```

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.

Project keys to specific paths and file permissions

You can project keys to specific paths and specific permissions on a per-file basis. The Secrets user guide explains the syntax.

Mounted ConfigMaps are updated automatically

When a ConfigMap already being consumed in a volume is updated, projected keys are eventually updated as well. Kubelet is checking whether the mounted ConfigMap is fresh on every periodic sync. However, it is using 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` .

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  # example of a simple property defined using --from-literal
  example.property.1: hello
  example.property.2: world
  # example of a complex property defined using --from-file
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

Restrictions

- You must create a ConfigMap before referencing it in a Pod specification (unless you mark the ConfigMap as "optional"). If you reference a ConfigMap that doesn't exist, the Pod won't start. Likewise, references to keys that don't exist in the ConfigMap will prevent the pod from starting.
- 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
  0s       0s        1     dapi-test-pod Pod              Warning   InvalidEnviro
  ```

- ConfigMaps reside in a specific Namespace. A ConfigMap can only be referenced by pods residing in the same namespace.
- You can't use ConfigMaps for static pods, because the Kubelet does not support this.

What's next

- Follow a real world example of Configuring Redis using a ConfigMap.

19 - Share Process Namespace between Containers in a Pod

FEATURE STATE: `Kubernetes v1.17 [stable]`

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 that 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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.10. To check the version, enter `kubectl version`.

Configure a Pod

Process Namespace Sharing is enabled using the `shareProcessNamespace` field of `v1.PodSpec`. For example:

pods/share-process-namespace.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: busybox
    securityContext:
      capabilities:
        add:
        - SYS_PTRACE
    stdin: true
    tty: true
```

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 attach -it nginx -c shell
```

If you don't see a command prompt, try pressing enter.

```
/ # ps ax
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.

```
/ # kill -HUP 8
/ # ps ax
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 another container image using the `/proc/$pid/root` link.

```
/ # head /proc/8/root/etc/nginx/nginx.conf

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 container images may expect to be isolated from other containers, though, so it's important to understand these differences:

1. **The container process no longer has PID 1.** Some container images 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.

20 - 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 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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

This page assumes you're using Docker 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/kubelet.d` and place a web server Pod definition there, for example `/etc/kubelet.d/static-web.yaml` :

   ```
   # Run this command on the node where kubelet is running
   mkdir /etc/kubelet.d/
   cat <<EOF >/etc/kubelet.d/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 your kubelet on the node to use this directory by running it with `--pod-manifest-path=/etc/kubelet.d/` argument. On Fedora edit `/etc/kubernetes/kubelet` to include this line:

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.local --pod-manif
```

   or add the `staticPodPath: <the directory>` field in the kubelet configuration file.

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.

```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
```

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-
```

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
docker ps
```

The output might be something like:

```
CONTAINER ID IMAGE         COMMAND  CREATED        STATUS         PORTS     NAMES
f6d05272b57e nginx:latest  "nginx"  8 minutes ago  Up 8 minutes            k8s_web.
```

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. See PodSecurityPolicy.

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          12s
```

Back on your node where the kubelet is running, you can try to stop the Docker 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
docker stop f6d05272b57e # replace with the ID of your container
sleep 20
docker ps
```

```
CONTAINER ID        IMAGE         COMMAND                CREATED       ...
5b920cbaf8b1        nginx:latest  "nginx -g 'daemon of   2 seconds ago ...
```

Dynamic addition and removal of static pods

The running kubelet periodically scans the configured directory (`/etc/kubelet.d` 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 kubelet is running
#
mv /etc/kubelet.d/static-web.yaml /tmp
sleep 20
docker ps
# You see that no nginx container is running
mv /tmp/static-web.yaml  /etc/kubelet.d/
sleep 20
docker ps
```

```
CONTAINER ID        IMAGE         COMMAND                CREATED           ...
e7a62e3427f1        nginx:latest  "nginx -g 'daemon of   27 seconds ago
```

21 - 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 http://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. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- 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 | CentOS package | Fedora package | 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/v1.22.0/kompose-

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/v1.22.0/kompose-

# Windows
curl -L https://github.com/kubernetes/kompose/releases/download/v1.22.0/kompose-

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

Alternatively, you can download the tarball.

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.

```
version: "2"

services:
```

```
  redis-master:
    image: k8s.gcr.io/redis:e2e
    ports:
      - "6379"

  redis-slave:
    image: gcr.io/google_samples/gb-redisslave:v3
    ports:
      - "6379"
    environment:
      - GET_HOSTS_FROM=dns

  frontend:
    image: gcr.io/google-samples/gb-frontend:v4
    ports:
      - "80:80"
    environment:
      - GET_HOSTS_FROM=dns
    labels:
      kompose.service.type: LoadBalancer
```

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 "frontend-service.yaml" created
   INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
   INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
   INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
   INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
   INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
   INFO Kubernetes file "redis-slave-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

```
 kubectl apply -f frontend-service.yaml,redis-master-service.yaml,redis-slave-s
```

The output is similar to:

```
redis-master-deployment.yaml,redis-slave-deployment.yaml
service/frontend created
service/redis-master created
service/redis-slave created
deployment.apps/frontend created
deployment.apps/redis-master created
deployment.apps/redis-slave created
```

Your deployments are running in Kubernetes.

3. Access your application.

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

   ```
   minikube service frontend
   ```

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

   ```
   kubectl describe svc frontend
   ```

   ```
   Name:                   frontend
   Namespace:              default
   Labels:                 service=frontend
   Selector:               service=frontend
   Type:                   LoadBalancer
   IP:                     10.0.0.183
   LoadBalancer Ingress:   192.0.2.89
   Port:                   80      80/TCP
   NodePort:               80      31144/TCP
   Endpoints:              172.17.0.4:80
   Session Affinity:       None
   No events.
   ```

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

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

User Guide

- CLI
  - `kompose convert`
  - `kompose up`
  - `kompose down`
- Documentation
  - Build and Push Docker Images
  - 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.y
db-svc.yaml         docker-voting.yml          redis-svc.yaml     result-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-ser
frontend-deployment.yaml  mongodb-claim0-persistentvolumeclaim.yaml  redis-master-se
frontend-service.yaml     mongodb-deployment.yaml                    redis-slave-dep
redis-master-deployment.yaml
```

When multiple docker-compose files are provided the configuration is merged. Any configuration that is common will be over ridden 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 .

`kompose up`

Kompose supports a straightforward way to deploy your "composed" application to Kubernetes or OpenShift via `kompose up`.

Kubernetes `kompose up` example

```
kompose --file ./examples/docker-guestbook.yml up
```

```
We are going to create Kubernetes deployments and services for your Dockerized appli
If you need different kind of resources, use the 'kompose convert' and 'kubectl appl

INFO Successfully created service: redis-master
INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created deployment: frontend

Your application has been deployed to Kubernetes. You can run 'kubectl get deploymen
```

```
kubectl get deployment,svc,pods
```

```
NAME                                         DESIRED       CURRENT       UP-TO-
deployment.extensions/frontend               1             1             1
deployment.extensions/redis-master           1             1             1
deployment.extensions/redis-slave            1             1             1

NAME                     TYPE              CLUSTER-IP      EXTERNAL-IP   PORT(S)
service/frontend         ClusterIP         10.0.174.12     <none>        80/TCP
service/kubernetes       ClusterIP         10.0.0.1        <none>        443/TCP
service/redis-master     ClusterIP         10.0.202.43     <none>        6379/TCP
service/redis-slave      ClusterIP         10.0.1.85       <none>        6379/TCP

NAME                               READY         STATUS        RESTARTS     AGE
pod/frontend-2768218532-cs5t5      1/1           Running       0            4m
pod/redis-master-1432129712-63jn8  1/1           Running       0            4m
pod/redis-slave-2504961300-nve7b   1/1           Running       0            4m
```

Note:

- You must have a running Kubernetes cluster with a pre-configured kubectl context.
- Only deployments and services are generated and deployed to Kubernetes. If you need different kind of resources, use the `kompose convert` and `kubectl apply -f` commands instead.

OpenShift `kompose up` example

```
kompose --file ./examples/docker-guestbook.yml --provider openshift up
```

```
We are going to create OpenShift DeploymentConfigs and Services for your Dockerized
If you need different kind of resources, use the 'kompose convert' and 'oc create -f

INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created service: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created ImageStream: redis-slave
INFO Successfully created deployment: frontend
INFO Successfully created ImageStream: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created ImageStream: redis-master

Your application has been deployed to OpenShift. You can run 'oc get dc,svc,is' for
```

```
oc get dc,svc,is
```

```
NAME              REVISION                              DESIRED       CURRENT     TR
dc/frontend       0                                     1             0           co
dc/redis-master   0                                     1             0           co
dc/redis-slave    0                                     1             0           co
NAME              CLUSTER-IP                            EXTERNAL-IP   PORT(S)     AG
svc/frontend      172.30.46.64                          <none>        80/TCP      8s
svc/redis-master  172.30.144.56                         <none>        6379/TCP    8s
svc/redis-slave   172.30.75.245                         <none>        6379/TCP    8s
NAME              DOCKER REPO                           TAGS          UPDATED
is/frontend       172.30.12.200:5000/fff/frontend
is/redis-master   172.30.12.200:5000/fff/redis-master
is/redis-slave    172.30.12.200:5000/fff/redis-slave    v1
```

Note: You must have a running OpenShift cluster with a pre-configured `oc` context (`oc login`).

`kompose down`

Once you have deployed "composed" application to Kubernetes, `kompose down` will help you to take the application out by deleting its deployments and services. If you need to remove other resources, use the 'kubectl' command.

```
kompose --file docker-guestbook.yml down
INFO Successfully deleted service: redis-master
INFO Successfully deleted deployment: redis-master
INFO Successfully deleted service: redis-slave
INFO Successfully deleted deployment: redis-slave
INFO Successfully deleted service: frontend
INFO Successfully deleted deployment: frontend
```

Note: You must have a running Kubernetes cluster with a pre-configured `kubectl` context.

Build and Push Docker Images

Kompose supports both building and pushing Docker images. When using the `build` key within your Docker Compose file, your image will:

- Automatically be built with Docker using the `image` key specified within your file
- Be pushed to the correct Docker repository using local credentials (located at `.docker/config`)

Using an [example Docker Compose file](https://raw.githubusercontent.com/kubernetes/kompose/master/examples/buildconfig/docker-compose.yml):

```yaml
version: "2"

services:
    foo:
        build: "./build"
        image: docker.io/foo/bar
```

Using `kompose up` with a `build` key:

```
kompose up
INFO Build key detected. Attempting to build and push image 'docker.io/foo/bar'
INFO Building image 'docker.io/foo/bar' from directory 'build'
INFO Image 'docker.io/foo/bar' from directory 'build' built successfully
INFO Pushing image 'foo/bar:latest' to registry 'docker.io'
INFO Attempting authentication credentials 'https://index.docker.io/v1/
INFO Successfully pushed image 'foo/bar:latest' to registry 'docker.io'
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeCla

INFO Deploying application in "default" namespace
INFO Successfully created Service: foo
INFO Successfully created Deployment: foo

Your application has been deployed to Kubernetes. You can run 'kubectl get deploymen
```

In order to disable the functionality, or choose to use BuildConfig generation (with OpenShift) `--build (local|build-config|none)` can be passed.

```
# Disable building/pushing Docker images
kompose up --build none

# Generate Build Config artifacts for OpenShift
kompose up --provider openshift --build build-config
```

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:

```yaml
version: "2"
services:
  nginx:
    image: nginx
    dockerfile: foobar
    build: ./foobar
    cap_add:
      - ALL
    container_name: foobar
    labels:
      kompose.service.type: nodeport
```

- `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:

```yaml
version: "2"
```

```yaml
services:
  web:
    image: tuna/docker-counter23
    ports:
     - "5000:5000"
    links:
     - redis
    labels:
      kompose.service.expose: "counter.example.com"
  redis:
    image: redis:3.0
    ports:
     - "6379"
```

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`.

```yaml
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 (eg. `web_service`), then it will be replaced by `-` and the service name will be renamed accordingly (eg. `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.